April 2019

Performing Binary Classification of Contests Profitability for Draftkings

Jon Samuel Venne
Worcester Polytechnic Institute

Saul Van Hook Woolf
Worcester Polytechnic Institute

Sean Dale Brady
Worcester Polytechnic Institute

Performing Binary Classification of Contest Profitability for DraftKings

A WPI Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements for the degree of Bachelor of Science

By

Brady, Sean Perry, Jackson Venne, Jonathan Woolf, Saul

April 24, 2019

Sponsored By

DraftKings

Advised By

WPI :	Prof. Rick Brown	Prof. Randy Paffenroth
DK :	Brandon Ward	

Abstract

In this Major Qualifying Project, we worked alongside the online daily fantasy sports company DraftKings to build an algorithm that would predict which of the company's contests would be profitable for them. Namely, our goal was to detect contests at risk of not filling to their maximum number of entrants by four hours before the contest closed. We combined categorical and numerical header data provided by DraftKings for hundreds of thousands of previous contests using modern data science techniques such as ensemble methods. We then utilized parameter estimation techniques such as linear regression and the Kalman Filter to model the time series data of entrants into a given contest. Finally, we fed these parameters and the predictions they generated, alongside the header data previously mentioned, into a Random Forest algorithm that provided our final prediction as to whether a contest would fill or not. The algorithm we developed outperformed previous methodologies involving only portions of the aforementioned data.

Acknowledgements

This Major Qualifying Project would not have been possible without the contributions of many individuals in both the WPI community and at DraftKings. Firstly, to our WPI advisors Professor Randy Paffenroth and Professor Rick Brown, we thank you for your constant guidance, wisdom and direction from the start of this project to its end. Thank you to DraftKings and WPI for coordinating this project sponsorship and beginning what will hopefully be a productive cooperative. We particularly want to thank Brandon Ward at DraftKings for his major role in the development of our project. His advisory role throughout the course of our work was invaluable. In addition to these major players, many individuals played helpful auxiliary roles. Graduate student Rasika Karkare greatly aided our understanding of imbalanced data and provided strategies for us to work with it. Another graduate student, Wenjing Li, had experience in ensemble learning that was crucial to the development of our methodology. Finally, we thank WPI's Turing computer and particularly Spencer Pruitt for his regular assistance in getting our code running efficiently. To all these people and the many more who supported us through this process, we thank you. This project was made possible by you.

Executive Summary

Fantasy sports is an area that has grown in popularity immensely over the last few decades. With this growth has come the development of many new platforms in the realm of online daily fantasy sports. Chief amongst these new platforms is DraftKings, a company founded in 2012 that champions online fantasy sports contests. On DraftKings, a fixed number of users may pay an entry fee to join a contest and compete for a fixed cash prize, which can be up to $1 million or more. After paying an entry fee, users select a roster of professional players and the users whose rosters score the most fantasy points win prizes. Since the entry fee, maximum number of entries, and prizes of each contest are pre-defined, DraftKings earns maximum revenue when each contest fills to its maximum number of entries. If a contest appears to be at risk of not filling, DraftKings can mitigate losses by directing marketing efforts towards this contest. These marketing efforts must be coordinated starting four hours before entries close. The goal of this project, then, is to flag the contests we expect to fail, four hours before entries close.

To explore this problem, DraftKings offered over 500,000 historical contests for our team to analyze. This data included information about each contest's header information such as entry fee, top prize, start time, maximum number of entrants and how many users had played in similar contests. The data also included time series information, namely measurements of the number of new entries into the contest from the contest's open to its close. The magnitude of the dataset led us to seek a solution using machine learning and data science. We immediately partitioned off a quarter of our data for testing purposes and began the data cleaning process.

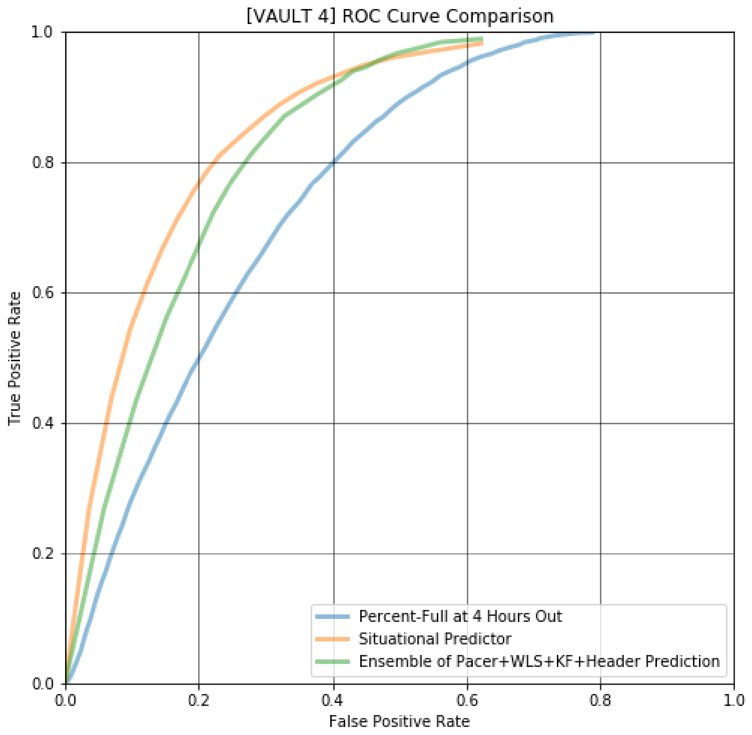
The dataset given to the team was reorganized and modified to fit into prediction algorithms. The Header Data we received was scaled and coded. Variables such as "ContestPayoutTime" that were not useful in predicting the outcome of a contest were eliminated. A contest that fills to its maximum was classified to be a success and a contest that does not fill was classified as a failure. The Time Series data lent itself to some common modeling techniques in data science, namely Linear Regression and Kalman Filtering.

For our project, we utilized a weighted linear, or least-squares, regression. This technique allowed points towards the end of the time series to be weighted more heavily than points early in the time series. For example, if a contest is on pace to fall short, but suddenly gains many entries towards our 4-hour out deadline, a traditional linear regression would still weight each point equally and predict a failure. However, a weighted least squares

would take those later entries into consideration and predict success. The model parameters we generated from multiple different weighting procedures were obtained and added as new variables to the complete dataset, along with the final predictions based on those parameters.

The Kalman Filter is another technique for modeling Time Series data that is traditionally used for dynamic system parameter estimation. However, for our problem, we are attempting to estimate the parameters of an approximately exponential curve, a model that most contests' entries follow. Using different inputs for our Kalman Filter, we obtained 15 different predictions that were also added to the complete dataset. In total, we had over 100 new variables in our cleaned dataset developed through Linear Regression, Kalman Filtering and the Header Data. The Random Forest ensemble was then selected to use these variables to predict success or failure of contests.

The Random Forest Algorithm in a classification setting takes a high-dimensional dataset and breaks it into smaller chunks that are fed into individual decisions trees. These trees use information from each variable, or feature, that they receive to make a prediction about whether a contest will succeed or fail. Taking these trees together in a random forest allows for more accurate results than any one tree on its own. Once our full process of cleaning, modeling, and using the algorithm was complete, we could repeat the process on the testing data we set aside at the start of the project.



To compare our methodology with other techniques, we utilized the receiver operating characteristic (ROC) curve. This curve compares the False Positive (predicting a contest will fail to fill, but it fills) rate versus True Positive (predicting a contest will fail to fill, and it fills) rate along many different threshold values. A greater area under the curve indicates a better prediction. As you can see in the above figure, our ensemble in green using all the aforementioned data predicts better than a simple model in blue using only the proportion of contests filling 4-hours out and better than the current method utilized by DraftKings.

Given our model's accurate predictions, it can serve as the foundation of future studies. Because we chose a classification problem instead of a regression problem, our algorithm only outputs whether or not a contest will fill or fail to fill. It would also be useful to know by how much a contest will miss, so contests that nearly fill would not be counted as failures. Another limitation of our work was the assumption of an exponential model. It is possible that the time series data follows a more complex model that could be explored in future work. In closing, if our process is adapted and used by DraftKings, the company will be able to better identify contests at risk of failing to ensure their continued success in online daily fantasy sports.

Contents

List of Figures

List of Tables

Introduction

As of 2017, there were nearly 60 million fantasy sports users in the United States alone, with each spending an annual average of $556, making the industry as a whole worth around 7 billion dollars at the time [9] [20] [10]. DraftKings is a Boston-based fantasy sport contest provider. Founded in 2012, they are a relatively new company, but they (alongside their main competitor FanDuel) already control the vast majority of the growing online fantasy sports market. Together in 2017, they brought in nearly 43% of the total fantasy sports revenue for that year, with DraftKings earning a little more than half of that. DraftKings runs daily fantasy sports competitions on virtually every major sport, including football, soccer, baseball, and even recently the e-sport League of Legends. In these competitions, users compete against each other for thousands of dollars in guaranteed prize money.

Guaranteed prize pools present DraftKings with a interesting business problem. If a contest does not achieve enough entries, DraftKings could suffer a loss when they pay out to the winner of the contest. Luckily, if a contest is flagged as failing to reach its maximum entries before closing, DraftKings can advertise the contest to improve its visibility on the site to users. DraftKings' current solution is having analysts check that large contests are on track to fill before closing. Each large contest on the website can be monitored for entries until its close, but using a human prediction can be inaccurate and tedious. Some smaller contests may not even be worth checking by hand.

Recently, a trend towards data science has produced many new and innovative solutions for complex problems such as this one. Data science is a multi-disciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data [3]. Famous examples of such insights include internet search engines, speech recognition and fraud detection. Knowledge gained from data science techniques id often used to automate mundane repetitive tasks. One popular field in data science is machine learning. Machine learning is a type of algorithm that automates analytical model building [18]. These models are created by examining previous data to predict future outcomes. A machine learning algorithm could help DraftKings track contests and flag failing ones for advertisement space. DraftKings has thousands of data entries of previous contests that the algorithm can learn from. Such an algorithm could greatly improve DraftKings' profits by allowing the tracking of smaller failing contests, reducing man power tracking larger contests and helping make more informed decisions on how to use their limited advertisement space to maximize profits.

This project's goal was to create a machine learning algorithm using DraftKings' vast data collection to identify failing contests early enough that DraftKings could take action to minimize their losses. This paper serves to record the different techniques in time series prediction, machine learning, and imbalanced data that were researched, implemented and tested.

Background

2.1 Fantasy Sports and DraftKings

Our sponsoring company, DraftKings, is one of the leaders in online fantasy sports. Fantasy sports are an online, skill-based game where users draft rosters of professional sports players to compete against other users on a given platform. Users earn points based off of the statistical performance of each professional player they draft in the games relevant to their respective contests. Examples of some standard statistics tracked by fantasy sports include yards gained, strikeouts thrown and free throws made. These points are then totalled up by an algorithm and the user who earns the highest amount of points from their drafted players wins the contest and often some cash prize. Fantasy sports often drive fierce competition to see who can draft the best team . As such, the market for daily fantasy sports platforms has grown immensely, with DraftKings taking the lead of this newly emerging industry.

DraftKings is a Boston-based online daily fantasy sports provider. Founded in 2012 by Paul Liberman, Jason Robins and Matt Kalish, DraftKings has quickly grown into a billion dollar company [5]. Their model differs from traditional fantasy sports in that their contests span only a few games rather than an entire professional sports season. DraftKings runs an online platform and a mobile application in which users can discover fantasy sports contests of many types, ranging in sport, entry fee, maximum number of entrants allowed and numerous other characteristics. On these platforms, users select a contest and pay a fee to enter up front. Similarly to traditional fantasy sports as described above, they can then select a roster of players from the professional sports games that the contest covers. Then the roster locks and the professional players earn points for the user based on their performance in game. Users whose rosters perform better in the contest are then eligible to receive prize money or free entry to other contests.

This project's goal is to create a predictive algorithm to correctly label if a DraftKings contest will reach its maximum number of participants. The vast majority of the contests the company has run over the last 7 years (approximately 90%) fill to the maximum number of entries, however, when a contest does not fill, DraftKings can lose money, as they guarantee an initial amount in prizes at the start of each contest. It would behoove DraftKings to predict whether a contest will fill to its maximum number of entries early enough that DraftKings could take action. Thus, if a contest is in danger of not filling, they can promote

	Predict H_0 Predict Contest Fills	**Predict** H_1 Predict Contest Fails
Actually H_0 Contest Fills	True Negative	False Positive
Actually H_1 Contest Fails	False Negative	True Positive

Table 2.1: This table is the confusion matrix of possible prediction alignments in our case of binary classification. For this application, the null hypothesis (H_0) is that a given contest will fill. The alternative (H_1) is that the given contest does not fill. Since the goal of this work is to detect failing contests, a failing contest is considered to be a positive and a successful contest is considered to be a negative. The prediction is considered "True" if the predicted class is the same as the actual class, and "False" otherwise.

it by advertising it to drive up entries. Here, the null hypothesis is that a contest will fill and be a "Success". The alternative hypothesis is then that a contest will not fill; this is what we are trying to detect. Table 2.1 shows a labeled confusion matrix for clarification. In this example, if a contest is predicted to fail (H_1) and it actually fails (H_1) then that is a "True Positive", the total number of which appears in the bottom right cell. If a contest is predicted to fail (H_1) and it actually succeeds (H_0) then that is a "False Positive" appearing in the top right cell and so on.

For DraftKings' purposes, a false negative is considered approximately ten times worse than a false positive. In other words, predicting that a contest will fill and having it fail to fill is about ten times worse than predicting it will fail to fill and it actually fills. Now that we have identified the important aspects of our classification problem, we can begin to explore the intricacies of the dataset that DraftKings provided our team.

2.2 The Dataset

The data consists of two main categories: what we call **Header Data** and **Time Series Data**. In total, we received Header and Time Series data for 630,446 contests for the team to analyze and predict on. In this section, we will explore both categories, describe the magnitude and makeup of each and explore some characteristics of the set.

2.2.1 Header Data

The Header Data in our set consists of all the label information for each of the hundreds of thousands of DraftKings contests over the last three years. A typical contest might be an NFL contest where users pay \$5 to enter, there are a total of 100,000 permitted entries, and the contest only spans 1 actual professional game. The users scoring the most points for that specific contest can win the top prize or any number of secondary prizes. For each contest, we were provided 21 header columns of data, which can be found in Table 2.2.

Feature Name	**Data Type**	**Description**
ContestId	Integer	a unique 7- or 8-digit number identifying each contest
DraftGroupId	Integer	number identifying a unique SportName, VariantName and ContestStartDatetimeEST combination
SportName	String	a three or four character string describing the sport of the contest
VariantName	String	a string description of the type of contest
GameSet	String	a string description of the time of day that the contest runs
ContestName	String	the string users see describing the contest
ContestStartDatetimeEST	Datetime Object	when the contest opens to be entered by users
ContestEndDatetimeEST	Datetime Object	when the contest closes for users
ContestPayoutDatetimeEST	Datetime Object	when DraftKings pays the prizes out to winning users
EntryFeeAmount	Double	value of the price a user must pay to enter the contest
TotalPrizeAmount	Double	value of the total amount in dollars that DraftKings will pay out to winning users in that contest
MaxNumberPlayers	Integer	value of the number of users allowed to enter that contest
MaxEntriesPerUser	Integer	value of the number of entries a single user can submit to that contest
Entries	Integer	value of the number of entries that contest received by its close
DistinctUsers	Integer	value of the number of unique individuals entered in the contest
Contest_Group	String	value of the type of contest
NumGames	Integer	value of the number of professional games covered by that contest; for example, one contest may span 2 NBA games or 1 NFL game or 11 PGA tournament events
DraftablePlayersInSet	Double	value of the number of professional players available to be drafted by users in the contest
PaidUsersInDraftGroup	Integer	value of the number of users who have previously participated in contests with the same DraftGroupId
TopPrize	Double	value of the dollar amount paid to the top user in the contest
MaxPayoutPosition	Integer	value of the index of the winning user in that contest, not particularly pertinent to our analysis

Table 2.2: Summary of the provided Header Data including names, data types, and brief descriptions of all 21 included features.

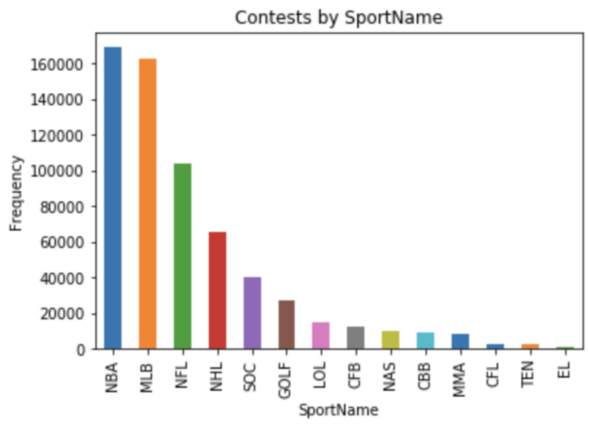


Figure 2.1: A histogram of the number of contests for each SportName sorted in descending order. Approximately 80% of all contests are of the type NBA, MLB, NFL, or NHL which stand for basketball, baseball, football and hockey respectively.

An important distinction in the Header Data is that between players and users. Here, players are considered to be the professional athletes in the actual games played and users are the DraftKings subscribers who enter into contests to select teams of those players.

Histograms of some of the Header Data columns can be found in the Figures 2.1 and 2.2. Most contests that DraftKings runs involve the four major professional US sports: basketball, football, baseball and hockey. In fact, 79.6% of contests involve these four professional sports. The **VariantName** variable describes the type of contest, with over half of contests taking place in Classic mode. The **GameSet** variable describes the time of day that the contest runs during, with about two in five taking place during the day, Eastern Time, with other major time periods being late at night and early in the morning.

A typical **ContestName** might be *CFB 1K Blitz 1,000 Guaranteed*, describing the sport, prizes and occasionally the **VariantName**, all information coded in other variables. **EntryFeeAmount** ranges from $0.10 to $28,000, but most values fall between $1 and $27 with a median of $5. **TotalPrizeAmount** ranges from $2 to $2,000,000 with most values falling between $100 and $25,000, with a median of $500. Similarly, **TopPrize** ranges from $1.80 and $2,000,000, with most prizes being between $20 and $400. **MaxNumberPlayers** ranges from 2 to nearly 2,000,000 with a median value of 98. For most contests, **MaxEntriesPerUser** is 1 or 2, but occasional contests are nearly uncapped, with up to a billion entries available to each user. **Entries** varies between 1 and over 1,000,000, but most contests fall between 23 and 441 entries; **DistinctUsers** varies similarly from 1 to about 500,000, with most falling between 23 and 277 distinct users in any given contest.

Most contests fall into three categories of **ContestGroup**: Headliners, which are the main featured contests; Satellites, whose prize is free entry into another contest with a large prize pool; and Single Entry in which only one entry is allowed per user. **NumGames**, describing how many professional games a contest covers, is less than eight 75% of the time, but can be as many as 64 in our dataset. **DraftablePlayersInSet** is the number of players available in the draft set for a contest with a median of 166, but is skewed left with a mean value of

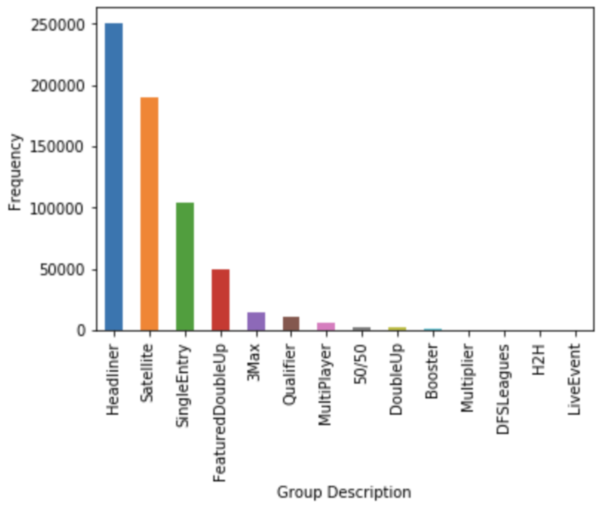


Figure 2.2: A histogram of the number of contests for each ContestGroup sorted in descending order. Approximately 94% of all contests are of the type Headliner, Satellite, SingleEntry, or FeaturedDoubleUp.

about 316. Finally, the draft groups that DraftKings creates consist of all users who have participated in similar contests recently, as defined by the company. These groups can be as small as 0 and as large as 750,000, but most fall between 7,000 and 62,000.

2.2.2 Time Series Data

The Time Series Data was separated into monthly files. For example, one file 2015-09.csv contained all the entry information for contests running in September of 2015. Each row in these monthly entry files included the **ContestId**, minutes until the contest closes and the number of entries the contest received in the last minute. An example can be seen in Table 2.3. Between 2 minutes to close and 1 minute to close, Contest 7962690 received 18 entries. Likewise from 1 minutes to close to 0 minutes to close, the contest received 24 entries. Unfortunately, many contests began near the end of one month and continued into the next, causing some individual contest data to be split between multiple month files. This presented a technical challenge in aggregating data discussed more in the methodology section.

Using the time series data, we can sum the entries in every interval to identify the total number of entries a contest receives at any filled minute prior to the contest start. Figure 2.3 shows a scatter plot of the time series data for a single contest as it was provided to us. Alternatively, Figure 2.4 shows a plot of the total number of entries in a contest summed over time.

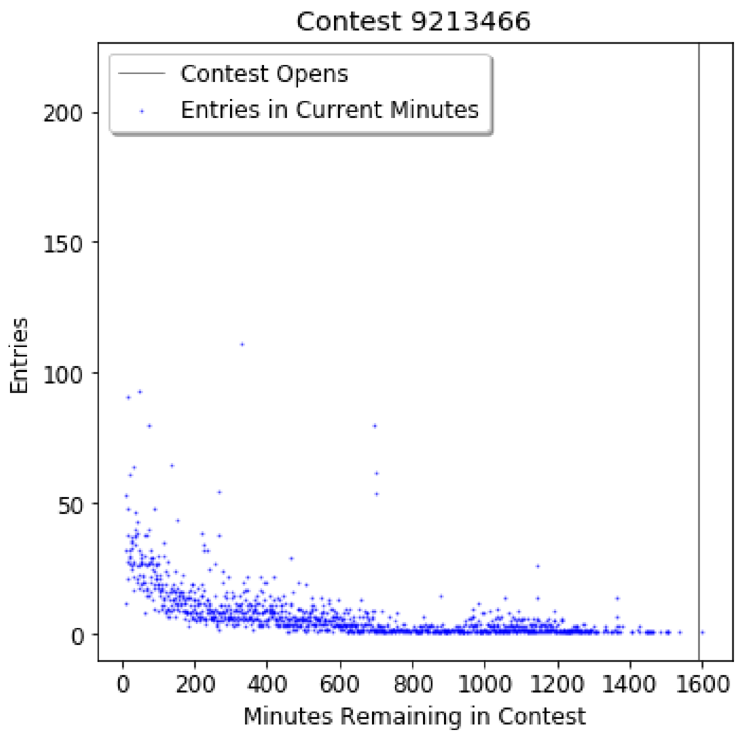


Figure 2.3: A scatter plot of the entries per minute for an example contest. This shows the form the time series data was originally provided in.

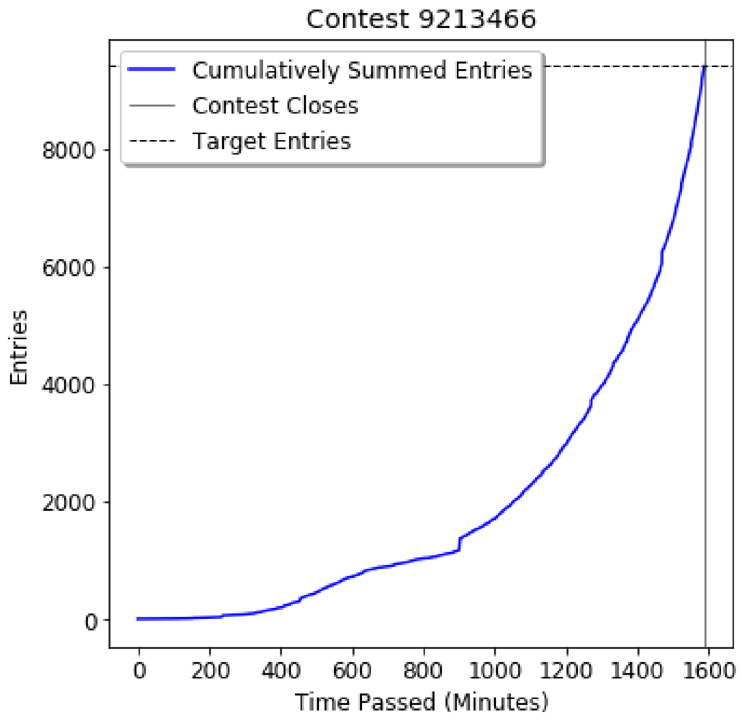


Figure 2.4: A plot of the total summed entries over time for an example contest. This shows the form of data we were more interested in using

Contest ID	**Minutes Remaining in Contest**	**Entries in Last Minute**
7962690	1000	3
7962690	999	4
7962690	997	2
⋮	⋮	⋮
7962690	2	20
7962690	1	18
7962690	0	24
7865930	600	2
7865930	596	1
7865930	594	2
⋮	⋮	⋮

Table 2.3: This is an example of the form the times series data was originally provided in. Each file included columns for the unique contest ID, minutes until the contest closed, and the number of entries that contest received in the last minute. Data only appears if users entered the contest within the last minute, so gaps in "Minutes Remaining in Contest" can appear if no new entries were made that minute. Additionally, each file contained multiple thousands of unique contests (this synthetic set shows only two, but would surely include thousands more).

