Bolt Case Study

Double-click (or enter) to edit

Introduction

- **Problem Statement:** We have geographical data for the city of Tailin for which we need to create an algorithmic order anticipation for the rider. The order recommendation should point towards higher order value.
- We will create a baseline model for this and describe the methodology taken to solve this problem.

```
!pip install haversine
!pip install lime
```

```
Collecting haversine
  Downloading haversine-2.5.1-py2.py3-none-any.whl (6.1 kB)
Installing collected packages: haversine
Successfully installed haversine-2.5.1
Collecting lime
  Downloading lime-0.2.0.1.tar.gz (275 kB)
     |████████████████████████████████| 275 kB 13.5 MB/s
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (from lime) (3.2.2)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from lime) (1.21.6)
Requirement already satisfied: scipy in /usr/local/lib/python3.7/dist-packages (from lime) (1.4.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from lime) (4.64.0)
Requirement already satisfied: scikit-learn>=0.18 in /usr/local/lib/python3.7/dist-packages (from lime) (1.0.2)
Requirement already satisfied: scikit-image>=0.12 in /usr/local/lib/python3.7/dist-packages (from lime) (0.18.3)
Requirement already satisfied: tifffile>=2019.7.26 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.12->lime) (2
Requirement already satisfied: imageio>=2.3.0 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.12->lime) (2.4.1)
Requirement already satisfied: PyWavelets>=1.1.1 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.12->lime) (1.3
Requirement already satisfied: networkx>=2.0 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.12->lime) (2.6.3)
Requirement already satisfied: pillow!=7.1.0,!=7.1.1,>=4.3.0 in /usr/local/lib/python3.7/dist-packages (from scikit-image>=0.12
```

```
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib->lime) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->lime) (1.4.2)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3.7/dist-packages (from matplot
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib->lime) (2.8.2)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from kiwisolver>=1.0.1->matplotlib-
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-dateutil>=2.1->matplotlib->lime)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18->lime) (
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18->lime) (1.1.0)
Building wheels for collected packages: lime
  Building wheel for lime (setup.py) ... done
  Created wheel for lime: filename=lime-0.2.0.1-py3-none-any.whl size=283857 sha256=8a830186777116f1343cac6f57e49093d9d194057fe
  Stored in directory: /root/.cache/pip/wheels/ca/cb/e5/ac701e12d365a08917bf4c6171c0961bc880a8181359c66aa7
Successfully built lime
Installing collected packages: lime
Successfully installed lime-0.2.0.1
```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# import libraries
import pandas as pd
import numpy as np
from haversine import haversine
from sklearn.preprocessing import StandardScaler as std
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.preprocessing import MinMaxScaler, power_transform
from sklearn.model_selection import train_test_split
from sklearn.metrics import silhouette_score,davies_bouldin_score
from sklearn.cluster import AgglomerativeClustering,DBSCAN,KMeans
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
import scipy.cluster.hierarchy as sch
import matplotlib.pyplot as plt
import seaborn as sns
```

```
from mpl_toolkits.mplot3d import Axes3D
import collections
from IPython.display import display, HTML
from termcolor import colored
import matplotlib.pyplot as plt
from termcolor import colored
import plotly.express as px
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.metrics import f1_score, precision_score, confusion_matrix, recall_score, accuracy_score,classification_report,roc_auc_s
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from sklearn import linear_model
from sklearn.tree import DecisionTreeClassifier
from xgboost.sklearn import XGBClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import MinMaxScaler
from collections import Counter
import lime
import lime.lime_tabular
%matplotlib inline
```

```
# read the data
df = pd.read_csv("robotex5.csv")
```

Features engineered

The features extracted from the data will play an pivotal role in modelling the data. The features extracted from the data are as follows:

- `distance_travelled:` We calculate haversine distance between the coordinates. The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitude.
- `ride_mile_expense:` This gives us the distance covered by the **rider** per unit of expense. This gives us a brief hint of operational expense for each ride carried out by the rider.

```
# get the distance travelled
df["distance_travelled"] = [haversine((df['start_lat'][i],df['start_lng'][i]),(df['end_lat'][i], df['end_lng'][i]), unit = 'mi') for

# getting ride distance for 1 unit of ride
df['ride_mile_expense'] = df['distance_travelled'] / df['ride_value']
```

- `start_hour:` The hour at which ride was placed.
- `week_day:` The week day number at which order was placed.

