Exploring Data Science in Supply Chain Project Report

Rishabh Kaushick
College of Engineering,
Northeastern University
Toronto, ON, Canada
kaushick.r@northeastern.edu

Abstract

This project is adding onto the mid-term project. In the mid-term project, the exploratory data analysis was performed on the dataset. Furthermore, the machine learning problem was defined. For the mid-term project two Decision Tree models were trained with highest test accuracy of 52%.

For the Final Project, I worked extensively on data preprocessing. Following this, I re-ran the decision tree model on this new data and noticed an improvement of nearly 20%. Next, I trained a random forest model on the same data which yielded an accuracy of around 93%, which was more than 40% improvement to the first model. Furthermore, this model was beaten when the top 15 features of this model was used to train a second random forest model. This model performed with an accuracy of 97.54 %. I also explored hyper-parameter tuning using Randomized Search CV, and the third and final random forest model performed best at 97.55% training accuracy and 97.58% testing accuracy.

Dataset

Dataset	Name	Dataset Characteristics	Attribute Characteristics	Associated Task	Number of Instances	Number of Attributes
1	DataCo Supply Chain Management for Big Data Analysis	Multivariate	Real	Classification	180,519	53

Table 1: Understanding the type of dataset.

The Kaggle Data set contains two csv files. Figure 1 below shows all the columns present in the structured csv and the un-structured csv files. The structured csv document contains a total of 53 columns, and 180,519 rows which results in 9,567,507 total data points. On the other hand, the un-structured CSV document contains 8 columns and 469,977 rows therefore resulting in 3,759,816 total data points. Although the unstructured csv has a greater number of rows, it still has lesser number of data points compared to the structured csv data. Since we cannot incorporate the data from the unstructured csv data to the structured csv data, we will not be using it for the rest of the project.

Why I Find the Dataset Interesting

It is very interesting to know that many of the products which we purchase with a single click on Amazon (or other e-commerce platforms) started their journey a few months before we place the

order. Nowadays most products are manufactured in Southeast Asian countries like Vietnam and are usually placed on a ship for a month-long journey across the Pacific Ocean to reach the North American continent. Therefore platforms like Amazon and Walmart must have to plan much in advance if they want to deliver products to their customers on time.

Data Characteristics

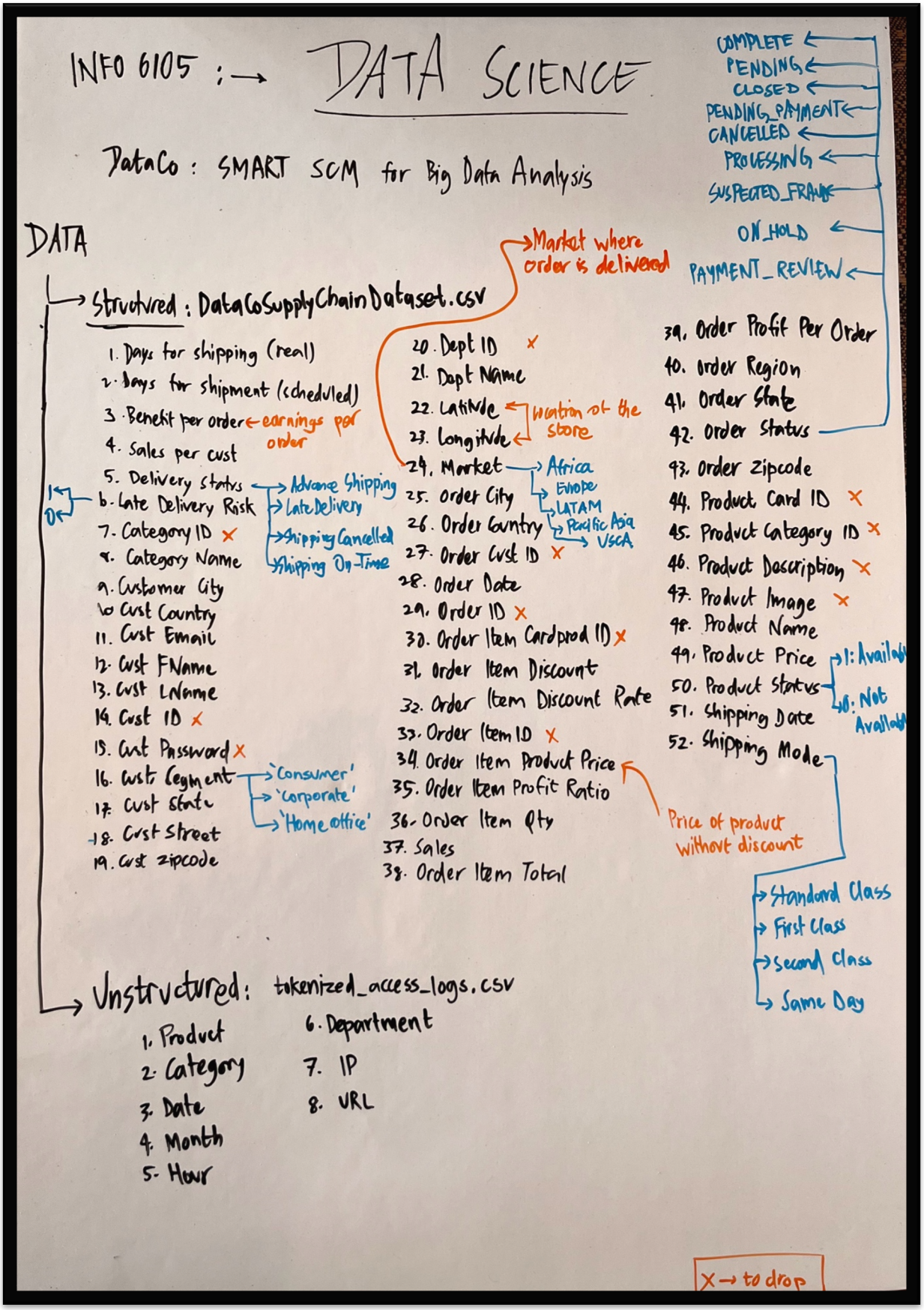


Figure 1: List of the columns present in the structured and unstructured dataset provided in Kaggle.

Figure 1 above lists all the columns in the dataset. The explanation of some columns are mentioned in orange and each of the columns containing categorical data have been mentioned in blue. The columns marked with '×' are columns which do not hold any importance while training the model. Some of these columns include primary keys like 'Customer ID', 'Order Item ID' and 'Product Card ID'. Other columns like 'Password' or 'Product Image' are also do not necessarily add any value for the machine learning classifier. Therefore, we can safely drop these columns before the model.

Understanding & Visualizing the Data

Types of Data

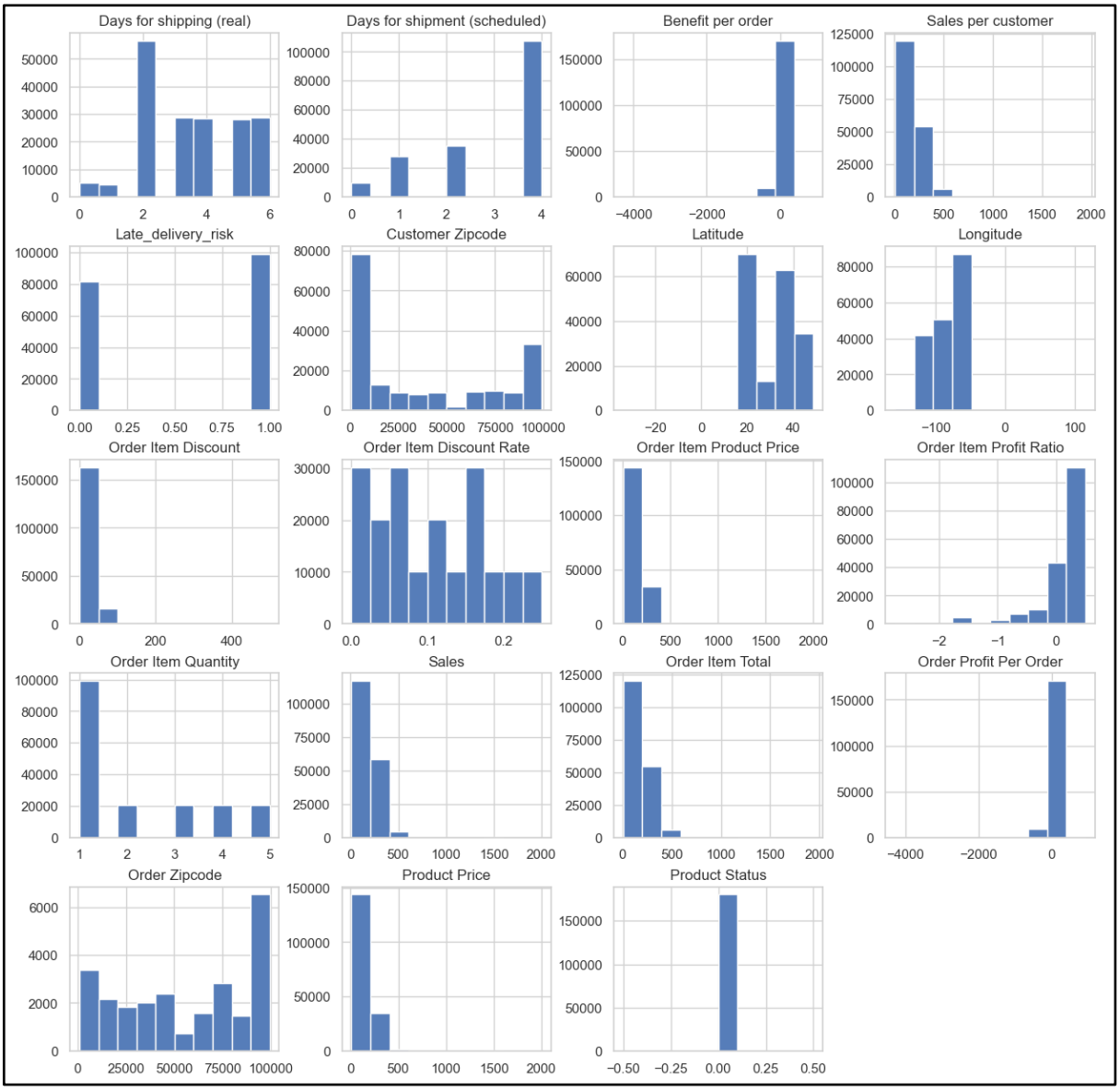


Figure 2: Distribution of data in relevant numeric columns

Delivery Status & Late Delivery Risk

Based on Figure 3 below, we can see that most of the orders have the delivery status as 'Late delivery'.