2.2.3 How to Approach the Problem

Given all this information about the dataset, we are still left with the issue of deciding how to best utilize it for the purpose of predicting contest profitability.

To summarize the points already made, the dataset is known to be large, both in size and dimension, as there are over 600,000 entries each with 21 potential features to analyze. Prediction on data of this scale is commonly performed using flexible modeling techniques that can better accommodate its complexities. One draw-back of flexible modeling, though, is that as the flexibility increases the interpretability of the result decreases. That is to say, it becomes more difficult to discern trends between the explanatory variable(s) and the response variable as flexibility increases. In this application, we care more about the actual prediction than being able to interpret trends so it would seem a flexible model may be what we want.

We should also recall that the dataset includes time series data for each contest, which would likely be far too large for any single modeling scheme to incorporate in its entirety. Assuming we aim to utilize some kind of flexible technique, it could be good to try to characterize each contest's time series data by fitting a common function to it. Then, using the predicted function's parameters as new features in addition to the initial 21 header data features, we may be able to achieve a higher level of accuracy from the chosen technique.

Lastly, this dataset contains both categorical and numerical features, so whatever technique we apply will need to be able to handle both at once. Moreover, all the data we have can be easily labeled as "Success" or "Failure". This means we can directly evaluate the outputs against the known true values. Even better might be if the technique we apply could take advantage of having labeled data to further improve its predictive performance.

With all this in mind, we can conclude that an ideal solution (at least at the theoretical level) would be to use a flexible predictive modeling technique that can handle both numerical and categorical data simultaneously. It would also be preferred if it could utilize labeled data for improved performance and that it use curve fitted functions from the time series data to engineer new features along side the original set. In the realm of data science, one intuitive solution that meets all these criteria is machine learning.

2.3 Basic Data Science Techniques

These next few sections serve to provide background into key tools we used in our methodology. These data science techniques are commonly employed for the type of classification problem we have. To learn more about these tools, we delve first into machine learning, then decision trees and finally random forests.

2.3.1 A Brief Overview of Machine Learning

Machine learning is a field of applied mathematical statistics and analytics where computers are used to model the behavior of datasets in ways human minds cannot perceive. Machine learning has a wide range of applications from computer vision to weather forecasting. These methods rely on the use of known sample data, which is split into two non-overlapping data subsets: training and testing. The training set is fed to the learner, enabling it to identify trends and patterns in the data. The testing set is used to validate how predictive the learned trends were. These two sets should be completely distinct (share no data points between them), as the efficacy of prediction on the training set does not reflect the efficacy of prediction on future data. For this application, we are concerned with predicting the success of online fantasy sport contests hosted by DraftKings.

Supervised learning is a form of machine learning where both the inputs and outputs are known in the sample data (i.e. all data is labeled). Entry i of this set would come in the form of

$$(X_i, y_i) = (x_{1i}, x_{2i}, \cdots, x_{ji}, y_i), \quad X \in \mathbb{R}^{n \times j}, \quad \boldsymbol{y} \in \mathbb{R}^n \tag{2.1}$$

where y_i is the response value and x_{ji} is the j^{th} feature.

Supervised learning uses observed trends in a given training set with the aim of generating a mapping from its set of features X to an estimated set $\hat{y}$ that minimizes the prediction error to the known true values y. This mapping function can then be used on new sets of features to predict their response values with a fair degree of confidence before the true values are known. When the label is a finite set of discrete categories, this is known as classification.

Contest ID	**EntryFeeAmount**	**TopPrize**	...	**Response**
1932122	5.0	1000.0	...	Success
2494993	2.0	650.0	...	Failure
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$

Table 2.4: A sample binary response dataset similar to our actual dataset. Note the actual dataset includes 21 columns of the features listed in Table 2.2 and has over 600,000 entries.

In our dataset, each contest can be labeled a "Success" if it completely fills or a "Failure" if it does not. This problem is then a form of binary classification (classification into two possible categories).

While one function mapping the features to a response can be effective, a system of functions can often be even better. Ensemble learning is a machine learning technique which uses multiple weaker learners in parallel to collectively output a new, stronger prediction than any of those single learners could. Errors are reduced in ensemble learning because of the nature of the collection. If a single model makes an error based on the limited data it has, that error can be easily corrected by numerous other models which have other data. In this way, by measuring averages rather than individual results, an ensemble can be more effective and consistent for predictive modeling than any individual technique.

2.3.2 Decision Trees

Random Forests are one example of ensemble learning, utilizing many decision trees as their predictors. However, one must first understand the Decision Tree algorithm. Recall that for this application of supervised learning, there are a number of features and one binary response variable for each entry. A sample dataset can be seen in Table 2.4 for a binary classification problem like this. To simplify the example, consider a dataset with only two features and a binary categorical response variable, as appears in Figure 2.5.

In Figure 2.5, we see a set of 20 randomly generated points in 2-dimensional space, each labeled as either a success or a failure. The goal of a decision tree is to draw axis-parallel separators that minimize the number of incorrect classifications. Again referring to our example, we first draw a vertical line where Feature 1 = 0.25; classifying everything left of that value as a failure and everything right of that value a success. This results in the misclassification of 7 points, the fewest of any line we could have drawn. From here, the next split we make will only apply to one of the zones formed by the previous split. The next split we make for the right region is a horizontal line where Feature 2 = 0.3; classifying everything below the line a success and everything above it a failure. This results in five misclassifications, again the fewest of any possible line for that region. We can continue creating these cuts until each region only contains one point or until each region only contains points of a single class. This example can also be extended to a case with more than two features, but a cut can only ever be along one axis.

Tree learning centers on using known features (X) of the dataset to section the predictive

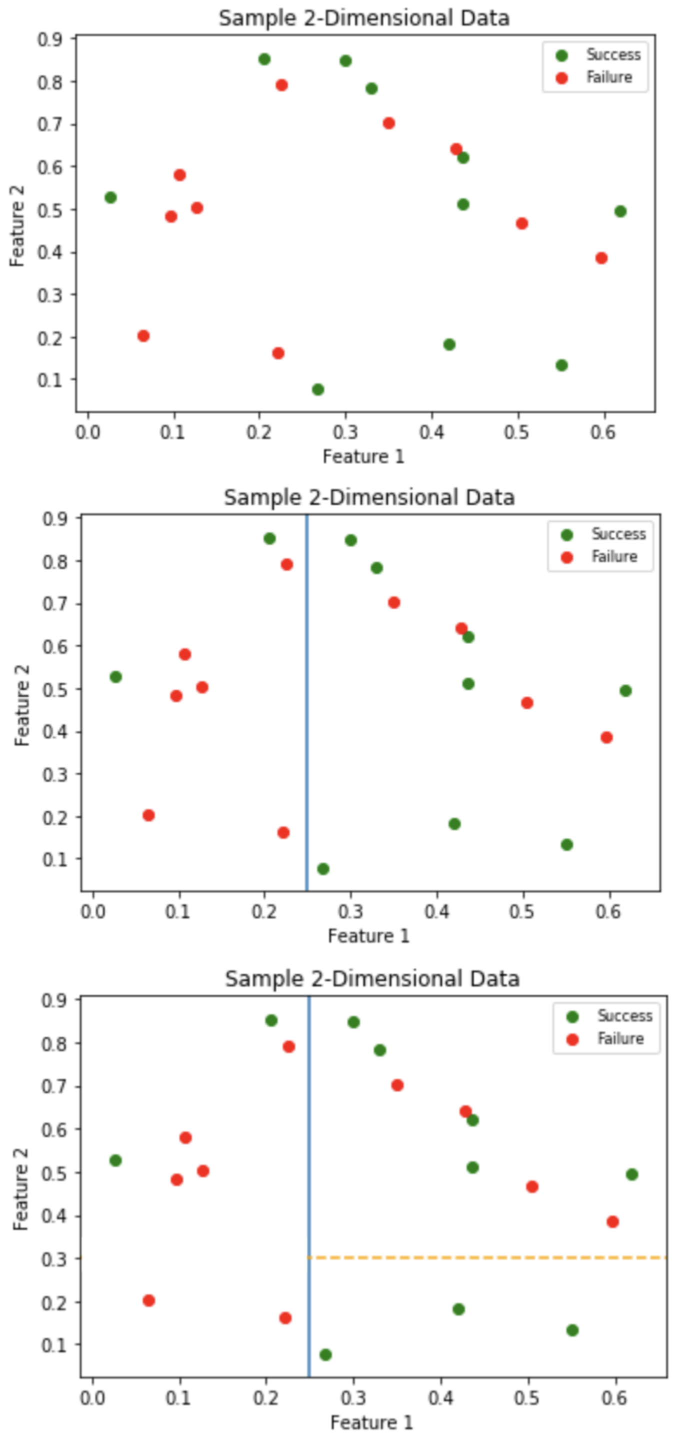


Figure 2.5: Sample set of 2-dimensional binary response data with two possible initial decision tree splits marked. In the top left image, we have only the data points. In the top right, the algorithm makes the cut that misclassifies the fewest number of points, at $x = 0.25$, classifying points with x values less than that as failures and greater as successes. The bottom picture is the second cut, again minimizing the mislabeled points. This process could continue until we only have regions with one point.

space into distinct regions [21]. Looking at the example dataset in Figure 2.5, we can see

$$X = (\boldsymbol{x_1}, \boldsymbol{x_2}), \quad \boldsymbol{y} \in [Success, Failure]$$

A line could be superimposed on Figure 2.5 where $x_1 = 0.25$. Then based on the sample data, we could predict that any point where $x_1 < 0.25$ should be a Failure. By repeatedly adding linear separators, it becomes possible to create multiple areas of classification. Certain criteria should be used to best split the space.

In our example, we start by partitioning the left most points because that cut limits the number of misclassifications, however, a more common way of choosing splits is by using the Gini Impurity (GI) [22]. Gini Impurity is a metric for the uniformity of response types in a region calculated by

$$GI = \sum_c p(y = c|x_i)(1 - p(y = c|x_i)) = 1 - \sum_c [p(y = c|x_i)]^2 \tag{2.2}$$

where $p(y = c|x_i)$ is the conditional probability of a point in a region being of response type c given some feature x_i. Similarly, $1 - p(y = c|x_i)$ is the conditional probability of a point in a region not being of response type c given feature x_i. For a given region R, $p(y = c|x_i)$ can be calculated as the percent of elements within R of class c, meaning $1 - p(y = c|x_i)$ is the percent of elements in R not of class c.

As the regions become more uniform, the impurity value decreases, eventually approaching 0 when all points in the region are of the same type. It it then possible to select the next split that best separates the data in one of the new regions by minimizing the sum of the Gini Impurities on either side of the new split.

In our example, we only consider the classes $c_1 = Success$ and $c_2 = Failure$. Since this is a binary classification, $(1 - p(y = Failure|x_i)) = p(y = Success|x_i))$ and vice versa. Substituting this into Equation 2.2 and simplifying, we find

$$GI = 2p(y = Success|x_i)p(y = Failure|x_i)) \tag{2.3}$$

From this, we can see that splitting at $x_1 = 0.25$ produces the lowest impurity sum and is thus the best option. After all splitting, a tree can be formed from the ordered list of splits which can be used to evaluate new points as either a "Success" or "Failure". We can also use decision trees for regression problems, where we would predict a numerical response at each cut, but for the purposes of this project, we need only consider the classification application.

A decision tree that uses all the features and can make as many branches as possible is likely to overfit, or build its predictions too closely off of the training data. This can be detrimental for future predictions as the overfitted tree will likely do well on the trained set and poorly on new data points in the testing set. Several techniques have been developed to prevent overfitting of decision trees including limiting the number of features provided to the algorithm and the number of branches (linear separators) it can produce. While a stunted tree such as this can help reduce overfitting, they are also prone to inaccuracy. To improve the predictive capabilities of this algorithm, it is common to use multiple stunted trees in parallel to form an aggregate prediction. The issue then becomes deciding how to effectively select the features and branching of each tree.

2.3.3 The Random Forest Algorithm

The Random Forest algorithm is a supervised ensemble learning method for problems in data science. The algorithm utilizes multiple randomly selected features in decision trees in order to predict outputs making it both easily implementable and interpretable. While random forests can be used for both regression and classification, for the purposes of this paper, we will only discuss classification.

In each tree, at each split k of the j features are randomly selected $k \leq j$. For the application of classification, k is often chosen to be $\sim \sqrt{j}$. The forest then considers all possible splits among those k features and selects the one that minimizes the impurity score. An important methodology to implement random forests is called bagging, or bootstrap aggregation [11]. Bootstrapping is the process by which a random sample is taken from the dataset for each model used in the ensemble. Each bootstrapped sample has the same number of elements n selected with replacement. This means each element in a bootstrap sample is selected randomly from the original data set without deleting it from the original set. This also ensures that each bootstrap sample preserves the approximate distribution of the original set. Thus, the general form of the data is maintained across all samples.

As an example, consider a random forest of 1000 decision trees where each tree receives a different bootstrapped sample set. A common implementation would split the trees into 10 equal batches of 100. Each batch would then have its branching limited by a different integer, creating an ensemble of trees with varying levels of complexity. Each tree (t) then gets a "vote" (weighted equally in most cases) as to the categorization of each point. The category receiving the majority vote from the set of all trees (T) is then predicted as the proper class.

$$\mathrm{v} = \frac{\sum_{\mathrm{i}=1}^{\mathrm{n}} \mathbb{I}_{\mathrm{i}}(\mathrm{t_i} = \mathrm{Success})}{|\mathrm{T}|}$$

$$\mathbb{I}_i = \begin{cases} 1, & t_i = Success \\ 0, & \text{otherwise} \end{cases}$$

$$\text{Prediction} = \begin{cases} Success, & \mathrm{v} \geq 0.5 \\ Failure, & \text{otherwise} \end{cases}$$

So, if 500 or more of the 1000 decision trees in our forest predict a point to be a success, it would be classified as a success.

The idea of bootstrapping may seem strange as it means each tree can receive different datasets, each of which may contain duplicate values. In fact, this sampling scheme ultimately improves the modeling performance by ensuring each tree is trained on different data. If each tree were trained on the same original dataset, the splits (and therefore predictive trends) would be similar or identical. This defeats the purpose of the random forest, as having many trees "voting" would be pointless if they always tend to vote the same. By bootstrapping, we ensure each tree receives a unique set of training data that is still representative of the original set allowing for decreased variance without increasing the bias of the model.

2.3.4 Advantages and Disadvantages

One distinct advantage of random forests is their flexibility. Forests are a non-parametric modeling technique, meaning they make no assumptions of the form of the data and therefore can work well for more complex sets of higher dimensional data. However, this comes at a cost. Due to their flexibility, random forests provide no insights into the nature of the data as trends cannot be discerned between features and the output response types, as opposed to a simple decision tree [4]. Additionally, depending on the complexity and number of trees, random forests can be computationally costly and are still at risk of overfitting.

2.4 Imbalanced Data

The proportion of occurrences belonging to each class in a dataset (class distribution) plays a key role in classification in Machine Learning. An imbalanced data problem refers to when a high priority class, (minority) infrequently appears in a dataset. This is due to another class instance (majority) outnumbering the minority class. This can lead to the evaluation criterion controlling the machine learning to treat minority class instances as noise, resulting in the loss of the classifiers ability to classify any new minority class instances [7]. Consider a dataset which has 1 member of the minority class to 100 members of the majority class. A classifier that maximizes accuracy and ignores imbalance will obtain an accuracy of about 99 percent by only predicting the majority class outcome. This section will go into more detail on how this problem occurs and the solutions investigated for this paper.

2.4.1 Class Imbalance Problem

This sections purpose is to refresh the reader's knowledge of supervised classification, to detail the Class Imbalance Problem and finally introduce a metric for performance evaluation.

Problem of Imbalanced Datasets

As stated, a dataset is said to be imbalanced when a minority class is underrepresented. When this occurs, standard classifiers tend to predict majority class for maximum accuracy. This is know as the class imbalance problem. However, the issue is more complicated than this. If a dataset is not skewed, meaning the dataset has significant set regions where only one class occurs, the class imbalance problem will not occur no matter how high the imbalanced ratio is. When a skewed data distribution does occurs, the problems of small sample size, overlapping and small disjuncts appear or are more relevant for minority class prediction. These problems collectively result in the class imbalance problem.

- Overlapping is when data samples belonging to different classes occupy the same space, making it difficult to effectively distinguish between different classes [25].
- Often the ratio between majority and minority class is so high that it can prove extremely difficult to record any minority class examples at all. Undersampling with these few instances can result in overfitting. In addition to overfitting, the bigger the imbalance ratio is, the stronger the bias to the majority class.
- Small disjuncts occur when the minority class instances are distributed in two or more feature spaces. This makes it harder to pin down where minority class instances are likely to occur.

Performance Evaluation

Traditionally, accuracy has been the metric for determining machine learning prediction efficiency. But, as stated before, accuracy is not the best metric when dealing with imbalanced data, as it may lead to removing minority class instances as noise. When working in imbalanced datasets, there exist better metrics to evaluate performance. The most common solution is to use a confusion matrix to measure the true positive rate, true negative rate, false positive rate, and false negative rate.

- True positive (minority) rate is the percentage of minority class correctly classified

  TPrate = True Positives / (True Positives + False Negatives)

- True negative (majority) rate is the percentage of majority class correctly classified

  TNrate = True Negative / (False Positives + True Negatives)

- False positive rate is the percentage of negative instances misclassified

  FPrate = False Positive / (False Positives + True Negatives)

- False negative rate is the percentage of positive instances misclassified

  FNrate = False Negative/(True Positives + False Negatives)

The goal in classification is to achieve high true positive rates and true negative rates. A common way of combining these results is through the use of a receiver operating character-

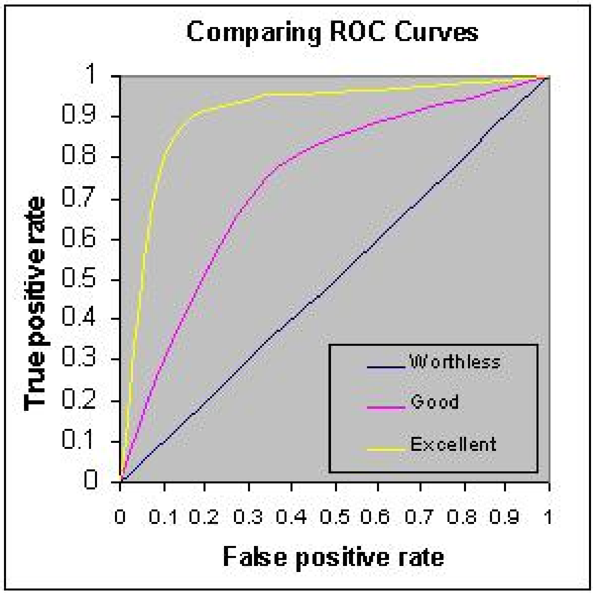


Figure 2.6: Examples of 3 different qualities of ROC curves. Yellow is an excellent curve representing a good ability to discern between classes. Purple is a useless curve equivalent to random classification of each point. The magenta line is better than the purple, but not nearly as good as yellow.

istic (ROC) curve. ROC curves serve as a metric of a classifier's ability to discern between classes. This allows for a visual representation of the trade-off between true positive and false positive rates. The area under a ROC "is equivalent to the probability that the classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance", so the goal is to maximize this area [6]. A desirable ROC would look like the yellow curve in Figure 2.6. Conversely, the purple line is the equivalent of randomly classifying each point; while it clearly is not the minimum possible area, it is by no means good.

2.4.2 Solutions to the Class Imbalance Problem

Class imbalance has emerged as one of the challenges in the data science community [2] [25]. Many real world classification problems attempt to classify infrequent instances such as fraud detection, medical diagnoses, and detection of oil spills. Many techniques have been proposed to solve imbalanced data problems, the majority of which fall into three groups: data-level, algorithm-level and cost sensitive learning.

Data-Level Techniques

Data level (Preprocessing) techniques are the most commonly use for solving imbalanced data problems. Data level solutions rebalance the class distribution by resampling the data space [2]. This solution avoids affecting the learning algorithm by decreasing the imbalanced ratio with a preprocessing step. This makes data level solutions extremely versatile as they are independent type of classifier used. The three preprocessing techniques considered for this project were Oversampling, Undersampling, and Hybrid Sampling.

Oversampling

Oversampling refers to any algorithm that rebalances a dataset by synthetically creating new minority class data. Oversampling is best paired with problems that have less data [8]. Two often used oversampling algorithms are Synthetic Minority Oversampling Technique (SMOTE) and random data duplication. SMOTE creates new data points by taking linear combinations of existing minority classes. Thus, SMOTE creates unique new data [14]. SMOTE is most effective for increasing the number of samples for clustered minority classes where Data duplication is much less biased. Comparatively, random data duplication does not create unique points, rather, it creates more instances of existing minority class points.

Undersampling

Undersampling is any algorithm that rebalances a dataset by removing majority class data points. This method is best for large amounts of data were data retention is less critical. The two most often used Undersampling algorithms are K-means clustering and Random Undersampling. Random Undersampling selects random majority class data points to remove. Similarly to the SMOTE and data duplication, K-means is best for clustered majority class data while random undersampling is best for extremely skewed data.

Hybrid Sampling

Hybrid sampling is the use of both Oversampling and Undersampling techniques to rebalance the dataset. The use of both oversampling and undersampling is selected normally over just undersampling in order to prevent the loss of large amounts of majority class data with little additional work to implement. To put this into context, lets take a example of data set with 100 points and an imbalance ratio of 99 to 1. We can undersample this data to have an imbalance ratio of 10 to 1. This is results in most of the majority class data being lost. Instead we can both Oversample and Undersample by duplication of the same minority class.

Algorithm-Level Techniques

Algorithm-level solutions adapt/rewrite the existing classifier learning algorithm to increase bias towards predicting the minority class [16]. Implementing these adaptations can be difficult and require a good knowledge of the imbalanced data problems discussed in the previous section.

Cost Sensitive Learning

Cost sensitive learning solutions are a hybrid of the previous two. They associate costs to instances and modifies the learning algorithm to accept costs. The cost of misclassified a minority class is higher than a majority class. This biases the classifier to the minority class as it seeks to minimize total cost errors. The flaw to this method is that it is difficult define exactly what these cost associations values should exactly be.

2.5 Linear Regression

In a dataset involving multiple variables, we can attempt to map a relation between them using linear regression. This technique can be used to model such a relation through its generation of a linear equation. In linear regression, one variable is considered to be the independent variable and the other the dependent variable. We typically denote the dependent, or response, variable as y and the independent, or explanatory, variable as x. Thus we can generate the linear equation

$$y = \beta x \tag{2.4}$$

where β is a real coefficient denoting the slope of the line. A more commonly used function includes an additional constant term (α) allowing the translation of the line turning it into an affine function. Adding α to Equation 2.4 yields

$$y = \alpha + \beta x \tag{2.5}$$

It is important to note that while a strong correlation may exist between x and y (we will explain in detail what that means later), this does not necessarily imply that x causes y or vice versa [11].

2.5.1 Least-Squares Regression

Most data in real world applications is discrete, or a countable number of points n, while a linear equation is a continuous approximation. How then do we generate such an equation given our dataset?

The Least-Squares Regression fits a continuous line to a discrete set of data by finding and minimizing the squared vertical distance between each data point and the line. This distance between the line and the observed data point is called the residual. A plot of the explanatory variable versus the residuals should have no discernible pattern. Otherwise, a linear model may not be the best fit for this data. A good example of residual plots that either indicate high goodness of fit or low goodness of fit can be found in Figure 2.7.

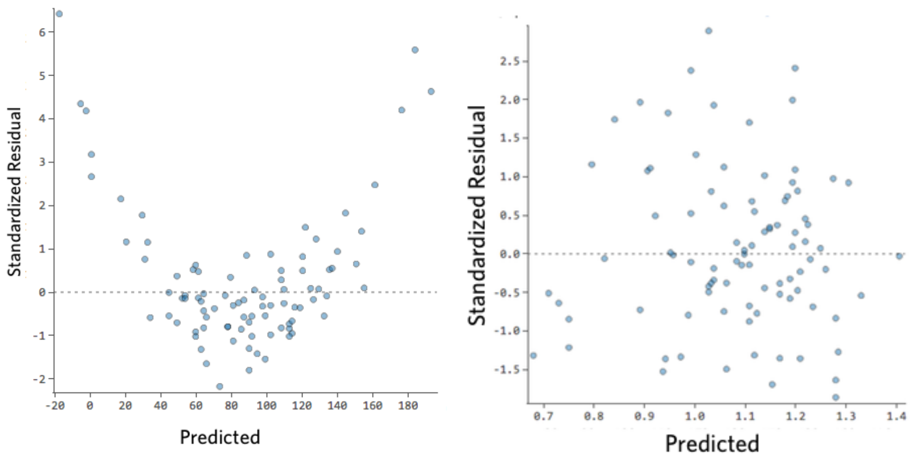


Figure 2.7: Two plots of residuals versus the explanatory variable, time. The figure on the left shows a residual plot that has a discernible quadratic pattern, suggesting a linear model is not a good fit. The figure on the right shows a residual plot that has no discernible pattern; that is it seems like random noise that could have been sampled from a Gaussian distribution where the variance is independent from the prediction. This suggests a linear model is a good fit.

In some cases, a transformation from the original data to something more linear may be appropriate. For example, if the observed data is approximately exponential in nature, a log transformation would take the data from the exponential space to linear space, where a Least-Squares linear fit would be appropriate. The linear parameters could then be converted back to exponential space.

To generate the least squares line, we consider the following system of equations. In the following example, our explanatory variable is t, for time, and our response variable is y. We have n discrete data points that are time/response pairs, and we are attempting to approximate α and β, the coefficients of the best fit line.