```
# get the datetime entities
df['start_time'] = pd.to_datetime(df['start_time'])
df['start_hour'] = [df['start_time'][i].hour for i in range(len(df))]
df['week_day']   = [df['start_time'][i].weekday() for i in range(len(df))]
```

Visualisation

Let's visualise some of the features to understand the data.

```
# Created a visualisation class
class visualisation():
    '''
    Generates visualisation of each type of graph in one-go

    '''
    def __init__(self,df):

        '''
        Takes the dataframe and converts into features
        '''
        self.X = df
```

```
    def hist_plot(self,x = None, category = None):

        fig = px.histogram(self.X, x=x, color=category).update_xaxes(categoryorder = "total descending")
        fig.show()

    def heat_map(self, df = None):

        if df == None:
            corr = self.X.corr()
        else:
            corr = df.corr
        # Generate a mask for the upper triangle
        mask = np.triu(np.ones_like(corr, dtype=bool))

        # Set up the matplotlib figure
        f, ax = plt.subplots(figsize=(11, 9))

        # Generate a custom diverging colormap
        cmap = sns.diverging_palette(230, 20, as_cmap=True)

        # Draw the heatmap with the mask and correct aspect ratio
        sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
                    square=True, linewidths=.5, annot = True, cbar_kws={"shrink": .5})

    def box_plot(self, x = None ,  y = None, color=None):
        fig = px.box(self.X, x=x, y=y, color=color)
        fig.show()

    def line_plot(self, x = None, y = None):
        fig = px.scatter(self.X, x=x, y=y)
        fig.show()

    def bar_plot(self, x = None ,  y = None, color=None):
        fig = px.bar(self.X, x=x, y=y, color=color)
        fig.show()

    def dis_plot(self, x = None , hue=None, kind='hist', fill=False):
```

```
        fig = sns.displot(self.X, x=x, hue=hue, kind=kind, fill=fill)

    def pair_plot(self,kind = 'reg'):
        plt.figure(figsize=(15, 5), dpi=80)
        sns.pairplot(self.X,kind='reg')
        plt.show()

    def density_plot(self, x = None):
        fig, ax = plt.subplots(figsize = [12, 7])
        sns.distplot(self.X[x])
        fig.show()
```