Figure 3: Types of Delivery Status

There is another column which has late delivery risk as shown below in Figure 4. There are 98,977 values with late delivery risk, and 81,542 values that do not have a late delivery risk.

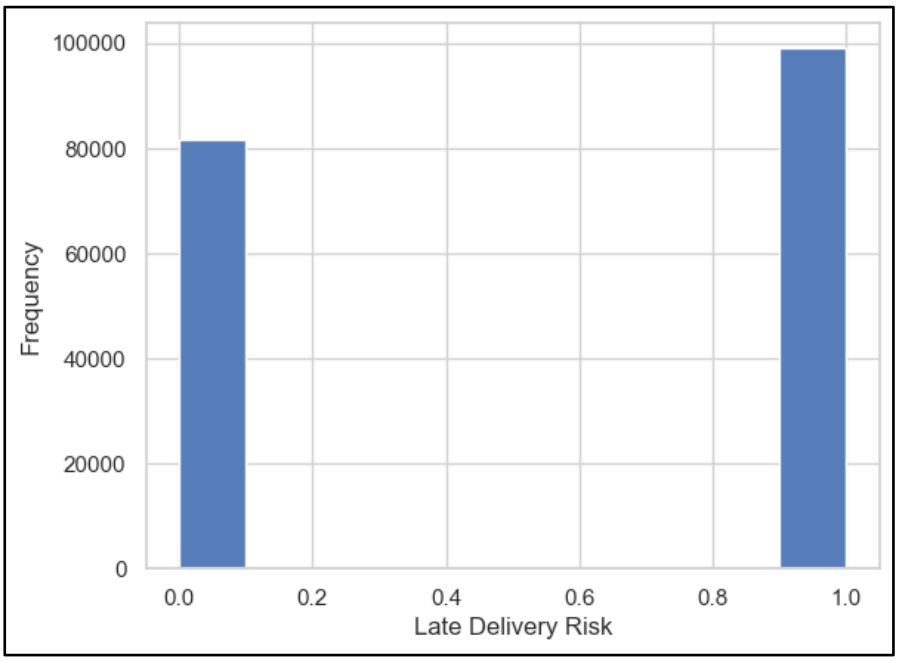


Figure 4: Late delivery risk (0 – no risk, 1 – high risk)

I tried to look deeper into whether the delivery status has any impact on the late delivery risk, and therefore plotted them together in one graph (Figure 5 below) – with x axis as the delivery status, y axis as the frequency and a color denoting if there is a high or low delivery risk.

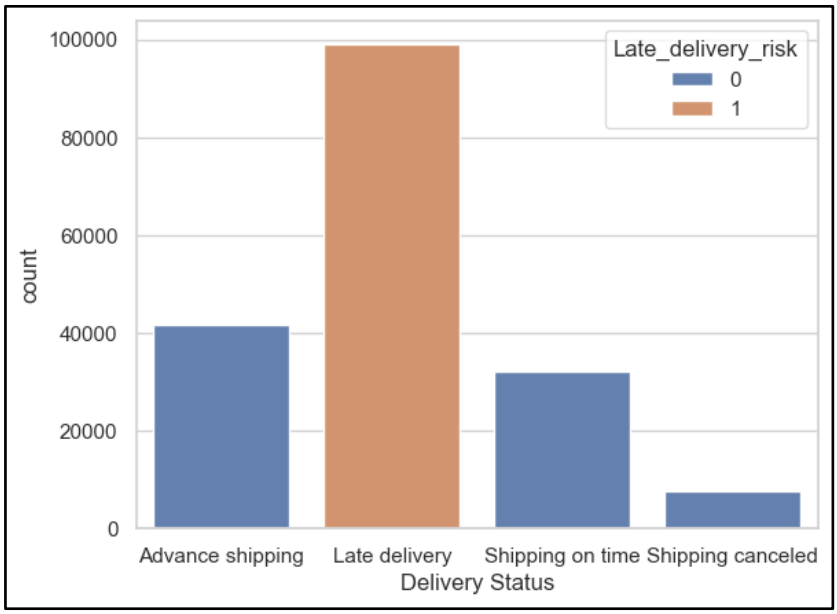


Figure 5: Delivery status and late delivery risk plotted together.

The above graph is counterintuitive. We can see that there is a late delivery risk only on those data which have delivery status as 'Late delivery', and all other delivery statuses do not have a late delivery risk. We can infer from this figure that the delivery status and the late delivery risk are two columns which are talking about the same data. Hence, we can drop one of these columns before training the model.

Another interesting visualization is the frequency of Late Delivery Risk color-coded based on each different region. The graph in Figure 6 shows that the late delivery risk does not seem to be region specific.

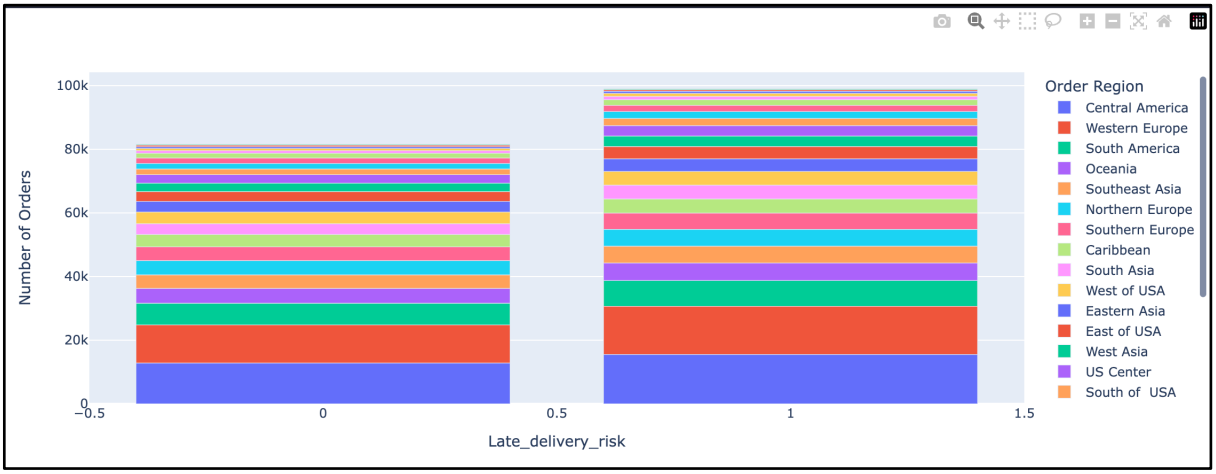


Figure 6: Late Delivery Risk vs Order Region

Store Locations on World Map

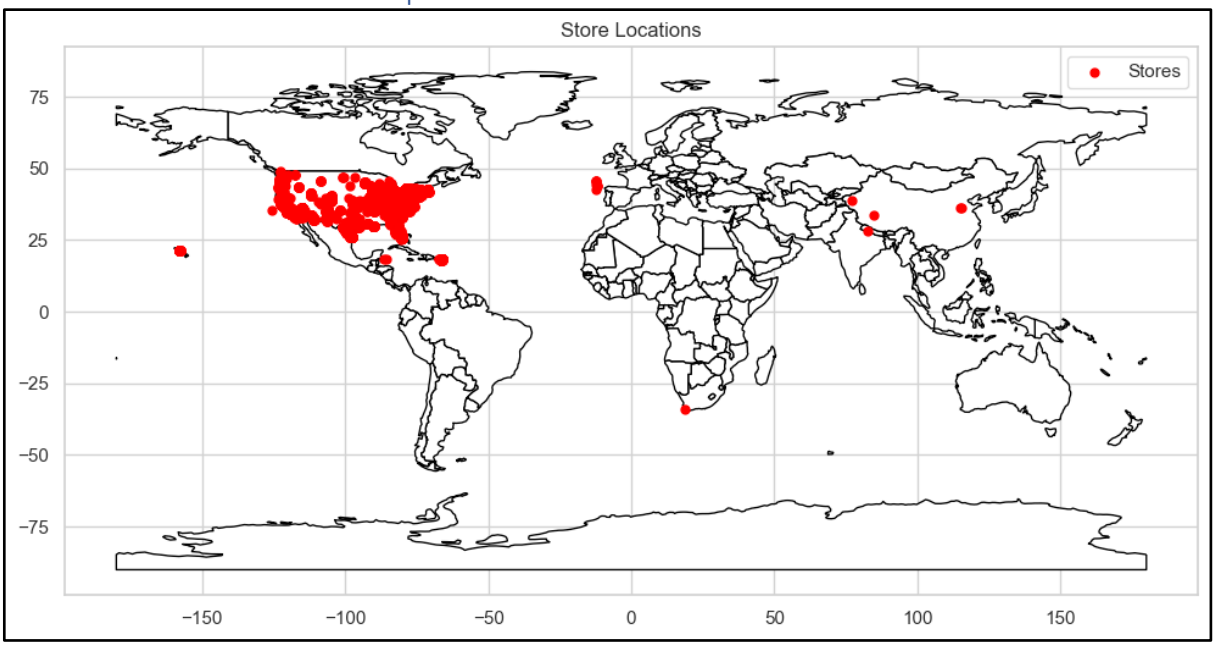


Figure 7: Store locations on the world map.

