Flight Cancellation Predictor

Using Logistic Regression

Courses assessment project for

Algorithms for massive datasets

Statistical Methods for Machine Learning

Professors

Dario Malchiodi

Nicolò Cesa-Bianchi

By

Ali Rafiei

University of Milan

September 2022

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

Abstract

In this paper, we will see how to build a predictor for flight cancellations based on the Logistic Regression machine learning algorithm. First, we will review the theory part of Logistic Regression and then step-by-step implement the project by using python codes.

1- Parts of the dataset which have been considered

The main dataset on the Kaggle website includes 10 .csv files and we will work with all of them.

1-1- Which columns?

Among columns we will work with these columns:

FL_DATE: This feature is used to mention the date of the flight. This feature is important because the probability of flight cancellation could be high or low on snowy or sunny days.

OP_CARRIER: This feature is related to the airline's company. Some companies are well-organized and more experienced to handle probable problems of flight cancellation and on the other hand, some others are new and cannot be successful to do flights regularly.

OP_CARRIER_FL_NUM: If I understood it correctly the flight number includes some information about each flight and due to they are used for more than one single flight so can be helpful to understand which kind of flight numbers are more probable to reach the destination

ORIGIN: Based on the location of the airport it is possible we have some well-located airports which are not located in snowy and rainy locations of the country so for this reason, the origin airport and destination airport are important features for us.

DEST: The reason why we are working with this feature was mentioned in the ORIGIN section.

CRS_DEP_TIME: It tells us the official time of start for the flight. By considering that some flights are at night, managing the taking of for start or landing in ending the flight could be more difficult in comparison to sunny hours of the day.

CRS_ARR_TIME: This feature tells us the official time of arriving to the destination. The reason why this feature is important for us was mentioned in the CRS_DEP_TIME section.

CANCELLED: The target column would be this feature. The value 0 means the flight was done successfully and the value 1 means the flight was cancelled.

CRS_ELAPSED_TIME: This feature is related to the official duration of the flight. Long and short flights are also interesting features to consider for our predictor.

DISTANCE: Like the CRS_ELAPSED_TIME feature, distance is also interesting for our predictor, we can check the effect of close and far destinations on flight cancellation probability.

1-2- Which rows?

The number of successful flights is really higher than cancelled flights and it means the weight of successfully done flights would be higher in our predictor and it will predict most of the new samples as done by mistake.

To overcome this issue in each dataset for example 2016.csv first we will find the number of cancelled flights. In this case, the result is 65,861 flights. Now we choose 65,000 successfully flights and 65,000 cancelled flights. We do it also for other .csv files and the result would be 1,900,000 flights which includes 950,000 successfully done flights and 950,000 cancelled flights.

Thus, our dataset will have 10 columns or features with 1,900,000 rows or samples.

2- Data organization

We will use the Google Colab environment for our implementation. Dataset will download directly from the Kaggle website into the Colab storage.

Data are separated into 10 .csv files and for more simplicity, we will merge all data in one dataframe to work easier with data.

After merging data and before starting the training we need to normalize our data so again we will create a new dataframe which includes all normalized data that they are ready for training.

The process of creating this dataframe will be explained in the next section.

3- Applied pre-processing techniques

3-1- First round

There exist some features like flight_date which is not calculable for us because of not being numerical value or official_start and official_end does not have suitable structure for processing, so in the first round we try to convert the type of our data to readable values by machine.

3-1-1- Rename columns

first of all to make the dataset more readable, we change the name of columns to easier titles.

3-1-2- Balance dataset

If we consider the 2009.csv dataset we will see the number of successful flights is 6,342,300 and cancelled flights are 87,038. It means 98% of samples are dedicated to the class of successful flights and only 2% for cancelled ones.

In these cases, the learning algorithms are really prone to predict all new samples which are for prediction as successful flights.

Thus for each .csv file, we select an equal number of each class, for example, for 2009.csv it would be 85,000 samples for class 0 and 85,000 samples for class 1.

3-1-3- Shuffling and Reindexing

After balancing datasets we need to shuffle the dataset to avoid having the same classes samples which are located exactly one after each other in the dataset. It helps machine learning algorithms to have access to well-distributed data and reindexing is only for being more readable.

3-1-4- Feature selection

Like we mentioned before we need to work with more valuable features that help us to predict samples more accurate and easier.

3-1-5- Data converting

3-1-5-1- flight_date

Pre-processing for flight_date feature is needed. We know there exist some days during the year which are not suitable for flights like rainy and snowy days. In this part, we considered each date as the first or the second part of each month. For example, when the date is 2016-1-14 it means it is located in the first 15 days of January so its label would be 1 or for 2016-1-23 we will have label 2 because it is located in the second 15 days of January. It means we will have 2 sections for each month and overall 24 parts. For example, 23 means the first 15 days of December.

3-1-5-2- official_start and official_end

Pre-processing for official_start and official_end is also needed. We know some hours of the day have some difficulties for pilots for landing and taking off like the flights at night. Thus, we select the hour part of each flight. It means we will have 24 unique values for this part.

3-1-5-3- official_duration

official_duration is based on the time duration between two airports. It is shown as minutes and for working easier with this feature we change it to an hour.