```
# create a viz class
v = visualisation(df)
```

```
v.density_plot('ride_mile_expense')
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a deprecated function and wi
  warnings.warn(msg, FutureWarning)
```

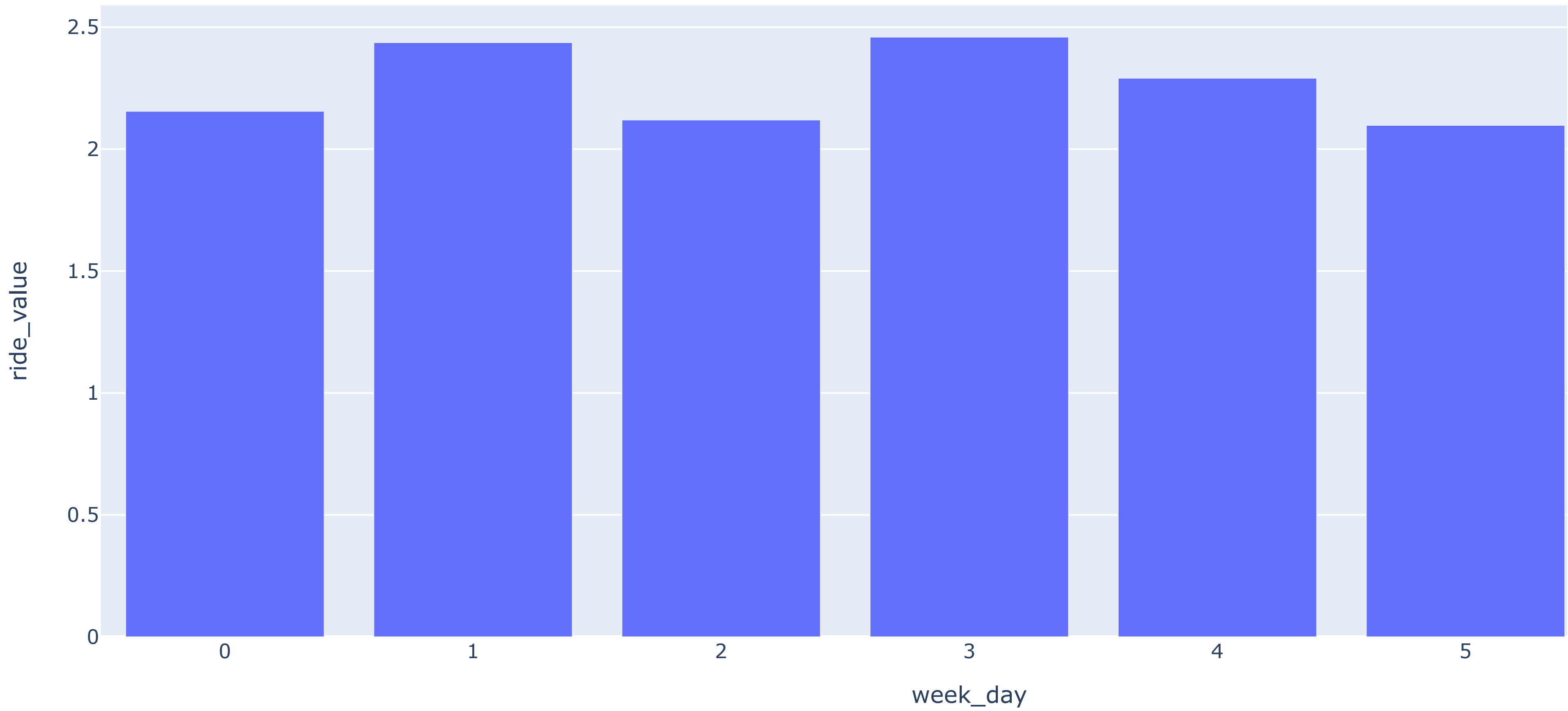
Observation: We see the density plot is more skewed towards bottom quartile signifying a smaller region in geography where riders mostly commute.

```
# mile vs distance
v.line_plot(x = 'distance_travelled', y = 'ride_value')
```

Observation: We see a linear relationship between ride value and distance travelled.

```
# get the expense of ride value by hour
t = pd.DataFrame(df.groupby(['start_hour']).agg({'ride_value':'mean'})).reset_index().sort_values(by=['ride_value'],ascending=False)
visualisation(t).bar_plot('start_hour','ride_value')
```

```
# get the expense of value by day
t = pd.DataFrame(df.groupby(['week_day']).agg({'ride_value':'mean'})).reset_index().sort_values(by=['ride_value'],ascending=False)
visualisation(t).bar_plot('week_day','ride_value')
```



Observation: These two observation shows we have order based on demand in each hour and weekday.

Additonal Features

The two more important features to scale would be `geography` and `weather`. Due to absence of API related query I am skipping these two features.

Clustering

- Since we have geographies that are spaced at various locations. We will cluster our geographies together to find out which clusters are having significance in assosiating ride value.
- For this purpose we will seperate out **3 clusters** based on our scores and metrics.

```
#### Clustering

# Clustering is done here with the help of following set of algorithms:
#  - KMeans
#  - Agglomerative clustering
#     - Hierarchial clustering
#     - DB scan clustering


