Prediction Analysis of Airline Delays

Gupta, Abhi
abhgupta@ucdavis.edu

Maysenhalder, Sofia
smaysenhalder@ucdavis.edu

Mittal, Aditya
adimittal@ucdavis.edu

Prado, Alex
alxprado@ucdavis.edu

Vu, Vanessa
vtvvu@ucdavis.edu

Abstract

Airline delays and cancellations cause significant inconveniences for passengers and losses of revenue for airlines. This study aims to explore the factors leading to airline delays and cancellations using several predictive models to forecast future delays and cancellations. The study investigates the relationship between airline delays/cancellations and factors such as day of the week, flight length, and departure time. It also evaluates the performance of algorithms including logistic regression, and Naive Bayes for predicting delays and cancellations. The analysis involves creating interactive visualizations, assessing the correlation between predictors and the dependent variable, and testing the accuracy and computation times of the predictive models. The challenges of predicting airline delays and assessing model accuracy and computational speed are discussed, along with the limitations of certain algorithms. The study concludes that Logistic Regression and Naive Bayes are the most efficient algorithms in terms of accuracy and computational efficiency from this study.

1 Introduction

Air travel is extremely common for passengers worldwide. According to the Federal Aviation Administration, the Air Traffic Organization (ATO) provides services to more than 45,000 flights with approximately 2.9 million passengers every day. Airlines are responsible for the safe and timely transportation of passengers to their destination, however, airline delays are inevitable. Unforeseen weather conditions, mechanical malfunctions, labor shortages, and late aircraft arrivals are some of the factors that lead to airline delays and cancellations. Airlines have continued to experience high volumes of delayed flights post-pandemic because of the lack of staff that had decreased during the height of the pandemic in response to low flying demand. [3] The holiday season inherently brings about large volumes of flyers, however, the massive number of flight delays last December 2022 was due to unexpected winter storms that impacted states across the country. On top of this natural disaster, Southwest canceled a majority of its flights during this time due to a failure in their internal communication system. [3] Consequently, the company lost many customers because passengers were forced to make last minute changes. This recent history of airline delays shows how airlines can be impacted by a multitude of factors beyond their control. Airline delays is an ongoing, post-pandemic problem that has been rampant in recent months, negatively affecting both flyers and airlines across the U.S.

With this background in mind, our project focuses on predicting whether a flight will be delayed by employing several different machine learning algorithms and testing their prediction effectiveness and computation time. Our Kaggle dataset, called âAirlines Delayâ, provides 539,382 entries of flight information data for the following eight variables: Flight (Flight ID), Time (Time of departure), Length (length of flight), Airline (ID), AirportFrom (which airport the flight flew from), airport to (which airport the flight flew to), Day of the week (day of the week of the flight), and Class (Delayed [1] or not [0]). Our dependent variable is Class, which tells us whether a flight is delayed or on time. To predict flight delays, we used the following techniques: Logistic Regression, K-Nearest Neighbors, Naive Bayes, and a Multilayer Perceptron. Furthermore, we employed optimization techniques such as gradient descent and gradient ascent to logistic regression and parallel computing to Naive Bayes in order expedite the computation efficiency. We chose our best model based on accuracy rate and computation time.

For our exploratory data analysis, we investigated the breakdown of the number of delayed versus on-time flights by airline using bar charts. We found that Southwest Airlines (WN) contributed to both the highest number of flight delays, as well as had about 70% of all their flights delayed, which is exceptionally high. Continent Airline (CO) also had more than half of their flights delayed (55%). When constructing a normalized histogram of the length of delayed flights compared to on-time flights, we found that shorter flights were more likely to be delayed. We also found that flights in the middle of the week were more likely to be delayed when constructing a normalized histogram of the days of the week. Finally, we wanted to see which days and times experienced the most delays by creating a heatmap of the time of the flights against the day of the week. To do this, we first created a new variable, 'Hours', that translated the time of flight variable into a 24-hour scale representing the hour of the flight. For example, any flight between 12am to 12:59am would represent the 0th hour and any flight between 11pm and 11:59pm would represent the 23rd hour. We found that most delays occurred on Wednesday and Thursday between the hours of 4pm and 7pm. More specifically, Thursday at 5pm experienced the most airline delays, followed by Wednesday at 5pm, and Thursday at 9pm.

2 Proposed Methodologies

In this project, we implemented four different machine learning algorithms to predict whether a flight would be delayed or on-time. In particular, we focus on the following machine learning techniques: Logistic Regression, Naive Bayes, K-Nearest Neighbor, Multi-Layer Perceptron. Our objective is to compute the prediction accuracy to determine how well the model could predict whether a flight is going to be delayed or not. We are also interested in comparing the computation efficiency of each algorithm based on the run-time of each method. For each algorithm, we split our dataset into a test/train set using 80:20 ratio â with 80% of the data being used for training our model and the other 20% used for evaluation of prediction accuracies. We ran each method 10 times and computed the average accuracy rate that we then compared between each method. Ultimately, we used factors of both accuracy and computation time to pick the best model.

2.1 Logistic Regression with Gradient Ascent and Gradient Descent

The logistic regression model employs a more complex cost function â the Sigmoid function. The algorithm works by taking a weighted sum of all the predictors and calculating the probability of the observation pertaining to the appropriate classification. Additionally, in the sigmoid function, the parameter theta represents the weight values, which undergoes optimization in order to calculate the prediction values. [10] In this project, the sigmoid function returns a numerical number between 0 and 1, which represents the probability of a flight being delayed. Therefore, a predicted value close to 1 â greater than 0.5 â indicates that the plane was delayed, and a predicted value close to 0 â less than 0.5 â indicates that the plane was on time for that observation. Furthermore, we implemented a cost function, using a logarithmic function, in order to illustrate how well our model is doing at predicting the outcomes, which is done so through measuring the difference between the predicted outcome and the actual outcome [11]. The cost function improves the model by helping us adjust the weight values in order to minimize the error.

We have the following relationship in a logistic regression:

$$\log\left(\frac{p(\mathbf{X})}{1-p(\mathbf{X})}\right) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p \tag{2.1}$$

Then, we can express $p(\mathbf{X})$ as:

$$p(\mathbf{X}) = \frac{e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p}} \tag{2.2}$$

We use the method of maximum likelihood to estimate the parameters $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$, $L(\beta_0, \beta_1, \beta_2, \ldots, \beta_p) = \Pi_{i:y_i=1} p(x_i) \Pi_{i:y_i=1} p(1-(x_i'))$

In logistic regression, deviance residuals are a measure of the discrepancy between the observed outcome and the predicted probability for each observation. For an observation i with binary response y_i and predicted probability $\hat{p_i}$, the deviance residual d_i is defined as:

$$d_i = \text{sign}(y_i - \hat{p}_i)\sqrt{2[y_i \log(\frac{y_i}{\hat{p_i}}) + (1-y_i)\log(\frac{(1-y_i)}{(1-\hat{p_i})})]} \tag{2.3}$$

The residual deviance is a measure of the goodness-of-fit of a logistic regression model, which is defined as the sum of the squared deviance residuals:

$$D = \sum_{i=1}^{n} d_i^2 \tag{2.4}$$

Where n is the number of observations. A lower residual deviance indicates a better fit of the logistic regression model to the data. It can be used to compare different models or assess the significance of predictor variables by comparing the change in residual deviance when a variable is added or removed from the model. The null deviance is calculated for the model with no slope.

The logistic function, denoted by $\sigma(z)$, is defined as:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.5}$$

where $\mathbf{z} = \mathbf{x} * \mathbf{w} + \mathbf{bias}$, $\mathbf{w}$ is a vector of weights and $\mathbf{x}$ is the input feature vector.

In logistic regression, we model the probability that an input $\mathbf{x}$ belongs to class 1 as:

$$P(y=1|\mathbf{x};\mathbf{w}) = \sigma(\mathbf{w}^T\mathbf{x}) \tag{2.6}$$

To train the logistic regression model, we aim to maximize the log-likelihood of the data. Given a dataset of N samples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, where $y_i \in \{0,1\}$, the log-likelihood is:

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N}[y_i \log \sigma(\mathbf{w}^T\mathbf{x}_i) + (1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i))] \tag{2.7}$$

To maximize the log-likelihood, we use gradient ascent.

Gradient ascent is an iterative optimization algorithm that is widely used to find the local maxima. [13] In our project, we computed the likelihood function and proceeded with finding the optimal weight values to maximize the function. In order to do so, gradient ascent was implemented on the parameters. Our weights were iteratively updated, moving in the direction of the gradient until the maxima or optimum weight value was reached.