Based on the latitude and longitude values provided for the store locations we can use geopandas library to visualize the location on the world map as shown in Figure 7. From this visualization, we can see that a vast majority of the stores are present in United States of America. There are a few stores South Africa, China, India and perhaps Portugal/ Spain, and no stores in Canada.

Defining The Problem

Based on this dataset, I have analyzed that there are multiple problems for which we can devise a machine learning algorithm:

1. Predicting the risk of late delivery: Classification problem
2. Predicting the Sales: Regression problem
3. Order suspected fraud: Classification problem.

Out of the above three issues, I decided to work on the first issue of predicting late delivery risk because of the following reasons:

- Unique issue – No one in the Kaggle has worked on predicting the risk of late delivery. There have been many people who have worked to analyze the suspected fraud orders.
- Enough data – there are enough data points for late delivery and other delivery statuses. This will ensure that the model has enough ways to learn for both the positive and negative scenarios. For suspected fraud orders, there is much lesser data for positive and negative scenarios.
- Business Use case – Predicting the risk of late delivery can help the business to understand which products usually are delayed and understand the factors behind why certain products are delivered late.

Splitting & Training the Dataset

Figure 8: Splitting data into train & test.

We have split the dataset into 80% training and 20% testing sets. There are a total of 144,416 records in the train-set and 36,103 records in the test-set.

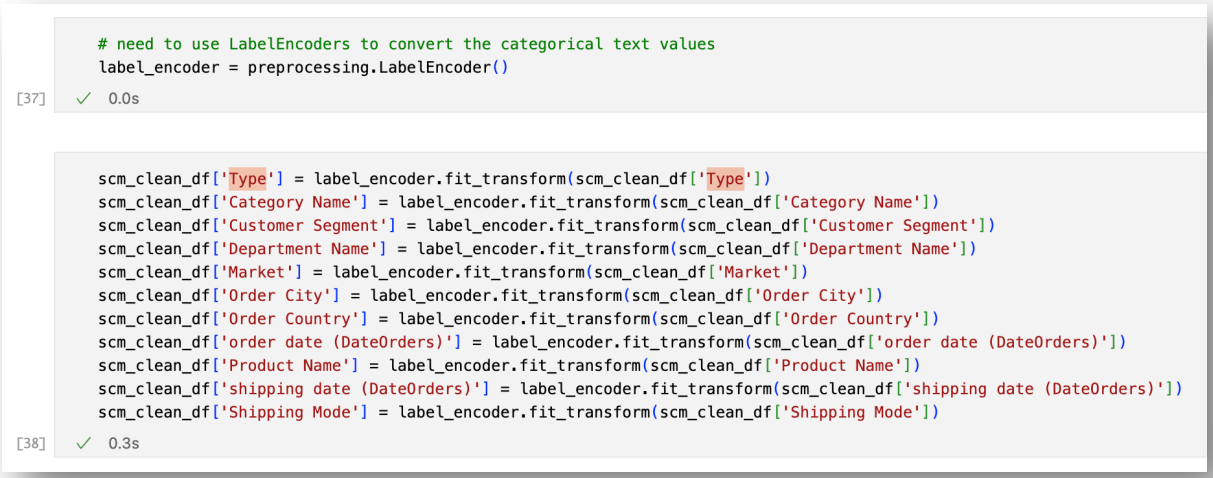
Categorical Data Processing

We must convert the categorical data which are in text fields, into numeric data such that the machine learning models will be able to understand them. I have used the concept of Label Encoding to convert such data into numbers.

The columns with categorical data are as follows:

1. Type

2. Category Name
3. Customer Segment
4. Department Name
5. Market
6. Order City
7. Order Country
8. Order Date (DateOrders)
9. Product Name
10. Shipping Date (DateOrders)
11. Shipping Mode



```
# need to use LabelEncoders to convert the categorical text values
label_encoder = preprocessing.LabelEncoder()
```

[37] ✓ 0.0s

```
scm_clean_df['Type'] = label_encoder.fit_transform(scm_clean_df['Type'])
scm_clean_df['Category Name'] = label_encoder.fit_transform(scm_clean_df['Category Name'])
scm_clean_df['Customer Segment'] = label_encoder.fit_transform(scm_clean_df['Customer Segment'])
scm_clean_df['Department Name'] = label_encoder.fit_transform(scm_clean_df['Department Name'])
scm_clean_df['Market'] = label_encoder.fit_transform(scm_clean_df['Market'])
scm_clean_df['Order City'] = label_encoder.fit_transform(scm_clean_df['Order City'])
scm_clean_df['Order Country'] = label_encoder.fit_transform(scm_clean_df['Order Country'])
scm_clean_df['order date (DateOrders)'] = label_encoder.fit_transform(scm_clean_df['order date (DateOrders)'])
scm_clean_df['Product Name'] = label_encoder.fit_transform(scm_clean_df['Product Name'])
scm_clean_df['shipping date (DateOrders)'] = label_encoder.fit_transform(scm_clean_df['shipping date (DateOrders)'])
scm_clean_df['Shipping Mode'] = label_encoder.fit_transform(scm_clean_df['Shipping Mode'])
```

[38] ✓ 0.3s

Figure 9: Label Encoding.

Preparing the Model

Selection Of the Model – Decision Trees, Random Forest & XG-Boost

In this dataset, I explored how decision trees specifically and Random Forest can be useful to make predictions. I chose decision trees because of their white box nature. Once the model has been trained on the data, we can visualize the decision tree which was created to understand why the machine would predict a certain value.

Furthermore, once we have a random forest model, we can go one step further to understand which of the features present in the dataset are important for improving the model to make better predictions. For this reason, we can use the XG-Boosting algorithm to generate the feature importance map. This is very useful for:

1. Developers:
    a. Improve model with only important features and remove unimportant unnecessary features.
2. Business Users:
    a. To understand which of the attributes play an important role to make sure that the products are not delivered late.

Evaluation of Model

Decision Tree Model

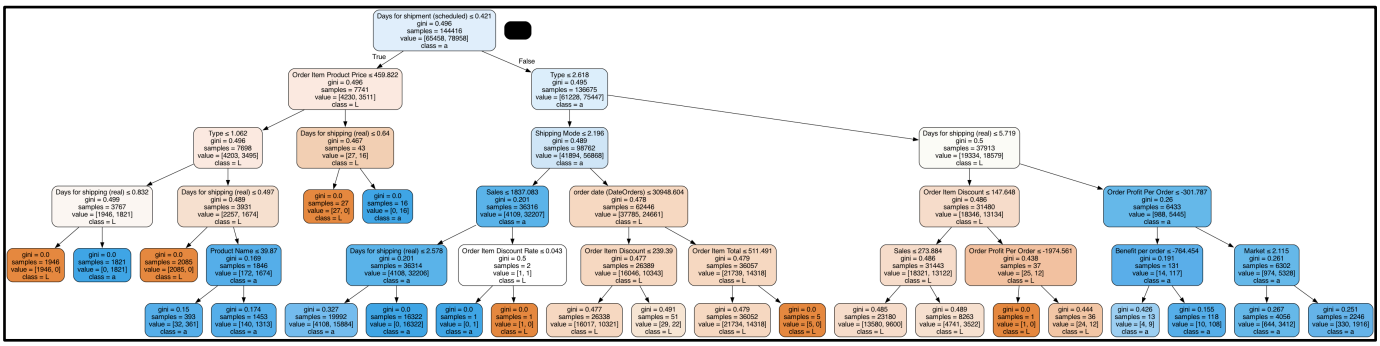


Figure 10: Decision Tree with 5 levels.

To evaluate the predictions, I have used the mean squared error method. When I ran the predictions for the first time, the percentage of error was 54.6%, therefore training accuracy was 46.4%.

Based on this information, I reckon that the model did not learn much from the training dataset. One reason for this could be that 5 levels of the decision tree is not enough for the model to make inferences from the data. Therefore, another model was created with a maximum depth of 10 levels, and a different random state – which yielded better training and test accuracy of 52%.

```
# Training the a second model on different parameters

tree_clf = DecisionTreeClassifier(max_depth=10, max_features="log2", random_state=5, spl
tree_clf.fit(X, y)
```

[47] ✓ 0.0s Python

```
DecisionTreeClassifier(max_depth=10, max_features='log2', random_state=5,
                       splitter='random')
```

```
from sklearn.tree import export_graphviz
import pydotplus
# from sklearn.externals.six import StringIO
from six import StringIO
from IPython.display import Image

dot_data = StringIO()
export_graphviz(
        tree_clf,
        out_file=dot_data,
        feature_names= X.columns.values.tolist(),
        class_names= "Late_delivery_risk",
        rounded=True,
        filled=True,
        special_characters=True
    )
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png('decision_tree_model_2.png')
Image(graph.create_png())
```

[48] ✓ 6.9s Python

```
dot: graph is too large for cairo-renderer bitmaps. Scaling by 0.658726 to fit

dot: graph is too large for cairo-renderer bitmaps. Scaling by 0.658726 to fit
```

Figure 11: Decision Tree with 10 levels.

A Second Look into the Data (Continuation for Final Project)

From the above work in the mid-term project, we can see that the decision tree had very bad training and testing accuracy of 52%. Let us revisit the dataset to explore the reasons why it performed so badly.