# The following library has all the validation tests and algorithms to perform clustering

class clustering:
    """
    This class has all the tests of clustering criterion to find the optimal number of clusters
    and all the clustering methods to do clustering
    Clustering methods dicussed over here are:
    1.  Agglomerative clustering
         1.1) Hierarchial clustering
         1.2) DB scan clustering
    2.  K-means clustering
    """
```

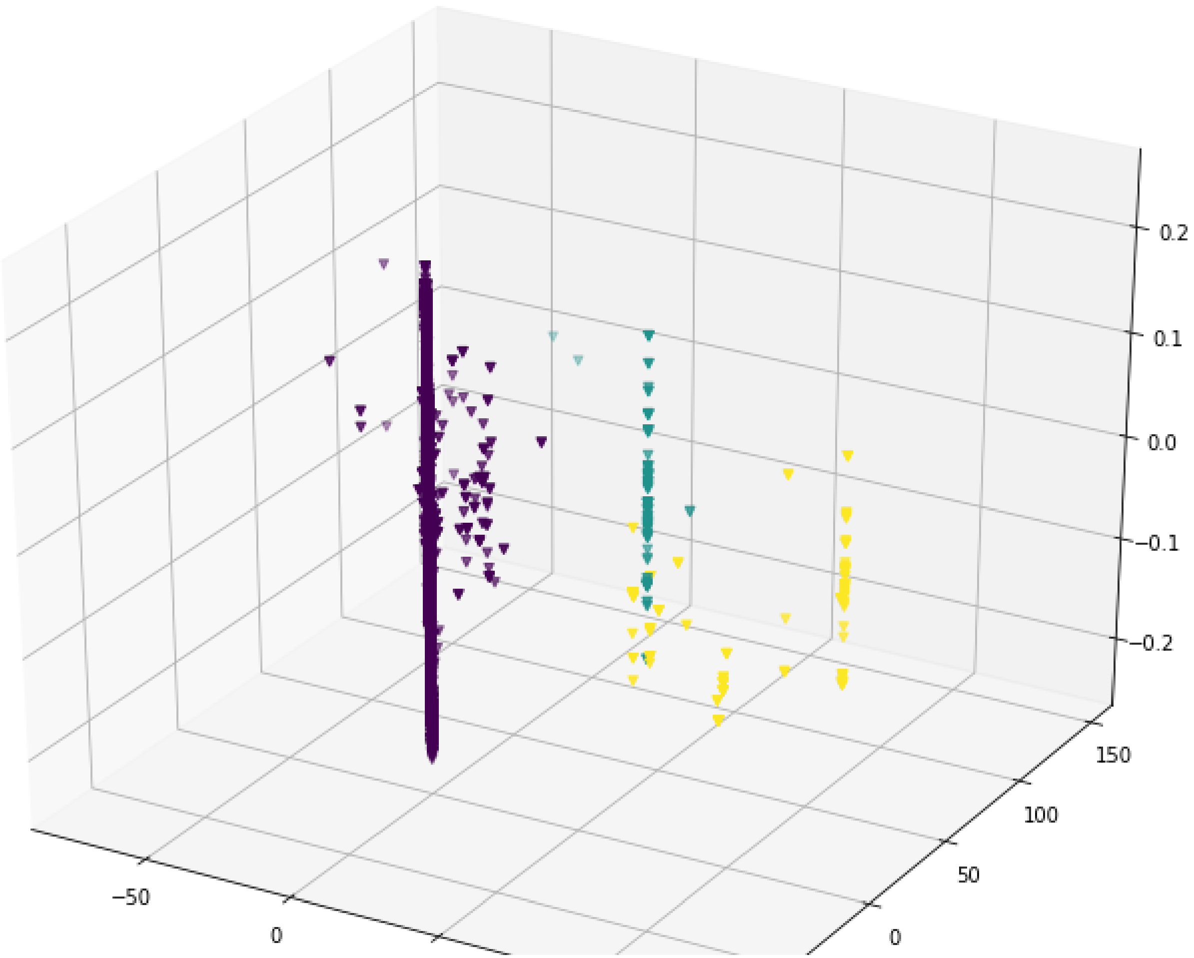
```
    def __init__(self, X):
        self.X = X

    def cluster_plot(self):
        """
        Cluster plotting for different cluster algorithms
        """
        train          = StandardScaler().fit_transform(self.X)
        pca            = PCA(n_components=3)
        pca_component  = pca.fit_transform(self.X)
        fig = plt.figure(figsize=(10,8))
        sns.set_palette(sns.color_palette("cubehelix", 8))
        ax = Axes3D(fig)
        ax.scatter(pca_component[:,0].tolist(),pca_component[:,1].tolist(),pca_component[:,2].tolist(),c=self.labels,marker='v')
        ax.legend()
        plt.show()

    def dendogram(self):
        """
        This method plots dendogram for hierarchial clustering
        """


        plt.figure(figsize=(20, 7))
        dendrogram = sch.dendrogram(sch.linkage(self.X, method='ward'))
        plt.title("Dendograms")
        plt.axhline(linestyle='--', y=5)
        plt.show()

    def silhouette_scores(self):
        """
        This method plots silhouette_scores for k-means clustering to find optimal number of clusters
        """
        kmeans_models = [KMeans(n_clusters=k, random_state=42).fit(self.X) for k in range(1, 10)]
        silhouette_scores = [silhouette_score(self.X, model.labels_) for model in kmeans_models[1:]]
        print(colored("The maximum silhouette score is %0.02f at the cluster number %d\n" % (np.max(silhouette_scores),(silhouette_s
        plt.figure(figsize=(16, 8))
```

