3-2020

Next-Generation Air Force Weather Metrics via Bayes Cost Analysis

Brandon M. Bailey

NEXT-GENERATION AIR FORCE WEATHER METRICS VIA BAYES COST ANALYSIS

THESIS

Brandon M. Bailey, Captain, USAF

AFIT-ENP-MS-20-M-077

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENP-MS-20-M-077

NEXT-GENERATION AIR FORCE WEATHER METRICS VIA BAYES COST ANALYSIS

THESIS

Presented to the Faculty

Department of Engineering Physics

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Atmospheric Science

Brandon M. Bailey, BS
Captain, USAF

March 2020

AFIT-ENP-MS-20-M-077

NEXT-GENERATION AIR FORCE WEATHER METRICS VIA BAYES COST ANALYSIS

Brandon M. Bailey, BS

Captain, USAF

Committee Membership:

Lt Col Andrew J. Geyer, PhD
Chair

Lt Col Robert C. Tournay, PhD
Member

Abstract

This research proposes a new methodology for U.S. Air Force weather forecast metrics. Military weather forecasters are essentially statistical classifiers. They categorize future conditions into an operationally relevant category based on current data, much like an Artificial Neural Net or Logistic Regression model. There is extensive literature on statistically-based metrics for these types of classifiers. Additionally, in the U.S. Air Force, forecast errors (errors in classification) have quantifiable operational costs and benefits associated with incorrect or correct classification decisions. There is a methodology in the literature, Bayes Cost, which provides a structure for creating statistically rigorous metrics for classification decisions that have such costs and benefits. Applying these types of metrics to Air Force weather yields more informative metrics that account for random chance while remaining simple to calculate.

This research conducts an analysis using notional values from an unnamed subject matter expert. Bayes Cost-based verification on Terminal Aerodrome Forecasts and Watches/Warnings/Advisories compared to surface observations from a selection of military installations in the continental United States during the period 01 May 2019 to 30 June 2019. The case study illustrates the added utility of the new metric paradigm.

Acknowledgments

I would like to express my sincere appreciation to my faculty advisor, Lt Col Geyer, for his guidance and support throughout the course of this thesis effort. The insight imparted unto me is greatly appreciated. I would, also, like to thank my co-advisor, Lt Col Tournay, for both the support and latitude provided to me in this endeavor.

Brandon M. Bailey

Table of Contents

Page

List of Figures

Page

List of Tables

Page

List of Acronyms and Abbreviations

NEXT-GENERATION AIR FORCE WEATHER METRICS VIA BAYES COST ANALYSIS

I. Introduction

General Issue

It is the Air Force's policy to provide "past, current, and predicted terrestrial weather, climate, hydrometeorological, and hydrological services for all elements of the Air Force, designated parts of the Intelligence Community (IC), and the U.S. Army…" (AFPD 15-1, 2019). Air Force Weather (AFW) personnel and equipment exist to implement this policy wherever and whenever the mission dictates. AFW's performance with respect to these mission requirements is evaluated via a series of verification metrics outlined in Air Force Instruction (AFI) 15-114. This document provides detailed requirements for AFW metrics reporting and evaluation, thereby ensuring a uniform evaluation system across all of AFW, regardless of each unit's individual operational mission requirements (AFI 15-114, 2017).

AFI 15-114 outlines procedures for metrics governing these areas:

- Weather Watch, Warning and Advisory (WWA) Verification (WARNVER)
- Terminal Aerodrome Forecast (TAF) Verification (TAFVER)
- Operational Verification (OPVER)
- Graphical Weather Depiction Verification (GRAPHVER)
- Weather Model Prediction Verification (MODVER)

The approach proposed in this research is applicable to all five technical health metric areas. However, the focus of this research will rest primarily on its application to

WARNVER, TAFVER and OPVER due to the limited availability of data for use as exemplar results in Chapter 4 based on the methodology outlined in Chapter 3.

Problem Statement

The current technical health metrics described in AFI 15-114 are problematic for two main reasons. The first problem is that the metrics related to operations do not convey the value of AFW services in measures valued by decision makers outside of AFW. Properly communicating the utility of a service necessitates metrics that relate directly to cost benefit analysis (Bossart & Bean, 2011) (Pauker & Kassirer, 1975). AFI 15-114 features a wide variety of Measures of Performance (MOPs) such as Positive Lead Time Performance (PLTF) for WWAs and accuracy requirements for TAFs.

While a large PLTF for WWAs seems like an intuitive measure, it may provide little insight into the value of those warnings with respect to the operational missions. Not all WWAs are of equal value. An AFW forecaster may be providing large PLTFs for frequent WWAs with small mission impact while providing small and potentially useless PLTFs for rare WWAs critical to mission success. That forecaster's MOPs will look very good at first glance. One would have to drill down to determine the meaning underneath the MOP value. Even with thorough examination it would not be obvious what precisely this forecaster has done to aid the mission in terms of things that decision makers care about.

One could just point to true accuracy as a MOP. Current TAFVER accuracy MOPs give a forecaster credit for being accurate if forecast wind speeds are within 9 knots of what occurred (AFI 15-114, 2017). However, if a parachute drop were

occurring on an airfield and the maximum winds for operations were, say, 13 knots, is a forecast of 20 knots when 11 knots occur or 10 knots when 19 knot wind speeds occur actually an accurate forecast in the eyes of the units performing the mission? OPVER intends to capture this sort of occurrence, not TAFVER. If the TAF MOPs deem a forecast to be accurate when it was inaccurate with respect to the needs of the supported mission, then what is the true utility of the TAF product in this scenario? Was the TAF product worth the effort to create it?

The current AFI 15-114 MOPs do not provide an obvious way to note these types of scenarios where the raw MOP values may seem positive when the true scenario is of little value to those executing the mission. Even when the MOPs are representative, they are in units of percent accuracy or percent precision, etc. These values do not provide an obvious mapping to MOPs in use by decision makers across the Air Force and Army. For example, a commander of an Air Force flying unit in garrison is primarily responsible for executing an annual flying hour program. This program assesses unit combat capability via the execution of a number and variety of flying missions measured in aircraft hours (AFPD 11-1, 2004). Current AFW MOPs do not provide an obvious mapping to a quantifiable contribution to the flying hour program.

The second problem is that the current technical health metrics are subject to random chance meaning that scores can depend more on what events a forecaster faces than the actual skill of the forecaster. For example, a forecaster could correctly predict one significant weather event while failing to predict 17 weather events, resulting in a 100% score for the "Criteria Event Accuracy" metric. Alternatively, a forecaster with one weather event false alarm prediction and only favorable weather otherwise would

receive a 0% “Criteria Event Accuracy” score (AFI 15-114, 2017). While this metric is intended to be used in conjunction with other metrics outlined in AFI 15-114, one can see how interpreting these metrics as a whole can become complicated and highly subjective based on the events of the time period being evaluated. Chance can play a large part the evaluation process for forecaster. This limits the effectiveness of having a standardized set of metrics across all AFW. A preferred set of metrics would account for the variability of events at different forecast locations and times, thereby separating out the contribution of the forecaster from all other factors beyond his or her control.

Preview

The next chapter provides a brief review of the literature pertaining to Air Force weather forecasting products and processes that illustrates that military weather forecasting is effectively a classification process. Next there is a summary of AFW MOPs and their relationship to other classifier metrics in academic literature. The chapter concludes with an introduction to the Bayes Cost metric for classifiers that face decisions with uneven costs or benefits. Chapter 3 then provides a description of how to apply the Bayes Cost metric to WARNVER, TAFVER and OPVER. Chapter 4 illustrates the utility of the Bayes Cost AFW metric paradigm via a case study conducted using data from the 15th Operational Weather Squadron during the period 01 May 2019 to 30 July 2019. Chapter 5 concludes this work with proposed future research on this topic to include additional applications useful to Air Force decision makers.

II. Literature Review

Chapter Overview

This chapter begins with a discussion of the five general types of Air Force weather forecasting products. Military weather forecasting is effectively a classification process. This chapter discusses AFW MOPs and standard classification metrics in academic literature. Military decision making requires choosing between options with varying costs/benefits. This chapter introduces the proposed Bayes Cost metric as a candidate for use as an AFW MOP.

AFW Products

AFW verifies the quality of weather support to the larger Air Force mission in five areas: MODVER, GRAPHVER, WARVER, TAFVER and OPVER. These five areas cover the spectrum of AFW support to most missions. WARNVER, TAFVER, and OPVER are the primary focus of this research. This chapter touches on MODVER and GRAPHVER principles because the BC framework will also work in those areas with some minor modifications.

MODVER is the verification of the primary AFW Model performance against the observation of the same parameter (AFI 15-114, 2017). The model of choice for AFW is the Global Air and Land Exploitation Model (GALWEM). Surface and upper air observations are not spatially dense. This spatial distribution along with challenges in data assimilation of available remote observations from satellites, and aircraft limit the effectiveness of any model verification program (16th Weather Squadron, 2018). Some model parameters such as temperature and windspeed are relatively easy to measure

consistently. Other parameters such as cloud measurements, visibility and precipitation type/amount are subject to more uncertainty. Satellite measurements are subject to spatial challenges limiting the effective measurement of parameters to scales much larger than a single forecast location. Such challenges exist regardless of the community (16th Weather Squadron, 2018). GALWEM is a starting point for creation of the other AFW products.

From this data and other sources, AFW produces graphics on a global scale for various phenomenon including cloud heights, visibility, and the tendency of ice to accumulate on planes wings in flight (icing). GRAPHVER is the measurement of the accuracy of these graphics from reported conditions experienced in flight as well as "other subjective verification data points." An example of such a datapoint is to verify icing in certain cloud types as a given height (AFI 15-114, 2017). These verification techniques are largely subjective.

One of the focus of this research is WARNVER. This MOP which considers the accuracy in space and time of a WWA. These weather alerts are inherently impactful to the mission and pose a "hazard to life or property" (AFI 15-114, 2017). The verification of these alerts is based on direct threshold observation within the verification radius of between 5 or 10 miles depending on the alert.

A longer range, more regularly produced product is a TAF. TAFs are 30-hour forecasts issued for military locations with airfields. These products are typically issued multiple times a day on days when the airfield is open. TAFVER is the verification of the accuracy of TAFs (AFI 15-114, 2017). AFW verifies forecast accuracy through

observation of weather sensing equipment at or near the forecast location. AFW verifies the initial TAFs and not amendments based upon categorical thresholds.

The final type of verification is closely related to TAFVER. OPVER examines the impact of weather forecasts on planning and execution of a supported mission. AFW conducts OPVER on forecasts outside of the standard TAF format to include Mission Execution Forecasts (MEFs) (AFI 15-114, 2017). These forecasts have various formats as determined by the mission needs or commander requests (AFI 15-114, 2017). The parameters included in these forecasts are subject to mission requirements.

National Weather Service (NWS)

NWS TAFs are largely the same format as AFW. The NWS has a national focus and does not focus on any assets. The NWS mission statement is to "Provide weather, water, and climate data, forecasts and warnings for the protection of life and property and enhancement of the national economy" (NWS, 2020). NWS products assume no particular type of aircraft or purpose for its forecast products. The NWS can focus on accurately categorizing every weather parameter. AFW faces a different task. AFW forecasters exists to support United States Air Force (USAF) missions. These missions rely on categories governing operational risks. The utility of AFW forecasts is in how forecasts correctly identify mission specific categorical thresholds.

Air Force Weather Forecasting Categories

AFW supports specific missions with explicit weather sensitivities. Correctly categorizing threshold-based conditions that directly impact weather conditions is valuable to military decision makers. Current AFW metrics largely focus on technical

accuracy in a similar fashion to the NWS. This technical accuracy does not directly relate to mission impact of forecasts. This disconnect is evident in the AFI 15-114 sections that outline procedures for WARNVER, TAFVER, and OPVER.

WARNVER categories focus on event timing, represented by percentages of Desired Lead Time (DLT), and event criteria, represented by WWA thresholds. There is no fixed maximum duration of WWA verification timeline of the WWA. Table 1 is an excerpt of some of the WARNVER criteria.

Table 1: WARNVER MOPs (AFI 15-114, 2017)

MOP	**Individual Event Calculation** (for all events that occur within the specified verification distance)	**Combined Event Calculation** (for all events that occur within the specified verification distance)
Met DLT Percentage	The total number of forecast events that occurred and met the full DLT divided by the total number of events that occurred.	The total number of forecast WWA events that occurred and met the full desired lead-time divided by the total number of WWA events that occurred.
Positive Lead Time Percentage	The total number of forecast events with positive lead time divided by the total number of forecast events that occurred	The total number of WWA events with positive lead time divided by the total number of events that occurred
False Alarm Rate (FAR)	The number of issued WWAs minus the number of required WWAs divided by the number of issued WWAs.	The total number of issued WWAs minus the number of required WWAs divided by the total number of issued WWAs.

Both conditions lend themselves well to utility assessments. However, the lack of limits on timing creates a virtually infinite number of cost scenarios. This makes it difficult to directly quantify the impact of WWAs as they can be issued for hours without verifying. To mitigate this, current MOPs describe nine potential categories for timing and magnitude errors. These categories, do not have costs associated with them.

TAFVER MOPs seek to tie forecasts to categories of TAFs. Table 2 features the wind speed TAFVER MOP as an example. A forecast for winds greater than 6 knots is accurate if that forecast is within ± 9 knots of wind. Inclement weather conditions are by nature intermittent or short-lived throughout most of the US. It is possible to mischaracterize all weather conditions of operational significance and receive a high percentage score. This is true because of the prevalence of correctly categorized low impact weather events. The operational utility of missing a forecast by 9 knots is dependent upon the forecast value and the mission sensitivity to windspeed. The current metric does not account for this variability. This metric does not connect to practical measurements of flying hours, money, or resources saved.

Table 2: TAFVER MOPs (AFI 15-114, 2017)

Criteria	**Evaluate**	**Requirement**	**Hourly Score and Overall Percentage Correct**
Wind Speed	+ or – 9 knots	Verify all forecast groups where wind speeds are GTE than 6 knots. If the forecast is within 9 knots it is a correct forecast. For 10 knots or greater of error the forecast is incorrect.	The hourly score is one point for a correct forecast and zero points for an incorrect forecast. The overall TAF Wind Speed percentage correct is the total number of points for correct forecasts (pcf) divided by the total number of available points (ap) multiplied by 100 ((pcf/ap)*100)

OPVER MOPs are more complex because missions are variable. For example, flying units within the USAF categorize weather conditions into Pilot Weather Categories (PWCs) when conducting training missions. PWCs define under what conditions pilots with certain skill and experience can fly. Many of the USAF Major Commands

(MAJCOMs) have customized PWC categories (AFI11-202V3_ACCSUP_I, 2012). Table 3 shows Air Combat Command (ACC), PWC takeoff and landing requirements.

Table 3: PWC Minimums Fighter Aircraft (AFI11-202V3_ACCSUP_I, 2012)

PWC	Flying Hours	Takeoff †	Approach †
1	50 primary in acft and 1000 total	Base minimums	Base minimums
2	100 primary in acft and flight lead or 750 total	Ceiling 300 ft and Visibility 1 mile (RVR 5000 ft)	Ceiling 300 ft and Visibility 1 mile (RVR 5000 ft)
3	50 primary in acft and BMC/CMR or 500 total	Ceiling 500 ft and Visibility 1½ miles	Ceiling 500 ft and Visibility 1½ miles
4	Instrument eval & BMC/CMR or 500 total	Ceiling 700 ft and Visibility 2 miles	Ceiling 700 ft and Visibility 2 miles
5	Initial Qual Training and Requal students	Ceiling 1500 ft and Visibility 3 miles	Ceiling 1500 ft and Visibility 3 miles

In this case, there are five categories with the lowest-number PWCs corresponding to the most experienced pilots. This means that different forecast errors have different mission impacts. Forecasting PWC 4 conditions when PWC 1 conditions can waste time and resources on canceled missions and relocations. Forecasting PWC 1 conditions when PWC 4 conditions occur can also risk personnel and equipment.

AFW uses an operationally based “Stoplight” categorization to support many operations as seen in Table 4. This system organizes mission sensitivity parameter into favorable, marginal, or unfavorable categories. This chart helps decision makers and forecasters communicate in the same language thus supporting USAF Risk Management goals (AFI 90-802, 2019).

Table 4: Weather Impacts on Operations (Stoplight Chart) (JMOC, 2011)

Operation	Favorable (No Degradation)	Marginal (Some Degradation)	Unfavorable (Significant Degradation)
Airborne Operations	Wind < 15 kts	Wind 15-19 kts	Wind > 19 kts
	Ceiling > 3000 ft		Ceiling < 3000 ft
	No Precipitation	Light Precipitation	Heavy Precipitation
	Density Altitude < 7000 ft	Density Altitude 7000-9000 ft	Density Altitude >9000 ft
RPA ISR	Wind < 15 kts	Wind 15-19 kts	Wind > 19 kts
	Ceiling > 3000 ft	Ceiling 1000-3000 ft	Ceiling < 1000 ft
	Crosswind < 15 kts	Crosswind 15-30 kts	Crosswind > 30 kts
Close Air Support	Ceiling > 3000 ft	Ceiling 1000-3000 ft	Ceiling < 1000 ft
	Visibility > 3200 m	Visibility 1600-3200 m	Visibility < 1600 m

This chart is notional based on the style of weather effects chart taught at the Army Support Weather Support Course. The Joint METOC (JMOC) Handbook serves as a quick reference for weather personnel operating in a Joint environment and features a similar chart (JMOC, 2011). Operational forecasts based on utility rather than technical accuracy are useful internationally (Mahringer, 2008). In example of this in an international combined operations setting are the standard North Atlantic Treaty Organization (NATO) color coded categories for weather impacts at an airfield listed in Table 5.

Table 5: NATO Airfield Weather Color Code (AFPAM:11-238, 2011)

Color Code	Color	† Ceiling at or above:	Visibility at or above:
BLU	blue	2500 feet	8000 meters
WHT	white	1500 feet	5000 meters
GRN	green	700 feet	3700 meters
YLO	yellow	300 feet	1600 meters
AMB	amber	200 feet	0800 meters
RED	red	< 200 feet	< 0800 meters
BLACK	black	Airfield not useable for other reasons	

Mission Execution Forecasts (MEFs) and other operational forecasts function as more mission specific versions of TAFs. AFW metrics for these custom focus on event capturing akin to the MOPs for WWAs. These event categories connect to a specific action by the supported commander. OPVER seeks to report technical accuracy rather than operational impact in terms of flying hours, dollars, or other utility consideration.

Table 6: WP/MEF (OPVER) MOPs (AFI 15-114, 2017)

Performance Metric	Formula	Description
Accuracy (Optional)	(A+D)/(A+B+C+D) x 100%	Accuracy indicates the percentage of accurate WPs/MEFs compared to all WPs/MEFs issued. CAUTION: Do not use this metric alone to judge the overall performance of the WP program due to the naturally occurring high percentage of "No Criteria Event WPs, No Criteria Event Observed" outcomes at many operating locations. Use "Criteria Event Accuracy" and "No Criteria Event Accuracy" metrics to shed light on problem areas
Criteria Event Accuracy (Mandatory)	(A/(A+C)) x 100%	Criteria Event Accuracy indicates percentage of Criteria Event WP/MEFs that verified correctly. This tells leadership how often a forecast for mission impacting weather verified.
No Criteria Event Accuracy (Optional)	(D/(B+D)) x 100%	No Criteria Event Accuracy indicates percentage of no operational criteria WP/MEF forecasts that were correctly made. This tells leadership how often a forecast for non-mission impacting weather verified.
Mitigation Rate (Optional)	((MEFC +WPC) / (criteria events forecast for MEFs/WPs))*100	Take the monthly total of MEFCs and WPCs (MEFs and WPs that resulted in mission changes) and divide by the monthly total of WP and MEF criteria event forecasts. Multiply by 100 to determine the mitigation rate, or percentage of time operators took action on criteria event forecasts for WPs/MEFs. Generally speaking, the higher the mitigation rate the more successful the MEF/WP program is. The most successful outcome of a mission impacting forecast is when operators accept the input and change their mission profiles to mitigate the risk. There will be instances where operators cannot change missions and must try to accomplish the sortie or mission despite a forecast. Consider tracking those situations separately.

The current metrics in Table 6 for "Criteria Event Accuracy" (CEA) and "No Criteria Event Accuracy" (NCEA) should more appropriately use the word "Precision" in place of "Accuracy" (AFI 15-114, 2017). Accuracy refers to the proportion of correct predictions while precision is the proportion of correct forecasts when an event is in the forecast. This paradigm does not adequately account for false alarms. If a forecaster correctly identifies 1 wind alert while missing 4 wind alerts his or her CEA will show 100%. A forecaster with no occurring high impact weather events scores 0%.

Table 7: Weather Forecast Technical Health Metrics (AFI 15-114, 2017)

Performance Metric	Formula	Description
Criteria Event Bias (Mandatory)	(A+C)/(A+B)	Criteria Event Bias reveals whether mission impacting events were either over or under forecast. Criteria Event WP Bias > 1 means Criteria Event WPs were over forecast. Criteria Event WP Bias < 1 means Criteria Event WPs were under forecast. For example, a Criteria Event Bias of 2 means mission impacting events were forecast 200% more than they occurred, a Criteria Event of 0.5 means mission impacting events were under forecast 50% of the time. It is important to compare Criteria Event Bias to the Criteria Event accuracy. An ideal balance would show the capability to predict mission impacting events without a high level of over forecasting.
No Criteria Event Bias (Optional)	(B+D)/(C+D)	"No Criteria Event WP Bias" reveals whether non-mission impacting weather forecasts events were either over or under forecast. "No Criteria Event Bias" > 1 means non-mission impacting weather forecasts were over forecast. "No Criteria Event Bias" < 1 means non-mission impacting weather forecasts were under forecast. For example, a No Criteria Event Bias of 2 means non-mission impacting weather forecasts were forecast 200% more than they occurred, a No Criteria Event of 0.5 means non-mission impacting weather forecasts events were under forecast 50% of the time. It is important to compare this metric with the No Criteria Event Accuracy.

AFW WP/MEF Technical Health Metrics account for the complexity of the OPVER MOPs. "Criteria Event Bias" is the inverse of Positive Predictive Value (PPV),

and “No Criteria Event Bias” is the inverse of Negative Predictive Value (NPV) (Heston, 2011). PPV and NPV are insufficient for AFW purposes because they fail to properly account for the “true rate of occurrence” (prevalence). By nature, high impact weather events are relatively rare. Relevant and representative metrics must have a mechanism for contextualizing the prevalence of low impact weather events.

Current Contingency Table Methods and Statistical Classification

Forecasters categorize a small sample of possible conditions and locations. AFW can use classification to fulfil the function of AFI 15-114 Weather Readiness Technical Evaluation, while minimizing the influence of variability in location and time. Current metrics practices do not intuitively connect MOPs to operational significance. Current metrics organize forecasts into a 2 by 2 confusion matrix with 4 possible categories with hourly counts in each category.

Table 8: Weather Conditions Chart (AFI 15-114, 2017)

		FORECAST Conditions (MEF/WPs)		
		Criteria Event WPs/MEFs	**No Criteria Event WPs/MEFs**	**Totals**
OBSERVED Conditions	**Criteria Event OBSERVED**	A	B (Miss)	Total Criteria Event OBSERVED: A + B
	No Criteria Event OBSERVED	C (False Alarm)	D	Total No Criteria Event OBSERVED: C + D
	Totals	Total Criteria Event WPs/MEF: A + C	Total No Criteria Event WPs/MEFs: B + D	Total WPs/MEFs: A + B + C + D

This contingency table forms the foundation for current OPVER practices, similarly it is the applicable for WWA evaluations as both lend themselves well to event-

based criteria evaluation. TAF evaluations uses a modified version of this thinking because they are multiple hour forecasts. This system evaluates TAF impacts hourly by phenomenon. False alarms and missed events, boxes B and C respectively earn 0 points. Similarly, the equivalent of boxes A and D are both scored as 1 point (AFI 15-114, 2017). The contingency table in Table 8 is more commonly referred to as a confusion matrix in the field of statistical classification (Mens, Korff, & Van Tol, 2012).

Statistical Classification

Forecasters utilize data at past points to categorize and predict future conditions. AFW forecasters essentially act as statistical classifiers similar to how Artificial Neural Networks (ANNs) can be used to classify data (Zhang, Patuwo, & Hu, 1998). Statistical classification relates to the problem of organizing new events into categories based on previous knowledge or data. In machine learning with ANNs, computers have a base set of known objects from which to recognize future objects (Bradley, Schwartz, & Hashino, 2008). Similarly, AFW forecasters consider previous knowledge on meteorological patterns and past observations at a given location. Future conditions will lie within one of multiple operationally relevant categories.

The prevalence of any weather condition relies upon the categorization scheme of the event. As seen in Table 4, unfavorable criteria for different operations can have different thresholds. High impact weather events occur at some locations more often with diurnal variation. Currently, AFW cannot fully account for the different forecast scenarios given to forecasters. This makes the reported values of "accuracy" a random and unfair to the forecaster because there is no representation of uncertainty.

PPV (NPV) describes the ratio of true positives to total positives (true negatives to total negatives), and thereby relates to the performance of a diagnostic test (Altman & Martin, 1994). The higher the PPV score, the more likely a diagnostic test is to correctly capture an event meeting the set criteria. PPV is thus related to the precision of a test., that is the ability to make a positive forecast that actually occurred (Altman & Martin, 1994; Heston, 2011).

$$\text{PPV} = 1 - \text{false detection rate} \quad (1)$$

$$\text{NPV} = 1 - \text{false rejection rate} \quad (2)$$

The medical field uses these metrics. In medicine, correctly obtaining true positives and true negatives has inherent value. Each good no call represents a patient who does not have to endure the trauma of medical treatment and testing for a disease that they do not have. Medical diagnosis positive or negative correspond to one of two outcomes, recommendation of treatment or non-recommendation of treatment (Heston, 2011). Stoplight charts have 3 sections as seen in the Table 4 example. This means that operators must consider weather as a factor in a decision and therefore all good no calls do not automatically imply a direct utility.

By extension, PPV and NPV results in statistics and diagnostic tests that are true positive and true negative results, respectively (Heston, 2011) (Altman & Martin, 1994). Both PPV and NPV rely heavily on prevalence, which is the actual rate of occurrence of a given parameter over a given period (Altman & Martin, 1994). In meteorology, the criteria threshold set for any given parameter is relatively rare in comparison to non-criteria events (Epstein, 1962) (Bradley, Schwartz, & Hashino, 2008) (McCreery & Phil,

2018). Therefore, statistics that do not consider the prevalence of the significant forecast events can heavily skew forecast metrics. Essentially, the count in block D in Table 8 is usually significantly larger than those in the other blocks. PPV and NPV represent the inverses of what AFI 15-114 calls "Criteria Event Bias" (CEB) and "No Criteria Event Bias" (NCEB) respectively. Equations 3 and 4 display PPV and NPV utilizing the block letters from Table 8. This research will refer to PPV and NPV rather than CEB and NCEB for readability purposes.

$$\text{PPV} = \frac{A + B}{A + C} \tag{3}$$

$$\text{NPV} = \frac{C + D}{B + D} \tag{4}$$

Both PPV and NPV, are sensitive to the rate of occurrence of an event, (prevalence) (Heston, 2011). PPV and NPV are insufficient foundations for use as AFW metrics because they both depend heavily on the prevalence of the event. PPV depends on the event occurring when predicted. Thus, Equation (3) depends heavily on block A in Table 8. Equation (4) depends heavily on block D in Table 8. **Error! Reference source not found.**PPV is difficult to contextualize when an event is rare. NPV is susceptible to large number of good no call forecasts. This influence of random chance makes comparing different forecasters and locations difficult, and disconnects the values of the metrics from the utility to the operator. The values do not directly indicate how many man-hours, flying hours, or dollars saved.

Positive likelihood ratios (PLRs) are an improvement over PPV because they are not subject to prevalence. PLRs depend on the ratio of correctly identified positive conditions, i.e. the "true positive rate" or (sensitivity) (Trevethan, 2017). PLRs also

depend on the ratio of correctly identified negative conditions, i.e. true "negative rate" (specificity) (Trevethan, 2017). As an example, a positive likelihood ratio is dependent upon the ability is the "True positivity rate" or sensitivity of a test divided by the "false positivity rate". PLR can be represented by the following equation utilizing the block labels from Table 8 (McGee, 2002).

$$PLR = \frac{\text{sensitivity}}{1 - \text{specificity}} = \frac{\frac{A}{A+B}}{1 - \frac{D}{C+D}} \quad (5)$$

PLRs allow AFW to change what constitutes a Good No Call and include it into replacements for PPV calculations. PPV can say how well a forecaster can forecast lightning given they forecast lighting. PLRs can answer how well a forecaster can predict lighting given that rain is either observed or in the forecast. This allows AFW to ask the same questions as the operator and therefore more closely relate skill scores to operational significance.

In diagnostic testing, PLRs perform a similar function to PPVs without the influence on prevalence (McGee, 2002). This is not automatically intuitive though and may be difficult to communicate to unfamiliar supported unit commanders. PLR values greater than 1 indicate skill while PLR values less than 1 indicate no skill in forecasts (McGee, 2002). PLRs account for random chance which means that the most pronounced improvement in current MOPs exists for WARNVER. WWAs have no skill until proven differently. This corresponds to a 50% probability of prediction. In statistical and medical literature PLRs correspond to the accuracy of a pretest (McGee, 2002). In this way PLR relates to change in probability of prediction by the relationship in Table 9.

Table 9: PLR and Probability of Prediction (McGee, 2002)

PLR	Probability of Prediction Change (%)	Probability of Prediction (%)
0.1	-45	5
0.2	-30	20
0.5	-15	35
1	0	50
2	15	65
5	30	80
10	45	95

PLR can relate directly to a probability of high impact weather event prediction. Further, PLRs limit the impact of random chance because it performs well regardless of the rarity of an event. PLRs cannot account for the variable costs of each type of error. There is no way to represent the difference between near misses and severe misses. Therefore, PLRs do not determine utility in terms of dollars, flying hours, or any other practical metric.

The current system lacks the flexibility to respond to variable costs associated with different forecast errors. Additionally, there is no way to account for the degree of error and the relative impact of that error on the mission. The influence of random chance is difficult to connect to the mission for comparison. Fortunately, there exists a metric in the literature that both accounts for random effects as well as varying consequences or benefits of incorrect and correct classification predictions.

Bayes Cost Metric

Conditional probability is a statistical representation of the probability of one event given that another event has occurred (Bradley, Schwartz, & Hashino, 2008). The conditional probability of two events A and $B$is Equation 6.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{6}$$

where $P(A \cap B)$ is the probability that events A and B occur simultaneously (McCreery & Phil, 2018).

The Bayes Cost (BC) metric is established in literature and meets all of the AFW metrics needs (Batterton & Schubert, 2013). This metric is responsive to changing mission needs, responds well to small sample sizes, and enables customizable mission cost impacts in terms of operational concerns (Batterton & Schubert, 2013). Bayes cost can be adapted to assign weight to various parameters of interest. These weights can consider both the value of precision of forecast as well as prevalence of the condition examined. The equation for Bayes Cost is:

$$BC = \sum_{\substack{i=1 \\ j \neq i}}^{n} \sum_{j=1}^{n} C_{i|j} \cdot p_j \cdot P_{i|j} \tag{7}$$

where n is the total number of categories, $C_{i|j}$ is the cost of classifying as category i when category j is observed, p_j is the prevalence of category j and $P_{i|j}$ is the conditional probability of classifying as category i given category j occurred (Skaltsa, Jover, Fuster, & Carrasco, 2012).

For BC, p_j and $P_{i|j}$ can be estimated. If x_{ijk} is defined as one if a classifier chooses category i when category j occurs in period k of m total periods and zero otherwise, then the estimate for p_j is

$$\hat{p}_j = \frac{\sum_{i=1}^{n}\sum_{k=1}^{m} x_{ijk}}{m} \tag{8}$$

with corresponding large-sample $100(1-\alpha)\%$ confidence interval:

$$\hat{p}_j \pm z_{\alpha/2}\sqrt{\hat{p}_j(1-\hat{p}_j)/m} \tag{9}$$

where $z_{\alpha/2}$ is the value of the standard normal random variable Z such that $P(Z \leq z_{\alpha/2}) = \alpha/2$. The estimates for the conditional probabilities $P_{i|j}$ over q periods are then

$$\hat{P}_{i|j} = \frac{\sum_{k=1}^{q} x_{ijk}}{\sum_{i=1}^{n}\sum_{k=1}^{q} x_{ijk}} \tag{10}$$

with corresponding large-sample $100(1-\alpha)\%$ confidence interval:

$$\hat{P}_{i|j} \pm z_{\alpha/2}\sqrt{\frac{\hat{P}_{i|j}(1-\hat{P}_{i|j})}{\sum_{i=1}^{n}\sum_{k=1}^{q} x_{ijk}}}. \tag{11}$$

The estimate for the BC metric calculated over m periods for the $\hat{p}_j$ values and over q periods for the $\hat{P}_{i|j}$ values is thus

$$\widehat{BC}_{mq} = \sum_{\substack{i=1\\ j\neq i}}^{n}\sum_{j=1}^{n} C_{i|j}\cdot\hat{p}_j\cdot\hat{P}_{i|j} = \sum_{\substack{i=1\\ j\neq i}}^{n}\sum_{j=1}^{n} C_{i|j}\frac{\sum_{i=1}^{n}\sum_{k=1}^{m} x_{ijk}}{m}\cdot\frac{\sum_{k=1}^{q} x_{ijk}}{\sum_{i=1}^{n}\sum_{k=1}^{q} x_{ijk}}. \tag{12}$$

The large-sample $100(1-\alpha)\%$ confidence interval for $\widehat{BC}_{mq}$ becomes somewhat more complicated, as it requires finding the joint multivariate normal distribution for all $\hat{p}_j$ and $\hat{P}_{i|j}$ and then performing the transformation of variables to find the resulting

distribution of the sum of the products in Equation (12). However, if the values of the p_js are known or if $m \gg q$ such that the estimates of the p_js are treated as known, then Equation (12) simplifies to

$$\widehat{BC}_{m \gg q} = \sum_{j=1}^{n} p_j \sum_{\substack{i=1 \\ j \neq i}}^{n} C_{i|j} \cdot \hat{P}_{i|j} = \sum_{j=1}^{n} p_j \sum_{\substack{i=1 \\ j \neq i}}^{n} C_{i|j} \cdot \frac{\sum_{k=1}^{q} x_{ijk}}{\sum_{i=1}^{n} \sum_{k=1}^{q} x_{ijk}} \tag{13}$$

When $m = q$, Equation (12) simplifies to

$$\widehat{BC}_{mq} = \sum_{\substack{i=1 \\ j \neq i}}^{n} \sum_{j=1}^{n} C_{i|j} \cdot \hat{p}_j \cdot \hat{P}_{i|j} = \sum_{\substack{i=1 \\ j \neq i}}^{n} \sum_{j=1}^{n} C_{i|j} \frac{\sum_{i=1}^{n} \sum_{k=1}^{m} x_{ijk}}{m} \cdot \frac{\sum_{k=1}^{m} x_{ijk}}{\sum_{i=1}^{n} \sum_{k=1}^{m} x_{ijk}} = \sum_{\substack{i=1 \\ j \neq i}}^{n} \sum_{j=1}^{n} C_{i|j} \frac{\sum_{k=1}^{m} x_{ijk}}{m} \tag{14}$$

Confidence intervals for BC theorem are legitimate regardless of the number of categories. The estimations of uncertainty of the variance of BC for three operational categories is quite complex. This is true even if each of the classes are normally distributed (Batterton & Schubert, 2013). From (Batterton & Schubert, 2013) we see that the estimate of the confidence interval for BC is:

$$\widehat{BC}_n \pm z_{\left(1-\frac{\propto}{2}\right)} \sqrt{Var\left(\widehat{BC}_n\right)} \tag{15}$$

Where n is the number of categories and the $Var(\widehat{BC}_n)$ is approximated as:

$$Var\left(\widehat{BC}_n\right) \approx \begin{bmatrix} \left(\frac{\partial BC_3}{\partial \mu_1}\right)^2 \sqrt{Var(\widehat{\mu_1})} + \left(\frac{\partial BC_3}{\partial \mu_1}\right)^2 \sqrt{Var(\widehat{\sigma_1})} + \\ \left(\frac{\partial BC_3}{\partial \mu_2}\right)^2 \sqrt{Var(\widehat{\mu_2})} + \left(\frac{\partial BC_3}{\partial \mu_2}\right)^2 \sqrt{Var(\widehat{\sigma_2})} + \\ \ldots \left(\frac{\partial BC_n}{\partial \mu_n}\right)^2 \sqrt{Var(\widehat{\mu_n})} + \left(\frac{\partial BC_n}{\partial \mu_n}\right)^2 \sqrt{Var(\widehat{\sigma_n})} \end{bmatrix} \tag{16}$$

where μ is the mean and σ is the standard deviation (Batterton & Schubert, 2013).

A non-parametric confidence interval sufficiently approximates of BC without the complexity associated with Equations 15 and 16. Further, a no assumptions regarding distribution are necessary. Batterton and Schubert used a non-parametric Bootstrap method to create confidence intervals. This method performed well regardless of the distribution of data (Batterton & Schubert, 2013).

Summary

The AFW mission is fundamentally different that the NWS in that the true accuracy of a predicted value is less important than identifying the correct category of weather conditions. These categories can change with the mission location and time. BC can account for the random chance associated with conditions and location. Also, the metric of choice needs to be directly related to utility of operational impacts such as fuel time and money saved.

PPV/NPV and PLR are incapable of meeting all the needs of AFW. Bayes Cost allows AFW to accurately quantify the contributions to the larger USAF mission. The framework is adaptable and responsive to changing missions and is effective regardless of the prevalence of the parameter examined. In the next chapter, this document will apply the statistical methodology detailed in this chapter to the AFW mission.

Representing uncertainty in its most accurate form can be computationally difficult. Commonly available techniques such as bootstrapping provide an easy way for AFW to represent uncertainty with no assumption of distribution.

III. Methodology

Chapter Overview

The purpose of this chapter is to chronical the methodology employed in this research. This chapter applies previous research to the demands of the AFW carrier field.

Practical Application of Binning

One can organize all possible weather conditions into one of 25 bins shown in Table 10 using the 5 PWC categories shown in Table 3.

Table 10: Weather Conditions Table (Pilot Weather Conditions)

	FORECAST (By Hour in MEF/WPs)					
OBSERVED (By Hour in MEF/WPs)	PWC 1	PWC 2	PWC 3	PWC 4	PWC 5	Total OBSERVED
PWC 1	x_{11}	x_{21}	x_{31}	x_{41}	x_{51}	$\sum_{i=1}^{5} x_{i1}$
PWC 2	x_{12}	x_{22}	x_{32}	x_{42}	x_{52}	$\sum_{i=1}^{5} x_{i2}$
PWC 3	x_{13}	x_{23}	x_{33}	x_{43}	x_{53}	$\sum_{i=1}^{5} x_{i3}$
PWC 4	x_{14}	x_{24}	x_{34}	x_{44}	x_{54}	$\sum_{i=1}^{5} x_{i4}$
PWC 5	x_{15}	x_{25}	x_{35}	x_{45}	x_{55}	$\sum_{i=1}^{5} x_{i5}$
Total FORECAST	$\sum_{j=1}^{5} x_{1j}$	$\sum_{j=2}^{5} x_{2j}$	$\sum_{j=3}^{5} x_{3j}$	$\sum_{j=1}^{5} x_{4j}$	$\sum_{j=1}^{5} x_{5j}$	Total Hours WPs/MEFs: $\sum_{i=1}^{5}\sum_{j=1}^{5} x_{ij}$

The above table shows over-forecasts in red and under-forecasts in yellow. Increasing over-forecasts indicate increased wasted time, and resources while increased under forecasts indicate increased exposure to mission limiting events. These categories can also be adapted to assess accuracy associated with each category.

AFW can evaluate utility regardless of the number of categories in each system. The stoplight chart in Table 4 organizes all forecast-observation pairs into one of 9 bins by phenomenon as illustrated in Table 11.

Table 11: Weather Scenario Matrix (Stoplight Chart)

	FORECAST (By Hour in MEF/WPs)			
OBSERVED (By Hour in MEF/WPs)	**Favorable**	**Marginal**	**Unfavorable**	**Total OBSERVED**
Favorable	x_{11}	x_{21}	x_{31}	$\sum_{i=1}^{3} x_{i1}$
Marginal	x_{12}	x_{22}	x_{32}	$\sum_{i=1}^{3} x_{i2}$
Unfavorable	x_{13}	x_{23}	x_{33}	$\sum_{i=1}^{3} x_{i3}$
Total FORECAST	$\sum_{j=1}^{3} x_{1j}$	$\sum_{j=2}^{5} x_{2j}$	$\sum_{j=3}^{5} x_{3j}$	**Total Hours WPs/MEFs:** $\sum_{i=1}^{3}\sum_{j=1}^{3} x_{ij}$

The confusion matrix in Table 11 displays over forecasts as yellow and under forecasts as red. This system can accommodate any number of classes. NATO categories work well with this system of organization as seen in Table 12.