The gradient of the log-likelihood with respect to the weights is:

$$\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N}(y_i - \sigma(\mathbf{w}^T\mathbf{x}_i))\mathbf{x}_i \tag{2.8}$$

In gradient ascent, we iteratively update the weights using the gradient of the log-likelihood:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\nabla_{\mathbf{w}}\mathcal{L}(\mathbf{w}) \tag{2.9}$$

Where α is the learning rate.

The cost function $(J(\theta))$ for logistic regression is the negative log-likelihood, which we want to minimize:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log h(x^{(i)}) + (1-y^{(i)})\log(1-h(x^{(i)}))] \tag{2.10}$$

Gradient descent is an iterative optimization algorithm, which is used in our project to minimize the cost function by finding the optimal weight value. This algorithm iteratively updates the parameters (weight values) by moving towards the opposite direction of the gradient, or simply, the negative gradient. The algorithm stops iterating once the cost function is minimized or close to zero. [12] In other words, a convergence is reached once the error (the difference between the predicted and actual outcome) is less than the tolerance value. At this point, the local minima and optimal weight values are found.

For gradient descent, we will iteratively update the parameters θ by subtracting the gradient of the cost function with respect to. θ:

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j} \tag{2.11}$$

2.2 K-Nearest Neighbors Algorithm

The second machine learning algorithm we employed is K-Nearest Neighbor. K-NN is a non-parametric, supervised machine learning algorithm intended for classification problems. The algorithm uses relative proximities to make classifications based on the groupings of the individual data point. Essentially, it computes distances between the query point and the training data to make its classification regarding the new test data. To calculate the ârelative proximities,â K-NN employs different distance metrics. Given a dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, where $\mathbf{x}_i \in \mathbb{R}^d$ is a feature vector and y_i is its associated label or target, the KNN algorithm aims to predict the label of a new, unseen data point $\mathbf{x}$ based on the labels of its k nearest neighbors in the dataset.

The first step in the KNN algorithm is to determine the distance between the new data point $\mathbf{x}$ and all data points in the dataset. For our data, we used the Euclidean distance â measuring the distance of the straight line between our query point and with our training dataset. Other distance options include (but are not limited to): Manhattan distance, Minkowski Distance, Hamming Distance, etc.

Euclidean Distance:

$$d(\mathbf{x}, \mathbf{x}_i) = \sqrt{\sum_{j=1}^{d} (\mathbf{x}_j - \mathbf{x}_{i,j})^2} \tag{2.12}$$

Once the distances are computed, the next step is to identify the k nearest neighbors of $\mathbf{x}$. The algorithm decides a number âkâ to choose the number of neighbors to make our new prediction with; for example, if k = 5, then the algorithm will look for the 5 nearest neighbors based on the smallest distance values to make its prediction. For our choice of k, we created a plot of error rate vs. K value and visually chose the value of K at which the error rate is lowest; with our data we found that the lowest error was found at k = 25.

For classification problems, the KNN algorithm predicts the class label of $\mathbf{x}$ based on the majority class of its k nearest neighbors. The predicted class label $\hat{y}$ can be computed as follows:

$$\hat{y} = \arg\max_{c \in \mathcal{C}} \sum_{i=1}^{k} I(y_i = c) \tag{2.13}$$

where $\mathcal{C}$ is the set of possible class labels, and I is an indicator function that takes the value 1 if the condition inside the parentheses is true, and 0 otherwise.

For regression problems, the KNN algorithm predicts the target value of $\mathbf{x}$ based on the average of the target values of its k nearest neighbors. The predicted target value $\hat{y}$ can be computed as follows:

$$\hat{y} = \frac{1}{k} \sum_{i=1}^{k} y_i \tag{2.14}$$

It is important to note that K-NN is known as a lazy algorithm due to its slow computation speed. The reason behind this is simple: KNN does not learn any discriminative function from the training data.

Rather, it memorized the entire training dataset instead for each iteration; while this can reduce training time, the prediction step becomes more time-consuming and computationally expensive. Essentially, each time a new data point comes in and we want to make a prediction, the KNN algorithm will search for the nearest neighbors in the entire training set which takes very long if our dataset is large. That being said, to optimize our algorithm, we utilized parallel computing techniques in order to run the algorithm on several different testing data points at once. The results from our code in terms of computational efficiency and prediction accuracy are shown below.

2.3 Naive Bayes Classifier

Naive Bayes is an algorithm that classifies dependent categorical variables, which would be on-time versus delayed flights in our case, using prior information. The term 'naive' in the Naive Bayes Algorithm refers to the assumption that the predictors are independent of each other. [5] Therefore, this assumption allows us to easily calculate the marginal probabilities of the predictors by simply taking the product of their probability distributions.

Bayes' theorem describes the relationship between the conditional probabilities of events. Given two events A and B, Bayes' theorem is expressed as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.15}$$

In the context of classification, we want to find the most likely class C_k for a given input $\mathbf{x}$. Using Bayes' theorem, this can be written as:

$$C_k = \arg\max_k P(C_k|\mathbf{x}) = \arg\max_k \frac{P(\mathbf{x}|C_k)P(C_k)}{P(\mathbf{x})} \tag{2.16}$$

Since $P(\mathbf{x})$ is constant for all classes, we can ignore it and focus on maximizing the numerator:

$$C_k = \arg\max_k P(\mathbf{x}|C_k)P(C_k) \tag{2.17}$$

The Naive Bayes Classifier makes a strong assumption that the features are conditionally independent given the class. For an input with n features $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, this assumption can be written as:

$$P(\mathbf{x}|C_k) = P(x_1, x_2, \ldots, x_n|C_k) = \prod_{i=1}^{n} P(x_i|C_k) \tag{2.18}$$

Applying the naive assumption to the classification problem, we get the Naive Bayes classification rule:

$$C_k = \arg\max_k \left[P(C_k) \prod_{i=1}^{n} P(x_i|C_k) \right] \tag{2.19}$$

To optimize the Bayes algorithm code from the built-in sklearn algorithm, we first minimized the use of object-oriented programming, which can slow down computation time. Next, we implemented parallel computing to parallel-ey compute the conditional probabilities of each of the predictor observations. To accomplish this, we used multiprocessing with 5 pools to compute the likelihood functions for all of the observations in parallel. [5]

The accuracy did not change significantly between using the built-in, sklearn Naive Bayes algorithm and the algorithm we wrote from scratch with parallel computing. For both algorithms, the accuracy for predicting whether a flight would be delayed or not was about 56%, which is not very high. However, the runtime in the algorithm written from scratch reduced as compared to the built-in package runtime.

In terms of accuracy, both the built-in and Bayes algorithm from scratch rendered a high value of true positives, which correctly predicts that a flight is delayed. Both the algorithms did not render a high value of false positives, which is the false prediction of a flight being delayed. Furthermore, both algorithms rendered a low value for true negatives which is the accurate prediction that a flight is on-time. The built-in function, however, got a lower value of false negatives than the algorithm from scratch, which is the inaccurate prediction of a flight being on-time.

2.4 A Multi-Layer Perceptron

A Multi-Layer Perceptron (MLP) is a type of feedforward artificial neural network that consists of multiple layers of neurons interconnected with weighted connections. A two hidden layer perceptron, with a sigmoid function and its derivative for forward propagation and backward propagation were respectively implemented. The hidden layer size for both layers is 32, its learning rate is 0.01, and it goes through 500 iterations

Other common activation functions include the sigmoid function (used in this paper), the hyperbolic tangent function, and the Rectified Linear Unit (ReLU) function. Given an input $\mathbf{x}$, the forward propagation process computes the output of the MLP by successively computing the weighted sum of inputs and applying the activation function for each neuron in each layer. For a neuron j in layer l, the weighted sum $z_j^{(l)}$ and the activation $a_j^{(l)}$ are computed as follows:

$$z_j^{(l)} = \sum_i w_{ij}^{(l-1)} a_i^{(l-1)} + b_j^{(l)} \tag{2.20}$$

$$a_j^{(l)} = f(z_j^{(l)}) \tag{2.21}$$

Where $w_{ij}^{(l-1)}$ is the weight connecting neuron i in layer $l-1$ to neuron j in layer l, $b_j^{(l)}$ is the bias term for neuron j in layer l, and $f(\cdot)$ is the activation function. To train an MLP, the backpropagation algorithm is used to compute the gradient of the loss function with respect to each weight and bias in the network. The gradient is then used to update the weights and biases using gradient descent or another optimization algorithm. The backpropagation algorithm consists of two steps:

1. Compute the error at the output layer:

$$\delta_j^{(L)} = (a_j^{(L)} - y_j) \cdot f'(z_j^{(L)}) \tag{2.22}$$

   Where L is the last layer, y_j is the target output, and $f'(\cdot)$ is the derivative of the activation function.