3-1-5-4- distance

Distances are very variant, to avoid complexity we divide the distance of each sample to a multiplier of 100 kilometres.

3-2- Second round

In the last section, we prepared our data more readable and understandable to machines. For example, flight_date had the values like 2016-1-14 which was not a numeric value and we changed the values od this column to numbers from 1 to 24 and each number shows the first or the second 15 days of each month. But it is still not clear for machines, for example, what is the meaning of the number 15? It is more probable to cancel or was done? In the second round of pre-processing part, we try to solve this problem.

3-2-1- flight_date

in flight_date we have the values from 1 to 24. So the machine is trying to understand how much these numbers are effective on the result. Here our approach we consider the percentage of cancelled flights every 24 parts of the year. For example, in part number 2 which means the second 15 days of January we have 10 flights, 8 of them were done successfully and 2 of them were cancelled. So the percentage of cancellations for part 2 of the year is 20%. Thus instead of giving the machine the raw value of each date, we will give it the probability of the date's cancellation.

3-2-2- Other columns

for the other 8 features we also do the same thing. We calculate the percentage of cancellation and we will put this calculated percentage instead of the original value. For example for airline company 'OH', we have 30916 flights which 22180 of them are cancelled and 8736 of them are done successfully and it means the percentage of cancellations for this airline would be about 72%.

3-3- Standardization

Standardization is also needed. Because the range of the values for flight_date or official_start is far from the values for official_duration and distance. If we do not have the standardization the weight of the features with greater values would be high and as a result, the sensitivity of the predictor would be low on features with lower values.

4- Considered algorithms and their implementations.

We want to solve this classification problem by using the Logistic Regression algorithm.

4-1- Linear Regression vs Logistic Regression

4-1-1- Linear Regression

In Linear Regression we try to find the best function among a set of functions that we call it **Hypothesis set**. Inside the Hypothesis set there is a function which is really close to the Unknown Target Function. **Unknown Target Function** is the best function that supports all correct answers for each data. In the real world it is almost impossible to fine Unknown Target Function exactly because the concept of the Noise is always a part of our problems and it is not possible to model the noises but we try to find that function which is accurate as much as possible to Unknown Target Function.

Hypothesis Set → $h_\theta(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d$

- $h_\theta(x)$: predictor (model)
- θ_0 : shift along the axis Y (bias)
- θ_1: weight (slope)
- x: training/test data
- d: $the\ number\ of\ features$
- $task$: $finding\ \theta_0\ and\ \theta_1\ and\ \ldots$

To find the best function (model) form Hypothesis set we need to find the error for each function (model). The cost function would be Mean Square Error which is:

$$J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$$

It is almost impossible to test all possible weights and find MSE. So we need a strategy to find the weights in a smarter way.

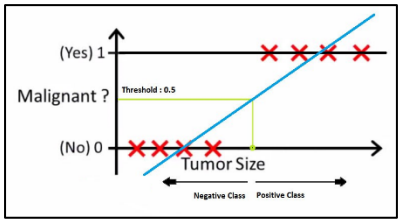
The best solution would be using **Gradient Descent**. If our cost function be a convex function by using Gradient Descent we can find the best weights which give us the minimum value of error. In this formula the θ_j is the weight of one of our features.

$$\left.\begin{aligned} \theta_j &:= \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1) \\ J(\theta_0,\theta_1) &= \frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2 \end{aligned}\right\} \rightarrow \theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}\left(\frac{1}{2m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})^2\right.$$

$\frac{\partial J}{\partial \theta_0} = 0$	$\theta_0 = \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})$
$\frac{\partial J}{\partial \theta_1} = 0$	$\theta_1 = \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)}).x^{(i)}$

4-1-2- Logistic Regression

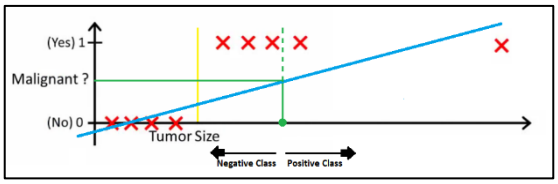
Logistic Regression is a supervised learning algorithm. Although the word of Regression is a part of this algorithm but is not used for solving Regression problems like what we saw in Linear Regression and it would be used for solving classification problems. In theory some basic parts are similar to Linear Regression and for this reason first we studied about Linear Regression to start understanding Logistic Regression better and easier.



Based on Linear Regression algorithm we can find a line to divide or samples to malignant and benign.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 \rightarrow \begin{cases} h_\theta(x) \geq 0.5 \rightarrow y = 1 \\ h_\theta(x) \leq 0.5 \rightarrow y = 0 \end{cases}$$

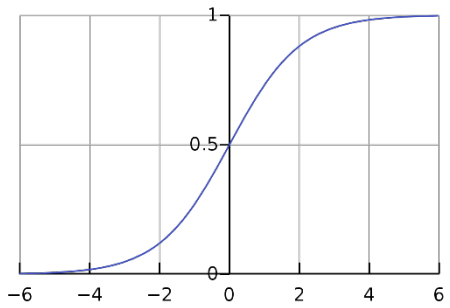
But by adding a new data to our training dataset the equation of the line will be changed and as a result we will have some wrong predictions so it means in this case Linear Regression and drawing a simple line is not a powerful way to reach to a strong classifier.