The Mean Square Error, which is minimized in linear regression, is given by the following equation:

$$\frac{1}{n}\sum_{i=1}^{n}(f(t_i) - y_i)^2 \tag{2.6}$$

a summation over all data points, where y is the observation and $f(t)$ is the function of time evaluated at that point. From this equation, called the objective equation, we can find a solution that minimizes mean squared error. To find the coefficients of the least squares line, we use the following equations [23].

$$H = \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ 1 & t_3 \\ \vdots & \vdots \\ 1 & t_n \end{bmatrix}, \quad H \in \mathbb{R}^{n\times 2} \tag{2.7}$$

$$\boldsymbol{y} = H \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \quad \boldsymbol{y} \in \mathbb{R}^n \tag{2.8}$$

Here, H is known as the design matrix. The first column vector is entirely 1s. The second column vector is all n known time values (t). Setting up H in this way makes it so Equation 2.8 is equivalent to

$$y_i = \alpha + \beta t_i \tag{2.9}$$

which is the exact functional form we were looking for in Equation 2.5.

For each observation point, we have some relation between time and the response which we can estimate with these two equations. To estimate our final relationship, the coefficients $\hat{\alpha}$ and $\hat{\beta}$ are found with the following equation.

$$\begin{bmatrix} \hat{\alpha} \\ \hat{\beta} \end{bmatrix} = (H^T H)^{-1} H^T \boldsymbol{y} \tag{2.10}$$

It should be noted that this example only involves prediction using a single input variable (t), however, the formulation is robust enough to be compatible with inputs of any dimension. Consider the case where the input is instead $X = (x_1, x_2, \ldots, x_i)$, thus we now want to predict for the equation

$$\boldsymbol{y} = \alpha + \beta X = \alpha + \beta_1 \boldsymbol{x_1} + \beta_2 \boldsymbol{x_2} + \ldots + \beta_i \boldsymbol{x_i} \tag{2.11}$$

H will then become

$$H = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1i} \\ 1 & x_{21} & x_{22} & \cdots & x_{2i} \\ 1 & x_{31} & x_{32} & \cdots & x_{3i} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{in} \end{bmatrix} \tag{2.12}$$

and Equation 2.10 will become

$$\begin{bmatrix} \hat{\alpha} \\ \hat{\beta}_1 \\ \hat{\beta}_2 \\ \vdots \\ \hat{\beta}_i \end{bmatrix} = (H^T H)^{-1} H^T \boldsymbol{y} \tag{2.13}$$

2.5.2 Extensions of Linear Regression

The variety of problems to which linear regression can be applied has created the need for various modifications to the technique. For example, there are some cases where certain points within a dataset must be weighted more heavily than others. This cannot be done in a traditional least squares, but can be achieved through the introduction of an additional diagonal matrix W. For a matrix to be diagonal, it must only have entries along its main diagonal as seen in Equation 2.14. For this application, the time series data is always provided in chronological order with newer points coming last. For the purposes of prediction, we care more about the more recent data as initial behavior would be expected to be non-influential on end behavior. W would then take the form

$$W = diag(\lambda^n, \lambda^{n-1}, \cdots, \lambda^1) = \begin{bmatrix} \lambda^n & 0 & 0 & \cdots & 0 \\ 0 & \lambda^{n-1} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \lambda^1 & 0 \\ 0 & 0 & \cdots & 0 & \lambda^0 \end{bmatrix} \tag{2.14}$$

where $\lambda \in (0, 1]$. The parameter λ acts as a "forgetting factor", making the later occurring points exponentially more important in the parameter search. Effectively, this means the residual used in the MSE calculation from Equation 2.6 for each point is weighted by its associated power of λ as in Equation 2.15.

$$\frac{1}{n} \sum_{i=1}^{n} (\lambda^{n-i}[f(t_i) - y_i])^2 \tag{2.15}$$

This allows for large residuals for the initial points while insisting on small residuals for the later points. The new equation to estimate the coefficients of our best fit line then becomes

$$\begin{bmatrix} \hat{\alpha} \\ \hat{\beta} \end{bmatrix} = (H^T W H)^{-1} H^T W \boldsymbol{y} \tag{2.16}$$

As before, this function can also be used for a set of i input variables reformatting the H and coefficient matrices as in Equations 2.12 and 2.13.

2.6 Kalman Filters

The concept of the Kalman filter was first published by Rudolph E. Kalman in 1960 in his paper "A new approach to linear filter and prediction problems". Kalman sought to create a new method of estimating linear dynamic systems that was more practical to use with machine computation. Kalman explains, "Present methods for solving the Wiener problem (linear dynamic systems) are subject to a number of limitations which seriously curtail their practical usefulness" [12]. Kalman details his newly invented algorithm which provides efficient computational means to recursively estimate the state and error covariance of a process, in a way that minimizes the mean of the squared error covariance [24]. The algorithm, now dubbed the Kalman filter, is a set of mathematical equations broken up into two steps: Prediction and Update, described in detail below.

Today the Kalman filter is used in several modern applications such as sensor fusion/filtering, data smoothing, and forecasting/prediction [12] [1] [19] [17]. The traditional Kalman Filter (KF) is a tool used to analyze a set of linear data points for an unknown dynamic model. Each data point is passed in 1-by-1 to the filter (thus the kth step involves the kth data point) using the following nomenclature:

$\boldsymbol{\hat{x}_{k\vert k}}$ $[\mathbb{R}^n]$:	the kth estimated state space given the first k observations $(z_1 \cdots z_k)$
$\boldsymbol{\hat{x}_{k\vert k-1}}$ $[\mathbb{R}^n]$:	the kth estimated state space given the first k-1 observations $(z_1 \cdots z_{k-1})$
F_k $[\mathbb{R}^{n\times n}]$:	the state transition function
B_k $[\mathbb{R}^{n\times n}]$:	the control input model
$\boldsymbol{u_k}$ $[\mathbb{R}^n]$:	the input control vector
$P_{k\vert k}$ $[\mathbb{R}^{n\times n}]$:	the error covariance matrix of (the confidence in) $x_{k\vert k}$
Q $[\mathbb{R}^{n\times n}]$:	the processing noise (confidence in each prediction)
H_k $[\mathbb{R}^{1\times n}]$:	the observation model at step k
z_k $[\mathbb{R}]$:	the kth observation
y_k $[\mathbb{R}]$:	the kth estimate residual
R $[\mathbb{R}]$:	the measurement noise (confidence in each observation)
s_k $[\mathbb{R}]$:	the innovation covariance
K_k $[\mathbb{R}^{n\times n}]$:	the Kalman gain

Assuming a state space with n parameters, each of these is a matrix of dimension $[\mathbb{R}^{d\times e}]$ meaning it has d rows and e columns [13]. For each dataset there exists a proper pairing of Q and R, however, they are usually not known. They represent artifacts of the Kalman filters' assumptions that the noise in the data is Gaussian (normally distributed) with mean 0 (i.e. noise). Q is the covariance matrix of a multivariate normal distribution centered at $\boldsymbol{\mu}$

$$\boldsymbol{\mu} = [\mu_1, \cdots, \mu_n], \quad \mu_i = 0 \tag{2.17}$$

where n is the number of elements in the state space $\boldsymbol{\hat{x}_k}$. Q then represents the assumed known variability in each of the n parameters in $\boldsymbol{\hat{x}_k}$. Larger entries in Q corresponds to larger variability in $\boldsymbol{x}$ which implies a lower confidence in the predicted $\boldsymbol{\hat{x}_k}$. R is the variance of a univariate normal distribution centered at 0. This acts as the assumed known variability in all observations z_k.

Given the appropriate values of Q and R, the KF acts as an optimal estimator as it minimizes the Mean Square Error (MSE) of the predicted $\hat{\boldsymbol{x}}$ [15]. In practice, this can be thought of as nearly equivalent to a recursive weighted least squares (WLS) estimate where Q acts as the forgetting factor for the KF similar to what W does in WLS. It should be noted this only works for our application because we provide the data in chronological order. In practice, the Kalman Filter can predict on data provided in any order so Q will more quickly "forget" the earlier data points provided. However, determining a proper $Q - R$ pair for a set of data can often be very difficult as it is still an open problem.

2.6.1 Parameter Estimation with Kalman Filters

Normally, the Kalman Filter is used to smooth out noise while maintaining the general form of the original data. As seen in Figure 2.8, the KF can take a set of noisy observations and reconstruct a good approximation of the true function's behavior. For a good example with step by step instructions on the implementation of how Kalman Filters are more traditionally meant to be used, we recommend viewing the SeatGeek price prediction article referenced in the bibliography. However, with minor adjustments to the algorithm, it can be converted from predicting function values to predicting function parameters.

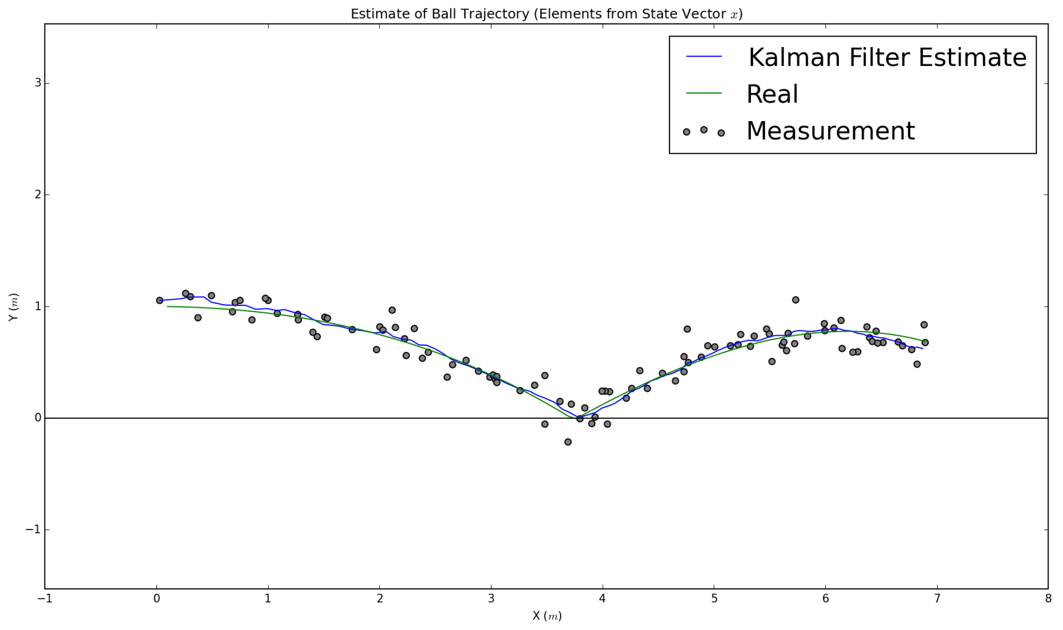


Figure 2.8: Example showing how the Kalman Filter is capable of taking in a set of noisy data (grey) and derives a smoothed estimate (blue) that is generally accurate to the true function (green).

We will assume we have a set of noisy linear data of the form (time, value) such that we want to find the best linear approximation of the form

$$v = \hat{\alpha} + \hat{\beta} t \tag{2.18}$$

that fits this data. We start with an initial state space estimate of $\hat{\boldsymbol{x}}_{\boldsymbol{0|0}}$, error covariance matrix estimate $P_{0|0}$ and chosen values of Q and R.

$$\boldsymbol{x}_{\boldsymbol{0|0}} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.19a}$$

$$P_k = \begin{bmatrix} P_\alpha & 0 \\ 0 & P_\beta \end{bmatrix} \tag{2.19b}$$

$$Q = \begin{bmatrix} Q_\alpha & 0 \\ 0 & Q_\beta \end{bmatrix} \tag{2.19c}$$

KF Prediction

The first step is prediction. We calculate $\boldsymbol{x}_{\boldsymbol{k|k-1}}$ and $P_{k|k-1}$ at the current iteration based on the last iteration's predicted $\boldsymbol{x}_{\boldsymbol{k-1|k-1}}$ and $P_{k-1|k-1}$.

$$\hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} = F_k \hat{\boldsymbol{x}}_{\boldsymbol{k-1|k-1}} + B_k \boldsymbol{u}_{\boldsymbol{k}} \tag{2.20a}$$
$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q \tag{2.20b}$$

For our purposes, assume F_k is always identity, meaning the model does not change with time. We also assume $B_k \boldsymbol{u}_{\boldsymbol{k}} = 0$.

$$F_k = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.21}$$

Since any matrix multiplied by the identity is always the original matrix, this reduces Equations 2.20a and 2.20b to the form

$$\hat{\boldsymbol{x}}_{\boldsymbol{k+1|k}} = \hat{\boldsymbol{x}}_{\boldsymbol{k|k}} \tag{2.22a}$$
$$P_{k+1|k} = P_{k|k} + Q \tag{2.22b}$$

This is a simplification in our application, as we know the true function is not constant as can be seen from Figures 2.9. These changes shift the filter from a dynamic to a nearly static model approximation. P represents the confidence in (or variability of) each parameter in the current state space with larger values of P implying less confidence in $\hat{\boldsymbol{x}}$. From Equation 2.22b, we can see that providing a larger Q causes consistently larger estimates of P. This makes sense as Q is a measure of the variability in each parameter of $\hat{\boldsymbol{x}}$, so larger Q's should cause the KF to be less confident in its predictions.

As a further simplification, we assumed P and Q to be 0 in the off diagonal as in Equations 2.19b and 2.19c. This assumes that the parameters α and β exist and change independent of each other where P_{α}, P_{β}, Q_{α}, and Q_{β} can be any non-negative real numbers.

KF Updating

The second step is updating. Each iteration of the KF utilizes a single data point, so the kth iteration will use the point (t_k, v_k). We begin by calculating the residual (or error from the kth known observation) for (t_k, v_k).

$$y_k = z_k - H_k \hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} \tag{2.23}$$

Here H_k is

$$H_k = \begin{bmatrix} 1 & t_k \end{bmatrix} \tag{2.24}$$

From Equation 2.19a, we can see

$$H_k \hat{\boldsymbol{x}}_{\boldsymbol{k|k-1}} = \alpha + \beta t_k = \hat{v}_k \tag{2.25}$$

Thus, y_k is simply the difference between the prediction of $\hat{v}_k$ at t_k and the actual observed v_k at t_k. We then perform the innovation step to calculate s_k. We understand s_k as a metric for the confidence in observation z_k as it represents the variability of the first k observations $(z_1, \cdots, z_k)$. If the values of $z_{1 \cdots k}$ tend to vary greatly, s_k will be large. If the values of $z_{1 \cdots k}$ only vary slightly, s_k will be small. Larger values of R also cause larger values of s_k as seen in Equation 2.26 since R is a measure of the variability for all observations.

$$s_k = R + H_k P_{k|k-1} H_k^T \tag{2.26}$$

P, H, and s come together to form K_k, the Kalman gain. Kalman gain can be thought of as a "velocity factor" of sorts for the KF, controlling the magnitude of adjustment to make to the current $\boldsymbol{\hat{x}_k}$. The formula for the optimal gain, minimizing the mean square error of the estimate, is

$$K_k = P_{k|k-1}H_k^T s_k^{-1} \tag{2.27}$$

From this we can see that as P_k gets small, so too does K_k. This is because a small P_k implies high confidence (or low variability) in $\boldsymbol{\hat{x}_k}$. Thus, having high confidence in the current state should yield only a small change to the new predicted $\boldsymbol{\hat{x}_k}$. We can also see that as s_k gets large, K_k gets small. This also makes sense since large s_k implies low observation confidence (or high observation variability). In that case we would want a smaller state adjustment for larger prediction errors as we don't trust the current observation as true (that is we want our state adjustment to be less sensitive to erroneous predictions).

We then improve the current state space estimate using information from the k^{th} iteration thus transitioning from $\boldsymbol{\hat{x}_{k|k-1}}$ to $\boldsymbol{\hat{x}_{k|k}}$

$$\boldsymbol{\Delta\hat{x}_k} = K_k y_k \tag{2.28a}$$

$$\boldsymbol{\hat{x}_{k|k}} = \boldsymbol{\hat{x}_{k|k-1}} + \boldsymbol{\Delta\hat{x}_k} \tag{2.28b}$$

From Equation 2.28a we see that y_k controls the sign of the state prediction adjustment. When $z_k < H_k\boldsymbol{\hat{x}_{k|k-1}}$, $y_k < 0$ making $\boldsymbol{\Delta\hat{x}_k}$ negative. This means if $\boldsymbol{\hat{x}_{k|k-1}}$ over/underestimates z_k, the new $\boldsymbol{\hat{x}_{k|k}}$ will respond accordingly. We can also see that the magnitudes of y_k and K_k control the magnitude of $\boldsymbol{\Delta\hat{x}_k}$.

The same improvement is done for P, changing $P_{k|k-1}$ to $P_{k|k}$ by

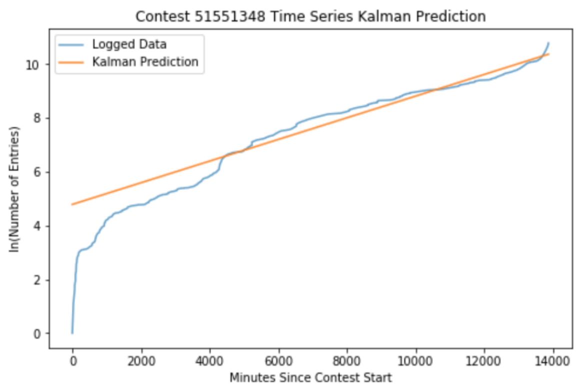
$$P_{k|k} = (I_2 - K_kH_k)P_{k|k-1}(I_2 - K_kH_k)^T + K_kRK_k^T \tag{2.29}$$

where I_2 is the 2-by-2 identity matrix. When using the optimal Kalman gain, as we do, this calculation can be reduced to

$$P_{k|k} = (I_2 - K_kH_k)P_{k|k-1} \tag{2.30}$$

Each iteration k will use the previous iterations estimates of $\boldsymbol{\hat{x}_{k-1|k-1}}$ and $P_{k-1|k-1}$ as the new starting guess for $\boldsymbol{\hat{x}}$ and P while maintaining the same Q and R throughout. Once completed for all data points, the final $\boldsymbol{\hat{x}_{k|k}}$ is treated as the model prediction. We can then use those values of $\hat{\alpha}$ and $\hat{\beta}$ to forecast what the value will be for some time in the future. Since the KF processes data point by point, data can be fed in in any order with the forgetting factor Q weighting the later processed points more heavily. Our time series data comes in chronological order, so Q allows us to essentially weight the more recent data more heavily. This is effectively equivalent to performing a WLS fit.

Figure 2.9 shows an example of a line whose parameters were found using the KF with an all zero Q along side the same contest fit with a line by ordinary least squares. Both approaches predict virtually the same line. However, if Q is changed such that $Q_\alpha = 0.3$ instead of 0 as in Figure 2.10, we can see the behavior changes significantly.



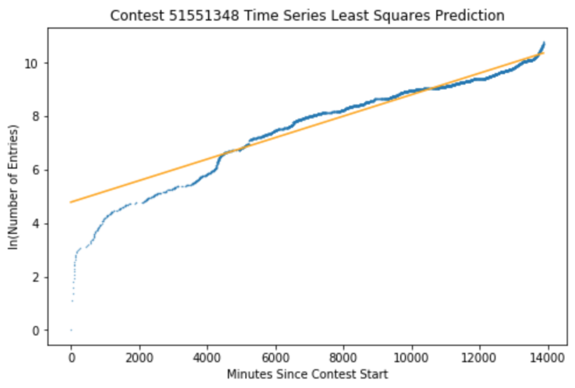


Figure 2.9: The left image shows a linear regression performed on a real contest using the Kalman Filter with a Q of all 0s. The right image shows a linear regression found using a weighted least squares fit on the same contest with $\lambda = 1$. This is the case of no forgetting factor for both methods which can be seen to produce what looks like the same approximating function.

Performing the fit by KF can be convenient for time series data as each iteration involves processing only one data point at a time. And while least squares is not very computationally intensive, it still requires redoing the entire dataset calculation when updating the parameter prediction.

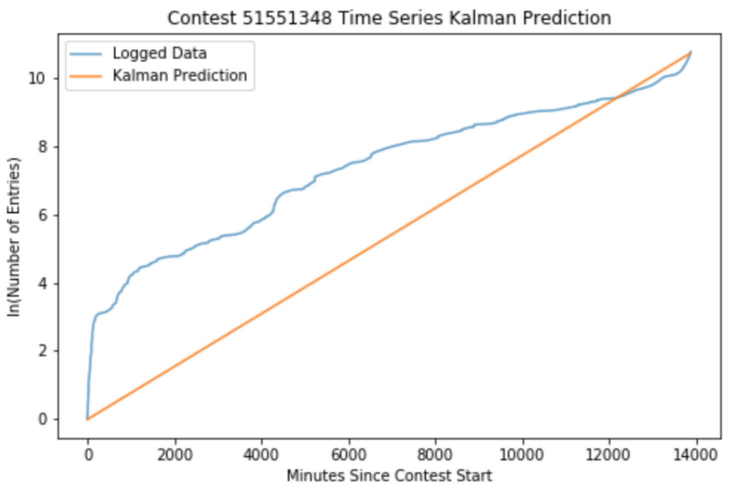


Figure 2.10: This is the same contest from Figure 2.9 except $Q_\alpha = 0.3$. We can see that even a small change in the Q matrix produces a significantly different fitting function.

2.6.2 Extended Kalman Filter

While the KF is an excellent method for estimating model parameters, it is limited in the same way as LS in that it can only predict for linear models. This works for estimations on logged data, however, it would be preferable to be able to directly estimate the parameters of the non-linear exponential function $\alpha e^{\beta t}$. To do this, we turn to the Extended Kalman Filter (EKF). The EKF works exactly the same as the normal KF except it can work with non-linear model functions. The only difference between the KF and EKF in this application is the values of H_k. For the EKF, H_k takes the form

$$H_k = \begin{bmatrix} \frac{\partial M(t)}{\partial B_1} & \cdots & \frac{\partial M(t)}{\partial B_i} \end{bmatrix} \tag{2.31}$$

where $B_1 \cdots B_i$ are the values of the state space $\hat{\boldsymbol{x}}$ and $M(t)$ is the non-linear model function. In our case, we assume $M(t) = \alpha e^{\beta t}$, thus

$$H_k = \begin{bmatrix} e^{\beta t} & \beta \alpha e^{\beta t} \end{bmatrix} \tag{2.32}$$

Otherwise the calculations are exactly the same as for the ordinary KF. Figure 2.11 shows some exponential model predictions using the EKF. $R = 30$ for both, but the left one has $Q_\alpha = Q_\beta = 0$ while the right has $Q_\alpha = Q_\beta = 10$.

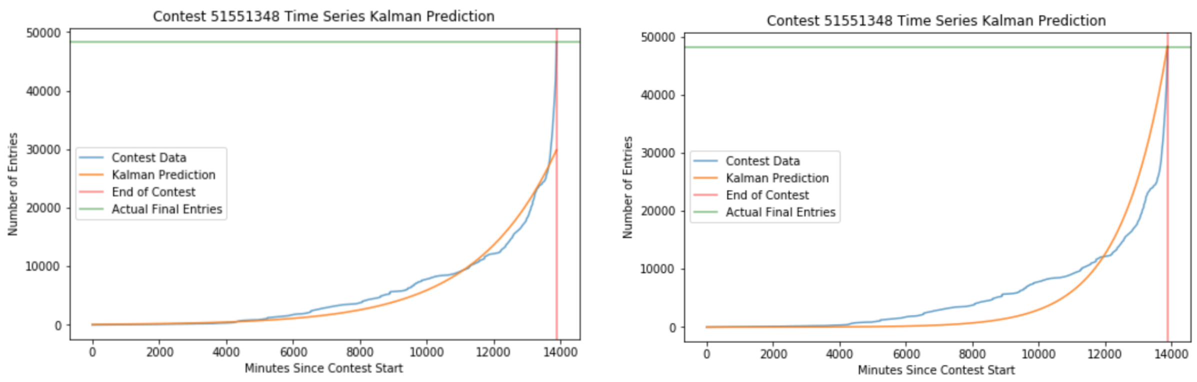


Figure 2.11: This is the same contest from Figure 2.9 with $R = 30$ for both. The left image is an exponential function found using a Q of all 0's. The right image shows the exponential function found when $Q_\alpha = Q_\beta = 10$. Again, we can see how changing the Q matrix produces a significantly different fitting function.

Methodology

3.1 Time Series Data Processing

The total data cleaning and classification occurred over 5 major steps: Time Series Data Processing, Exponential Model Fitting, Final Dataset Setup, Classification Prediction, Performance Evaluation. In the following chapter, we explore each step, detailing the techniques we utilized to generate our results.

3.1.1 Data Chunking

In its original format, the time series data was organized by month. This meant that a single contest could have its data split into multiple files. Knowing that this organization would involve extra load time for processing contests, we split the time series data into chunks based on each series' ContestIds. Each chunk consists of approximately 10,000 contests, and were saved with a naming scheme "chunkN.csv" where N is a positive natural number. In total, we ended with 65 chunks of data. Reorganizing the data as such ensures that each chunk contained every data point for each of its assigned contests, which will save time when loading data. Along with the new chunk formatting, we generated a chunk map: a csv that lists each contest's id and the name of the chunk file that contest is found in.

3.1.2 Data Cleaning

The original time series data format had columns for "Minutes Remaining" (minutes remaining in contest) and "Entries in Last Minute" (number of entries received in that minute) as shown in Table 3.1. For our purposes, we preferred to instead have the data in the form of "Minutes Since Start" (minutes since contest opened) and "Summed Entries" (total number of entries since contest opened). To do this, we performed a cumulative summation per contest in chronological order over the number of entries at each minute.