2. Compute the error for each neuron in the hidden layers:

$$\delta_j^{(l)} = \left(\sum_k \delta_k^{(l+1)} w_{jk}^{(l)} \right) \cdot f'(z_j^{(l)}) \tag{2.23}$$

After computing the error terms $\delta_j^{(l)}$ using the backpropagation algorithm, the weights and biases are updated using an optimization algorithm, in this case a gradient descent:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \tag{2.24}$$

$$b_j^{(l)} \leftarrow b_j^{(l)} - \alpha \frac{\partial \mathcal{L}}{\partial b_j^{(l)}} \tag{2.25}$$

Where α is the learning rate, and the gradients are computed as:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} = a_i^{(l-1)} \delta_j^{(l)} \tag{2.26}$$

$$\frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{2.27}$$

3 Simulation Study

The purpose of our simulation study is to evaluate the performance of each proposed machine learning algorithm that we have implemented so far using python. More specifically, we are interested in learning more about the effect of different sample sizes on the computational performance and the predictive accuracies for each of our algorithms. With this, we generated two separate datasets using the $make\_classification$ function: Small simulated data with 300 observations and large simulated data with 15,000 observations. For both of these datasets, we have included 6 informative covariates and one binary response variable. Furthermore, this simulated study is important because our simulated dataset is built to meet all model assumptions for logistic regression and to ensure we can run this algorithm effectively. Essentially, we ran each of our proposed methods ten times on each dataset and computed the average prediction accuracy based on all of the runs. The results from the all our simulated runs are included below:

Simulated Data - 300 observations

Logistic Regression G. Ascent:	Accuracy: 0.9133333333333334	Runtime: 0.05619406700134277 seconds
Logistic Regression G. Descent:	Accuracy: 0.9133333333333334	Runtime: 0.02618408203125 seconds
Naive Bayes :	Accuracy: 0.9133333333333334	Runtime: 0.11333608627319336 seconds
K-Nearest Neighbor	Accuracy: 0.9133333333333334	Runtime: 0.661552906036377 seconds
Neural Networks (MLP):	Accuracy: 0.5833333333333334	Runtime: 1.641437292098999 seconds

Simulated Data - 15,000 observations

Logistic Regression G. Ascent:	Accuracy: 0.8605333333333333	Runtime: 0.6370530128479004 seconds
Logistic Regression G. Descent:	Accuracy: 0.8977333333333334	Runtime: 0.5909037590026855 seconds
Naive Bayes :	Accuracy: 0.893000000000000	Runtime: 1.94310998916625986 seconds
K-Nearest Neighbor	Accuracy: 0.86493222	Runtime: 987.34 seconds
Neural Networks (MLP):	Accuracy: 0.5613333333333334	Runtime: 16.658402681350708 second

Based on our initial results above, we can see that each algorithm had extremely high average accuracy ratings with our both datasets with the exception of the Neural Networks model. More specifically, we can see that each method (except NN) has an average over accuracy rate over 90% with the small simulated dataset; we can see that both optimized logistic regression models, Naive Bayes, and K-NN algorithms resulted in the same prediction accuracy. It is important to note that our multi-layer perceptron has significantly lower accuracy with only 58%, which is significantly lower than our other models. This might indicate to us that our implementation of the neural network model might not be suitable in terms of predictive accuracy. Furthermore, we can notice very slight computational time differences among the methods. This makes sense as we run these algorithms on a very small dataset to observe true differences in computation speeds between our methods. In particular, the gradient descent algorithm had the fastest computation time with 0.026 seconds â this was approximately twice as fast as our gradient ascent optimization algorithm. Similarly, the Naive Bayes had a computation time of 0.11 seconds which is still very fast. K-NN and Neural network models were much slower computationally â we can run our algorithms on the larger simulated dataset to see more pronounced differences between our algorithms.

Moving on to the larger simulated dataset, we can see that the average accuracy for each of our methods has reduced slightly. For instance, optimized logistic regression with gradient ascent now has an average prediction accuracy of 86% over 10 different runs; the optimized logistic regression model with gradient descent has an average prediction accuracy of 89% (slightly higher than our other optimization method).

Furthermore, we can see that the computation speed for our gradient descent algorithm is the fastest with 0.6 seconds. In comparison, our algorithm with gradient ascent is just slightly slower (almost negligible) with the run-time of 0.63 seconds. Similarly, the computation speed of our Naive Bayes algorithm resulted at 1.94 seconds (slower than both optimized LR models) but had 89% prediction accuracy, which is equal to our result from the gradient descent algorithm. Additionally, the run-time for the K-NN algorithm with a large dataset is over 900 seconds and is significantly slower than every other method. This makes sense as weâve explained that K-NN is a lazy algorithm â we wonât select this method due to it being very expensive computationally. Again, our NN model had a much lower accuracy at 56% and had a run-time of approximately 16 seconds. Thus, in terms of computation efficiency, we found that every method saw an increase in speed when tested on the bigger dataset, which intuitively makes sense as an increased volume of data is being analyzed. However, it is notable that the increase in computation time was not proportional among the methods, as evident by our increase through the K-NN algorithm. Based on these results, we can conclude that optimized logistic regression methods with gradient descent and the Naive Bayes algorithm produced the best accuracy scores, with the gradient descent implementation being significantly faster than Naive Bayes. Ultimately, we can choose our logistic regression model with gradient descent as it has a most appropriate balance between high prediction accuracies and a fast computation time. We can now move on to conducting our analysis on our âAirlines Delayâ real world dataset to see how each of these methods fared.

4 Real Data Study

We can now move on to testing our proposed machine learning algorithms on the real dataset extracted from kaggle. In order to proceed with logistic regression, it is necessary to check whether each of the model assumptions are met. In binary logistic regression, the dependent variable must be binary. For this dataset, this condition is met as our response variable is binary (0 for plane on time, 1 for plane being delayed). In addition, each of the observations must be independent of each other. In this case, because each individual flight does not have an impact on the other flights, this dataset passes this assumption. Furthermore, using VIF scores and the correlation plot, we were able to determine how much multicollinearity there was between the predictors. Based on the correlation plot and the VIF scores being relatively low, we can conclude that there is an insignificant amount of multicollinearity between the predictors. Also, the diagnostic plots show that there is linearity between the predictors and the log odds of the dependent variable. Finally, the dataset we are working with has 10,000 entries, making the sample size large enough for this algorithm. Overall, we can safely continue with binary logistic regression as the dataset satisfies all of the assumptions.

We can now test the effectiveness of our proposed algorithms on a real dataset by randomly splitting our data in test/train splits. More specifically, we split the dataset with 80% of the data being assigned for training and the remaining 20% of it being assigned for testing and finding prediction accuracies. The five implemented algorithms are to be tested on the actual training and testing data for the binary classification problem. For each implemented algorithm as explained in the above sections, the running time and the accuracy scores across 10 runs were compared. The results obtained for each algorithm are as displayed below, and compared to the simulated data most of the classifiers have slightly lower accuracy scores and increased the required running time as well.

Sample Data - 10,000 observations

Logistic Regression G. Ascent:	Accuracy: 0.5589999999999999	Runtime: 0.5636610984802246 seconds
Logistic Regression G. Descent:	Accuracy: 0.5492	Runtime: 0.3654489517211914 seconds
Naive Bayes (our implementation) :	Accuracy: 0.5608	Runtime: 1.2618098258972168 seconds
K-Nearest Neighbor	Accuracy: 0.525	Runtime: 894 seconds
Neural Networks (MLP):	Accuracy: 0.5565	Runtime: 11.47421383857727 seconds

5 Conclusion

Many factors can lead to a flight being delayed, which may or may not be the airline's fault. While this problem involves a lot of uncertainty, our project focused on predicting whether or not a flight would be delayed based on the information we were given, including the time, day, length, and departure/destination of the flight. To do so, we implemented Logistic Regression with gradient descent and ascent, as well as K-Nearest Neighbor with parallel computing, Naive Bayes with parallel computing, and Multilayer Perceptron in order to weigh which model best predicted whether a flight would be delayed and which model produced the best runtime. Based on our results, we found that the Logistic Regression model optimized with the gradient descent algorithm had the fastest computation time out of all the methods, while also having an approximately 55% prediction accuracy. Alternatively, the Naive Bayes algorithm had the highest accuracy rating with 56% (only a 1% increase from the gradient descent algorithm), but had a slightly higher computation time. Since the accuracy does not substantially differ between these two best-performing algorithms, we conclude that the Logistic Model with gradient descent is better suited for our dataset since it has a faster runtime and comparable prediction accuracy.