Linear Regression vs Logistic Regression

For simplicity we can consider the Logistic Regression as the Linear Regression which has a threshold. In Linear Regression the prediction function was $h_\theta(x) = \theta^T x$, and by applying a Logistic function (Sigmoid function) on it we will have the Logistic Regression like $h_\theta(x) = g(\theta^T x)$.c

Image source: https://en.wikipedia.org/wiki/Sigmoid_function#/media/File:Logistic-curve.svg



$$g(z) = \frac{1}{1+e^{-z}}$$

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}} \quad , \quad 0 < h_\theta(x) < 1$$

$$\begin{cases} if\ h_\theta(x) \geq 0.5 : y = 1 \\ if\ h_\theta(x) < 0.5 : y = 0 \end{cases} \rightarrow \begin{cases} if\ \theta^T x \geq 0.5 : y = 1 \\ if\ \theta^T x < 0.5 : y = 0 \end{cases}$$

$$g: logistic\ function \rightarrow sigmoid$$

Now the task is finding the weights (θ). By minimizing this cost function (cross entropy) we will find them but first, we will see some examples to understand better the topic and we will back and study the cost function in more details.

$$Cost\ function \rightarrow \ J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right]$$

Cost function:

In Linear regression the cost function was $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$, but the function of Linear Regression was $h_\theta(x)$, but in Logistic Regression the function is $g(\theta^T x)$, and if we want to use this function in this cost function the result will have a function which is not convex. We know that to find global minimum we need a convex cost function so we will change the cost function.

$h_\theta(x)$ in linear regression is changed to $g(\theta^T x)$ and as a result $J(\theta)$ or cost function for it is not a convex function so I need to use another cost function to avoid getting stuck in local minimum.

$$Cost(h_\theta(x), y) = \begin{cases} -\log\big(h_\theta(x)\big) : y = 1 \\ -\log\big(1 - h_\theta(x)\big) : y = 0 \end{cases}$$

We can write above function in one single line easily like this:

$$Cost(h_\theta(x), y) = -ylog\big(h_\theta(x)\big) - (1-y)\log\big(1 - h_\theta(x)\big)$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big[-y^{(i)}\log\Big(h_\theta\big(x^{(i)}\big)\Big) - \big(1-y^{(i)}\big)\log\Big(1 - h_\theta\big(x^{(i)}\big)\Big)\Big]$$

- $h_\theta(x)$: $predicted\ target$
- y: $Actual\ target$

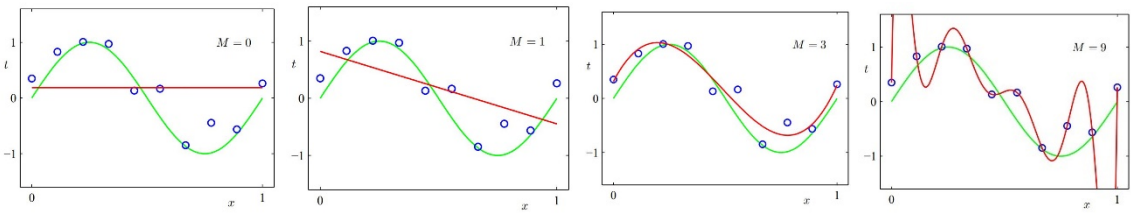
Now this new cost function is convex so we can use **gradient descent** to find weights (θ parameters) without getting stuck in local minimum.

$$\left.\begin{array}{l} J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\Big[-y^{(i)}\log\Big(h_\theta\big(x^{(i)}\big)\Big) - \big(1-y^{(i)}\big)\log\Big(1 - h_\theta\big(x^{(i)}\big)\Big)\Big] \\ \theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \end{array}\right\} \rightarrow \frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}).x_j^{(i)})$$

4-2- Regularization

4-2-1- Regularized Linear Regression

Image source: https://www.cs.cmu.edu/~atalwalk/teaching/winter17/cs260/lectures/lec09.pdf



	$M=0$	$M=1$	$M=3$	$M=9$
w_0	0.19	0.82	0.31	0.35
w_1		-1.27	7.99	232.37
w_2			-25.43	-5321.83
w_3			17.37	48568.31
w_4				-231639.30
w_5				640042.26
w_6				-1061800.52
w_7				1042400.18
w_8				-557682.99
w_9				125201.43

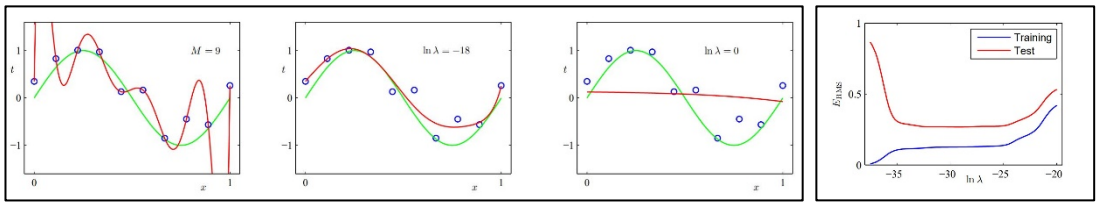
Overfitting problem is related to predict training data with high accuracy and predict test data with low accuracy.