Table 12: Weather Conditions Table (NATO)

	FORECAST (By Hour in MEF/WPs)						
OBSERVED (By Hour in MEF/WPs)	RED	AMB	YLO	GRN	WHT	BLU	Total OBSERVED
RED	x_{11}	x_{21}	x_{31}	x_{41}	x_{51}	x_{61}	$\sum_{i=1}^{6} x_{i1}$
AMB	x_{12}	x_{22}	x_{32}	x_{42}	x_{52}	x_{62}	$\sum_{i=1}^{6} x_{i2}$
YLO	x_{13}	x_{23}	x_{33}	x_{43}	x_{53}	x_{63}	$\sum_{i=1}^{6} x_{i3}$
GRN	x_{14}	x_{24}	x_{34}	x_{44}	x_{54}	x_{64}	$\sum_{i=1}^{6} x_{i4}$
WHT	x_{15}	x_{25}	x_{35}	x_{45}	x_{55}	x_{65}	$\sum_{i=1}^{6} x_{i5}$
BLU	x_{16}	x_{26}	x_{36}	x_{46}	x_{56}	x_{66}	$\sum_{i=1}^{6} x_{i6}$
Total FORECAST	$\sum_{j=1}^{6} x_{1j}$	$\sum_{j=2}^{6} x_{2j}$	$\sum_{j=3}^{6} x_{3j}$	$\sum_{j=1}^{6} x_{4j}$	$\sum_{j=1}^{6} x_{5j}$	$\sum_{j=1}^{6} x_{5j}$	Total Hours WPs/MEFs: $\sum_{i=1}^{6}\sum_{j=1}^{6} x_{ij}$

Applying Bayes Cost

In Equation 7, $C_{i|j}$ represents the cost of each of the possible categories. The corresponding cost matrix containing each of the $C_{i|j}s$. AFW can generate a $C_{i|j}$ matrix for dollars, fuel, flying hours etc. In this way, the supported unit determines the $C_{i|j}$, and approves the impact of each forecast conditions possibility.

Utility based $C_{i|j}s$ quantify forecasters mission impact rather than true forecast skill. This is helpful for communicating AFW utility with supported units. Correctly forecasting low impact weather conditions would therefore reward the forecaster less than correctly forecasting high impact weather conditions.

In this system, AFW retains the ability to assess the technical accuracy of forecasters. $C_{i|j}s$ can be set to equal to identify training needs. Or, AFW can measure true forecast skill by setting adjusting $C_{i|j}$ categories to a linear impact scale. Changing $C_{i|j}s$

can help AFW identify training needs and prepare for unknown future mission requirements.

The prevalence of each observed condition in Equation 7 is p_j. In many cases, p_j of various conditions have diurnal, seasonal, or annual fluctuations. AFW retains extensive observation records for many locations. Estimates of p_j improve with the volume of data for a given location. This allows for less error in BC approximations by reducing p_js contribution to BC uncertainty represented in Equation 9.

In BC, the $P_{i|j}$ is the only portion of the equation that is dependent upon the forecaster. The forecast represents the probability of each forecast-observation pairs. The number of $P_{i|j}$ is equal to the number of forecast categories squared. The stoplight chart in Table 4 would facilitate 9 $P_{i|j}$ categories conditions while missions utilizing NATO charts would necessitate at least 36 conditions.

The flexibility with the number of $P_{i|j}$ allows for interoperability in joint or combined operations through synchronization of mission criteria. This allows AFW to remain responsive to supported unit needs. As demonstrated in Table 4Table 4, not all operations are sensitive to the same parameters. Changes to the number of categories of classification result in changes to both $P_{i|j}$s and the corresponding cost $C_{i|j}s$.

$P_{i|j}$ are essentially telling us how a forecaster performed in any given hour. This is a statistic regardless of the sample size. These statistics will continue to improve the estimate over time but will always be an approximation of forecaster capability because of temporal and spatial uncertainty.

Representing Uncertainty

AFW can represent uncertainty on the raw BC metric on the hourly basis. If weather categorizations are normal then estimates can be proximate by z score through Equation 9. Python or R-code can utilize bootstrap confidence intervals regardless of distribution. This resampling method works well for small sample sizes. This makes bootstrap ideal for uncertainty for individuals or small teams.

These confidence intervals are responsive to mission needs. This allows for fair comparison between forecasters, units, and models. If confidence intervals do not overlap then there is statistical significance. More data will narrow the confidence interval and provide more certainty with time.

Applying Statistical Principles to WARNVER

Advanced warning also referred to as lead time is a primary focus of AFW WARNVER. Since WWAs are issued to support direct action on the part of the receiver. Utility is related to whether the receiver has enough time to execute the expected action.. WWAs organized into categories will have $P_{i|j}s$ with costs assigned as $C_{i|j}s$.

The only issue with this adjustment is that the length of the WWA valid period is not standard. This is unlike TAFs which are inherent representations of exposure to risk on the order of one hour at a time. By assuming that each WWA is issued for a standard duration AFW can compare forecasters more fairly. AFW would then weight p_i equally for every instance of WWA issuance.

Applying Statistical Principles to TAFVER

BC is an intuitive fit for TAFVER. TAFs forecast in hourly increments resulting in equal p_j weights to each observed condition. AFW can then measure each phenomenon separately and applied to an operationally based binning system such as those in earlier sections of this document.

The introduction of BC would change the categorization scheme from the current precision based point structure seen in Table 2 to the operationally based binning system created in Table 10. The utility of each forecast would be determined by the magnitude and type of forecast error $P_{i|j}s$ and the cost associated with that error $C_{i|j}s$.

Applying Statistical Principles to OPVERs

Forecasts for which OPVER applies are variable. However, the fundamental application of BC relates to the forecast ability to identify the correct operational category of observed weather conditions. For forecasts in hourly increments the application of BC is much the same as TAFVER. In operational forecasts timing error could determine utility. In this case $C_{i|j}s$ can be adapted to account for variable utility.

The most significant benefit of BC to OPVER is the ability to quantify skill in terms of operational significance without the susceptibility to the prevalence. In BC, $P_{i|j}$ has a corresponding $C_{i|j}$ the prevalence of any condition does not impact that relationship. This ensures that forecasters and supported mission commanders can view performance in terms of the same units of measurements. This enables comparisons across forecasters and units.

Conditional Probability in AFW Ops

AFW can leverage conditional probability to identify areas for future training, and support acquisitions, and mission planning. Conditional probabilities allow AFW to know how often a forecast condition occurs given that the condition was forecasted and vice versa. Operationally based binning allows AFW to examine forecast skill by category.

The prevalence of lower category conditions can dominate skill scores. Utilizing conditional probabilities, AFW can identify forecast accuracy by operational category.

Currently, weather intelligence for planning purposes includes a climatology review. Conditional probability allows AFW to consider the ability to mitigate conditions with the climatology. As an example, plans propose location A or location B for the same mission. These locations have the same prevalence of mission limiting winds. Using Conditional probability AFW determines that the probability that mission limiting winds are observed given that they are forecasted is 40% at location A but 75% at location B. Further, AFW determines that the probability that mission limiting winds are forecasted given that they are observed is 35% at location A but 85% at location B. This means that AFW can mitigate risks much better at the recommended location B. Uncertainties can be represented on this percentage utilizing z score for sufficiently large sample sizes with normal distributed.

Implementation of Bayes into AFW Ops

To implements BC as a primary metric the first step is to establish a process for assigning $C_{i|j}$s. Each type of supported mission can have its own set of $C_{i|j}s$ matrices.

Wing level leadership could more easily establish multiple $C_{i|j}$ matrices for flying hours, dollars, fuel and money.

Secondarily, AFW can account for timing error as needed in several ways. Operations tempo can vary by time of day so in this regard some units of utility can change diurnally. Similarly, due to manpower variations can change sensitivity to other parameters. $C_{i|j}s$ can change with timing error. Forecasts of the correct category with 0 hours of timing error can have a different value than the same forecast with some timing error. This timing error is an extra dimension on the $C_{i|j}$ matrix.

Error is represented by resampling the overall BC value to produce a system of Cis impervious to the distribution of data. This would allow AFW to use a consistent technique for most applications of BC. The number of resamples is flexible ensuring adequate representation of the entire dataset.

The next step is to construct an automated system to calculate these quantities. AFW can incorporate the BC metric evaluation tool into existing infrastructure such as the metrics page housed on the Operational Weather Squadrons web pages, or the Integrated Weather Warnings Capability (IWWC).

The utility of these metrics relies on the assumption that the weather products have some value. Substantial research has concluded that weather forecast products such as TAFs have a quantifiable impact on operationally significant parameters.

Value of Weather Products

TAFs define the expected take-off and landing conditions. The value of TAFs seems to be most pronounced in the redux of unnecessary fuel and mitigation of

unnecessary delays. In one study, TAFs reduced fuel consumption by 6% by reducing unnecessary over fueling and eliminating unnecessary weight (Anaman, Quaye, & Owusu-Brown, 2017). In this same study, TAFs were responsible for the 12% reduction in avoidable delays, and the 7% redux in avoidable cancelations (Anaman, Quaye, & Owusu-Brown, 2017). Additionally, fuel and diversions associated with weather were found to total 25% of operating expenses for airports further establishing the value of TAF based services (Klein, Kavoussi, & Lee, 2009).

Summary

This chapter applies the statistical principles in Chapter 2 to the AFW mission. AFW forecasts have a quantifiable impact on the larger USAF mission. Bayes Cost can be applied to meet the needs of AFW while being responsive to changing mission demands.

IV. Analysis and Results

Chapter Overview

The purpose of this chapter is to examine the results of a brief application of the statistical principles outlined in the first two chapters of this document. This case study involving data from 21 locations during May and June 2019. This represents a small proof of concept of the power Bayes Cost analysis as a primary metric.

Data Collection

The USAF does not retain a historical database of forecast or computer generated TAFs. This research used the open source weather information service "OGIMET" to obtain historical human TAFs for all sites. This research uses only regular 30-hour TAFs and not amendments. TAF only queries are available in 30-day increments. This research utilized downloaded text files though html is also available at (OGIMET, 2019).

This research uses observation data in text file format from the 14th Weather Squadron. These data organize the worst conditions for all parameters within each hour at each location regardless of the duration of that worst condition. The worst conditions use the METAR and any SPECIs within that hour at that location.

Model TAFs are issued simultaneously with human TAFs for each location. A data save can archive these TAFs for some time. The data requested were incomplete and insufficient for this research. The text model data for each model run was more complete. The case study simulated Auto TAF data from the first 30 hours of each run for each location. The model of consistency has four runs per day for each location. Auto TAF start times are the model run times.

WWA data is available directly from AFW websites. As a demonstration, this study includes the aggregated WWA totals of all 21 locations for Moderate Thunderstorms and Lightning. These data include 2 months of data from all 21 locations. Further this research assumes 1 hour long valid periods for each WWA.

Preparing TAF Data for Analysis

This research formatted both auto and human generated TAFs into a 30-line comma-separated values (CSV). Each line of every TAF has location time information unique. This research compares each forecast hour to the corresponding observation using python code. This analysis categorizes human and auto TAFs into notional operational categories seen in Table 13.

Table 13: Categories by Parameter

Wind Speed	Ceiling	Visibility	Wind Direction Error	Categories
15	2500	5	30	0
20	1500	3	45	1
25	700	2	60	2
35	300	1	90	3
45	200	0.5	120	4
55	<200	<.5	150	5

The python code iterates through each hour of the TAF and identifies various aspects of the forecast of expected conditions. Collected observation data at all locations contained instances of missing data for various parameters of interest. This study replaces

missing data with a “Category 0” value from Table 13. This carries the assumption that all missing data is not part of a high impact weather event.

Preparing WWA Data for Analysis

This research assumes that all WWA was issued and active for only one hour. WWAs with at least 50% lead time are hits counted as hits while fewer than 50% desired lead time are misses. This study calculates PPV, PLR, PLR rain, and BC. PLR rain reduces the number of good non-forecasts by 90% to simulate reduction in interesting events. In this case, the ability to discern lighting events from non-lighting events given that it is raining or forecast to rain. Lastly, BC uses a simulated value for hits, misses, false alarms.

Case Study Methodology

TAFs forecast for the same hour multiple times since the forecast is for 30 hours and the gap between TAFs is typically every 6 to 16 hours. The additional events for auto TAFs constitute additional opportunities for high impact weather events to occur and therefore skew the total exposure within the control calculations.

Cost analysis development

The structure for cost analysis begins by establishing a reliable baseline for a control corresponding to maximum exposure to weather events. In this case, the weather events that occur will define the maximum exposure in hours to each weather phenomenon. This study aggregates data from all 21 locations. This assumes no variation in mission or mission impact with location. Further, costs of each type of forecast error remain constant regardless of time of day.

From Equation 7, $C_{i|j} \cdot p_j \cdot P_{i|j}$ are the terms in BC. These calculations use notional $C_{i|j}$s as seen in Table 14 using SME input. Auto and human TAFs will have the same $C_{i|j}s$ since they are for the same notional mission at the same notional locations. This study uses $P_{i|j}$s developed by sorting all forecast-observation pairs into 1 of 36 bins. This yields a percentage of occurrence for each of the 36 possible conditions. Auto and human TAFs will have different $P_{i|j}$s because the number of TAFs and the valid hours of the TAFs are not the same. Some human TAFs skip days while auto TAFs do not. Auto and human TAFs will have different $p_j s$ as each1440-hour scenario have different weather conditions as represented by the $P_{i|j}$s. BC metric quantifies the value of weather forecasts in each scenario in comparison to some control value.

This control value for both human and auto TAFs evaluated in this study is the exposure created by multiplying the aggregated observed events for each j by $C_{i=1|j}$. That is analogous to the "Total Observed" column from Table 12, and the "Forecast 0" from Table 14 respectively. This corresponds to the occurrences of each scenario assuming there was no advanced warning. This study represents error on BC rather than the individual components of BC. This estimate is sufficiently accurate as outlined in Chapter 2.

Table 14: Cost Table for Winds (Bayes Cost Analysis)

Wind Cost	Forecast 0	Forecast 1	Forecast 2	Forecast 3	Forecast 4	Forecast 5
Observation 0	0	-1	-100	-500	-8000	-10000
Observation 1	-100	100	-50	-400	-5000	-8000
Observation 2	-1000	-500	1000	-500	-2500	-5000
Observation 3	-5000	-5000	1,500	4,000	-1,000	-2,500
Observation 4	-10,000	-10,000	2,000	4,000	10,000	8,000
Observation 5	-1,000,000	-1,000,000	200,000	400,000	600,000	800,000

Table 14 is a table of the notional $C_{i|j}s$ used to calculate BC on wind values. The $C_{i|j}s$ used to calculate BC on ceiling data are equivalent to 1/75 of each $C_{i|j}$ for wind data.

Data Distribution Results

Human generated TAFs and auto generated TAFs exhibit similar distributions; a direct comparison of raw forecast error for each of the two primary phenomena examined as seen in Figure 1.

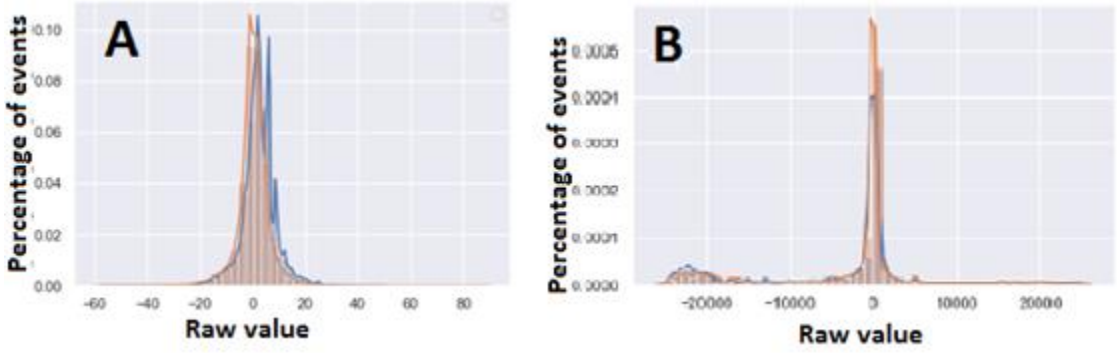


Figure 1: Raw Distribution X- axis is the observed value Y- axis is percentage of occurrence A) Wind distribution (kts); B) Ceiling distribution (feet)

The similarity in distribution of error across forecast parameters proved to be a limiting factor in establishing statistical significance of any differences in forecasts. This research applies confidence intervals to the average raw error for each hour aggregated across all the 21 locations. Utilizing confidence intervals derived from z-score it is difficult to establish statistical significance at the 95% confidence level as seen in Figure 2.

Figure 2 applies operationally based binning. In the ceiling example, it is apparent that it is possible for the more erroneous forecast to be the more useful. In Figure 2.A, we can see that the Auto TAF has the more consistently accurate forecast across all hours of the day. However, in part Figure 2.B, the human TAF is consistently closer to the correct category based on utility. This highlights the power of operationally based binning. In terms of utility, error is tolerable when the error does not prompt any adverse actions or additional risk.

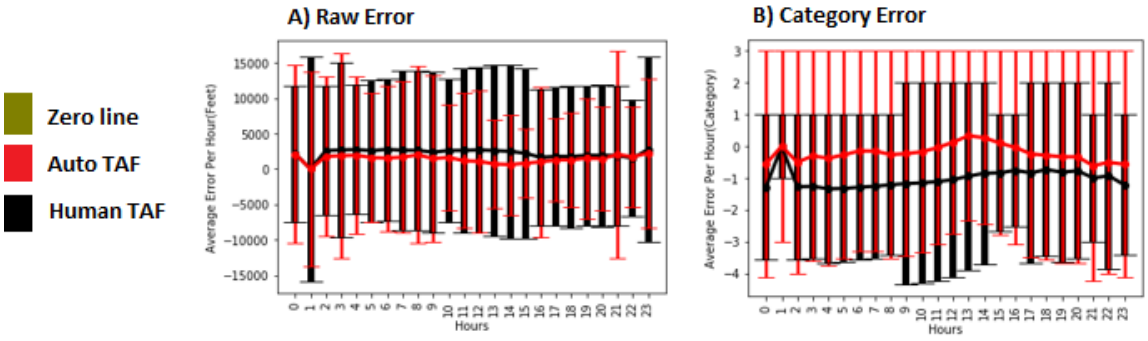


Figure 2: Wind Raw Vs Category Error A) Wind raw difference error by hour B) Wind category error by hour

There is significant overlap in the 95% confidence intervals for both the raw and category differences. This overlap means that the quantifiable difference between human

and auto forecasts could be due to random chance instead of skill. Figure 2.B indicates that there is a tendency for human forecasters to make ceiling forecasts less extreme when the auto forecast expects to see a ceiling event.

More common events dominate averages of error. It is helpful to consider only events of interest. Figure 3 examines the ceiling forecast data by considering the percentage of observed and forecasted events at or above category 1 in Figure 3.A and at or above category 2 in Figure 3.B. The total area of each Venn diagram represents the number of interesting events. An interesting event here is when a Category 1 event or higher occurred or was forecasted to occur.

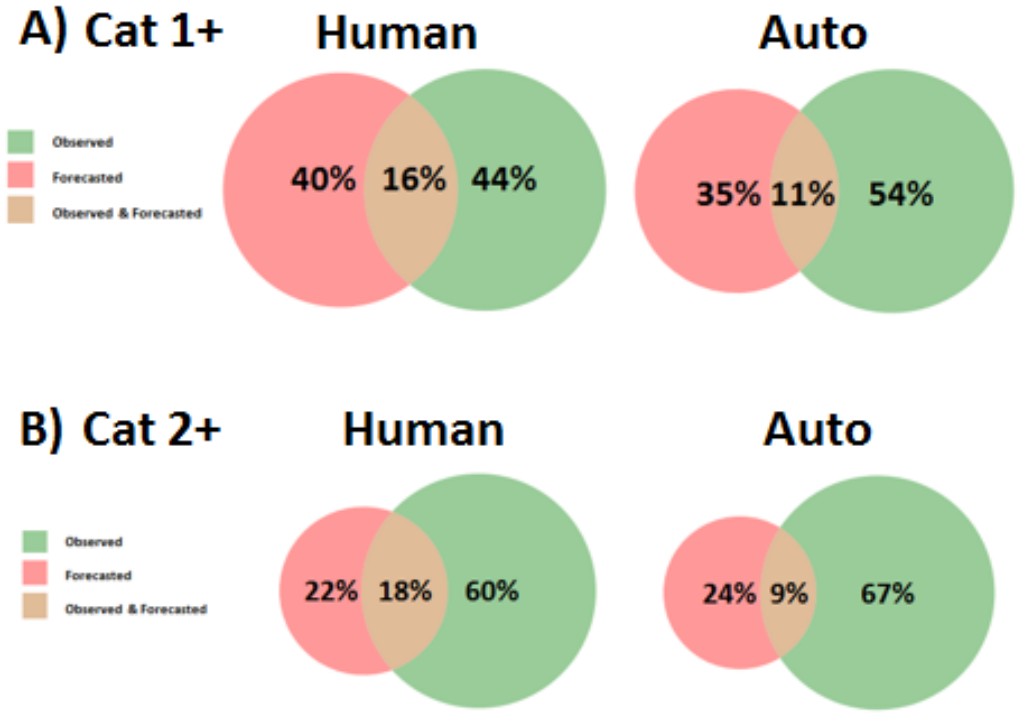


Figure 3: Ceiling Venn Diagram Ceiling forecast A) Category 1 and higher events forecast and/or observed. B) Category 2 and higher events forecast and/or observed.

In Figure 3: Ceiling Venn Diagram **Ceiling forecast A) Category 1 and higher events forecast and/or observed. B) Category 2 and higher events forecast and/or observed.**Figure 3, the circles are proportional within each of the four subplots. Human and auto TAFs both saw an increase in the percentage of observed events that went forecasted. This means that as the category increased and the events became more intense, the forecasts yielded a higher percentage of missed events.

Though human forecast error was greater than the auto forecast for both raw and category error in Figure 2, Figure 3 indicates that human forecasts have a higher percentage of identifying events that are forecasted. Both human and auto TAFs forecast significantly less events than what occur, however, a higher percentage of human forecasts occur. Beyond this, there is a slight increase in the ratio in the overlapping section for human TAFs while the percentage drops in auto TAFs. The percentage of overlap for human TAFs is double that of auto TAFs in the Category 2 and up section.

This indicates that human's ability to identify more significant events may be better than auto TAFs, though there may be more false alarms as well.

Due to the non-linear operational significance of category this table indicates that averaging of overall forecast accuracy is insufficient to calculate the utility. Further, basic frequentist statistics do not highlight how actionable a forecast is when a category event is issued.

Conditional probabilities are the next step in understanding the utility of these data. Confidence intervals assume normal distribution over time of the accuracy values. Table 15 notionally assumed to be equivalent to 1 location over 21 years. This simulates annual analysis of AFW forecasts by location.

Table 15: Conditional Probability Sample with Confidence Interval (Wind Speed)

Forecaster	Phenomenon	Category	Probability (A|B)	Uncertainty (A|B)	Probability (B|A)	Uncertainty (B|A)
Human	Wind Speed	1 +	59	Upper: 65 Lower: 52	41	Upper: 53 Lower: 34
Human	Wind Speed	2 +	45	Upper: 57 Lower: 31	16	Upper: 21 Lower: 9
Auto	Wind Speed	1 +	52	Upper: 63 Lower: 44	51	Upper: 66 Lower: 38
Auto	Wind Speed	2 +	29	Upper: 44 Lower: 16	22	Upper: 38 Lower: 7

The Probability of (A|B) represents the reliability of a forecast. This is the probability that an event was in the forecast when it is known that an event occurred. The functionality of the forecast is best highlighted by the probability of (B|A). This is the probability that an event occurred when it is known that an event was in the forecast. The Appendix includes a table with of windspeed conditional probabilities.

Figure 4 shows conditional probabilities with forecast error type. This figure shows the distribution of forecast error by category given that an event either occurred or was forecasted. Significantly over or under forecast indicates 2 categories or more of difference between forecast category and observed category.

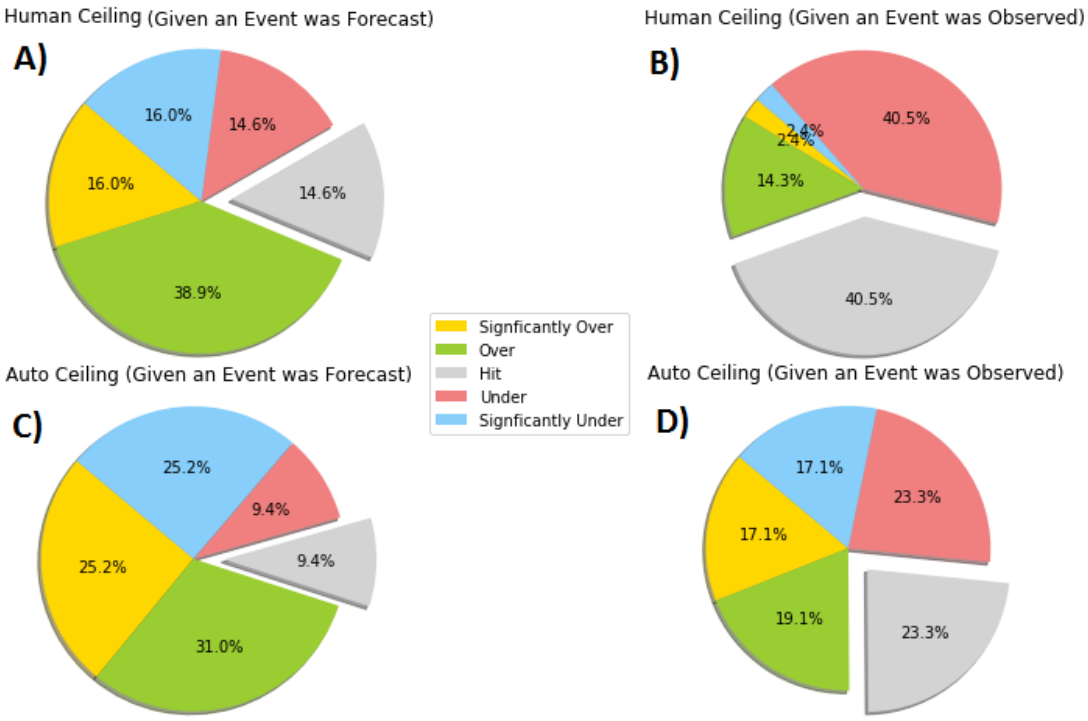


Figure 4: Ceiling Conditional Probabilities A) Human [Observed given event forecasted] B) Human [Forecasted given event observed] C) Auto [Observed given event forecasted] D) Auto [Forecasted given event observed]

Figure 4 reveals at the human TAF is more precise than the auto TAF within each category. This figure shows that human forecasts are more precise than auto forecasts. There is nearly double the correct category identification, shown as “Hit”.

Table 16 shows the percentage correct forecasts by category. Table 16 is from the diagonal along the column normalized contingency tables included in full in the appendix of this document.

Table 16: Precision by Forecast Category

Forecaster	Phenomenon	Category 0	Category 1	Category 2	Category 3	Category 4	Category 5
Human	Ceiling	94	12	26	36	27	15
Auto	Ceiling	96	16	29	34	6	4
Human	Wind Speed	93	29	12	4	2	0
Auto	Wind Speed	91	39	17	7	4	6

This table shows that the accuracy in forecasting Category 0 events is no less than 91 for either human or auto generated TAFs. At or above Category 3, the accuracy decreases with increasing Category. This means as events get more rare and impactful forecast accuracy decreases. Table 17 features the contingency table by percentage for all possible combinations of observation and forecast categories.

Table 17: Percentage of Occurrence Contingency Table Example (Ceilings)

Percent by Occurrence	Forecast Cat 0	Forecast Cat 1	Forecast Cat 2	Forecast Cat 3	Forecast Cat 4	Forecast Cat 5
Observed Cat 0	0.7157	0.0777	0.0285	0.0058	0.0007	0.0015
Observed Cat 1	0.0173	0.0147	0.0095	0.0017	0.0001	0.0001
Observed Cat 2	0.0166	0.0173	0.0206	0.0080	0.0003	0.0002
Observed Cat 3	0.0081	0.0087	0.0140	0.0121	0.0011	0.0007
Observed Cat 4	0.0017	0.0014	0.0029	0.0045	0.0010	0.0003
Observed Cat 5	0.0023	0.0008	0.0013	0.0016	0.0005	0.0005

In Table 17 the sum of all the values are equal to one. The values along the green diagonal from upper left to bottom right represent the correct forecast for each category. The yellow section above the green diagonal represents over-forecasts and a false alarm. The red section below the green diagonal represents missed events or under-forecasts. In this format, the prevalence of lower end events is evident of the likelihood of lower

category events. Table 14 displays the costs associated with the wind events while the similar table for ceilings is Table 18.

Table 18: Operational Costs Per Hour (Ceilings)

Ceiling Costs/Hour in Dollars	Forecast Cat 0	Forecast Cat 1	Forecast Cat 2	Forecast Cat 3	Forecast Cat 4	Forecast Cat 5
Observed Cat 0	0	0	-1	-7	-107	-133
Observed Cat 1	-1	1	-1	-5	-67	-107
Observed Cat 2	-13	-7	13	-7	-33	-67
Observed Cat 3	-67	-67	20	53	-13	-33
Observed Cat 4	-133	-133	27	53	133	107
Observed Cat 5	-13333	-13333	2667	5333	8000	10667

Table 19 shows case study BC results to human and auto TAF scenarios. Auto TAFs have more hours included and therefore different p_js for the $P_{i|j}s$. This results in a different control value associated with the maximum risk of the theoretical scenario.

Table 19: Case Study Results

Phenomenon	Factors	Control (Human)	Forecast (Human)	Control (Auto)	Forecast (Auto)
Winds	Total Risk Exposure	-$290,707	-$198,781	-$246,564	-$164,883
	Hourly Risk Exposure	-$202	-$138	-$171	-$114
	% Value Added	N/A	32%	N/A	33%
95% (Confidence Interval)	(Hourly Risk Exposure)	N/A	(-$211, -$73)	N/A	(-$173, -$62)
	(Total Risk Exposure)	N/A	(-$303,840, -$105,120)	N/A	(-$249,120, -$89,280)
	(Value Added)		*(-5%, 64%)*		*(-1%, 64%)*
Ceiling	Total Risk Exposure	-$171,963	-$27,897	-$144,168	-$29,866
	Hourly Risk Exposure	-$119	-$19	-$100	-$21
	% Value Added	N/A	84%	N/A	79%
95% (Confidence Interval)	(Hourly Risk Exposure)	N/A	(-$-26, -$15)	N/A	(-$25, -$15)
	(Total Risk Exposure)	N/A	(-$37,440, -$21,600)	N/A	(-$36,000, -$20,160)
	(Value Added)		(78%, 87%)		*(75%, 86%)*

The hourly estimate of uncertainty is broad even with 90 thousand human TAF forecasts and 133 thousand Auto TAF forecasts. The true value of TAFs relies on reducing risk exposure. Wind value added indicates a lack of skill in identifying and capturing high impact events. The relatively high value added for cig indicates elevated skill in identifying high impact events when they occur.

The bootstrap confidence intervals use 10 thousand resamples for human TAFs categories and 15 thousand resamples for auto TAFs. These sample means are gathered and calculated 1000 times. The size of the sample is to maintain the near 10% subsample of the total number of forecasts. The distribution with the mean value identified is in

Figure 5.

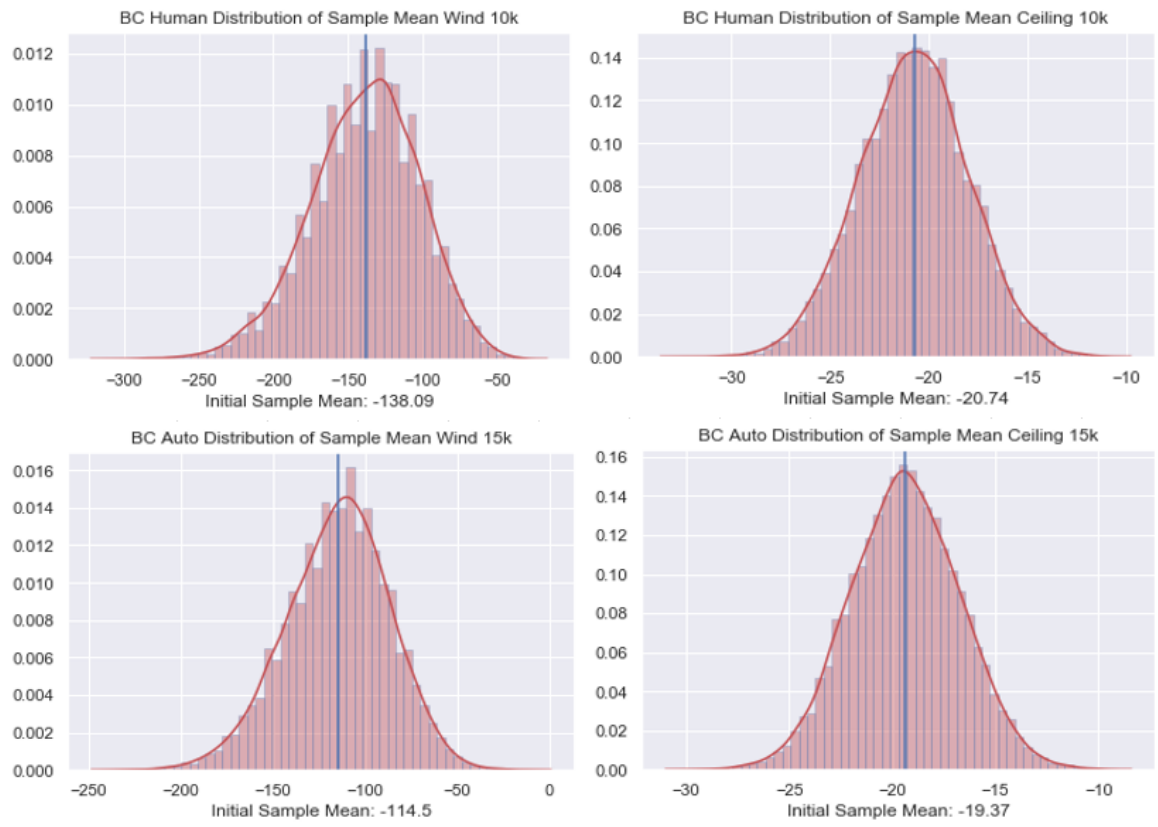


Figure 5: Bootstrap Distribution of Sample Mean

This method of bootstrapping from subsamples allows for a relatively accurate approximation while invoking the Central Limit Theorem. This process supports the consistent application of error estimation in BC calculations. This consistency does increase uncertainty from more rigorous methods, but results provide relative assessment of performance for AFW mission sets.

The distribution of mean wind is more skew for both auto and human data in Figure 5. These data are skew for wind forecasts. The nature of distribution of BC values could be different in the event there are no high impact under and over forecasts. The distribution of values depends heavily on the nature of categories and the relative value of $C_{i|j}s$.

Beyond the estimates of human value added, it is possible to utilize conditional probability to provide additional information about the utility of forecasts. Table 20 research displays conditional probabilities are along with errors based on the credibility interval. These ranges allow researchers to assess whether it is likely that these differences are significant.

Table 20: Windspeed Conditional Probabilities

Forecaster	Phenomenon	Category	Probability (A|B)	Uncertainty (A|B)	Probability (B|A)	Uncertainty (B|A)
Human	Wind Speed	1 +	59	Upper: 65 Lower: 52	41	Upper: 53 Lower:34
Human	Wind Speed	2 +	45	Upper: 57 Lower: 31	18	Upper: 21 Lower: 9
Auto	Wind Speed	1 +	52	Upper: 63 Lower: 44	51	Upper: 66 Lower: 38
Auto	Wind Speed	2 +	29	Upper: 44 Lower: 16	22	Upper: 38 Lower: 7

Table 20 illustrates that human and auto forecasts perform more similarly in category 2 and higher than with category 1 data included. These conditional probability data indicate that human TAFs are more likely than auto TAFs to identify wind events but with more false alarms. This case study also illustrates BC applied to WWAs. Table 21 shows PPV, PLR, and BC for two types of WWAs organized into 4 categories.

Table 21: Moderate Thunderstorms and Lightning

Moderate Thunderstorms	**Criteria (Forecasted)**	**Criteria (No Forecasted)**	**Totals**
Criteria (Observed)	(A) [Hit] {16 events} ($75,000 per event)	(B) [Miss] {20 events} (-$75,000 per event)	Total Criteria Event Observed (A+B) {36}
Criteria (Not Observed)	(C) [False Alarm] {81 events} (-$ 7.5 per event)	(D) [Good Non-Forecast] {1323 events} ($1 per event)	Total Non-Criteria Event Observed (C+D) {1404}
Totals	Total Criteria Event (A+C) {97}	Total Non-Criteria Event (B+D) {1343}	Total of all Boxes (A+B+C+D) {1440}
PPV: 2.69	**PLR: 7.70**	**Sensitivity: 0.44**	**Specificity: 0.94**
PPV (Rain): 2.69	**PLR (Rain): 1.17**	**Sensitivity (Rain): 0.44**	**Specificity (Rain): 0.62**
Control Exposure: 2.7M	**Control Exposure hours: 36 = (A+B)**	**Human Adjusted Exposure: 1.5M**	
BC Scenario Net Value: .4M	**BC Average WWA Value: -207**	**Lower Hourly Uncertainty: -832**	**Upper Hourly Uncertainty: 469**

Lightning	**Criteria (Forecasted)**	**Criteria (No Forecasted)**	**Totals**
Criteria (Observed)	(A) [Hit] {222} ($10,000 per event)	(B) [Miss] {12} (-$10,000 per event)	*Total Criteria Event Observed (A+B) {234}*
Criteria (Not Observed)	(C) [False Alarm] {90} (-$1 per event)	(D) [Good Non-Forecast] {1116} ($0.01 per event)	*Total Non-Criteria Event Observed (A+B) {1206}*
Totals	*Total Criteria Event (A+C) {312}*	*Total Non-Criteria Event (B+D) {1128}*	*Total of all Boxes (A+B+C+D) {1440}*
PPV: 1.33	**PLR: 12.71**	**Sensitivity: 0.95**	**Specificity: 0.93**
PPV: 1.33	**PLR (Rain): 2.34**	**Sensitivity (Rain): 0.95**	**Specificity (Rain): 0.60**
Control Exposure: 0.7M	**Control Exposure hours: 234= (A+B)**	**Human Adjusted Exposure: .12M**	
BC Scenario Value: 0.58M	**BC Average Hourly Value: 1625**	**Lower Hourly Uncertainty: 1305**	**Upper Hourly Uncertainty: 1638**

Table 21 demonstrates that PPV is not sensitive to changes in the frequency of the "Good Non-Forecast." PLR technique yields an extremely high skill. Reducing block "D" by 90% simulates changing the definition of a good non-forecasts. This definition means that good-non-forecast must have precipitation observed or forecast in that hour. The lower "PLR Rain" skill score better describes the human ability to discriminate

between non criteria rain events vs criteria rain events. Neither PPV nor PLR quantify the impact in terms of dollars. BC estimates the value the WWAs and bootstrapping provides estimates of uncertainty using the given values ascribed to each phenomenon in Table 21. The distribution of BC for WWA is in the appendix.

Summary

This chapter demonstrates the value of BC as a primary metric. The case study uses two months of weather data at 21 locations. This sample of forecast data outlines the flexibility of BC.

V. Conclusions and Recommendations

Chapter Overview

The purpose of this chapter is to discuss the significance of the research conducted in this thesis. This research serves as a proof of concept of applying well sited statistical methods to military TAF and WWA metrics. The application of new metrics to military forecast allows forecasters to examine the utility of human military forecasts and WWAs. In applying these statistical methods, several conclusions are helpful for understanding the implications of these findings, suggesting actions for the AFW community, as well as, focusing further research efforts.

Conclusions of Research

Ultimately, this research demonstrates the BC metric performance for both TAFs and WWAs. BC estimates the skill of AFW forecasters in the unit that directly relates to utility. BC performs well regardless of the number of categories of classification. This metric is useful over time and across multiple locations. The $C_{i|j}s$ in BC will have supported unit input by mission type. This allows AFW to consistently quantify impact across weather units. BC uses confidence intervals to support fair comparison between forecasters and computers. Prevalence of weather conditions do not distort BC. This property makes BC an ideal choice for AFW. This is because high impact weather conditions are relatively rare.

Representation of error and utility assessments rely on consistency within the sample of compared items. TAFs have this consistency in that the duration of each forecast condition is generally 1 hour. WWAs do not have this consistency introducing

large sources of error into the p_j term. The case study removes this by assuming all WWAs are issued and verified as a hit miss or false alarm within 1 hour. This research recommends standardizing WWA lengths to maintain consistency in verification.

Significance of Research

As discussed extensively in Chapter 1 of this document, metrics are key to relating services provided to customer utility. This initial proof of concept proposes a second generation of statistics that, if continued, has the potential to address fundamental gaps in current AFW metrics. BC metric system can enhance AFW support and manpower distribution and even acquisitions when used in conjunction with climatology.

Potential for Manpower Distribution

By quantifying the utility of TAF production by location, it is plausible for AFW to use similar calculations as a component of the decision to re-distribute manpower. Analysis using these techniques have the potential to optimize task distribution and assigning the forecasters to an appropriate ratio of TAFs to WWAs.