Data Preprocessing

One Hot Encoding

Firstly, let us revisit the categorical data. In the mid-term project, I used Label-Encoding to convert all the text columns to numbers. However, in some cases it might benefit the model to use One Hot Encoding rather than Label Encoders. Examples of such columns are as follows:

1. Type
2. Delivery Status
3. Customer Segment
4. Market
5. Shipping Mode

We can use One Hot Encoding on the above columns because they have a small amount of specific categorical data. The above columns have less than or equal to 5 categorical values each. If there are too many different categories, then One Hot Encoding is not a good idea as it will make the model very complex. For the 'Type' column, let us look at the One Hot Encoded values.

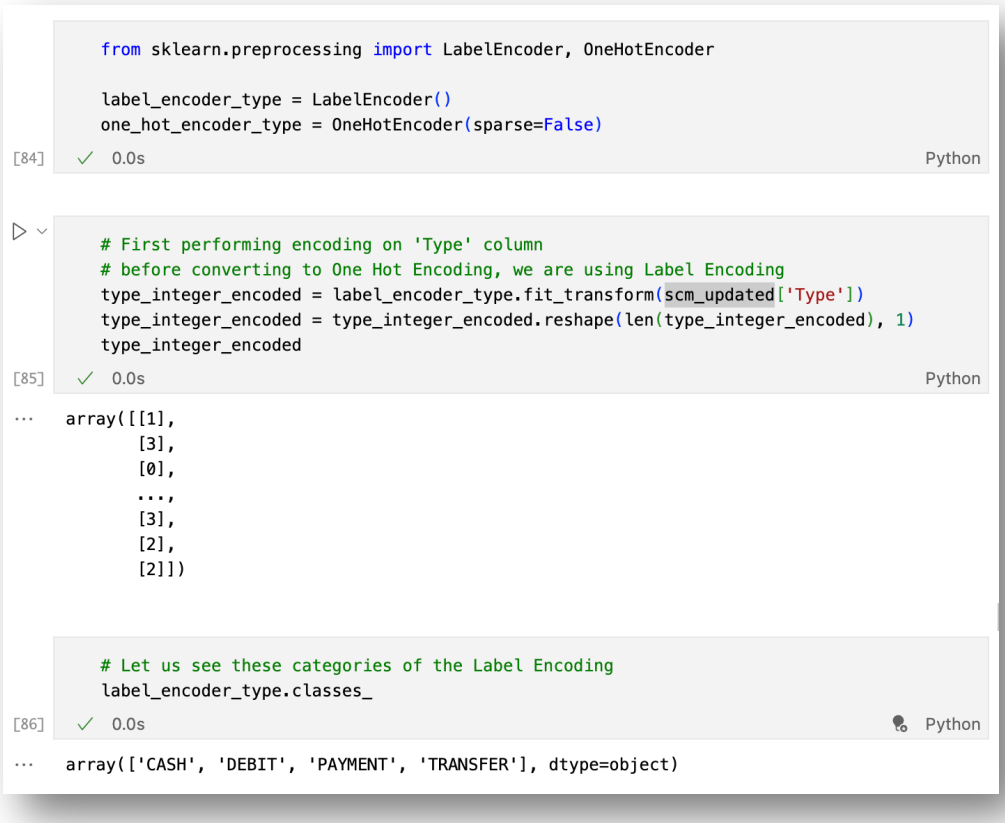
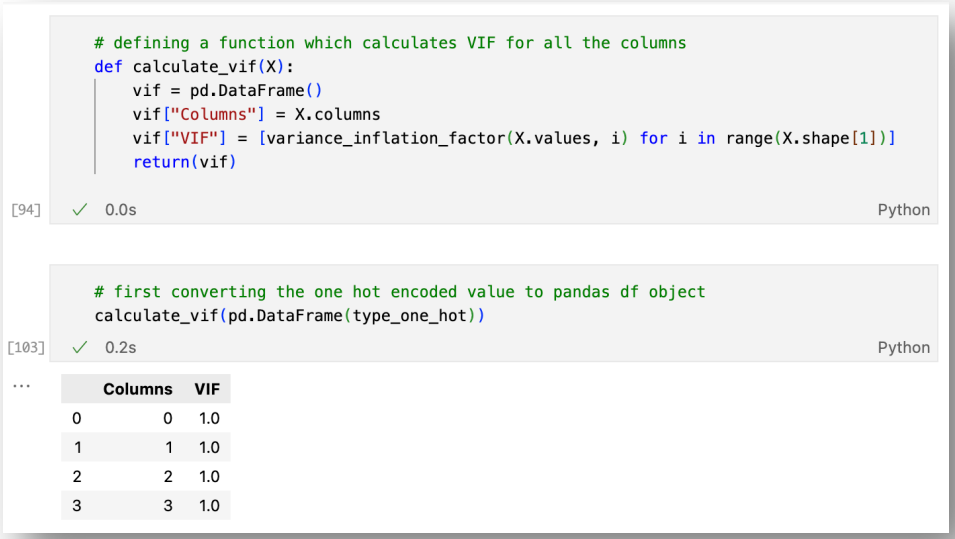


Figure 12: One Hot Encoding of 'Type'.

One Hot Encoding comes with some challenges which must be tackled – such as the Dummy Variable trap. When we create one hot encoding, some of the data becomes highly correlated to one another.

This causes the issue of multicollinearity. To find out whether multicollinearity is an issue we can use variance inflation factor (VIF) as a test. When the value of VIF is more than 5, it means that there is extremely high multi-collinearity.



```
# defining a function which calculates VIF for all the columns
def calculate_vif(X):
    vif = pd.DataFrame()
    vif["Columns"] = X.columns
    vif["VIF"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
    return(vif)
```

[94] ✓ 0.0s Python

```
# first converting the one hot encoded value to pandas df object
calculate_vif(pd.DataFrame(type_one_hot))
```

[103] ✓ 0.2s Python

	Columns	VIF
0	0	1.0
1	1	1.0
2	2	1.0
3	3	1.0

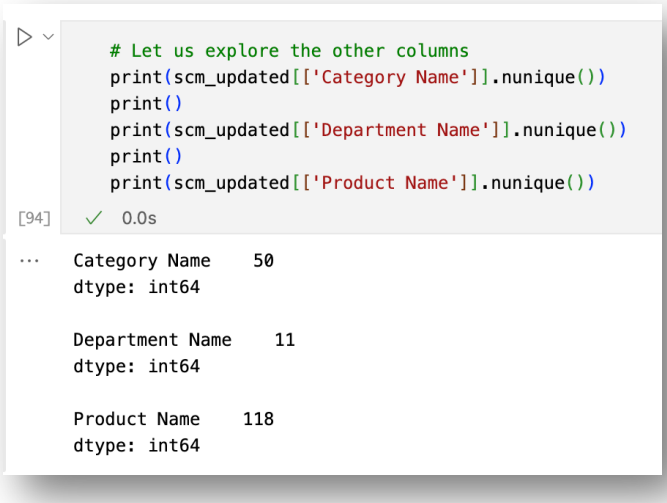
Figure 13: VIF on One Hot Encoding for 'Type' column.

From the above figure, we can see that all the columns have the value of VIF equal to 1. When the value is 1, it means that there is very little multi-collinearity. Therefore, we can add this one hot encoded value into the panda data frame object. I have used the same approach as above to One Hot Encode the rest of the above-mentioned columns.

Binary Encoding

Apart from the above One Hot Encoded columns, there are still columns which have text data such as:

1. Category Name
2. Department Name
3. Product Name



```
# Let us explore the other columns
print(scm_updated[['Category Name']].nunique())
print()
print(scm_updated[['Department Name']].nunique())
print()
print(scm_updated[['Product Name']].nunique())
```

[94] ✓ 0.0s

```
Category Name    50
dtype: int64

Department Name    11
dtype: int64

Product Name    118
dtype: int64
```

For the columns listed above, there are too many categories of data. This makes it unreasonable to use One Hot Encoding as the model will become highly complex. Furthermore, the values in this column do not have any inherent order. This means that Label Encoders will also not be a good idea. For this project I have decided to explore the use of Binary Encoders as a solution to encode the above columns.

Binary encoders are a combination of hash encoders and one hot encoders. They first convert the text into numeric data just like label encoders. Once this is complete, each number is converted into its binary equivalent. Finally, the binary number is split into various columns. For example, we have 118 categories of data in the 'Product Name' column. This would be first converted into binary which is 1110110. Now finally, it would create 7 columns to store each bit. This is much more memory efficient, as it only needs 7 columns to store 118 categories, whereas one hot encoders will require 118 columns.

```
binary_encoder_cat_name = ce.BinaryEncoder(cols=['Category Name'], return_df=True)
cat_name_binary_encoded_data = binary_encoder_cat_name.fit_transform(scm_updated['Category Name'])
cat_name_binary_encoded_data
```

[96] ✓ 0.0s

	Category Name_0	Category Name_1	Category Name_2	Category Name_3	Category Name_4	Category Name_5
0	0	0	0	0	0	1
1	0	0	0	0	0	1
2	0	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
...	...	...	...	...	...	...
180514	1	0	0	0	0	0
180515	1	0	0	0	0	0
180516	1	0	0	0	0	0
180517	1	0	0	0	0	0
180518	1	0	0	0	0	0

180519 rows × 6 columns

Converting 50 (decimal) to binary will result in 110010. This requires 6 columns (since there are 6 bits) to represnt all the data. Practically, from the above we can see that the binary encoding has created 6 columns - column 0 to column 5.