One reason that leads to have overfitted model is increasing the parameters of model. As it is shown when the line equation is degree 1 we have only one weight with the value of 0.19 but by increasing the degree of equation the number and value of parameters are starting to increase.

To avoid having this problem we will use regression Regularization. The cost function was $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2$, and now we need to add $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$, to it. The result would be:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Image source: https://haipeng-luo.net/courses/CSCI567/2021_fall/lec2.pdf



	$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
w_0	0.35	0.35	0.13
w_1	232.37	4.74	-0.05
w_2	-5321.83	-0.77	-0.06
w_3	48568.31	-31.97	-0.06
w_4	-231639.30	-3.89	-0.03
w_5	640042.26	55.28	-0.02
w_6	-1061800.52	41.32	-0.01
w_7	1042400.18	-45.95	-0.00
w_8	-557682.99	-91.53	0.00
w_9	125201.43	72.68	0.01

For degree 9 we will have:

$$column\ 1:\ \lambda = 0 \rightarrow \ln\lambda = -\infty \rightarrow \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 = 0$$

$$column\ 2:\ \lambda = e^{-18} \rightarrow \ln\lambda = -18$$

$$column\ 3:\ \lambda = 1 \rightarrow \ln\lambda = 0$$

Gradient descent in Regularized Linear Regression

Before we saw when we use gradient descent for Linear Regression the updating formulas were $\theta_0 = \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)})$, and $\theta_1 = \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(\theta_0 + \theta_1 x^{(i)} - y^{(i)}).x^{(i)}$, and now our cost function is $J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)} - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$, so updating formulas for weight would be:

- $\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}).x^{(i)}$
- $\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}).x_j^{(i)} + \frac{\lambda}{m}\theta_j\right]$

4-2-2- Regularized Logistic Regression

Overfitting problem is related to predict training data with high accuracy and predict test data with low accuracy. One reason that leads to have overfitted model is increasing the parameters of model.

To avoid having this problem we will use Regularized Logistic Regression. The cost function was

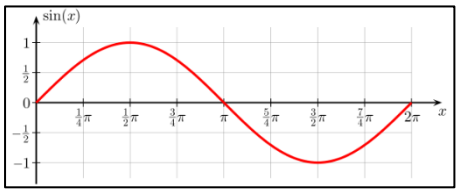
$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log\left(h_\theta(x^{(i)})\right) - (1-y^{(i)})\log\left(1-h_\theta(x^{(i)})\right)\right]$, and $h_\theta(x)$, was $g(\theta^T x)$ which was $\frac{1}{1+e^{-\theta^T x}}$, now we need to add $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$, to it. The result would be:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log\left(h_\theta(x^{(i)})\right) - (1-y^{(i)})\log\left(1-h_\theta(x^{(i)})\right)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

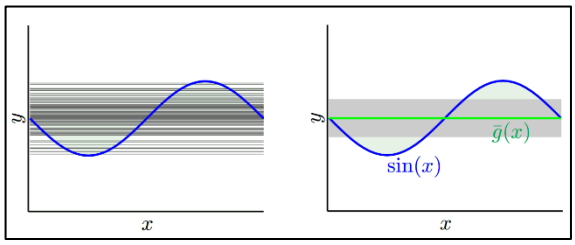
Bias & Variance

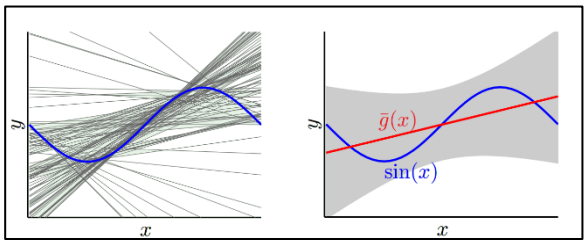
The amount of Error is a summation of **Bias**, **Variance** and **Noise.** To reducing the amount of Error we cannot solve the problem of existence of the noise because modeling the noise is impossible but we can reduce the amount of Bias and Variance.

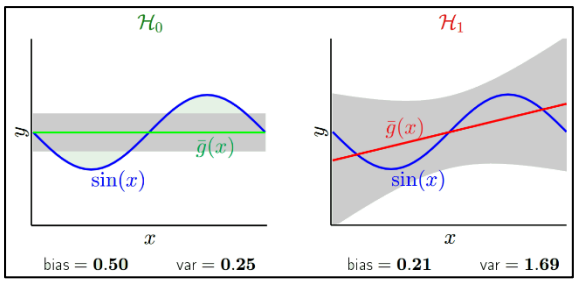
Image source: https://stats.stackexchange.com/questions/4284/intuitive-explanation-of-the-bias-variance-tradeoff



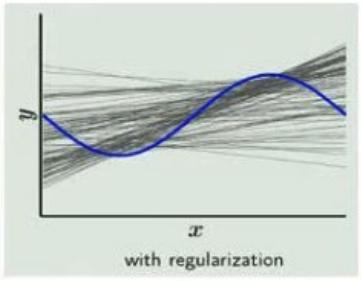
In this Sine function we can work with simple and complex Hypothesis set. The simple one is $H_0: f(x) = b$, and if K times we create training set as a result we will have K number of $f(x) = b$, which are shown by black lines and the mean of all found functions would be $\bar{f}$ which is shown by green line. We have the same strategy for the complex one which is $H_1: f(x) = ax + b$.