We reformatted the time column, calculating the "Time Since Start" value by subtracting the current minutes remaining from the maximum minutes remaining. This inverts the

Contest ID	Minutes Remaining	Entries in Last Minute
10486	400	2
10486	395	1
10486	394	3
⋮	⋮	⋮
10486	210	5
⋮	⋮	⋮
10486	0	4

Table 3.1: A representative set of fake time series data as it would have appeared in the originally provided file. Note this reflects only a single contest while the actual files included one full months worth of contests.

Contest ID	Minutes Since Start	Summed Entries	4 Hours Out
10486	0	2	1
10486	5	3	1
10486	6	6	1
⋮	⋮	⋮	⋮
10486	190	60	0
⋮	⋮	⋮	⋮
10486	400	310	0

Table 3.2: This is the same representative set of fake data after processing "Minutes Remaining" into "Minutes Since Start" by subtracting the "Minutes Remaining" value from the maximum "Minutes Remaining" value. "Entries in Last Minute" was also transformed into "Summed Entries" by taking a cumulative sum of entry values. A fourth "4 Hours Out" boolean column was added with value 1 if "Minutes Remaining" was ≥ 240 and 0 otherwise.

numbering so time starts from 0 and increases until the contest closes. Additionally, we added a Boolean column for each point to tell whether that point occurs in the last 240 minutes (4 hours) of the contest or not. A value of 1 means the point occurs before 4 hours remaining (Time Remaining > 240), 0 otherwise. This column will be used later to separate out which part of the time series occurs before there are 4 hours left in the contest (The time we want to make a prediction at). This separation is intended to simulate the data that would be available when we need to predict a contest's success. Altogether, the structure for each contest is changed from what's seen in Table 3.1 to something more like Table 3.2.

We also go through each contest and scale "Minutes Since Start" and "Summed Entries" to 100 by dividing each point by the max value of its column, then multiplying by 100. When we later fit exponential models to the data, this scaled format will ensure that every model is in the same range, holding the predictive power consistent across contests of varying sizes. The final scaled values from Table 3.2 can be found in Table 3.3.

Contest ID	Minutes Since Start	Summed Entries	4 Hours Out
10486	0	0.6452	1
10486	1.25	0.9677	1
10486	1.5	1.935	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
10486	47.5	19.35	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$
10486	100	100	0

Table 3.3: This is the same representative fake data set after scaling both "Minute Since Start" and "Summed Entries" to 100 in order to standardize the range of values in each column.

3.2 Exponential Model Fitting

We found the time data tended to have the form of a noisy exponential, as can be seen in Figures 2.5 - 2.7, since the cumulative sum of entries grows monotonically and most rapidly towards the end of each contest. To keep the model simple, we opted to use an exponential model of the form $\alpha e^{\beta t}$. This was meant to capture the basic nature of entries growing more rapidly with time. We made predictions with exponential fits in two ways: a series of Weighted Least Square (WLS) estimates (varying values of λ) and a series of Kalman Filter (KF) estimates (varying values of the Q matrix).

3.2.1 Least Squares

For each contest, 15 WLS estimates were performed using the following set of 15 λ values. These values were chosen to ensure a wide range of values were applied in the hope that more information could be drawn from them.

Starting with data of the form found in Table 3.3, we begin by excluding all data collected after the "4 Hours Out" point (i.e. we only used the data where "4 Hours Out" = 1). This allowed us to pretend as if we were analyzing a live contest 4 hours before it closed. We then changed the "Summed Entries" data by taking its natural log to convert it from pseudo exponential to pseudo linear. This made fitting with WLS possible as it can only be applied to linear functions. The result of this can be seen in Figures 2.9 and 2.10 with Figure 2.11 showing the original form of the data. The final adjusted data set for performing WLS on Table 3.3 can be seen in Table 3.5. Using this information, we could then create the design matrix (H), weighting matrix (W), and response matrix (y) as described in Section 2.5. These matrices, as they apply to our example dataset from Table 3.5, can be found in Equations 3.1 - 3.3.

$\lambda1$	0.1
$\lambda2$	0.2
$\lambda3$	0.3
$\lambda4$	0.4
$\lambda5$	0.5
$\lambda6$	0.6
$\lambda7$	0.7
$\lambda8$	0.8
$\lambda9$	0.9
$\lambda10$	0.99
$\lambda11$	0.999
$\lambda12$	0.9999
$\lambda13$	0.99999
$\lambda14$	0.999999
$\lambda15$	0.9999999

Table 3.4: The set of λ values used in WLS to predict the parameters and final number of entries for each contest.

Contest ID	**Minutes Since Start**	**ln(Summed Entries)**	**4 Hours Out**
10486	0	-0.43819	1
10486	1.25	-0.03283	1
10486	1.5	0.6601	1
⋮	⋮	⋮	⋮
10486	47.5	2.9627	0
⋮	⋮	⋮	⋮
10486	100	4.6052	0

Table 3.5: The same representative fake data set after taking the natural log of the "Summed Entries" column in order to put the data in a pseudo linear space.

$$H = \begin{bmatrix} 1 & 0 \\ 1 & 1.25 \\ 1 & 1.5 \\ \vdots & \vdots \\ 1 & 47.5 \\ \vdots & \vdots \\ 1 & 100 \end{bmatrix} \tag{3.1}$$

$$W = \begin{bmatrix} \lambda^{100} \sim 0 & 0 & 0 & \cdots & 0 \\ 0 & \lambda^{98.75} \sim 0 & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \lambda^{0.25} = & 0 \\ 0 & 0 & \cdots & 0 & \lambda^0 = 1 \end{bmatrix} \tag{3.2}$$

$$y = \begin{bmatrix} -0.43819 \\ -0.03283 \\ 0.6601 \\ \vdots \\ 2.9627 \\ \vdots \\ 4.6052 \end{bmatrix} \tag{3.3}$$

It should be noted that while normal WLS requires constant interval data (i.e. $x_{i+1} - x_i = c$ where x_{i+1} and x_i are consecutive time entries and c is some constant) our data tends to have sporadic intervals as values are only recorded at times when people enter the contest. To get around this, instead of the normal method for choosing powers of λ described in Section 2.6.1, we calculated powers of lambda by subtracting the relevant time value from the maximum time value (which is always 100 after being scaled during preprocessing). Thus the ith entry of the diagonal matrix W becomes

$$W_i = \lambda^{100-x_i} \tag{3.4}$$

This effectively assigns the same powers of λ as if the data were interpolated (were made to have constant intervals by filling in gaps with linear approximations based on the two surrounding points), without the need for interpolation. It also ensured that contests with thousands of entries did not cause massive exponents which can cause massively small values for bases less-than 1. (Even 0.8^{1000} is on the order of 10^{-97})

Since the dimensions of H, W, and y are dependent on the number of entries in a given contest, we wanted to avoid the computational load of having to generate 15 W matrices in $\mathbb{R}^{n\times n}$ where n can be over 1000 for each contest. We instead created an augmented version of the H^T matrix. As seen in Equation 2.12, H^T is always right-multiplied by W . Taking advantage of this, we opted to merge the powers of λ directly into H^T. This then left us with three distinct matrices with H and y remaining the same and H^T becoming H_W^T where each column is multiplied by its respective power of λ. The WLS formulation then became

$$\begin{bmatrix}\hat{\alpha}\\ \hat{\beta}\end{bmatrix} = (H_W^T H)^{-1} H_W^T y \tag{3.5}$$

For our current example, W and H from equations 3.1 and 3.2 merge to form H_W^T as seen in Equation 3.6.

$$H_W^T = \begin{bmatrix}\lambda^{100} & \lambda^{98.75} & \lambda^{98.5} & \cdots & \lambda^{0}\\ 0\lambda^{100} & 1.25\lambda^{98.75} & 1.5\lambda^{98.5} & \cdots & 100\lambda^{0}\end{bmatrix} \tag{3.6}$$

With this structure established, we simply then performed each of the 15 WLS parameter estimates outputting 15 pairs of A and B values. Since this estimate was done in log space, we then converted each set of parameters back to normal space. Our WLS predicted the optimal A and B for the line

$$ln(y) = \alpha + \beta x \tag{3.7}$$

To convert this, we raise both sides to powers of e, thus

$$y = e^{\alpha + \beta x} \tag{3.8}$$

which is equivalent to

$$y = e^{\alpha} e^{\beta x} \tag{3.9}$$

Here, β doesn't change as it remains in the exponential. To transition back to normal space, we needed only to calculate a new α; $\alpha' = e^{\alpha}$. We concatenated the α' and β values generated from each value of λ into a dataframe. In the event that no data exists beyond the "4 Hours Out" mark, we set the values of α' and β to 0. The WLS can also output values *nan* or *inf* for "not a number" or "infinity" respectively in some cases. We deal with these later on.

3.2.2 Kalman Filter

Our application of the Extended Kalman Filter for non-linear parameter estimation follows much the same steps as the WLS implementation. Just like WLS, we ran each contest's time data using 15 different Q matrices with $R = 1$ in all cases.

These were chosen by performing an exhaustive search over values of R, Q_α and Q_β on a set of 20 contests which included various sports, lengths, entry fees and total entries. R ranged exponentially in the form 2^N with $N \in \{0, ..., 10\}$ and Q_α and Q_β each ranged exponentially

Label	Q_α	Q_β
v1	8000	60
v2	9000	60
v3	100000000	10
v4	10000000	100
v5	1000000	100
v6	10000000	10
v7	1000	1000
v8	1000000000	10
v9	10000000000	10
v10	100000000	32
v11	3981072	16
v12	208929613	13
v13	794328234	251
v14	60000	58000
v15	2691534	1778

Table 3.6: The set of values on the main diagonal of the Q matrix used in KF to predict the parameters and final number of entries for each contest. While we recognize these values may seem arbitrary, they were chosen with the explicit goal of providing a wide variety of predictions from which a machine learner may be able to derive trends.

in the form 10^N with $N \in \{0, ..., 10\}$. Each set of R_i, $Q_{\alpha i}$ and $Q_{B\beta i}$ was evaluated by the sum of residuals using three weighting schemes: a flat weighting where all residuals are weighted equally, a linear weighting where more recent data is weighted linearly more than older data, and an exponential weighting where more recent data is weighted exponentially more than older data. From analyzing these results, we selected 15 that appeared to most often reduce the sum of errors. It should be noted that this method of selecting Q values will not necessarily produce statistically "proper" Q's. It was our intent to have Kalman Filters with a variety of Q values, whose diversity could be instructive to a random forest. As long as a single Kalman Filter behaves consistently across contests, a Random Forest could theoretically use the noise in the prediction for its own predictive power.

Since the Extended Kalman Filter is able to perform estimations on non-linear data, we did not need to convert to log space beforehand. In our example, this means the data from Table 3.3 will work as is. Once again, we filter out contests that were recorded less than four hours before the contest ends (we only use data where "4 Hours Out" = 1). In this case, the output α and β values are already in normal space so no conversion is required. We concatenated the α and β values generated from each value of Q into a dataframe. In the event that no data exists beyond the "4 Hours Out" mark, we set the values of α' and β to 0. The Kalman Filter can also output values *nan* or *inf* for "not a number" or "infinity" respectively in some cases. We deal with these later on.

$$\hat{y}_{final} = \alpha e^{\beta 100} \tag{3.10}$$

Feature	**Reason for Removing**
GameSet	Proved unimportant
ContestName	Complex string with information already present in other features
ContestStartDatetimeEST	Proved irrelevant for the user
ContestSEndDatetimeEST	Proved irrelevant for the user
ContestPayoutDatetimeEST	Proved irrelevant for the user
Entries	This would be cheating to know ahead of time

Table 3.7: The list of features removed during processing because they allow for unknowable information or were determined to be irrelevant to contest users.

since the duration (or final recorded time) of all contests is 100 after scaling. Figure 3.1 visualizes the full progression of time series data processing used for both our Kalman Filters and Weighted Least Square predictions.

3.3 Final Data Set Setup

At this stage, we preprocess the header data into a more usable format. We also bring together all the data we've calculated so far and conglomerate it into our single final dataset.

We start by removing unusable / irrelevant columns. Features such as "Contest Name" were full of incomparable Strings which would be unlikely to hold predictive power. Time parameters also were not relevant to users and were determined to also be unusable for the purposes of prediction.

In addition to removing ineffective features, we also removed features that would allow the predictor to "cheat" (i.e. use information that could not be known before the end of a contest). Features like final number of entries were removed to ensure the predictions were made using fair and viable data. The full list of removed features appears in Table 3.7.

Next we standardized all numerical value columns. Features such as "Entry Fee" and "Total Prize" can take any positive value so we divide each numerical column by its respective maximum value to put them in the range of 0 to 1. We also had to translate the categorical columns into a format a Random Forest could handle. We chose to create a column for every categorical value. Each contest has a 1 in the columns of the categories it fit in and 0 in the rest. This sort of splitting is known as one-hot encoding and is extremely helpful in instances where Strings cannot be interpreted.

We found that 2 sets of "Sport Name" needed to be merged. "SOCC" and "SOC" both represent soccer contests, the naming code was simply changed at one point. "PGA" and "GOLF" were also both present in data. Cases such as these with equivalent categories were merged into singular respective columns.

Next, we created our "Success" column. It was necessary to have an easy-to-access metric for whether a given contest met its goal. DraftKings informed us that they measure a contests'

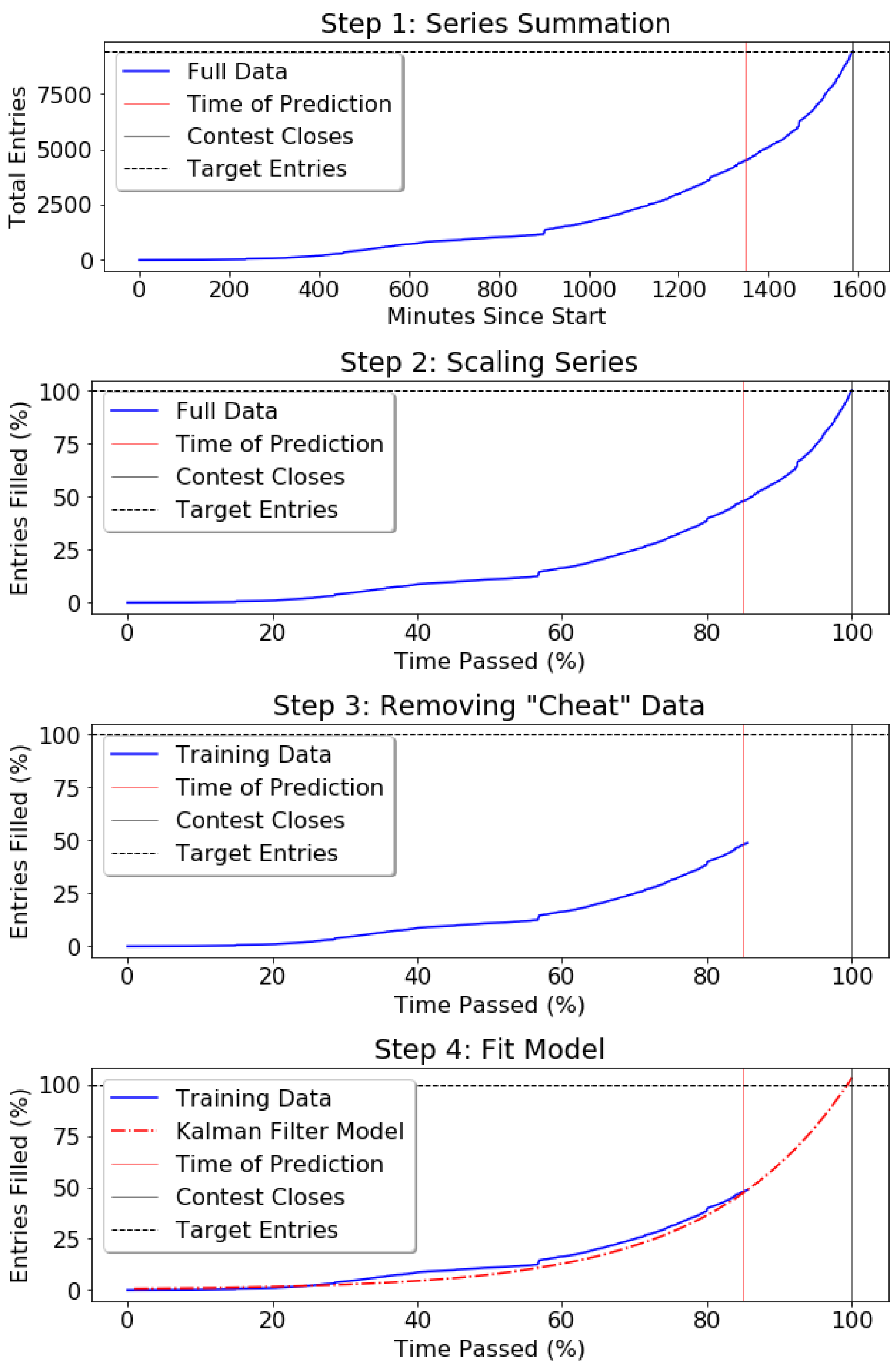


Figure 3.1: This is a visual progression showing the steps we take in getting a prediction from each Kalman Filter. The methodology is identical for Weighted Least Squares except we take the natural log of the scaled entries in order to convert to a pseudo linear space.

success as whether the number of entries matched the player limit. In accordance, we created a new Boolean “Success” feature measuring whether the total number of entries equalled the max number of players allowed.

The last step in processing data was to determine predictions based on both the Weighted Least Squares and Kalman Filter results. Both algorithms were set up to return an α and β value from our model $\alpha e^{\beta x}$. Since all of our time series data was scaled 0-to-100 on both axes, predictions are simply calculated as $\alpha e^{\beta \times 100}$. This computation was performed on all 15 versions of both the WLS and KF.

We combined each of the above-mentioned processes in our final dataframe. The final set of features included the standardized numerical features, one-hot encoded features, “Success” column, and the α and β parameters of each WLS and KF with their associated percent-full predictions.

3.4 Classification Prediction

Before performing any sort of classification, we first divided the whole data set into five subsets. The largest subset (about 40 percent) was the training set. This set was used for all training of each iteration of our classifiers. The rest of the data was split into four separate sets. These four sets were used for the purposes of cross-validation of results. For predictive purposes, we created an ensemble of 20 random forests using the scikit learn RandomForestClassifier module. Each forest was built using scikit learn’s default parameters and contained 100 trees.

If you recall, our dataset is highly imbalanced with approximately 91% of contests filling successfully. If we were to train our forest on such a skewed dataset, it would inevitably over-fit. That is to say that it would tend to predict “Success” simply because there were many successful contests in the training set. To address this, we start by balancing the training set via undersampling. To undersample, we get the number of “Successes” and “Fails” in the dataset. Whichever occurs less often, we select all training points of that type and sample a set of equal size from the other type. In our case, failures are always less common, so we select all failures then randomly sample a number of successes to match.

We implemented additional parameters to control the minimum acceptable accuracy for each classifier. If a forest is found to be less accurate than the minimum, it is discarded and new versions are generated until the desired number of forests have been collected. This accuracy is measured as the percent of contests correctly labeled by the Random Forest from the training set.

Contest	**Forest 1**	**Forest 2**	**Forest 3**	**Forest 4**	**Forest 5**	**True Value**
Contest 1	1	1	1	1	1	1
Contest 2	0	1	0	0	1	0
Contest 3	0	0	0	1	0	1
Contest 4	1	1	1	0	0	1
Contest 5	0	1	0	1	1	0
Contest 6	0	1	1	0	1	0
Contest 7	1	0	1	1	1	0
Contest 8	1	1	0	0	0	1
Contest 9	1	1	0	1	1	0
Contest 10	1	0	0	0	1	1

Table 3.8: An example set of predictions from 5 different Random Forests on 10 different contests.

Contest	**Overall Prediction**	**True Value**
Contest 1	1	1
Contest 2	0	0
Contest 3	0	1
Contest 4	1	1
Contest 5	0	0
Contest 6	1	0
Contest 7	1	0
Contest 8	0	1
Contest 9	1	0
Contest 10	1	1

Table 3.9: One possible set of overall predictions from the example data in Table 3.7 for when the threshold for predicting a value of 1 is 60% (i.e. 3 of the forests must predict 1 in order to have an overall prediction of 1).

3.5 Performance Evaluation

To evaluate the efficacy of each classifier, we made a function to extract the values from the confusion matrix of the ensemble of 20 forests. This function then calculates the True Positive and False Positive Rates to be plotted in a ROC curve. Here, the threshold we vary is the percent of Random Forests in the whole ensemble required to predict “True” before the ensemble as a whole will predict “True” (where True means we predict the contest will not fill). By plotting multiple ROC curves on the same graph, we found it relatively easy to deduce which methodology of classification was most effective. A better performing predictor will always tend towards the top left corner (i.e. have a larger area under its curve).

Consider the example dataset in Table 3.7. If we use a threshold of 50%, the overall prediction matrix appears in Table 3.8. Comparing these values to the known True Values, we can produce the confusion matrix in Table 3.9. From this, we can see that with a threshold of

	Predict 0	**Predict 1**
Actually 0	2	3
Actually 1	4	1

Table 3.10: The confusion matrix for the set of predictions shown in Table 3.8 when compared against their respective known True Values.

60% we get an overall accuracy of 30% with a True Positive Rate of 0.4 and a False Positive Rate of 0.8. Expanding this to all possible thresholds we get the blue ROC curve found in Figure 3.2.

Since this curve appears to be concave up (as opposed to concave down) the area under the curve is fairly small. In fact, because it falls below the line of True Positive Rate = False Negative Rate, we can conclude that the classifier does a very poor job of discerning between 1 and 0 response types and would be better off randomly guessing. However, we could also invert this line by inverting every predicted value. Doing so would produce the orange curve seen in Figure 3.2 which appears to do reasonably well at classification based on the increased area under its curve.

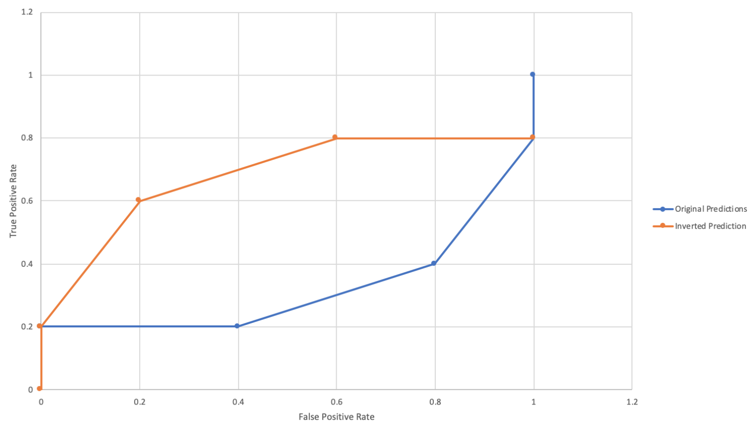


Figure 3.2: In Blue: the ROC curve generated by varying the prediction threshold for the ensemble of forests. In Orange: the ROC curve generated by taking the opposite of every prediction for each threshold of the ensemble of forests. It should be noted that because there were only 5 example forests, only 5 points appear on the curve causing it to b jagged. Given more forests, there could be more possible thresholds and thus a smoother curve.

Results

Recall from Section 2.4 that a receiver operating characteristic (ROC) curve is a metric for the ability of a classifier to discern between classes, where a larger area under the curve indicates better predictive performance. The y-axis is the true positive rate (sensitivity) and the x-axis is the false positive rate (1 - specificity). In this chapter, we discuss the changing predictive ability of our classifier as it went through various iterations visualized through ROC curves on four distinct validation sets.

For each of the below curves generated using our ensemble method, the threshold was the required percent of random forests in the ensemble to flag a contest as a positive for it to be labeled as a positive. (Once again, a flagged contest is one that we predict will not fill to its target entries within the time that it is open to entries). Since our ensemble used 20 forests, each ROC was made from 20 threshold values. A ROC curve representing an optimal algorithm will flag every True Positive (TP) contest without flagging a single False Positive (FP). It is nearly impossible to create an algorithm with perfect sensitivity and specificity, so our goal was simply to maximize the area below the ROC curve. In the following graphics, many of the ROC curves are rainbow colored. This coloring is to help identify the threshold used to produce the indicated TP and FP rate pair.

4.1 Header Data Results

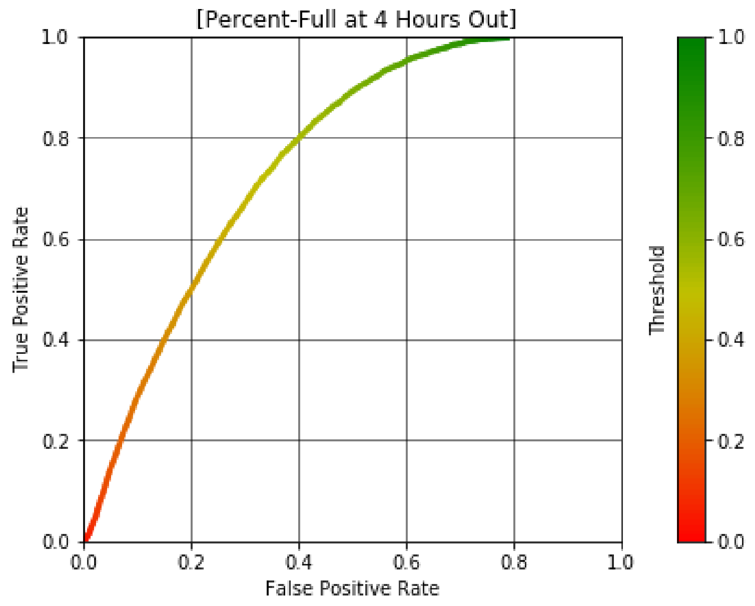


Figure 4.1: ROC generated using a threshold of percent filled at the "4 Hour Out" point. This represents an initial baseline estimation we needed to outperform. The fact that such a simple edtimate appears to form a decent result already is a good way of showing just how imbalanced the data set is.