```
        plt.plot(range(2, 10), silhouette_scores, "bo-")
        plt.xlabel("$k$", fontsize=14)
        plt.ylabel("Silhouette score", fontsize=14)
        plt.show()


    def davies_bouldin_score(self):

        """
        Validation test to check score after clustering
        """
        print(colored("The davies bouldin score of the clustering is %0.002f\n" %(davies_bouldin_score(self.X, self.labels)),color =
        print()
        print(colored("The points in each cluster are : ",color = 'yellow', attrs=['bold']))
        print(collections.Counter(self.labels))


    def kmeans_clustering(self,k):

        """
        Performs k-means algorithm with given clusters 'k'
        Input  : The input to this algorithm is clusters
        Output : Output is clustering labels
        """

        print(colored("Performing K-means clustering with %d clusters\n"%k,color = 'yellow', attrs=['bold']))
        kmeans = KMeans(n_clusters=k, random_state=0, n_init=10, max_iter=100).fit(self.X)
        self.labels = kmeans.labels_
        self.davies_bouldin_score()
        print()
        print(colored("The k-means inertia is %0.002f\n" %(kmeans.inertia_),color = 'red', attrs=['bold']))
        self.cluster_plot()
        return self.labels , kmeans.cluster_centers_,kmeans


    def hierarchial_clustering(self,k):

        """
```

```
        Performs hierarchial clustering with given clusters'k'
        Input  : The input to this algorithm are clusters
        Output : Output is clustering labels
        """

        print(colored("Performing hierarchial clustering",color = 'yellow', attrs=['bold']))
        self.clustering = AgglomerativeClustering(affinity='euclidean', linkage='ward').fit(self.X)
        self.labels = self.clustering.labels_
        self.davies_bouldin_score()
        print()
        print(colored("The number of cluster centers formed are %d\n" %(self.clustering.n_clusters_),color = 'red', attrs=['bold']))
        self.cluster_plot()
        return self.labels


    def DBscan_clustering(self,d,s):

        """
        Performs DBscan clustering with given distance 'd' and 'sample size 's'
        Input  : The input to this algorithm is clustering distance and samples
        Output : Output is clustering labels
        """
        print(colored("Performing agglomerative clustering",color = 'yellow', attrs=['bold']))
        self.clustering = DBSCAN(eps=d,min_samples=s,metric = 'euclidean').fit(self.X)
        self.labels = self.clustering.labels_
        self.davies_bouldin_score()
        print()
        print(colored("The number of cluster centers formed are %d\n"%len(np.unique(self.labels)),color = 'red', attrs=['bold']))
        self.cluster_plot()
        return self.labels
```

```
# clustering the data
c = clustering(df.iloc[:,1:5].values)
k = c.kmeans_clustering(3)
```

```
Performing K-means clustering with 3 clusters

The davies bouldin score of the clustering is 0.52


The points in each cluster are :
Counter({0: 626959, 2: 128, 1: 123})

The k-means inertia is 671214.07

No handles with labels found to put in legend.
```

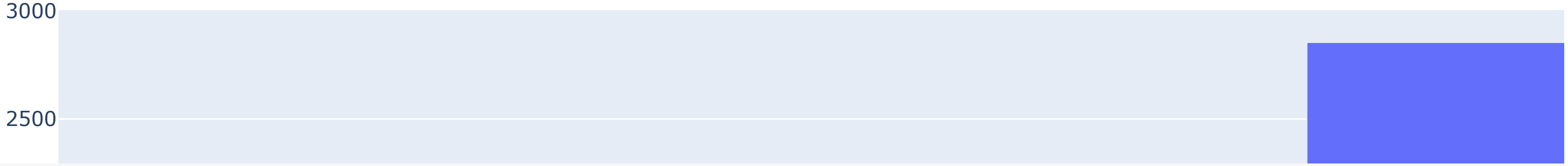


Clustering: Since its only city of Talin our geographies are also concentrated to one point.