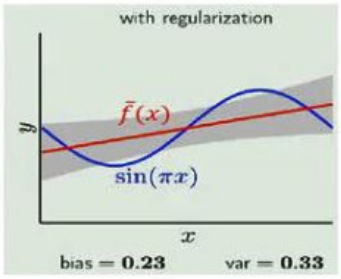






As we can see the amount of (Bias + Variance) in simple Hypothesis set is (0.50 + 0.25) and in complex Hypothesis set is (0.21 + 1.69) so it means by working with simple one which is $H_0: f(x) = b$, we will have lower error.





But when we use regularized term in complex hypothesis set which is $H_1: f(x) = ax + b$, we can the amout of (bias + variance) is (0.23 + 0.33) and it mean amont these three models, the last one is the best.

5- How the proposed solution scales up with data size.

In next section which is related to implementation we will see how we append new samples to our dataset. A high level overview is that we first load 2009.csv file and after that we add other datasets like 2010.csv file to our loaded dataset to improve the process of training the model.

6- Description of the experiments.

6-1- Load data

First we download data from Kaggle website.

```
!pip install opendatasets
import opendatasets as od
# username: alirafiei
# api: 70819616fb3608417bd3826c331e9a31
od.download('https://www.kaggle.com/datasets/yuanyuwendymu/airline-delay-and-cancellation-data-2009-2018')
```

6-2- Create balance dataset

The number of successfully done flights are really higher than cancelled flights and if we train the model without considering this fact, the model will prone to predict the most number of new samples as successfully done. Thus, we balance our dataset by working with the same number for done and cancelled flights.

```
raw_dataFrame = pd.read_csv('/content/dataset/2009.csv')
print('The number of done flights: ', len(raw_dataFrame[raw_dataFrame['CANCELLED'] == 0]))
print('The number of cancelled flights: ', len(raw_dataFrame[raw_dataFrame['CANCELLED'] == 1]))
```

output:
The number of done flights: 6342300
The number of cancelled flights: 87038

```
done_flights = raw_dataFrame[raw_dataFrame['CANCELLED'] == 0].sample(85000, random_state=29)
cancelled_flights = raw_dataFrame[raw_dataFrame['CANCELLED'] == 1].sample(85000, random_state=29)
balanced_data_2009 = done_flights.append([cancelled_flights])
balanced_data_2009.shape
```

Now we have a balanced dataset for 2009.csv dataset and we can save it by this code:

```
balanced_data_2009.to_csv('/content/balanced_dataset/balanced_f09_t09.csv', index=False)
```

To scale up the dataset we can do the same thing for other datasets and append them to 2009 dataset by this code:

```
appended_df = pd.concat([balanced_data_2009, balanced_data_2010])
appended_df.shape
```

After loading and appending all datasets we will have:

```
print('The number of done flights: ', len(appended_df[appended_df['CANCELLED'] == 0]))
print('The number of cancelled flights: ', len(appended_df[appended_df['CANCELLED'] == 1]))
```

output:
The number of done flights: 950000
The number of cancelled flights: 950000

Now this new data are available as balanced_f09_t18.csv, so we load it:

```
raw_dataFrame = pd.read_csv('/content/balanced_dataset/balanced_f09_t18.csv')
```

To read the name of columns easier we can change the name of them. Here I just write one of them and full code is available at the Colab:

```
raw_dataFrame.rename(columns = {
    'FL_DATE':'flight_date',
}, inplace = True)
```

Sometimes during data loading the samples of each class will be placed close to each other and as a result we will not have a good training process. So first we shuffle data and then reindex them.

```
balanced_data = raw_dataFrame.sample(frac=1, random_state=30)
balanced_data = balanced_data.reset_index(drop=True)
```

Now we can try to find the features that give us valuable information to create better predictor which can divide the classes more accurate.

```
balanced_data.count()
```

6-3- Feature selection

Among all features some columns have the minimum number of NaN values. These columns are 'flight_date', 'airline_company', 'flight_number', 'origin_airport', 'destination_airport', 'official_start', 'official_end', 'official_duration', 'distance', 'cancelled'.

6-4- Prepare flight_date column

The first column is flight_date. This column's values are string and not calculable for Logistic Regression algorithm. So we need to convert it into a meaningful value. The concept of date can consider as the day in the year. For example 2017-03-14 is the (Jan:31 + Feb:28 + Mar:14) 73rd day in the year. Another solution which is easier and faster for conversion is considering each day as the first or the second part of the month. The example of 2017-03-14 is located in 5th part of the year which is 15 and 16 days for the first and second half of January and 15 and 14 days for the first and the second half of February and March the 14th is located in the first half of March so the result would be 5th part of the year. In this scaling each year would have 24 parts.
The reason why we separate dates into these 24 parts is related to weather status. The number of flight cancellation is probably higher that sunny days so we need to attention to the date of flight.