Date Columns

```
scm_updated['order date (DateOrders)'].head()
```

[101] ✓ 0.0s

```
0    1/31/2018 22:56
1    1/13/2018 12:27
2    1/13/2018 12:06
3    1/13/2018 11:45
4    1/13/2018 11:24
Name: order date (DateOrders), dtype: object
```

```
scm_updated['shipping date (DateOrders)'].head()
```

[102] ✓ 0.0s

```
0     2/3/2018 22:56
1    1/18/2018 12:27
2    1/17/2018 12:06
3    1/16/2018 11:45
4    1/15/2018 11:24
Name: shipping date (DateOrders), dtype: object
```

From the above figure, we can see that there are two columns which contain the date/ time information. Pandas has stored these values as Strings (object) datatype; therefore we must convert it into some numeric value. I will do so by splitting the date values into multiple different columns. For instance, order date will be split into the following:

- Order Date Month
- Order Date Day
- Order Date Year
- Order Date Hour
- Order Date Min

First, I converted the string values into date-time 64 datatype using to_datetime() function provided by pandas library. Next, I split each value into a new column, and the result is as follows:

```
# Let us look at the new columns now:
scm_updated[['order date (DateOrders)', 'Order Date Year', 'Order Date Month', 'Order Date Day', 'Order Date Hour', 'Order Date Min']].head()
```

[111] ✓ 0.0s

	order date (DateOrders)	Order Date Year	Order Date Month	Order Date Day	Order Date Hour	Order Date Min
0	2018-01-31 22:56:00	2018	1	31	22	56
1	2018-01-13 12:27:00	2018	1	13	12	27
2	2018-01-13 12:06:00	2018	1	13	12	6
3	2018-01-13 11:45:00	2018	1	13	11	45
4	2018-01-13 11:24:00	2018	1	13	11	24

```
# Let us look at the new columns now:
scm_updated[['shipping date (DateOrders)', 'Shipping Date Year', 'Shipping Date Month', 'Shipping Date Day', 'Shipping Date Hour', 'Shipping Date Min']].head()
```

[112] ✓ 0.0s

	shipping date (DateOrders)	Shipping Date Year	Shipping Date Month	Shipping Date Day	Shipping Date Hour	Shipping Date Min
0	2018-02-03 22:56:00	2018	2	3	22	56
1	2018-01-18 12:27:00	2018	1	18	12	27
2	2018-01-17 12:06:00	2018	1	17	12	6
3	2018-01-16 11:45:00	2018	1	16	11	45
4	2018-01-15 11:24:00	2018	1	15	11	24

Handling Null Values

Machine learning models will throw an error if they encounter null values, therefore we must handle them accordingly. In our dataset, we can see that the Order Zip Code has 155,679 null rows.

```
# setting pandas to display all rows
pd.set_option('display.max_rows', None)

scm_updated.isna().sum()
```

[136] ✓ 0.0s

```
Days for shipping (real)              0
Days for shipment (scheduled)         0
Benefit per order                     0
Sales per customer                    0
Late_delivery_risk                    0
Customer City                         0
Customer Country                      0
Customer State                        0
Customer Street                       0
Customer Zipcode                      3
Latitude                              0
Longitude                             0
Order City                            0
Order Country                         0
Order Item Discount                   0
Order Item Discount Rate              0
Order Item Product Price              0
Order Item Profit Ratio               0
Order Item Quantity                   0
Sales                                 0
Order Item Total                      0
Order Profit Per Order                0
Order Region                          0
Order State                           0
Order Zipcode                    155679
```

Considering that our dataset has a total of 180,519 rows, it would be better to drop the order zip code column altogether. However, the customer Zip Code has only 3 rows. Therefore for these three values we can drop only the null rows:

```
# dropping entire column for the order zipcode
scm_updated.drop(columns=['Order Zipcode'], inplace=True)
```

[146] ✓ 0.0s

```
# dropping the 3 null rows
scm_updated.dropna(inplace=True)
```

[148] ✓ 0.1s

Understanding & Handling Location Based Data

We can see that there are columns which contain address data as shown in the figure below:

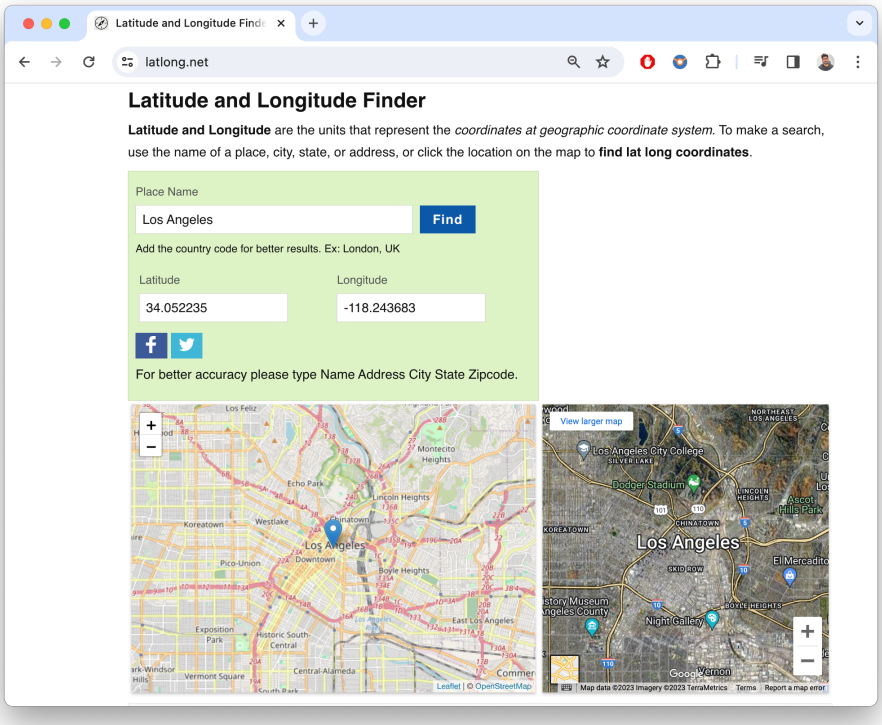
```
# setting pandas to display 10 rows
pd.set_option('display.max_rows', 10)

scm_updated[['Customer Street', 'Customer City', 'Customer State', 'Customer Country', 'Customer Zipcode', 'Order City', 'Order State', 'Order Country', 'Latitude', 'Longitude']].head(-10)
```

[145] ✓ 0.0s Python

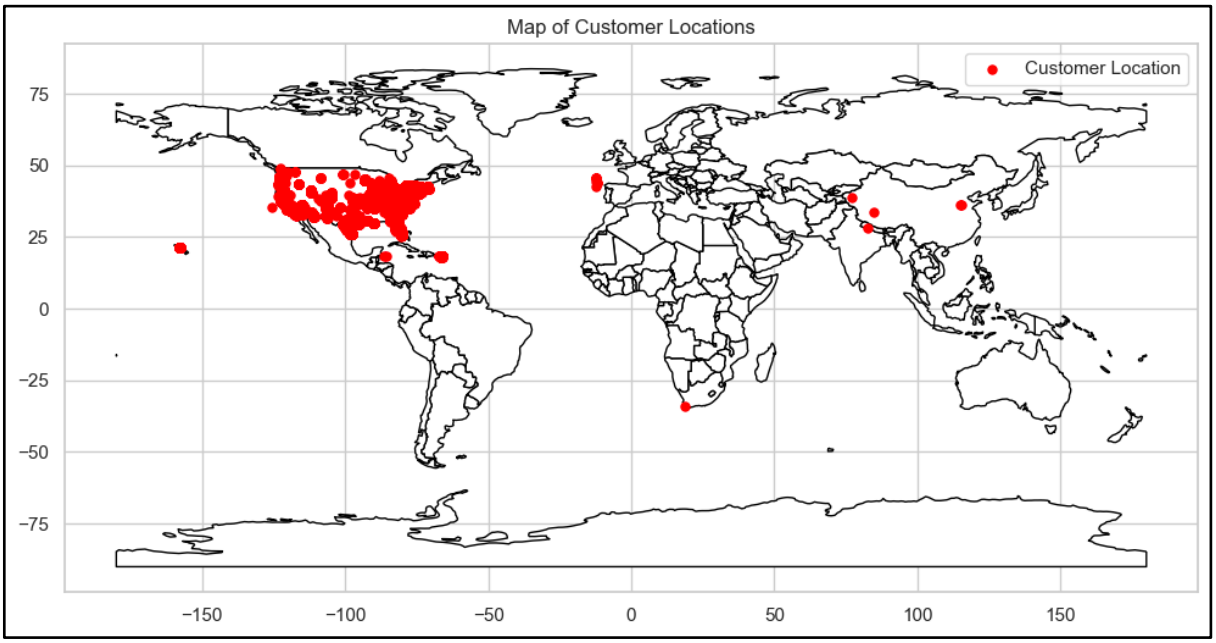
	Customer Street	Customer City	Customer State	Customer Country	Customer Zipcode	Order City	Order State	Order Country	Latitude	Longitude
0	5365 Noble Nectar Island	Caguas	PR	Puerto Rico	725.0	Bekasi	Java Occidental	Indonesia	18.251453	-66.037056
1	2679 Rustic Loop	Caguas	PR	Puerto Rico	725.0	Bikaner	Rajastán	India	18.279451	-66.037064
2	8510 Round Bear Gate	San Jose	CA	EE. UU.	95125.0	Bikaner	Rajastán	India	37.292233	-121.881279
3	3200 Amber Bend	Los Angeles	CA	EE. UU.	90027.0	Townsville	Queensland	Australia	34.125946	-118.291016
4	8671 Iron Anchor Corners	Caguas	PR	Puerto Rico	725.0	Townsville	Queensland	Australia	18.253769	-66.037048
...	...	...	...	...	...	...	...	...	...	...
180504	2137 Silver Brook Square	Caguas	PR	Puerto Rico	725.0	Canberra	Territorio de la Capital Australiana	Australia	18.240986	-66.370552
180505	7127 Quiet Zephyr Nook	Highland	CA	EE. UU.	92346.0	Qingdao	Shandong	China	34.135822	-117.220947
180506	7106 Rustic Hickory Front	San Antonio	TX	EE. UU.	78228.0	Sanya	Hainan	China	29.476566	-98.579216
180507	5057 Fallen Knoll	Philadelphia	PA	EE. UU.	19134.0	Hanoi	Th? Dô Hà N?i	Vietnam	39.986546	-75.108467
180508	5237 Rustic Village	Waipahu	HI	EE. UU.	96797.0	Guangshui	Hubei	China	21.394211	-157.998016

We can see that the customer address is present in the dataset, and the order location as well. In the data we can see some discrepancy. In rows 3 & 4, we can see that they are cities in the United States of America (USA), however in the Customer Country column – we can see the value as ' EE. UU.'. Let us ignore this for now. We can see that many of the customer locations are in USA, or Puerto Rico, and that the order location is India, Vietnam, China and Australia. It could be possible that the order location is the location of the store from where the order might originate from.