Recommendations for Future Research

Investigate Diurnal Variations in Utility

This research treats every hour of a forecast as equally important. There is some inherent error in this assumption. In general, supported clients do not maintain a constant operations tempo for the duration of a 24-hour day. This means forecast error is likely variable on a diurnal time scale. During these hours, the statistical difference in many of

the aspects of TAFs would be less impactful. Investigating the diurnal variations in operations tempo could yield more precise estimates of forecast utility.

Investigate TAF and WWA Issuing Location

Some TAFs issued by personnel that are physically located within the forecast area. This co-location yields potential advantages in observing initial conditions on site that could be evident within the first couple of hours of a TAF. Similarly, some WWAs are issued by forecasters who are physically located on station. This proximity may result in fewer false alarms of WWAs. It is possible that familiarity with local weather patterns and terrain features unresolvable by the model could improve high impact weather forecasts.

Investigating the relationship between forecaster proximity to the forecast area has the potential to yield an improved understanding of maximum efficiency of forecasters. It is possible to enhance AFW understanding of human generated forecast variability attributable to forecaster proximity.

Investigate WWA Issue Format

Issuing such warnings every hour standardized duration is like the process of issuing convective alerts known as Significant Meteorological Alerts (Convective SIGMETS) at the Aviation Weather Center. Currently, the ability to extend WWAs without reissuing and validating them separately skews the effective utility assessment by hiding False Alarm hours from the technical health metric. Establishing standard one or two-hour WWA length would improve utility assessment and subsequently communication of mission contribution.

Examine Utility of Civilian TAFs within an Aerodrome

As a reminder, civilian TAFs do not regard to the size of the aircraft for certain parameters of a TAF like turbulence. Comparing civilian TAFs to military TAFs for basic parameters such as wind, ceiling, and visibility could have value. This is especially true for civilian locations that are within 5 miles of military TAF locations. This assessment could help assess the spatial variability in point forecasting.

Investigate Spatial Uncertainty in Observations

As stated in Chapter 2, part of the challenge of TAFs is associated with taking data with inherent spatial uncertainty to produce a point forecast. Conducting the techniques proposed in this document, but utilizing all available observation sites in an aerodrome could improve understanding of spatial variability. The uneven spatial distribution of observations sites will reduce the comparability of one site to another, however, it is possible to reduce error bars on statistics conducted by increasing the number of observation sites and better representing forecasts as areal as opposed to point forecasts. The improvement in this representation will improve estimates given by the utility calculation proposed. AFW verifies WWAs using any observation reported within an aerodrome.

Mission Variability

Every mission has a different sensitivity to weather and the interview used to inspire the notional parameters in the utility calculator are from one type of mission. More surveys and an in-depth study can provide more precise data for the utility calculator and thus yield a more accurate representation of AFW utility by-product type.

Complete Picture Assessment

AFW provides aviation hazard charts, special forecasts, and briefings in person and by various media, and consultants to headquarters and other services. BC can is valid for all these services.

The effort that AFW expends to conduct analysis by product type could pay dividends by allowing for the accurate representation of contribution to the mission.

AFW provides consultants to combatant and component command headquarters. This may increase in utility, as weather intelligence both military and commercial, becomes more diverse in complexity and availability. AFW could apply BC to these special services and products to provide a holistic picture of the future of AFW.

Investigation of Complex Weather Events

This research neglects the existence of non-zero category events across multiple phenomenon. To develop a more rigorous estimate of AFW contributions to the mission, it is necessary to consider these cases. Other instances of complex events include single phenomena events that alternate. For example, it is possible for a wind event to follow a ceiling event. In this case, utility of each event may have a reduced impact given the action taken in advance of the other.

Complex weather analysis due to timing error is important. This research evaluates a forecast on the hourly basis. A forecast that was accurate in magnitude but off in time was incorrect. This leads to error in the outcome of the case study. From a utility perspective a timing error may not be important to a customer. More research can determine the effect of temporal accuracy on in garrison missions.

Recommendations for Action

Data Collection

The most urgent recommended action is data collection. AFW can use previous TAFs and Computer TAFs to understand which types of forecasts and locations have utility. This proof of concept is a comparison between model TAFs and Human TAFs. Issued TAFs are available open source; it is arduous to accumulate the volume of data and these data are available from a very limited number of sources. However, without model data to compare, the accuracy of the TAF loses its meaning. This is because in absence of human forecasters, supported missions would not have zero weather data, but rather they would have "model" data which is commercially available from a variety of civilian sources. The utility of human TAFs is the amount of exposure reduced in comparison to the model.

This document recommends that AFW invests in building a data repository containing every TAF issued for every unclassified location and the accompanying model TAFs for that location and time. This would include the model of choice and other models. Building this repository would allow AFW to retroactively conduct analysis on any weather phenomenon.

Beyond this, collection of data would allow for year to year variability of any weather sensitivity, and ability to capture that phenomenon. With time, data collection could enable machine learning for various parameters and support computer calculations on the most efficient and effective location for mission sets long-term. This could have application for long-term mission planning or even acquisitions purposes.

TAF Metrics

Summary

Understanding customer needs, mission's sensitivity to weather, and how human efforts improve utility rely on accurate metrics. Based upon the research conducted, it is plausible to identify the mission specific utility of TAFs and WWAs. That said, data collection is necessary. Maintaining accurate metrics is helpful, but without the context of the next best source of data, it is impossible to determine the true value of a service. As a proof of concept, this method is successful in proving that AFW can quantify its impact. An accurate estimate of this number will have to begin with data collection.

Appendix

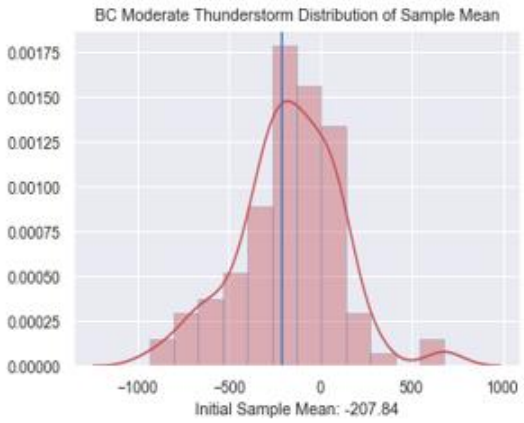
Data Key

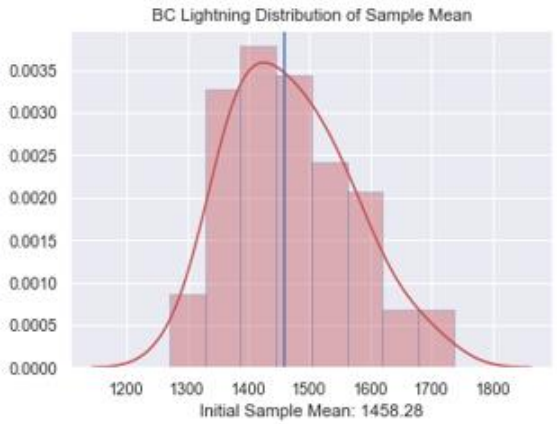
A 0)location
B 1)month[x]
C 2)date[x]
D 3)str(tafhr)
E 4)hour[x]
F 5)max_wind_speed[x]
G 6)max_wind_dir[x]
H 7)ceiling_height[x]
I 8)lightning[x]
J 9)visibility[x]
K 10)wind_speed[x]
L 11)wind_dir[x]
M 12)gust_speed[x]
N 13)gust_dir[x]]
O 14)fcst_wind_spd_capped[x]
P 15)obs_wind_spd_capped[x]
Q 16)wind_spd_values_dif[x]
R 17)fcst_wind_spd_capped_binned[x]
S 18)obs_wind_spd_capped_binned[x]
T 19)wind_spd_values_binned_dif[x]
U 20)wind_spd_event_fcst[x]
V 21)wind_spd_event_obs[x]
W 22)wind_spd_event_capped_dif_given_fcst[x]
X 23)wind_spd_event_capped_dif_given_obs[x]
Y 24)wind_spd_event_capped_binned_dif_given_fcst[x]
Z 25)wind_spd_event_capped_binned_dif_given_obs[x]
AA 26)wind_spd_event_identified_not_criteria_specific[x]
AB 27)wind_spd_missed[x]
AC 28)wind_spd_false_alarm[x]
AD 29)fcst_wind_dir_capped[x]
AE 30)obs_wind_dir_capped[x]
AF 31)wind_dir_values_dif[x]
AG 32)wind_dir_values_binned_dif[x]
AH 33)fcst_vis_capped[x]
AI 34)obs_vis_capped[x]
AJ 35)vis_values_dif[x]
AK 36)fcst_vis_capped_binned[x]
AL 37)obs_vis_capped_binned[x]
AM 38)vis_values_binned_dif[x]
AN 39)vis_event_fcst[x]
AO 40)vis_event_obs[x]
AP 41)vis_event_capped_dif_given_fcst[x]
AQ 42)vis_event_capped_dif_given_obs[x]
AR 43)vis_event_capped_binned_dif_given_fcst[x]
AS 44)vis_event_capped_binned_dif_given_obs[x]
AT 45)vis_event_identified_not_criteria_specific[x]
AU 46)vis_missed[x]
AV 47)vis_false_alarm[x]
AW 48)fcst_ceiling_capped[x]
AX 49)obs_ceiling_capped[x]
AY 50)ceiling_values_dif[x]
AZ 51)fcst_ceiling_capped_binned[x]
BA 52)obs_ceiling_capped_binned[x]
BB 53)ceiling_values_binned_dif[x]
BC 54)ceiling_event_fcst[x]
BD 55)ceiling_event_obs[x]
BE 56)ceiling_event_capped_dif_given_fcst[x]
BF 57)ceiling_event_capped_dif_given_obs[x]
BG 58)ceiling_event_capped_binned_dif_given_fcst[x]
BH 59)ceiling_event_capped_binned_dif_given_obs[x]
BI 60)ceiling_event_identified_not_criteria_specific[x]

Wind Speed	NATO Ceiling	NATO Visibility	Wind Direction	Categories
15	2500	5	30	0
20	1500	3	45	1
25	700	2	60	2
35	300	1	90	3
45	200	0.5	120	4
55	<200	<.5	150	5

BJ 61)ceiling_missed[x]
BK 62)ceiling_false_alarm[x]
BL 63)lightning_dif[x]
BM 64)lightning_given_fcst[x]
BN 65)lightning_given_obs[x]
BO 66)correct_month_day_and_hour[x]
BP 67)original_correct_month_day_and_hour[x]

Data Percentage of Occurrence





		Percentage occurrence				
auto windspeed	0.7843	0.062661	0.007374	0.000511	5.26E-05	9.01E-05
	0.058772	0.044332	0.015911	0.000924	0.000105	3E-05
	0.008838	0.006337	0.005174	0.001314	3E-05	0
	0.001667	0.000541	0.000338	0.000218	7.51E-06	0
	0.000188	0.00012	3E-05	7.51E-06	7.51E-06	0
	7.51E-05	2.25E-05	1.5E-05	0	0	7.51E-06
human windspeed	0.732402	0.092972	0.025594	0.003163	0.000441	3.23E-05
	0.051198	0.039741	0.02668	0.00284	0.000398	2.15E-05
	0.006745	0.004712	0.007316	0.002033	0.000366	0
	0.001119	0.000538	0.000656	0.000334	0.00014	0
	0.000118	0.000183	6.45E-05	1.08E-05	3.23E-05	0
	5.38E-05	6.45E-05	3.23E-05	0	0	0
auto cig	0.715706	0.077706	0.028477	0.005831	0.000732	0.001474
	0.01731	0.014728	0.009489	0.001668	0.00014	0.000108
	0.016589	0.017321	0.020634	0.008004	0.00029	0.000247
	0.008069	0.00866	0.013953	0.012146	0.00114	0.000732
	0.001711	0.001356	0.002915	0.004465	0.001033	0.000269
	0.002345	0.000796	0.00128	0.001635	0.000538	0.000506
human cig	0.722037	0.055093	0.024486	0.011323	0.002823	0.003447
	0.01233	0.014695	0.011631	0.002883	0.000796	0.001457
	0.007471	0.011226	0.019876	0.013689	0.003972	0.009161
	0.003717	0.005076	0.008763	0.018374	0.002891	0.010062
	0.001832	0.001472	0.001374	0.005279	0.000826	0.003439
	0.003199	0.000563	0.000721	0.001892	0.000976	0.001149

Cost Function

			Cost				Hours in scenario	
auto windspeed	0	-1	-100	-500	-8000	-10000	hours ->	1440
	-100	100	-50	-400	-5000	-8000		
	-1000	-500	1000	-500	-2500	-5000		
	-5000	-5000	1500	4000	-1000	-2500		
	-10000	-10000	2000	4000	10000	8000		
	-1000000	-1000000	200000	400000	600000	800000		
human windspeed	0	-1	-100	-500	-8000	-10000		
	-100	100	-50	-400	-5000	-8000		
	-1000	-500	1000	-500	-2500	-5000		
	-5000	-5000	1500	4000	-1000	-2500		
	-10000	-10000	2000	4000	10000	8000		
	-1000000	-1000000	200000	400000	600000	800000		
auto cig	0	-0.01333	-1.33333	-6.66667	-106.667	-133.333		
	-1.33333	1.333333	-0.66667	-5.33333	-66.6667	-106.667		
	-13.3333	-6.66667	13.33333	-6.66667	-33.3333	-66.6667		
	-66.6667	-66.6667	20	53.33333	-13.3333	-33.3333		
	-133.333	-133.333	26.66667	53.33333	133.3333	106.6667		
	-13333.3	-13333.3	2666.667	5333.333	8000	10666.67		
human cig	0	-0.01333	-1.33333	-6.66667	-106.667	-133.333		
	-1.33333	1.333333	-0.66667	-5.33333	-66.6667	-106.667		
	-13.3333	-6.66667	13.33333	-6.66667	-33.3333	-66.6667		
	-66.6667	-66.6667	20	53.33333	-13.3333	-33.3333		
	-133.333	-133.333	26.66667	53.33333	133.3333	106.6667		
	-13333.3	-13333.3	2666.667	5333.333	8000	10666.67		

Scenario Statistics				
Fcst Exposure	Control exposure	**Upper Scenario**	**Lower Scenario**	
-164883.7446	-246564.4286	**-89280**	**-249120**	
BC Hourly Exposure	Hourly Control	upper hourly	lower hourly	
-114.5026004	-171.2252976	-62	-173	
HVA	Hourly Value Added	upper VA	lower va	Value Added %
-81680.68401	-56.72269723	63.7903973	-1.036472054	33.12752147
Fcst Exposure	**Control exposure**	**Upper Scenario**	**Lower Scenario**	
-198781.5377	**-290707.0023**	**-105120**	**-303840**	
Hourly Exposure	**Hourly Control**	**upper hourly**	**lower hourly**	
-138.0427345	**-201.8798627**	**-73**	**-211**	
HVA	**Hourly Value Added**	upper VA	lower va	VA %
-91925.46458	**-63.83712818**	63.8398803	-4.517606253	31.62134516
Fcst Exposure	Control exposure	Upper Scenario	Lower Scenario	
-29866.55666	-144168.4051	-20160	-36000	
Hourly Exposure	Hourly Control	upper hourly	lower hourly	
-20.74066435	-100.116948	-14	-25	
HVA	Hourly Value Added	upper VA	lower va	VA %
-114301.8484	-79.37628364	86.01635359	75.02920285	79.28356311
Fcst Exposure	Control exposure	Upper Scenario	Lower Scenario	
-27897.78151	-171963.5538	-21600	-37440	
Hourly Exposure	Hourly Control	upper hourly	lower hourly	
-19.37345938	-119.4191346	-15	-26	
HVA	Hourly VA	upper VA	lower va	VA %
-144065.7723	-100.0456752	87.43919887	78.22794472	83.77692197

Presentation Ready Winds

Col Percent

Auto	FCST_0	FCST_1	FCST_2	FCST_3	FCST_4	FCST_5
OBS_0	91.85567	54.95917	25.56626	17.17172	25.92593	70.58824
OBS_1	6.883239	38.88303	55.16793	31.06061	51.85185	23.52941
OBS_2	1.03508	5.558483	17.93804	44.19192	14.81481	0
OBS_3	0.195232	0.474183	1.17157	7.323232	3.703704	0

OBS_4	0.021986	0.105374	0.10414	0.252525	3.703704	0
OBS_5	0.008794	0.019758	0.05207	0	0	5.882353
All Locations	Bayes Auto Wind	Cat 1 n Up	14.616	14.50111	52.04536	51.63627
All Locations	Bayes Auto Wind	Cat 2 n Up	3.214543	2.493693	28.66606	22.23779

Forecaster

Events	FCST_0	FCST_1	FCST_2	FCST_3	FCST_4	FCST_5
OBS_0	92.5175	67.26862	42.41398	37.74069	32.03125	60
OBS_1	6.467351	28.75379	44.21466	33.8896	28.90625	40
OBS_2	0.852076	3.409356	12.12337	24.26187	26.5625	0
OBS_3	0.141333	0.389196	1.087538	3.979461	10.15625	0
OBS_4	0.014949	0.132327	0.106971	0.12837	2.34375	0
OBS_5	0.006795	0.046704	0.053485	0	0	0
All Locations	Bayes Human Wind	Cat 1 n Up	20.83634	14.53961	59.26008	41.3517
All Locations	Bayes Human Wind	Cat 2 n Up	7.015373	2.451777	44.80035	15.6571

Ceiling percent occurrences all possibilities

Percent by occurrence Cig Auto

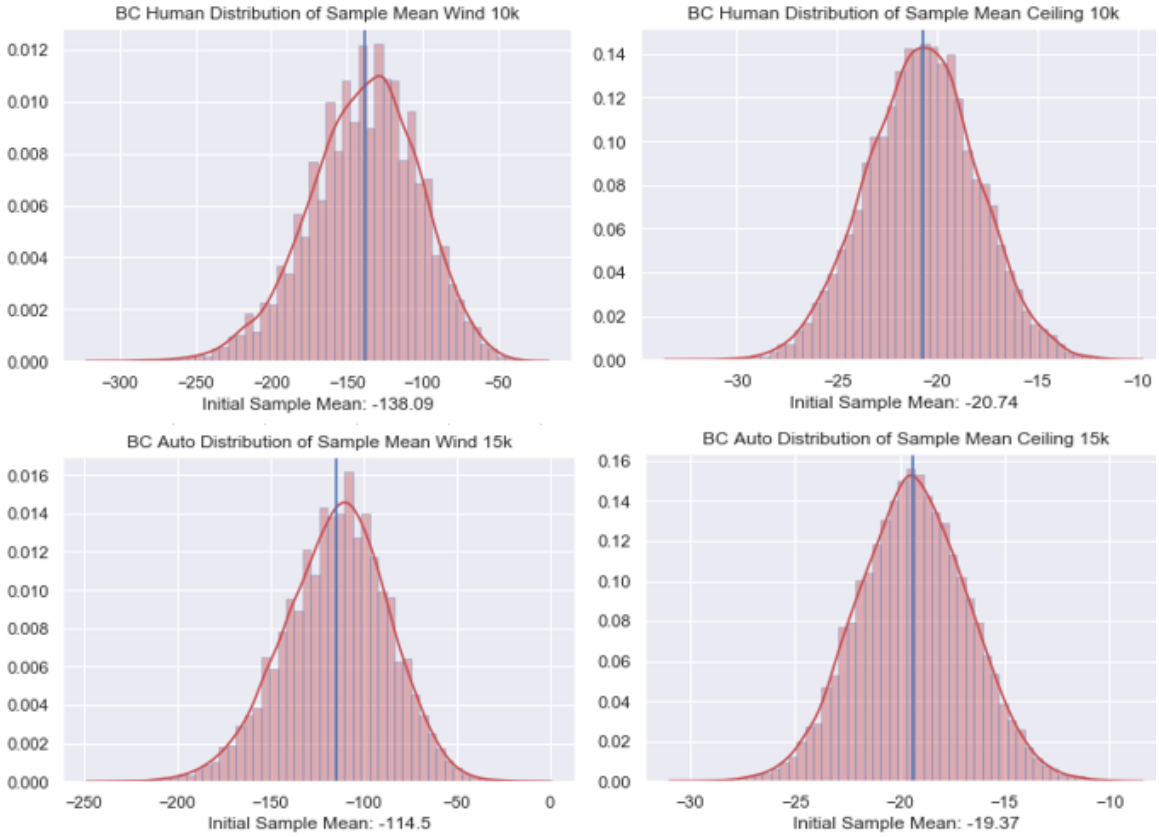
Events	FCST_0	FCST_1	FCST_2	FCST_3	FCST_4	FCST_5
OBS_0	0.722037	0.055093	0.024486	0.011323	0.002823	0.003447
OBS_1	0.01233	0.014695	0.011631	0.002883	0.000796	0.001457
OBS_2	0.007471	0.011226	0.019876	0.013689	0.003972	0.009161
OBS_3	0.003717	0.005076	0.008763	0.018374	0.002891	0.010062
OBS_4	0.001832	0.001472	0.001374	0.005279	0.000826	0.003439
OBS_5	0.003199	0.000563	0.000721	0.001892	0.000976	0.001149

Percent by occurrence auto Cig human

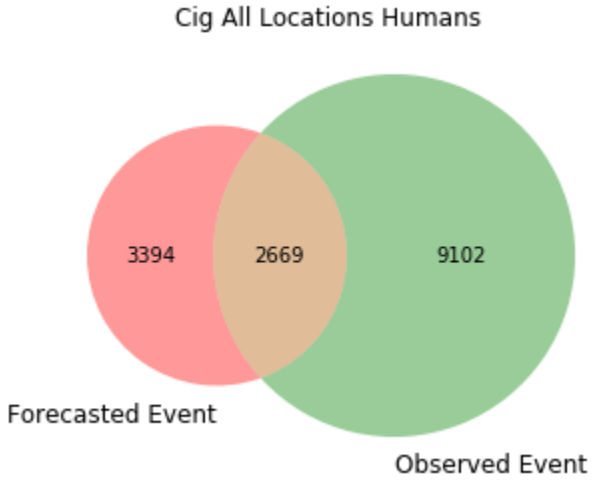
Percent by occurrence Cig
Human

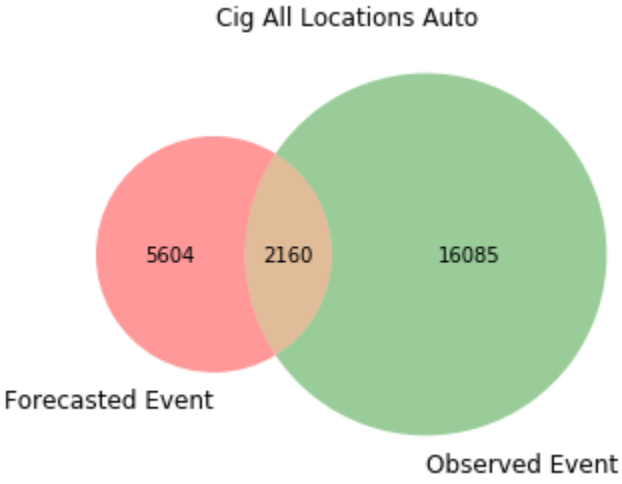
Events	FCST_0	FCST_1	FCST_2	FCST_3	FCST_4	FCST_5
OBS_0	0.715706	0.077706	0.028477	0.005831	0.000732	0.001474
OBS_1	0.01731	0.014728	0.009489	0.001668	0.00014	0.000108
OBS_2	0.016589	0.017321	0.020634	0.008004	0.00029	0.000247
OBS_3	0.008069	0.00866	0.013953	0.012146	0.00114	0.000732
OBS_4	0.001711	0.001356	0.002915	0.004465	0.001033	0.000269
OBS_5	0.002345	0.000796	0.00128	0.001635	0.000538	0.000506

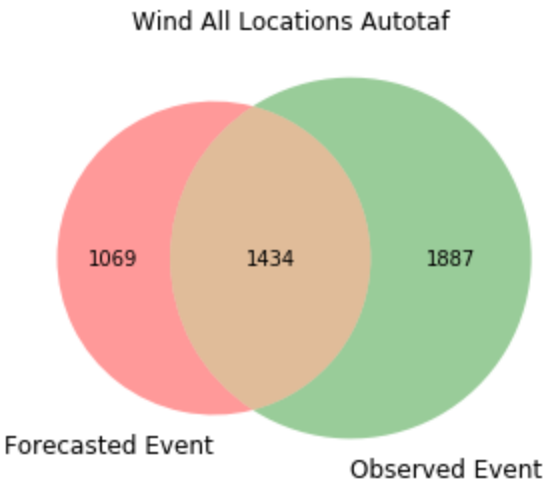
Cost Sheet

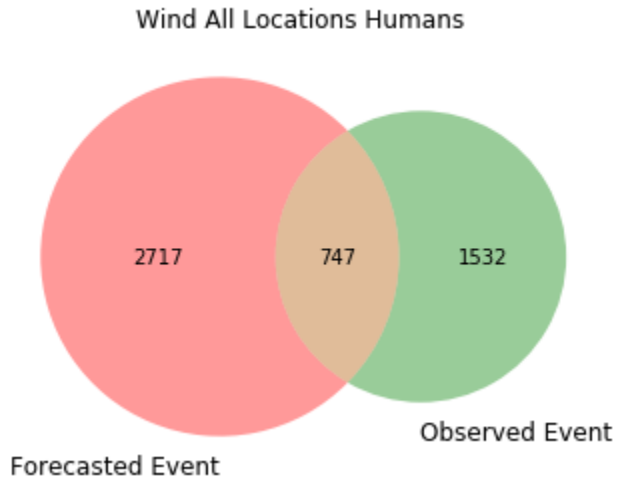


2nup venn diagrams






Wind All Locations Autotaf
1069
1434
1887
Forecasted Event
Observed Event


Wind All Locations Humans
2717
747
1532
Observed Event
Forecasted Event

Conditionals

runfile('C:/Users/Bailey/Desktop/AFIT/Thesis/python/Final Code/Images 4 Dec/bayes charts/ceiling Corrected Bayes Stats with Error Bars v2.py', wdir='C:/Users/Bailey/Desktop/AFIT/Thesis/python/Final Code/Images 4 Dec/bayes charts')

21 a)max 33.36925287356322 min 8.072392990519965 7.6737000986295705

21 b)max 29.365079365079367 min 6.463659867854064 6.690346389406787

ceiling a|b error bounds upper 75.82697201017811 lower 64.43037974683544

ceiling b|a error bounds upper 64.87663280116111 lower 41.63385826771653

21 a) max 17.226050233385195 min 5.976277372262774 3.303248645293075

21 b) max 18.715270060013335 min 6.158759124087592 3.620741803695734

human 2nup ceiling a|b error bounds upper 60.679611650485434 lower 45.77922077922078

human 2nup ceiling b|a error bounds upper 67.89772727272727 lower 49.32614555256065

Human overall prob a 23.827095413811282 prob b 17.0075199294267 prob a|b 72.93946486178758 prob b|a 52.063391728372764

Human overall 2nup prob a 11.770464643421946 prob b 12.6633890245608 prob a|b 55.11001614136437 prob b|a 59.290741248514756

21 a) max 41.107277289836894 min 11.779607346421786 6.938155658590852

21 b) max 31.02258469259724 min 7.267257758074732 5.896316130892324

ceiling a|b error bounds upper 90.7673860911271 lower 79.58579881656804

ceiling b|a error bounds upper 72.22222222222221 lower 52.183650615901456

21 max a) 24.613930034667508 min 6.6022799240025325 4.379189717229185

21 max b) 26.176286072772896 min 5.113996200126662 5.453353546068943

auto 2nup ceiling a|b error bounds upper 86.91141260973663 lower 71.107544141252

auto 2nup ceiling b|a error bounds upper 73.94578313253012 lower 52.27502527805864

Auto overall prob a 24.94143088844837 prob b 18.07908331831561 prob a|b 84.20899613739253 prob b|a 61.039860308285164

Auto overall 2nup prob a 16.129032258064516 prob b 13.6999159007629 prob a|b 74.77665113729789 prob b|a 63.514897579143394

runfile('C:/Users/Bailey/Desktop/AFIT/Thesis/python/Final Code/Images 4 Dec/bayes charts/Wind speed Corrected Bayes Stats with Error Bars.py', wdir='C:/Users/Bailey/Desktop/AFIT/Thesis/python/Final Code/Images 4 Dec/bayes charts')

21 max a) 33.6397586900316 min 15.221579961464354 4.432243773700006

21 max b) 25.165182418845163 min 9.364161849710982 4.199795383026722

wind a|b error bounds upper 65.39473684210526 lower 52.12201591511937

wind b|a error bounds upper 52.861952861952865 lower 34.43850267379679

21 a) max 11.885245901639344 min 4.286365709074327 1.8529866307855951

21 b) max 3.996447602131439 min 1.44476550344521 0.6398653205425291

human 2nup wind a|b error bounds upper 57.01754385964912 lower 31.818181818181817

human 2nup wind b|a error bounds upper 20.50473186119874 lower 9.25925925925926

Human overall prob a 20.836336643249815 prob b 14.53960603745979 prob a|b 59.26008139104698 prob b|a 41.3517141676993

Human overall 2nup prob a 7.015373360730692 prob b 2.451776704355965 prob a|b 44.800351031154015 prob b|a 15.657107805551295

21 a) max 20.595650803655847 min 2.8858218318695106 3.822602337188161

21 b) max 24.445851804939835 min 9.738417901040025 4.065314873382713

wind a|b error bounds upper 62.67409470752089 lower 43.529411764705884

wind b|a error bounds upper 65.82568807339449 lower 38.4083044982699

21 a) max 4.821935077213993 min 0.4705144291091593 1.1481195973591316

21 b) max 7.873274780426599 min 0.5042546485975418 1.72658429939725

auto 2nup wind a|b error bounds upper 44.53125 lower 15.909090909090908

auto 2nup wind b|a error bounds upper 38.388625592417064 lower 6.5743944636678195

Auto overall prob a 14.61599687631405 prob b 14.501111311347389 prob a|b 52.0453603976802 prob b|a 51.636270228615466

Auto overall 2nup prob a 3.2145431609298973 prob b 2.493692557217517 prob a|b 28.66606443842216 prob b|a 22.23779490773184

Forecaster	Phenomenon	Category	Probability (A|B)	Uncertainty (A|B) (71.4%)		Probability (B|A)	Uncertainty (B|A)
Human	Ceiling	1 and higher	73	Upper: 76 Lower:64		52	Upper: 65 Lower:42
Human	Ceiling	2 or higher	18	Upper: 24 Lower:11		19	Upper: 26 Lower:15
Auto	Ceiling	1 and higher	84	Upper: 91 80	Lower:	61	Upper: 72 Lower:52
Auto	Ceiling	2 or higher	75	Upper: 87 71	Lower:	64	Upper: 74 Lower:53

Human	Wind Speed	1 and higher	59	Upper: 65 52	Lower:	41	Upper: 53 Lower:34
Human	Wind Speed	2 or higher	45	Upper: 57 31	Lower:	16	Upper: 21 Lower: 9
Auto	Wind Speed	1 and higher	52	Upper: 63 44	Lower:	51	Upper: 66 Lower: 38
Auto	Wind Speed	2 or higher	29	Upper: 44 16	Lower:	22	Upper: 38 Lower: 7

TAF Parser

```
import numpy as np

##Need to recreate
#lineparser function
#pop out TEMPO lines
#section to get firstlines,numlines
#section to get linestart/enddatetime (look for FM,BE or first line datetime group then parse out
date time)
#section to build the npy files
def lineparser(inline):
   wdir,wspd,gust,vis,alst,lightning,cig,possible_cig,possible_startmonth, lightninglist=-9999,-
9999,-9999,-9999,-9999,-9999,-9999,35000,[35000],[4],[0]
   for p in range(len(inline)):
      if inline[p][-2:]=='KT' and inline[p][0:2]!='WS':

         if inline[p][0:3]=='VRB':
            wdir=0
         else:
            wdir=int(inline[p][0:3])
         if len(inline[p])>8:
            gust=int(inline[p].split('G')[1].rstrip('KT'))
            wspd=int(inline[p].split('G')[0][3:])
         else:
            wspd=int(inline[p].rstrip('KT')[3:])
      if inline[p][:2]=='FG':
         possible_cig.append(0)
      if inline[p][:2]=='VV':
            possible_cig.append(int(inline[p][2:])*100)
      if inline[p][:3]=='BKN':
         possible_cig.append(int(inline[p][3:6])*100)
      if inline[p][:3]=='OVC' and len(inline[p])>=4:
         possible_cig.append(int(inline[p][3:6])*100)
      cig=min(possible_cig)
      #print(cig)
```

```
        if inline[p][-2:]=='SM':
            visline=inline[p].rstrip('SM')
            if visline[0]=='P':
                visline=visline.lstrip('P')
            if len(visline)>2 and visline[1]=='/':
                vis=(1.*(int(visline.split('/')[0])))/int(visline.split('/')[1])
            else:
                vis=int(visline)

        if inline[p][0:3]=='QNH':
            alst=int(inline[p][3:7])

        if 'TS' in inline[p] or inline[p][-2:]=='CB' or inline[p][0:3]=='+TS'  or inline[p][0:3]=='-TS' or
inline[p][0:2]=='TS':
            lightninglist.append(1)
        else:
            lightninglist.append(0)
        lightning= max(lightninglist)

        #test for vis for miliitary tafs
        if inline[p].isdigit()==True:
            if float(inline[p])<10000:
                vis=int(inline[p])

        # startmonth
        #monthcounter=0

        if  len(inline[p])>10 and inline[p][0:2]!='WS' and inline[p][0:2]!='TN' and
inline[p][0:2]!='TX447':

            if type(int(inline[p]))==int:
                #monthlist=[int(inline[p][4:6])]
                startmo=int(inline[p][4:6])
                #print(startmonth)
                possible_startmonth.append(startmo)
                print(possible_startmonth)
                #monthlist.pop(0)
                #print(monthlist)
                startmonth=max(possible_startmonth)


    startmonth=max(possible_startmonth)
        #print(monthlist)
    #print('myguy',startmonth)
```

```
   return wdir,wspd,gust,vis,alst,cig,lightning,startmonth

def lastlineparser(inline):
   tmax,tmin=-9999,-9999
   for p in range(len(inline)):
      if inline[p][0:2]=='TX':
         t=inline[p].split('/')[0]
         if t[2]=='M':
            tmax=-int(t.split('M')[1])
         else:
            tmax=int(t[2:])
      if inline[p][0:2]=='TN':
         t=inline[p].split('/')[0]
         if t[2]=='M':
            tmin=-int(t.split('M')[1])
         else:
            tmin=int(t[2:])
   return tmax,tmin



#open text file
f=open('BGTL_6.txt')
#read all the lines from the text file into a list
g=f.readlines()
alllines=[]
nextline=[]
location=str(f.name)[:4]
startmonthpick=str(f.name)[5]

#run through all lines of the list 'g' which has all lines of the taf
for x in range(len(g)):
   #get rid of all whitespace and carriage returns then space delimit
   a = g[x].strip().split(' ')


   for xx in range(len(nextline)-1,-1,-1):
      a.insert(0,nextline[xx])

   nextline=[]
   #run through all elements of each line...if an embedded change group is found it copies it to
the 'nextline' list
   #then attaches it to the start of the next line on the next 'x' iteration through the outside loop
   xxcnt=0
   ifembedded=0
```

```
   for xx in range(len(a)):
      if (a[xx][0:2]=='TE' or a[xx][0:2]=='BE' or a[xx][0:2]=='FM' or (len(a[xx])>11 and
a[xx][0:2]=='20')) and xxcnt!=0:
         ifembedded=1
         #print xxcnt
         nextline=[]
         for xxx in range(xxcnt,len(a)):
            nextline.append(a[xxx])
         for xxx in range(len(a)-1,xxcnt-1,-1):
            a.pop(xxx)
         #break out of the 'xx' loop
         break

         #print nextline
      xxcnt=xxcnt+1
   #run through all elements of each line...if an embedded change group is found it copies it to
the 'nextline' list
   #then attaches it to the start of the next line on the next 'x' iteration through the outside loop
   prevline=[]
   if ifembedded==0:
      if (a[0][0:2]=='TE' or a[0][0:2]=='BE' or a[0][0:2]=='FM' or (len(a[0])>11 and a[0][0:2]=='20')):
         pass
      else:
         for xx in range(len(a)):
            alllines[x-1].append(a[xx])
         #print(alllines[x-1])
   #print(a)
   alllines.append(a)
   #print(a)
alllinestempo=alllines.copy()

#pop out TEMPO lines or non valid change lines
for xx in range(len(alllines)-1,-1,-1):
   if (alllines[xx][0][0:2]=='BE' or alllines[xx][0][0:2]=='FM' or (len(alllines[xx][0])>11 and
alllines[xx][0][0:2]=='20')):
   #if (alllines[xx][0][0:2]=='TE' or (len(alllines[xx][0])>11 and alllines[xx][0][0:2]=='20')):
      pass
   else:
      alllines.pop(xx)

#pop out all lines but first lines and TEMPOS
for xx in range(len(alllinestempo)-1,-1,-1):
   if (alllinestempo[xx][0][0:2]=='TE' or (len(alllinestempo[xx][0])>11 and
alllinestempo[xx][0][0:2]=='20')):
   #if (alllines[xx][0][0:2]=='TE' or (len(alllines[xx][0])>11 and alllines[xx][0][0:2]=='20')):
      pass
```

```
   else:
      alllinestempo.pop(xx)
#print alllines

flines,nhours,nlines=[],[],[]
for x in range(len(alllines)):
   if len(alllines[x][0])>10 and alllines[0][0][0:2]=='20': #then we know we are in the first line of a
TAF
      validtaf=0
      #startmonth=int(alllines[x][0][4:6])
      for xx in range(len(alllines[x])):
         if len(alllines[x][xx])==9: #find the taf start/end date time group (i.e. 2600/2624)
            dt=alllines[x][xx]
            dt=dt.split('/')
            #determine start date/times
            startdate=int(dt[0][0:2])
            starthour=int(dt[0][2:4])
            enddate=int(dt[1][0:2])
            endhour=int(dt[1][2:4])
            #the if statement below determines if the taf is valid (not an amendment) and if it is 24
or 30 hrs
            if ((enddate-startdate==0) and (endhour-starthour==24)) or \
              ((enddate-startdate==1) and (endhour-starthour==0)):
               nhours.append(24)
               flines.append(x)
               validtaf=1
            elif ((enddate-startdate==1) and (endhour-starthour==6)) or \
                ((enddate-startdate==2) and ((endhour+24)-starthour==6)):
               nhours.append(30)
               flines.append(x)
               validtaf=1
      if validtaf==1:  #this section determines the number of lines in the taf and stores in nlines
list
         linelength=1
         if x==(len(alllines)-1):#account for a 1-line TAF as the last line of the file will break the
script
            nlines.append(linelength)
         else:
            for xx in range(x+1,len(alllines)):
               if len(alllines[xx][0])>10:
                  nlines.append(linelength)
                  break
               elif xx==(len(alllines)-1):
                  nlines.append(linelength)
                  break
               else:
```

```
                linelength=linelength+1
xcnt=0
tcnt=0
for x in range(len(flines)):
   #creating a startdt,enddt arrays of size [nlines,2] for the start and end time dates
   startdt=np.zeros((nlines[x],2))
   enddt=np.zeros((nlines[x],2))
   for xx in range(len(alllines[flines[x]])):
      if len(alllines[flines[x]][xx])==9:
         dt=alllines[flines[x]][xx]
         dt=dt.split('/')
         startdate=int(dt[0][0:2])
         starthour=int(dt[0][2:4])
         enddate=int(dt[1][0:2])
         endhour=int(dt[1][2:4])
         startdt[0,0]=startdate
         startdt[0,1]=starthour
         #startmonth=int(dt[0])
         enddt[-1,0]=enddate
         enddt[-1,1]=endhour
   xxcnt=1
   for xx in range(flines[x]+1,flines[x]+nlines[x]):
      if alllines[xx][0][0:2]=='FM':
         startdt[xxcnt,0]=int(alllines[xx][0][2:4])
         startdt[xxcnt,1]=int(alllines[xx][0][4:6])
         enddt[xxcnt-1,0]=int(alllines[xx][0][2:4])
         enddt[xxcnt-1,1]=int(alllines[xx][0][4:6])
      if alllines[xx][0][0:2]=='BE':
         startdt[xxcnt,0]=int(alllines[xx][1][5:7])
         startdt[xxcnt,1]=int(alllines[xx][1][7:9])
         enddt[xxcnt-1,0]=int(alllines[xx][1][5:7])
         enddt[xxcnt-1,1]=int(alllines[xx][1][7:9])
      xxcnt=xxcnt+1
   #print (flines[x],nlines[x],startdt)
#   if xcnt>0:
#      np.save(str(x)+'.npy',outf)
#      np.savetxt(str(x)+'.csv',outf,delimiter=',')
#      outf[:,2:]=-9999
   outf=np.zeros((nhours[x],12), dtype=object)
   hrcnt=0

   #goes through each line of the taf and runs the function 'lineparser' (at the top of this
program)
   #then writes output to file 'outf'
   startmonth=4
   for xx in range(nlines[x]):
```

```
      if hrcnt==0:
         m1,m2,m3,m4,m5,m6,m7,startmonth=lineparser(alllines[flines[x]+xx])
      startmonth=startmonth
      print(nlines[0],'testing11111111111111111111',startmonth )
      if startdt[xx,0]>=enddt[xx,0]:
         for xxx in range(int(startdt[xx,1]),int(enddt[xx,1])):

wdir0,wspd0,gust0,vis0,alst0,cig0,lightning0,startmonth0=lineparser(alllines[flines[x]+xx])
            #outf[hrcnt,:]=[startdt[xx,0],startdt[xx,1]+xx,wdir0,wspd0,gust0,vis0,alst0,0,0]
            hrintaf=startdt[0,1]+hrcnt
            dateintaf=startdt[xx,0]
            if hrintaf >23:
               hrintaf=hrintaf-24
               #dateintaf= dateintaf+1
               #if dateintaf >31:
               #    dateintaf=1.1
            print('this is the start month!!!!!!!!!!!!!', startmonth)

outf[hrcnt,:]=[location,startmonthpick,startdate,starthour,dateintaf,hrintaf,wdir0,wspd0,gust0,v
is0,cig0,lightning0]
            #print (wdir0)
            hrcnt=hrcnt+1
      if startdt[xx,0]<enddt[xx,0]:
         for xxx in range(int(startdt[xx,1]),24):

wdir0,wspd0,gust0,vis0,alst0,cig0,lightning0,startmonth0=lineparser(alllines[flines[x]+xx])
            #outf[hrcnt,:]=[startdt[xx,0],startdt[xx,1]+xx,wdir0,wspd0,gust0,vis0,alst0,0,0]
            hrintaf=startdt[0,1]+hrcnt
            dateintaf=startdt[xx,0]
            if hrintaf >23:
               hrintaf=hrintaf-24
               #dateintaf=dateintaf+1
               #if dateintaf >31:
               #    dateintaf=1.1
            print('!!!!!!!!!!!!!', startmonth)

outf[hrcnt,:]=[location,startmonthpick,startdate,starthour,dateintaf,hrintaf,wdir0,wspd0,gust0,v
is0,cig0,lightning0]
            #outf[hrcnt,:]=[startdt[xx,0],hrintaf,wdir0,wspd0,gust0,vis0,alst0,0,0]
            #print (wdir0)
            hrcnt=hrcnt+1

         for xxx in range(0,int(enddt[xx,1])):

wdir0,wspd0,gust0,vis0,alst0,cig0,lightning0,startmonth0=lineparser(alllines[flines[x]+xx])
            #outf[hrcnt,:]=[startdt[xx,0],startdt[xx,1]+xx,wdir0,wspd0,gust0,vis0,alst0,0,0]
```

```
            #hrintaf[x]=startdt[0,1]+hrcnt
            hrintaf=startdt[0,1]+hrcnt
            dateintaf=startdt[xx,0]
            if hrintaf >23:
               hrintaf=hrintaf-24
               dateintaf=dateintaf+1
               if dateintaf >31:
                  dateintaf=1.1

outf[hrcnt,:]=[location,startmonthpick,startdate,starthour,dateintaf,hrintaf,wdir0,wspd0,gust0,v
is0,cig0,lightning0]
            #print (wdir0)
            hrcnt=hrcnt+1
#       if xx==(nlines[x]-1):
#          #additional function to get temperatures from military tafs and puts them into last
columns of output
#          tmax0,tmin0=lastlineparser(alllines[flines[x]+xx])
#          outf[0:nhours[x],7]=tmax0
#          outf[0:nhours[x],8]=tmin0
   np.savetxt(str(x)+'.csv',outf,fmt='%s',delimiter=',')
   #blank out all values in the output array except for date/time
   #outf[:,2:]=-9999

   outf[:,6:]=-9999
   #print(outf.shape)

   for xtmpo in range(len(alllinestempo)):
      if alllinestempo[xtmpo][0]==alllines[flines[x]][0]:
         #print('IN')
         for t in range(min(len(alllinestempo)-1,xtmpo+1),len(alllinestempo)):
            #print(alllinestempo[t][0][0:2])
            if alllinestempo[t][0][0:2]=='TE': #if there is a tempo group(s) in the TAF
               #get start date/times from the line
               tstartdate=int(alllinestempo[t][1][0:2])
               tenddate=int(alllinestempo[t][1][5:7])
               tstarthour=int(alllinestempo[t][1][2:4])
               tendhour=int(alllinestempo[t][1][7:9])
               tstarthrintempo=int(alllinestempo[t][1][2:4])
               tstartdateintempo=int(alllinestempo[t][1][0:2])
               #print(tstartdate,tenddate,tstarthour,tendhour)
               #print(tstartdate,tstarthour,tenddate,tendhour)
#                yo= outf[:,1]
#                print(yo)
               #if the tempo group ends at hour 24 this pushes the end date/hour to date+1 and
hour=0
```

```
            if tendhour==24:
               tendhour,tenddate=0,tenddate+1

            #extract elements from TEMPO line
            wdir0,wspd0,gust0,vis0,alst0,cig0,lightning0,startmonth=lineparser(alllinestempo[t])
            spos=np.where((outf[:,4]==tstartdate) & (outf[:,5]==tstarthour))[0][0]
            epos0=np.where((outf[:,4]==tenddate) & (outf[:,5]==tendhour))

            #this conditional catches tempo groups that go after the last hour in the .csv file
            #for example, the last hour in the file is 24 but the tempo group is 2619/2701
            #it makes sure the end at the end of the file
            if len(epos0[0])==0:
               epos=len(outf[:,0])
            else:
               epos=epos0[0][0]

            #write in values from the spos line to the epos line of the .csv file
            for tt in range(spos,epos):

outf[tt,:]=[location,outf[tt,1],startdate,starthour,outf[tt,4],outf[tt,5],wdir0,wspd0,gust0,vis0,cig0
,lightning0]
               #print('do the tempo thing')
            else:
               #print('IM TO THE NEXT LINE')

               break
         tcnt=tcnt+1

   np.savetxt(str(x)+'tempo.csv',outf,fmt='%s',delimiter=',')

   #from here you will need to re-open XXtempo.csv and XX.csv with np.genfromtxt function

   #open an output array

   #then go line by line keeping what you want from the predominant and tempo csv, write to
your
   #new output array and save

   #print(alllines[flines[x]][0])
   xcnt=xcnt+1
print(tcnt)
```

```
            #else:
               #print 'Bad',dt

##
##      #check the first element of each line and see if it is a number
##      if xxcnt==0 and len(xx)>11:
##         #first we split the entire line by '/' which will give us two parts
##         flinesplit= g[x].strip().split('/')
##         #grab the last 4 characters of the left split
##         startdatetime=flinesplit[0][-4:]
##         #first 4 characters of hte right split
##         enddatetime=flinesplit[1][:4]
##         print startdatetime,enddatetime
##         fout=open(enddatetime+'.txt','w')
##         xxcnt=xxcnt+1
##
##      #if the first element is not a number it will default to the operation below, doing nothing
##
##      #look for an element of the line with the first two characters 'FM' to find the hidden
FROM groups
##      if xx[0:2]=='FM':
##
##         print xxcnt
##      xxcnt=xxcnt+1
##      #xxcnt=xxcnt+1
##      #continue
```