It can be noted that our models did not yield very high accuracy, which we can potentially attribute to our datasets' lack of explanatory variables for why a flight was delayed (i.e., extreme weather, machine malfunction, etc.). Factors, such as weather, would likely have a high impact on whether a flight is delayed or not. Another reason our accuracy could have been low is because of the large amount of observations (10,000) that our models were trained with. However, while this was the case, we were able to optimize our code using various methods from this course, including gradient descent and parallel computing, which is important for processing large datasets like ours.

6 References

[1] Thanda, Anamika, et al. âWhat Is Logistic Regression? A Beginner's Guide [2023].â CareerFoundry, 19 Dec. 2022, https://careerfoundry.com/en/blog/data-analytics/what-is-logistic-regression/.

[2] â2022 Southwest Airlines Scheduling Crisis.â Wikipedia, Wikimedia Foundation, 16 Feb. 2023, https://en.wikipedia.org/wiki/2022$_{S}outhwest_{A}irlines_{s}cheduling_{c}risis$.

[3] Baran, Michelle. âHow Did Air Travel Get so Bad?â AFAR Media, AFAR Media, 25 Jul. 2022, https://www.afar.com/magazine/why-there-are-so-many-delayed-and-canceled-flights-in-2022.

[4] Qin, Amy, et al. âThousands of Canceled Flights Cap Holiday Weekend of Travel Nightmares.â The New York Times, The New York Times, 26 Dec. 2022, https://www.nytimes.com/2022/12/26/us/flights-canceled-holiday-travel.html.

[5] Loeber, 'Patrick. âNaive Bayes in Python - ML from Scratch 05.â Python Engineer, 29 Sept. 2019, https://www.python-engineer.com/courses/mlfromscratch/05$_{n}aivebayes/$.

[6] Stanina, Iuliia. âImplementing Naive Bayes Algorithm from Scratchâ-âPython.â Medium, Towards Data Science, 22 Oct. 2020, https://towardsdatascience.com/implementing-naive-bayes-algorithm-from-scratch-python-c6880cfc9c41.

[7] S, Surabhi. âGet Started with Naive Bayes Algorithm: Theory Implementation.â Analytics Vidhya, 16 Jan. 2021, https://www.analyticsvidhya.com/blog/2021/01/a-guide-to-the-naive-bayes-algorithm/.

[8] Tarhini, Ali. âParallel NaÃ¯ve Bayesian Classifier.â Ali Tarhini, 2 Mar. 2011, https://alitarhini.wordpress.com/2011/03/02/parallel-naive-bayesian-classifier/.

[9] Vamsi, Ranga. âNaÃ¯ve Bayes Algorithm -Implementation from Scratch in Python.â Medium, Medium, 14 July 2020, https://medium.com/%40rangavamsi5/na%C3%AFve-bayes-algorithm-implementation-from-scratch-in-python-7b2cc39268b9.

[10] Jessica, Sonia. âHow Does Logistic Regression Work?â
KDnuggets, https://www.kdnuggets.com/2022/07/logistic-regression-work.html.

[11] TokuÃ§, Written by: A. Aylin. âGradient Descent Equation in Logistic Regression.â Baeldung on Computer Science, 11 Nov. 2022, https://www.baeldung.com/cs/gradient-descent-logistic-regression.

[12] âWhat Is Gradient Descent?â IBM, https://www.ibm.com/topics/gradient-descent.

[13] Vasugupta. âImplementation of Gradient Ascent Using Logistic Regression.â Medium, Medium, 10 Nov. 2020, https://vasugupta2000.medium.com/implementation-of-gradient-ascent-using-logistic-regression-7f5343877c21.

[14] Jason Brownlee, âDevelop k-Nearest Neighbors in Python From Scratchâ, Machine Learning Mastery, https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/.

[15] Hoang Phong, âGradient Descent for Logistics Regression in Pythonâ, Medium, https://medium.com/@IwriteDSblog/gradient-descent-for-logistics-regression-in-python-18e033775082

[16] Rumelhart, David E., Geoffrey E. Hinton, and R. J. Williams. "Learning Internal Representations by Error Propagation". David E. Rumelhart, James L. McClelland, and the PDP research group. (editors), Parallel distributed processing: Explorations in the microstructure of cognition, Volume 1: Foundation. MIT Press, 1986. https://apps.dtic.mil/dtic/tr/fulltext/u2/a164453.pdf

7 Appendix: Python Script

7.1 Data Acquisition/Pre-Processing/Exploratory Data analysis:

```
# Import libraries
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import train_test_split
from sklearn import linear_model
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
from sklearn.metrics import confusion_matrix
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
import time
import warnings
from datetime import datetime
from sklearn import preprocessing
init_notebook_mode(connected=True)
import multiprocessing
from multiprocessing.pool import ThreadPool
import timeit
warnings.filterwarnings('ignore')

# Data pre-processing
df = pd.read_csv("airlines_delay.csv", sep = ",") # read dataset
df.info() # check data

# Do label encoding for the categorical variables
AirlineUnique = df.Airline.unique() # get unique levels
AirportFromUnique = df.AirportFrom.unique()
AirportToUnique = df.AirportFrom.unique()

lenAirlineUnique = len(df.Airline.unique()) # length of list
lenAirportFromUnique = len(df.AirportFrom.unique())
lenAirportToUnique = len(df.AirportTo.unique())

Airlinelst = list(range(0,lenAirlineUnique)) # create list of numbers
```

```
df['NumAirline'] = df['Airline'] # label encode
df['NumAirline'].replace(AirlineUnique, Airlinelst , inplace=True)

AirportFromlst = list(range(0,lenAirportFromUnique))
df['NumAirportFrom'] = df['AirportFrom']
df['NumAirportFrom'].replace(AirportFromUnique, AirportFromlst , inplace=True)

AirportTolst = list(range(0,lenAirportToUnique))
df['NumAirportTo'] = df['AirportTo']
df['NumAirportTo'].replace(AirportToUnique, AirportTolst , inplace=True)

df = df.sample(n=10000) # Get reduced dataset

X = df[['Length','NumAirline', 'NumAirportFrom','NumAirportTo',
'DayOfWeek']] # get X dataframe
y = df['Class'] # get response

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

# Create Simulated Dataset
from sklearn.datasets import make_classification

# Get simulated Data
X_simulated_small, y_simulated_small = make_classification(
    n_samples=300,
    n_features=6,
    n_classes=2,
    random_state=1
)

X_simulated_large, y_simulated_large = make_classification(
    n_samples=15000,
    n_features=6,
    n_classes=2,
    random_state=1
)

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

# %delays by airline
flights_per_airline = df.groupby('Airline').agg({'Class':'count'})
# number of flights by airline
delays_by_airline =df.groupby('Airline').agg({'Class':'sum'})
# summing up delays by each airline
flights_per_airline['delays'] = delays_by_airline['Class']
# total flights and delays per airline
flights_per_airline.rename(columns={"Class":"flights"})
# renaming col
flights_per_airline['%_delays'] = flights_per_airline['delays']/flights_per_airline.iloc[:
# % of delays per airline
flights_per_airline = flights_per_airline.reset_index()
```

```
# resetting index
flights_per_airline


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


# Number of Flights On time and Delayed
plt.title('Number of Flights On Time and Delayed by Airline', fontsize=16)
fig = plt.figure(1, figsize=(10,7))
ax = sns.countplot(y="Airline", hue='Class', data=df) # getting horizontal boxplots
# formatting legend
legend= plt.legend()
legend.get_texts()[1].set_text('On-time')
legend.get_texts()[0].set_text('Delayed')
plt.xlabel('Number of Flights')
plt.show()

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

# Percent Delays by Airline
sns.barplot(y= 'Airline', x= '%_delays', data=flights_per_airline, orient='h', color='C0')
# plotting percentage of flights delayed by airline
plt.xlabel('Percent of Flights Delayed')
plt.title('Percent of Delays by Airline', fontsize=16)

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

delayed = df[df['Class']==1] # subset df of only delayed flights
fig, ax = plt.subplots(figsize = (10,5))
ax.hist([df['Length'], delayed['Length']], label=['All', 'Delayed'], density = True)
# histogram of lengths of flights for delayed vs. on-time flights
ax.set_xlabel('Length of Flight')
ax.set_title('Normalized Histogram of Length of Flight (All vs. Delayed Flights)')
plt.legend()
plt.show()

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

delayed = df[df['Class']==1]
# subset df of only delayed flights
fig, ax = plt.subplots(figsize = (10,5))
ax.hist([df['DayOfWeek'], delayed['DayOfWeek']], label=['All', 'Delayed'], density=True)
# histogram of lengths of flights for delayed vs. on-time flights
ax.set_xlabel('Day of Week')
ax.set_title('Normalized Histogram of Day of Week (All vs. Delayed Flights)')
plt.legend()
plt.show()

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