```
# skipping due to RAM issues
#k = c.hierarchial_clustering(3)
```

```
# assign the clusters
df["clusters"] = k[0]
```

```
# lets see cluster wise analysis
t = pd.DataFrame(df.groupby(['clusters']).agg({'ride_value':'median'})).reset_index().sort_values(by=['ride_value'],ascending=False)
visualisation(t).bar_plot('clusters','ride_value')
```




```
# mean distance in each clusters
t = pd.DataFrame(df.groupby(['clusters']).agg({'distance_travelled':'median'})).reset_index().sort_values(by=['distance_travelled'],a
visualisation(t).bar_plot('clusters','distance_travelled')
```

Observation: These two graphs show us that most of the ride value is concentrated with far apart points in geography signifying as distance increases ride value also tends to increase.

Classification

- We will try to create a classification model that will classify our order value into different `categories:` low, medium and high. These categories will assist the rider in picking up a order towards higher ride value.
- The **target variable** defined in our case will be named as `category` and we have set it based on the density distribution of `ride_mile_expense`.

```
# ride mile expense categorisation
q1 = df['ride_mile_expense'].quantile(0.33)
q2 = df['ride_mile_expense'].quantile(0.66)

df['category'] = 'Low'

# defining other categories
for i in range(len(df)):
    if q1 < df['ride_mile_expense'][i] < q2:
               df['category'][i] = 'Medium'
    elif df['ride_mile_expense'][i] > q2:
               df['category'][i] = 'High'
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:12: SettingWithCopyWarning:


A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
```

```
## Preprocessing data for classification

# one hot encode categorical data
one_hot_encode = df[['clusters','week_day','start_hour']]

# define one hot encoding
encoder = OneHotEncoder()

# transform data
onehot = encoder.fit_transform(one_hot_encode).toarray()

# get the cols
cols = encoder.get_feature_names().tolist()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning:

Function get_feature_names is deprecated; get_feature_names is deprecated in 1.0 and will be removed in 1.2. Please use get_fea
```

```
# in case if we need to save
df.to_csv("sample.csv")
```

```
# get the train variables
X = df[['distance_travelled']].values
cols.append('distance_travelled')
```

```
# concatenate the values
X = np.concatenate((X, onehot), axis=1)

# categorical encode the target variable
encoder = LabelEncoder()
y = encoder.fit_transform(df['category'])
```

```
# summarize the class distribution
counter = Counter(y)
print(counter)
```

```
Counter({0: 213251, 1: 206981, 2: 206978})
```

```
# Skipping SMOTE as classes are mostly balanced
'''
# transform the dataset with class distribution
sm = SMOTE(k_neighbors=2)
X, y = sm.fit_resample(X,y)
# summarize the new class distribution
counter = Counter(y)
print(counter)
'''
```

```
'\n# transform the dataset with class distribution\nsm = SMOTE(k_neighbors=2)\nX, y = sm.fit_resample(X,y)\n# summarize the ne
w class distribution\ncounter = Counter(y)\nprint(counter)\n'
```

```
# splitting the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)
```

```
# classification class and modelling
'''
Class function with all models and their hyperparameters which can be called by the user

'''
```

```
class classification:

    def __init__(self, X_train,y_train,X_test,y_test,cols):
           """
           Pass the dataframe in the form of train and test seperately with column names for feature info

           """
           ## get the vectors
           self.X_train,self.y_train,self.X_test,self.y_test = X_train,y_train,X_test,y_test

           self.labels       = np.unique(self.y_train).tolist()

           # reshape the values
           self.y_train = self.y_train.reshape(-1,1)
           self.y_test  = self.y_test.reshape(-1,1)

           # dropping label col as it is it target variable
           self.column_name  = cols

           # the labels
           self.labels = np.unique(self.y_train).tolist()

           # Making a dataframe to store model results
           self.results = pd.DataFrame(columns=['Model','Datatype','Precision','Recall','Accuracy','F1_score'])

           # Getting the cols
           self.column_name = cols

    def confusion_matrix(self, pred, y_test):
            #"For plotting matrix plot in kernel returns in the form of plot"
            cm = confusion_matrix(y_test, pred)
            print(colored("The confusion matrix is :",color = 'green', attrs=['bold']))
            print(cm)
            fig = plt.figure()
            ax = fig.add_subplot(111)
            cax = ax.matshow(cm)
            plt.title('Confusion matrix of the classifier')
```