Wind speed Conditional

```
# -*- coding: utf-8 -*-

"""
Created on Sun Dec 22 14:09:04 2019
Windspeed Conditionals
```

```
@author: Bailey
"""

################################################## importing necessary libraries
import os
import numpy as np
import datetime
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
# importing the statistics module
from statistics import stdev

# importing frations as parameter values
from fractions import Fraction as fr
#import plotly.graph_objects as go
datestart=datetime.datetime.now()
#%%
################################################## defining directories and opening
files
directory = os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final Code\Images 4
Dec')
autotaf_lines=[]
humantaf_lines=[]
autotaf_lines = open('all_compaired_auto_0_4dec.csv').readlines()
humantaf_lines = open('all compaired_human_0_4dec.csv').readlines()
#import numpy as np
#def find_nearest(array, value):
#    array = np.asarray(array)
#    idx = (np.abs(array - value)).argmin()
#    return array[idx], idx
import numpy as np
def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return idx
################################################## location information in file name
and within file
supercounter=0
stations=['BGTL','KADW','KBLV','KCEF','KDAA','KDOV','KFAF','KFFO','KFTK','KGTB','KGUS','KHOP','
KLFI','KMIB','KMTC','KMUI','KOFF','KRCA','KRDR','KVOK','KWRI']
times_hours_zulu=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]
```

```
all_autotaf_lines_as_lists=[]
all_humantaf_lines_as_lists=[]
################################################## get rid of all whitespace and
carriage returns then space delimit
for x in range(len(autotaf_lines)):
   autotaf_lines_as_lists = autotaf_lines[x].strip().split(',')
   all_autotaf_lines_as_lists.append(autotaf_lines_as_lists)
for x in range(len(humantaf_lines)):
   humantaf_lines_as_lists = humantaf_lines[x].strip().split(',')
   all_humantaf_lines_as_lists.append(humantaf_lines_as_lists)
##################################################

#%%
      ##################################################
####### Wind speed conditionals
##################################################
#  prob(A) prob fcst =
count(all_humantaf_lines_as_lists[20]==1)/len(all_humantaf_lines_as_lists)
#  prob(B) prob obs =
count(all_humantaf_lines_as_lists[21]==1)/len(all_humantaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/ len(all_humantaf_lines_as_lists[21])
#  prob(B|A) prob observed given fcst = count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/ len(all_humantaf_lines_as_lists[20])
#
#
# By Station 1 and up human
#
wind_prob_a_list=[]
wind_prob_b_list=[]
wind_prob_a_giv_b_list=[]
wind_prob_b_giv_a_list=[]
wind_prob_a_list_2nup=[]
wind_prob_b_list_2nup=[]
wind_prob_a_giv_b_list_2nup=[]
wind_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_humantaf_lines_as_lists)):
      if str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:
         eventcount=eventcount+1
```

```
        #print(all_humantaf_lines_as_lists[prob])
        #print(all_humantaf_lines_as_lists[prob][1])
        if float(all_humantaf_lines_as_lists[prob][20])==float(1):
          count_prob_a=count_prob_a+1
        if float(all_humantaf_lines_as_lists[prob][21])==float(1):
          count_prob_b=count_prob_b+1
        if float(all_humantaf_lines_as_lists[prob][21])==float(1) and
float(all_humantaf_lines_as_lists[prob][20])==float(1):
          count_prob_a_given_b=count_prob_a_given_b+1
          count_prob_b_given_a=count_prob_b_given_a+1

  prob_A= ((count_prob_a/(eventcount))*100)
#   print(x,'', prob_A)
  prob_B= ((count_prob_b/(eventcount))*100)
#   print(x,'', prob_B)
  prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
  prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
  wind_prob_a_list.append(prob_A)
  wind_prob_b_list.append(prob_B)
  wind_prob_a_giv_b_list.append(prob_A_giv_B)
  wind_prob_b_giv_a_list.append(prob_B_giv_A)
wind_prob_a_giv_b_list.sort()
wind_prob_b_giv_a_list.sort()
print(len(wind_prob_a_list), 'max',max((wind_prob_a_list)),'min',min((wind_prob_a_list)),
stdev(wind_prob_a_list) )
print(len(wind_prob_b_list), 'max',max((wind_prob_b_list)),'min',min((wind_prob_b_list)),
stdev(wind_prob_b_list)  )
print('wind a|b error bounds',' upper', wind_prob_a_giv_b_list[17], 'lower',
wind_prob_a_giv_b_list[3])
print('wind b|a error bounds',' upper', wind_prob_b_giv_a_list[17], 'lower',
wind_prob_b_giv_a_list[3])

for x in range(len(stations)):
  count_prob_a2= 0
  count_prob_b2= 0
  count_prob_a_given_b2= 0
  count_prob_b_given_a2= 0
  eventcount2=0
  for prob in range(len(all_humantaf_lines_as_lists)):
    if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
      eventcount2=eventcount2+1
      #print(all_autotaf_lines_as_lists[prob])
      #print(all_autotaf_lines_as_lists[prob][1])
      if float(all_humantaf_lines_as_lists[prob][17])>=float(2):
        count_prob_a2=count_prob_a2+1
      if float(all_humantaf_lines_as_lists[prob][18])>=float(2):
```

```
            count_prob_b2=count_prob_b2+1
         if float(all_humantaf_lines_as_lists[prob][18])>=float(2) and
float(all_humantaf_lines_as_lists[prob][17])>=float(2):
            count_prob_a_given_b2=count_prob_a_given_b2+1
            count_prob_b_given_a2=count_prob_b_given_a2+1
   prob_A2= ((count_prob_a2/(eventcount2))*100)
   prob_B2= ((count_prob_b2/(eventcount2))*100)
   prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
   prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

   wind_prob_a_list_2nup.append(prob_A2)
   wind_prob_b_list_2nup.append(prob_B2)
   wind_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
   wind_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)
wind_prob_a_giv_b_list_2nup.sort()
wind_prob_b_giv_a_list_2nup.sort()
#print(wind_prob_b_giv_a_list_2nup)

print(len(wind_prob_a_list_2nup),
'max',max((wind_prob_a_list_2nup)),'min',min((wind_prob_a_list_2nup)),
stdev(wind_prob_a_list_2nup) )
print(len(wind_prob_b_list_2nup),
'max',max((wind_prob_b_list_2nup)),'min',min((wind_prob_b_list_2nup)),
stdev(wind_prob_b_list_2nup)  )
print('human 2nup wind a|b error bounds',' upper', wind_prob_a_giv_b_list_2nup[17], 'lower',
wind_prob_a_giv_b_list_2nup[3])
print('human 2nup wind b|a error bounds',' upper', wind_prob_b_giv_a_list_2nup[17], 'lower',
wind_prob_b_giv_a_list_2nup[3])
#%%
      ##################################################
####### Wind speed  conditionals
##################################################
#  prob(A) prob fcst =
count(all_humantaf_lines_as_lists[20]==1)/len(all_humantaf_lines_as_lists)
#  prob(B) prob obs =
count(all_humantaf_lines_as_lists[21]==1)/len(all_humantaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/ len(all_humantaf_lines_as_lists[21])
#  prob(B|A) prob observed given fcst = count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/ len(all_humantaf_lines_as_lists[20])

# overall human

wind_prob_a_list=[]
wind_prob_b_list=[]
wind_prob_a_giv_b_list=[]
```

```
wind_prob_b_giv_a_list=[]
wind_prob_a_list_2nup=[]
wind_prob_b_list_2nup=[]
wind_prob_a_giv_b_list_2nup=[]
wind_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_humantaf_lines_as_lists)):
      #if str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:
      eventcount=eventcount+1
         #print(all_humantaf_lines_as_lists[prob])
         #print(all_humantaf_lines_as_lists[prob][1])
      if float(all_humantaf_lines_as_lists[prob][20])==float(1):
         count_prob_a=count_prob_a+1
      if float(all_humantaf_lines_as_lists[prob][21])==float(1):
         count_prob_b=count_prob_b+1
      if float(all_humantaf_lines_as_lists[prob][21])==float(1) and
float(all_humantaf_lines_as_lists[prob][20])==float(1):
         count_prob_a_given_b=count_prob_a_given_b+1
         count_prob_b_given_a=count_prob_b_given_a+1

   prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
   prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
   prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
   prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
   wind_prob_a_list.append(prob_A)
   wind_prob_b_list.append(prob_B)
   wind_prob_a_giv_b_list.append(prob_A_giv_B)
   wind_prob_b_giv_a_list.append(prob_B_giv_A)

for x in range(len(stations)):
   count_prob_a2= 0
   count_prob_b2= 0
   count_prob_a_given_b2= 0
   count_prob_b_given_a2= 0
   eventcount2=0
   for prob in range(len(all_humantaf_lines_as_lists)):
      #if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
      eventcount2=eventcount2+1
```

```
      #print(all_autotaf_lines_as_lists[prob])
      #print(all_autotaf_lines_as_lists[prob][1])
      if float(all_humantaf_lines_as_lists[prob][17])>=float(2):
         count_prob_a2=count_prob_a2+1
      if float(all_humantaf_lines_as_lists[prob][18])>=float(2):
         count_prob_b2=count_prob_b2+1
      if float(all_humantaf_lines_as_lists[prob][18])>=float(2) and
float(all_humantaf_lines_as_lists[prob][17])>=float(2):
         count_prob_a_given_b2=count_prob_a_given_b2+1
         count_prob_b_given_a2=count_prob_b_given_a2+1
   prob_A2= ((count_prob_a2/(eventcount2))*100)
   prob_B2= ((count_prob_b2/(eventcount2))*100)
   prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
   prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

   wind_prob_a_list_2nup.append(prob_A2)
   wind_prob_b_list_2nup.append(prob_B2)
   wind_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
   wind_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)
#print(len(wind_prob_a_list_2nup),
'max',max((wind_prob_a_list_2nup)),'min',min((wind_prob_a_list_2nup)),
stdev(wind_prob_a_list_2nup) )
#print(len(wind_prob_b_list_2nup),
'max',max((wind_prob_b_list_2nup)),'min',min((wind_prob_b_list_2nup)),
stdev(wind_prob_b_list_2nup)  )

print('Human overall','prob a',prob_A,'prob b',prob_B,'prob a|b',prob_A_giv_B,'prob
b|a',prob_B_giv_A)
print('Human overall 2nup','prob a',prob_A2,'prob b',prob_B2,'prob a|b',prob_A_giv_B2,'prob
b|a',prob_B_giv_A2)

#%%


#Auto


#%%
      ##################################################
####### Wind speed  conditionals
##################################################
#  prob(A) prob fcst = count(all_autotaf_lines_as_lists[20]==1)/len(all_autotaf_lines_as_lists)
#  prob(B) prob obs = count(all_autotaf_lines_as_lists[21]==1)/len(all_autotaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_autotaf_lines_as_lists[21]==1 and
all_autotaf_lines_as_lists[20])/ len(all_autotaf_lines_as_lists[21])
```

```
#  prob(B|A) prob observed given fcst = count(all_autotaf_lines_as_lists[21]==1 and
all_autotaf_lines_as_lists[20])/ len(all_autotaf_lines_as_lists[20])

#
# By Station 1 and up auto
#
wind_prob_a_list=[]
wind_prob_b_list=[]
wind_prob_a_giv_b_list=[]
wind_prob_b_giv_a_list=[]
wind_prob_a_list_2nup=[]
wind_prob_b_list_2nup=[]
wind_prob_a_giv_b_list_2nup=[]
wind_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_autotaf_lines_as_lists)):
      if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
         eventcount=eventcount+1
         #print(all_autotaf_lines_as_lists[prob])
         #print(all_autotaf_lines_as_lists[prob][1])
         if float(all_autotaf_lines_as_lists[prob][20])==float(1):
            count_prob_a=count_prob_a+1
         if float(all_autotaf_lines_as_lists[prob][21])==float(1):
            count_prob_b=count_prob_b+1
         if float(all_autotaf_lines_as_lists[prob][21])==float(1) and
float(all_autotaf_lines_as_lists[prob][20])==float(1):
            count_prob_a_given_b=count_prob_a_given_b+1
            count_prob_b_given_a=count_prob_b_given_a+1

   prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
   prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
   prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
   prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
   wind_prob_a_list.append(prob_A)
   wind_prob_b_list.append(prob_B)
   wind_prob_a_giv_b_list.append(prob_A_giv_B)
   wind_prob_b_giv_a_list.append(prob_B_giv_A)
wind_prob_a_giv_b_list.sort()
```

```
wind_prob_b_giv_a_list.sort()
print(len(wind_prob_a_list), 'max',max((wind_prob_a_list)),'min',min((wind_prob_a_list)),
stdev(wind_prob_a_list) )
print(len(wind_prob_b_list), 'max',max((wind_prob_b_list)),'min',min((wind_prob_b_list)),
stdev(wind_prob_b_list)  )
print('wind a|b error bounds',' upper', wind_prob_a_giv_b_list[17], 'lower',
wind_prob_a_giv_b_list[3])
print('wind b|a error bounds',' upper', wind_prob_b_giv_a_list[17], 'lower',
wind_prob_b_giv_a_list[3])

for x in range(len(stations)):
   count_prob_a2= 0
   count_prob_b2= 0
   count_prob_a_given_b2= 0
   count_prob_b_given_a2= 0
   eventcount2=0
   for prob in range(len(all_autotaf_lines_as_lists)):
      if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
         eventcount2=eventcount2+1
         #print(all_autotaf_lines_as_lists[prob])
         #print(all_autotaf_lines_as_lists[prob][1])
         if float(all_autotaf_lines_as_lists[prob][17])>=float(2):
            count_prob_a2=count_prob_a2+1
         if float(all_autotaf_lines_as_lists[prob][18])>=float(2):
            count_prob_b2=count_prob_b2+1
         if float(all_autotaf_lines_as_lists[prob][18])>=float(2) and
float(all_autotaf_lines_as_lists[prob][17])>=float(2):
            count_prob_a_given_b2=count_prob_a_given_b2+1
            count_prob_b_given_a2=count_prob_b_given_a2+1
   prob_A2= ((count_prob_a2/(eventcount2))*100)
   prob_B2= ((count_prob_b2/(eventcount2))*100)
   prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
   prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

   wind_prob_a_list_2nup.append(prob_A2)
   wind_prob_b_list_2nup.append(prob_B2)
   wind_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
   wind_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)
wind_prob_a_giv_b_list_2nup.sort()
wind_prob_b_giv_a_list_2nup.sort()
#print(wind_prob_b_giv_a_list_2nup)

print(len(wind_prob_a_list_2nup),
'max',max((wind_prob_a_list_2nup)),'min',min((wind_prob_a_list_2nup)),
stdev(wind_prob_a_list_2nup) )
```

```
print(len(wind_prob_b_list_2nup),
'max',max((wind_prob_b_list_2nup)),'min',min((wind_prob_b_list_2nup)),
stdev(wind_prob_b_list_2nup)  )
print('auto 2nup wind a|b error bounds',' upper', wind_prob_a_giv_b_list_2nup[17], 'lower',
wind_prob_a_giv_b_list_2nup[3])
print('auto 2nup wind b|a error bounds',' upper', wind_prob_b_giv_a_list_2nup[17], 'lower',
wind_prob_b_giv_a_list_2nup[3])

      ##################################################
####### Wind speed  conditionals
##################################################
#  prob(A) prob fcst = count(all_autotaf_lines_as_lists[20]==1)/len(all_autotaf_lines_as_lists)
#  prob(B) prob obs = count(all_autotaf_lines_as_lists[21]==1)/len(all_autotaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_autotaf_lines_as_lists[21]==1 and
all_autotaf_lines_as_lists[20])/ len(all_autotaf_lines_as_lists[21])
#  prob(B|A) prob observed given fcst = count(all_autotaf_lines_as_lists[21]==1 and
all_autotaf_lines_as_lists[20])/ len(all_autotaf_lines_as_lists[20])
#

# overall auto


wind_prob_a_list=[]
wind_prob_b_list=[]
wind_prob_a_giv_b_list=[]
wind_prob_b_giv_a_list=[]
wind_prob_a_list_2nup=[]
wind_prob_b_list_2nup=[]
wind_prob_a_giv_b_list_2nup=[]
wind_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_autotaf_lines_as_lists)):
      #if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
      eventcount=eventcount+1
         #print(all_autotaf_lines_as_lists[prob])
         #print(all_autotaf_lines_as_lists[prob][1])
      if float(all_autotaf_lines_as_lists[prob][20])==float(1):
         count_prob_a=count_prob_a+1
      if float(all_autotaf_lines_as_lists[prob][21])==float(1):
         count_prob_b=count_prob_b+1
```

```
      if float(all_autotaf_lines_as_lists[prob][21])==float(1) and
float(all_autotaf_lines_as_lists[prob][20])==float(1):
         count_prob_a_given_b=count_prob_a_given_b+1
         count_prob_b_given_a=count_prob_b_given_a+1

   prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
   prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
   prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
   prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
   wind_prob_a_list.append(prob_A)
   wind_prob_b_list.append(prob_B)
   wind_prob_a_giv_b_list.append(prob_A_giv_B)
   wind_prob_b_giv_a_list.append(prob_B_giv_A)
#print(len(wind_prob_a_list), 'max',max((wind_prob_a_list)),'min',min((wind_prob_a_list)),
stdev(wind_prob_a_list) )
#print(len(wind_prob_b_list), 'max',max((wind_prob_b_list)),'min',min((wind_prob_b_list)),
stdev(wind_prob_b_list)  )


for x in range(len(stations)):
   count_prob_a2= 0
   count_prob_b2= 0
   count_prob_a_given_b2= 0
   count_prob_b_given_a2= 0
   eventcount2=0
   for prob in range(len(all_autotaf_lines_as_lists)):
      #if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
      eventcount2=eventcount2+1

      if float(all_autotaf_lines_as_lists[prob][17])>=float(2):
         count_prob_a2=count_prob_a2+1
      if float(all_autotaf_lines_as_lists[prob][18])>=float(2):
         count_prob_b2=count_prob_b2+1
      if float(all_autotaf_lines_as_lists[prob][18])>=float(2) and
float(all_autotaf_lines_as_lists[prob][17])>=float(2):
         count_prob_a_given_b2=count_prob_a_given_b2+1
         count_prob_b_given_a2=count_prob_b_given_a2+1
   prob_A2= ((count_prob_a2/(eventcount2))*100)
   prob_B2= ((count_prob_b2/(eventcount2))*100)
   prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
   prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

   wind_prob_a_list_2nup.append(prob_A2)
   wind_prob_b_list_2nup.append(prob_B2)
```

```
    wind_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
    wind_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)

print('Auto overall','prob a',prob_A,'prob b',prob_B,'prob a|b',prob_A_giv_B,'prob
b|a',prob_B_giv_A)
print('Auto overall 2nup','prob a',prob_A2,'prob b',prob_B2,'prob a|b',prob_A_giv_B2,'prob
b|a',prob_B_giv_A2)
```

TAF collector

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct  8 14:35:32 2019

@author: bbailey
"""

##################################################### importing necessary libraries
import os
import numpy as np
import datetime
datestart=datetime.datetime.now()
##################################################### defining directories and file names
directory = os.fsencode(r'I:\setup\Desktop\Thesis\Code\All Human Tafs\TAFs\KCEF')
openlines=[]
openlines2=[]
##################################################### opening file name and checking
key information in file name and within file
supercounter=0
```

```
letter4= str(directory[-4:])
for file in os.listdir(directory):
   #print(filename)
   #filename = str(file)
   filename = os.fsdecode(file)
   #print(len(directory))
   if filename[-9:]!="tempo.csv" and filename[-3:]!="txt" and filename[0:3]!="Fin":
#    if filename.endswith(".py") or filename.startswith("SAR"):
       openfile = open(filename).readlines()
       model_run_name=filename

       for lines in openfile:
          openlines.append(lines.split(','))
print(len(openlines))
np.savetxt('single_station_' + str(letter4) + '.csv',openlines,fmt='%s',delimiter=',')
#with open('all_auto_tafs', 'w+') as f:
#    for item in openlines:
#       f.writelines("%s\n" % item)
dateend=datetime.datetime.now()
print('start'+ str(datestart), 'end'+str(dateend))

for file in os.listdir(directory):
   #print(filename)
   #filename = str(file)
   filename = os.fsdecode(file)
   #print(len(directory))
   if filename[-9:]=="tempo.csv" and filename[-3:]!="txt" and filename[0:3]!="Fin" and
filename[0:3]!="sin":
#    if filename.endswith(".py") or filename.startswith("SAR"):
       openfile2 = open(filename).readlines()
       model_run_name=filename

       for lines in openfile2:
          openlines2.append(lines.split(','))
print(len(openlines2))
np.savetxt('tempo_single_station_' + str(letter4) + '.csv',openlines2,fmt='%s',delimiter=',')
#with open('all_auto_tafs', 'w+') as f:
#    for item in openlines:
#       f.writelines("%s\n" % item)
dateend=datetime.datetime.now()
print('start'+ str(datestart), 'end'+str(dateend))




#for folder in os.listdir(directory):
```

```
#    open_folder=open(str(folder)[2:6])
#    #for filename in os.listdir(os.getcwd()):
#    print(open_folder)
#    #print(str(folder)[2:6])
#    for file in os.fsdecode(folder):
#        filename = os.fsdecode(file)
#        print(filename)
#        if filename.endswith(".txt"): #and len(filename)<=24:#or filename.endswith(".py"):
#            print('month: ' + filename[-11:-9],'day: ' + filename[-9:-7],'hour: '+filename[-7:-4],
filename)
#    for file in os.listdir(directory):
#        filename = os.fsdecode(file)
#        if filename.endswith(".txt"):
#            openfile = open(filename).readlines()
#            model_run_name=filename
#            openlines=[]
#            for lines in openfile:
#                openlines.append(lines.split(','))
#            location=model_run_name[16:20]
```

TAF Comparer

```
# -*- coding: utf-8 -*-
"""
Created on Thu Oct 10 11:43:29 2019

@author: bbailey
"""
import os
import numpy as np
#########################################################################define
directory and indexes
directory = os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final
Code\2Dec_human_taf_complete_redo\all_main_and_all_tempo')
#directory_for_obs=os.fsencode(r'I:\setup\Desktop\Thesis\Code\OBS')
#alllines=[]
```

```
################################################################################
############################################################ open and
set up both taf and ob file ##########################
################################################################################
############################################################

################################################################################define
reference indexes that will be used for indexing
lgt_reference_index= 10
wind_spd_reference_index= 6
wind_dir_reference_index= 4
vis_index= 7
cig_index= 8
counter=0
supercounter=0
file_list=[]

################################################################################opening
TAF files and creating blank lists/also create variable to ease naming of output file

################################################################################
############################################################
################################################################################This
is where the loop will begin#########################
################################################################################
############################################################
#for file in os.listdir(directory):
#    filename = os.fsdecode(file)
#    counter=counter+1
#    print(filename[0:3])
#print(counter)
   #if filename.endswith(".txt"): #and len(filename)<=24:#or filename.endswith(".py"):
   #    print('month: ' + filename[-11:-9],'day: ' + filename[-9:-7],'hour: '+filename[-7:-4],
filename)
for file in os.listdir(directory):
   filename = os.fsdecode(file)
   #print(len(directory)) filename=='KFFO6118Zautotaf.csv' and
   if filename[0:15]=='final_all_human' and filename[0:3]!="Int" and filename[0:3]!="all" and
filename[0:3]!="SAR" and filename[0:3]!="TAF" and filename[0:3]!="Fin"and
filename[0:3]!="5CA" and filename[0:3]!="com"  and filename[4]!="4" and
filename[4]!="7":#and filename[0:3]!="BGT"
#    if filename.endswith(".py") or filename.startswith("SAR"):
       print(filename)
       taf_file=open(filename)
       #taf_file=open('KFTK52112Zautotaf.csv')
```

```
    taf_lines=taf_file.readlines()
    #print(taf_lines)
    alllines=[]
    all_taflines_as_lists=[]

##########################################################################preparing
read in lines of TAF info (putting them back in lines of lists)
    for x in range(len(taf_lines)):
    ##########################################################################get rid
of all whitespace and carriage returns then space delimit
      taf_lines_as_lists = taf_lines[x].strip().split(',')
      all_taflines_as_lists.append(taf_lines_as_lists)
      #print(taf_lines_as_lists)
      #counter=counter+1
    #print(counter)
    ##########################################################################
esbablish an easy way to determine what observation file is needed
      obs_index=str(all_taflines_as_lists[0][0])
      obs_index2=str(all_taflines_as_lists[0][1])
      obs_index3=str(all_taflines_as_lists[0][2])
      #print(obs_index)
      all_obs_lines_as_lists=[]
      ##########################################################################use
information to identify path to obseravtion file
      obs_file=open('SAR_10591_'+str(obs_index)+'.csv')
      obs_lines=obs_file.readlines()
    for x in range(len(obs_lines)):
      obs_lines_as_lists = obs_lines[x].strip().split(',')
      all_obs_lines_as_lists.append(obs_lines_as_lists)
      #print(obs_lines_as_lists)
      #counter=counter+1
    #print(counter)
    ##########################################################################filling
in missing data from obs


#%%
    line_for_missing_data=[]

##############################################################################
##########################################################
    ##########################################################################
identifying the correct row of data  ##########################

##############################################################################
##########################################################
```

```
    ##########################################################################
finding the correct ob for the taf
    correct_month_day_and_hour=[]
    taf_hrs_for_check=[]
    ob_hrs_for_check=[]
#     print(all)
    #print(len(all_taflines_as_lists),len(all_obs_lines_as_lists))
    listcount=0
    ##################################################this part will need to change for
human tafs
    for x in range(len(all_taflines_as_lists)):
      # print(all_taflines_as_lists[x])
       #if all_taflines_as_lists[x][4][0]=='0':
          #all_taflines_as_lists[x][4][0:2]=all_taflines_as_lists[x][4][1]
       #print(all_taflines_as_lists[x])
#        if all_taflines_as_lists[x][1]=='01':
#           all_taflines_as_lists[x][1]=='1'
#        if all_taflines_as_lists[x][1]=='02':
#           all_taflines_as_lists[x][1]=='2'
#        if all_taflines_as_lists[x][1]=='03':
#           all_taflines_as_lists[x][1]=='3'
#        if all_taflines_as_lists[x][1]=='04':
#           all_taflines_as_lists[x][1]=='4'
#        if all_taflines_as_lists[x][1]=='05':
#           all_taflines_as_lists[x][1]=='5'
#        if all_taflines_as_lists[x][1]=='06':
#           all_taflines_as_lists[x][1]=='6'
#        if all_taflines_as_lists[x][1]=='07':
#           all_taflines_as_lists[x][1]=='7'
#        if all_taflines_as_lists[x][1]=='08':
#           all_taflines_as_lists[x][1]=='8'
#        if all_taflines_as_lists[x][1]=='09':
#           all_taflines_as_lists[x][1]=='9'


        taf_hrs_for_check.append(all_taflines_as_lists[x][4])
        print('checkpoint1')
        if x>listcount:
            correct_month_day_and_hour.append(0)
            ob_hrs_for_check.append(0)
            listcount=listcount+1
        for xx in range(len(all_obs_lines_as_lists)):
          #print(all_taflines_as_lists[0][1],all_obs_lines_as_lists[0][1])
```

```
            if int(all_taflines_as_lists[x][1])==int(all_obs_lines_as_lists[xx][1]) and
all_taflines_as_lists[x][2]==all_obs_lines_as_lists[xx][2] and
all_taflines_as_lists[x][4]==all_obs_lines_as_lists[xx][3]:
                correct_month_day_and_hour.append(xx+1)
                ob_hrs_for_check.append(all_obs_lines_as_lists[xx][3])
                listcount=listcount+1
            else:
               continue
         #print(correct_month_day_and_hour)
      #for x in correct_month_day_and_hour:
      #print(len(correct_month_day_and_hour))
      #print((correct_month_day_and_hour))
      original_correct_month_day_and_hour=correct_month_day_and_hour
      if correct_month_day_and_hour[0]==0:
         correct_month_day_and_hour.insert(0,correct_month_day_and_hour[1]-1)
         correct_month_day_and_hour.pop(1)
      ##########################################################################
correcting for missing data
#       print(len(correct_month_day_and_hour))
      if len(correct_month_day_and_hour) <(len(all_taflines_as_lists)):

         if not int(taf_hrs_for_check[0]) == int(ob_hrs_for_check[0])-1 and not
int(taf_hrs_for_check[0]) == int(ob_hrs_for_check[0])+23:
            y=int(taf_hrs_for_check[0])-(int(ob_hrs_for_check[0])-1)
            correct_month_day_and_hour.insert(0,0)
            print('wwwwwwwwwwwwwwwwwwooooooooooooooaaaaaaah boy i had to add one to
the beginning',len(correct_month_day_and_hour))
#
         if len(correct_month_day_and_hour) <(len(all_taflines_as_lists)):
            if not int(taf_hrs_for_check[-1])== int(ob_hrs_for_check[-1])-1 and not
int(taf_hrs_for_check[-1])== int(ob_hrs_for_check[-1])+23:
               z=int(taf_hrs_for_check[-1])-(int(ob_hrs_for_check[-1])-1)
               correct_month_day_and_hour.append(0)
               print('ooooooooooooooooooooooooooooh boy i had to add one to the
end',len(correct_month_day_and_hour))

         ################################## add to the end of the list
      #if len(correct_month_day_and_hour) <30:
         ################################## add to the interior of the taf

#               continue

##############################################################################
##########################################################
      ######################################################################## filling
lists of obs data  ####################################
```

```
##############################################################################
#############################################################

    ########################################################################## obs
file indexing is as follows: YEAR,MO,DAY,HR,WDIR,WSPD,WMAX,VIS,CIG,PRCP,TSTM
    #lgt_reference_index= 10
    #wind_spd_reference_index= 6
    #wind_dir_reference_index= 4
    #vis_index= 7
    #cig_index= 8

    obs_lightning=[]
    obs_wind_spd=[]
    obs_wind_dir=[]
    obs_vis=[]
    obs_ceiling=[]
    #print(all_obs_lines_as_lists[x][4])
    print('heeeeeeeeeeeeeeey', len(correct_month_day_and_hour),
(correct_month_day_and_hour))
    for x in correct_month_day_and_hour:
      #print(all_obs_lines_as_lists[x][4])
      obs_wind_spd.append(float(all_obs_lines_as_lists[x][6]))
      obs_wind_dir.append(float(all_obs_lines_as_lists[x][4]))
      obs_vis.append(float(all_obs_lines_as_lists[x][7]))
      obs_ceiling.append(float(all_obs_lines_as_lists[x][8]))
      obs_lightning.append(float(all_obs_lines_as_lists[x][10]))


##############################################################################
#############################################################
    ##########################################################################
creating quick reference of taf data    ####################

##############################################################################
#############################################################

    ########################################################################## taf
file indexing is as follows: YEAR,MO,DAY,HR,WDIR,WSPD,WMAX,VIS,CIG,PRCP,TSTM
    #lgt_reference_index= 8
    #wind_spd_reference_index= 5
    #wind_dir_reference_index= 6
    #vis_index= 9
    #cig_index= 7

    fcst_lightning=[]
```

```
    fcst_wind_spd=[]
    fcst_wind_dir=[]
    fcst_vis=[]
    fcst_ceiling=[]

    for x in range(len(all_taflines_as_lists)):
      fcst_lightning.append(all_taflines_as_lists[x][8])
      fcst_wind_spd.append(all_taflines_as_lists[x][5])
      fcst_wind_dir.append(all_taflines_as_lists[x][6])
      fcst_vis.append(all_taflines_as_lists[x][9])
      fcst_ceiling.append(all_taflines_as_lists[x][7])


################################################################################
############################################################
    ############################################################################
doing some quick math    ####################################

################################################################################
############################################################



################################################################################
#  checking length of columns
    #print(len(correct_month_day_and_hour))
    #print(len(obs_lightning))
    #print(len(obs_wind_dir))
    #print(len(obs_wind_spd))
    #print(len(obs_vis))
    #print(len(obs_ceiling))


################################################################################
############################################################
    ############################################################################
Wind direction calculations  ####################################

################################################################################
############################################################


################################################################################func
tion for creating distances
    def angle_calculator(angle_1, angle_2):
      phi = abs(angle_2-angle_1) % 360
```

```
        sign = 1
        # used to calculate sign
        if not ((angle_1-angle_2 >= 0 and angle_1-angle_2 <= 180) or (
              angle_1-angle_2 <= -180 and angle_1-angle_2 >= -360)):
           sign = -1
        if phi > 180:
           result = 360-phi
        else:
           result = phi

        return result*sign

     # distance unit2 needs to move to reach unit1
#      print (angle_calculator(90,360))  # output 45
#      print (angle_calculator(0, 350)) # output -80
#      print (angle_calculator(350, 30)) # output -40 (unit2 moves -40 degrees)
#      print (angle_calculator(30, 350)) # output 40

     #1) wind_dir values (fcst)= fcst_wind_dir
     #2) wind_dir values (obs)= obs_wind_dir
     #3) wind_dir values (dif)
     fcst_wind_dir_capped=[]
     obs_wind_dir_capped=[]
     wind_dir_values_dif=[]

     ######################################## just to keep naming convention for this code

     #print('yooooooooo',len(fcst_wind_dir),len(obs_wind_dir))
     for x in range(len(all_taflines_as_lists)):
        if float(fcst_wind_dir[x])<= 360:
           fcst_wind_dir_capped.append(float(fcst_wind_dir[x]))
        else:
           fcst_wind_dir_capped.append(float(fcst_wind_dir[x]))


        if float(obs_wind_dir[x])<= 360:
           obs_wind_dir_capped.append(float(obs_wind_dir[x]))
        else:
           obs_wind_dir_capped.append(float(obs_wind_dir[x]))

        wind_dir_values_dif.append(angle_calculator(fcst_wind_dir_capped[x],
obs_wind_dir_capped[x]))
     ##################
     #Nato wind_dir
     #points|cat|SM lower bound value|
     # 0,1,2,3,4,5|1,2,3,4,5,6|0,15,25,35,45,55
```