```
# Number of delays by day of week and time of the day (DC)
time_bins = [0]
initial_time = 0
for i in range(24): # for 24 hours in day
    initial_time += 60 # every 180 minutes, there's a new hour
    time_bins.append(initial_time) # storing all minute counts of the day

# Labels for Hours of the Day
time_labels = ['0','1','2','3','4','5','6','7','8','9', '10', '11',
        '12','13','14','15','16','17','18','19','20','21', '22', '23']

df['Hours'] = pd.cut(x = df['Time'],
bins = time_bins, labels = time_labels, right = False, include_lowest = True)
# Adding Hours of the Day to dataframe

sub_df = df[df.Class != 0] # only keeping flights that were delayed
sub_df = sub_df.groupby(['DayOfWeek', 'Hours','Class']) # subsetting data
num_delays = sub_df.Class.sum() # Getting # of delays by day of week and time of day
sub_df = num_delays.reset_index(name = 'Delays') # resetting index
sub_df = sub_df.drop(['Class'], axis = 1).reset_index() # dropping class col
sub_df = sub_df.pivot_table(values='Delays', index='Hours', columns='DayOfWeek',
aggfunc='sum')
# pivoting data frame

fig, ax = plt.subplots(figsize=(20, 10)) # set figure size
sns.heatmap(sub_df, annot=True, linewidth=.5, cmap="Blues",ax = ax) # create heat plot
plt.title('Number of Delays by Time of Day and Day of Week')
plt.show()
```

7.2 Optimization Logistic Regression:

In order to optimize our Logistic Regression model, our goal was to choose the best values for theta and to minimize the error. A cost function was defined to measure the difference between the predicted and actual probability values. First, we generated a vector with random values to use as our initial guess for the weight values. Gradient descent was then implemented to update the weights each iteration. To compute the prediction probabilities, theta*X was passed through the sigmoid function, returning a value of 0 or 1, representing the probability of each class occurring (on-time vs. delayed flights). In each iteration, the weights were updated in the direction of the negative gradient; the algorithm converged once the cost function reached a local minima and the optimum theta value was found.

Similarly, Gradient Ascent was also implemented to optimize the model. First, we initiated the weight values with 0âs and defined our likelihood model. We then proceeded to compute the gradient of our log-likelihood model, which was found by taking the partial derivative of logL with respect to theta. The gradient was then used to update the weight values through a gradient ascent algorithm; the weights were iteratively updated in the direction of the gradient until the local maxima or optimal weight values were found.

```
# Logistic Regression
class logregwMLE:
    '''Reference: https://vasugupta2000.medium.com/implementation-
    of-grad-ascent-using-logistic-regression-7f5343877c21
    Author: Vasugupta '''
    def __init__(self, lr=0.01, numiterations=100):
        '''Initialize lr and numiterations variables. 'lr' is the learning rate,
        'numiterations' is the max number of iterations.'''
        self.lr  = lr # initialize learning rate & number of iterations
        self.numiterations = numiterations
        self.lls = []
        self.eps = 1e-10 # log(0) does not exist
        # so set epsilon=1e-10 to prevent log(0) error

    def sig(self, z):
        '''Computes sigmoid function:
        sigmoid(z)=1/(1+e^(-z)), where z=transpose(theta)*x'''
        sigmoidz = (1/(1+np.exp(-z)))
        return sigmoidz

    def logL(self, ycorrect, predY):
        '''Computes the maximum log-likelihood function. ycorrect is the true y values,
        predY are the predicted y values. '''
        predY = np.maximum(np.full(predY.shape, self.eps),
        np.minimum(np.full(predY.shape, 1-self.eps), predY))
        # make sure that there are no 0 or 1 values to prevent log # error
        ll = (ycorrect*np.log(predY)+(1-ycorrect)*np.log(1-predY))
        # compute likelihood function
        llavg = ll.mean()
        # mean of likelihood
        return llavg

    def fit(self, X, y):
        '''fit function to later fit X_train
        and y_train to the logistic regression model'''
        numex = X.shape[0] # examples
        feats = X.shape[1] # features
        self.wghs = np.zeros((X.shape[1]))
        # choose weight values

        # apply gradient ascent
        for i in range(self.numiterations):
            z  = X @ self.wghs
            predY = self.sig(z)
            # compute sigmoid(z) to find probability values
            grad = np.mean((y-predY)*X.T, axis=1)
            # compute gradient
            self.wghs +=  self.lr*grad
            # update the weights with respect to the gradient
            # compute log-likelihood function
            ll = self.logL(y,predY)
            self.lls.append(ll)
            # add to the list

    def predprob(self,X):
        '''Returns the prediction probabilities (0 or 1) for X'''
        if self.wghs is None:
            raise Exception('Need to fit the model')
            # make sure that the model has been fitted
        z = X @ self.wghs
        # compute z using the formula z = transpose(theta)*X = weights*X
        probs = self.sig(z)
        # 0 or 1 probabilities
        return probs

    def predict(self, X, threshold=0.5):
        '''Function to classify X values based given threshold=0.5.
        Threshold sets x value to 0 if not equal to 1'''
        bpredY = np.array(list(map(lambda x: 1 if x>threshold else 0, self.predprob(X))))
        # binary predictions, 0 or 1
        return bpredY


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

# Simulated Study Gradient Ascent (small data)
gamle_start = time.time() # start time
model = logregwMLE()
accuracy_list_gda = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_simulated_small,
    y_simulated_small, test_size=0.2, random_state=i)
    model.fit(X_train,y_train)
    predY = model.predict(X_test)
    gamle_accuracy = test_accuracy(y_test, predY)
    accuracy_list_gda.append(gamle_accuracy)

avg_accuracy_gda = sum(accuracy_list_gda)/len(accuracy_list_gda)

gamle_end = time.time() # end time
gamle_speed = gamle_end - gamle_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_gda}')
print(f'Training time for Logistic Regression with Gradient Ascent: {gamle_speed}')

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

# Simulated Study Gradient Ascent (large data)
gamle_start = time.time() # start time
model = logregwMLE()
accuracy_list_gda = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_simulated_large,
    y_simulated_large, test_size=0.2, random_state=i)
    model.fit(X_train,y_train)
    predY = model.predict(X_test)
    gamle_accuracy = test_accuracy(y_test, predY)
    accuracy_list_gda.append(gamle_accuracy)

avg_accuracy_gda = sum(accuracy_list_gda)/len(accuracy_list_gda)

gamle_end = time.time() # end time
gamle_speed = gamle_end - gamle_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_gda}')
print(f'Training time for Logistic Regression with Gradient Ascent: {gamle_speed}')

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

# Real data study
gamle_start = time.time() # start time
model = logregwMLE()
accuracy_list_gda = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_std, y, test_size=0.2,
    random_state=i)
    model.fit(X_train,y_train)
    predY = model.predict(X_test)
    gamle_accuracy = test_accuracy(y_test, predY)
    accuracy_list_gda.append(gamle_accuracy)

avg_accuracy_gda = sum(accuracy_list_gda)/len(accuracy_list_gda)

gamle_end = time.time() # end time
gamle_speed = gamle_end - gamle_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_gda}')
print(f'Training time for Logistic Regression with Gradient Ascent: {gamle_speed}')