In the data definition csv file, it is mentioned that the latitude & longitude belong to the store, however I did some analysis and realized that the latitude & longitude data is of the customer location. For example on row 3, we can see that customer city is Los Angeles and order city is Townsville. I used an online Latitude & Longitude Finder and noticed that the latitude & longitude value as mentioned in the data frame correspond to the customer city.



Therefore, when we mapped the latitude & longitude on the map of the world initially, we were mapping the customer location, and not the store location.



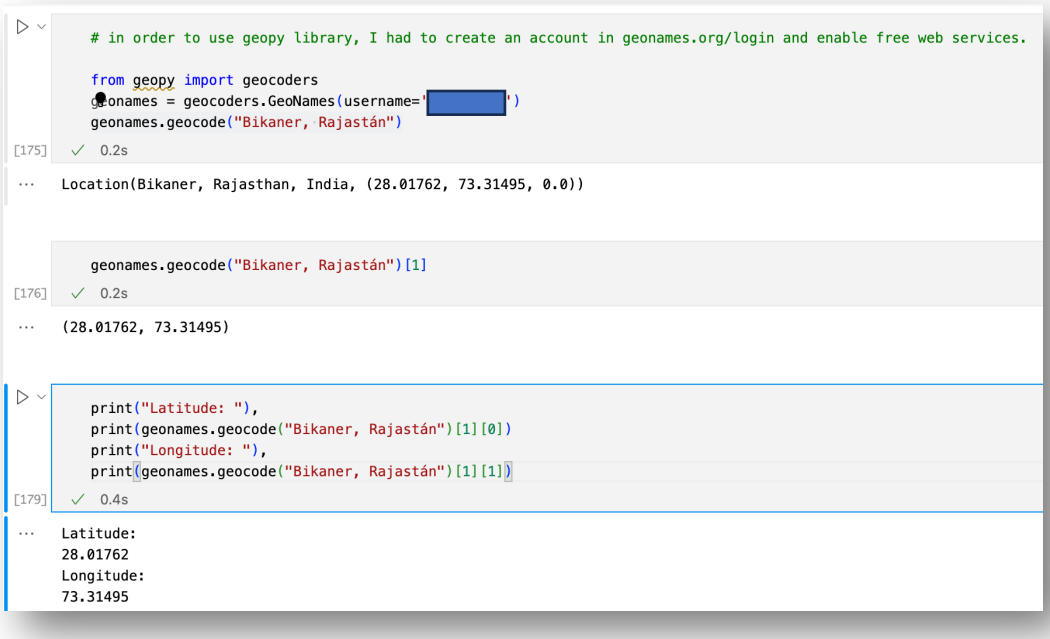
We can see that most of the orders have their customers present in North American region, and with very few customers in China, India and South Africa and Spain / Portugal. Since we have the latitude and longitude of the customers, we do not require the other customer address columns – such as street, city, state, etc.

Feature Engineering

Store Location – Latitude & Longitude

Similar to the above figure, it is important for us to understand the order store locations. One major issue for the late deliveries could be because the customer locations are far away from the stores. Therefore, using the data of the order location (presumably the store location) let us try to create a new feature by finding the latitude and longitude values.

To achieve this, I created an account on the website geonames.org/login and enabled the free web services. Next I tried to test whether this works using my credentials, and as we can see in the below screenshot, it has worked.



```
# in order to use geopy library, I had to create an account in geonames.org/login and enable free web services.

from geopy import geocoders
geonames = geocoders.GeoNames(username='[redacted]')
geonames.geocode("Bikaner, Rajastán")
```

[175] ✓ 0.2s

```
Location(Bikaner, Rajasthan, India, (28.01762, 73.31495, 0.0))
```

```
geonames.geocode("Bikaner, Rajastán")[1]
```

[176] ✓ 0.2s

```
(28.01762, 73.31495)
```

```
print("Latitude: "),
print(geonames.geocode("Bikaner, Rajastán")[1][0])
print("Longitude: "),
print(geonames.geocode("Bikaner, Rajastán")[1][1])
```

[179] ✓ 0.4s

```
Latitude:
28.01762
Longitude:
73.31495
```

Now I am writing a function which can convert all the data of the order city and order state to obtain the latitude and longitude values of the order location.

```
# crearted this function to iterate through the data frame, and return a new dataframe contianing the latitude and longitude
from geopy import geocoders
import pandas as pd

def get_latitude_longitude_values(df):
    geoname = geocoders.GeoNames(username='[redacted]')
    new_df = pd.DataFrame(columns=['Order Latitude', 'Order Longitude'])
    for ind, row in df.iterrows():
        new_row = pd.DataFrame({'Order Latitude': geoname.geocode(str(row['Order City'])+", "+str(row['Order State']))[1][0],
                                'Order Longitude': geoname.geocode(str(row['Order City'])+", "+str(row['Order State']))[1][1]
                               },
                               index=[ind]
                              )
        new_df = pd.concat([new_df, new_row], ignore_index=True)
        print("Done with "+str(ind))
        # if(ind == 9):
            #safety break
            # break
    return new_df
```

[291] ✓ 0.0s

While running this function there were some data points for which the latitude and longitude values failed to get generated. For example ‘Mandurah, Australia Occidental’, ‘Wollongong, Nueva Gales del Sur’, ‘Murray Bridge, Australia del Sur’ and ‘Hayange, Alsacia-Champaña-Ardenas-Lorena‘ to name a few. It is clear that the names of the states has been collected in a foreign language – perhaps Spanish.

```
scm_updated.loc[scm_updated['Order State'] =='Australia Occidental', 'Order State']
```

[308] ✓ 0.0s

```
25        Australia Occidental
26        Australia Occidental
245       Australia Occidental
396       Australia Occidental
397       Australia Occidental
                ...
180161    Australia Occidental
180196    Australia Occidental
180273    Australia Occidental
180333    Australia Occidental
180377    Australia Occidental
Name: Order State, Length: 917, dtype: object
```

```
scm_updated.loc[scm_updated['Order State'] =='Australia Occidental', 'Order State'] = 'Australia'
```

[309] ✓ 0.0s

```
scm_updated.loc[scm_updated['Order State'] =='Australia Occidental', 'Order State']
```

[310] ✓ 0.0s

```
Series([], Name: Order State, dtype: object)
```

Therefore, for the rest of the values, I decided to find the latitude and longitude only based on the name of the city. After calculating more than 1,280 latitude and longitude values, I encountered the problem wherein I exhausted the number of credits for the free account on the geopy library. Hence, I will abandon the feature engineering of the Order store latitude and longitude out of scope for this final project.

```
File /opt/homebrew/anaconda3/envs/info6105/lib/python3.8/site-packages/geopy/geocoders/base.py:386, in Geocoder._call_geocoder(self, url, callback, timeout, is_json, headers)
    384         return fut()
...
    311 if code in (18, 19, 20):
--> 312     raise GeocoderQuotaExceeded(message)
    313 raise GeocoderServiceError(message)

GeocoderQuotaExceeded: the hourly limit of 1000 credits for [redacted] has been exceeded. Please throttle your requests or use the commercial service.
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

We could additionally work on normalization or feature scaling; however, it is not very important when working on decision tree models. It is more important for algorithms like gradient descent or those which use the neighboring points for classification.

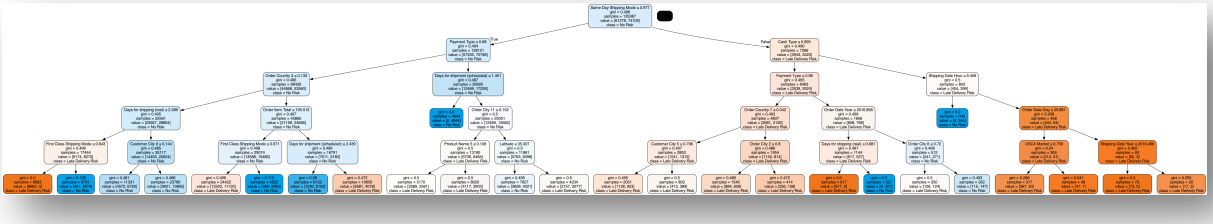
Decision Tree Model #3 (With 5 levels)

We have completed extensive data pre-processing, and now we are ready to train the model. First let us train a decision tree model on the same parameters as we did in the mid-term project, and let us see if it has better performance.