```
    #4) wind_dir bin val (fcst)
    #5) wind_dir bin val (obs)
    #6) wind_dir bin val (dif)
    fcst_wind_dir_capped_binned=[]
    obs_wind_dir_capped_binned=[]
    wind_dir_values_binned_dif=[]

    for x in range(len(all_taflines_as_lists)):

       if abs(float(wind_dir_values_dif[x]))<= 30:
          wind_dir_values_binned_dif.append(float(0))
       elif abs(float(wind_dir_values_dif[x]))> 30 and abs(float(wind_dir_values_dif[x])) <=45:
          wind_dir_values_binned_dif.append(float(1))
       elif abs(float(wind_dir_values_dif[x]))> 45 and abs(float(wind_dir_values_dif[x])) <=60:
          wind_dir_values_binned_dif.append(float(2))
       elif abs(float(wind_dir_values_dif[x]))> 60 and abs(float(wind_dir_values_dif[x])) <=90:
          wind_dir_values_binned_dif.append(float(3))
       elif abs(float(wind_dir_values_dif[x]))> 90 and abs(float(wind_dir_values_dif[x])) <=120:
          wind_dir_values_binned_dif.append(float(4))
       elif abs(float(wind_dir_values_dif[x]))> 120 and abs(float(wind_dir_values_dif[x])) <= 180:
          wind_dir_values_binned_dif.append(float(5))



#print('testing',wind_dir_values_dif,wind_dir_values_binned_dif,len(fcst_wind_dir_capped),len(
fcst_wind_dir_capped_binned), len(obs_wind_dir_capped_binned))




##############################################################################
############################################################
    ########################################################################
Wind speed calculations  ####################################

##############################################################################
############################################################
    #1) wind_spd values (fcst)= fcst_wind_spd
    #2) wind_spd values (obs)= obs_wind_spd
    #3) wind_spd values (dif)=wind_spd_values_dif
    fcst_wind_spd_capped=[]
    obs_wind_spd_capped=[]
    wind_spd_values_dif=[]
    for x in range(len(all_taflines_as_lists)):
       if float(fcst_wind_spd[x])>= 100:
```

```
      fcst_wind_spd_capped.append(float(100))
   else:
      fcst_wind_spd_capped.append(float(fcst_wind_spd[x]))


   if obs_wind_spd[x]>= 100:
      obs_wind_spd_capped.append(float(100))
   else:
      obs_wind_spd_capped.append(float(obs_wind_spd[x]))

   wind_spd_values_dif.append(fcst_wind_spd_capped[x]- obs_wind_spd_capped[x])
###################
#Nato wind_spd
#points|cat|SM lower bound value|
# 0,1,2,3,4,5|1,2,3,4,5,6|0,15,25,35,45,55

#4) wind_spd bin val (fcst)=fcst_wind_spd_capped_binned
#5) wind_spd bin val (obs)=obs_wind_spd_capped_binned
#6) wind_spd bin val (dif)=wind_spd_values_binned_dif
fcst_wind_spd_capped_binned=[]
obs_wind_spd_capped_binned=[]
wind_spd_values_binned_dif=[]

for x in range(len(all_taflines_as_lists)):
   if fcst_wind_spd_capped[x]< 15:
      fcst_wind_spd_capped_binned.append(float(0))
   elif fcst_wind_spd_capped[x]>= 15 and fcst_wind_spd_capped[x]< 25:
      fcst_wind_spd_capped_binned.append(float(1))
   elif fcst_wind_spd_capped[x]>= 25 and fcst_wind_spd_capped[x]< 35:
      fcst_wind_spd_capped_binned.append(float(2))
   elif fcst_wind_spd_capped[x]>= 35 and fcst_wind_spd_capped[x]< 45:
      fcst_wind_spd_capped_binned.append(float(3))
   elif fcst_wind_spd_capped[x]>= 45 and fcst_wind_spd_capped[x]< 55:
      fcst_wind_spd_capped_binned.append(float(4))
   elif fcst_wind_spd_capped[x]>= 55 :
      fcst_wind_spd_capped_binned.append(float(5))

   if obs_wind_spd_capped[x]< 15:
      obs_wind_spd_capped_binned.append(float(0))
   elif obs_wind_spd_capped[x]>= 15 and obs_wind_spd_capped[x]< 25 :
      obs_wind_spd_capped_binned.append(float(1))
   elif obs_wind_spd_capped[x]>= 25 and obs_wind_spd_capped[x]< 35:
      obs_wind_spd_capped_binned.append(float(2))
   elif obs_wind_spd_capped[x]>= 35 and obs_wind_spd_capped[x]< 45:
      obs_wind_spd_capped_binned.append(float(3))
   elif obs_wind_spd_capped[x]>= 45 and obs_wind_spd_capped[x]< 55:
```

```
            obs_wind_spd_capped_binned.append(float(4))
         elif obs_wind_spd_capped[x]>= 55:
            obs_wind_spd_capped_binned.append(float(5))

         wind_spd_values_binned_dif.append(fcst_wind_spd_capped_binned[x]-
obs_wind_spd_capped_binned[x])

#       print(len(fcst_wind_spd_capped))
#       print((fcst_wind_spd_capped))
#       print((obs_wind_spd_capped))
#       print(wind_spd_values_dif)
#       print(len(obs_wind_spd_capped))
#       print(len(wind_spd_values_dif))
#       print(len(fcst_wind_spd_capped_binned))
#       print(len(obs_wind_spd_capped_binned))
#       print((obs_wind_spd_capped_binned))
#       print(len(wind_spd_values_binned_dif))
#       print((wind_spd_values_binned_dif))

      #7) wind_spd event (fcst)=wind_spd_event_fcst
      #8) wind_spd event (obs)=wind_spd_event_obs
      #9) wind_spd capped dif given fcst=wind_spd_event_capped_dif_given_fcst
      #10) wind_spd capped dif given obs=wind_spd_event_capped_dif_given_obs
      #11) wind_spd capped binned dif given
fcst=wind_spd_event_capped_binned_dif_given_fcst
      #12) wind_spd capped binned dif given
obs=wind_spd_event_capped_binned_dif_given_obs
      #13) wind_spd matched event given hrs=wind_spd_event_identified_not_criteria_specific
      #14) wind_spd missed event hours=wind_spd_missed
      #15) wind_spd false alarm=wind_spd_false_alarm
      wind_spd_event_fcst=[]
      wind_spd_event_obs=[]
      wind_spd_event_capped_dif_given_fcst=[]
      wind_spd_event_capped_dif_given_obs=[]
      wind_spd_event_capped_binned_dif_given_fcst=[]
      wind_spd_event_capped_binned_dif_given_obs=[]
      wind_spd_event_identified_not_criteria_specific=[]
      wind_spd_missed=[]
      wind_spd_false_alarm=[]

      for x in range(len(all_taflines_as_lists)):
      ########### 7)
         if fcst_wind_spd_capped_binned[x] != 0:
            wind_spd_event_fcst.append(float(1))
         else:
            wind_spd_event_fcst.append(0)
```

```
    ########### 8)
      if obs_wind_spd_capped_binned[x] != 0 :
        wind_spd_event_obs.append(float(1))
      else:
        wind_spd_event_obs.append(0)
    ########### 9)
      if fcst_wind_spd_capped[x] >= 15:
        wind_spd_event_capped_dif_given_fcst.append(float(wind_spd_values_dif[x]))
      else:
        wind_spd_event_capped_dif_given_fcst.append(-9999)
    ########### 10)
      if obs_wind_spd_capped[x] >= 15:
        wind_spd_event_capped_dif_given_obs.append(float(wind_spd_values_dif[x]))
      else:
        wind_spd_event_capped_dif_given_obs.append(-9999)
    ########### 11)
      if fcst_wind_spd_capped_binned[x] != 0:

wind_spd_event_capped_binned_dif_given_fcst.append(float(wind_spd_values_binned_dif[x]))
      else:
        wind_spd_event_capped_binned_dif_given_fcst.append(-9999)
    ########### 12)
      if obs_wind_spd_capped_binned[x] != 0:

wind_spd_event_capped_binned_dif_given_obs.append(float(wind_spd_values_binned_dif[x]))
      else:
        wind_spd_event_capped_binned_dif_given_obs.append(-9999)
    ########## 13)
      if fcst_wind_spd_capped_binned[x] > 0 and obs_wind_spd_capped_binned[x] > 0:
        wind_spd_event_identified_not_criteria_specific.append(1.0)
      else:
        wind_spd_event_identified_not_criteria_specific.append(0.0)
    ########## 14)
      if fcst_wind_spd_capped_binned[x] > 0 and obs_wind_spd_capped_binned[x] == 0:
        wind_spd_false_alarm.append(1.0)
      else:
        wind_spd_false_alarm.append(0.0)
    ########## 15)
      if fcst_wind_spd_capped_binned[x] == 0 and obs_wind_spd_capped_binned[x] > 0:
        wind_spd_missed.append(1.0)
      else:
        wind_spd_missed.append(0.0)

    #print(len(wind_spd_event_capped_dif_given_obs))
    #print((wind_spd_event_capped_dif_given_obs))
    #print(len(obs_wind_spd_capped))
```

```
    #print(len(wind_spd_values_dif))
    #print(len(fcst_wind_spd_capped_binned))
    #print(len(wind_spd_event_capped_binned_dif_given_fcst))
    #print((wind_spd_event_capped_binned_dif_given_fcst))
    #print(len(wind_spd_event_capped_binned_dif_given_obs))
    #print((wind_spd_event_capped_binned_dif_given_obs))
    #print(fcst_wind_spd_capped_binned)



################################################################################
############################################################
    ########################################################################
Visibility calculations ####################################

################################################################################
############################################################

    #1) Visibilty values (fcst)= fcst_vis
    #2) Visibiltiy values (obs)= obs_vis
    #3) Vis values (dif)
    fcst_vis_capped=[]
    obs_vis_capped=[]
    vis_values_dif=[]
    for x in range(len(all_taflines_as_lists)):
      #### needs the first line changed for auto taf because of the 6.5
      if float(fcst_vis[x])>= 9600 or float(fcst_vis[x])<0:
      #if float(fcst_vis[x])>= 6.5  or float(fcst_vis[x])>= 9600 or float(fcst_vis[x])<0:
        fcst_vis_capped.append(float(7))
      elif float(fcst_vis[x])>= 6 and float(fcst_vis[x]) < 6.5 or float(fcst_vis[x])>= 9000 and
float(fcst_vis[x]) < 9600:
        fcst_vis_capped.append(float(6))
      elif float(fcst_vis[x])>= 5 and float(fcst_vis[x]) < 6 or float(fcst_vis[x])>= 8000 and
float(fcst_vis[x]) < 9000:
        fcst_vis_capped.append(float(5))
      elif float(fcst_vis[x])>= 4 and float(fcst_vis[x]) < 5 or float(fcst_vis[x])>= 6000 and
float(fcst_vis[x]) < 8000:
        fcst_vis_capped.append(float(4))
      elif float(fcst_vis[x])>= 3 and float(fcst_vis[x]) < 4 or float(fcst_vis[x])>= 4800 and
float(fcst_vis[x]) < 6000:
        fcst_vis_capped.append(float(3))
      elif float(fcst_vis[x])>= 2.5 and float(fcst_vis[x]) < 3 or float(fcst_vis[x])>= 4000 and
float(fcst_vis[x]) < 4800:
        fcst_vis_capped.append(float(2.5))
      elif float(fcst_vis[x])>= 2.25 and float(fcst_vis[x]) < 2.5 or float(fcst_vis[x])>= 3600 and
float(fcst_vis[x]) < 4000:
```

```
            fcst_vis_capped.append(float(2.25))
        elif float(fcst_vis[x])>= 2 and float(fcst_vis[x]) < 2.25 or float(fcst_vis[x])>= 3200 and
float(fcst_vis[x]) < 3600:
            fcst_vis_capped.append(float(2))
        elif float(fcst_vis[x])>= 1.875 and float(fcst_vis[x]) < 2 or float(fcst_vis[x])>= 3000 and
float(fcst_vis[x]) < 3200:
            fcst_vis_capped.append(float(1.875))
        elif float(fcst_vis[x])>= 1.75 and float(fcst_vis[x]) < 1.875 or float(fcst_vis[x])>= 2800 and
float(fcst_vis[x]) < 3000:
            fcst_vis_capped.append(float(1.75))
        elif float(fcst_vis[x])>= 1.625 and float(fcst_vis[x]) < 1.75 or float(fcst_vis[x])>= 2600 and
float(fcst_vis[x]) < 2800:
            fcst_vis_capped.append(float(1.625))
        elif float(fcst_vis[x])>= 1.5 and float(fcst_vis[x]) < 1.625 or float(fcst_vis[x])>= 2400 and
float(fcst_vis[x]) < 2600:
            fcst_vis_capped.append(float(1.5))
        elif float(fcst_vis[x])>= 1.375 and float(fcst_vis[x]) < 1.5 or float(fcst_vis[x])>= 2200 and
float(fcst_vis[x]) < 2400:
            fcst_vis_capped.append(float(1.375))
        elif float(fcst_vis[x])>= 1.25 and float(fcst_vis[x]) < 1.375 or float(fcst_vis[x])>= 2000 and
float(fcst_vis[x]) < 2200:
            fcst_vis_capped.append(float(1.25))
        elif float(fcst_vis[x])>= 1.125 and float(fcst_vis[x]) < 1.25 or float(fcst_vis[x])>= 1800 and
float(fcst_vis[x]) < 2000:
            fcst_vis_capped.append(float(1.125))
        elif float(fcst_vis[x])>= 1 and float(fcst_vis[x]) < 1.125 or float(fcst_vis[x])>= 1600 and
float(fcst_vis[x]) < 1800:
            fcst_vis_capped.append(float(1))
        elif float(fcst_vis[x])>= .875 and float(fcst_vis[x]) < 1 or float(fcst_vis[x])>= 1400 and
float(fcst_vis[x]) < 1600:
            fcst_vis_capped.append(float(.875))
        elif float(fcst_vis[x])>= .75 and float(fcst_vis[x]) < .875 or float(fcst_vis[x])>= 1200 and
float(fcst_vis[x]) < 1400:
            fcst_vis_capped.append(float(.75))
        elif float(fcst_vis[x])>= .625 and float(fcst_vis[x]) < .75 or float(fcst_vis[x])>= 1000 and
float(fcst_vis[x]) < 1200:
            fcst_vis_capped.append(float(.625))
        elif float(fcst_vis[x])>= .5 and float(fcst_vis[x]) < .625 or float(fcst_vis[x])>= 800 and
float(fcst_vis[x]) < 1000:
            fcst_vis_capped.append(float(.5))
        elif float(fcst_vis[x])>= .375 and float(fcst_vis[x]) < .5 or float(fcst_vis[x])>= 600 and
float(fcst_vis[x]) < 800:
            fcst_vis_capped.append(float(.375))
        elif float(fcst_vis[x])>= .3125 and float(fcst_vis[x]) < .375 or float(fcst_vis[x])>= 500 and
float(fcst_vis[x]) < 600:
            fcst_vis_capped.append(float(.3125))
```

```
        elif float(fcst_vis[x])>= .25 and float(fcst_vis[x]) < .3125 or float(fcst_vis[x])>= 400 and
float(fcst_vis[x]) < 500:
            fcst_vis_capped.append(float(.25))
        elif float(fcst_vis[x])>= .1875 and float(fcst_vis[x]) < .25 or float(fcst_vis[x])>= 300 and
float(fcst_vis[x]) < 400:
            fcst_vis_capped.append(float(.1875))
        elif float(fcst_vis[x])>= .125  and float(fcst_vis[x]) < .1875 or float(fcst_vis[x])>= 200 and
float(fcst_vis[x]) < 300:
            fcst_vis_capped.append(float(.125))
        elif float(fcst_vis[x])>= .0625 and float(fcst_vis[x]) < .125 or float(fcst_vis[x])>= 100 and
float(fcst_vis[x]) < 200:
            fcst_vis_capped.append(float(.0625))
        elif float(fcst_vis[x])< .0625 and float(fcst_vis[x])>= 0 or float(fcst_vis[x])< 100 and
float(fcst_vis[x])>= 0:
            fcst_vis_capped.append(float(0))


##%%


        if float(obs_vis[x])>= 6.5 and float(obs_vis[x])<= 15 or float(obs_vis[x])>= 9600 or
float(obs_vis[x])< -1 :
            obs_vis_capped.append(float(7))
        elif float(obs_vis[x])>= 6 and float(obs_vis[x]) < 6.5 or float(obs_vis[x])>= 9000 and
float(obs_vis[x]) < 9600:
            obs_vis_capped.append(float(6))
        elif float(obs_vis[x])>= 5 and float(obs_vis[x]) < 6 or float(obs_vis[x])>= 8000 and
float(obs_vis[x]) < 9000:
            obs_vis_capped.append(float(5))
        elif float(obs_vis[x])>= 4 and float(obs_vis[x]) < 5 or float(obs_vis[x])>= 6000 and
float(obs_vis[x]) < 8000:
            obs_vis_capped.append(float(4))
        elif float(obs_vis[x])>= 3 and float(obs_vis[x]) < 4 or float(obs_vis[x])>= 4800 and
float(obs_vis[x]) < 6000:
            obs_vis_capped.append(float(3))
        elif float(obs_vis[x])>= 2.5 and float(obs_vis[x]) < 3 or float(obs_vis[x])>= 4000 and
float(obs_vis[x]) < 4800:
            obs_vis_capped.append(float(2.5))
        elif float(obs_vis[x])>= 2.25 and float(obs_vis[x]) < 2.5 or float(obs_vis[x])>= 3600 and
float(obs_vis[x]) < 4000:
            obs_vis_capped.append(float(2.25))
        elif float(obs_vis[x])>= 2 and float(obs_vis[x]) < 2.25 or float(obs_vis[x])>= 3200 and
float(obs_vis[x]) < 3600:
            obs_vis_capped.append(float(2))
        elif float(obs_vis[x])>= 1.875 and float(obs_vis[x]) < 2 or float(obs_vis[x])>= 3000 and
float(obs_vis[x]) < 3200:
```

```
            obs_vis_capped.append(float(1.875))
        elif float(obs_vis[x])>= 1.75 and float(obs_vis[x]) < 1.875 or float(obs_vis[x])>= 2800 and
float(obs_vis[x]) < 3000:
            obs_vis_capped.append(float(1.75))
        elif float(obs_vis[x])>= 1.625 and float(obs_vis[x]) < 1.75 or float(obs_vis[x])>= 2600 and
float(obs_vis[x]) < 2800:
            obs_vis_capped.append(float(1.625))
        elif float(obs_vis[x])>= 1.5 and float(obs_vis[x]) < 1.625 or float(obs_vis[x])>= 2400 and
float(obs_vis[x]) < 2600:
            obs_vis_capped.append(float(1.5))
        elif float(obs_vis[x])>= 1.375 and float(obs_vis[x]) < 1.5 or float(obs_vis[x])>= 2200 and
float(obs_vis[x]) < 2400:
            obs_vis_capped.append(float(1.375))
        elif float(obs_vis[x])>= 1.25 and float(obs_vis[x]) < 1.375 or float(obs_vis[x])>= 2000 and
float(obs_vis[x]) < 2200:
            obs_vis_capped.append(float(1.25))
        elif float(obs_vis[x])>= 1.125 and float(obs_vis[x]) < 1.25 or float(obs_vis[x])>= 1800 and
float(obs_vis[x]) < 2000:
            obs_vis_capped.append(float(1.125))
        elif float(obs_vis[x])>= 1 and float(obs_vis[x]) < 1.125 or float(obs_vis[x])>= 1600 and
float(obs_vis[x]) < 1800:
            obs_vis_capped.append(float(1))
        elif float(obs_vis[x])>= .875 and float(obs_vis[x]) < 1 or float(obs_vis[x])>= 1400 and
float(obs_vis[x]) < 1600:
            obs_vis_capped.append(float(.875))
        elif float(obs_vis[x])>= .75 and float(obs_vis[x]) < .875 or float(obs_vis[x])>= 1200 and
float(obs_vis[x]) < 1400:
            obs_vis_capped.append(float(.75))
        elif float(obs_vis[x])>= .625 and float(obs_vis[x]) < .75 or float(obs_vis[x])>= 1000 and
float(obs_vis[x]) < 1200:
            obs_vis_capped.append(float(.625))
        elif float(obs_vis[x])>= .5 and float(obs_vis[x]) < .625 or float(obs_vis[x])>= 800 and
float(obs_vis[x]) < 1000:
            obs_vis_capped.append(float(.5))
        elif float(obs_vis[x])>= .375 and float(obs_vis[x]) < .5 or float(obs_vis[x])>= 600 and
float(obs_vis[x]) < 800:
            obs_vis_capped.append(float(.375))
        elif float(obs_vis[x])>= .3125 and float(obs_vis[x]) < .375 or float(obs_vis[x])>= 500 and
float(obs_vis[x]) < 600:
            obs_vis_capped.append(float(.3125))
        elif float(obs_vis[x])>= .25 and float(obs_vis[x]) < .3125 or float(obs_vis[x])>= 400 and
float(obs_vis[x]) < 500:
            obs_vis_capped.append(float(.25))
        elif float(obs_vis[x])>= .1875 and float(obs_vis[x]) < .25 or float(obs_vis[x])>= 300 and
float(obs_vis[x]) < 400:
            obs_vis_capped.append(float(.1875))
```

```
        elif float(obs_vis[x])>= .125  and float(obs_vis[x]) < .1875 or float(obs_vis[x])>= 200 and
float(obs_vis[x]) < 300:
            obs_vis_capped.append(float(.125))
        elif float(obs_vis[x])>= .0625 and float(obs_vis[x]) < .125 or float(obs_vis[x])>= 100 and
float(obs_vis[x]) < 200:
            obs_vis_capped.append(float(.0625))
        elif float(obs_vis[x])< .0625 and float(obs_vis[x])>= 0 or float(obs_vis[x])< 100 and
float(obs_vis[x])>= 0:
            obs_vis_capped.append(float(0))

        print(obs_vis_capped[x], obs_vis[x])
##%%
        vis_values_dif.append(fcst_vis_capped[x]- obs_vis_capped[x])
    ###################
    #Nato Vis
    #points|cat|SM lower bound value|
    # 0,1,2,3,4,5|1,2,3,4,5,6|5,3,2,1,.5,0

    #4) Vis bin val (fcst)
    #5) Vis bin val (obs)
    #6) Vis bin val (dif)
    fcst_vis_capped_binned=[]
    obs_vis_capped_binned=[]
    vis_values_binned_dif=[]

    for x in range(len(all_taflines_as_lists)):
        if fcst_vis_capped[x]>= 5 and fcst_vis_capped[x] < 50 or fcst_vis_capped[x]>= 8000 :
            fcst_vis_capped_binned.append(float(0))
        elif fcst_vis_capped[x]>= 3 and fcst_vis_capped[x] < 5 or fcst_vis_capped[x]>= 4800 and
fcst_vis_capped[x] < 8000:
            fcst_vis_capped_binned.append(float(1))
        elif fcst_vis_capped[x]>= 2 and fcst_vis_capped[x] < 3 or fcst_vis_capped[x]>= 3200 and
fcst_vis_capped[x] < 4800:
            fcst_vis_capped_binned.append(float(2))
        elif fcst_vis_capped[x]>= 1 and fcst_vis_capped[x] < 2 or fcst_vis_capped[x]>= 1600 and
fcst_vis_capped[x] < 3200:
            fcst_vis_capped_binned.append(float(3))
        elif fcst_vis_capped[x]>= .5 and fcst_vis_capped[x] < 1 or fcst_vis_capped[x]>= 800 and
fcst_vis_capped[x] < 1600:
            fcst_vis_capped_binned.append(float(4))
        else:
            fcst_vis_capped_binned.append(float(5))

        if obs_vis_capped[x]>= 5 and obs_vis_capped[x] < 50 or obs_vis_capped[x]>= 8000 :
            obs_vis_capped_binned.append(float(0))
```

```
        elif obs_vis_capped[x]>= 3 and obs_vis_capped[x] < 5 or obs_vis_capped[x]>= 4800 and
obs_vis_capped[x] < 8000:
            obs_vis_capped_binned.append(float(1))
        elif obs_vis_capped[x]>= 2 and obs_vis_capped[x] < 3 or obs_vis_capped[x]>= 3200 and
obs_vis_capped[x] < 4800:
            obs_vis_capped_binned.append(float(2))
        elif obs_vis_capped[x]>= 1 and obs_vis_capped[x] < 2 or obs_vis_capped[x]>= 1600 and
obs_vis_capped[x] < 3200:
            obs_vis_capped_binned.append(float(3))
        elif obs_vis_capped[x]>= .5 and obs_vis_capped[x] < 1 or obs_vis_capped[x]>= 800 and
obs_vis_capped[x] < 1600:
            obs_vis_capped_binned.append(float(4))
        else:
            obs_vis_capped_binned.append(float(5))

        vis_values_binned_dif.append(fcst_vis_capped_binned[x]- obs_vis_capped_binned[x])

    #print(len(fcst_vis_capped))
    #print((fcst_vis_capped))
    #print(len(obs_vis_capped))
    #print(len(vis_values_dif))
    #print(len(fcst_vis_capped_binned))
    #print(len(obs_vis_capped_binned))
    #print((obs_vis_capped_binned))
    #print(len(vis_values_binned_dif))
    #print((vis_values_binned_dif))

    #7) Vis event (fcst)
    #8) Vis event (obs)
    #9) vis capped dif given fcst
    #10) vis capped dif given obs
    #11) vis capped binned dif given fcst
    #12) vis capped binned dif given obs
    #13) vis matched event given hrs
    #14) vis missed event hours
    #15) vis false alarm
    vis_event_fcst=[]
    vis_event_obs=[]
    vis_event_capped_dif_given_fcst=[]
    vis_event_capped_dif_given_obs=[]
    vis_event_capped_binned_dif_given_fcst=[]
    vis_event_capped_binned_dif_given_obs=[]
    vis_event_identified_not_criteria_specific=[]
    vis_missed=[]
    vis_false_alarm=[]
```

```
for x in range(len(all_taflines_as_lists)):
########### 7)
   if fcst_vis_capped_binned[x] != 0:
      vis_event_fcst.append(float(1))
   else:
      vis_event_fcst.append(0)
########### 8)
   if obs_vis_capped_binned[x] != 0 :
      vis_event_obs.append(float(1))
   else:
      vis_event_obs.append(0)
########### 9)
   if fcst_vis_capped[x] < 5:
      vis_event_capped_dif_given_fcst.append(float(vis_values_dif[x]))
   else:
      vis_event_capped_dif_given_fcst.append(-9999)
########### 10)
   if obs_vis_capped[x] < 5:
      vis_event_capped_dif_given_obs.append(float(vis_values_dif[x]))
   else:
      vis_event_capped_dif_given_obs.append(-9999)
########### 11)
   if fcst_vis_capped_binned[x] != 0:
      vis_event_capped_binned_dif_given_fcst.append(float(vis_values_binned_dif[x]))
   else:
      vis_event_capped_binned_dif_given_fcst.append(-9999)
########### 12)
   if obs_vis_capped_binned[x] != 0:
      vis_event_capped_binned_dif_given_obs.append(float(vis_values_binned_dif[x]))
   else:
      vis_event_capped_binned_dif_given_obs.append(-9999)
########## 13)
   if fcst_vis_capped_binned[x] > 0 and obs_vis_capped_binned[x] > 0:
      vis_event_identified_not_criteria_specific.append(1.0)
   else:
      vis_event_identified_not_criteria_specific.append(0.0)
########## 14)
   if fcst_vis_capped_binned[x] > 0 and obs_vis_capped_binned[x] == 0:
      vis_false_alarm.append(1.0)
   else:
      vis_false_alarm.append(0.0)
########## 15)
   if fcst_vis_capped_binned[x] == 0 and obs_vis_capped_binned[x] > 0:
      vis_missed.append(1.0)
   else:
      vis_missed.append(0.0)
```

```
    #print(len(vis_event_capped_dif_given_obs))
    #print((vis_event_capped_dif_given_obs))
    #print(len(obs_vis_capped))
    #print(len(vis_values_dif))
    #print(len(fcst_vis_capped_binned))
    #print(len(vis_event_capped_binned_dif_given_fcst))
    #print((vis_event_capped_binned_dif_given_fcst))
    #print(len(vis_event_capped_binned_dif_given_obs))
    #print((vis_event_capped_binned_dif_given_obs))
    #print(fcst_vis_capped_binned)


################################################################################
#########################################################
    ########################################################################
Ceiling calculations  ###################################

################################################################################
#########################################################

    #1) Ceiling values (fcst)= fcst_ceiling
    #2) Ceiling values (obs)= obs_ceiling
    #3) ceiling values (dif)
    fcst_ceiling_capped=[]
    obs_ceiling_capped=[]
    ceiling_values_dif=[]
    for x in range(len(all_taflines_as_lists)):
      if float(fcst_ceiling[x])>= 25000:
        fcst_ceiling_capped.append(float(25000))
      else:
        fcst_ceiling_capped.append(float(fcst_ceiling[x]))


      if obs_ceiling[x]>= 25000:
        obs_ceiling_capped.append(float(25000))
      else:
        obs_ceiling_capped.append(float(obs_ceiling[x]))

      ceiling_values_dif.append(fcst_ceiling_capped[x]- obs_ceiling_capped[x])
    ###################
    #Nato ceiling
    #points|cat|SM lower bound value|
```

```
    # 0,1,2,3,4,5|1,2,3,4,5,6|2500,1500,700,300,200,<200

    #4) ceiling bin val (fcst)
    #5) ceiling bin val (obs)
    #6) ceiling bin val (dif)
    fcst_ceiling_capped_binned=[]
    obs_ceiling_capped_binned=[]
    ceiling_values_binned_dif=[]

    for x in range(len(all_taflines_as_lists)):
       if fcst_ceiling_capped[x]>= 2500 or fcst_ceiling_capped[x]< 0:
          fcst_ceiling_capped_binned.append(float(0))
       elif fcst_ceiling_capped[x]>= 1500:
          fcst_ceiling_capped_binned.append(float(1))
       elif fcst_ceiling_capped[x]>= 700:
          fcst_ceiling_capped_binned.append(float(2))
       elif fcst_ceiling_capped[x]>= 300:
          fcst_ceiling_capped_binned.append(float(3))
       elif fcst_ceiling_capped[x]>= 200:
          fcst_ceiling_capped_binned.append(float(4))
       elif fcst_ceiling_capped[x]< 200 and fcst_ceiling_capped[x]>= 0:
          fcst_ceiling_capped_binned.append(float(5))

       if obs_ceiling_capped[x]>= 2500 or obs_ceiling_capped[x]< 0:
          obs_ceiling_capped_binned.append(float(0))
       elif obs_ceiling_capped[x]>= 1500:
          obs_ceiling_capped_binned.append(float(1))
       elif obs_ceiling_capped[x]>= 700:
          obs_ceiling_capped_binned.append(float(2))
       elif obs_ceiling_capped[x]>= 300:
          obs_ceiling_capped_binned.append(float(3))
       elif obs_ceiling_capped[x]>= 200:
          obs_ceiling_capped_binned.append(float(4))
       elif obs_ceiling_capped[x]< 200 and obs_ceiling_capped[x]>= 0:
          obs_ceiling_capped_binned.append(float(5))

       ceiling_values_binned_dif.append(fcst_ceiling_capped_binned[x]-
obs_ceiling_capped_binned[x])

    #print(len(fcst_ceiling_capped))
    #print((fcst_ceiling_capped))
    #print(len(obs_ceiling_capped))
    #print(len(ceiling_values_dif))
    #print(len(fcst_ceiling_capped_binned))
    #print(len(obs_ceiling_capped_binned))
    #print((obs_ceiling_capped_binned))
```

```
#print(len(ceiling_values_binned_dif))
#print((ceiling_values_binned_dif))

#7) ceiling event (fcst)
#8) ceiling event (obs)
#9) ceiling capped dif given fcst
#10) ceiling capped dif given obs
#11) ceiling capped binned dif given fcst
#12) ceiling capped binned dif given obs
#13) ceiling matched event given hrs
#14) ceiling missed event hours
#15) ceiling false alarm
ceiling_event_fcst=[]
ceiling_event_obs=[]
ceiling_event_capped_dif_given_fcst=[]
ceiling_event_capped_dif_given_obs=[]
ceiling_event_capped_binned_dif_given_fcst=[]
ceiling_event_capped_binned_dif_given_obs=[]
ceiling_event_identified_not_criteria_specific=[]
ceiling_missed=[]
ceiling_false_alarm=[]

for x in range(len(all_taflines_as_lists)):
########### 7)
   if fcst_ceiling_capped_binned[x] != 0:
      ceiling_event_fcst.append(float(1))
   else:
      ceiling_event_fcst.append(0)
########### 8)
   if obs_ceiling_capped_binned[x] != 0 :
      ceiling_event_obs.append(float(1))
   else:
      ceiling_event_obs.append(0)
########### 9)
   if fcst_ceiling_capped[x] < 2500:
      ceiling_event_capped_dif_given_fcst.append(float(ceiling_values_dif[x]))
   else:
      ceiling_event_capped_dif_given_fcst.append(-9999)
########### 10)
   if obs_ceiling_capped[x] < 2500:
      ceiling_event_capped_dif_given_obs.append(float(ceiling_values_dif[x]))
   else:
      ceiling_event_capped_dif_given_obs.append(-9999)
########### 11)
   if fcst_ceiling_capped_binned[x] != 0:
```

```
ceiling_event_capped_binned_dif_given_fcst.append(float(ceiling_values_binned_dif[x]))
        else:
          ceiling_event_capped_binned_dif_given_fcst.append(-9999)
      ########### 12)
        if obs_ceiling_capped_binned[x] != 0:

ceiling_event_capped_binned_dif_given_obs.append(float(ceiling_values_binned_dif[x]))
        else:
          ceiling_event_capped_binned_dif_given_obs.append(-9999)
      ########## 13)
        if fcst_ceiling_capped_binned[x] > 0 and obs_ceiling_capped_binned[x] > 0:
          ceiling_event_identified_not_criteria_specific.append(1.0)
        else:
          ceiling_event_identified_not_criteria_specific.append(0.0)
      ########## 14)
        if fcst_ceiling_capped_binned[x] > 0 and obs_ceiling_capped_binned[x] == 0:
          ceiling_false_alarm.append(1.0)
        else:
          ceiling_false_alarm.append(0.0)
      ########## 15)
        if fcst_ceiling_capped_binned[x] == 0 and obs_ceiling_capped_binned[x] > 0:
          ceiling_missed.append(1.0)
        else:
          ceiling_missed.append(0.0)

      #print(len(ceiling_event_capped_dif_given_obs))
      #print((ceiling_event_capped_dif_given_obs))
      #print(len(obs_ceiling_capped))
      #print(len(ceiling_values_dif))
      #print(len(fcst_ceiling_capped_binned))
      #print(len(ceiling_event_capped_binned_dif_given_fcst))
      #print((ceiling_event_capped_binned_dif_given_fcst))
      #print(len(ceiling_event_capped_binned_dif_given_obs))
      #print((ceiling_event_capped_binned_dif_given_obs))
      #print(fcst_ceiling_capped_binned)



#########################################################################################
#########################################################
      ###########################################################################
lightining calculations    ####################################
```

```
################################################################################
#############################################################

    fcst_lightning
    obs_lightning

    #1 lightning_dif
    #2 given fcst
    #3 given obs

    lightning_dif=[]
    lightning_given_fcst=[]
    lightning_given_obs=[]

    for x in range(len(all_taflines_as_lists)):
      lightning_dif.append(float(fcst_lightning[x])-float(obs_lightning[x]))
      if fcst_lightning[x]==1 and obs_lightning[x]==1:
        lightning_given_fcst.append(1)
      elif fcst_lightning[x]==1 and obs_lightning[x]==0:
        lightning_given_fcst.append(0)
      else:
        lightning_given_fcst.append(-9999)

    for x in range(len(all_taflines_as_lists)):
      if fcst_lightning[x]==1 and obs_lightning[x]==1:
        lightning_given_obs.append(1)
      elif fcst_lightning[x]==0 and obs_lightning[x]==1:
        lightning_given_obs.append(0)
      else:
        lightning_given_obs.append(-9999)

    outf=np.empty(((len(all_taflines_as_lists)), 68), dtype=object)


    #print(len(fcst_wind_spd_capped),len( obs_wind_spd_capped),len(
wind_spd_values_dif),len(
fcst_wind_spd_capped_binned),len(obs_wind_spd_capped_binned),len(wind_spd_values_binn
ed_dif),len(wind_spd_event_fcst),len(wind_spd_event_obs),len(wind_spd_event_capped_dif_gi
ven_fcst),len(wind_spd_event_capped_dif_given_obs),len(wind_spd_event_capped_binned_dif
_given_fcst),len(wind_spd_event_capped_binned_dif_given_obs),len(wind_spd_event_identifie
d_not_criteria_specific),len(wind_spd_missed),len(wind_spd_false_alarm),len(fcst_wind_dir_ca
pped),len(obs_wind_dir_capped),len(wind_dir_values_dif),len(wind_dir_values_binned_dif
),len(fcst_vis_capped),len(obs_vis_capped),len(vis_values_dif),len(fcst_vis_capped_binned),len(
obs_vis_capped_binned),len(vis_values_binned_dif),len(vis_event_fcst),len(vis_event_obs),len(
vis_event_capped_dif_given_fcst),len(vis_event_capped_dif_given_obs),len(vis_event_capped_
```

```
binned_dif_given_fcst),len(vis_event_capped_binned_dif_given_obs),len(vis_event_identified_
not_criteria_specific),len(vis_missed),len(vis_false_alarm),len(fcst_ceiling_capped),len(obs_ceili
ng_capped),len(ceiling_values_dif),len(fcst_ceiling_capped_binned),len(obs_ceiling_capped_bin
ned),len(ceiling_values_binned_dif),len(ceiling_event_fcst),len(ceiling_event_obs),len(ceiling_e
vent_capped_dif_given_fcst),len(ceiling_event_capped_dif_given_obs),len(ceiling_event_cappe
d_binned_dif_given_fcst),len(ceiling_event_capped_binned_dif_given_obs),len(ceiling_event_i
dentified_not_criteria_specific),len(ceiling_missed),len(ceiling_false_alarm),len(lightning_dif),le
n(lightning_given_fcst),len(lightning_given_obs))
      for x in range(len(all_taflines_as_lists)):

outf[x,:]=[all_taflines_as_lists[x][0],all_taflines_as_lists[x][1],all_taflines_as_lists[x][2],all_tafline
s_as_lists[x][3],all_taflines_as_lists[x][4],all_taflines_as_lists[x][5],all_taflines_as_lists[x][6],all_t
aflines_as_lists[x][7],all_taflines_as_lists[x][8],all_taflines_as_lists[x][9],all_taflines_as_lists[x][1
0],all_taflines_as_lists[x][11],all_taflines_as_lists[x][12],all_taflines_as_lists[x][13],fcst_wind_sp
d_capped[x], obs_wind_spd_capped[x], wind_spd_values_dif[x],
fcst_wind_spd_capped_binned[x],obs_wind_spd_capped_binned[x],wind_spd_values_binned_
dif[x],wind_spd_event_fcst[x],wind_spd_event_obs[x],wind_spd_event_capped_dif_given_fcst[
x],wind_spd_event_capped_dif_given_obs[x],wind_spd_event_capped_binned_dif_given_fcst[x
],wind_spd_event_capped_binned_dif_given_obs[x],wind_spd_event_identified_not_criteria_s
pecific[x],wind_spd_missed[x],wind_spd_false_alarm[x],fcst_wind_dir_capped[x],obs_wind_dir
_capped[x],wind_dir_values_dif[x],wind_dir_values_binned_dif[x]
,fcst_vis_capped[x],obs_vis_capped[x],vis_values_dif[x],fcst_vis_capped_binned[x],obs_vis_cap
ped_binned[x],vis_values_binned_dif[x],vis_event_fcst[x],vis_event_obs[x],vis_event_capped_d
if_given_fcst[x],vis_event_capped_dif_given_obs[x],vis_event_capped_binned_dif_given_fcst[x]
,vis_event_capped_binned_dif_given_obs[x],vis_event_identified_not_criteria_specific[x],vis_m
issed[x],vis_false_alarm[x],fcst_ceiling_capped[x],obs_ceiling_capped[x],ceiling_values_dif[x],fcs
t_ceiling_capped_binned[x],obs_ceiling_capped_binned[x],ceiling_values_binned_dif[x],ceiling_
event_fcst[x],ceiling_event_obs[x],ceiling_event_capped_dif_given_fcst[x],ceiling_event_cappe
d_dif_given_obs[x],ceiling_event_capped_binned_dif_given_fcst[x],ceiling_event_capped_binn
ed_dif_given_obs[x],ceiling_event_identified_not_criteria_specific[x],ceiling_missed[x],ceiling_f
alse_alarm[x],lightning_dif[x],lightning_given_fcst[x],lightning_given_obs[x],correct_month_day
_and_hour[x],original_correct_month_day_and_hour[x]]
      np.savetxt('dec1hopefullyallcompaired_humantaf_' + str(supercounter) +
'.csv',outf,fmt='%s',delimiter=',')
      supercounter=supercounter+1
#%%
   else:
      continue
      #supercounter=supercounter+1
```

TAF Polisher

```
# -*- coding: utf-8 -*-
```

```
"""
Created on Wed Oct 30 20:13:34 2019

@author: bbailey
"""