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

import math

# define sigmoid function
def sig(X, weight):
    z = X @ weight
    return 1 / (1 + np.exp(-z))

# gradient descent function
def graddescent(X, g, y):
    gd = (X.T @ (g - y)) / y.shape[0]
    return gd

# function to update weight loss
def new_weightloss(weight, lr, grad):
    return (weight - lr * grad)

def generateX(X):
    """This function takes in the predictor variables and
    adds a row of 1's
    which corresponds to x_0"""
    vectorX = np.c_[np.ones((len(X), 1)), X]
    return vectorX

def get_initial_vec(X):
    """This function generate an
    initial value of vector Î¸ from the original independent variables matrix"""
    initial_vec = np.random.randn(len(X[0])+1, 1)
    return initial_vec

def sigmoid_function(X):
    """ This function calculates the
    sigmoid value of the inputs"""
    sigmoid_val = 1/(1+math.e**(-X))
    return sigmoid_val

def Logistic_Fit(X,y,learningrate, iterations):
    """This function creates a logistic model function for our
    dataset. X is the independent variables, y are dependent

    variables matrix learningrate is learningrate of Gradient
    Descent, iterations is the number of iterations. This

    function will return guess vector.
    """
    y_new = np.reshape(y, (len(y), 1))
    cost_lst = []
    vectorX = generateX(X)
    theta = get_initial_vec(X)
   m = len(X)
    for i in range(iterations):
        gradients = 2/m * vectorX.T.dot(sigmoid_function(vectorX.dot(theta)) - y_new)
        theta =
        theta - learningrate * gradients
    return theta

def column(matrix, i):
    """ This function will return all the values in a specific columns"""
    return [row[i] for row in matrix]


def accuracy_metric(X,y,learningrate, iteration,X_test, y_test):
    """ Returning the accuracy score for a training model"""

    fit_model = Logistic_Fit(X,y,learningrate, iteration)

    line = fit_model[0]

    for i in range(1,len(fit_model)):
        line = line + fit_model[i]*column(X_test,i-1)
    logistic_function = sigmoid_function(line)
    for i in range(len(logistic_function)):
        if logistic_function[i] >= 0.5:
            logistic_function[i] = 1
        else:
            logistic_function[i] = 0

    last1 = np.concatenate((logistic_function.reshape(len(logistic_function),
    1), y_test.reshape(len(y_test),1)),1)

    count = 0
    for i in range(len(y_test)):
        if last1[i][0] == last1[i][1]:
            count = count+1
    acc = count/(len(y_test))
    return acc

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

# real dataset
gamle_start = time.time() # start time
accuracy_list_gdd = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_simulated_small,
    y_simulated_small, test_size=0.2, random_state=i)

    accuracy_val = accuracy_metric(X_train,y_train, .1, 100,X_test, y_test)

    accuracy_list_gdd.append(accuracy_val)

avg_accuracy_gda = sum(accuracy_list_gdd)/len(accuracy_list_gdd)

gamle_end = time.time() # end time
gamle_speed = gamle_end - gamle_start
# get speed
print(f'Prediction accuracy of model: {avg_accuracy_gda}')
print(f'Training time for Logisitc Regression using
Gradient Ascent: {gamle_speed}')

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

# real dataset
gamle_start = time.time() # start time
accuracy_list_gdd = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_simulated_large,
    y_simulated_large, test_size=0.2, random_state=i)
    accuracy_val = accuracy_metric(X_train,y_train, .1, 100,X_test, y_test)
    accuracy_list_gdd.append(accuracy_val)
avg_accuracy_gda = sum(accuracy_list_gdd)/len(accuracy_list_gdd)

gamle_end = time.time() # end time
```

```
gamle_speed = gamle_end - gamle_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_gda}')
print(f'Training time for Logisitc Regression using Gradient Ascent: {gamle_speed}')

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

# real dataset
gamle_start = time.time() # start time
accuracy_list_gdd = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_std, y, test_size=0.2,
    random_state=i)
    accuracy_val = accuracy_metric(X_train,y_train, .1, 100,X_test, y_test)
    accuracy_list_gdd.append(accuracy_val)
avg_accuracy_gda = sum(accuracy_list_gdd)/len(accuracy_list_gdd)

gamle_end = time.time() # end time
gamle_speed = gamle_end - gamle_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_gda}')
print(f'Training time for Logisitc Regression using Gradient Ascent: {gamle_speed}')
```

7.3 K-NN:

K-NN is known as a very lazy algorithm as it is extremely computationally expensive to implement. The reason behind this is simple: KNN does not learn any discriminative function from the training data. Rather, the algorithm memorizes the entire training dataset instead for each iteration, making the prediction step much more time-consuming and computationally expensive. Essentially, each time a new data point comes in and we want to make a prediction, the K-NN algorithm will search for the nearest neighbors in the entire training set which takes very long if our dataset is large â which is the case for us. To optimize our algorithm with K-NN, we utilized parallel computing techniques in order to run the algorithm on several different testing data points at once rather than computing each test row at one at a time. The results from our code in terms of computational efficiency and prediction accuracy are shown below.

```
# Get K
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
X_train, X_test, y_train, y_test = train_test_split(X_std, y, test_size=0.2,
random_state=1, shuffle = True)
error_rate = []
for i in range(1,40):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train,y_train)
    pred_i = knn.predict(X_test)
    error_rate.append(np.mean(pred_i != y_test))

plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color='blue', linestyle='dashed',
         marker='o',markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
print("Minimum error:-",min(error_rate),"at K =",error_rate.index(min(error_rate)))
```

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

# Example of calculating Euclidean distance :: from source no changes
# calculate the Euclidean distance between two vectors
from math import sqrt

def euclidean(row1, row2):
    '''This function returns the euclidean distance between two rows'''
    dist = 0.0
    for i in range(len(row1)-1):
        dist += (row1[i] - row2[i])**2
    return sqrt(dist)

def get_neighbors(train, test_row, num_neighbors): # get neighbors
    '''This function returns the rows frow
    the training data based on the smallest distances'''
    i = 0
    index_list = list()
    distances = list()
    for train_row in train:
        dist = euclidean(test_row, train_row)
        distances.append((train_row, dist, i))
        i += 1
        distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append((distances[i][0],distances[i][2]))
    return neighbors

def predict_classification(x_train, y_train, test_row, num_neighbors):
    neighbors = get_neighbors(x_train, test_row, num_neighbors)
    index_list = list()
    for i in neighbors:
        index_list.append(i[1])
    output_values = y_train[index_list].tolist()
    prediction = max(set(output_values), key=output_values.count)
    return prediction

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

# Simulated Data - small
from joblib import Parallel, delayed
import multiprocessing
knn1_start = time.time() # start time
accurage_metric_list_1 = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_simulated_small,
    y_simulated_small, test_size=0.2, random_state=i)
    inputs = list(range(len(X_test)))
    def multi_run(t):
        return predict_classification(X_train, y_train, X_test[t], 25)
    num_cores = multiprocessing.cpu_count()
    results = Parallel(n_jobs=num_cores)(delayed(multi_run)(i) for i in inputs)
    accurage_metric_list_1.append(test_accuracy(y_test, results))
avg_accuracy_1 = sum(accurage_metric_list_1)/len(accurage_metric_list_1)

knn1_end = time.time() # end time
knn1_speed = knn1_end - knn1_start # time elapsed
print('K-NN runtime:', knn1_speed, 'seconds')
print('K-NN accuracy:', avg_accuracy_1, '%')

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

# Simulated Data - Large
knn1_start = time.time() # start time
accurage_metric_list_1 = list()
for i in range(0, 1):
    X_train, X_test, y_train, y_test = train_test_split(X_simulated_large,
    y_simulated_large, test_size=0.2, random_state=i)
    inputs = list(range(len(X_test)))
    def multi_run(t):
        return predict_classification(X_train, y_train, X_test[t], 25)
    num_cores = multiprocessing.cpu_count()
    results = Parallel(n_jobs=num_cores)(delayed(multi_run)(i) for i in inputs)
    accurage_metric_list_1.append(test_accuracy(y_test, results))
avg_accuracy_1 = sum(accurage_metric_list_1)/len(accurage_metric_list_1)

knn1_end = time.time() # end time
knn1_speed = knn1_end - knn1_start # time elapsed
print('K-NN runtime:', knn1_speed, 'seconds')
print('K-NN accuracy:', avg_accuracy_1, '%')

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

# real data test
knn2_start = time.time() # start time
accurage_metric_list_2 = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test = train_test_split(X_std, y, test_size=0.2,
    random_state=i)
    inputs = list(range(len(X_test)))
    def multi_run(t):
        return predict_classification(X_train, y_train, X_test[t], 25)
    num_cores = multiprocessing.cpu_count()
    results = Parallel(n_jobs=num_cores)(delayed(multi_run)(i) for i in inputs)
    accurage_metric_list_2.append(test_accuracy(y_test, results))
avg_accuracy_2 = sum(accurage_metric_list_2)/len(accurage_metric_list_2)

knn2_end = time.time() # end time
knn2_speed = knn2_end - knn2_start # time elapsed
print('K-NN runtime:', knn2_speed, 'seconds')
print('K-NN accuracy:', avg_accuracy_2, '%')
```