```
tree_clf_3 = DecisionTreeClassifier(max_depth=5, max_features="sqrt", random_state=23, splitter="random")
tree_clf_3.fit(X_train_v2, y_train_v2)
```

[344] ✓ 0.1s

```
DecisionTreeClassifier(max_depth=5, max_features='sqrt', random_state=23,
                       splitter='random')
```



Now let us see how the model performs on training and testing data. As we can see in the below figure, our latest decision tree model performed with 65.96 % training accuracy and 66.08 % testing accuracy.

```
# Let us see how well the training does:
tree_clf_3.score(X_train_v2, y_train_v2)
```

[346] ✓ 0.0s

```
0.6596128136379416
```

```
tree_clf_3.score(X_test_v2, y_test_v2)
```

[347] ✓ 0.0s

```
0.6607724523033969
```

In the mid-term project the first decision tree model which had similar hyper-parameters performed much worse with only 46.40% training accuracy, however this decision tree performed more than 20% better. This model is also better than the second decision tree model from the mid-term project which had 10 levels and an accuracy of only 51.8%.

Random Forest Model #1

Random Forest usually create better model when compared to decision trees. This is because it is a type of ensemble learning technique which uses bagging to

```
from sklearn.ensemble import RandomForestClassifier

rf_clf = RandomForestClassifier(random_state=23,
                                n_jobs=-1,
                                max_depth=5,
                                n_estimators=100,
                                oob_score=True)
```

[348] ✓ 0.2s

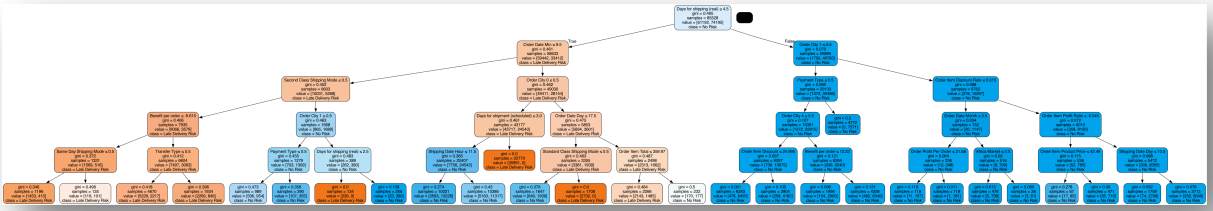
```
%%time
rf_clf.fit(X_train_v2, y_train_v2)
```

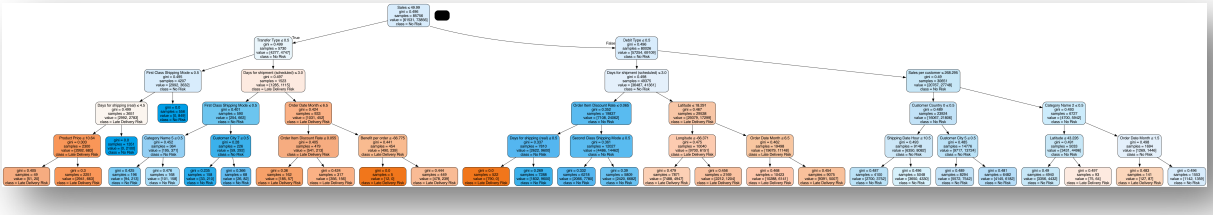
[350] ✓ 3m 49.7s

```
CPU times: user 25min 27s, sys: 8.24 s, total: 25min 36s
Wall time: 3min 49s
```

RandomForestClassifier

```
RandomForestClassifier(max_depth=5, n_jobs=-1, oob_score=True, random_state=23)
```





We have trained the above model. As you can see it took 3 min & 49 seconds to completely train the model. The algorithm random forest created multiple decision trees which make up a ‘forest’, hence the name. Above I have plotted the first and second estimator tree for the same model. As shown below, the training accuracy of the model (best estimator) was much higher at 93.60%, and the test accuracy was similar at 93.58%.

```
# checking the training accuracy
rf_clf.score(X_train_v2, y_train_v2)
```
[351] ✓ 0.2s

```
0.9359982863938192
```

For the random forest model, we can see that the training acuracy is 93.60%. Let us look at the accuracy of the test set.

```
# checking the test accuracy
rf_clf.score(X_test_v2, y_test_v2)
```
[352] ✓ 0.0s

```
0.9358284030224467
```

The test accuracy is very close at 93.58%.

This model was much better than the 3rd decision tree model by 28%, and significantly better than the first decision tree model by 47%.

Feature Importance

Now that the random forest model is trained, we can find out the feature importance map. This shows which of the attributes from the dataset had the highest impact to the model. The below figure shows the feature importance of the previous model.

```
pd.set_option('display.max_rows', None)
important_features_rf_clf = pd.DataFrame({
    "Column": X_train_v2.columns,
    "Importance Score": rf_clf.feature_importances_
})
important_features_rf_clf.sort_values(by='Importance Score', ascending=False)
```
[364] ✓ 0.0s

	Column	Importance Score
0	Days for shipping (real)	0.346944
1	Days for shipment (scheduled)	0.216117
30	Standard Class Shipping Mode	0.202429
27	First Class Shipping Mode	0.120478
29	Second Class Shipping Mode	0.045490
56	Shipping Date Hour	0.017773
28	Same Day Shipping Mode	0.016205
51	Order Date Hour	0.015866
18	Transfer Type	0.007911
55	Shipping Date Day	0.001983
50	Order Date Day	0.001364
16	Debit Type	0.001320
17	Payment Type	0.001060
4	Latitude	0.000513
5	Longitude	0.000413

Let us consider the top 15 features and train another Random Forest model on only these features.

Random Forest Model #2 with Top 15 Important Features

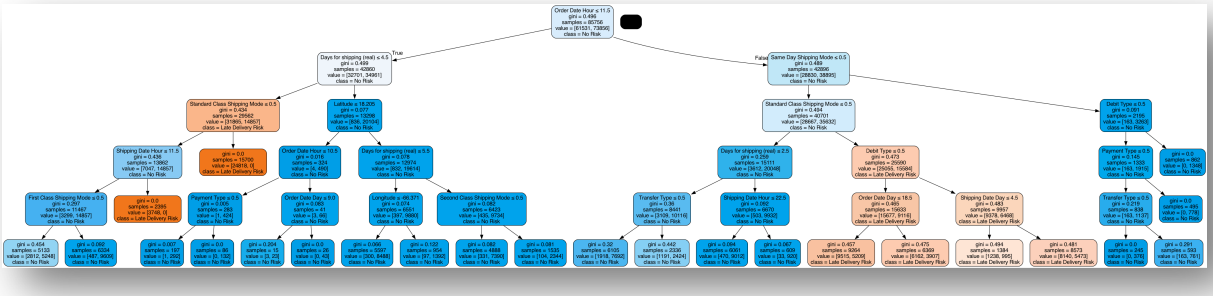
```
rf_clf_2 = RandomForestClassifier(random_state=23,
                                  n_jobs=-1,
                                  max_depth=5,
                                  n_estimators=100,
                                  oob_score=True)
```

[369] ✓ 0.0s

```
rf_clf_2.fit(X_15_train, y_15_train)
```

[370] ✓ 55.5s

```
RandomForestClassifier(max_depth=5, n_jobs=-1, oob_score=True, random_state=23)
```



Let us see how the model performs on training and testing data.

```
rf_clf_2.score(X_15_train, y_15_train)
```

[372] ✓ 0.1s

```
0.9754038423186865
```

```
rf_clf_2.score(X_15_test, y_15_test)
```

[373] ✓ 0.0s

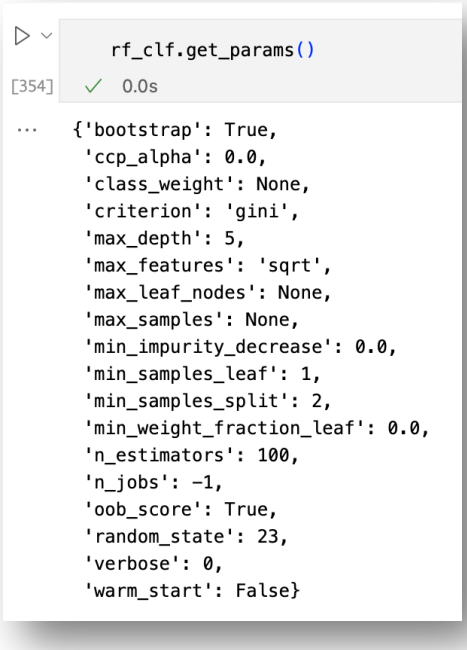
```
0.975780540229121
```

From the figure above, we can see that the second model has outperformed the previous random forest model. The training accuracy of this model is 97.54% and the testing accuracy of this model is 97.58%. So far this is the best model that we have trained.

Hyper Parameter Tuning with Randomized Search CV

We understand that by adjusting the hyper-parameters of the machine learning models, we will obtain accuracy which is sometimes better or sometimes worse than the initial ones. Hyper parameter values are impossible for us to find out ahead of time. How do we know which hyper-parameters will yield the best results? The answer is that we do not know until we try to find them using trial and error method. There are ways in which we can try to find the best hyper-parameters using K-Fold Cross Validation (CV). Sci-kit learn library has predefined methods for us to use for such tasks.

Let us look at the current parameters of the 2nd Random Forest Model:

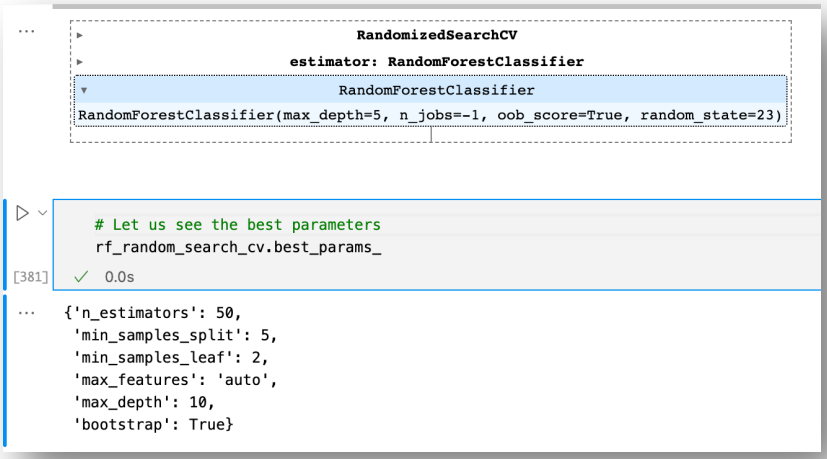


I will use the Randomized Search CV algorithm which will at random sample a wide range of data and give us the likely best parameter values.