To ensure we weren't over-complicating our approach, we started by generating a simple baseline ROC curve. Figure 4.1 was generated for this purpose by using "Percent Full at 4 Hours Out" as our threshold for each contest. At a threshold of 50%, any contest which was less than 50% full at the Time Series data point 4 hours before contest start was flagged as a positive. At that threshold, approximately 80% of True Positives are flagged with only 40% of False Positives flagged. This visual is a good way of showing the dataset is inherently imbalanced as it would seem we can achieve a decent prediction by simply evaluating each contest at the "4 Hour Out" mark with no need for a complicated prediction. Nevertheless, we aimed to improve upon it.

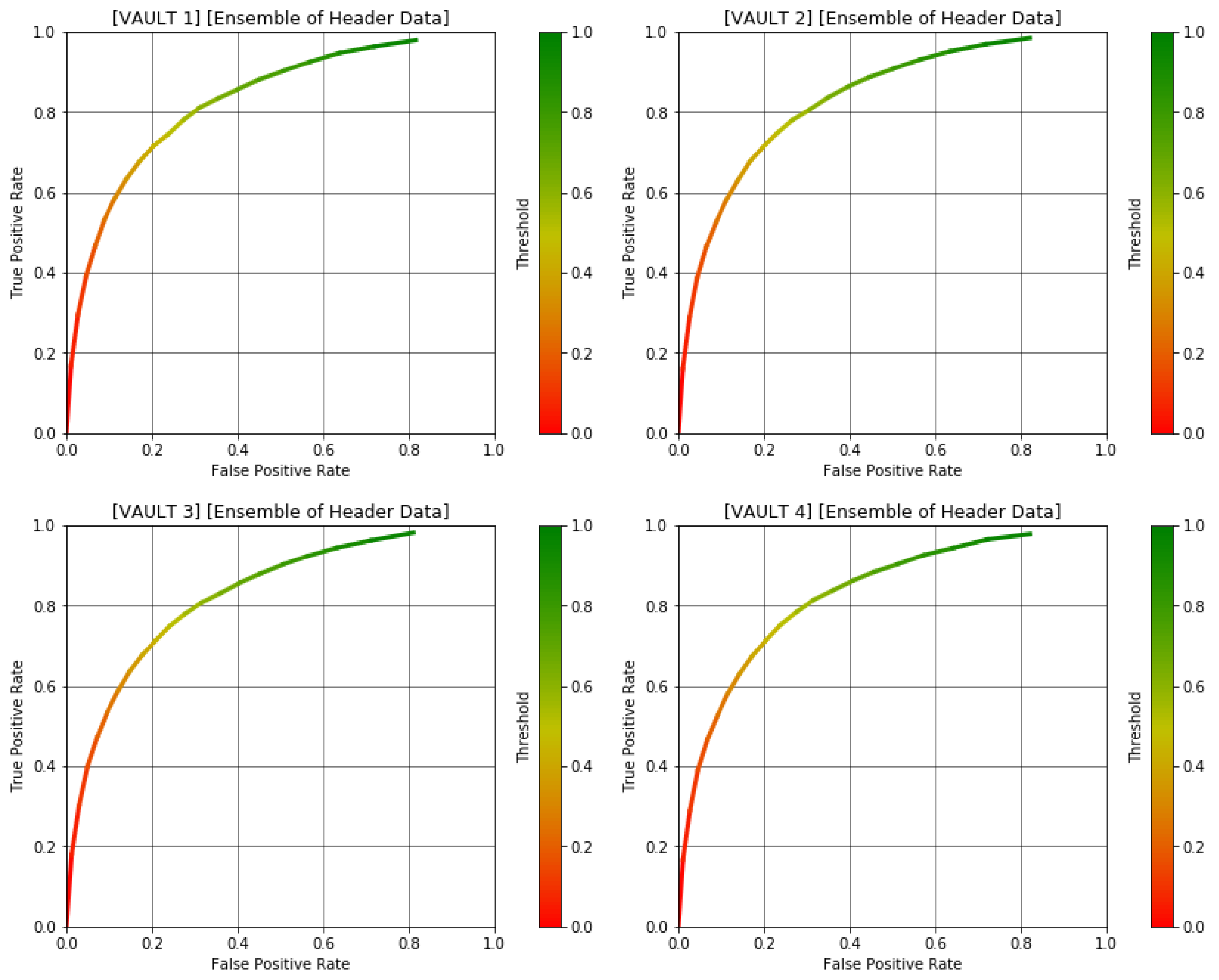


Figure 4.2: ROCs generated based on our ensemble of random forests using only processed Header data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.2 was generated using processed Header data in our ensemble of 20 Random Forest Classifiers (RFC). At a threshold of 50%, approximately 80% of True Positives are flagged with only 30% False Positives being flagged in all four validation sets. These ROC curves consistently have 10% better specificity (True Positive rate) than that of Figure 4.1 at equivalent thresholds. Figure 4.3 shows a direct comparison between the baseline curve and the curve from predicting using an ensemble of processed Header data for each validation set.

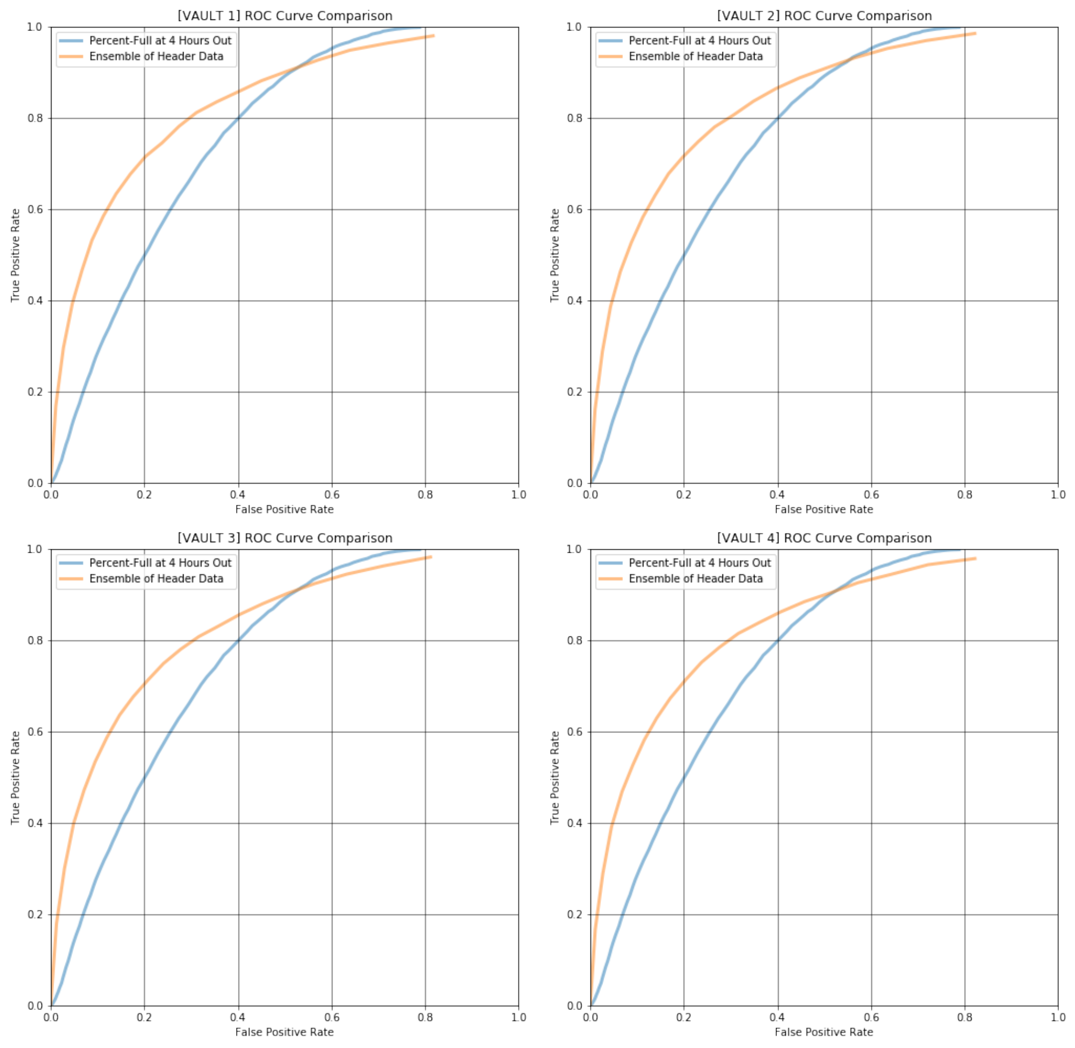


Figure 4.3: Comparison of ROCs generated based on our ensemble of random forests using only processed Header data against the baseline. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figures 4.2 and 4.3 indicate that using an ensemble of Header data is useful for classifying contests. It could be tempting to see this result and be satisfied with it, however, we should remember that this does not yet include any of the Time Series Data. Theoretically, a model which can effectively utilize both sets of data should be able to perform even better.

4.2 Time Series Results

Figures 4.4 - 4.10 were generated by feeding the time series data into our Kalman Filter (KF) and Weighted Least Squares (WLS) methods with varying Q and λ parameters respectively. In these graphs, both methods predicted final percent-fill by averaging the predictions from all 15 KF and all 15 WLS. The threshold used in all these ROC's was the overall predicted percent-fill.

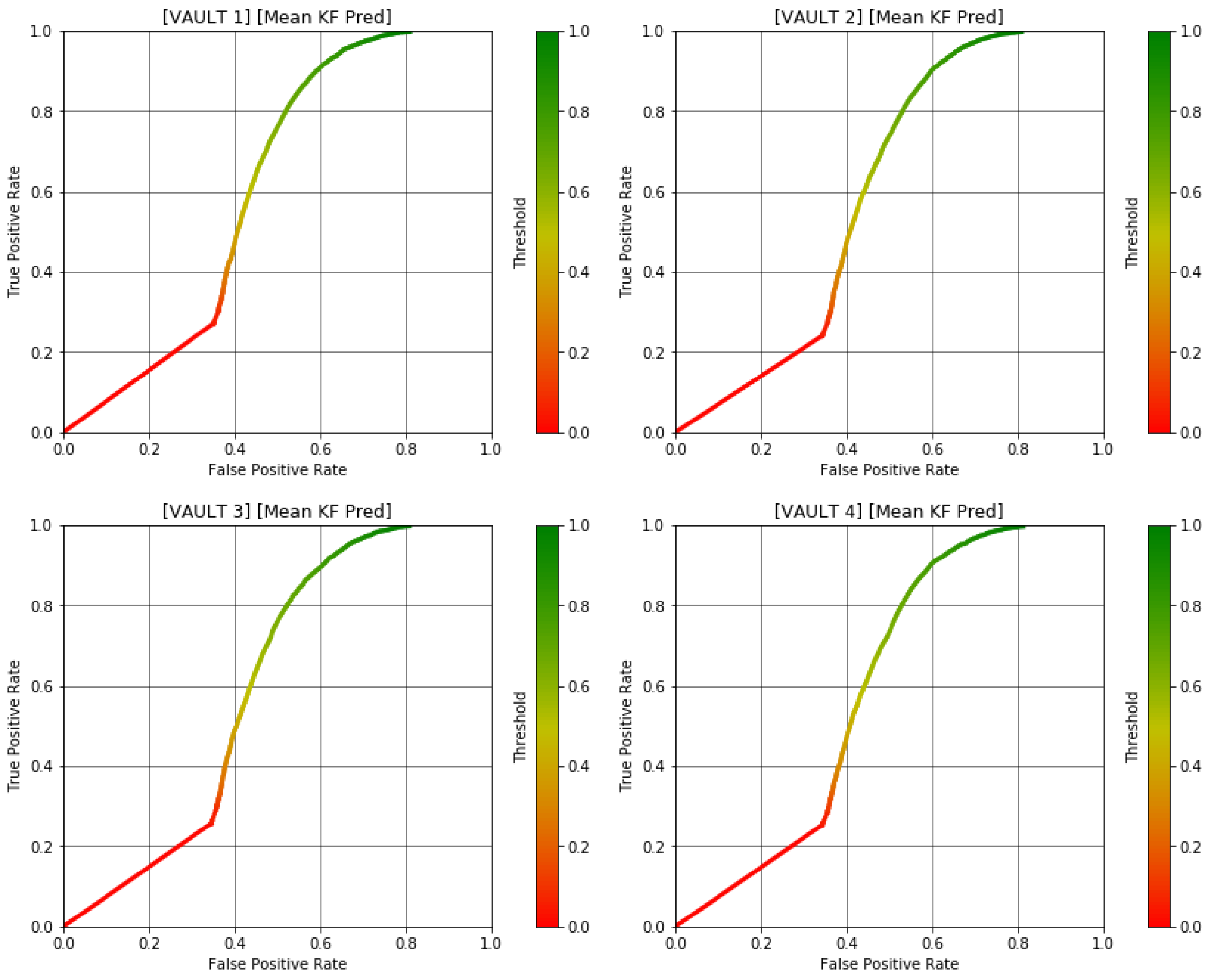


Figure 4.4: ROCs generated based on the average percent-fill prediction of all 15 of our KF's on the entirety of each validation set. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

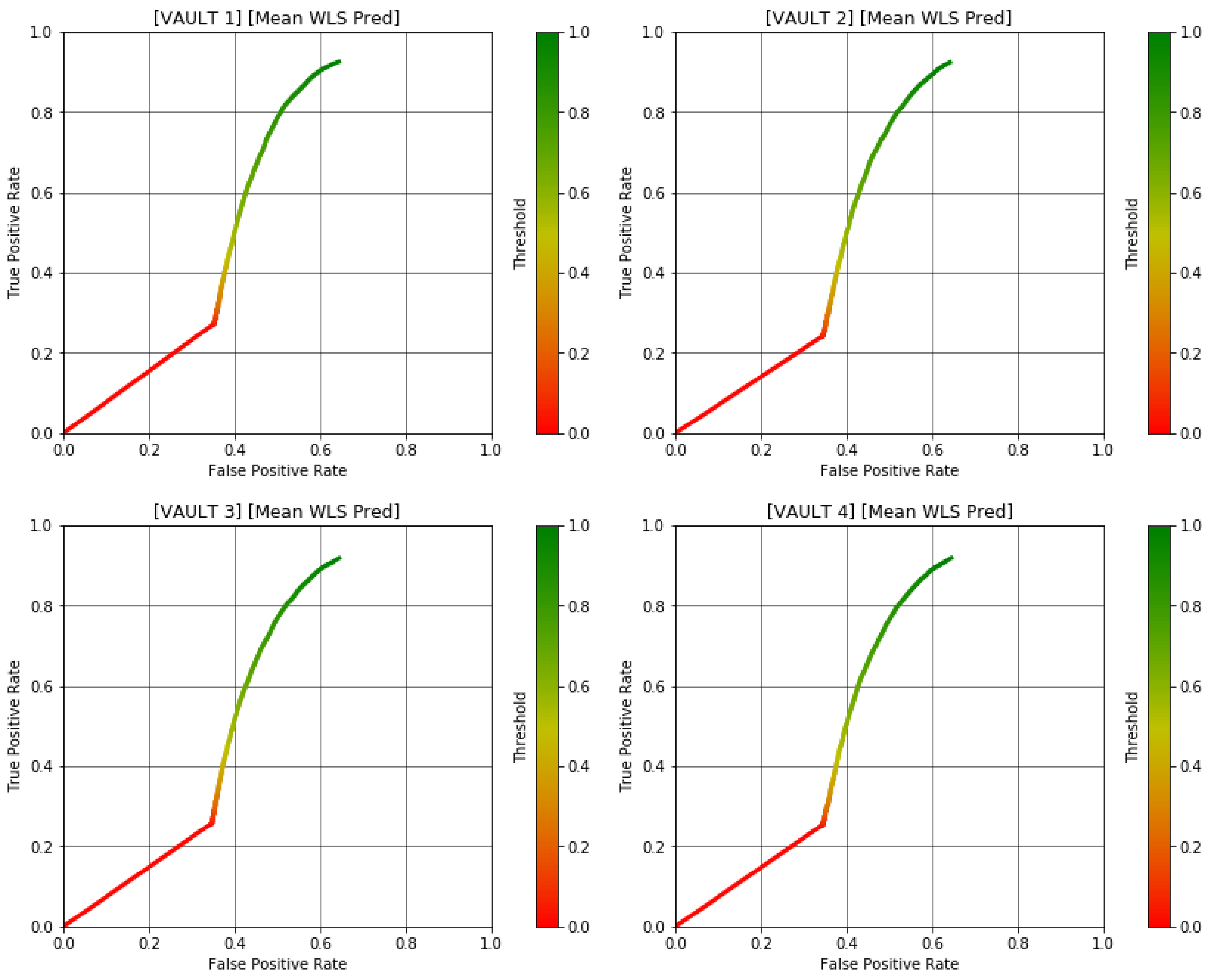


Figure 4.5: ROCs generated based on the average percent-fill prediction of all 15 of our WLS's on the entirety of each validation set. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.4 shows the result of using the averaged percent-fill predictions of all 15 KF's on all 4 validation sets. Similarly, Figure 4.5 shows the result of using the averaged percent-fill predictions of all 15 WLS's on all 4 validation sets. Immediately, we can see the averaged prediction of our 15 KF and WLS independently perform worse than even the baseline from Figure 4.1. A direct comparison of all three can be found in Figure 4.6. It should be noted that KF and WLS produce nearly identical curves, with KF tending to perform slightly better in the False Positive range of around 0.4 - 0.5. This may imply both methods are equal or very nearly equal in terms of predictive performance. This would be expected as both perform an exponential fit on the data, just in different ways.

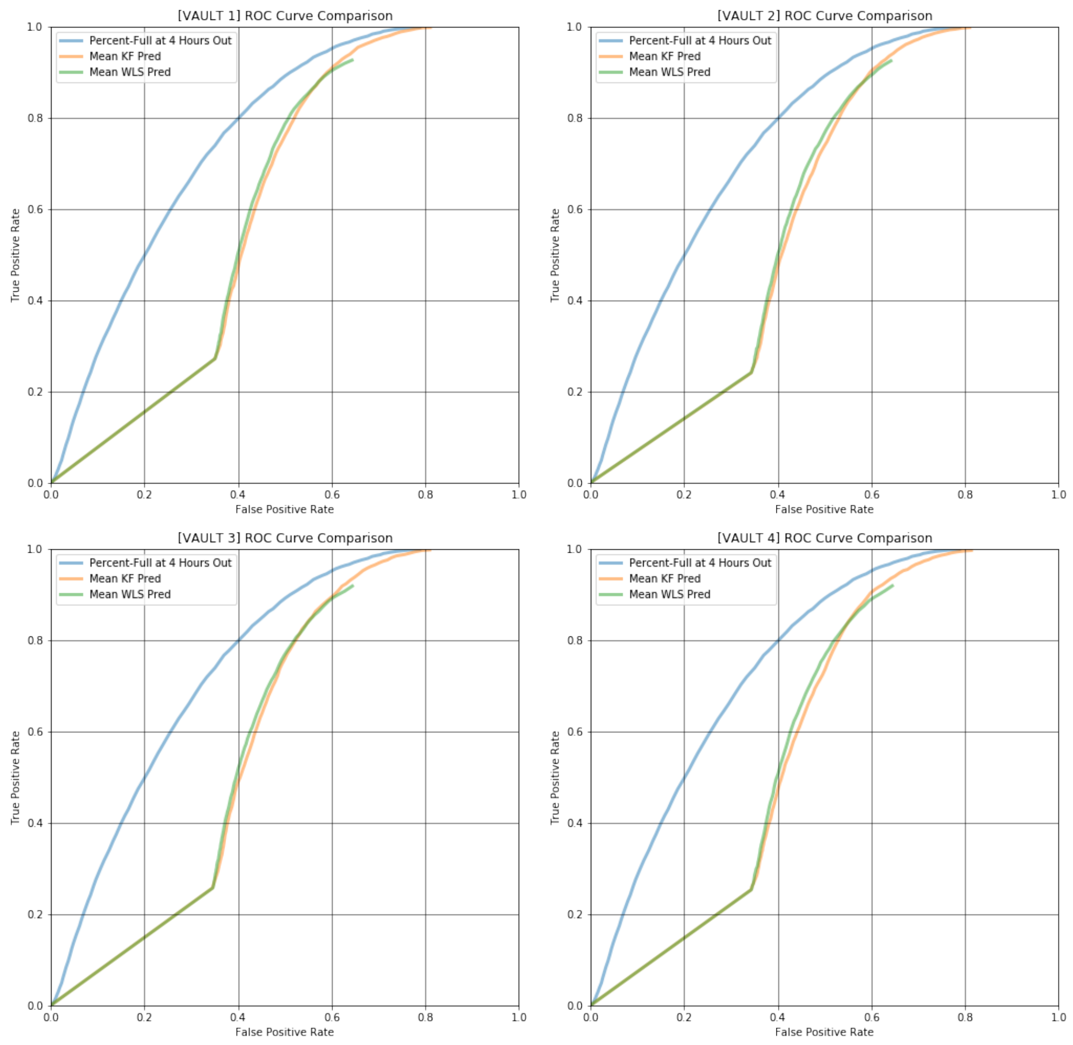


Figure 4.6: Comparison of ROCs generated based on the average percent-fill prediction of all 15 of our KF's and all 15 of our WLS's separately against our baseline curve. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

One aspect all these graphics appear to share is a long, flat section at the beginning. We believed this piece appeared because there were a large portion of contests which had no time series data available before the "4 Hour Out" point. This meant our KF and WLS methods could not make any predictions as they had no starting data. In order to get a better sense for how well averaging the KF and WLS predictions performs, we recreated Figures 4.4 - 4.7 considering only contests which had data before the "4 Hour Out" point. The results of limiting to only "non-zero" data as we call it, (contests with data before "4 Hours Out") can be found in Figures 4.8 - 4.10.

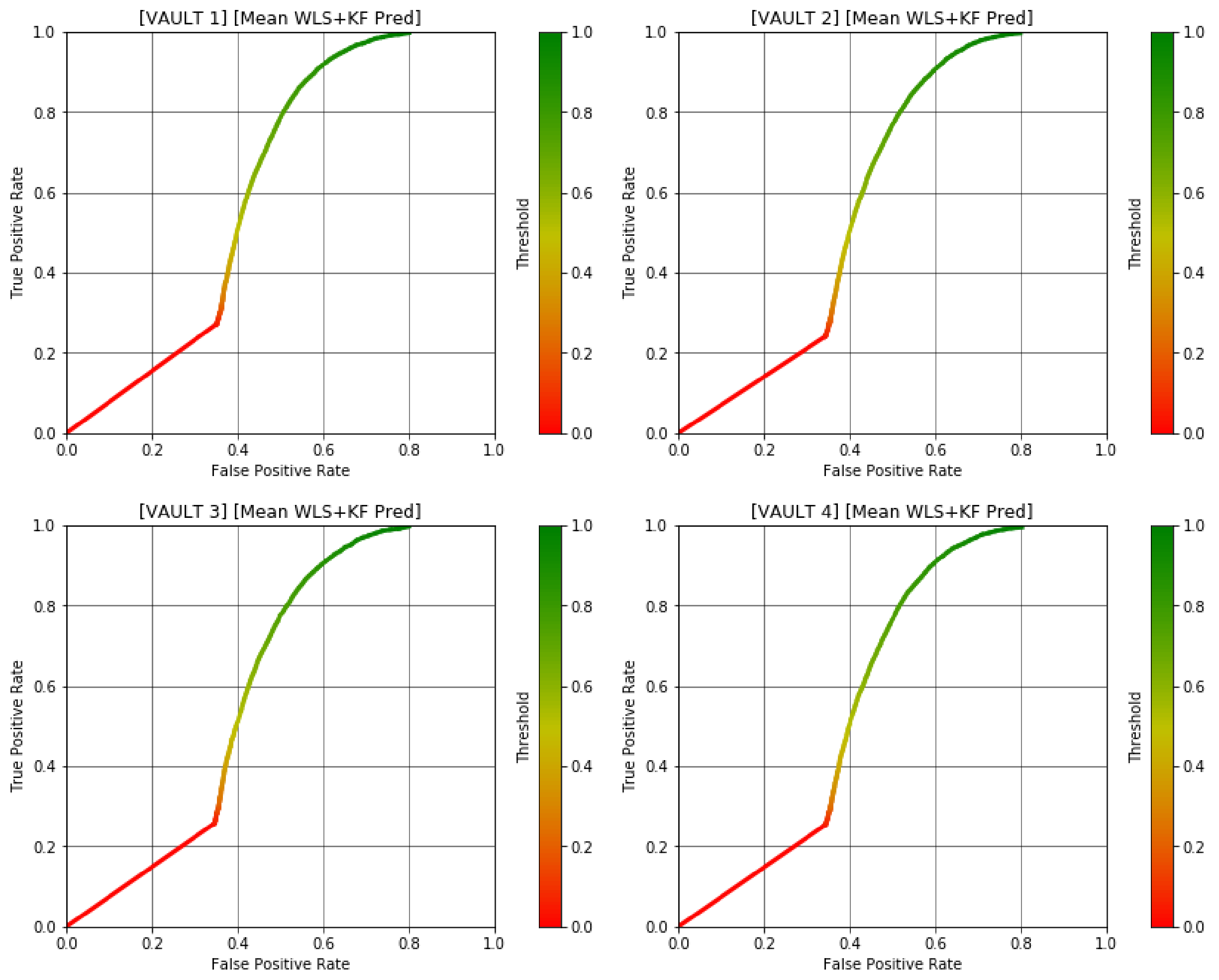


Figure 4.7: ROCs generated based on the average percent-fill prediction of all 15 of our KF's and all 15 of our WLS's combined for the entirety of each validation set. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

We thought averaging both the KF and WLS together may improve the result. Figure 4.7 shows the outcome of averaging both together. As might be expected, the result was nearly identical to those previous with the initial flat section remaining and no apparent improvement over the baseline.