7.4 Naive Bayes:

To optimize the Bayes algorithm, we first tried to minimize the use of object-oriented programming and for loops, which can slow down computation time. Since computing the likelihood conditional probabilities for each observation requires iterating through all the predictors, we decide to implement parallel computing. To do this, we created 5 thread pools using the multiprocessing package in Python. We then parallel-ey computed the likelihood conditional probabilities for each factor observation in a function that we defined as *max_class*. After parallel-ey computing the probabilities, this function combines the likelihoods with prior information to find the posterior probability, and then returns the class (delayed or on-time) that has the maximal posterior probability.

```
def max_class(x, prior_list, mean_list, variance_list):
    ''' This function computes the likelihood for each x observation that it receives
    given each class. It multiplies
    this likelihood by the prior information
    (probability of the class), and then divides this by the marginal probability
    of the x observation.
    The function then returns the class that yields the maximum posterior.'''

    likelihoods = [] # storing likelihoods
    post = [] # storing posteriors
    for index in range(2): # for the two classes
        likelihood_num = np.exp((-1/2)*((x-mean_list[index])**2) /
        (2 * variance_list[index])) # calculating likelihood numerator
        likelihood_den = np.sqrt(2 * np.pi * variance_list[index])
        # calculating likelihood denominator
        likelihoods.append(likelihood_num/likelihood_den)
        # storing all likelihods
    post.append(np.log(prior_list[0]) + np.sum(np.log(likelihoods[0])))
    # getting posterior for 1st class
    post.append(np.log(prior_list[1]) + np.sum(np.log(likelihoods[1])))
    # getting posterior for 2nd class
    return(np.argmax(post))
    # getting class with max posterior


import multiprocessing
from multiprocessing.pool import ThreadPool
import timeit

def Parallel_NB(X_train, X_test, y_train):
    ''' This function takes in training and X testing data and references the max_class
    function to
    compute the posterior probability for both classes and get the
    class with the max posterior. It uses parallel

    computing via multiprocessing to parallely compute the conditional
    likelihood probabilities for each x observation.
    This function returns the output from the max_class function, which is
    the class that yields the maximum posterior.'''

    # Get mean, variance & prior
    n = len(X_train) # number of observations
    m = len(X_train.columns) # number of predictors
    class_types = y_train.unique() # all classes
    total_classes = 2 # 2 classes total
    prior_list = np.zeros(total_classes, dtype = float) # initializing prior
    mean_list = np.zeros((total_classes, m), dtype = float) # initializing mean
    variance_list = np.zeros((total_classes, m), dtype = float) # initializing variance
    for index in range(2): # classes are 0,1
            sub_df = X_train[y_train == index]
            # only keeping observations with first class
            prior_list[index] = len(sub_df) / n
            # all observations for one class
            mean_list[index, :] = sub_df.mean(axis=0)
            # getting means of each observation
            variance_list[index, :] = sub_df.var(axis=0)
            # getting variances of each observation
    X_testing = X_test.to_numpy() # converting X_test data frame to array
    Xis = []
    # storing each X observation from X_test
    #X_testing = X_testing.to_numpy()
    for Xi in X_testing:
        Xis.append(Xi.tolist()) # getting list of all X observations
    pool = ThreadPool(5) # creating pool
    y_predicted_parallel = [pool.apply(max_class, args=([Xi],
    prior_list, mean_list, variance_list)) for Xi in Xis]
    # parallely computing the posterior probability + getting best
    # class for each x observation
    return(y_predicted_parallel)
    # returning the indexes that yielded the highest posterior probabilityÃ

def test_accuracy(true_values, predicted_values):
    '''This function returns the accuracy % between the actual and predicted values'''
    correct_id = 0
    for i in range(len(true_values)):
        if true_values[i] == predicted_values[i]:
            correct_id += 1
    return correct_id / float(len(true_values))


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

# Simulated Study Naive Bayes (small data)
bayes_start = time.time() # start time
accuracy_list_bayes = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test =
    train_test_split(X_simulated_small, y_simulated_small, test_size=0.2, random_state=i)
    X_train = pd.DataFrame(X_train)
    X_test = pd.DataFrame(X_test)
    y_train = pd.Series(y_train)
    predY = Parallel_NB(X_train, X_test, y_train)
    bayes_accuracy = test_accuracy(y_test, predY)
    accuracy_list_bayes.append(bayes_accuracy)

avg_accuracy_bayes = sum(accuracy_list_bayes)/len(accuracy_list_bayes)

bayes_end = time.time() # end time
bayes_speed = bayes_end - bayes_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_bayes}')
print(f'Training time for Naive Bayes: {bayes_speed}')

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

# Simulated Study Naive Bayes (large data)
bayes_start = time.time() # start time
accuracy_list_bayes = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test =
    train_test_split(X_simulated_large, y_simulated_large, test_size=0.2,
    random_state=i)
    X_train = pd.DataFrame(X_train)
    X_test = pd.DataFrame(X_test)
    y_train = pd.Series(y_train)
    predY = Parallel_NB(X_train, X_test, y_train)
    bayes_accuracy = test_accuracy(y_test, predY)
    accuracy_list_bayes.append(bayes_accuracy)

avg_accuracy_bayes = sum(accuracy_list_bayes)/len(accuracy_list_bayes)

bayes_end = time.time() # end time
bayes_speed = bayes_end - bayes_start # get speed
print(f'Prediction accuracy of model: {avg_accuracy_bayes}')
print(f'Training time for Naive Bayes: {bayes_speed}')

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

# Real data study for Naive Bayes
naive_beg = time.time() # start time
accurage_metric_naive = list()
for i in range(0, 10):
    X_train, X_test, y_train, y_test =
    train_test_split(X, y, test_size=0.2, random_state=i, shuffle = True)
    y_predicted_parallel = Parallel_NB(X_train, X_test, y_train)
    # getting predicted y observations
    y_test = y_test.to_numpy()
    # convert to numpy
    naive_accuracy = test_accuracy(y_test, y_predicted_parallel)
    accurage_metric_naive.append(naive_accuracy)
```

```
avg_accuracy_naive= sum(accurage_metric_naive)/len(accurage_metric_naive)
naive_end = time.time() # end time
naive_speed = naive_end - naive_beg
# time elapsed

print(f'Prediction accuracy of model: {avg_accuracy_naive}')
print(f'Training time for Naive Bayes: {naive_speed}')
```