################################################## importing necessary libraries
import os
import numpy as np
import datetime
datestart=datetime.datetime.now()
################################################## defining directories and file names
directory = os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final
Code\2Dec_human_taf_complete_redo\all_main_and_all_tempo')
#regular lines=os.fsencode(r'I:[tempo_all_station_b'tion'.csv]
openlines2=[]
################################################## opening file name and checking
key information in file name and within file
supercounter=0
tempo_taf_lines=[]
main_taf_lines=[]
#open text file
starttime=datestart
main=open("All_station_compiled_nontempob'ions'.csv")
tempo=open("tempo_all_station_b'ions'.csv")

main_taf_file=main.readlines()
tempo_taf_file=tempo.readlines()

print(len(main_taf_file))
for x in range(len(main_taf_file)):
    mtl= main_taf_file[x].strip().split(',')
    main_taf_lines.append(mtl)
    tfl= tempo_taf_file[x].strip().split(',')
    tempo_taf_lines.append(tfl)

#print(len(main_taf_lines))
#print(main_taf_lines[1])

################################################################## set up arrays for
comparisons

indexes_for_tempo_lines_corresponding_to_main_lines=[]

for x in range(len(main_taf_lines)):
    for y in range(len(tempo_taf_lines)):
```

```
      if main_taf_lines[x][0]==tempo_taf_lines[y][0] and
main_taf_lines[x][1]==tempo_taf_lines[y][1] and main_taf_lines[x][2]==tempo_taf_lines[y][2]
and main_taf_lines[x][3]==tempo_taf_lines[y][3] and
main_taf_lines[x][4]==tempo_taf_lines[y][4] and main_taf_lines[x][5]==tempo_taf_lines[y][5]:
         indexes_for_tempo_lines_corresponding_to_main_lines.append(y)
         supercounter=supercounter+1
         break
   print('x= ',x, supercounter-1, 'continuing now ')

key_indexes=indexes_for_tempo_lines_corresponding_to_main_lines
print('yo',len(key_indexes))
print('start ', datestart, '      end ', datetime.datetime.now())
#%%
#####################################################################comparisons
print(len(key_indexes))
#print((key_indexes))
minicount=0
my_list=key_indexes
duplicatelines=[]
problemindexes=[]
problemicaos=[]
my_list.sort()
for i in range(0,len(my_list)-1):
           if my_list[i] == my_list[i+1]:
              minicount=minicount+1
              problemindexes.append(my_list[i])
              problemicaos.append(tempo_taf_lines[my_list[i]][1])
              duplicatelines.extend(tempo_taf_lines[my_list[i]])
              print (str(my_list[i]),'month ', tempo_taf_lines[my_list[i]][0], 'month ',
tempo_taf_lines[my_list[i]][1], tempo_taf_lines[my_list[i]][2], tempo_taf_lines[my_list[i]][3],
tempo_taf_lines[my_list[i]][4], tempo_taf_lines[my_list[i]][5]+ ' is a duplicate')
print(len(duplicatelines), duplicatelines[0:10])
key_indexes_experiment=key_indexes
print(len(key_indexes_experiment))
#for x in range(len(problemindexes)):
#    key_indexes_experiment.remove(problemindexes[x])
print(len(key_indexes_experiment))

key_indexes=key_indexes_experiment
#print(minicount)
#print(len(problemindexes))
#print(len(problemicaos))
#
#print (problemicaos)

max_wind_speed=[]
```

```
wind_direction=[]
vis=[]
cig=[]
lightning=[]
gusts=[]
windspacefiller=[]
#%%
for x in range(len(main_taf_lines)):
   potential_max_wind_speed=[]
   potential_wind_direction=[]
   potential_vis=[]
   potential_cig=[]
   potential_gusts=[]
   potential_lightning=[]
   potential_windspacefiller=[]


############################################################making lists for grouping
comparitive terms

   potential_wind_direction.extend((main_taf_lines[x][6],tempo_taf_lines[key_indexes[x]][6]))

potential_max_wind_speed.extend((main_taf_lines[x][7],main_taf_lines[x][8],tempo_taf_lines[
key_indexes[x]][7],tempo_taf_lines[key_indexes[x]][8]))
   potential_vis.extend((main_taf_lines[x][9],tempo_taf_lines[key_indexes[x]][9]))
   if min(potential_vis)=="-9999":
      potential_vis.remove("-9999")
   if min(potential_vis)=="-9999":
      potential_vis.remove("-9999")
   if len(potential_vis)==0:
      potential_vis.append((9999))

   potential_cig.extend((main_taf_lines[x][10],tempo_taf_lines[key_indexes[x]][10]))
   if min(potential_cig)=="-9999":
      potential_cig.remove("-9999")
   if min(potential_cig)=="-9999":
      potential_cig.remove("-9999")
   if len(potential_cig)==0:
      potential_cig.append((35000))
   potential_gusts.extend((main_taf_lines[x][8],tempo_taf_lines[key_indexes[x]][8]))
   potential_lightning.extend((main_taf_lines[x][11],tempo_taf_lines[key_indexes[x]][11]))
   potential_windspacefiller.extend((main_taf_lines[x][7],tempo_taf_lines[key_indexes[x]][7]))
   potential_winddirspacefiller=potential_wind_direction

   max_wind_speed.append(max(potential_max_wind_speed))
   if main_taf_lines[x][7] < tempo_taf_lines[key_indexes[x]][7]:
```

```
      wind_direction.append(tempo_taf_lines[key_indexes[x]][6])
   else:
      wind_direction.append(main_taf_lines[x][6])

   vis.append(min(potential_vis))
   cig.append(min(potential_cig))
   lightning.append(max(potential_lightning))
   gusts.append(max(potential_gusts))
   windspacefiller.append(max(potential_windspacefiller))
#%%
print(len(max_wind_speed), len(wind_direction), len(vis), len(cig), len(gusts), len(lightning),
len(windspacefiller), (len(main_taf_lines)))

outf=np.empty((len(main_taf_lines), 15), dtype=object)
for x in range(len(main_taf_lines)):
   outf[x,:]=[main_taf_lines[x][0], main_taf_lines[x][1], main_taf_lines[x][4],main_taf_lines[x][3],
main_taf_lines[x][5], max_wind_speed[x], wind_direction[x], cig[x], lightning[x], vis[x],
windspacefiller[x],wind_direction[x], gusts[x], wind_direction[x],max_wind_speed[x]]
#print(location,start_month,start_date,start_hour,outf)
#print(supercounter)
#print(outf.shape)
np.savetxt('final_all_human_tafs_wtempo.csv',outf,fmt='%s',delimiter=',')
np.savetxt('keyindexes_for_final_all_human_tafs_wtempo.csv',key_indexes,fmt='%s',delimiter=
',')
print(starttime,datetime.datetime.now())
#
##open text file
#f=open('KDAA_ 09April_09July.txt')
##read all the lines from the text file into a list
#g=f.readlines()
#alllines=[]
#nextline=[]
#location=str(f.name)[:4]
#
##run through all lines of the list 'g' which has all lines of the taf
#for x in range(len(g)):
#   #get rid of all whitespace and carriage returns then space delimit
#   a = g[x].strip().split(' ')
#
#
#   for xx in range(len(nextline)-1,-1,-1):
#      a.insert(0,nextline[xx])
#
#
#
#
```

```
#letter4= str(directory[-4:])
#for file in os.listdir(directory):
#    #print(filename)
#    #filename = str(file)
#
#    #print(len(directory))
#    if filename[0:5]!="tempo" and filename[-3:]!="txt" and filename[0:3]!="Fin":
##
#    if filename.endswith(".py") or filename.startswith("SAR"):
#        openfile = open(filename).readlines()
#        model_run_name=filename
#
#        for lines in openfile:
#            openlines.append(lines.split(','))
```

Ceiling Conditional

```
# -*- coding: utf-8 -*-
"""
Created on Sun Dec 22 14:09:04 2019
Ceiling Conditional
@author: Bailey
"""

################################################## importing necessary libraries
import os
import numpy as np
import datetime
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from scipy import stats
# importing the statistics module
from statistics import stdev

# importing frations as parameter values
from fractions import Fraction as fr
#import plotly.graph_objects as go
datestart=datetime.datetime.now()
#%%
################################################## defining directories and opening
files
directory = os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final Code\Images 4
Dec')
```

```
autotaf_lines=[]
humantaf_lines=[]
autotaf_lines = open('all_compaired_auto_0_4dec.csv').readlines()
humantaf_lines = open('all compaired_human_0_4dec.csv').readlines()

import numpy as np
def find_nearest(array, value):
   array = np.asarray(array)
   idx = (np.abs(array - value)).argmin()
   return idx
################################################## location information in file name
and within file
supercounter=0
stations=['BGTL','KADW','KBLV','KCEF','KDAA','KDOV','KFAF','KFFO','KFTK','KGTB','KGUS','KHOP','
KLFI','KMIB','KMTC','KMUI','KOFF','KRCA','KRDR','KVOK','KWRI']
times_hours_zulu=[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23]

all_autotaf_lines_as_lists=[]
all_humantaf_lines_as_lists=[]
################################################## get rid of all whitespace and
carriage returns then space delimit
for x in range(len(autotaf_lines)):
   autotaf_lines_as_lists = autotaf_lines[x].strip().split(',')
   all_autotaf_lines_as_lists.append(autotaf_lines_as_lists)
for x in range(len(humantaf_lines)):
   humantaf_lines_as_lists = humantaf_lines[x].strip().split(',')
   all_humantaf_lines_as_lists.append(humantaf_lines_as_lists)
##################################################

#%%
      ##################################################
####### Ceiling speed conditional prob
##################################################
#  prob(A) prob fcst =
count(all_humantaf_lines_as_lists[54]==1)/len(all_humantaf_lines_as_lists)
#  prob(B) prob obs =
count(all_humantaf_lines_as_lists[55]==1)/len(all_humantaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_humantaf_lines_as_lists[55]==1 and
all_humantaf_lines_as_lists[54])/ len(all_humantaf_lines_as_lists[55])
#  prob(B|A) prob observed given fcst = count(all_humantaf_lines_as_lists[55]==1 and
all_humantaf_lines_as_lists[54])/ len(all_humantaf_lines_as_lists[54])
#

#
# By Station 1 and up human
#
```

```
ceiling_prob_a_list=[]
ceiling_prob_b_list=[]
ceiling_prob_a_giv_b_list=[]
ceiling_prob_b_giv_a_list=[]
ceiling_prob_a_list_2nup=[]
ceiling_prob_b_list_2nup=[]
ceiling_prob_a_giv_b_list_2nup=[]
ceiling_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_humantaf_lines_as_lists)):
      if str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:
         eventcount=eventcount+1
         #print(all_humantaf_lines_as_lists[prob])
         #print(all_humantaf_lines_as_lists[prob][1])
         if float(all_humantaf_lines_as_lists[prob][54])==float(1):
            count_prob_a=count_prob_a+1
         if float(all_humantaf_lines_as_lists[prob][55])==float(1):
            count_prob_b=count_prob_b+1
         if float(all_humantaf_lines_as_lists[prob][55])==float(1) and
float(all_humantaf_lines_as_lists[prob][54])==float(1):
            count_prob_a_given_b=count_prob_a_given_b+1
            count_prob_b_given_a=count_prob_b_given_a+1

   prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
   prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
   prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
   prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
   ceiling_prob_a_list.append(prob_A)
   ceiling_prob_b_list.append(prob_B)
   ceiling_prob_a_giv_b_list.append(prob_A_giv_B)
   ceiling_prob_b_giv_a_list.append(prob_B_giv_A)
ceiling_prob_a_giv_b_list.sort()
ceiling_prob_b_giv_a_list.sort()
print(len(ceiling_prob_a_list), 'max',max((ceiling_prob_a_list)),'min',min((ceiling_prob_a_list)),
stdev(ceiling_prob_a_list) )
print(len(ceiling_prob_b_list), 'max',max((ceiling_prob_b_list)),'min',min((ceiling_prob_b_list)),
stdev(ceiling_prob_b_list)  )
```

```
print('ceiling a|b error bounds',' upper', ceiling_prob_a_giv_b_list[17], 'lower',
ceiling_prob_a_giv_b_list[3])
print('ceiling b|a error bounds',' upper', ceiling_prob_b_giv_a_list[17], 'lower',
ceiling_prob_b_giv_a_list[3])

for x in range(len(stations)):
   count_prob_a2= 0
   count_prob_b2= 0
   count_prob_a_given_b2= 0
   count_prob_b_given_a2= 0
   eventcount2=0
   for prob in range(len(all_humantaf_lines_as_lists)):
      if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
         eventcount2=eventcount2+1
         #print(all_autotaf_lines_as_lists[prob])
         #print(all_autotaf_lines_as_lists[prob][1])
         if float(all_humantaf_lines_as_lists[prob][51])>=float(2):
            count_prob_a2=count_prob_a2+1
         if float(all_humantaf_lines_as_lists[prob][52])>=float(2):
            count_prob_b2=count_prob_b2+1
         if float(all_humantaf_lines_as_lists[prob][52])>=float(2) and
float(all_humantaf_lines_as_lists[prob][51])>=float(2):
            count_prob_a_given_b2=count_prob_a_given_b2+1
            count_prob_b_given_a2=count_prob_b_given_a2+1
   prob_A2= ((count_prob_a2/(eventcount2))*100)
   prob_B2= ((count_prob_b2/(eventcount2))*100)
   prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
   prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

   ceiling_prob_a_list_2nup.append(prob_A2)
   ceiling_prob_b_list_2nup.append(prob_B2)
   ceiling_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
   ceiling_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)
ceiling_prob_a_giv_b_list_2nup.sort()
ceiling_prob_b_giv_a_list_2nup.sort()
#print(ceiling_prob_b_giv_a_list_2nup)

print(len(ceiling_prob_a_list_2nup),
'max',max((ceiling_prob_a_list_2nup)),'min',min((ceiling_prob_a_list_2nup)),
stdev(ceiling_prob_a_list_2nup) )
print(len(ceiling_prob_b_list_2nup),
'max',max((ceiling_prob_b_list_2nup)),'min',min((ceiling_prob_b_list_2nup)),
stdev(ceiling_prob_b_list_2nup)  )
print('human 2nup ceiling a|b error bounds',' upper', ceiling_prob_a_giv_b_list_2nup[17],
'lower', ceiling_prob_a_giv_b_list_2nup[3])
```

```
print('human 2nup ceiling b|a error bounds',' upper', ceiling_prob_b_giv_a_list_2nup[17],
'lower', ceiling_prob_b_giv_a_list_2nup[3])
#%%
      ##################################################
####### Ceiling speed conditional prob
##################################################
#  prob(A) prob fcst =
count(all_humantaf_lines_as_lists[54]==1)/len(all_humantaf_lines_as_lists)
#  prob(B) prob obs =
count(all_humantaf_lines_as_lists[55]==1)/len(all_humantaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_humantaf_lines_as_lists[55]==1 and
all_humantaf_lines_as_lists[54])/ len(all_humantaf_lines_as_lists[55])
#  prob(B|A) prob observed given fcst = count(all_humantaf_lines_as_lists[55]==1 and
all_humantaf_lines_as_lists[54])/ len(all_humantaf_lines_as_lists[54])

# overall human


ceiling_prob_a_list=[]
ceiling_prob_b_list=[]
ceiling_prob_a_giv_b_list=[]
ceiling_prob_b_giv_a_list=[]
ceiling_prob_a_list_2nup=[]
ceiling_prob_b_list_2nup=[]
ceiling_prob_a_giv_b_list_2nup=[]
ceiling_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_humantaf_lines_as_lists)):
      #if str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:
      eventcount=eventcount+1

      if float(all_humantaf_lines_as_lists[prob][54])==float(1):
         count_prob_a=count_prob_a+1
      if float(all_humantaf_lines_as_lists[prob][55])==float(1):
         count_prob_b=count_prob_b+1
      if float(all_humantaf_lines_as_lists[prob][55])==float(1) and
float(all_humantaf_lines_as_lists[prob][54])==float(1):
         count_prob_a_given_b=count_prob_a_given_b+1
         count_prob_b_given_a=count_prob_b_given_a+1
```

```
    prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
    prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
    prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
    prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
    ceiling_prob_a_list.append(prob_A)
    ceiling_prob_b_list.append(prob_B)
    ceiling_prob_a_giv_b_list.append(prob_A_giv_B)
    ceiling_prob_b_giv_a_list.append(prob_B_giv_A)

for x in range(len(stations)):
    count_prob_a2= 0
    count_prob_b2= 0
    count_prob_a_given_b2= 0
    count_prob_b_given_a2= 0
    eventcount2=0
    for prob in range(len(all_humantaf_lines_as_lists)):
        #if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
        eventcount2=eventcount2+1

        if float(all_humantaf_lines_as_lists[prob][51])>=float(2):
            count_prob_a2=count_prob_a2+1
        if float(all_humantaf_lines_as_lists[prob][52])>=float(2):
            count_prob_b2=count_prob_b2+1
        if float(all_humantaf_lines_as_lists[prob][52])>=float(2) and
float(all_humantaf_lines_as_lists[prob][51])>=float(2):
            count_prob_a_given_b2=count_prob_a_given_b2+1
            count_prob_b_given_a2=count_prob_b_given_a2+1
    prob_A2= ((count_prob_a2/(eventcount2))*100)
    prob_B2= ((count_prob_b2/(eventcount2))*100)
    prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
    prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

    ceiling_prob_a_list_2nup.append(prob_A2)
    ceiling_prob_b_list_2nup.append(prob_B2)
    ceiling_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
    ceiling_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)

print('Human overall','prob a',prob_A,'prob b',prob_B,'prob a|b',prob_A_giv_B,'prob
b|a',prob_B_giv_A)
print('Human overall 2nup','prob a',prob_A2,'prob b',prob_B2,'prob a|b',prob_A_giv_B2,'prob
b|a',prob_B_giv_A2)

#%%
```

```
#Auto


#%%
      ##################################################
####### Ceiling speed  bayes
##################################################
#  prob(A) prob fcst = count(all_autotaf_lines_as_lists[54]==1)/len(all_autotaf_lines_as_lists)
#  prob(B) prob obs = count(all_autotaf_lines_as_lists[55]==1)/len(all_autotaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_autotaf_lines_as_lists[55]==1 and
all_autotaf_lines_as_lists[54])/ len(all_autotaf_lines_as_lists[55])
#  prob(B|A) prob observed given fcst = count(all_autotaf_lines_as_lists[55]==1 and
all_autotaf_lines_as_lists[54])/ len(all_autotaf_lines_as_lists[54])
#

#
# By Station 1 and up auto
#
ceiling_prob_a_list=[]
ceiling_prob_b_list=[]
ceiling_prob_a_giv_b_list=[]
ceiling_prob_b_giv_a_list=[]
ceiling_prob_a_list_2nup=[]
ceiling_prob_b_list_2nup=[]
ceiling_prob_a_giv_b_list_2nup=[]
ceiling_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
   eventcount=0
   for prob in range(len(all_autotaf_lines_as_lists)):
      if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
         eventcount=eventcount+1
         #print(all_autotaf_lines_as_lists[prob])
         #print(all_autotaf_lines_as_lists[prob][1])
         if float(all_autotaf_lines_as_lists[prob][54])==float(1):
            count_prob_a=count_prob_a+1
         if float(all_autotaf_lines_as_lists[prob][55])==float(1):
            count_prob_b=count_prob_b+1
         if float(all_autotaf_lines_as_lists[prob][55])==float(1) and
float(all_autotaf_lines_as_lists[prob][54])==float(1):
            count_prob_a_given_b=count_prob_a_given_b+1
```

```
            count_prob_b_given_a=count_prob_b_given_a+1

    prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
    prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
    prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
    prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
    ceiling_prob_a_list.append(prob_A)
    ceiling_prob_b_list.append(prob_B)
    ceiling_prob_a_giv_b_list.append(prob_A_giv_B)
    ceiling_prob_b_giv_a_list.append(prob_B_giv_A)
ceiling_prob_a_giv_b_list.sort()
ceiling_prob_b_giv_a_list.sort()
print(len(ceiling_prob_a_list), 'max',max((ceiling_prob_a_list)),'min',min((ceiling_prob_a_list)),
stdev(ceiling_prob_a_list) )
print(len(ceiling_prob_b_list), 'max',max((ceiling_prob_b_list)),'min',min((ceiling_prob_b_list)),
stdev(ceiling_prob_b_list)  )
print('ceiling a|b error bounds',' upper', ceiling_prob_a_giv_b_list[17], 'lower',
ceiling_prob_a_giv_b_list[3])
print('ceiling b|a error bounds',' upper', ceiling_prob_b_giv_a_list[17], 'lower',
ceiling_prob_b_giv_a_list[3])

for x in range(len(stations)):
    count_prob_a2= 0
    count_prob_b2= 0
    count_prob_a_given_b2= 0
    count_prob_b_given_a2= 0
    eventcount2=0
    for prob in range(len(all_autotaf_lines_as_lists)):
        if str(all_autotaf_lines_as_lists[prob][0]).upper()==stations[x]:
            eventcount2=eventcount2+1
            #print(all_autotaf_lines_as_lists[prob])
            #print(all_autotaf_lines_as_lists[prob][1])
            if float(all_autotaf_lines_as_lists[prob][51])>=float(2):
                count_prob_a2=count_prob_a2+1
            if float(all_autotaf_lines_as_lists[prob][52])>=float(2):
                count_prob_b2=count_prob_b2+1
            if float(all_autotaf_lines_as_lists[prob][52])>=float(2) and
float(all_autotaf_lines_as_lists[prob][51])>=float(2):
                count_prob_a_given_b2=count_prob_a_given_b2+1
                count_prob_b_given_a2=count_prob_b_given_a2+1
    prob_A2= ((count_prob_a2/(eventcount2))*100)
    prob_B2= ((count_prob_b2/(eventcount2))*100)
    prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
    prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)
```

```
   ceiling_prob_a_list_2nup.append(prob_A2)
   ceiling_prob_b_list_2nup.append(prob_B2)
   ceiling_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
   ceiling_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)
ceiling_prob_a_giv_b_list_2nup.sort()
ceiling_prob_b_giv_a_list_2nup.sort()
#print(ceiling_prob_b_giv_a_list_2nup)

print(len(ceiling_prob_a_list_2nup),
'max',max((ceiling_prob_a_list_2nup)),'min',min((ceiling_prob_a_list_2nup)),
stdev(ceiling_prob_a_list_2nup) )
print(len(ceiling_prob_b_list_2nup),
'max',max((ceiling_prob_b_list_2nup)),'min',min((ceiling_prob_b_list_2nup)),
stdev(ceiling_prob_b_list_2nup)  )
print('auto 2nup ceiling a|b error bounds',' upper', ceiling_prob_a_giv_b_list_2nup[17], 'lower',
ceiling_prob_a_giv_b_list_2nup[3])
print('auto 2nup ceiling b|a error bounds',' upper', ceiling_prob_b_giv_a_list_2nup[17], 'lower',
ceiling_prob_b_giv_a_list_2nup[3])

      ###################################################
####### Ceiling speed
###################################################
#  prob(A) prob fcst = count(all_autotaf_lines_as_lists[54]==1)/len(all_autotaf_lines_as_lists)
#  prob(B) prob obs = count(all_autotaf_lines_as_lists[55]==1)/len(all_autotaf_lines_as_lists)
#  prob(A|B) prob fcst given event observed = count(all_autotaf_lines_as_lists[55]==1 and
all_autotaf_lines_as_lists[54])/ len(all_autotaf_lines_as_lists[55])
#  prob(B|A) prob observed given fcst = count(all_autotaf_lines_as_lists[55]==1 and
all_autotaf_lines_as_lists[54])/ len(all_autotaf_lines_as_lists[54])

# overall

ceiling_prob_a_list=[]
ceiling_prob_b_list=[]
ceiling_prob_a_giv_b_list=[]
ceiling_prob_b_giv_a_list=[]
ceiling_prob_a_list_2nup=[]
ceiling_prob_b_list_2nup=[]
ceiling_prob_a_giv_b_list_2nup=[]
ceiling_prob_b_giv_a_list_2nup=[]

for x in range(len(stations)):
   count_prob_a= 0
   count_prob_b= 0
   count_prob_a_given_b= 0
   count_prob_b_given_a= 0
```

```
    eventcount=0
    for prob in range(len(all_autotaf_lines_as_lists)):
        eventcount=eventcount+1

        if float(all_autotaf_lines_as_lists[prob][54])==float(1):
            count_prob_a=count_prob_a+1
        if float(all_autotaf_lines_as_lists[prob][55])==float(1):
            count_prob_b=count_prob_b+1
        if float(all_autotaf_lines_as_lists[prob][55])==float(1) and
float(all_autotaf_lines_as_lists[prob][54])==float(1):
            count_prob_a_given_b=count_prob_a_given_b+1
            count_prob_b_given_a=count_prob_b_given_a+1

    prob_A= ((count_prob_a/(eventcount))*100)
#    print(x,'', prob_A)
    prob_B= ((count_prob_b/(eventcount))*100)
#    print(x,'', prob_B)
    prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
    prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
    ceiling_prob_a_list.append(prob_A)
    ceiling_prob_b_list.append(prob_B)
    ceiling_prob_a_giv_b_list.append(prob_A_giv_B)
    ceiling_prob_b_giv_a_list.append(prob_B_giv_A)

for x in range(len(stations)):
    count_prob_a2= 0
    count_prob_b2= 0
    count_prob_a_given_b2= 0
    count_prob_b_given_a2= 0
    eventcount2=0
    for prob in range(len(all_autotaf_lines_as_lists)):
        eventcount2=eventcount2+1
        if float(all_autotaf_lines_as_lists[prob][51])>=float(2):
            count_prob_a2=count_prob_a2+1
        if float(all_autotaf_lines_as_lists[prob][52])>=float(2):
            count_prob_b2=count_prob_b2+1
        if float(all_autotaf_lines_as_lists[prob][52])>=float(2) and
float(all_autotaf_lines_as_lists[prob][51])>=float(2):
            count_prob_a_given_b2=count_prob_a_given_b2+1
            count_prob_b_given_a2=count_prob_b_given_a2+1
    prob_A2= ((count_prob_a2/(eventcount2))*100)
    prob_B2= ((count_prob_b2/(eventcount2))*100)
    prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
    prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

    ceiling_prob_a_list_2nup.append(prob_A2)
```

```
    ceiling_prob_b_list_2nup.append(prob_B2)
    ceiling_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
    ceiling_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)

print('Auto overall','prob a',prob_A,'prob b',prob_B,'prob a|b',prob_A_giv_B,'prob
b|a',prob_B_giv_A)
print('Auto overall 2nup','prob a',prob_A2,'prob b',prob_B2,'prob a|b',prob_A_giv_B2,'prob
b|a',prob_B_giv_A2)
```

WWA Code (non BC)

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 30 16:49:40 2019
```

```
@author: Bailey
"""

################################################## importing necessary libraries
import os
import numpy as np
import datetime
import matplotlib.pyplot as plt
#import plotly.graph_objects as go
datestart=datetime.datetime.now()

################################################## defining directories and opening
files
directory = os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final Code\Images 4
Dec\WWA')
wwa_lines=[]
wwa_lines = open('wwa_all_sations.csv').readlines()
################################################## location information in file name
and within file
supercounter=0
stations=['BGTL','KADW','KBLV','KCEF','KDAA','KDOV','KFAF','KFFO','KFTK','KGTB','KGUS','KHOP','
KLFI','KMIB','KMTC','KMUI','KOFF','KRCA','KRDR','KVOK','KWRI']
wwa_lines_as_lists=[]
all_wwa_lines_as_lists=[]
################################################## get rid of all whitespace and
carriage returns then space delimit

for x in range(len(wwa_lines)):
   wwa_lines_as_lists = wwa_lines[x].strip().split(',')
   all_wwa_lines_as_lists.append(wwa_lines_as_lists)
#%%
#
#   Moderate TS
#
mod_events_obs=0
mod_issued=0
mod_false_alarm=0
mod_hit=0
mod_miss=0
mod_extra_hrs=0
mod_extra_hrs2=0
#aggregated results
for x in range(len(all_wwa_lines_as_lists)):
   if all_wwa_lines_as_lists[x][1]=='Moderate Thunderstorms':
      mod_events_obs=mod_events_obs+int(all_wwa_lines_as_lists[x][2])
      mod_issued=mod_issued+int(all_wwa_lines_as_lists[x][3])
```

```
      mod_false_alarm=mod_false_alarm+int(all_wwa_lines_as_lists[x][9])
      mod_hit=mod_hit+int(all_wwa_lines_as_lists[x][6])
      mod_miss=mod_miss+int(all_wwa_lines_as_lists[x][8])+(int(all_wwa_lines_as_lists[x][7])-
int(all_wwa_lines_as_lists[x][6]))
   mod_extra_hrs=1440-(mod_issued)

print(mod_events_obs, mod_issued, mod_false_alarm, mod_hit, mod_miss, mod_extra_hrs)

A=mod_hit
B=mod_miss
C=mod_false_alarm
mod_extra_hrs2=1440-A-B-C
D=mod_extra_hrs2
d_experimental=D/10
print('A= ',A,' B= ',B,' C= ',C,' D= ',D)
#    print(all_wwa_lines_as_lists[x][0])
#    print(len(all_wwa_lines_as_lists))
#%%
# PPV (Criteria event Bias (A+C)/(A+B))
# PLR (sensitivity/(1-specificity))= (A/(A+B))/(1-(D/(C+D)))
PPV=Criteria_event_bias= ((A+C)/(A+B))
PLR=(A/(A+B))/(1-(D/(C+D)))
sensitivity=(A/(A+B))
specificity=((D/(C+D)))

PLR_exp=(A/(A+B))/(1-(d_experimental/(C+d_experimental)))
sensitivity_exp=(A/(A+B))
specificity_exp=((d_experimental/(C+d_experimental)))
print('PPV= ',PPV, 'PLR= ',PLR, 'Sensitivity= ',sensitivity, 'Specificity= ',specificity)
print('PLR exp= ',PLR_exp, 'Sensitivity exp= ',sensitivity_exp, 'Specificity exp= ',specificity_exp)

#%%
#
#    Lighting
#
lightinging_events_obs=0
lightinging_issued=0
lightinging_false_alarm=0
lightinging_hit=0
lightinging_miss=0
lightinging_extra_hrs=0
lightinging_extra_hrs2=0
#aggregated results
for x in range(len(all_wwa_lines_as_lists)):
   if all_wwa_lines_as_lists[x][1]=='Lightning':
      lightinging_events_obs=lightinging_events_obs+int(all_wwa_lines_as_lists[x][2])
```

```
        lightinging_issued=lightinging_issued+int(all_wwa_lines_as_lists[x][3])
        lightinging_false_alarm=lightinging_false_alarm+int(all_wwa_lines_as_lists[x][9])
        lightinging_hit=lightinging_hit+int(all_wwa_lines_as_lists[x][6])

lightinging_miss=lightinging_miss+int(all_wwa_lines_as_lists[x][8])+(int(all_wwa_lines_as_lists[x
][7])-int(all_wwa_lines_as_lists[x][6]))
    lightinging_extra_hrs=1440-(lightinging_issued)
print(lightinging_events_obs, lightinging_issued, lightinging_false_alarm, lightinging_hit,
lightinging_miss, lightinging_extra_hrs)

A=lightinging_hit
B=lightinging_miss
C=lightinging_false_alarm
lightinging_extra_hrs2=1440-A-B-C
D=lightinging_extra_hrs2
print('A= ',A,' B= ',B,' C= ',C,' D= ',D)
#%%
# PPV (Criteria event Bias (A+C)/(A+B))
# PLR (sensitivity/(1-specificity))= (A/(A+B))/(1-(D/(C+D)))
PPV=Criteria_event_bias= ((A+C)/(A+B))
PLR=(A/(A+B))/(1-(D/(C+D)))
sensitivity=(A/(A+B))
specificity=((D/(C+D)))

PLR_exp=(A/(A+B))/(1-(d_experimental/(C+d_experimental)))
sensitivity_exp=(A/(A+B))
specificity_exp=((d_experimental/(C+d_experimental)))
print('PPV= ',PPV, 'PLR= ',PLR, 'Sensitivity= ',sensitivity, 'Specificity= ',specificity)
print('PLR exp= ',PLR_exp, 'Sensitivity exp= ',sensitivity_exp, 'Specificity exp= ',specificity_exp)
```