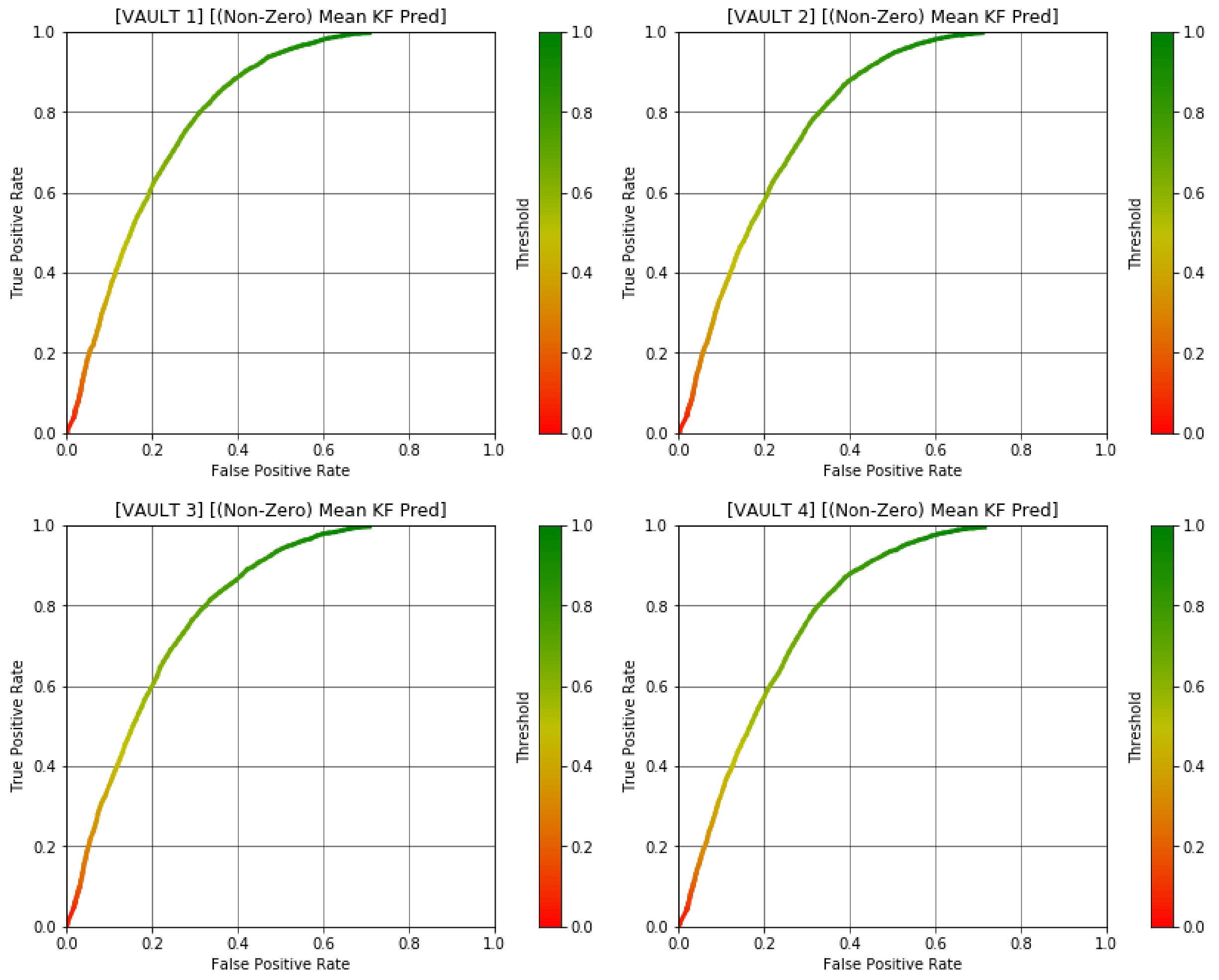


Figure 4.8: ROCs generated based on the average percent-fill prediction of all 15 of our KF's on just non-zero data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.8 is the same as Figure 4.4, except it only uses "non-zero" data. We can plainly see significant performance improvements overall. At a False Positive Rate of 20%, non-zero data performs 20% better in terms of True Positive Rate. This would seem to bolster our theory that the abundance of dataless contests was detrimental for the average prediction.

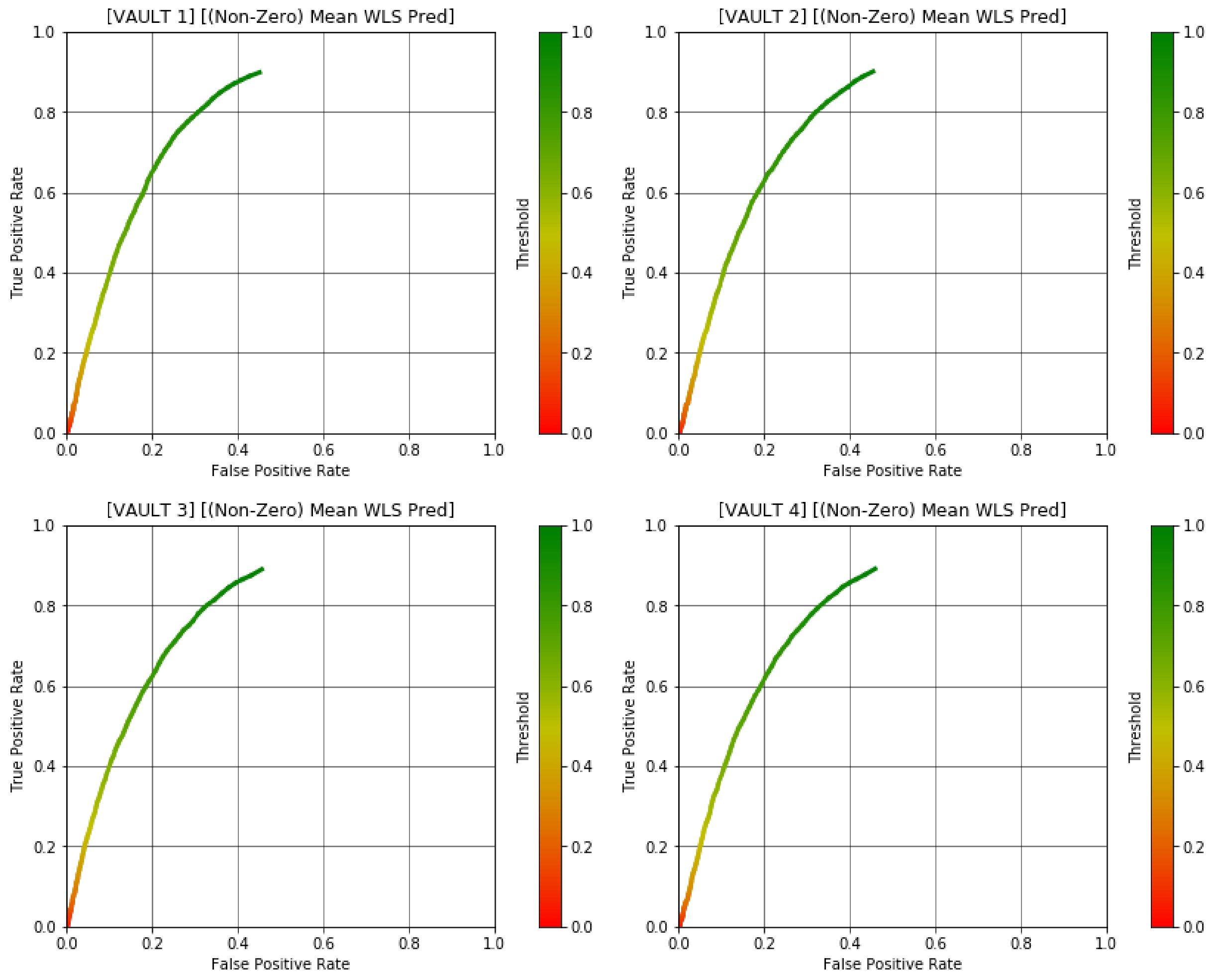


Figure 4.9: ROCs generated based on the average percent-fill prediction of all 15 of our WLS's on just non-zero data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.9 is the same as Figure 4.5, except it only uses "non-zero" data. Once again, we can see limiting the dataset in this way leads to significant improvements over predicting on the entire set. In all 4 validation sets, the WLS curves can be seen to perform marginally better than those from the KF up to a False Positive rate of 0.3. This is unexpected, especially considering the KF performed better on the full sets. However, the WLS graphs also never reach a 100% True Positive rate while the KF does. This is a trade-off between the methods when averaging. A more direct comparison between the two can be seen in Figure 4.11.

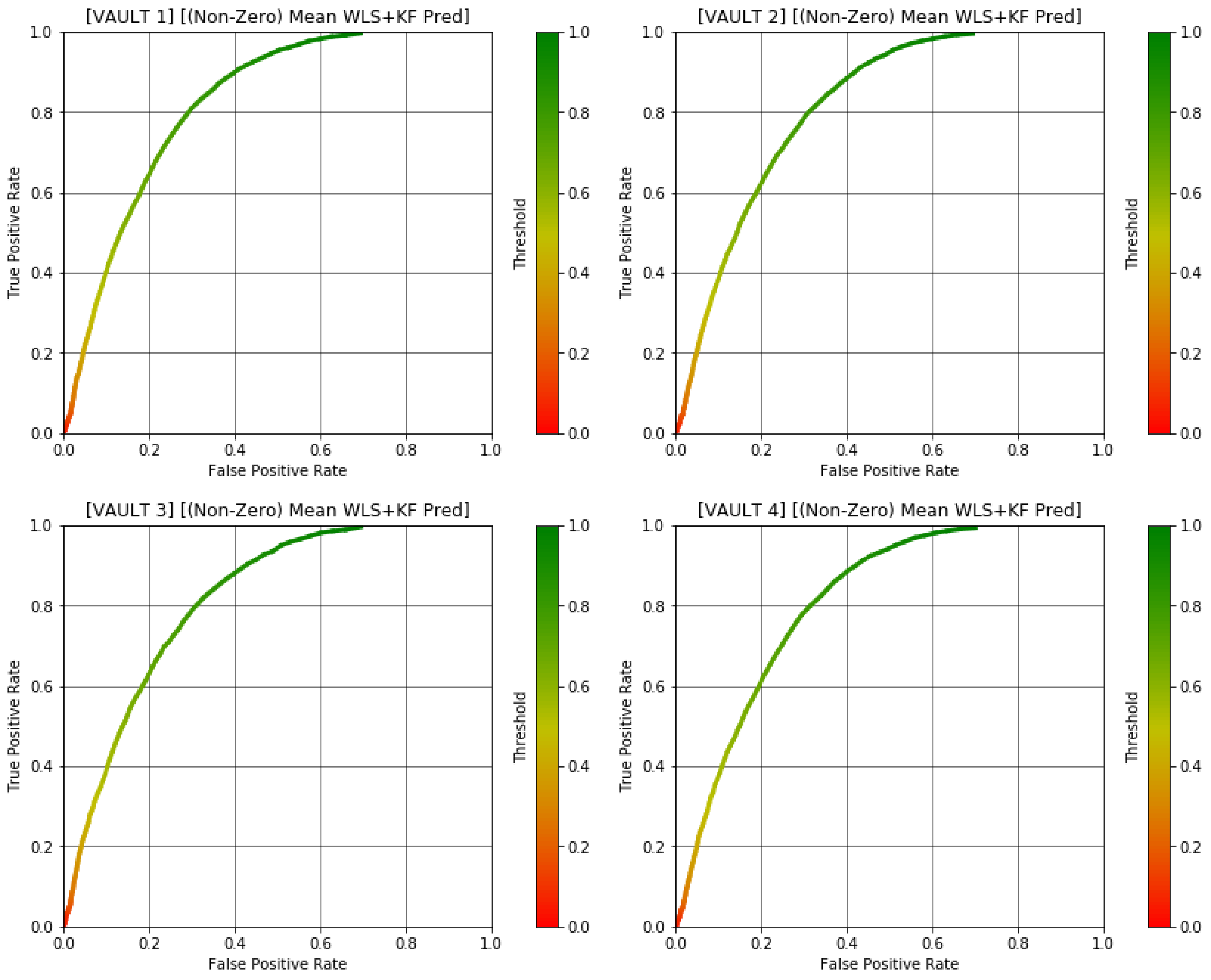


Figure 4.10: ROCs generated based on the average percent-fill prediction of all 15 of our KF's and all 15 of our WLS's combined on just non-zero data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.10 shows the ROC generated by taking the average of all 15 KF's and WLS together. It's interesting to note that averaging both together produces a curve that seems to perform better than either method individually as if they somehow compensate for one another. Figure 4.10 also appears in 4.11 along side both the KF and WLS methods independently as well as the baseline.

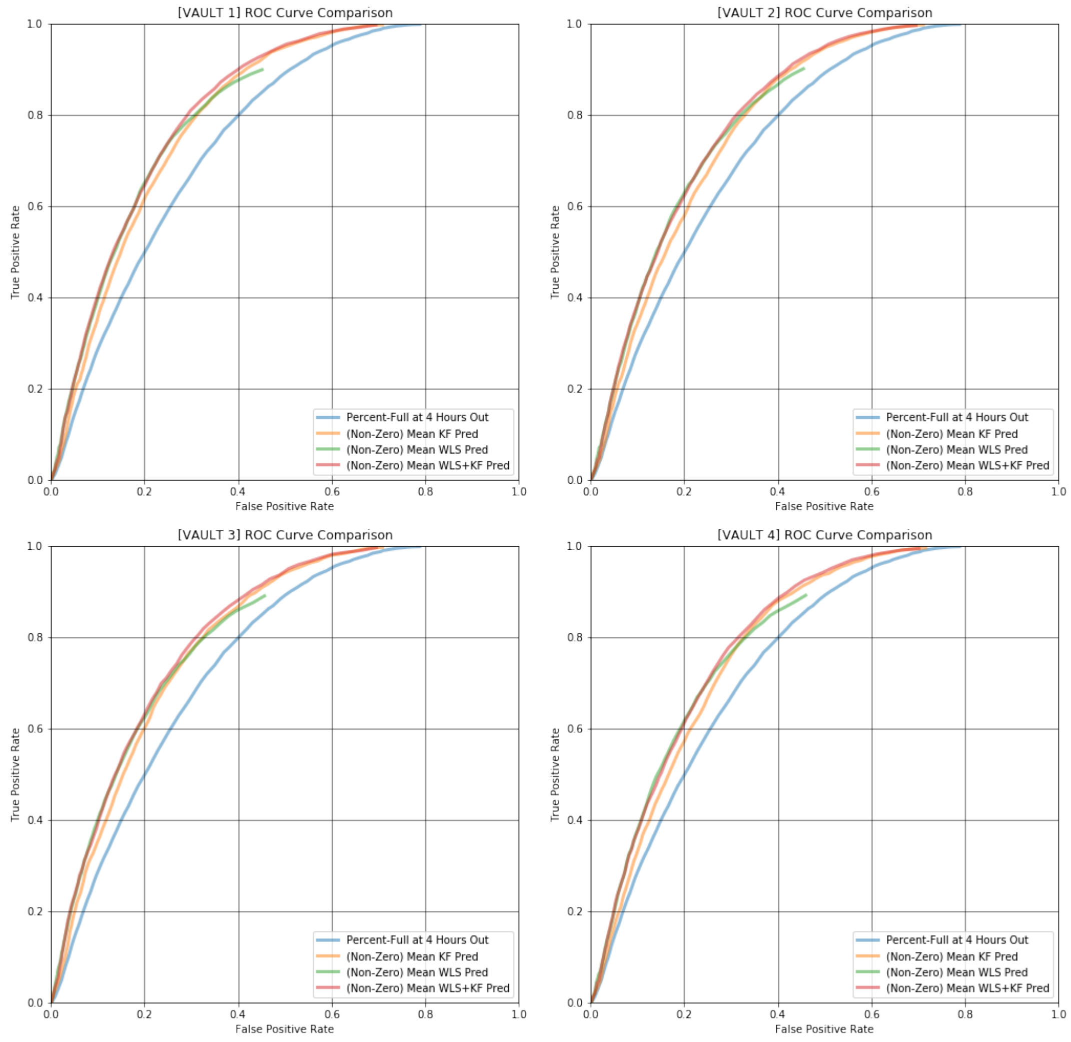


Figure 4.11: Comparison of ROCs generated based on the average percent-fill prediction of all 15 of our KF's and WLS's independently and combined using only non-zero data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

From Figure 4.11, we can see that averaging the predictions (in the case of non-zero data) already manages to outperform the baseline using time series data alone. However, considering the simplicity of the baseline that should not be considered a significant accomplishment. Seeing how well the ensemble of random forests did at predicting on just the Header data in Figure 4.2, we decided to investigate how well the ensemble could perform on time series predictions.

Figures 4.12 - 4.15 were generated based on the predictions from feeding the time series data into our KF and WLS with varying Q and λ parameters respectively. The final prediction was then found by feeding those predictions along with their respective α and β parameters into out ensemble of 20 random forests. The threshold being varied in these cases is the number of random forests required to flag a contest as a Positive (not filling) before classifying it as a Positive.

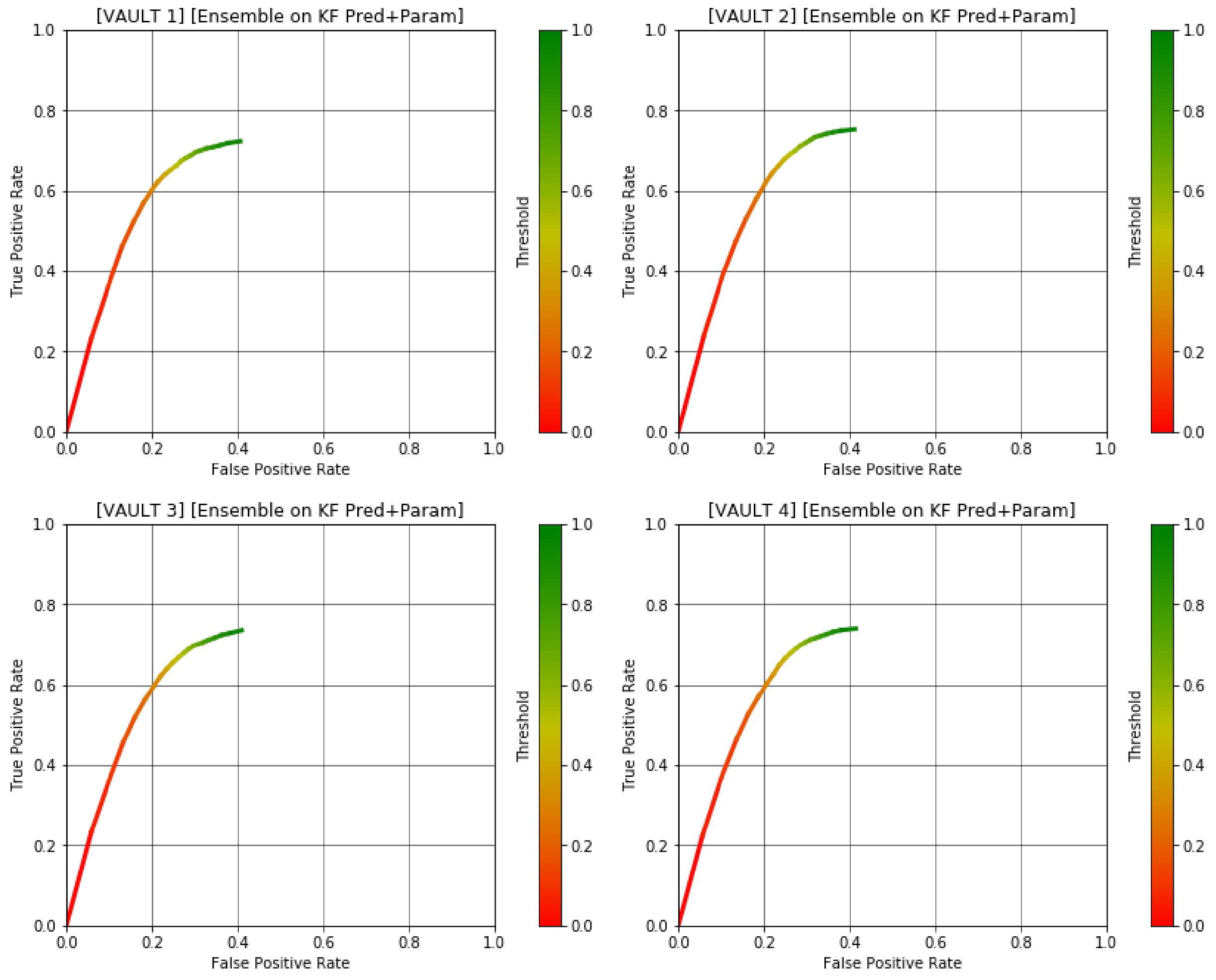


Figure 4.12: ROCs generated based on the ensemble prediction of the outputs of all 15 of our KF's (predicted final entries and model parameters) on the entirety of each validation set. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.12 was generated using the predicted final number of entries and α and β parameters from each of our 15 KF in our ensemble of random forests. Immediately, we can see the ensemble improves the predictive performance significantly compared to Figure 4.4, without even having to remove dataless contests. A very similar result can be seen in Figure 4.13 which shows the ROCs from using only the outputs from our WLS in the ensemble. While this is encouraging, we still wanted to look at how the ensemble performs on the "non-zero" data.

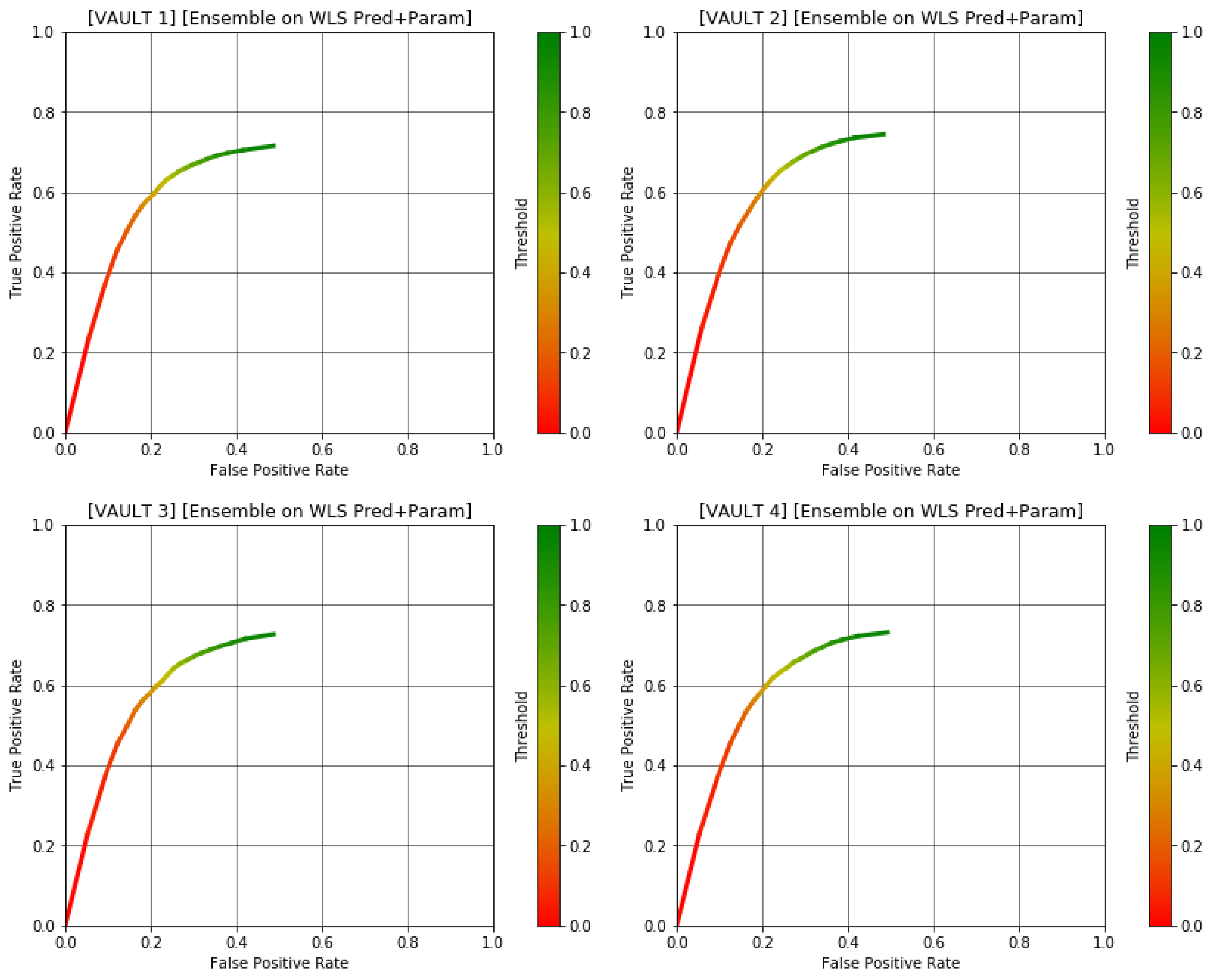


Figure 4.13: ROCs generated based on the ensemble prediction of the outputs of all 15 of our WLS's (predicted final entries and model parameters) on the entirety of each validation set. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

We also repeated the same ensemble predictions as in Figures 4.12 and 4.13 on only non-zero data. The results can be seen in Figure 4.14 along side the WLS and KF methods individually and the baseline for reference.