To find we have how many NaN values :

```
print('Number of missed data for [flight_date]: ', balanced_data['flight_date'].isna().sum())
```

By using this code we can separate the part of the date that shows the month of flight. Then we divide it into 2 parts. And finally we replace new value into dataframe:

```
flight_date = balanced_data['flight_date']
for i, date in enumerate(flight_date):
  day = int(date[8:10])
  month = int(date[5:7])
  day_in_year = ((month-1) * 2)
  day_in_year += 1 if day <= 15 else 2
  balanced_data.at[i,'flight_date']= day_in_year
```

The columns like 'airline_company', 'origin_airport', 'destination_airport' include string values which are not suitable type for our algorithm, other other hand, all of them are important for us and it is better we try to convert and use them in our training. The reason why they are important features is that some airlines or airports due to not being well-experienced in this economy may have high number of flight cancellation. By using this code we can change the type of airline company from string to integer.

```
balanced_data.airline_company = balanced_data.airline_company.apply(list(balanced_data['airline_company'].unique()).index)
```

We need to attention just indexing the airlines is not enough for algorithm to detect which index helps us to have a successful flight. We will make a more valuable data from this column in the second part of pre-processing.

The columns of 'origin_airport', 'destination_airport' also need to convert to integer but we need to attention use the same index for the same airports. The number of all airports are 373 here we just mention to 3 of them.

```
balanced_data.origin_airport = balanced_data.origin_airport.apply(['EWR', 'ORD', 'SAT'].index)
```

6-5- Prepare official_start, official_end columns

The type of 'official_start', 'official_end', column is numeric but it is not shown in suitable style. For example, 1530.0 mean hour 15 and minute 30 or 105.0 means hour 1 and minute 05. For solving this problem one solution is that we can consider each time as the number of minutes in a day, for example, 105.0 means (1*60)+5 minute number 65 of a day. In this scaling the time would be between 0 and (24*60) 1440. For more simplicity and faster execution we just consider the hour of the time. For example, 105.0 would be 1.

```
official_start = balanced_data['official_start']
for i, time in enumerate(official_start):
    if time == 2400:
        balanced_data.at[i,'official_start'] = 0
    else:
        balanced_data.at[i,'official_start']= int(time/100)
```

6-6- Prepare official_duration column

By using this code we can see the values for the feature of 'official_duration' is variant from 1 to 1865.

```
print(balanced_data['official_duration'].min())
print(balanced_data['official_duration'].max())
```

Here also like official_start and official_end we can consider the minutes of the flight as hours of the flight by this code:

```
official_duration = balanced_data['official_duration']
for i, duration in enumerate(official_duration):
    balanced_data.at[i,'official_duration']= int(duration/60)
```

Something which is different in this column is that it has some NaN values so we need to fill them. A simple way to fill them is that just putting a fixed value like 0 for each NaN cell. Another smart way for this problem is that filling NaN cells with meaningful values. For example the values in official_duration and distance are correlated. So we can group the values of distance column and find the mean of each group and then use the mean value for each group in the right place for official_duration column.

```
balanced_data['official_duration'] = balanced_data.groupby(['distance'])['official_
duration'].transform(lambda x: x.fillna(x.mean()))
```

6-7- Prepare distance column

For distance column we have variant values from 11 to 4983 by running this code:

```
print(balanced_data['distance'].min())
print(balanced_data['distance'].max())
```

To make the processing easier we can consider each 100 km as a group of distances:

```
distance = balanced_data['distance']
for i, distance in enumerate(distance):
    balanced_data.at[i,'distance']= int(distance/100)
```

6-8- The result of pre-processing part 1

After doing the first round of pre-processing we can check the correlation between cancelled column and other columns:

```
correlation = balanced_data.corr()
correlation['cancelled']
```

Output:

flight_date	-0.128044
airline_company	0.029387
origin_airport	0.057012
destination_airport	0.023890
official_start	0.046203
official_end	0.030328
official_duration	-0.110333
distance	-0.138880
cancelled	1.000000

We can see there are some weak correlation between cancelled column and other columns. When we want to start the training the algorithm can understand there some correlation between successfully flights with flight_date column values like 15 and 16 which mention to sunny days in August. But for some other columns which are indexed like airport_company the algorithm does not have any idea the differences between airline number 1 and airline number 154. To solve the problems like this we need to start the second round of pre-processing.

6-9- More pre-processing

For the second part we have the same approach for all columns which is finding the score of each cell. This score shows us the probability of cancellation. For example if the flight_date is 3 or the first part of February will convert to 72 and it means among all flights that they had the flight_date 3 or the first part of February 72% of them were cancelled.

This approach is more useful for the columns which have not any meaningful orders. For example the cancellation for airline number 53 is 20% for airline 97 is 52% and airline 159 is 5%. In this case if we work with the percentage of cancellation the values are more readable for algorithm because they have an order.