Hourly BC values and error bar example

```
# -*- coding: utf-8 -*-
"""
Created on Sat Feb 22 01:47:40 2020

@author: Bailey
"""

# -*- coding: utf-8 -*-
"""
Created on Sun Jan 12 16:16:18 2020
```

```
@author: Bailey
"""

################################################## importing necessary libraries
import os
import numpy as np
import datetime
import matplotlib.pyplot as plt
#import plotly.graph_objects as go
datestart1=datetime.datetime.now()
print('heeeeeeey im Starting part 1',datetime.datetime.now())
################################################## defining directories and opening
files
directory = os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final Code\Images 4
Dec\utility error bars')
autotaf_lines=[]
humantaf_lines=[]
autotaf_lines = open('all_compaired_auto_0_4dec.csv').readlines()
humantaf_lines = open('all compaired_human_0_4dec.csv').readlines()
################################################## location information in file name
and within file
supercounter=0
stations=['BGTL','KADW','KBLV','KCEF','KDAA','KDOV','KFAF','KFFO','KFTK','KGTB','KGUS','KHOP','
KLFI','KMIB','KMTC','KMUI','KOFF','KRCA','KRDR','KVOK','KWRI']
all_autotaf_lines_as_lists=[]
all_humantaf_lines_as_lists=[]
################################################## get rid of all whitespace and
carriage returns then space delimit

for x in range(len(autotaf_lines)):
   autotaf_lines_as_lists = autotaf_lines[x].strip().split(',')
   all_autotaf_lines_as_lists.append(autotaf_lines_as_lists)
for x in range(len(humantaf_lines)):
   humantaf_lines_as_lists = humantaf_lines[x].strip().split(',')
   all_humantaf_lines_as_lists.append(humantaf_lines_as_lists)
#%%
########################################
###### Wind cost    ####################
########################################
cost0= 0
cost1= -1
cost2=-100
cost3= -500
cost4= -8000
cost5= -10000
```

```
cost6= -100
cost7= 100
cost8= -50
cost9= -400
cost10= -5000
cost11= -8000
cost12= -1000
cost13= -500
cost14= 1000
cost15= -500
cost16= -2500
cost17= -5000
cost18= -5000
cost19= -5000
cost20= 1500
cost21= 4000
cost22= -1000
cost23= -2500
cost24= -10000
cost25= -10000
cost26= 2000
cost27= 4000
cost28= 10000
cost29= 8000
cost30= -1000000
cost31= -1000000
cost32= 200000
cost33= 400000
cost34= 600000
cost35= 800000
hourlyvalue_list_human=[]
hourlyvalue_list_auto=[]

#%%
# Bootstrap method
import numpy as np

def bootstrap(data, n=1000, func=np.mean):
    """
    Generate `n` bootstrap samples, evaluating `func`
    at each resampling. `bootstrap` returns a function,
    which can be called to obtain confidence intervals
    of interest.
    """
    simulations = list()
    sample_size = len(data)
```

```
    xbar_init = np.mean(data)
    for c in range(n):
        itersample = np.random.choice(data, size=sample_size, replace=True)
        simulations.append(func(itersample))
    simulations.sort()
    def ci(p):
        """
        Return 2-sided symmetric confidence interval specified
        by p.
        """
        u_pval = (1+p)/2.
        l_pval = (1-u_pval)
        l_indx = int(np.floor(n*l_pval))
        u_indx = int(np.floor(n*u_pval))
        return(simulations[l_indx],simulations[u_indx])
    return(ci)
#%%
# human wind manual entry

a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,xxxx,y,z,aa,bb,cc,dd,ee,ff,gg,hh,ii,jj=0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
cat35count=0
cat34count=0
cat33count=0
hourlyvalue_list_human=[]

for number in range(68079):
    hourlyvalue_list_human.append(cost0)
    a=a+1

for number in range(8642):
    hourlyvalue_list_human.append(cost1)
    b=b+1
for number in range(2379):
    hourlyvalue_list_human.append(cost2)
    c=c+1

for number in range(294):
    hourlyvalue_list_human.append(cost3)
    d=d+1

for number in range(41):
    hourlyvalue_list_human.append(cost4)
    e=e+1

for number in range(3):
```

```
    hourlyvalue_list_human.append(cost5)
    f=f+1

for number in range(4759):
    hourlyvalue_list_human.append(cost6)
    g=g+1

for number in range(3694):
    hourlyvalue_list_human.append(cost7)
    h=h+1

for number in range(2480):
    hourlyvalue_list_human.append(cost8)
    i=i+1
for number in range(264):
    hourlyvalue_list_human.append(cost9)
    j=j+1
for number in range(37):
    hourlyvalue_list_human.append(cost10)
    k=k+1
for number in range(2):
    hourlyvalue_list_human.append(cost11)
    l=l+1
for number in range(627):
    hourlyvalue_list_human.append(cost12)
    m=m+1
for number in range(438):
    hourlyvalue_list_human.append(cost13)
    n=n+1
for number in range(680):
    hourlyvalue_list_human.append(cost14)
    o=o+1
for number in range(189):
    hourlyvalue_list_human.append(cost15)
    p=p+1
for number in range(34):
    hourlyvalue_list_human.append(cost16)
    q=q+1
for number in range(0):
    hourlyvalue_list_human.append(cost17)
    r=r+1
for number in range(104):
    hourlyvalue_list_human.append(cost18)
    s=s+1
for number in range(50):
    hourlyvalue_list_human.append(cost19)
```

```
    t=t+1
for number in range(61):
    hourlyvalue_list_human.append(cost20)
    u=u+1
for number in range(31):
    hourlyvalue_list_human.append(cost21)
    v=v+1
for number in range(13):
    hourlyvalue_list_human.append(cost22)
    w=w+1
for number in range(0):
    hourlyvalue_list_human.append(cost23)
    xxxx=xxxx+1
for number in range(11):
    hourlyvalue_list_human.append(cost24)
    y=y+1
for number in range(17):
    hourlyvalue_list_human.append(cost25)
    z=z+1
for number in range(6):
    hourlyvalue_list_human.append(cost26)
    aa=aa+1
for number in range(1):
    hourlyvalue_list_human.append(cost27)
    bb=bb+1
for number in range(3):
    hourlyvalue_list_human.append(cost28)
    cc=cc+1
for number in range(0):
    hourlyvalue_list_human.append(cost29)
    dd=dd+1
for number in range(5):
    hourlyvalue_list_human.append(cost30)
    ee=ee+1
for number in range(6):
    hourlyvalue_list_human.append(cost31)
    ff=ff+1
for number in range(3):
    hourlyvalue_list_human.append(cost32)
    gg=gg+1
for number in range(0):
    hourlyvalue_list_human.append(cost33)
    cat33count=cat33count+1
    #print(all_humantaf_lines_as_lists[x][0])
    hh=hh+1
for number in range(0):
```

```
    hourlyvalue_list_human.append(cost34)
    cat34count=cat34count+1

    ii=ii+1
for number in range(0):
    hourlyvalue_list_human.append(cost35)
    cat35count=cat35count+1
    jj=jj+1
print(len(hourlyvalue_list_human), len(all_humantaf_lines_as_lists),
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+xxxx+y+z+aa+bb+cc+dd+ee+ff+gg+hh+ii+j
j))



#%%
#auto wind manual entry
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,xxxx,y,z,aa,bb,cc,dd,ee,ff,gg,hh,ii,jj=0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
cat35count=0
cat34count=0
cat33count=0
hourlyvalue_list_auto=[]

for number in range(104450):
    hourlyvalue_list_auto.append(cost0)
    a=a+1

for number in range(8345):
    hourlyvalue_list_auto.append(cost1)
    b=b+1
for number in range(982):
    hourlyvalue_list_auto.append(cost2)
    c=c+1

for number in range(68):
    hourlyvalue_list_auto.append(cost3)
    d=d+1

for number in range(7):
    hourlyvalue_list_auto.append(cost4)
    e=e+1

for number in range(12):
    hourlyvalue_list_auto.append(cost5)
    f=f+1
```

```
for number in range(7827):
   hourlyvalue_list_auto.append(cost6)
   g=g+1
for number in range(5904):
   hourlyvalue_list_auto.append(cost7)
   h=h+1
for number in range(2119):
   hourlyvalue_list_auto.append(cost8)
   i=i+1
for number in range(123):
   hourlyvalue_list_auto.append(cost9)
   j=j+1
for number in range(14):
   hourlyvalue_list_auto.append(cost10)
   k=k+1
for number in range(4):
   hourlyvalue_list_auto.append(cost11)
   l=l+1
for number in range(1177):
   hourlyvalue_list_auto.append(cost12)
   m=m+1
for number in range(844):
   hourlyvalue_list_auto.append(cost13)
   n=n+1
for number in range(689):
   hourlyvalue_list_auto.append(cost14)
   o=o+1
for number in range(175):
   hourlyvalue_list_auto.append(cost15)
   p=p+1
for number in range(4):
   hourlyvalue_list_auto.append(cost16)
   q=q+1
for number in range(0):
   hourlyvalue_list_auto.append(cost17)
   r=r+1
for number in range(222):
   hourlyvalue_list_auto.append(cost18)
   s=s+1
for number in range(72):
   hourlyvalue_list_auto.append(cost19)
   t=t+1
for number in range(45):
   hourlyvalue_list_auto.append(cost20)
   u=u+1
for number in range(29):
```

```
    hourlyvalue_list_auto.append(cost21)
    v=v+1
for number in range(1):
    hourlyvalue_list_auto.append(cost22)
    w=w+1
for number in range(0):
    hourlyvalue_list_auto.append(cost23)
    xxxx=xxxx+1
for number in range(25):
    hourlyvalue_list_auto.append(cost24)
    y=y+1
for number in range(16):
    hourlyvalue_list_auto.append(cost25)
    z=z+1
for number in range(4):
    hourlyvalue_list_auto.append(cost26)
    aa=aa+1
for number in range(1):
    hourlyvalue_list_auto.append(cost27)
    bb=bb+1
for number in range(1):
    hourlyvalue_list_auto.append(cost28)
    cc=cc+1
for number in range(0):
    hourlyvalue_list_auto.append(cost29)
    dd=dd+1
for number in range(10):
    hourlyvalue_list_auto.append(cost30)
    ee=ee+1
for number in range(3):
    hourlyvalue_list_auto.append(cost31)
    ff=ff+1
for number in range(2):
    hourlyvalue_list_auto.append(cost32)
    gg=gg+1
for number in range(0):
    hourlyvalue_list_auto.append(cost33)
    cat33count=cat33count+1
    #print(all_autotaf_lines_as_lists[x][0])
    hh=hh+1
for number in range(0):
    hourlyvalue_list_auto.append(cost34)
    cat34count=cat34count+1

    ii=ii+1
for number in range(1):
```

```
   hourlyvalue_list_auto.append(cost35)
   cat35count=cat35count+1
   jj=jj+1
print(len(hourlyvalue_list_auto), len(all_autotaf_lines_as_lists),
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+xxxx+y+z+aa+bb+cc+dd+ee+ff+gg+hh+ii+j
j))

#%%
#Bootstrapping human winds

boot = bootstrap(hourlyvalue_list_human, n=10000)
cintervals = [boot(i) for i in (.90, .95)]
#cintervals = [boot(i) for i in (.95)]
print("10 thousand resamples human winds CI 90&95", cintervals)

#boot = bootstrap(hourlyvalue_list_human, n=5000)
#cintervals = [boot(i) for i in (.90, .95)]
##cintervals = [boot(i) for i in (.95)]
#print("5 thousand resamples", cintervals)
#%%
#Bootstrapping auto
boot = bootstrap(hourlyvalue_list_auto, n=15000)
cintervals = [boot(i) for i in (.90, .95)]
#cintervals = [boot(i) for i in (.95)]
print("15 thousand resamples auto winds CI 90&95", cintervals)

#boot = bootstrap(hourlyvalue_list_auto, n=5000)
#cintervals = [boot(i) for i in (.90, .95)]
##cintervals = [boot(i) for i in (.95)]
#print("5 thousand resamples", cintervals)
#%%
"""
Generate a histogram of the distribution of sample means for
10000 bootstrap resamplings.

For Human winds
"""
import numpy as np
import matplotlib
import seaborn as sns
sns.set(style="darkgrid")

#v = [10.3, 10.6, 11.7, 14.0, 14.2, 15.0, 16.8, 18.2, 21.3, 21.0]
xbar_init = np.round(np.mean(hourlyvalue_list_human), 2)

# generate 5000 resampled sample means  =>
```

```
means =
[np.mean(np.random.choice(hourlyvalue_list_human,size=len(hourlyvalue_list_human),replace
=True)) for i in range(10000)]
sns.distplot(means, color='r', kde=True, hist_kws=dict(edgecolor="b", linewidth=.675))
plt.xlabel("Initial Sample Mean: {}".format(xbar_init))
plt.title("BC Human Distribution of Sample Mean Wind 10k")
plt.axvline(x=xbar_init) # vertical line at xbar_init
plt.show()

"""
Generate a histogram of the distribution of sample means for
10000 bootstrap resamplings.

For Humans
"""
import numpy as np
import matplotlib
import seaborn as sns
sns.set(style="darkgrid")

xbar_init = np.round(np.mean(hourlyvalue_list_auto), 2)

# generate 5000 resampled sample means  =>
means =
[np.mean(np.random.choice(hourlyvalue_list_auto,size=len(hourlyvalue_list_auto),replace=Tru
e)) for i in range(15000)]
sns.distplot(means, color='r', kde=True, hist_kws=dict(edgecolor="b", linewidth=.675))
plt.xlabel("Initial Sample Mean: {}".format(xbar_init))
plt.title("BC Auto Distribution of Sample Mean Wind 15k")
plt.axvline(x=xbar_init) # vertical line at xbar_init
plt.show()
dateend1=datetime.datetime.now()
print(datetime.datetime.now())
##############################################################################
######################
############    cig
###########
#########################################################

#%%
# cig cost
cost0= 0
cost1= -1/75
cost2=-100/75
cost3= -500/75
cost4= -8000/75
```

```
cost5= -10000/75
cost6= -100/75
cost7= 100/75
cost8= -50/75
cost9= -400/75
cost10= -5000/75
cost11= -8000/75
cost12= -1000/75
cost13= -500/75
cost14= 1000/75
cost15= -500/75
cost16= -2500/75
cost17= -5000/75
cost18= -5000/75
cost19= -5000/75
cost20= 1500/75
cost21= 4000/75
cost22= -1000/75
cost23= -2500/75
cost24= -10000/75
cost25= -10000/75
cost26= 2000/75
cost27= 4000/75
cost28= 10000/75
cost29= 8000/75
cost30= -1000000/75
cost31= -1000000/75
cost32= 200000/75
cost33= 400000/75
cost34= 600000/75
cost35= 800000/75
hourlyvalue_list_human=[]
#%%
# human cig manual entry

a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,xxxx,y,z,aa,bb,cc,dd,ee,ff,gg,hh,ii,jj=0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
cat35count=0
cat34count=0
cat33count=0
hourlyvalue_list_human=[]

for number in range(66527):
    hourlyvalue_list_human.append(cost0)
    a=a+1
```

```
for number in range(7223):
   hourlyvalue_list_human.append(cost1)
   b=b+1
for number in range(2647):
   hourlyvalue_list_human.append(cost2)
   c=c+1

for number in range(542):
   hourlyvalue_list_human.append(cost3)
   d=d+1

for number in range(68):
   hourlyvalue_list_human.append(cost4)
   e=e+1

for number in range(137):
   hourlyvalue_list_human.append(cost5)
   f=f+1

for number in range(1609):
   hourlyvalue_list_human.append(cost6)
   g=g+1

for number in range(1369):
   hourlyvalue_list_human.append(cost7)
   h=h+1

for number in range(882):
   hourlyvalue_list_human.append(cost8)
   i=i+1
for number in range(155):
   hourlyvalue_list_human.append(cost9)
   j=j+1
for number in range(13):
   hourlyvalue_list_human.append(cost10)
   k=k+1
for number in range(10):
   hourlyvalue_list_human.append(cost11)
   l=l+1
for number in range(1542):
   hourlyvalue_list_human.append(cost12)
   m=m+1
for number in range(1610):
   hourlyvalue_list_human.append(cost13)
   n=n+1
for number in range(1918):
```

```
    hourlyvalue_list_human.append(cost14)
    o=o+1
for number in range(744):
    hourlyvalue_list_human.append(cost15)
    p=p+1
for number in range(27):
    hourlyvalue_list_human.append(cost16)
    q=q+1
for number in range(23):
    hourlyvalue_list_human.append(cost17)
    r=r+1
for number in range(750):
    hourlyvalue_list_human.append(cost18)
    s=s+1
for number in range(805):
    hourlyvalue_list_human.append(cost19)
    t=t+1
for number in range(1297):
    hourlyvalue_list_human.append(cost20)
    u=u+1
for number in range(1129):
    hourlyvalue_list_human.append(cost21)
    v=v+1
for number in range(106):
    hourlyvalue_list_human.append(cost22)
    w=w+1
for number in range(68):
    hourlyvalue_list_human.append(cost23)
    xxxx=xxxx+1
for number in range(159):
    hourlyvalue_list_human.append(cost24)
    y=y+1
for number in range(126):
    hourlyvalue_list_human.append(cost25)
    z=z+1
for number in range(271):
    hourlyvalue_list_human.append(cost26)
    aa=aa+1
for number in range(415):
    hourlyvalue_list_human.append(cost27)
    bb=bb+1
for number in range(96):
    hourlyvalue_list_human.append(cost28)
    cc=cc+1
for number in range(25):
    hourlyvalue_list_human.append(cost29)
```

```
   dd=dd+1
for number in range(218):
   hourlyvalue_list_human.append(cost30)
   ee=ee+1
for number in range(74):
   hourlyvalue_list_human.append(cost31)
   ff=ff+1
for number in range(119):
   hourlyvalue_list_human.append(cost32)
   gg=gg+1
for number in range(152):
   hourlyvalue_list_human.append(cost33)
   cat33count=cat33count+1
   #print(all_humantaf_lines_as_lists[x][0])
   hh=hh+1
for number in range(50):
   hourlyvalue_list_human.append(cost34)
   cat34count=cat34count+1

   ii=ii+1
for number in range(47):
   hourlyvalue_list_human.append(cost35)
   cat35count=cat35count+1
   jj=jj+1
print(len(hourlyvalue_list_human), len(all_humantaf_lines_as_lists),
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+xxxx+y+z+aa+bb+cc+dd+ee+ff+gg+hh+ii+j
j))



#%%
#auto cig manual entry
a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,xxxx,y,z,aa,bb,cc,dd,ee,ff,gg,hh,ii,jj=0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
cat35count=0
cat34count=0
cat33count=0
hourlyvalue_list_auto=[]

for number in range(96158):
   hourlyvalue_list_auto.append(cost0)
   a=a+1

for number in range(7337):
   hourlyvalue_list_auto.append(cost1)
   b=b+1
```

```
for number in range(3261):
   hourlyvalue_list_auto.append(cost2)
   c=c+1

for number in range(1508):
   hourlyvalue_list_auto.append(cost3)
   d=d+1

for number in range(376):
   hourlyvalue_list_auto.append(cost4)
   e=e+1

for number in range(459):
   hourlyvalue_list_auto.append(cost5)
   f=f+1

for number in range(1642):
   hourlyvalue_list_auto.append(cost6)
   g=g+1
for number in range(1957):
   hourlyvalue_list_auto.append(cost7)
   h=h+1
for number in range(1549):
   hourlyvalue_list_auto.append(cost8)
   i=i+1
for number in range(384):
   hourlyvalue_list_auto.append(cost9)
   j=j+1
for number in range(106):
   hourlyvalue_list_auto.append(cost10)
   k=k+1
for number in range(194):
   hourlyvalue_list_auto.append(cost11)
   l=l+1
for number in range(995):
   hourlyvalue_list_auto.append(cost12)
   m=m+1
for number in range(1495):
   hourlyvalue_list_auto.append(cost13)
   n=n+1
for number in range(2647):
   hourlyvalue_list_auto.append(cost14)
   o=o+1
for number in range(1823):
   hourlyvalue_list_auto.append(cost15)
   p=p+1
```

```
for number in range(529):
   hourlyvalue_list_auto.append(cost16)
   q=q+1
for number in range(1220):
   hourlyvalue_list_auto.append(cost17)
   r=r+1
for number in range(495):
   hourlyvalue_list_auto.append(cost18)
   s=s+1
for number in range(676):
   hourlyvalue_list_auto.append(cost19)
   t=t+1
for number in range(1167):
   hourlyvalue_list_auto.append(cost20)
   u=u+1
for number in range(2447):
   hourlyvalue_list_auto.append(cost21)
   v=v+1
for number in range(385):
   hourlyvalue_list_auto.append(cost22)
   w=w+1
for number in range(1340):
   hourlyvalue_list_auto.append(cost23)
   xxxx=xxxx+1
for number in range(244):
   hourlyvalue_list_auto.append(cost24)
   y=y+1
for number in range(196):
   hourlyvalue_list_auto.append(cost25)
   z=z+1
for number in range(183):
   hourlyvalue_list_auto.append(cost26)
   aa=aa+1
for number in range(703):
   hourlyvalue_list_auto.append(cost27)
   bb=bb+1
for number in range(110):
   hourlyvalue_list_auto.append(cost28)
   cc=cc+1
for number in range(458):
   hourlyvalue_list_auto.append(cost29)
   dd=dd+1
for number in range(426):
   hourlyvalue_list_auto.append(cost30)
   ee=ee+1
for number in range(75):
```

```
    hourlyvalue_list_auto.append(cost31)
    ff=ff+1
for number in range(96):
    hourlyvalue_list_auto.append(cost32)
    gg=gg+1
for number in range(252):
    hourlyvalue_list_auto.append(cost33)
    cat33count=cat33count+1
    #print(all_autotaf_lines_as_lists[x][0])
    hh=hh+1
for number in range(130):
    hourlyvalue_list_auto.append(cost34)
    cat34count=cat34count+1

    ii=ii+1
for number in range(153):
    hourlyvalue_list_auto.append(cost35)
    cat35count=cat35count+1
    jj=jj+1
print(len(hourlyvalue_list_auto), len(all_autotaf_lines_as_lists),
(a+b+c+d+e+f+g+h+i+j+k+l+m+n+o+p+q+r+s+t+u+v+w+xxxx+y+z+aa+bb+cc+dd+ee+ff+gg+hh+ii+j
j))


#%%
#Bootstrapping human

boot = bootstrap(hourlyvalue_list_human, n=10000)
cintervals = [boot(i) for i in (.90, .95)]
#cintervals = [boot(i) for i in (.95)]
print("10 thousand resamples human cig CI  90&95", cintervals)

#boot = bootstrap(hourlyvalue_list_human, n=5000)
#cintervals = [boot(i) for i in (.90, .95)]
##cintervals = [boot(i) for i in (.95)]
#print("5 thousand resamples", cintervals)

#Bootstrapping auto
boot = bootstrap(hourlyvalue_list_auto, n=15000)
cintervals = [boot(i) for i in (.90, .95)]
#cintervals = [boot(i) for i in (.95)]
print("15 thousand resamples auto cig CI 90&95", cintervals)

#boot = bootstrap(hourlyvalue_list_auto, n=5000)
#cintervals = [boot(i) for i in (.90, .95)]
##cintervals = [boot(i) for i in (.95)]
```

```
#print("5 thousand resamples", cintervals)
#%%
"""
Generate a histogram of the distribution of sample means for
5000 bootstrap resamplings.

For Human cig
"""
import numpy as np
import matplotlib
import seaborn as sns
sns.set(style="darkgrid")

#v = [10.3, 10.6, 11.7, 14.0, 14.2, 15.0, 16.8, 18.2, 21.3, 21.0]
xbar_init = np.round(np.mean(hourlyvalue_list_human), 2)

# generate 5000 resampled sample means  =>
means =
[np.mean(np.random.choice(hourlyvalue_list_human,size=len(hourlyvalue_list_human),replace
=True)) for i in range(10000)]
sns.distplot(means, color='r', kde=True, hist_kws=dict(edgecolor="b", linewidth=.675))
plt.xlabel("Initial Sample Mean: {}".format(xbar_init))
plt.title("BC Human Distribution of Sample Mean Ceiling 10k")
plt.axvline(x=xbar_init) # vertical line at xbar_init
plt.show()

"""
Generate a histogram of the distribution of sample means for
5000 bootstrap resamplings.

For auto cig
"""
import numpy as np
import matplotlib
import seaborn as sns
sns.set(style="darkgrid")

xbar_init = np.round(np.mean(hourlyvalue_list_auto), 2)

# generate 5000 resampled sample means  =>
means =
[np.mean(np.random.choice(hourlyvalue_list_auto,size=len(hourlyvalue_list_auto),replace=Tru
e)) for i in range(15000)]
sns.distplot(means, color='r', kde=True, hist_kws=dict(edgecolor="b", linewidth=.675))
plt.xlabel("Initial Sample Mean: {}".format(xbar_init))
plt.title("BC Auto Distribution of Sample Mean Ceiling 15k")
```

```
plt.axvline(x=xbar_init) # vertical line at xbar_init
plt.show()
dateend2=datetime.datetime.now()
print(datetime.datetime.now())
```

```
# -*- coding: utf-8 -*-

"""

Created on Sun Feb 23 08:30:25 2020



@author: Bailey

"""



hourlyvalue_list_human=[]

hourlyvalue_list_auto=[]



modTS=[75000,	75000, 75000, 75000, 75000, 75000, 75000, 75000, 75000, 75000,

	75000, 75000, 75000, 75000, 75000, 75000, -7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,

	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-7.5,	-

75000, -75000,	-75000,	-75000,	-75000,	-75000,	-

75000, -75000,	-75000,	-75000,	-75000,	-75000,	-
```

75000, -75000, -75000, -75000, -75000, -75000, -75000, -75000, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

```
        1,      1,      1,      1,      1,      1,      1,      1,      1,      1,      1,
        1,      1,      1,      1,      1,      1,      1,      1,      1,      1,      1,
        1,      1,      1,      1,      1,      1,      1,      1,      1,      1,      1,
        1,      1,      1,      1,      1,      1,      1,      1,      1,      1,      1,
        1,      1,      1,      1,      1,      1,      1,      1,      1,      1,      1,
        1,      1,      1,      1,      1,      1,      1,      1,      1,      1,      1,
        1,      1,      1,      1,      1]
lightning=[10000,       10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
        10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
```

10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000, 10000,
10000, 10000, 10000, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -10000, -10000, -10000,
-10000, -10000, -10000, -10000, -10000, -
10000, -10000, -10000, -10000, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,

0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,

0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,

0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,

0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,

```
        0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,
        0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01,  0.01]
#%%
# Bootstrap method
import numpy as np


def bootstrap(data, n=1000, func=np.mean):
    """
    Generate `n` bootstrap samples, evaluating `func`
    at each resampling. `bootstrap` returns a function,
    which can be called to obtain confidence intervals
    of interest.
    """
    simulations = list()
    sample_size = len(data)
    xbar_init = np.mean(data)
    for c in range(n):
        itersample = np.random.choice(data, size=sample_size, replace=True)
        simulations.append(func(itersample))
    simulations.sort()
    def ci(p):
        """
        Return 2-sided symmetric confidence interval specified
```

```
        by p.
        """
        u_pval = (1+p)/2.
        l_pval = (1-u_pval)
        l_indx = int(np.floor(n*l_pval))
        u_indx = int(np.floor(n*u_pval))
        return(simulations[l_indx],simulations[u_indx])
    return(ci)

boot = bootstrap(modTS, n=100)
cintervals = [boot(i) for i in (.90, .95)]
#cintervals = [boot(i) for i in (.95)]
print("10 thousand resamples human cig CI  90&95", cintervals)

boot = bootstrap(lightning, n=100)
cintervals = [boot(i) for i in (.90, .95)]
#cintervals = [boot(i) for i in (.95)]
print("10 thousand resamples human cig CI  90&95", cintervals)

10 thousand resamples human cig CI  90&95 [(-780.6774305555556,
417.1048611111111), (-832.790625, 469.26354166666664)]
10 thousand resamples human cig CI  90&95 [(1333.2870972222222,
1624.9387986111108), (1305.4981249999998, 1638.8347013888888)]
```

```
#%%
"""
Generate a histogram of the distribution of sample means for
10000 bootstrap resamplings.

For Human winds
"""
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(style="darkgrid")

#v = [10.3, 10.6, 11.7, 14.0, 14.2, 15.0, 16.8, 18.2, 21.3, 21.0]
xbar_init = np.round(np.mean(modTS), 2)

# generate 5000 resampled sample means  =>
means = [np.mean(np.random.choice(modTS,size=len(modTS),replace=True)) for i in
range(100)]
sns.distplot(means, color='r', kde=True, hist_kws=dict(edgecolor="b", linewidth=.675))
plt.xlabel("Initial Sample Mean: {}".format(xbar_init))
plt.title("BC Moderate Thunderstorm Distribution of Sample Mean")
plt.axvline(x=xbar_init) # vertical line at xbar_init
```

```
plt.show()

"""
Generate a histogram of the distribution of sample means for
10000 bootstrap resamplings.

For Humans
"""
import numpy as np
import matplotlib
import seaborn as sns
sns.set(style="darkgrid")

xbar_init = np.round(np.mean(lightning), 2)

# generate 5000 resampled sample means  =>
means = [np.mean(np.random.choice(lightning,size=len(lightning),replace=True)) for i in
range(100)]
sns.distplot(means, color='r', kde=True, hist_kws=dict(edgecolor="b", linewidth=.675))
plt.xlabel("Initial Sample Mean: {}".format(xbar_init))
plt.title("BC Lightning Distribution of Sample Mean")
plt.axvline(x=xbar_init) # vertical line at xbar_init
plt.show()
```

```
##########################################################################

############################

############  cig
```