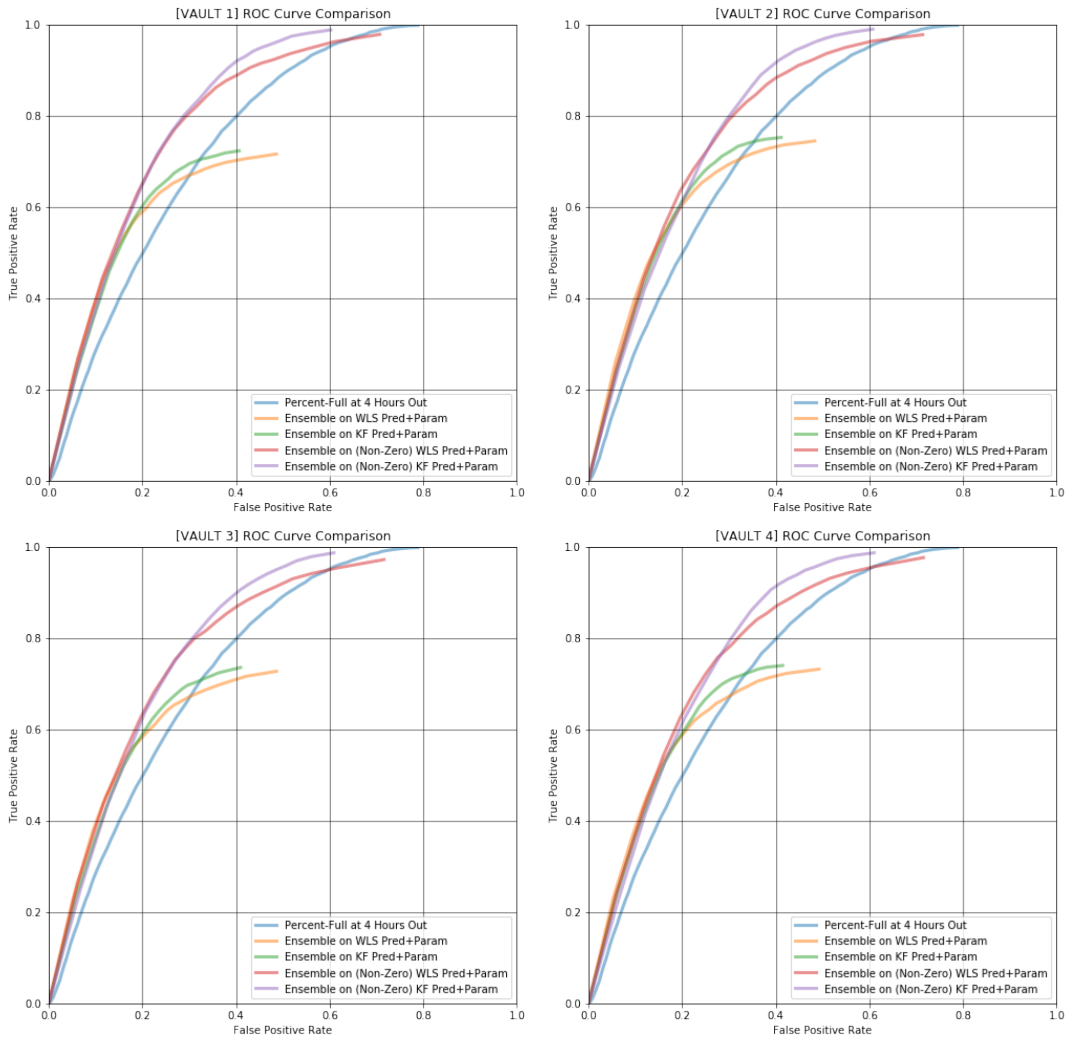


Figure 4.14: Comparison of ROCs generated based on the ensemble prediction of the outputs of all 15 of our WLS's and KF's (predicted final entries and model parameters) separately. The curves from using the entirety of each validation set and just the "non-zero" data both appear. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

From Figure 4.14 we can see that using non-zero data in the ensemble produces a negligible improvement in the predictive performance of both KF and WLS up until a False Positive rate of 0.2. From there, the performance disparity grows rapidly. There also appears to be no significant difference in performance between WLS and KF, however, for small False Positive rates WLS does mildly better, while KF does better for False Positive rates above 0.2 on the full set and above 0.3 on the non-zero set. In general, though, it seems all time series ensemble methods tend to outperform the baseline.

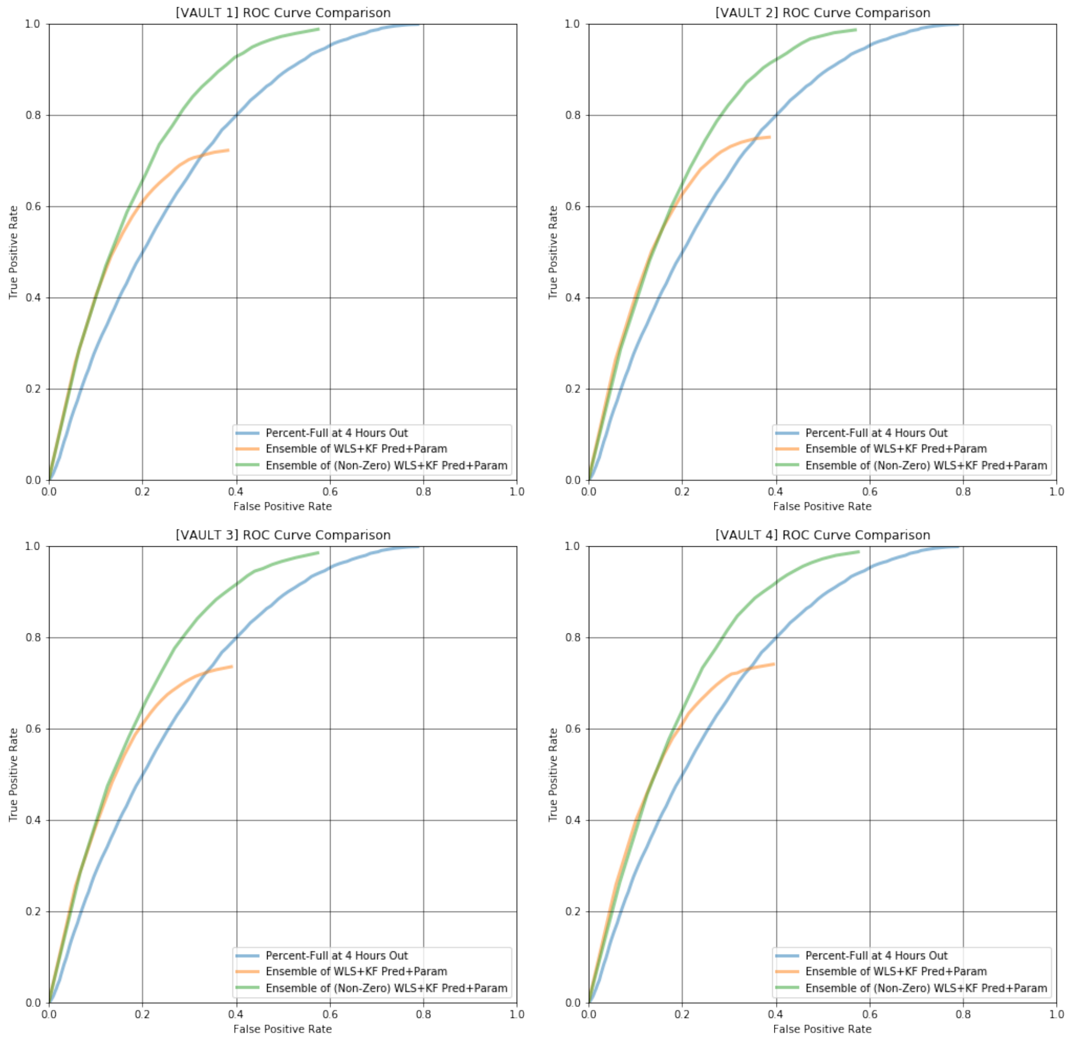


Figure 4.15: Comparison of ROCs generated based on the ensemble prediction of the outputs of all 15 of our WLS's and KF's (predicted final entries and model parameters) together on the entirety of each validation set and on just "non-zero" data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Random forests are known to be good at ignoring less useful data, meaning when provided more information they tend to perform strictly better or the same. With that in mind, we decided to see how well the ensemble could predict when given both the KF and WLS data. Figure 4.15 shows a comparison of the outcomes. In general, the ensemble of both outperforms either individual ensemble, though it seems only marginally. Once again, the performance is far better when using only non-zero data and both outperform the baseline.

4.3 Combining Time Series and Header Data

Now that we have fairly good results on Header and Time Series data separately, we wanted to try and see if we could achieve a better performance by using both at once. Given what we've already discussed about random forests being better on larger datasets, we decided to try providing both sets of data to our ensemble method.

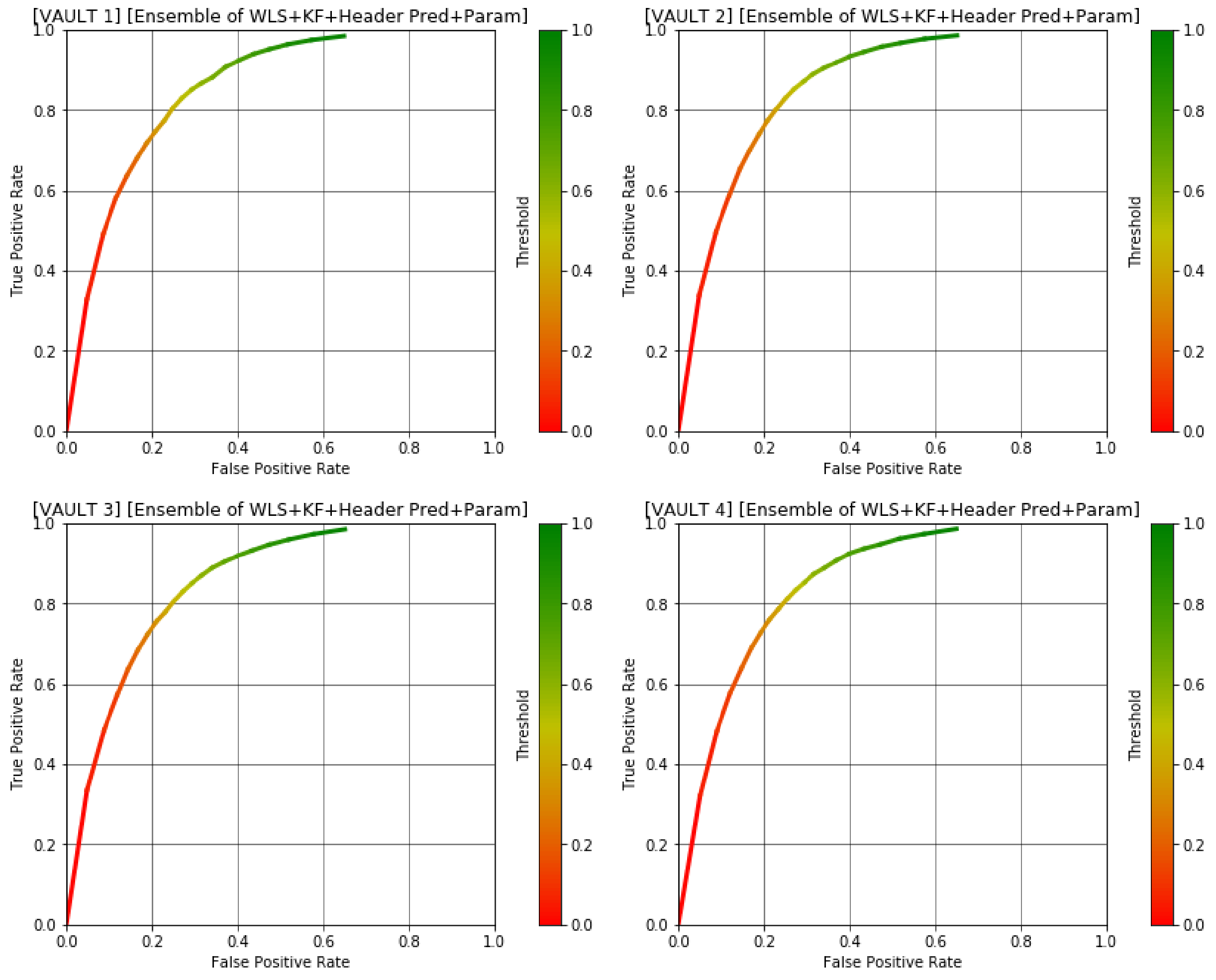


Figure 4.16: ROCs generated based on our ensemble predictions using Header data and the outputs of all 15 of our WLS's and KF's together (predicted final entries and model parameters) on the entirety of each validation set. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.16 shows the ROCs generated from the predictions of our ensemble when provided with all KF, WLS, and Header information. Comparing these curves with the best when using just Header data (Figure 4.2) and when using just KF and WLS data (Figure 4.15), we can see that using all available data improves the overall performance. It should be noted, these graphs were made using each full validation set. To see how using only non-zero data performs, we can turn to Figure 4.17.

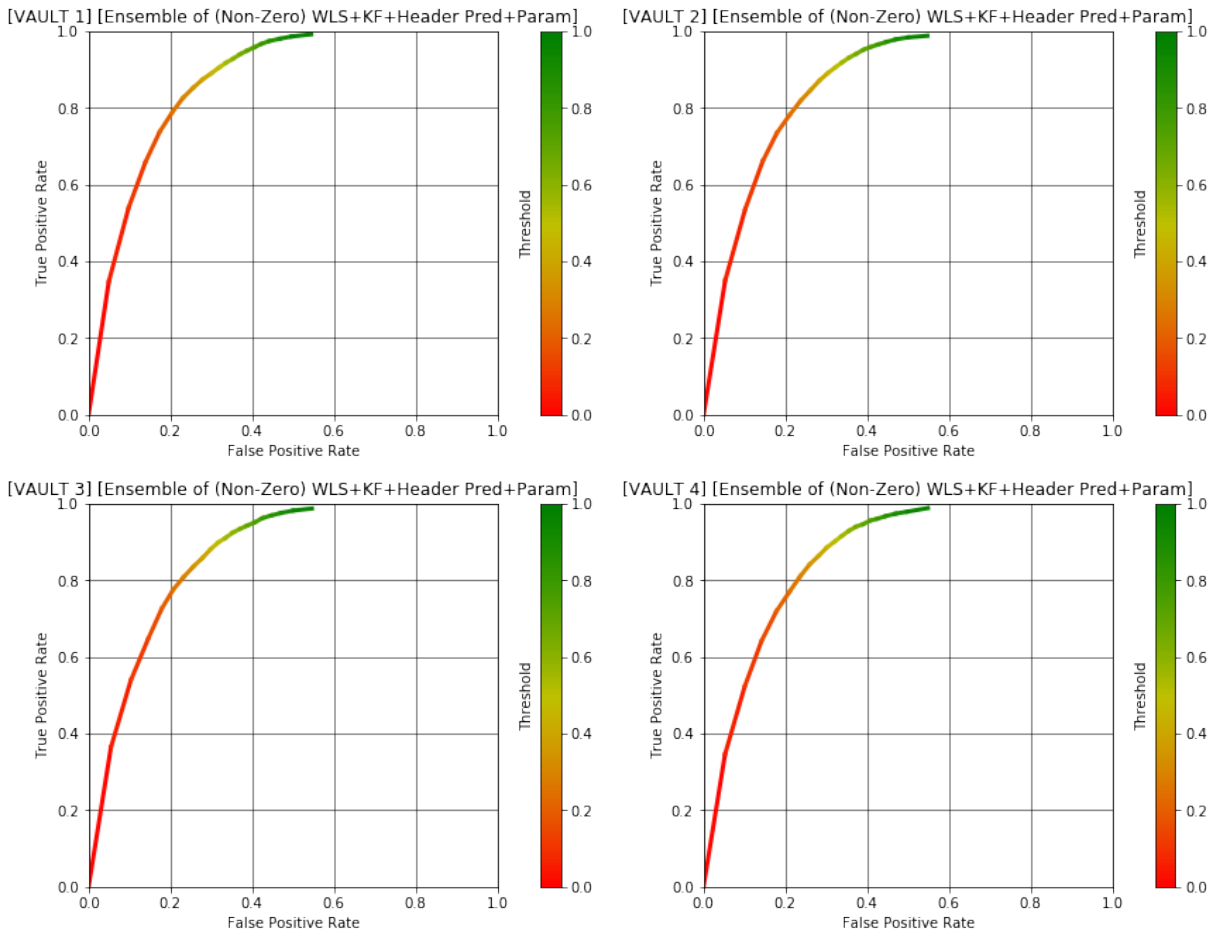


Figure 4.17: ROCs generated based on our ensemble predictions using Header data and the outputs of all 15 of our WLS's and KF's together (predicted final entries and model parameters) on just non-zero data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Once again, Figure 4.17 was made using only non-zero data and appears to outperform every other method used to this point. A direct comparison of the full ensemble on all data and on just non-zero data can be found in Figure 4.18.

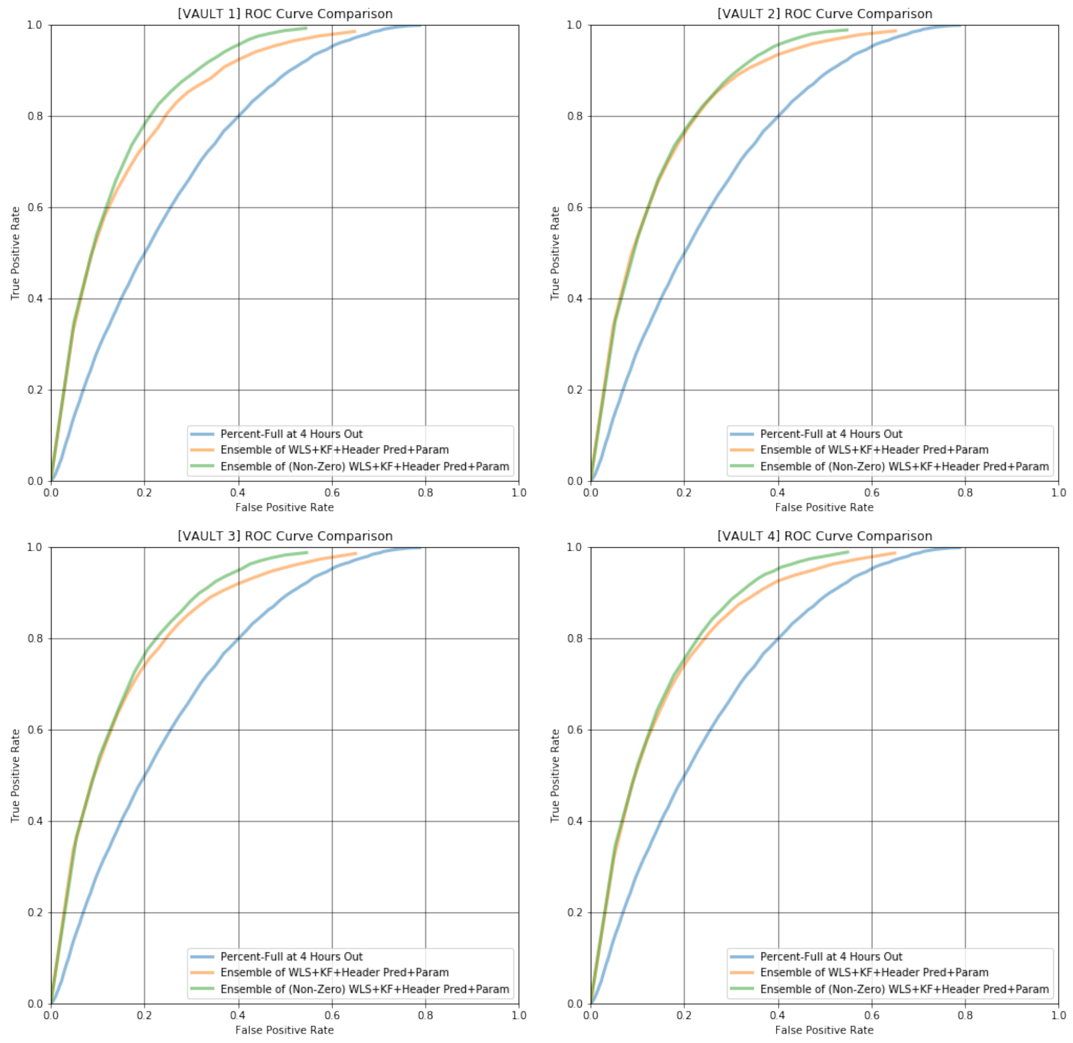


Figure 4.18: Comparison of ROCs generated based on our ensemble predictions using Header data and the outputs of all 15 of our WLS's and KF's together (predicted final entries and model parameters) on the full validation sets and on just non-zero data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Here we can see that both ensemble methods significantly improve over the baseline. Moreover, using only non-zero data when predicting on an ensemble of all available data appears to perform best overall, though not noticeably until a Flase Positive rate of around 0.2 and even then not significantly.

4.4 Pacer Data Results

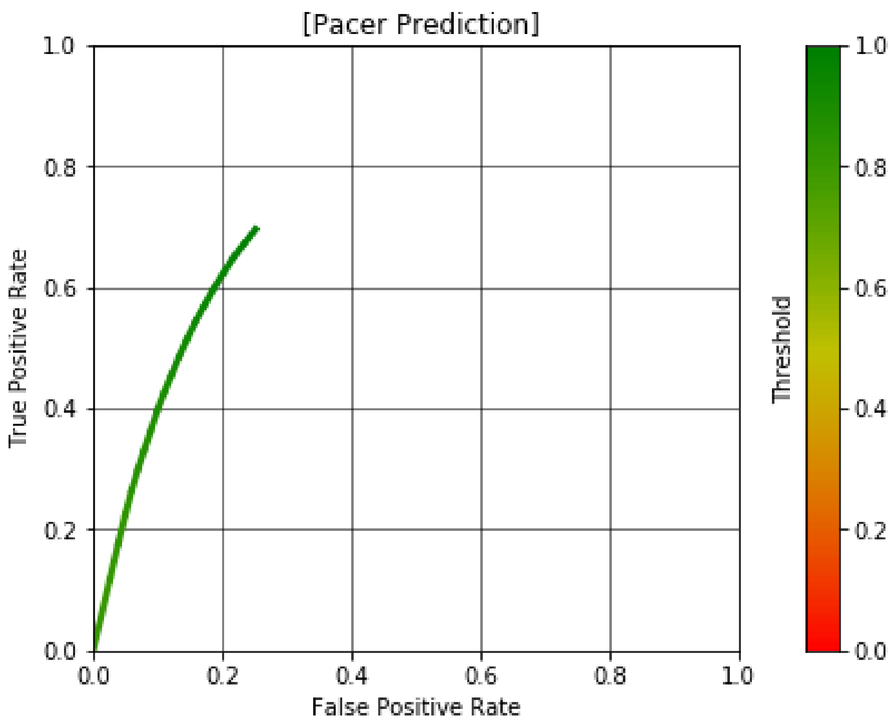


Figure 4.19: ROC generating using the predictions from the predictive methods currently implemented at DraftKings, known as "Pacer Data".

Figure 4.19 shows a ROC curve made using a set of provided "pacer data" which contained the predictions from the predictive methods currently implemented at DraftKings. As we can see, this curve is already better than the baseline achieving higher True Positive rates at equivalent False Positive rates. For the rest of this section, we investigate what adding this pacer data to our ensemble does to its predictive performance. For the purposes of this section, we show the results from only 1 of the 4 validation sets, however, all 4 validation curves are included in the subsequent section. It should be noted that DraftKings has only recently started recording this data, so we have far fewer contests for which a pacer prediction can be made. This significantly lowers the size and variability of our training set which can lead to decreased performance by our ensemble method.

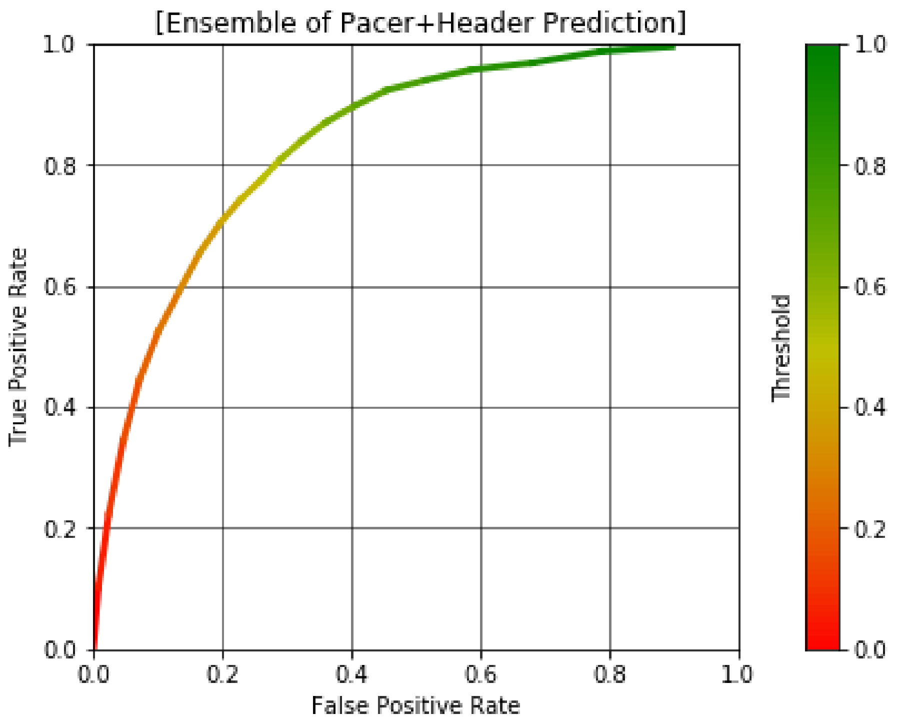


Figure 4.20: ROC curve generate using the processed Header data and available Pacer data.