First we find the unique values for flight_date which are from 1 to 24. Then for each unique value like 14 (second half of July) we find the rows separately for done and cancelled flights. By doing this it is possible for us to find the percentage of cancellation in the second half of July.

```
done_date_list, cancelled_date_list, date_list = [], [], []
done_flights_count, cancelled_flights_count = 0, 0
for date in df['flight_date'].unique():
    done_flights_count = len(df[(df['flight_date'] == date) & df['cance
lled'].isin([0])])
    done_date_list.append(done_flights_count)
    cancelled_flights_count = len(df[(df['flight_date'] == date) & df['
cancelled'].isin([1])])
    cancelled_date_list.append(cancelled_flights_count)
    date_list.append(date)
```

By using this code we can find the percentage of flights cancellation for each unique values in flight_date column:

```
date_score, date_all_flights = [], []

for i, (done, cancelled) in enumerate(zip(done_date_list, cancelled_dat
e_list)):
    date_score.append(round(cancelled/(done+cancelled)*100))
    date_all_flights.append(cancelled + done)
```

In the first row we can see the number of done, cancelled and all flights are 36849, 66757 and 103606 so the the percentage of cancellation for the first half of January would be 64%. To see the result we use this code.

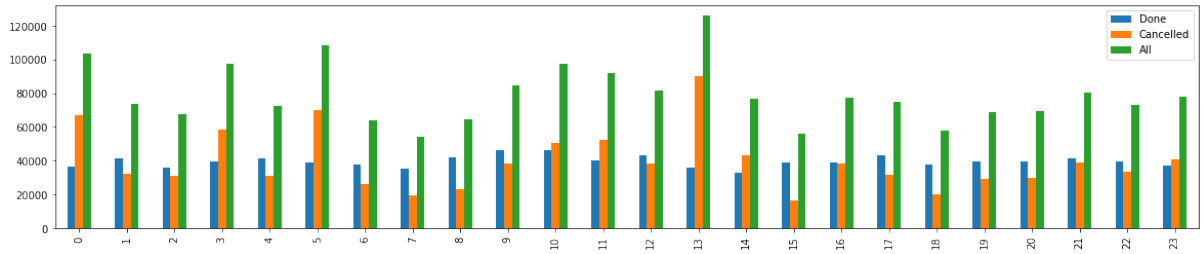
```
columns = [date_list, done_date_list, cancelled_date_list, date_all_fli
ghts, date_score]
date_df = pd.DataFrame(columns, index=['Date', 'Done', 'Cancelled', 'Al
l', 'DateScore']).T
date_df.head()
```

	Date	Done	Cancelled	All	DateScore
0	1	36849	66757	103606	64
1	17	41181	32505	73686	44
2	13	36031	31357	67388	47

3	5	39324	58322	97646	60
4	10	41630	30920	72550	43

In this plot blue, orange and green bars are related to done, cancelled and all flights.

```
date_df.iloc[0:len(df['flight_date'].unique()), 1:4].plot.bar(figsize=(20, 4));
```



For replacing the old values with new ones we can use dictionaries:

```
date_zip = dict(zip(date_df.Date, date_df.DateScore))
df=df.replace({"flight_date": date_zip})
```

For all other features we can do the same thing. As a result when the training algorithm read each row it can find the probability of cancellation for each features. As much as being higher for each column the model would be more sensitive to detect the sample as a flight that prone to be cancelled.

6-10- The result of pre-processing part 2

To see the correlation between cancelled column with other columns:

```
correlation = df.corr()
correlation['cancelled']
```

The left result is related to correlation between cancelled column and other columns after the first round of pre-processing and the left one is for correlations after the second round of pre-processing.

Features	Pre-processing (Part I)	Pre-processing (Part II)
flight_date	-0.128044	0.210020
airline_company	0.029387	0.240945
origin_airport	0.057012	0.238207
destination_airport	0.023890	0.229512
official_start	0.046203	0.069547
official_end	0.030328	0.055731
official_duration	-0.110333	0.113828
distance	-0.138880	0.157421
cancelled	1.000000	1.000000

6-11- Z-Score standardize function

All values for all features are between 0 and 100 so we do not need Z-Score scaling but when our data are not scaled by using this code we can standardize them.

```
normalized_df=(df-df.mean())/df.std()
```

6-12- Logistic Regression from scratch

Like SKLearn libraries first we define a class and step by step we implement each functionality of Logistic Regression.

```
class Logistic_Regression:
    def __init__(self, learning_rate, epochs):
      pass
    def train(self, x_train, y_train):
      pass
    def test(self, x_test):
      pass
```

Inside __init__ method we define some initializations like learning_rate and the number of epochs. These two variables are chosen randomly. Then we implement train methos which is related to understanding the best values for weights and bias based on training set. Finally we can test how our model works well by using test method.

The implementation of __init__ method is:

```
    def __init__(self, learning_rate, epochs):
        self.learning_rate = learning_rate
        self.epochs = epochs
```

Implementation of train method includes different part that the most important of them is updating the weights and the bias. During finding the best values for weights and bias we need to find some parameters like the number of rows in training set or the number of features and initiate the weights and bias by random values.

```
x_train_rows = x_train.shape[0]
x_train_columns = x_train.shape[1]
self.weights = np.zeros(x_train_columns)
self.bias = 0
```

Now based on the number of epochs we try to find the best values for weights and bias. The Logistic Regression is Linear Regression which is affected by sigmoid function. So first we find the Linear Regression formula. The Linear Regression is $h_\theta(x) = \theta_0 + \theta_1 x_1 + \cdots + \theta_d x_d$ or $\theta_0 + \theta^T x$ and its implementation is:

```
linear_regression = np.dot(x_train, self.weights) + self.bias
```