```
# Creating the random grid as a dict
random_grid = {'n_estimators': [25, 50, 100, 200, 400],
               'max_features': ['auto', 'sqrt'],
               'max_depth': [3, 5, 10, 15, 20],
               'min_samples_split': [2, 3, 5],
               'min_samples_leaf': [1, 2],
               'bootstrap': [True, False]}
```
[378] ✓ 0.0s

```
from sklearn.model_selection import RandomizedSearchCV

rf_random_search_cv = RandomizedSearchCV(estimator=rf_clf_2,
                                         param_distributions=random_grid,
                                         n_iter=25,
                                         cv=2,
                                         verbose=2,
                                         random_state=23,
                                         n_jobs=-1)
```
[379] ✓ 0.0s

```
%%time
rf_random_search_cv.fit(X_15_train, y_15_train)
```
[380] ✓ 9m 11.5s




```
# Let us see the best parameters
rf_random_search_cv.best_params_
```
[381] ✓ 0.0s

```
{'n_estimators': 50,
 'min_samples_split': 5,
 'min_samples_leaf': 2,
 'max_features': 'auto',
 'max_depth': 10,
 'bootstrap': True}
```

Using these hyper-parameters, let us train a third Random Forest model to see how it performs.

Random Forest Model #3 using Randomized Search CV

```
rf_clf_3 = RandomForestClassifier(random_state=23,
                                  n_jobs=-1,
                                  max_depth=10,
                                  n_estimators=50,
                                  min_samples_split=5,
                                  min_samples_leaf=2,
                                  max_features='auto',
                                  bootstrap=True,
                                  oob_score=True)
```

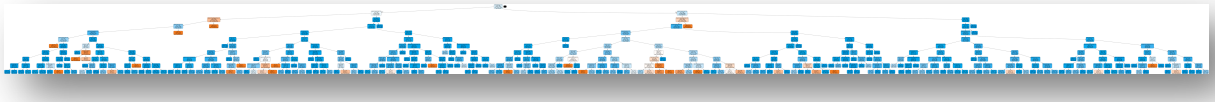
[382] ✓ 0.0s

```
rf_clf_3.fit(X_15_train, y_15_train)
```

[383] ✓ 29.8s

```
/opt/homebrew/anaconda3/envs/info6105/lib/python3.8/site-packages/sklearn/ensemble/_forest.py:427: FutureWarning:

`max_features='auto'` has been deprecated in 1.1 and will be removed in 1.3. To keep the past behaviour, explicitly set `max_features='sqrt'`
```

```
RandomForestClassifier(max_depth=10, max_features='auto', min_samples_leaf=2,
                       min_samples_split=5, n_estimators=50, n_jobs=-1,
                       oob_score=True, random_state=23)
```



The above two figures show the random forest model #3. As the max-depth of the model is 10, it is very large for us to be able to visualize in this report. The figure below shows the accuracy of this model is 97.55% for training and 97.58% for testing. These results are very similar to the previous random forest model, but just 0.01% better on the training dataset.

```
rf_clf_3.score(X_15_train, y_15_train)
```

[384] ✓ 0.0s

```
0.9755146358217555
```

```
rf_clf_3.score(X_15_test, y_15_test)
```

[385] ✓ 0.0s

```
0.9758470163309624
```

Therefore, this is the best model we have been able to generate in this final project.

Results

Model	Description	Train Accuracy	Test Accuracy	Covered In
Decision Tree #1	With 5 levels.	46.40 %	Decided Not to Test.	Mid Term
Decision Tree #2	With 10 levels.	51.82 %	51.86 %	Mid Term
Decision Tree #3	With 5 levels post extensive data preprocessing.	65.96 %	66.08 %	Final Project
Random Forest #1	With 5 Levels post extensive data processing.	93.60 %	93.58 %	Final Project
Random Forest #2	**With 15 most important features from previous Random Forest model**	**97.54 %**	**97.58 %**	**Final Project**
Random Forest #3	**With Randomized Search CV**	**97.55 %**	**97.58 %**	**Final Project**

Conclusion

In this mid-term exam, I have worked on a large dataset using Decision Trees, to try to solve a real-world problem in the Supply Chain industry – which is to predict whether an item has the risk of a late delivery. I have performed data visualization, train decision tree models and received a training and testing accuracy of 52%.

During the final project, I have revisited the data. I have used specialized ways of encoding the categorical data using One Hot Encoding and Binary Encoders based on the nature of the data. Furthermore, I converted the date columns and split them by Date, Month, Year, Hour, and Minute. I also attempted to perform feature engineering by generating latitude and longitude data using the order city, and order state. However, I aborted the feature engineering task as the free credits for using the library expired for my user. Once all the data preprocessing was completed, I re-ran the same decision tree model and noticed nearly 20% improvement in the accuracy from the first model (66% accuracy). Next, I trained a random forest model with the preprocessed data which yielded a high accuracy of 93%. This was nearly 30% better than the previous model and 47% better than the first decision tree model. I continued trying to improve the model by finding the top 15 most important features and then training a second random forest (RF) model. This 2nd RF model toped all the previous models with a 97% training and testing accuracy. To improve this model even further, I attempted to use Randomized Search CV to find out the optimum hyper-parameters for the random forest model. With the result from this, I re-trained the third random forest model and received training accuracy of 97.55% which was 0.01% better than the last model and 51% better than the first decision tree model.

Acknowledgements

The dataset for this project has been obtained from the following websites:

- https://www.kaggle.com/datasets/shashwatwork/dataco-smart-supply-chain-for-big-data-analysis/data
- https://data.mendeley.com/datasets/8gx2fvg2k6/5

Additionally, the following resources have been very useful while performing the latitude & longitude feature engineering:

- https://www.latlong.net/
- https://www.geonames.org/

References

1. *Why Global Supply Chains May Never Be the Same: WSJ Documentary*. YouTube. (2022, March 23). https://youtu.be/1KtTAb9Tl6E?si=11--rHEKcscN7Gs5
2. *Pandas.dataframe.plot.pie*. pandas.DataFrame.plot.pie - pandas 2.1.1 documentation. (n.d.). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.pie.html
3. *Seaborn.barplot*. seaborn.barplot - seaborn 0.13.0 documentation. (n.d.). https://seaborn.pydata.org/generated/seaborn.barplot.html
4. *Seaborn.countplot#*. seaborn.countplot - seaborn 0.13.0 documentation. (n.d.). https://seaborn.pydata.org/generated/seaborn.countplot.html
5. *Pandas.dataframe.hist#*. pandas.DataFrame.hist - pandas 2.1.2 documentation. (n.d.). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.hist.html
6. Lucaswolff. (2023, September 10). *Inferential statistical analysis on Supply Chain*. Kaggle. https://www.kaggle.com/code/lucaswolff/inferential-statistical-analysis-on-supply-chain
7. Navlani, A. (2023, February 23). *Python decision tree classification tutorial: Scikit-Learn Decisiontreeclassifier*. DataCamp. https://www.datacamp.com/tutorial/decision-tree-classification-python
8. *Sklearn.tree.decisiontreeclassifier*. scikit. (n.d.). https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
9. Sethi, A. (2023, June 15). *One hot encoding vs. label encoding using Scikit-Learn*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/03/one-hot-encoding-vs-label-encoding-using-scikit-learn/
10. Bhandari, A. (2023, November 9). *Multicollinearity: Causes, effects and detection using VIF (updated 2023)*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/03/what-is-multicollinearity/
11. Saxena, S. (2023, September 24). *What are categorical data encoding methods: Binary encoding*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2020/08/types-of-categorical-data-encoding/#:~:text=Two%20commonly%20used%20techniques%20for,performance%20of%20a%20ML%20model.%E2%80%A6
12. *Binary - Category Encoders* . Binary - Category Encoders 2.6.3 documentation. (n.d.). https://contrib.scikit-learn.org/category_encoders/binary.html
13. Long, A. (2020a, February 2). *Machine learning with Datetime feature engineering: Predicting Healthcare appointment no-shows*. Medium. https://towardsdatascience.com/machine-learning-with-datetime-feature-engineering-predicting-healthcare-appointment-no-shows-5e4ca3a85f96
14. *Pandas.to_datetime#*. pandas.to_datetime - pandas 2.1.3 documentation. (n.d.). https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html
15. *Datetime - basic date and time types*. Python documentation. (n.d.). https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior
16. *Pandas.dataframe.groupby#*. pandas.DataFrame.groupby - pandas 2.1.4 documentation. (n.d.). https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html
17. *Welcome to GeoPy's documentation!ℑ*. Welcome to GeoPy's documentation! - GeoPy 2.4.1 documentation. (n.d.). https://geopy.readthedocs.io/en/stable/
18. R, S. E. (2023a, October 26). *Understand random forest algorithms with examples (updated 2023)*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/
19. Koehrsen, W. (2018, January 10). *Hyperparameter tuning the random forest in python*. Medium. https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74