Figure 4.20 shows the ROC generated using our Header data and DraftKing's pacer data in our ensemble method. Comparatively, Figure 4.21 shows the ROC generated using our Header data, DraftKing's pacer data, and our KF and WLS data in our ensemble method. Both approaches seem to produce nearly equivalent curves, with the latter performing better for False Positive rates ≥ 0.3.

Figure 4.22 shows a comparison between both the aforementioned methods and an ensemble of just Header, WLS, and KF data relative to our baseline. Here, we can see that for most False Positive rates one of the Pacer methods performs best or tied for best. Only in the False Positive range from 0.1 - 0.3 do the pacer methods seem to perform worse than the ensemble using just Header, WLS, and KF data. This is impressive given that the Pacer dataset is so much smaller than the total dataset. In fact, if we had the relavent pacer data for all contests, we expect that the pacer methods would be equal to or better than the pacerless ensemble method.

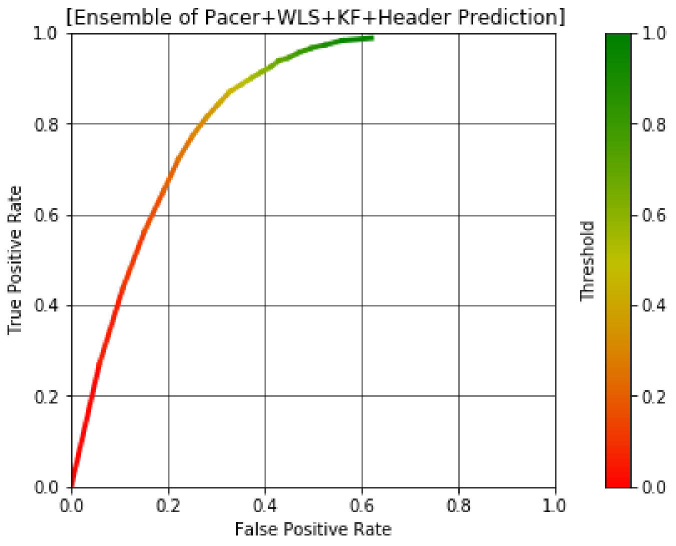


Figure 4.21: ROC generated using an ensemble of Header, Pacer, and WLS and KF output data.

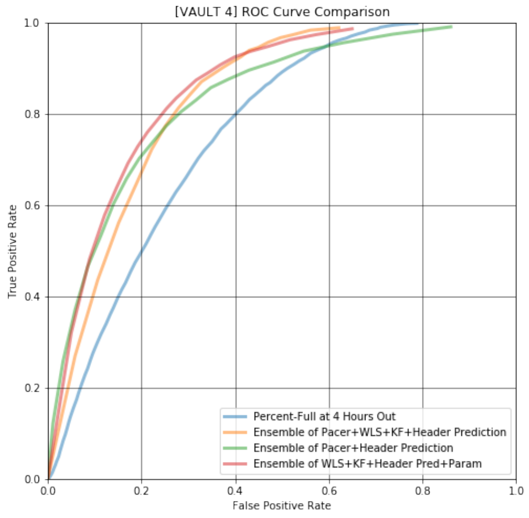


Figure 4.22: Comparison of the methods involving Pacer data.

4.5 Situational Predictions

Up until now, we've shown that time series predictions perform strictly better when using only non-zero data. However, when making our final prediction we can not simply ignore all contests that lack time series data before "4 Hours Out". To address this, we decided to use two distinct random forest ensembles: one including all Header, WLS and KF data and the other including only Header data. The first ensemble would then be used to predict on contests that have available time data and the second for those that don't. Pacer data has been omitted for this situational approach as there was not enough Pacer data to cover the entire dataset. Figure 4.23 shows the ROCs generated by following this situational predictive scheme.

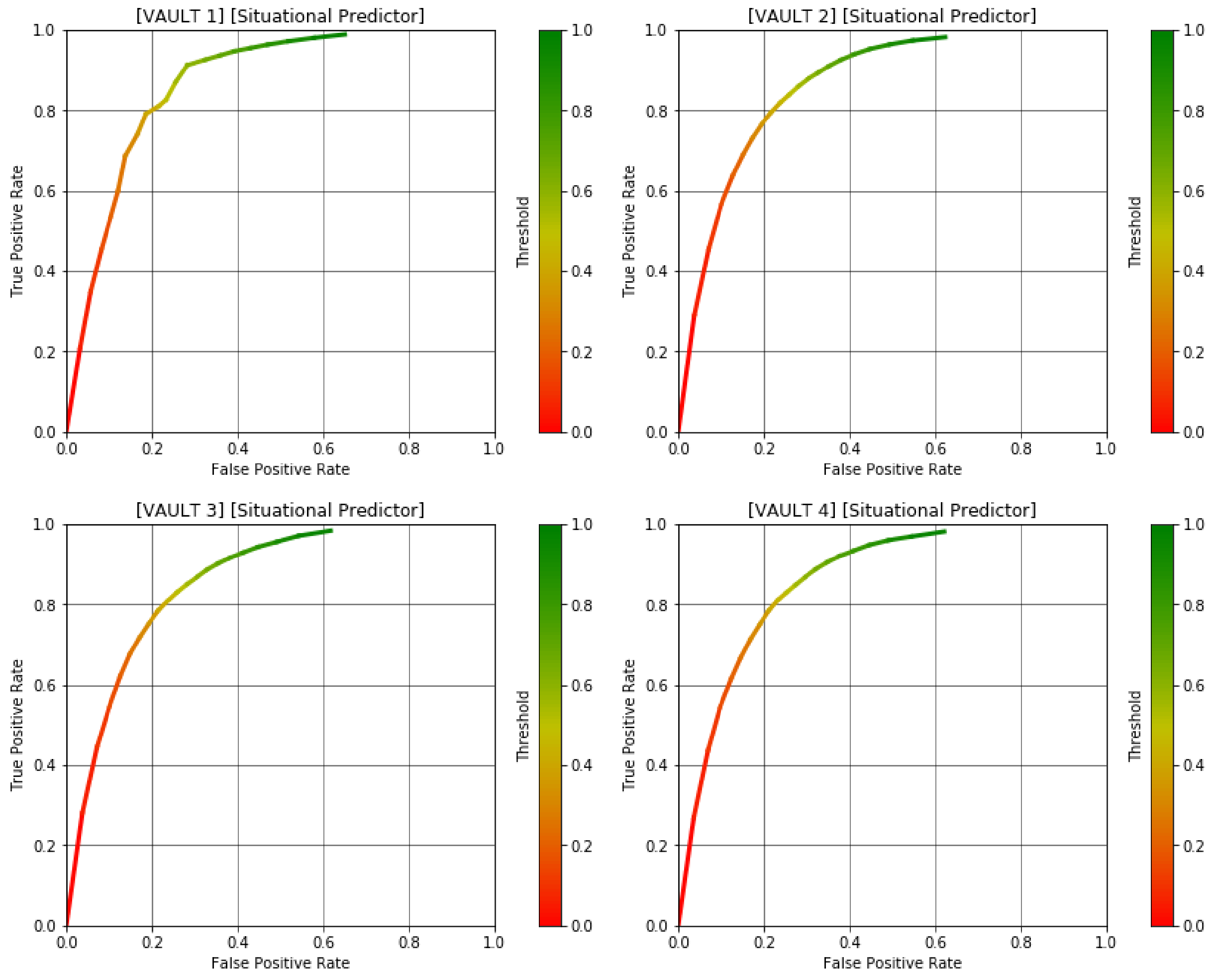


Figure 4.23: ROCs generated based on the situational ensemble prediction where the outputs of all 15 of our WLS's and KF's and Header data are used to predict non-zero contests and just Header data is used to predict contests with no available data before 4 Hours Out. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

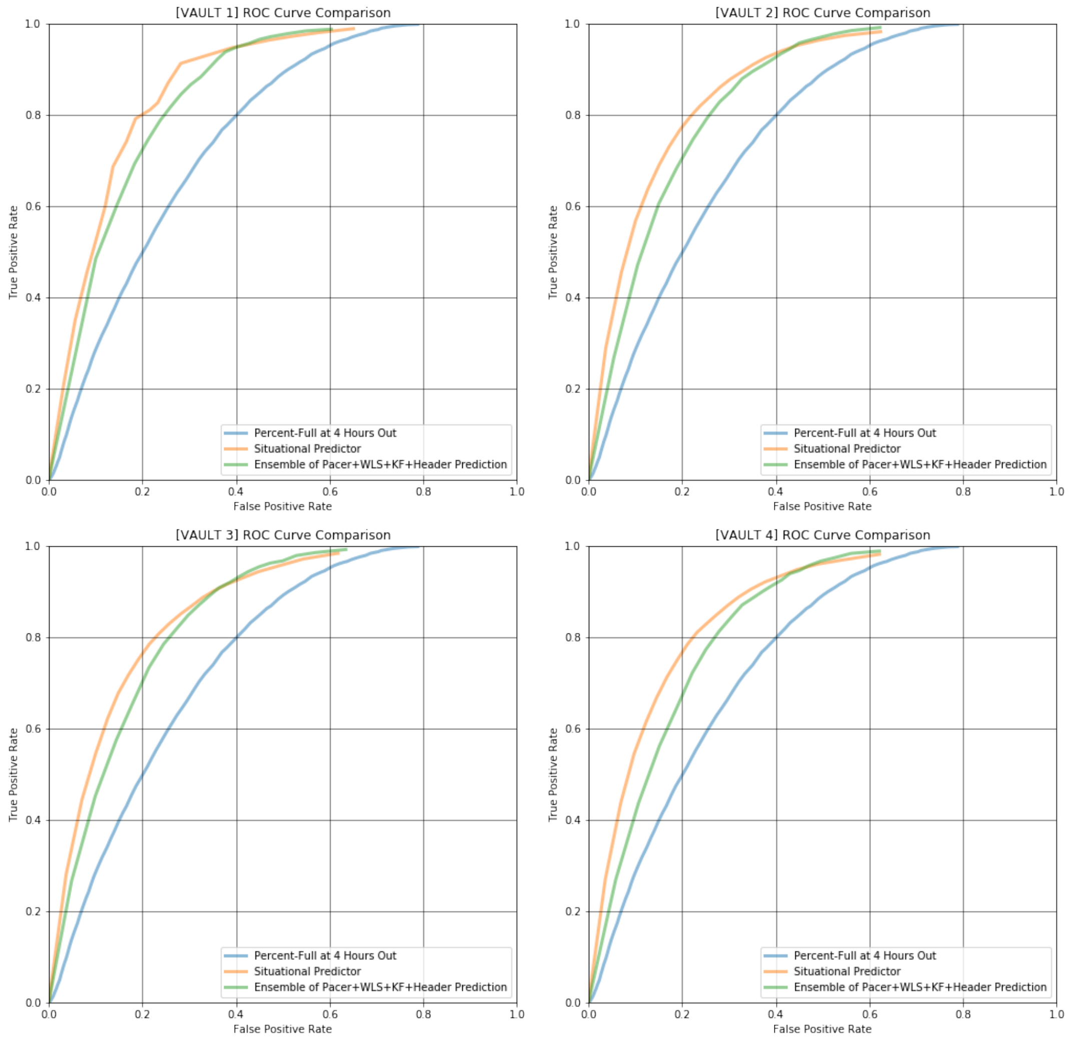


Figure 4.24: ROCs generated based on the ensemble prediction of the outputs of all 15 of our WLS's and KF's (final entries prediction and model parameters) together on the entirety of each validation set and on just "non-zero" data. Top Left: Validation set 1. Top Right: Validation set 2. Bottom Left: Validation set 3. Bottom Right: Validation set 4.

Figure 4.24 shows a side by side comparison between using the situational predictor and using just the full ensemble. In nearly all cases, we can see the situational predictor tends to outperform the full ensemble alone. Overall, we believe the situational predictor to be the best classification method achieved by this project.

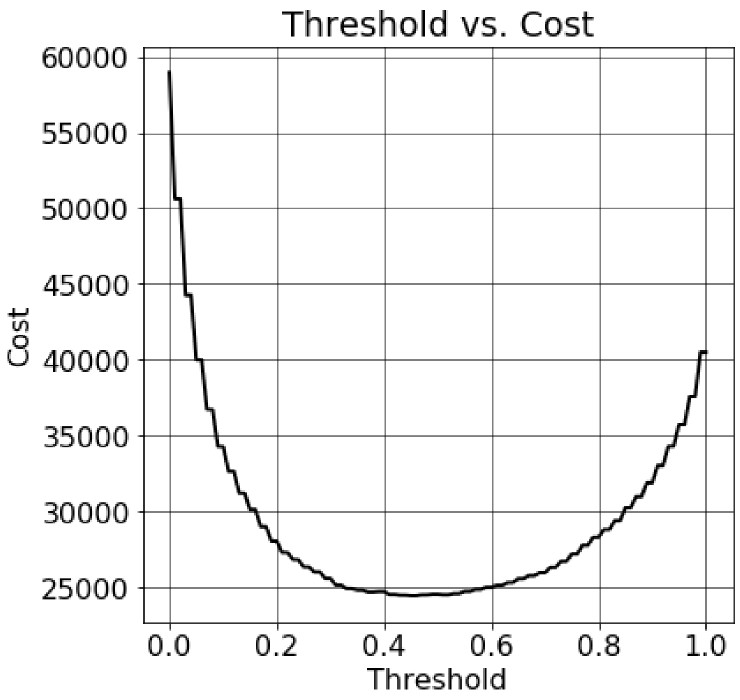


Figure 4.25: A plot of threshold used for our situational ensemble classifier vs the cost of misclassified contest calculated by assuming false negatives cost 10 times as much as false positives. This graph achieves a minimum value at a threshold of 0.46.

4.6 Considering Costs

Now that we've determined what constitutes our best classifier, we can analyze the financial savings possible because of it. In this application, correctly identifying failing contests directly impacts the amount of possible revenue DraftKings will receive. If a contest is labeled as Succeeding, but actually Fails (a false negative) then DraftKings loses the potential revenue from every entry not filled. Conversely, if a contest is labeled as Failing, but actually Succeeds (a false positive) then DraftKings could needlessly spend money advertising a contest (which could also take up advertising space from a real failing contest that needs it). For the purposes of addressing the cost of a mislabeled contest DraftKings considers a false negative 10 times worse than a false negative. Using this cost approximation, we calculated the cost at each threshold for our situational ensemble classifier. Figure 4.25 shows a plot of the assumed total cost using this 10:1 weighting scheme at each threshold for our ensemble. Looking at this plot, we can see that the cost is minimized at a threshold of 0.46, meaning 46% of the forests in our ensemble need to classify the contest as a Failure to predict it as a Failure.

We also analyzed the actual lost revenue correctly identified by our classifier. Figure 4.26 shows a plot of the total percent of contests predicted as Failing vs the percent of lost revenue correctly identified. From it, we can see that when our classifier identifies 20% of contests as Failing, 95% of lost revenue are correctly identified; meaning up to 95% of the lost revenue

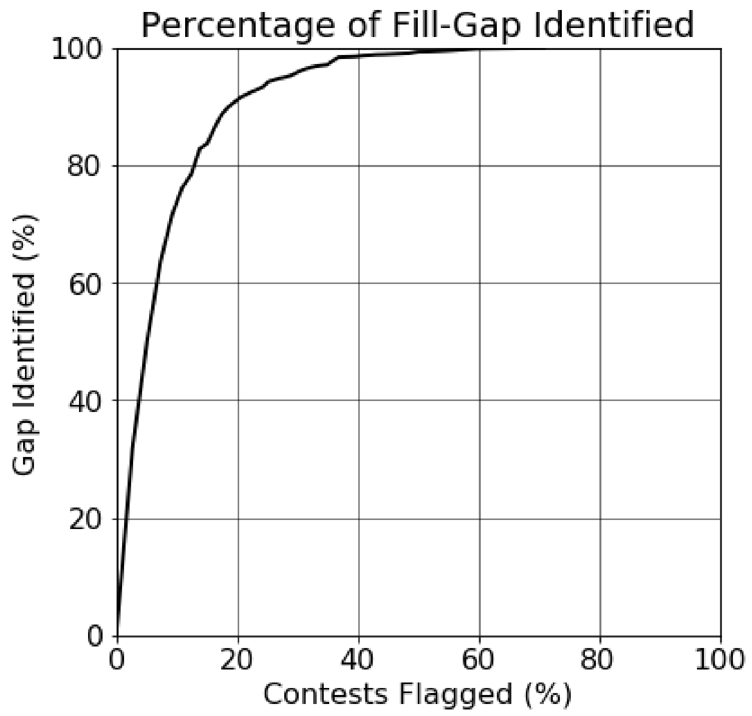


Figure 4.26: A plot of the percent of contest flagged as failing vs the percent of lost revenue correctly flagged by our ensemble classifier. Approximately 95% of lost revenue appears to be correctly identified when only 20% of contests are flagged as failing.

could have been recuperated using our ensemble. In total, that could have meant $66,000,000 in extra profit over the last four years from just the validation set. Now, it is unrealistic to believe that DraftKings would have the ability to advertise every contest predicted to fail, however, the vast majority of contests represent very small potentials for profits, so by focusing on the largest contests DraftKings could still expect to recuperate a large sum of lost revenue.

Conclusion

5.1 Takeaways

In this application, we were able to make non-trivial improvements in predicting whether a given sporting contest would fill to its maximum capacity. Further more, using a highly flexible machine learning technique like Random Forests proved invaluable for turning our set of highly processed data into good predictions. In general, we've found, with minor exceptions, that providing more information to our ensemble method improved performance in most cases. In particular, this was true when we provided more information while maintaining the size of the dataset. But overall, this project has shown that binary classification, while deceptively simple, can still present a challenging problem both in academic theory and in practical industry implementations. In our case, we had to utilize a combination of powerful machine learning and mathematical modeling algorithms in order to make significant headway in improving the prediction. And though our final situational ensemble method proved impactful in this application, it still leaves ample room for improvement, both in terms of its raw predictive accuracy and in its lack of provided information. To that end, the next section outlines ideas we had that we either didn't have time to implement or that were out of scope for this project, but would be interesting for future groups to work on.

5.2 Future Work

Throughout this project, we had to constantly be cognizant of the fact that our dataset was inherently imbalanced with a Success to Failure ratio of 9:1. We only ever used one method to ensure our training set was evenly distributed - undersampling. We think our results could be immediately improved given a different sampling method such as an oversampling scheme or some hybrid implementation of both. Specifically, we were interested in trying to grow our dataset by creating new artificial points based on current point. One common way of doing this is the synthetic minority over-sampling technique or SMOTE for short. However, because SMOTE merely creates new points along linear combinations of existing one, we were drawn more towards generative adversarial networks or GANs for short. GANs uses a deep neural network architecture to create new points that are theoretically indistinguishable from the originals. We believe using GANs to expand the set of Failing contest data would allow us to have a larger, balanced training set for the ensemble which should theoretically improve performance.

We also think the current ensemble output is too simple. That is, it outputs a set of predictions and nothing else. DraftKings surely cannot take action on every contest predicted to fail as there is only so much available space to advertise, so it would be helpful if the predictions came with some sort of confidence rating if even possible. This would allow DraftKings to focus their efforts on bolstering contests that are more certain to fail. Moreover, our ensemble is strictly limited to binary classification. This means a contest that is predicted to fail by 1 entry and a contest that is predicted to fail by 1000 entries invoke the same response with no way to tell which is which. It would be more useful to have a regression estimate so that DraftKings can get a sense for how significantly a contest will fail or succeed by along with an associated 95% confidence interval on each prediction.

We also would have liked to see how using a more complicated fitting function could affect the results. Right now, our Kalman Filter and Weighted Least Squares methods only predict the parameters and final number of entries using a simple exponential model. We've observed that most contests exhibit a pseudo sinusoidal behavior, experiencing periods of quasi exponential growth followed by lulls of inactivity. It is believed this is because the majority of users are in the US or Canada, and so they are most active during the day and inactive at night. We think it could be fruitful to derive a modeling function that also exhibits this kind of behavior. Alternatively, if an algorithm can be made to perform an exponential fit only on the data after the last lull, that may also perform quite well.

Lastly, we were interested in testing minor tweaks to the ensemble as a whole. We would be interested to see if the quality of prediction can be maintained beyond the 4 Hour mark. In general, we'd want to see how early we can reliably predict failing contests. We'd also have liked to see how tweaking the random forest parameters affects the predictions. As of now, all forest parameters were the defaults for scikit-learn. In particular, it could be impactful to vary the number of trees per forest and/or the number of variables tried at each branching split. We'd also be interested to see how changing the number of random forests used in the overall ensemble affects the predictions. To conclude, there are many possible directions that future teams could take to continue our work in improving DraftKings' predictive algorithm.

Bibliography

[1] A. Assa and F. Janabi-Sharifi. A kalman filter-based framework for enhanced sensor fusion. *IEEE Sensors Journal*, 15:3281–3292, 2015.

[2] G. Bastista, R. Prati, and M. Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM SIGKDD Explorations Newsletter*, 6:20–29, 2004.

[3] V. Dhar. Data science and prediction. `https://dl.acm.org/citation.cfm?id=2500499`, 2013.

[4] N. Donges. The random forest algorithm. `https://towardsdatascience.com/the-random-forest-algorithm-d457d499ffcd`, 2018.

[5] DraftKings. Our story. `https://about.draftkings.com/?_ga=2.262532108.126649850.1554691891-710459506.1552529947`, 2019. Accessed: 2019-03-21.

[6] T. Fawcett. An introduction to roc analysis. *Pattern Recognition Letters*, 27:861–874, 2006.

[7] M. Galar, A. Fernanadez, E. Barrenechea, H. Bustince, and F. Herrera. A review on ensembles for the class imbalance problem: Bagging, boosting, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics*, 42:463–484, 2011.

[8] G. Haixiang, L. Yijing, J. Shang, G. Mingyun, H. Yuanyue, and G. Bing. Learning from class-imbalanced data: Review of methods and applications. *Expert Systems with Applications*, 73:220–239, 2017.

[9] E. Hall. The dark side of fantasy football. `https://www.washingtonpost.com/news/made-by-history/wp/2017/09/10/the-dark-side-of-fantasy-football/?noredirect=on&utm_term=.076ead7f75e6`, 2017.

[10] D. Heitner. The hyper growth of daily fantasy sports is going to change our culture and our laws. `https://www.forbes.com/sites/darrenheitner/2015/09/16/the-hyper-growth-of-daily-fantasy-sports-is-going-to-change-our-culture\-and-our-laws/#308b5cc5aca1`, 2015.

[11] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*. Springer, 2013.

[12] R. Kalman. A new approach to linear filtering and prediction problem. *Journal of Basic Engineering*, 82:35–45, 1960.

[13] L. Kleeman. Understanding and applying kalman filtering. `http://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/integrated3/kleeman_kalman_basics.pdf`.

[14] R. Kunert. Smote explained for noobs - synthetic minority over-sampling technique line by line. `http://rikunert.com/SMOTE_explained`, 2017. Accessed: 2018-11-15.

[15] T. Lacey. Kalman filter tutorial. `http://web.mit.edu/kirtley/kirtley/binlustuff/literature/control/Kalman%20filter.pdf`.

[16] B. Liu, Y. Ma, and C. Wong. Improving an assocaition rule based classifier. `https://link.springer.com/chapter/10.1007/3-540-45372-5_58`, 2002.

[17] Y. Long, X. Huo, H. Sun, Y. Liu, Q. Wang, Z. Li, and T. Xu. Research on kalman filter prediction method based on decision tree analysis. *International Conference on Information Science and Control Engineering (ICISCE)*, 4, 2017.

[18] N/A. Machine learning: What it is and why it matters. `https://www.sas.com/en_us/insights/analytics/machine-learning.html`, 2019. Accessed: 2019-4-23.

[19] J. Pan, X. Yang, H. Cai, and B. Mu. Image noise smoothing using a modified kalman filter. *Neurocomputing*, 15:1625–1629, 2016.

[20] A. Rodriguez. How the $7 billion us fantasy football industry makes its money in 2017. `https://qz.com/1068534/how-the-7-billion-us-fantasy-football-industry-makes-its-money-in-2017/`, 2017.

[21] M. Sanjeevi. Chapter 4: Decision trees algorithms. `https://medium.com/deep-math-machine-learning-ai/chapter-4-decision-trees-algorithms-b93975f7a1f1`, 2017.

[22] P. Tan, M. Steinbach, A. Karpatne, and V. Kumar. *Introduction to Data Mining*. Pearson, 2018.

[23] E. Weisstein. Least squares fitting. `http://mathworld.wolfram.com/LeastSquaresFitting.html`, 2019. Accessed: 2019-04-06.

[24] G. Welch. History: The use of the kalman filter for human motion tracking in virtual reality. *Presence*, 18:72–91, 2009.

[25] Q. Yang and X. Wu. 10 challenging problems in data mining research. *International Journal of Information Technology Decision Making*, 5:597–604, 2006.