We know in Logistic Regression we are looking for the probability so we apply the sigmoid function on Linear Regression formula which is $(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$ and its implementation is:

```
y_pred = 1 / (1 + np.exp(-1 * linear_regression))
```

Now we need to update the weights and the bias. For doing this we need the derivatives respect to weights and bias for the cost function.

$$\begin{cases} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1-y^{(i)}\right)\log\left(1-h_\theta\left(x^{(i)}\right)\right)\right] \\ \theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \end{cases}$$

For $\frac{\partial}{\partial\theta_j}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}).x_j^{(i)})$ we have:

```
weights_derivative = (1 / x_train_rows) * np.dot(x_train.T, (y_pred - y_train))
```

And for $\frac{\partial}{\partial\theta_0}J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}))$ we have:

```
bias_derivative = (1 / x_train_rows) * np.sum(y_pred - y_train)
```

Now we need to update weights and bias, for weights updating $\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}).x_j^{(i)})$ we have:

```
self.weights -= self.learning_rate * weights_derivative
```

And for bias updating $\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}))$ we have:

```
self.bias -= self.learning_rate * bias_derivative
```

To know how the training method works we can see the log for each 10 epochs by:

```
if (i%10 == 0):
  print('Epoch number: ', i)
  print('weights: ', self.weights)
  print('bias: ', self.bias)
  print('='*50)
```

To test how much the predictor works well we can use test method. The first part is similar to train part we just predict the sample by using found weights and bias from train method:

```
linear_regression = np.dot(x_test, self.weights) + self.bias
y_pred = 1 / (1 + np.exp(-1 * linear_regression))
```

Then based on the output of sigmoid function we divide samples into 2 classes of 0 and 1 based on being greater or less than 0.5 threshold.

```
class_ped = [1 if i > 0.5 else 0 for i in y_pred]
return np.array(class_ped)
```

7- Comment on the experimental results.

7-1- Confusion Matrix

To evaluate how much the results are good or not we can analyze the confusion matrix. It includes:

True Positive: Actually, the flight cancelled and model predicted it as cancelled.

False Negative: Actually, the flight cancelled and model predicted it as done.

False Positive: Actually, the flight was done and model predicted it as cancelled.

True Negative: Actually, the flight was done and model predicted it as done.

```
true_positive=0
true_negative=0
false_positive=0
false_negative=0
for i in range(len(evaluation_df)):
    if evaluation_df.at[i, 'Actual']==1 and evaluation_df.at[i, 'Predicted']==1:
        true_positive += 1
    elif evaluation_df.at[i, 'Actual']==1 and evaluation_df.at[i, 'Predicted']==0:
        false_negative += 1
    elif evaluation_df.at[i, 'Actual']==0 and evaluation_df.at[i, 'Predicted']==1:
        false_positive += 1
    elif evaluation_df.at[i, 'Actual']==0 and evaluation_df.at[i, 'Predicted']==0:
        true_negative += 1
print('TP: ' + str(true_positive) + '\tFN: ' + str(false_negative))
print('FP: ' + str(false_positive) + '\tTN: ' + str(true_negative))
```

7-2- Accuracy

```
accuracy = (true_positive+true_negative) / (true_positive+false_negative+false_posi
tive+true_negative)
print('The accuracy is: ', accuracy)
```

Output:

```
The accuracy is:  0.63559
```

7-3- Precision

```
precision = true_positive / (true_positive+false_positive)
print('The precision is: ', precision)
```

Output:

```
The precision is:  0.62760658345802
```

7-4- Recall (sensitivity or true positive rate)

```
recall = true_positive / (true_positive+false_negative)
print('The recall is: ', recall)
```

Output:

```
The recall is:  0.6638552768651335
```

7-5- Specificity (true negative rate)

```
specificity = true_negative / (true_negative+false_positive)
print('The specificity is: ', specificity)
```

Output:

```
The specificity is:  0.6074195351808961
```

7-6- F1-Score

```
f1_score = (2*true_positive) / (2*true_positive + false_positive + false_negative)
print('The f1_score is: ', f1_score)
```

Output:

```
The f1_score is:  0.6452222168135131
```

7-7- Execution time

The code is implemented in Google colab. There are 3 checkpoints in the code.

The time for downloading all datasets from 2009 to 2018 is: 00:09:05

The time for pre-processing dataset is: 00:04:16

The time for training (1000 epochs) and prediction of 1,900,000 samples is: 00:03:17

7-8- Suggestions for improvement

For example, the column of flight_date is converted to the first and the second half of each month so all data would be between 1 and 24. But if we consider the date as the number of the day in the year as a result all data would be between 1 and 365. It means we can find the sensitive days better.

The same strategy would be useful for time columns like official_start and official_end. We considered the time as the hour of activity which is between 1 and 24. But if we consider it as the minute if the day it would be between 1 and 1440 and as a result we can critical time more accurate.

Also distance and official_duration are divided into 100 kilometers and 60 minutes intervals. By reducing these intervals we will have more group of data and the result for training is more precise and the result of prediction would be more accurate.