Code containing Matrix

```
# -*- coding: utf-8 -*-
"""
Created on Wed Nov  6 10:33:57 2019

@author: bbailey
"""

"""
Created on Tue Oct  8 14:35:32 2019

@author: bbailey
"""

################################################## importing
necessary libraries
import os
import numpy as np
import datetime
import matplotlib.pyplot as plt
#import plotly.graph_objects as go
datestart=datetime.datetime.now()

################################################## defining
directories and opening files
directory =
os.fsencode(r'C:\Users\Bailey\Desktop\AFIT\Thesis\python\Final
Code')
autotaf_lines=[]
humantaf_lines=[]
autotaf_lines = open('all compaired_autotaf_0.csv').readlines()
humantaf_lines = open('all compaired_human_0.csv').readlines()
################################################## location
information in file name and within file
supercounter=0
#stations=['BGTL','KADW','KBLV','KCEF','KDAA','KDOV','KFAF','KFFO','
KFTK','KGTB','KGUS','KHOP','KLFI','KMIB','KMTC','KMUI','KOFF','KRCA'
,'KRDR','KVOK','KWRI']
stations=['BGTL','KADW','KBLV','KCEF','KDAA','KDOV','KFAF','KFFO','K
FTK','KGTB','KGUS','KHOP','KLFI','KMIB','KMTC','KMUI','KOFF','KRCA',
'KVOK','KWRI']
wind_prob_a_list=[]
```

```
wind_prob_b_list=[]
wind_prob_a_giv_b_list=[]
wind_prob_b_giv_a_list=[]
wind_prob_a_list_2nup=[]
wind_prob_b_list_2nup=[]
wind_prob_a_giv_b_list_2nup=[]
wind_prob_b_giv_a_list_2nup=[]
wind_error=[]
wind_error_given_fcst=[]
wind_error_given_fcst=[]
wind_binned_dif=[]
line_0ag=[0,0,0,0,]
wind_raw_dif=[]
wind_dif_given_event=[]
vis_prob_a_list=[]
vis_prob_b_list=[]
vis_prob_a_giv_b_list=[]
vis_prob_b_giv_a_list=[]
vis_prob_a_list_2nup=[]
vis_prob_b_list_2nup=[]
vis_prob_a_giv_b_list_2nup=[]
vis_prob_b_giv_a_list_2nup=[]
cig_prob_a_list=[]
cig_prob_b_list=[]
cig_prob_a_giv_b_list=[]
cig_prob_b_giv_a_list=[]
cig_prob_a_list_2nup=[]
cig_prob_b_list_2nup=[]
cig_prob_a_giv_b_list_2nup=[]
cig_prob_b_giv_a_list_2nup=[]
all_autotaf_lines_as_lists=[]
all_humantaf_lines_as_lists=[]
################################################## get rid of all
whitespace and carriage returns then space delimit

for x in range(len(autotaf_lines)):
    autotaf_lines_as_lists = autotaf_lines[x].strip().split(',')
    all_autotaf_lines_as_lists.append(autotaf_lines_as_lists)
for x in range(len(humantaf_lines)):
    humantaf_lines_as_lists = humantaf_lines[x].strip().split(',')
    all_humantaf_lines_as_lists.append(humantaf_lines_as_lists)
############################################################
############################################################
#########      Making Contingency Tables and  #################
#########      And conditional Probabilities  #################
############################################################
############################################################
#


#%%
#lines=[[line_0],[line_1],[line_2],[line_3],[line_4],[line_5]]
```

```
lines=[]
line_0ag=np.zeros((1,6), dtype=int)
line_1ag=np.zeros((1,6), dtype=int)
line_2ag=np.zeros((1,6), dtype=int)
line_3ag=np.zeros((1,6), dtype=int)
line_4ag=np.zeros((1,6), dtype=int)
line_5ag=np.zeros((1,6), dtype=int)
windspeed_contingency_table=np.zeros((18,7), dtype=object)
for x in range(len(stations)):
    fcst_cols_in_con_table=6
    obs_cols_in_con_table=6
    line_0=[]
    line_1=[]
    line_2=[]
    line_3=[]
    line_4=[]
    line_5=[]
    percentline_0=[]
    percentline_1=[]
    percentline_2=[]
    percentline_3=[]
    percentline_4=[]
    percentline_5=[]



    uninteresting_events=0
    interesting_event_instances=0
    linesperstation=0

    #counts=[0,0,0,0,0]

    for xxx in range(fcst_cols_in_con_table):
        count0=0
        count1=0
        count2=0
        count3=0
        count4=0
        count5=0
        percentcount0=0
        percentcount1=0
        percentcount2=0
        percentcount3=0
        percentcount4=0
        percentcount5=0
        for xx in range(len(all_humantaf_lines_as_lists)):
            if
str(all_humantaf_lines_as_lists[xx][0]).upper()==stations[x]:
                linesperstation=linesperstation+1
                if
int(float(all_humantaf_lines_as_lists[xx][17]))==int(xxx):
```

```
                    if
int(float(all_humantaf_lines_as_lists[xx][18]))==int(0):
                        count0=count0+1
#                     if
int(float(all_humantaf_lines_as_lists[xx][17]))==0 and
int(float(all_humantaf_lines_as_lists[xx][18]))==0:
#                         uninteresting_events=uninteresting_events+1
                    elif
int(float(all_humantaf_lines_as_lists[xx][18]))==int(1):
                        count1=count1+1
                    elif
int(float(all_humantaf_lines_as_lists[xx][18]))==int(2):
                        count2=count2+1
                    elif
int(float(all_humantaf_lines_as_lists[xx][18]))==int(3):
                        count3=count3+1
                    elif
int(float(all_humantaf_lines_as_lists[xx][18]))==int(4):
                        count4=count4+1
                    elif
int(float(all_humantaf_lines_as_lists[xx][18]))==int(5):
                        count5=count5+1
#                 elif
float(all_humantaf_lines_as_lists[xx][17])==int(xxx):
#                     count1=count1+1
#                 elif
float(all_humantaf_lines_as_lists[xx][17])==int(xxx):
#                     count2=count2+1
#                 elif
float(all_humantaf_lines_as_lists[xx][17])==int(xxx):
#                     count3=count3+1
#                 elif
float(all_humantaf_lines_as_lists[xx][17])==int(xxx):
#                     count4=count4+1
#                 elif
float(all_humantaf_lines_as_lists[xx][17])==int(xxx):
#                     count5=count5+1
        #print(count0,count1,count2,count3,count4,count5)
#
interesting_event_instances=count1+count2+count3+count4+count5
#         for x in len(6):
#             if x==0
#                 percentcount0=0
#                 percentcount1=count1/interesting_event_instances
#                 percentcount2=0
#                 percentcount3=0
#                 percentcount4=0
#                 percentcount5=0
#             else:
#                 percentcount0=0
#                 percentcount1=0
#                 percentcount2=0
```

```
#                 percentcount3=0
#                 percentcount4=0
#                 percentcount5=0

line_0.append(count0),line_1.append(count1),line_2.append(count2),li
ne_3.append(count3),line_4.append(count4),line_5.append(count5)
    line_0ag=line_0ag+np.asarray(line_0)
    line_1ag=line_1ag+np.asarray(line_1)
    line_2ag=line_2ag+np.asarray(line_2)
    line_3ag=line_3ag+np.asarray(line_3)
    line_4ag=line_4ag+np.asarray(line_4)
    line_5ag=line_5ag+np.asarray(line_5)
        #interesting_event_instances=linesperstation-
uninteresting_events

#print(linesperstation,uninteresting_events,interesting_event_instan
ces)
#    lines.append(line_0_array)
#    lines.append(line_1)
#    lines.append(line_2)
#    lines.append(line_3)
#    lines.append(line_4)
#    lines.append(line_5)
    print(line_0ag,'im an aggregate')
    print(line_1ag,'im an aggregate2')
    print(line_2ag,'im an aggregate3')
    print(line_3ag,'im an aggregate4')
    print(line_4ag,'im an aggregate5')
    print(line_5ag,'im an aggregate6')


#    print(line_0)
#    print(line_1)
#    print(line_2)
#    print(line_3)
#    print(line_4)
#    print(line_5)
#%%

total_events=sum(line_0[:]+line_1[:]+line_2[:]+line_3[:]+line_4[:]+l
ine_5[:])
    uninteresting_events=line_0[0]
    interesting_events=total_events-uninteresting_events
    typetag='Human'
    percent_interesting= interesting_events/total_events
    print(stations[x],'total events= ', total_events, 'interesting
events= ', interesting_events,
'percent_interesting',str(percent_interesting))
    infoline=[stations[x], typetag, total_events,
interesting_events, uninteresting_events, percent_interesting,'fil']
```

```
frameline_hor=['Events','FCST_0','FCST_1','FCST_2','FCST_3','FCST_4'
,'FCST_5']

frameline_hor2=['Percentage','FCST_0','FCST_1','FCST_2','FCST_3','FC
ST_4','FCST_5']
    ################################################
####### Wind speed  bayes
################################################
#  prob(A) prob fcst =
count(all_humantaf_lines_as_lists[20]==1)/len(all_humantaf_lines_as_
lists)
#  prob(B) prob obs =
count(all_humantaf_lines_as_lists[21]==1)/len(all_humantaf_lines_as_
lists)
#  prob(A|B) prob fcst given event observed =
count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/
len(all_humantaf_lines_as_lists[21])
#  prob(B|A) prob observed given fcst =
count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/
len(all_humantaf_lines_as_lists[20])
#
#  Prob(A|B)= P(B|A) P(A)
#             ------------
#                P(B)
#

#




    count_prob_a= 0
    count_prob_b= 0
    count_prob_a_given_b= 0
    count_prob_b_given_a= 0
    count_prob_a2= 0
    count_prob_b2= 0
    count_prob_a_given_b2= 0
    count_prob_b_given_a2= 0

#for x in range(len(stations)):
    for prob in range(len(all_humantaf_lines_as_lists)):
        if
str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:
            #print(all_humantaf_lines_as_lists[prob])
            #print(all_humantaf_lines_as_lists[prob][1])
```

```
            if
float(all_humantaf_lines_as_lists[prob][20])==float(1):
                count_prob_a=count_prob_a+1
            if
float(all_humantaf_lines_as_lists[prob][21])==float(1):
                count_prob_b=count_prob_b+1
            if
float(all_humantaf_lines_as_lists[prob][21])==float(1) and
float(all_humantaf_lines_as_lists[prob][20])==float(1):
                count_prob_a_given_b=count_prob_a_given_b+1
                count_prob_b_given_a=count_prob_b_given_a+1

    prob_A= ((count_prob_a/len(all_humantaf_lines_as_lists))*100)
    prob_B= ((count_prob_b/len(all_humantaf_lines_as_lists))*100)
    prob_A_giv_B= (((count_prob_a_given_b)/count_prob_b) *100)
    prob_B_giv_A= (((count_prob_b_given_a)/count_prob_a) *100)
    wind_prob_a_list.append(prob_A*100)
    wind_prob_b_list.append(prob_B*100)
    wind_prob_a_giv_b_list.append(prob_A_giv_B)
    wind_prob_b_giv_a_list.append(prob_B_giv_A)


    for prob in range(len(all_humantaf_lines_as_lists)):
        if
str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:
            #print(all_humantaf_lines_as_lists[prob])
            #print(all_humantaf_lines_as_lists[prob][1])
            if
float(all_humantaf_lines_as_lists[prob][17])>=float(2):
                count_prob_a2=count_prob_a2+1
            if
float(all_humantaf_lines_as_lists[prob][18])>=float(2):
                count_prob_b2=count_prob_b2+1
            if
float(all_humantaf_lines_as_lists[prob][18])>=float(2) and
float(all_humantaf_lines_as_lists[prob][17])>=float(2):
                count_prob_a_given_b2=count_prob_a_given_b2+1
                count_prob_b_given_a2=count_prob_b_given_a2+1
    prob_A2= ((count_prob_a2/len(all_humantaf_lines_as_lists))*100)
    prob_B2= ((count_prob_b2/len(all_humantaf_lines_as_lists))*100)
    prob_A_giv_B2= (((count_prob_a_given_b2)/count_prob_b2) *100)
    prob_B_giv_A2= (((count_prob_b_given_a2)/count_prob_a2) *100)

    wind_prob_a_list_2nup.append(prob_A2*100)
    wind_prob_b_list_2nup.append(prob_B2*100)
    wind_prob_a_giv_b_list_2nup.append(prob_A_giv_B2)
    wind_prob_b_giv_a_list_2nup.append(prob_B_giv_A2)
    newline_0=[(x/interesting_events*100) for x in line_0]
    newline_0_fin=newline_0
    newline_0_fin.insert(0,'OBS_0')
    newline_1=[(x/interesting_events*100) for x in line_1]
    newline_1_fin=newline_1
```

```
    newline_1_fin.insert(0,'OBS_1')
    newline_2=[(x/interesting_events*100) for x in line_2]
    newline_2_fin=newline_2
    newline_2_fin.insert(0,'OBS_2')
    newline_3=[(x/interesting_events*100) for x in line_3]
    newline_3_fin=newline_3
    newline_3_fin.insert(0, 'OBS_3')
    newline_4=[(x/interesting_events*100) for x in line_4]
    newline_4_fin=newline_4
    newline_4_fin.insert(0,'OBS_4')
    newline_5=[(x/interesting_events*100) for x in line_5]
    newline_5_fin=newline_5
    newline_5_fin.insert(0,'OBS_5')

    print('paaaaaaaaaaaaaaaaaaaayyyyyyyyyyyyy_attention',line_0)
    line_0_fin=line_0
    line_0_fin.insert(0,'OBS_0')
    print('paaaaaaaaaaaaaaaaaaaayyyyyyyyyyyyy_attention',line_0_fin)
    line_1_fin=line_1
    line_1_fin.insert(0,'OBS_1')
    line_2_fin=line_2
    line_2_fin.insert(0,'OBS_2')
    line_3_fin=line_3
    line_3_fin.insert(0,'OBS_3')
    line_4_fin=line_4
    line_4_fin.insert(0,'OBS_4')
    line_5_fin=line_5
    line_5_fin.insert(0,'OBS_5')
    conditional_intel_line=['Conditional info (non cat)', 'prob A
(fcst)',prob_A*100, prob_B*100, 'prob a (given B)', prob_A_giv_B,
prob_B_giv_A]
    conditional_intel_line2=['Conditional info (cat 2 and up)',
'prob A (fcst)',prob_A2*100, prob_B2*100, 'prob A (given B)',
(prob_A_giv_B2), prob_B_giv_A2]

#    for s in range(7):
#
#        if s==0:
#            percent_windspeed_contingency_table[s,:]=infoline
#        elif s==1:
#            percent_windspeed_contingency_table[s,:]=newline_0
#        elif s==2:
#            percent_windspeed_contingency_table[s,:]=newline_1
#        elif s==3:
#            percent_windspeed_contingency_table[s,:]=newline_2
#        elif s==4:
#            percent_windspeed_contingency_table[s,:]=newline_3
#        elif s==5:
#            percent_windspeed_contingency_table[s,:]=newline_4
#        elif s==6:
#            percent_windspeed_contingency_table[s,:]=newline_5
#        continue
```

```
frameline_vert=['','','OBS_0','OBS_1','OBS_2','OBS_3','OBS_4','OBS_5
','','' ,'OBS_0','OBS_1','OBS_2','OBS_3','OBS_4','OBS_5','','']

    for s in range(18):

        if s==0:
            windspeed_contingency_table[s,:]=infoline
        elif s==1:
            windspeed_contingency_table[s,:]=frameline_hor
        elif s==2:
            windspeed_contingency_table[s,:]=line_0_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==3:
            windspeed_contingency_table[s,:]=line_1_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==4:
            windspeed_contingency_table[s,:]=line_2_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==5:
            windspeed_contingency_table[s,:]=line_3_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==6:
            windspeed_contingency_table[s,:]=line_4_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==7:
            windspeed_contingency_table[s,:]=line_5_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==8:
            windspeed_contingency_table[s,:]=infoline
        elif s==9:
            windspeed_contingency_table[s,:]=frameline_hor2
        elif s==10:
            windspeed_contingency_table[s,:]=newline_0_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==11:
            windspeed_contingency_table[s,:]=newline_1_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==12:
            windspeed_contingency_table[s,:]=newline_2_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==13:
```

```
            windspeed_contingency_table[s,:]=newline_3_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==14:
            windspeed_contingency_table[s,:]=newline_4_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==15:
            windspeed_contingency_table[s,:]=newline_5_fin

#windspeed_contingency_table[s,:].insert(frameline_vert[s],0)
        elif s==16:
            windspeed_contingency_table[s,:]=conditional_intel_line
        elif s==17:
            windspeed_contingency_table[s,:]=conditional_intel_line2
        continue
    print(stations[x], windspeed_contingency_table)
    np.savetxt('Human_TAF_Windspeed_Contingency_Table_' +
stations[x]+'_total events=_'+ str(total_events)+ '_interesting
events=_'+ str(interesting_events)+
'.csv',windspeed_contingency_table,fmt='%s',delimiter=',')
    #percent_windspeed_contingency_table=np.zeros((7,6),
dtype=object)



print(len(wind_prob_a_list),len(wind_prob_b_list),len(wind_prob_a_gi
v_b_list),len(wind_prob_b_giv_a_list),len(wind_prob_a_list_2nup),len
(wind_prob_b_list_2nup),len(wind_prob_a_giv_b_list_2nup),len(wind_pr
ob_b_giv_a_list_2nup))

    print('prob_a: ', prob_A*100, 'prob_b: ',prob_B*100, 'prob_a|b:
',prob_A_giv_B, 'prob_b|a: ',prob_B_giv_A, stations[x])
    #percent_windspeed_contingency_table=
(windspeed_contingency_table/interesting_events)
#    print('percentage: ',percent_windspeed_contingency_table )
#    np.savetxt('Human_TAF_Percentage_Contingency_Table_' +
stations[x]+ '_total events=_'+ str(total_events)+ '_interesting
events=_'+ str(interesting_events)+
'.csv',percent_windspeed_contingency_table,fmt='%s',delimiter=',')
#

#print(counts[0],counts[1],counts[2],counts[3],counts[4])
                #lines[xxx].append(counts[xxx])
                #print(len(line_0), len(line_1), len(line_2),
len(line_3), len(line_4))
    wind_raw_dif_count=0
    wind_raw_dif_sum=0
    wind_dif_given_event_sum=0
    wind_dif_given_event_count=0
```

```
    for prob in range(len(all_humantaf_lines_as_lists)):
        if
str(all_humantaf_lines_as_lists[prob][0]).upper()==stations[x]:

wind_raw_dif_sum=float(wind_raw_dif_sum)+float(all_humantaf_lines_as
_lists[prob][16])
            wind_raw_dif_count=wind_raw_dif_count+1
            if
float(all_humantaf_lines_as_lists[prob][21])==float(1):

wind_dif_given_event_sum=wind_dif_given_event_sum+float(all_humantaf
_lines_as_lists[prob][16])

wind_dif_given_event_count=wind_dif_given_event_count+1
    wind_raw_dif.append(wind_raw_dif_count)
    wind_dif_given_event.append(wind_dif_given_event_count)
    print(stations[x],wind_raw_dif,wind_dif_given_event)
#%%
################################################
####### Wind speed  bayes
################################################
#  prob(A) prob fcst =
count(all_humantaf_lines_as_lists[20]==1)/len(all_humantaf_lines_as_
lists)
#  prob(B) prob obs =
count(all_humantaf_lines_as_lists[21]==1)/len(all_humantaf_lines_as_
lists)
#  prob(A|B) prob fcst given event observed =
count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/
len(all_humantaf_lines_as_lists[21])
#  prob(B|A) prob observed given fcst =
count(all_humantaf_lines_as_lists[21]==1 and
all_humantaf_lines_as_lists[20])/
len(all_humantaf_lines_as_lists[20])
#
#  Prob(A|B)= P(B|A) P(A)
#             ------------
#                P(B)

#
#
#for stations in range(len(stations)):
#    count_prob_a= 0
#    count_prob_b= 0
#    count_prob_a_given_b= 0
#    count_prob_b_given_a= 0
#
#
#
#    for prob in range(len(all_humantaf_lines_as_lists)):
#        print(all_humantaf_lines_as_lists[prob])
```

```
#         #if all_humantaf_lines_as_lists[xx][20]==1:
#         #    count_prob_a=count_prob_a+1
##          if all_humantaf_lines_as_lists[xx][21]==1:
##              count_prob_b=count_prob_b+1
##          if all_humantaf_lines_as_lists[xx][21]==1 and
all_humantaf_lines_as_lists[xx][20]==1:
##              count_prob_a_given_b=count_prob_a_given_b+1
##              count_prob_b_given_a=count_prob_b_given_a+1
#
#
#
#    prob_A= (count_prob_a/len(all_humantaf_lines_as_lists))
#    prob_B= (count_prob_b/len(all_humantaf_lines_as_lists))
#    prob_A_giv_B= (count_prob_a_given_b)/count_prob_b
#    prob_B_giv_A= (count_prob_b_given_a)/count_prob_a
#
#    print(prob_A, prob_B, prob_A_giv_B, prob_B_giv_A, stations[x])

#%%
##############################################
##############################################
### by hour evvectiveness
### raw dif total locations
##########################################
##########################################
zero_list,one_list,two_list,three_list,four_list,five_list,six_list,
seven_list,eight_list,nine_list,ten_list,eleven_list,twelve_list,thi
rteen_list,fourteen_list,fifteen_list,sixteen_list,seventeen_list,ei
ghteen_list,ninteen_list,twenty_list,twentyone_list,twentytwo_list,t
wentythree_list=[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[
],[],[],[],[],[]
#for x in range(24):


for b in range(len(all_humantaf_lines_as_lists)):
    print(all_humantaf_lines_as_lists[b][3], 'testing')
    if float(all_humantaf_lines_as_lists[b][3])==float(0):
        zero_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(1):
        one_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(2):
        two_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(3):
        three_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(4):
        four_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(5):
```

```
        five_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(6):
        six_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(7):
        seven_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(8):
        eight_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(9):
        nine_list.append(all_humantaf_lines_as_lists[b][16])
        print(len(nine_list))
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(10):
        ten_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(11):
        eleven_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(12):
        twelve_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(13):
        thirteen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(14):
        fourteen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(15):
        fifteen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(16):
        sixteen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(17):
        seventeen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(18):
        eighteen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(19):
        ninteen_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(20):
        twenty_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(21):
        twentyone_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(22):
```

```
        twentytwo_list.append(all_humantaf_lines_as_lists[b][16])
        continue
    elif float(all_humantaf_lines_as_lists[b][3])==float(23):
        twentythree_list.append(all_humantaf_lines_as_lists[b][16])
        continue

    else:
        print('okthen')

print(len(twentythree_list),len(twentytwo_list),len(zero_list),len(o
ne_list))

#%%


################################################################
################################################################
#########      hours 24 wind dif|  #################
#########      Wind conditional prob plots  #################
################################################################
################################################################

# libraries


import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
n_groups=20
index = np.arange(n_groups)
#[a_i - b_i for a_i, b_i in zip(a, b)]
wind_prob_fcst_giv_obs_dif=[wind_prob_a_giv_b_list-
wind_prob_a_giv_b_list_2nup for wind_prob_a_giv_b_list,
wind_prob_a_giv_b_list_2nup in zip(wind_prob_a_giv_b_list,
wind_prob_a_giv_b_list_2nup)]
wind_prob_obs_giv_fcst_dif=[wind_prob_b_giv_a_list-
wind_prob_b_giv_a_list_2nup for wind_prob_b_giv_a_list,
wind_prob_b_giv_a_list_2nup in zip(wind_prob_b_giv_a_list,
wind_prob_b_giv_a_list_2nup)]
wind_prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif=[wind_prob_fc
st_giv_obs_dif-wind_prob_obs_giv_fcst_dif for
wind_prob_fcst_giv_obs_dif, wind_prob_obs_giv_fcst_dif in
zip(wind_prob_fcst_giv_obs_dif, wind_prob_obs_giv_fcst_dif)]
#prob_obs_giv_fcst_dif=wind_prob_b_giv_a_list-
wind_prob_b_giv_a_list_2nup
#prob_of_detection_dif=[]
# Data
print(len(wind_prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif))
zero_line=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
df=pd.DataFrame({'x': range(len(stations)), 'Forecast Given Obs':
wind_prob_fcst_giv_obs_dif, 'Obs Given Forecast':
wind_prob_obs_giv_fcst_dif, 'Difference [Forecast Given Obs-Obs
```

```
Given Forecast]':
wind_prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif, 'Zero
Line':zero_line })
#df=pd.DataFrame({'x': range(len(stations)), 'Forecast Given Obs':
wind_prob_a_list=, 'Obs Given Forecast': wind_prob_obs_giv_fcst_dif,
'Difference [Forecast Given Obs-Obs Given Forecast]':
wind_prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif, 'Zero
Line':zero_line })

#df=pd.DataFrame({'x': stations, 'y1': np.random.randn(10), 'y2':
np.random.randn(10)+range(1,11), 'y3':
np.random.randn(10)+range(11,21) })

#zero_line=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
# multiple line plot

plt.plot( 'x', 'Forecast Given Obs', data=df, marker='o',
markerfacecolor='blue', markersize=12, color='skyblue', linewidth=4)
plt.plot( 'x', 'Obs Given Forecast', data=df, marker='',
color='olive', linewidth=2)
plt.plot( 'x', 'Difference [Forecast Given Obs-Obs Given Forecast]',
data=df, marker='o', color='olive', linewidth=2)
plt.plot('x', 'Zero Line', data=df, marker='', color='k',
linewidth=1)
plt.xticks(index, stations, rotation='vertical')
#plt.plot( 'x', 'y3', data=df, marker='', markerfacecolor='blue',
markersize=12, color='skyblue', linewidth=4)
#plt.plot( 'x', 'y4', data=df, marker='', color='olive',
linewidth=2, linestyle='dashed', label="toto")
plt.legend()
plt.xlabel('Locations')
plt.ylabel('Instances')
plt.title('Conditional Statements')



#plt.tight_layout()
#plt.show()
#plt.savefig('Fig 3 Non Category Specific Vis Events by location
(Autotaf)')
#plt.legend()


#%%
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
n_groups=20
index = np.arange(n_groups)
#[a_i - b_i for a_i, b_i in zip(a, b)]
prob_fcst_giv_obs_dif=[wind_prob_a_giv_b_list-
wind_prob_a_giv_b_list_2nup for wind_prob_a_giv_b_list,
```

```
wind_prob_a_giv_b_list_2nup in zip(wind_prob_a_giv_b_list,
wind_prob_a_giv_b_list_2nup)]
prob_obs_giv_fcst_dif=[wind_prob_b_giv_a_list-
wind_prob_b_giv_a_list_2nup for wind_prob_b_giv_a_list,
wind_prob_b_giv_a_list_2nup in zip(wind_prob_b_giv_a_list,
wind_prob_b_giv_a_list_2nup)]
prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif=[prob_fcst_giv_obs
_dif-prob_obs_giv_fcst_dif for prob_fcst_giv_obs_dif,
prob_obs_giv_fcst_dif in zip(prob_fcst_giv_obs_dif,
prob_obs_giv_fcst_dif)]
#prob_obs_giv_fcst_dif=wind_prob_b_giv_a_list-
wind_prob_b_giv_a_list_2nup
#prob_of_detection_dif=[]
# Data
print(len(prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif))
zero_line=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
fifty_line=[50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50,50
,50]
df=pd.DataFrame({'x': range(len(stations)), 'Forecast':
wind_prob_a_list, 'Obs': wind_prob_b_list, 'Obs given Forecast':
wind_prob_b_giv_a_list, 'Forecast given Obs':
wind_prob_a_giv_b_list, 'Fifty Percent Line':fifty_line })
#df=pd.DataFrame({'x': stations, 'y1': np.random.randn(10), 'y2':
np.random.randn(10)+range(1,11), 'y3':
np.random.randn(10)+range(11,21) })

#zero_line=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
# multiple line plot

plt.plot( 'x', 'Forecast', data=df, marker='o',
markerfacecolor='blue', markersize=12, color='skyblue', linewidth=3)
plt.plot( 'x', 'Obs', data=df, marker='', color='blue', linewidth=2)
plt.plot( 'x', 'Obs given Forecast', data=df, marker='o', color='k',
linewidth=3)
plt.plot('x', 'Forecast given Obs', data=df, marker='', color='k',
linewidth=2)
plt.plot('x', 'Fifty Percent Line', data=df, marker='', color='k',
linewidth=1)

plt.xticks(index, stations, rotation='vertical')
#plt.plot( 'x', 'y3', data=df, marker='', markerfacecolor='blue',
markersize=12, color='skyblue', linewidth=4)
#plt.plot( 'x', 'y4', data=df, marker='', color='olive',
linewidth=2, linestyle='dashed', label="toto")
plt.legend()
plt.xlabel('Locations')
plt.ylabel('Instances')
plt.title('Conditional Statements Human Tafs')



#plt.tight_layout()
```

```
#plt.show()
#plt.savefig('Fig 3 Non Category Specific Vis Events by location
(Autotaf)')
#plt.legend()
#%%
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
n_groups=20
index = np.arange(n_groups)
#[a_i - b_i for a_i, b_i in zip(a, b)]
prob_fcst_giv_obs_dif=[wind_prob_a_giv_b_list-
wind_prob_a_giv_b_list_2nup for wind_prob_a_giv_b_list,
wind_prob_a_giv_b_list_2nup in zip(wind_prob_a_giv_b_list,
wind_prob_a_giv_b_list_2nup)]
prob_obs_giv_fcst_dif=[wind_prob_b_giv_a_list-
wind_prob_b_giv_a_list_2nup for wind_prob_b_giv_a_list,
wind_prob_b_giv_a_list_2nup in zip(wind_prob_b_giv_a_list,
wind_prob_b_giv_a_list_2nup)]
prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif=[prob_fcst_giv_obs
_dif-prob_obs_giv_fcst_dif for prob_fcst_giv_obs_dif,
prob_obs_giv_fcst_dif in zip(prob_fcst_giv_obs_dif,
prob_obs_giv_fcst_dif)]
#prob_obs_giv_fcst_dif=wind_prob_b_giv_a_list-
wind_prob_b_giv_a_list_2nup
#prob_of_detection_dif=[]
# Data
print(len(prob_fcst_giv_obs_dif_minus_prob_obs_giv_fcst_dif))
zero_line=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
df=pd.DataFrame({'x': range(len(stations)), 'Forecast':
wind_prob_a_list_2nup, 'Obs': wind_prob_b_list_2nup, 'Obs given
Forecast': wind_prob_b_giv_a_list_2nup, 'Forecast given Obs':
wind_prob_a_giv_b_list_2nup, 'zero line ':zero_line, 'Fifty Percent
Line':fifty_line })
#df=pd.DataFrame({'x': stations, 'y1': np.random.randn(10), 'y2':
np.random.randn(10)+range(1,11), 'y3':
np.random.randn(10)+range(11,21) })

#zero_line=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
# multiple line plot

plt.plot( 'x', 'Forecast', data=df, marker='o',
markerfacecolor='blue', markersize=12, color='skyblue', linewidth=3)
plt.plot( 'x', 'Obs', data=df, marker='', color='blue', linewidth=2)
plt.plot( 'x', 'Obs given Forecast', data=df, marker='o', color='k',
linewidth=3)
plt.plot('x', 'Forecast given Obs', data=df, marker='', color='k',
linewidth=2)
plt.plot('x', 'Fifty Percent Line', data=df, marker='', color='k',
linewidth=1)

plt.xticks(index, stations, rotation='vertical')
```

```
#plt.plot( 'x', 'y3', data=df, marker='', markerfacecolor='blue',
markersize=12, color='skyblue', linewidth=4)
#plt.plot( 'x', 'y4', data=df, marker='', color='olive',
linewidth=2, linestyle='dashed', label="toto")
plt.legend()
plt.xlabel('Locations')
plt.ylabel('Percentages')
plt.title('Conditional Statements')



#plt.tight_layout()
#plt.show()
#plt.savefig('Fig 3 Non Category Specific Vis Events by location
(Autotaf)')
#plt.legend()


#%%
```

```
##lines=[[line_0],[line_1],[line_2],[line_3],[line_4],[line_5]]
#lines=[]
#windspeed_contingency_table=np.zeros((7,6), dtype=object)
#for x in range(len(stations)):
#    fcst_cols_in_con_table=6
#    obs_cols_in_con_table=6
#    line_0=[]
#    line_1=[]
#    line_2=[]
#    line_3=[]
#    line_4=[]
#    line_5=[]
#    percentline_0=[]
#    percentline_1=[]
#    percentline_2=[]
#    percentline_3=[]
#    percentline_4=[]
#    percentline_5=[]
#
#
#
#    uninteresting_events=0
#    interesting_event_instances=0
#    linesperstation=0
#
#    #counts=[0,0,0,0,0]
#
#    for xxx in range(fcst_cols_in_con_table):
#        count0=0
#        count1=0
#        count2=0
#        count3=0
#        count4=0
#        count5=0
#        percentcount0=0
#        percentcount1=0
#        percentcount2=0
#        percentcount3=0
#        percentcount4=0
#        percentcount5=0
#        for xx in range(len(all_autotaf_lines_as_lists)):
#            if
str(all_autotaf_lines_as_lists[xx][0]).upper()==stations[x]:
#                linesperstation=linesperstation+1
#                if
int(float(all_autotaf_lines_as_lists[xx][17]))==int(xxx):
#                    if
int(float(all_autotaf_lines_as_lists[xx][18]))==int(0):
```

```
#                       count0=count0+1
##                    if
int(float(all_autotaf_lines_as_lists[xx][17]))==0 and
int(float(all_autotaf_lines_as_lists[xx][18]))==0:
##
uninteresting_events=uninteresting_events+1
#                   elif
int(float(all_autotaf_lines_as_lists[xx][18]))==int(1):
#                       count1=count1+1
#                   elif
int(float(all_autotaf_lines_as_lists[xx][18]))==int(2):
#                       count2=count2+1
#                   elif
int(float(all_autotaf_lines_as_lists[xx][18]))==int(3):
#                       count3=count3+1
#                   elif
int(float(all_autotaf_lines_as_lists[xx][18]))==int(4):
#                       count4=count4+1
#                   elif
int(float(all_autotaf_lines_as_lists[xx][18]))==int(5):
#                       count5=count5+1
##                elif
float(all_autotaf_lines_as_lists[xx][17])==int(xxx):
##                    count1=count1+1
##                elif
float(all_autotaf_lines_as_lists[xx][17])==int(xxx):
##                    count2=count2+1
##                elif
float(all_autotaf_lines_as_lists[xx][17])==int(xxx):
##                    count3=count3+1
##                elif
float(all_autotaf_lines_as_lists[xx][17])==int(xxx):
##                    count4=count4+1
##                elif
float(all_autotaf_lines_as_lists[xx][17])==int(xxx):
##                    count5=count5+1
#        #print(count0,count1,count2,count3,count4,count5)
##
interesting_event_instances=count1+count2+count3+count4+count5
##        for x in len(6):
##            if x==0
##                percentcount0=0
##                percentcount1=count1/interesting_event_instances
##                percentcount2=0
##                percentcount3=0
##                percentcount4=0
##                percentcount5=0
##            else:
##                percentcount0=0
##                percentcount1=0
##                percentcount2=0
##                percentcount3=0
```

```
##                percentcount4=0
##                percentcount5=0
#
line_0.append(count0),line_1.append(count1),line_2.append(count2),li
ne_3.append(count3),line_4.append(count4),line_5.append(count5)
#        #interesting_event_instances=linesperstation-
uninteresting_events
#
#print(linesperstation,uninteresting_events,interesting_event_instan
ces)
#    lines.append(line_0)
#    lines.append(line_1)
#    lines.append(line_2)
#    lines.append(line_3)
#    lines.append(line_4)
#    lines.append(line_5)
#
total_events=sum(line_0[:]+line_1[:]+line_2[:]+line_3[:]+line_4[:]+l
ine_5[:])
#    uninteresting_events=line_0[0]
#    interesting_events=total_events-uninteresting_events
#    percent_interesting= interesting_events/total_events
#    typetag='Auto'
#    print(stations[x],'total events= ', total_events, 'interesting
events= ', interesting_events,
'percent_interesting',str(percent_interesting))
#    infoline=[stations[x], typetag, total_events,
interesting_events, uninteresting_events, percent_interesting]
#    percent_windspeed_contingency_table=np.zeros((7,6),
dtype=object)
#
#    newline_0=[(x/interesting_events*100) for x in line_0]
#    newline_1=[(x/interesting_events*100) for x in line_1]
#    newline_2=[(x/interesting_events*100) for x in line_2]
#    newline_3=[(x/interesting_events*100) for x in line_3]
#    newline_4=[(x/interesting_events*100) for x in line_4]
#    newline_5=[(x/interesting_events*100) for x in line_5]
#    for s in range(7):
#
#        if s==0:
#            percent_windspeed_contingency_table[s,:]=infoline
#        elif s==1:
#            percent_windspeed_contingency_table[s,:]=newline_0
#        elif s==2:
#            percent_windspeed_contingency_table[s,:]=newline_1
#        elif s==3:
#            percent_windspeed_contingency_table[s,:]=newline_2
#        elif s==4:
#            percent_windspeed_contingency_table[s,:]=newline_3
#        elif s==5:
#            percent_windspeed_contingency_table[s,:]=newline_4
#        elif s==6:
```

```
#            percent_windspeed_contingency_table[s,:]=newline_5
#        continue
#    print(stations[x], windspeed_contingency_table)
#    np.savetxt('Auto_TAF_Contingency_Table_' + stations[x]+'_total
events=_'+ str(total_events)+ '_interesting events=_'+
str(interesting_events)+
'.csv',windspeed_contingency_table,fmt='%s',delimiter=',')
#
#    #percent_windspeed_contingency_table=
(str(windspeed_contingency_table[0])+
str(windspeed_contingency_table[1:]/interesting_events))
#    print('percentage: ',percent_windspeed_contingency_table )
#    np.savetxt('Auto_TAF_Percentage_Contingency_Table_' +
stations[x]+ '_total events=_'+ str(total_events)+ '_interesting
events=_'+ str(interesting_events)+
'.csv',percent_windspeed_contingency_table,fmt='%s',delimiter=',')
#
#
#print(counts[0],counts[1],counts[2],counts[3],counts[4])
#                #lines[xxx].append(counts[xxx])
#                #print(len(line_0), len(line_1), len(line_2),
len(line_3), len(line_4))
#

TAF Extractor

# -*- coding: utf-8 -*-
"""
Created on Thu Oct  3 22:25:33 2019

@author: bbailey
"""

################################################## importing
necessary libraries
import os
import numpy as np
################################################## defining
directories and file names
directory =
os.fsencode(r'I:\setup\Desktop\Thesis\Code\testingautotaf')
################################################## opening file
name and checking key information in file name and within file
supercounter=1
for file in os.listdir(directory):
    filename = os.fsdecode(file)
    print(filename)
    if filename.endswith(".txt"): #and len(filename)<=24:#or
filename.endswith(".py"):
        print('month: ' + filename[-11:-9],'day: ' + filename[-9:-
7],'hour: '+filename[-7:-4], filename)
for file in os.listdir(directory):
```

```
    filename = os.fsdecode(file)
    if filename.endswith(".txt"):
        openfile = open(filename).readlines()
        model_run_name=filename
        openlines=[]
        for lines in openfile:
            openlines.append(lines.split(','))
        location=model_run_name[16:20]
#        print('wndspd',openlines[28][1].strip())
#        print('wnddir',openlines[27][1].strip())
#        print('gust',openlines[31][1].strip())
#        print('gdir',openlines[30][1].strip())
#        print('vis',openlines[51][1].strip())
#        print('cig',float(openlines[64][1].strip())*100)
#        print('lgt',openlines[113][1].strip())
#        print('cig',openlines[28][1].strip())
#        print('mon',model_run_name[-11:-9])
#        print('date',model_run_name[-9:-7])
#        print('hour',model_run_name[-7:-4])
#        print(location)

#########################################################################
#######

#########################################################################
#######
        #########           Dealing with Data
##########

#########################################################################
#######

#########################################################################
#######

        ######################################################## Defining
necessary lists
        wind_speed=[]
        wind_dir=[]
        gust_speed=[]
        gust_dir=[]
        visibility=[]
        ceiling_height=[]
        lightning=[]
        month=[]
        date=[]
        hour=[]
        max_wind_speed=[]
        max_wind_dir=[]
        location=str(model_run_name[16:20])
        tafhr=str(filename[-7:-4])
#
```

```
        ########################################## extracting
primary wx paramaters
        for x in range(30):
#            wind_speed=[]
#            wind_dir=[]
#            gust_speed=[]
#            gust_dir=[]
#            visibility=[]
#            ceiling_height=[]
#            lightning=[]
#            month=[]
#            date=[]
#            hour=[]
#            max_wind_speed=[]
#            max_wind_dir=[]
#            location=str(model_run_name[16:20])
#            tafhr=str(filename[-7:-4])
            if x==0:
                wind_speed.append(openlines[28][0][-6:].strip())
                wind_dir.append(openlines[27][0][-6:].strip())
                gust_speed.append(openlines[31][0][-6:].strip())
                gust_dir.append(openlines[30][0][-6:].strip())
                visibility.append(openlines[51][0][-6:].strip())
                ceiling_height.append(float(openlines[64][0][-
9:].strip()) * 100)
                lightning.append(openlines[28][0][-6:].strip())
                month.append(model_run_name[-11:-9])
                date.append(model_run_name[-9:-7])
                hour.append(model_run_name[-7:-5])
            else:
                wind_speed.append(openlines[28][x].strip())
                wind_dir.append(openlines[27][x].strip())
                gust_speed.append(openlines[31][x].strip())
                gust_dir.append(openlines[30][x].strip())
                visibility.append(openlines[51][x].strip())

ceiling_height.append(float(openlines[64][x].strip()) * 100)
                lightning.append(openlines[113][x].strip())
        ########################################################
rendering max winds
        for x in range(len(wind_speed)):
            if float(wind_speed[x]) >=9.8  and
abs(float(wind_speed[x])-float(gust_speed[x]))>=4.8:
                max_wind_speed.append(gust_speed[x])
                max_wind_dir.append(gust_dir[x])
            else:
                max_wind_speed.append(wind_speed[x])
                max_wind_dir.append(wind_dir[x])

#####################################################################
#######
```

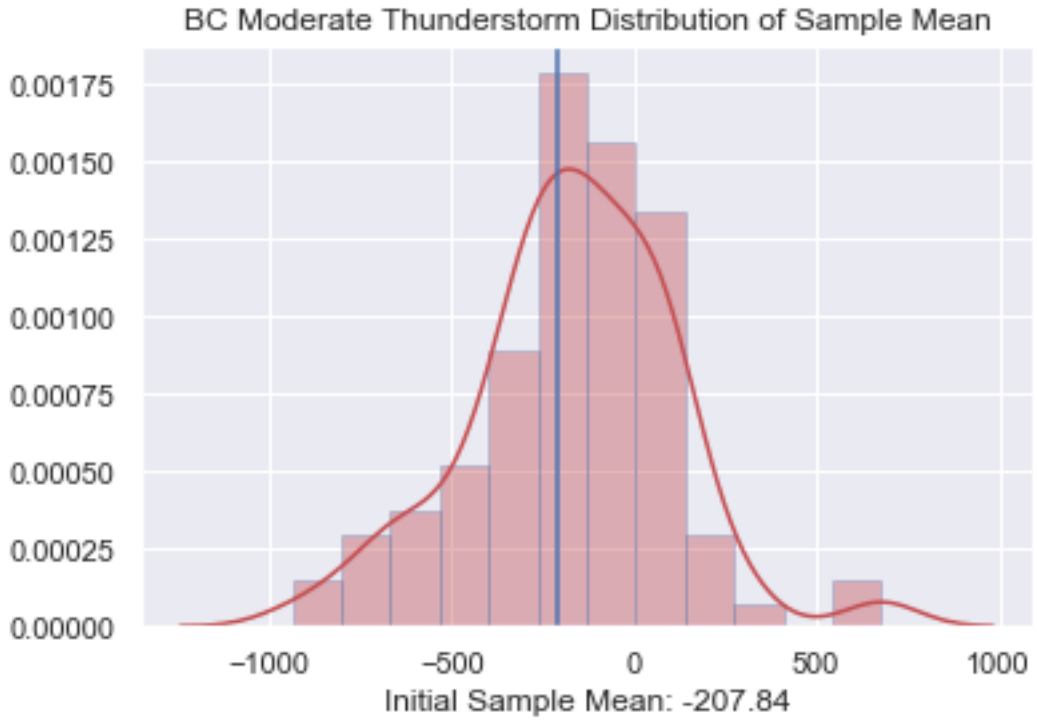
```
#########################################################################
#######
        #########           Dealing with Time
##########

#########################################################################
#######

#########################################################################
#######

        start_month= int(model_run_name[-10:-9])
        start_date= int(model_run_name[-9:-7])
        start_hour= int(model_run_name[-7:-5])
        monthinloop=start_month
        dateinloop=start_date
        hrcount=0
        hourinloop=start_hour +hrcount
        hrout=0
        for xx in range(1,30):
        ###################################Counting Hours n days
            hrcount=hrcount+1
            hourinloop=start_hour+hrcount
            if hourinloop < 24:
                hour.append(str(hourinloop))
                date.append(str(dateinloop))


            if hourinloop >= 24:
                hrout=hourinloop-24
                hour.append(str(hrout))
                dateinloop=start_date+1
                if start_month==5 and dateinloop <=31:
                    date.append(str(dateinloop))
                if start_month==5 and dateinloop>=31:
                    date.append(str(dateinloop-31))
                if start_month==6 and dateinloop <=30:
                    date.append(str(dateinloop))
                if start_month==6 and dateinloop>=30:
                    date.append(str(dateinloop-30))
                else:
                    date.append(str(dateinloop))

        ###################################Counting Months
            if start_month==5:
                if dateinloop>31:
                    monthinloop= monthinloop + 1
                    month.append(str(monthinloop))
                else:
                    month.append(str(monthinloop))
            if start_month==6:
```

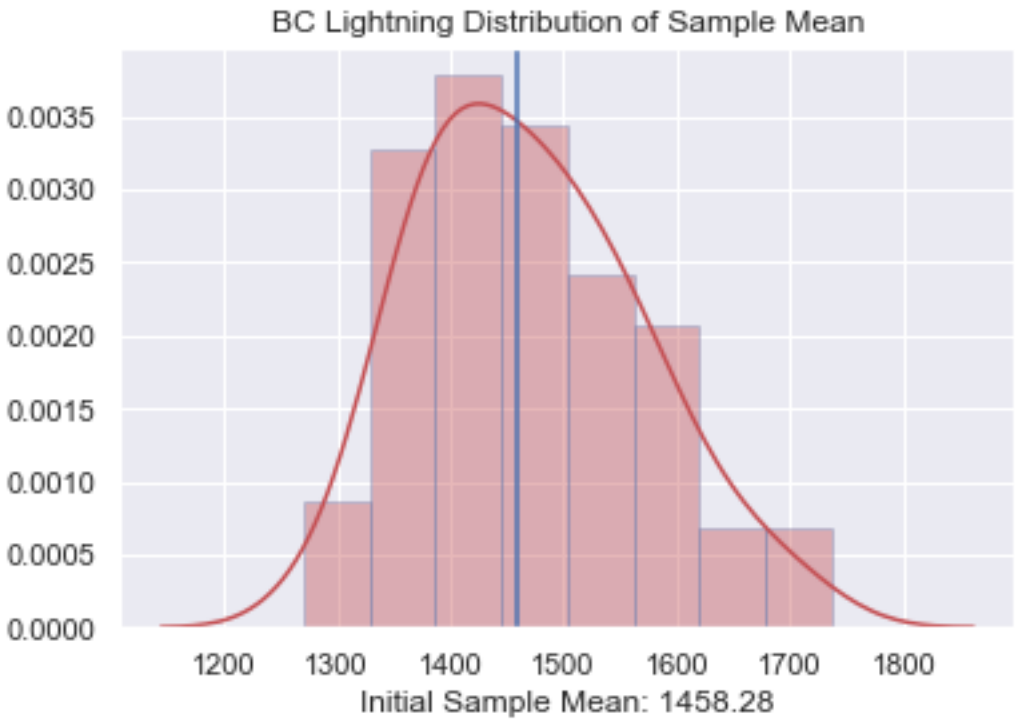
```
                if dateinloop>30:
                    monthinloop= monthinloop + 1
                    month.append(str(monthinloop))
                else:
                    month.append(str(monthinloop))
            if start_month!=5 and start_month!=6:
                month.append(str(monthinloop))



#####################################################################
#######

#####################################################################
#######
        #########           Dealing with Outputs
##########

#####################################################################
#######

#####################################################################
#######
        ######################################## testing print outs
#           print('date',date,'len=',len(date))
#           print('month',month ,'len=',len(month))
#           print('hours',hour,'len=',len(hour))
#           print('wnd spd',wind_speed,'len=',len(wind_speed))
#           print('wnd dir',wind_dir,'len=',len(wind_dir))
#           print('gust spd',gust_speed,'len=',len(gust_speed))
#           print('gust dir',gust_dir,'len=',len(gust_dir))
#           print('cig', ceiling_height,'len=',len(ceiling_height))
#           print('lght',lightning,'len=',len(lightning))
#           print('vis',visibility ,'len=',len(visibility))
#           print('max
spd',max_wind_speed,'len=',len(max_wind_speed))
#           print('max dir',max_wind_dir ,'len=',len(max_wind_dir))
#    print('date',len(date))
#    print('month',len(month))
#    print('hours',len(hour))
#    print('wnd spd',len(wind_speed))
#    print('wnd dir',len(wind_dir))
#    print('gust spd',len(gust_speed))
#    print('gust dir',len(gust_dir))
#    print('cig', len(ceiling_height))
#    print('lght', len(lightning))
#    print('vis', len(visibility))
#    print('max spd',len(max_wind_speed))
#    print('max dir', len(max_wind_dir))
#    print('date',date)
#    print('month',month)
#    print('hours', hour)
```

```
        ######################################## creating output
array
        #outf=np.none((30,14))

#         for x in os.listdir(directory):
#             word=str(x)
#             print(word[-4:])
#             if word.endswith(".txt'"):
#                 for xx in range(len(os.listdir(directory))):
#                     print(word)
#                     outf2=np.empty([30, 14], dtype=object)
#                     print(outf2)
#                     #print(type(str(location)), type(month[x]),
date[x], str(tafhr), hour[x], max_wind_speed[x], max_wind_dir[x],
ceiling_height[x], lightning[x], visibility[x], wind_speed[x],
wind_dir[x], gust_speed[x], gust_dir[x]])
#                     outf2[:]=[str(location), month[xx], date[xx],
str(tafhr), hour[xx], max_wind_speed[xx], max_wind_dir[xx],
ceiling_height[xx], lightning[xx], visibility[x], wind_speed[x],
wind_dir[x], gust_speed[x], gust_dir[x]]
#
print(location,start_hour,start_month,start_date,outf2)
    outf=np.empty((30, 14), dtype=object)
#for x in range(30):
#    outf2[x,:]=[str(location), month[x], date[x],str(tafhr),
hour[x], max_wind_speed[x], max_wind_dir[x], ceiling_height[x],
lightning[x], visibility[x], wind_speed[x], wind_dir[x],
gust_speed[x], gust_dir[x]]
#    print(location,start_month,start_date,start_hour,outf2)
#print(supercounter)
#supercounter=supercounter+1
    for x in range(0,30):
        outf[x,:]=[str(location), month[x], date[x],str(tafhr),
hour[x], max_wind_speed[x], max_wind_dir[x], ceiling_height[x],
lightning[x], visibility[x], wind_speed[x], wind_dir[x],
gust_speed[x], gust_dir[x]]
    print(location,start_month,start_date,start_hour,outf)
    print(supercounter)
    print(outf.shape)
    np.savetxt(str(location)+ str(month[1]) + str(date[1]) +
str(filename[-7:-4])+'autotaf.csv',outf,fmt='%s',delimiter=',')
    supercounter=supercounter+1
#    wind_speed=[]
#    wind_dir=[]
#    gust_speed=[]
#    gust_dir=[]
#    visibility=[]
#    ceiling_height=[]
#    lightning=[]
#    month=[]
#    date=[]
#    hour=[]
```

```
#    max_wind_speed=[]
#    max_wind_dir=[]


        #month=[]
        #date=[]
        #hour=[]

        #output date and time
        #output to file for excell in taf format

        #wind_speed=[]
        #wind_dir=[]
        #gust_speed=[]
        #gust_dir=[]
        #ceiling_height=[]
        #lightning=[]
        #month=[]
        #date=[]
        #hour=[]

        #print(type(openfile), type(openlines))


                #print(len(openfile))
                #print(len(openfile[851]))


                #openfile=open(filename)
                #linesinfile=openfile.read()
                #print(linesinfile[15])
                #for xx in range(100):
                #for xx in range(len(linesinfile)):
                    #alllines.append(linesinfile)
                    #print(len(alllines))
                    #print(len(linesinfile))
                #print(len(alllines))

                #print(os.path.join(directory, filename))

        #print(len(alllines))
        #with open('14Mrightmyguy7.csv', 'w+') as f:
        #    for item in alllines:
         #       f.writelines("%s\n" % item)

        #print(alllines[1:10])
        #print(alllines.shape)
        #outf=np.zeros((len(alllines),9))
        #outf=alllines
        #np.savetxt('14Mright.csv',outf,delimiter=',')
        #outf=np.zeros((len(alllines),9))
```

```
#outf[len(alllines),:]=[dateintaf,hrintaf,wdir0,wspd0,gust0,vis0,als
t0,0,0]
        #np.savetxt('14mall.csv',outf,delimiter=',')

        #with open('14mall', 'w') as f:
        # #   f.write('\n'.join(alllines))
```




BC Lightning Distribution of Sample Mean
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
1200
1300
1400
1500
1600
1700
1800
Initial Sample Mean: 1458.28

Bibliography

16th Weather Squadron. (2018). Numerical Modeling Capabilities. *29th Conference on Weather Analysis and Forecasting/25th Conference on Numerical Weather Prediction.* Austin, Texas: American Meteorological Society. Retrieved 01 28, 2020, from https://ams.confex.com/ams/29WAF25NWP/videogateway.cgi/id/47681?recordingid=47681&uniqueid=Paper344737&entry_password=null

AFI 15-114. (2017, March 16). *Weather Technical Readiness Evaluation.* (R. O. Stoffler, Ed.) Retrieved January 26, 2020, from Air Force Electronic Publishing: https://static.e-publishing.af.mil/production/1/af_a3/publication/afi15-114/afi15-114.pdf

AFI 90-802. (2019, April 1). Special Management. (J. T. Rauch, Ed.) Electronic Publishing . Retrieved Februrary 18, 2020, from Risk management.

AFI11-202V3_ACCSUP_I. (2012). *AIR COMBAT COMMAND Supplement Flying Operations.* Retrieved Februrary 15, 2020, from General Flight Rules: https://upload.wikimedia.org/wikipedia/commons/4/47/USAF_AFI_11-202_ACC_General_Flight_Rules_-_28_NOV_12.pdf

AFPAM:11-238. (2011, March 7). *Flying Operations.* (A. F. Publishing, Ed.) Retrieved Februrary 15, 2020, from Aircrew Quick Reference To The METAR and TAF Codes: https://www.skybrary.aero/bookshelf/books/1543.pdf

AFPD 11-1. (2004, August 10). *Flying Hour Program.* (R. E. Keys, Ed.) Retrieved January 28, 2020, from Air Force Electronic Publications: https://static.e-publishing.af.mil/production/1/af_a3_5/publication/afpd11-1/afpd11-1.pdf

AFPD 15-1. (2019, November 14). *Weather Operations.* (M. D. Kelly, Ed.) Retrieved January 26, 2020, from United States Air Force Electronic Publishing: https://static.e-publishing.af.mil/production/1/af_a3/publication/afpd15-1/afpd15-1.pdf

Altman, D. G., & Martin, B. J. (1994). Diagnostics Test 2: Predictive Values. *British Medical Journal (BMJ), 309*, 102.

Anaman, K. A., Lellyett, S. C., Drake, L., Leigh, R. J., & Henderson-Sellers, A. N. (1997). Benefits of meteorological services: evidence from recent research in

Autstrailia. *Meteorological Applications*, 103-115. doi:10.1017/S1350482798000668

Anaman, K. A., Quaye, R., & Owusu-Brown, B. (2017, May 26). Benefits of Aviation Weather Services: A Review of International Literature. *Research in World Economy, 8*(1), 45-58. doi:10.5430/rwe.v8nlp45

Batterton, K. A., & Schubert, C. M. (2013, March 29). Confidence intervals around Bayes Cost in multi-state diagnostic settings to estimate optimal performance. *Statistics in Medicine,* 1-20. doi:10.1002/sim.6174

Bossart, S. J., & Bean, J. E. (2011). Metrics and Benefit Analysis and Challenges for Smart Grid Field projects. *IEEE 2011 EnergyTech*, 1-5. doi:10.1109/EnergyTech.2011.5948539

Bradley, A. A., Schwartz, S. S., & Hashino, T. (2008). Sampling Uncertainty and Confidence Intervals for the Breier Score and Brier Skill Score. *AMS Weather and Forecasting, 23*, 992-1006. doi:10.1175/2007@AF2007049.1

Epstein, E. S. (1962, June). A Bayesian Approach to Decision Making in Applied Meteorology. *American Meteorological Society: Journal of Applied Meteorology*, 169-177. Retrieved from https://www.researchgate.net/publication/249605146_A_Bayesian_Approach_to_Decision_Making_in_Applied_Meteorology

Heston, T. F. (2011). Standardizing Predictive Values in Diagnostic Imaging Research. *Journal of Magnetic Resonance Imaging*, 505-507. doi:10.1002/jmri.22466

JMOC. (2011). Joint Meteorology and Oceanography Handbook. Retrieved Feb 4, 2020, from https://www.jcs.mil/Portals/36/Documents/Doctrine/pams_hands/metoc_hbk.pdf

Klein, A., Kavoussi, S., & Lee, R. (2009). Weather Forecast Accuracy: Study of Impact on Airport Capacity and Estimation of Avoidable costs. *Eith USA/ Europe Air Traffic Management Research and Development Seminar.* ATMseminar.org. Retrieved from http://www.atmseminar.org/seminarContent/seminar8/papers/p_008_W.pdf

Mahringer, G. (2008). Terminal Aerodrome Forecast Verification in Austro Control Using Time Windows and Ranges of Forecast Conditions. *Meteorological Applications: Royal Meteorological Society*, 113-123. doi:doi:10.1002/met.62

McCreery, C., & Phil, D. (2018). Probability and Bayes Theorem. Oxford University. Retrieved from celiagreen.com/charlesmccreery/statistics/bayestutorial.pdf

McGee, S. (2002, August). Simplifying Liklihood Ratios. *Journal of Geral Internal Medicine*, 647-650. doi:10.1046/j.1525-1497.2002.10750.x

Mens, A., Korff, M., & Van Tol, A. (2012). Validating and Improving Models for Vibratory Installation. *Geotechnical and Geological Engineering*, 1085-1095. doi:10.1007/s10706-012-9506-5

NWS. (2020). *The National Weather Service (NWS)*. Retrieved 01 28, 2020, from Weather.gov: https://www.weather.gov/about/

OGIMET. (2019). *OGIMET*. Retrieved August 2019, from https://www.ogimet.com/metars.phtl.en

Pauker, S. G., & Kassirer, J. P. (1975). Therapeutic Decision Making: A Cost-Benefit Analysis. *NEJM*, 229-234. doi:10.1056/NEJM197507312930505

Skaltsa, K., Jover, L., Fuster, D., & Carrasco, J. (2012). "Optimum Threshold Estimation Based. *Statistics in Medicine*, 1098-1109. doi: doi:10.1002/sim.4369

Trevethan, R. (2017). Sensitivity, Specificity, and Predictive Values: Foundations, Pliabilities, and Pitfalls in Research and Practice. (A. A. Alford, Ed.) *Frontiers in Public Health*. doi:10.3389/fpubh.2017.00307

Zhang, G., Patuwo, B. E., & Hu, M. Y. (1998). Forecasting With Artificial Neural Networks: The State of the Art. *International Journal of Forecasting*, 35-62.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From – To)*
06-03-2020	Master's Thesis	June 2018 – March 2020

TITLE AND SUBTITLE

Next-Generation Air Force Weather Metrics Via Bayes Cost Analysis

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Bailey, Brandon M., Capt

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/ENP)
2950 Hobson Way, Building 640
WPAFB OH 45433-8865

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENP-MS-20-M-077

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AGENCY Intentionally left blank
ADDRESS Intentionally left blank
PHONE and EMAIL Intentionally left blank
ATTN: POC Intentionally left blank
(no sponsor enter: Intentionally left blank)

10. SPONSOR/MONITOR'S ACRONYM(S)

Intentionally left blank
AFRL/RHIQ (example)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

Intentionally left blank

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRUBTION STATEMENT A. Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

This research proposes a new methodology for U.S. Air Force weather forecast metrics. Military weather forecasters are essentially statistical classifiers. They categorize future conditions into an operationally relevant category based on current data, much like an Artificial Neural Net or Logistic Regression model. There is extensive literature on statistically-based metrics for these types of classifiers. Additionally, in the U.S. Air Force, forecast errors (errors in classification) have quantifiable operational costs and benefits associated with incorrect or correct classification decisions. There is a methodology in the literature, Bayes Cost, which provides a structure for creating statistically rigorous metrics for classification decisions that have such costs and benefits. Applying these types of metrics to Air Force weather yields more informative metrics that account for random chance while remaining simple to calculate. Using notional costs and benefits from Air Force operations subject matter experts, a case study was conducted by performing Bayes Cost-based verification on Terminal Aerodrome Forecasts and Watches/Warnings/Advisories compared to surface observations from a selection of military installations in the continental United States during the period 01 May 2019 to 30 June 2019. The case study illustrates the added utility of the new metric paradigm.

15. SUBJECT TERMS

(Bayes Cost, Conditional Probability, TAF, WWA, Air Force Weather.)

16. SECURITY CLASSIFICATION OF: Unclassified			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Lt Col Andrew J. Geyer, AFIT/ENC
U	**U**	**U**	**UU**	211	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext. 4594 (Andrew.Geyer@afit.edu)

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18