7.5 Multilayer Perceptron with Two Hidden Layers:

Our algorithm consists of several methods. First, a function initializes random weights by multiplying 0.01 and sets biases to zero. Second, forward propagation predicts values using a sigmoid function. Third, backward propagation computes gradients of the predicted error with respect to the input, weights, the derivative of the sigmoid, and biases. Fourth, parameters are updated and multiplied by 0.01. Lastly, an iterative method progresses from forward to back propagation, and updates parameters. Multiplying the weights by 0.01 avoids issues with large weight values, which can lead to large activations and cause the sigmoid to saturate, resulting in small or large gradients, slowing convergence or getting stuck in suboptimal solutions. MLPs face other issues, such as overfitting, getting trapped in a local minima, and time-consuming experimentation with layer numbers and sizes. Furthermore, MLPs' complex structure makes it difficult to interpret their predictions. The following heatmap was generated from all the data.

```
# Multilayer Perceptron with two hidden layers
import numpy as np
import pandas as pd
from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt
import scipy.sparse as sps
import time


# Initializes the weights (W1, W2, W3) and biases (b1, b2, b3)
# for the two layers (input-hidden and hidden-output) with random    # values

def initialize_parameters(input_size, hidden_size1, hidden_size2, output_size):

# W1: The weight matrix connecting the input layer to the hidden
# layer is initialized with random values from a
# normal distribution, scaled by a factor of 0.01.
# Its shape are the two hidden sizes and input_size.

    W1 = np.random.randn(hidden_size1, input_size) * 0.01

# b1: The bias vector for the hidden layer is initialized
# as an array of zeros with shape (hidden_size, 1).

    b1 = np.zeros((hidden_size1, 1))
    W2 = np.random.randn(hidden_size2, hidden_size1) * 0.01
    b2 = np.zeros((hidden_size2, 1))
    W3 = np.random.randn(output_size, hidden_size2) * 0.01

# b3: The bias vector for the output layer is initialized as
# an array of zeros with shape (output_size, 1).

    b3 = np.zeros((output_size, 1))

    parameters = {
        "W1": W1,
        "b1": b1,
        "W2": W2,
        "b2": b2,
        "W3": W3,
        "b3": b3
    }

    return parameters

# The sigmoid activation function, which is applied to the neurons' output.
# This function takes an input value x (scalar, vector, or matrix) and applies
# the sigmoid activation function element-wise.
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# The derivative of the sigmoid function, used for backpropagation.
def sigmoid_derivative(x):
    return x * (1 - x)

# The forward_propagation function computes the output of the neural
# network given its input X and the current values of
# its parameters (weights and biases). This function performs the following steps:
# Performs forward propagation through the network, calculating
# the output of each layer (A1, A2) and intermediate values (Z1, Z2)

def forward_propagation(X, parameters):

# Retrieve the weight matrices W1, W2 and the bias vectors
# b1, b2 from the parameters dictionary.

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

# Calculate Z1, the input to the hidden layer, as the
# dot product of W1 and X, and then add the bias vector b1.
# This represents the linear combination of the input features
# with the weights and biases of the connections between the
# input and hidden layers.

    Z1 = np.dot(W1, X) + b1

# Apply the sigmoid activation function to Z1 to compute A1,
# the output of the hidden layer.

    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

# Calculate Z3, the input to the output layer, as the dot
# product of W3 and A2, and then add the bias vector b3.
# This represents the linear combination of the hidden
# layer's output with the weights and biases of the
# connections between the hidden and output layers.

    Z3 = np.dot(W3, A2) + b3

# Apply the sigmoid activation function to Z3 to compute A3,
# the final output of the neural network.

    A3 = sigmoid(Z3)

# Create a dictionary called cache to store intermediate values
# (Z1, A1, Z2, A2) that will be needed during the backpropagation
# step.

    cache = {
        "Z1": Z1,
        "A1": A1,
        "Z2": Z2,
        "A2": A2,
        "Z3": Z3,
        "A3": A3
    }

# Notice only A3 is computed becuase thats the final ouput
# after forward propagation processes

    return A3, cache

# Computes the gradients of the neural network's parameters
# (weights and biases) with respect to the error made by
# the network during the forward propagation step. These
# gradients are later used to update the parameters and minimize
# the error during training. The function performs the
# following steps:
# Implements the backpropagation algorithm, calculating
# gradients for the weights and biases.
```

```
def backward_propagation(X, Y, cache, parameters):

# Compute the number of samples m from the input matrix X.

    m = X.shape[1]

# Retrieve the activations A1, A2 and A3
# (final output of the network) from the cache dictionary.

    A1 = cache["A1"]
    A2 = cache["A2"]
    A3 = cache["A3"]

# Compute the gradient dZ2 of the error with respect to the
# input of the output layer Z2 as the difference between the
# predicted output A2 and the true labels Y.

    W2 = parameters["W2"]
    W3 = parameters["W3"]

# Compute the gradient dZ3 of the error with respect to the
# input of the output layer Z3 as the difference between the
# predicted output A3 and the true labels Y.

    dZ3 = A3 - Y

# Compute the gradient dW3 of the error with respect to the
# weight matrix W2 as the product of dZ3 and the transpose
# of the hidden layer output A2, divided by the number of samples

    dW3 = 1 / m * np.dot(dZ3, A2.T)

# Compute the gradient db2 of the error with respect to the bias
# vector b2 as the sum of dZ2 across samples, divided by m

    db3 = 1 / m * np.sum(dZ3, axis=1, keepdims=True)
    dZ2 = np.dot(W3.T, dZ3) * sigmoid_derivative(A2)
    dW2 = 1 / m * np.dot(dZ2, A1.T)
    db2 = 1 / m * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * sigmoid_derivative(A1)
    dW1 = 1 / m * np.dot(dZ1, X.T)
    db1 = 1 / m * np.sum(dZ1, axis=1, keepdims=True)

# Create a dictionary called grads to store the computed
# gradients dW1, db1, dW2, and db2.

    grads = {
        "dW1": dW1,
        "db1": db1,
        "dW2": dW2,
        "db2": db2,
        "dW3": dW3,
        "db3": db3
    }

# The function returns the grads dictionary containing the
# gradients of the parameters with respect to the error.
# These gradients are used to update the parameters during
# the training process, helping the neural network learn
# from the input data and minimize the error in its predictions

    return grads

# The update_parameters function updates the parameters
# (weights and biases) of the neural network using the
# gradients computed during the backward propagation
# step. The updates are performed using gradient descent
# optimization. The function does the following:
# Updates the weights and biases using the gradients and a learning rate

def update_parameters(parameters, grads, learning_rate):

# Retrieve the current values of the weight matrices W1, W2
# and W3, and the bias vectors b1, b2 and b3 from the parameters dictionary.

    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

# Retrieve the gradients of the weight matrices dW1, dw2 and dW3,
# and the bias vectors db1, db2 and db3 from the grads dictionary.

    dW1 = grads["dW1"]
    db1 = grads["db1"]
    dW2 = grads["dW2"]
    db2 = grads["db2"]
    dW3 = grads["dW3"]
    db3 = grads["db3"]

# Update the weight matrix W1 by subtracting the product
# of the learning rate learning_rate and the gradient dW1

    W1 -= learning_rate * dW1

# Update the bias vector b1 by subtracting the product
# of the learning rate learning_rate and the gradient db1.
```

```
    b1 -= learning_rate * db1
    W2 -= learning_rate * dW2
    b2 -= learning_rate * db2
    W3 -= learning_rate * dW3
    b3 -= learning_rate * db3

# the function returns the updated_parameters dictionary
# containing the updated values of the neural network's parameters.
# These updated parameters will be used in the next iteration
# of forward and backward propagation during the training process,
# allowing the network to learn from the input data
# and improve its predictions.

    updated_parameters = {
        "W1": W1,
        "b1": b1,
        "W2": W2,
        "b2": b2,
        "W3": W3,
        "b3": b3
    }

    return updated_parameters

# The train_mlp function trains a multi-layer perceptron (MLP)
# neural network with one hidden layer on input data X and output labels Y.
# Trains the MLP using the given data, input size, hidden layers'
# size, output size, number of iterations, and learning rate.

def train_mlp(X, Y, input_size, hidden_size1, hidden_size2, output_size, num_iterations,
learning_rate):

    parameters = initialize_parameters(input_size, hidden_size1,

    hidden_size2, output_size)

    for i in range(num_iterations):
        # Forward propagation
        A3, cache = forward_propagation(X, parameters)

        # Compute cost
        cost = -1 / X.shape[1] * np.sum(Y * np.log(A3) + (1 - Y) * np.log(1 - A3))

        # Backward propagation
        grads = backward_propagation(X, Y, cache, parameters)

        # Update parameters
        parameters = update_parameters(parameters, grads, learning_rate)

    return parameters
```

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

# CSV file with 500,000 rows
input_file = 'airlines_delay.csv'
df = pd.read_csv(input_file)

# Selecting 10,000 random rows
df_sample = df.sample(n=15000, random_state=1)
output_file = 'output_data.csv'
df_sample.to_csv(output_file, index=False)



# Load data
data = pd.read_csv("output_data.csv")  # Replace with the path to your dataset

# Preprocess data
cat_cols = ["Airline", "AirportFrom", "AirportTo"]
cont_cols = ["Time", "Length"]
target_col = "Class"

# One-hot encode categorical variables
encoder = OneHotEncoder(sparse=False)
cat_data = encoder.fit_transform(data[cat_cols])

# Normalize continuous variables
scaler = MinMaxScaler()
cont_data = scaler.fit_transform(data[cont_cols])

# Combine preprocessed data
X = np.hstack([cat_data, cont_data, data["DayOfWeek"].values.reshape(-1, 1)]).T
Y = data[target_col].values.reshape(1, -1)

# Train-test split
X_train, X_test, Y_train, Y_test =
train_test_split(X.T, Y.T, test_size=0.2, random_state=1)
# print("X_train, X_test, Y_train, Y_test \n", X_train, X_test, Y_train, Y_test)
X_train, X_test, Y_train, Y_test = X_train.T, X_test.T, Y_train.T, Y_test.T
# print("X_train, X_test, Y_train, Y_test \n", X_train, X_test, Y_train, Y_test)

# Model parameters
input_size = X.shape[0]
#  hidden layer size is often problem-dependent and should be determined
# through experimentation or validation.
hidden_size1 = 32
hidden_size2 = 32
output_size = 1
num_iterations = 500
learning_rate = 0.01
```

```
# get the start time
st = time.time()

# Train the MLP
parameters = train_mlp(X_train, Y_train, input_size, hidden_size1,  hidden_size2,
output_size, num_iterations, learning_rate)

# Make predictions
predictions_train, _ = forward_propagation(X_train, parameters)
predictions_test, _ = forward_propagation(X_test, parameters)


# get the end time
et = time.time()

# Calculate accuracy
accuracy_train = np.mean(np.round(predictions_train) == Y_train)
accuracy_test = np.mean(np.round(predictions_test) == Y_test)

print("Training accuracy:", accuracy_train)
print("Test accuracy:", accuracy_test)


# Calculate the correlation matrix
correlation_matrix = data.corr()

# Plot the correlation heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, cmap="coolwarm")
plt.title("Feature Correlation Heatmap")
plt.show()

print('Execution time:', elapsed_time, 'seconds')
```