```
            fig.colorbar(cax)
            #ax.set_xticklabels([''] + labels)
            #ax.set_yticklabels([''] + labels)
            plt.xlabel('$Predicted$')
            plt.ylabel('$True$')
            plt.show()


    def calc_metrics_class(self,model_name,pred,y_test,label):
        # Print's the model's performance overall
        print(colored("Generating the results wait for it....",color = 'red', attrs=['bold']))
        # Lets see the classification metrics
        precision = precision_score(pred, y_test,average='weighted')
        recall = recall_score(pred,y_test,average='weighted')
        f1 = f1_score(pred,y_test,average='weighted')
        accuracy = accuracy_score(pred,y_test)

        self.results = self.results.append({'Model':model_name,'Datatype':label,'Precision':precision,'Recall':recall,'Accuracy':accu
                                            'F1_score':f1}, ignore_index=True)




        # print classification report
        print(classification_report(y_test,pred))
        # Visualise the results in dataset of "test"
        print(colored("The results of your model are:",color = 'yellow', attrs=['bold']))
        print(display(HTML(self.results.to_html())))
        self.confusion_matrix(pred, y_test)

    def feature_importance_lime(self, model, i = 0):
         '''
         This method shows the feature importance of each set of params in getting the result
         It can be called by word index number with the model
         '''
         print()
         print("The feature importance viz for data index  %d is:"%i)
         explainer = lime.lime_tabular.LimeTabularExplainer(self.X_train,
```

```
                      feature_names= self.column_name,
                      class_names=self.labels)

        # Predict the result of model
        predict_fn = lambda x: model.predict_proba(x).astype(float)

        # Visualise it to pictorially view
        exp = explainer.explain_instance(self.X_test[i], predict_fn, num_features=10)
        exp.show_in_notebook(show_all=False)

    def feature_importance_info(self, model):
        # Time to see feature importance
        print(colored("The feature importance is :",color = 'green', attrs=['bold']))

        # feature importance of the models
        feature_importance = pd.DataFrame()
        feature_importance['variable'] = self.column_name
        feature_importance['importance'] = model.feature_importances_

        # feature_importance values in descending order
        print(feature_importance.sort_values(by='importance', ascending=False).head(15))

        # By lime
        self.feature_importance_lime(model)

    def random_forest(self, feature_importance = False):

        print(colored("Performing modelling for Random forest",color = 'green', attrs=['bold']))
        # Create Random Forest Model
        rf_model = RandomForestClassifier(random_state=1)
        # Specifying hyperparams for the search
        param_grid = {
                        'n_estimators': [75],
                        'max_features': [0.1],
                        'min_samples_split': [2]
                        }
        # Fit the model and find best hyperparams
```

```
        grid_model = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=5, n_jobs=-1)
        grid_model.fit(self.X_train,self.y_train)

        # Fit the model with best params
        print("Best parameters =", grid_model.best_params_)
        model_clf = rf_model.set_params(**grid_model.best_params_)
        model_clf.fit(self.X_train, self.y_train)

        # Time to test the model
        # Time to test the model for test set
        print(colored("Test results for test set",color = 'yellow', attrs=['bold']))
        self.pred = model_clf.predict(self.X_test)
        self.calc_metrics_class("Random Forest",self.pred, y_test = self.y_test, label = 'test')


        # Let's see feature importance if called
        if feature_importance:
            self.feature_importance_info(model_clf)

        # Returning model
        return model_clf


    def gradient_boost(self, feature_importance = False):

        print(colored("Performing modelling for Gradient Boosting",color = 'green', attrs=['bold']))
        # Create gradient boosting
        GradBoostClasCV = GradientBoostingClassifier(random_state=42)

        # Specifying hyperparams for the search
        model_params = {
                          "max_depth": [10],
                          "subsample": [0.9],
                          "n_estimators":[200,300],
                          "learning_rate": [0.01]
                        }
        # Fit the model and find best hyperparams
```

```
        grid_model = GridSearchCV(estimator=GradBoostClasCV, param_grid=model_params, cv=5, n_jobs=-1)
        grid_model.fit(self.X_train,self.y_train)

        # Fit the model with best params
        print("Best parameters =", grid_model.best_params_)
        model_clf = GradBoostClasCV.set_params(**grid_model.best_params_)
        model_clf.fit(self.X_train, self.y_train)

        # Time to test the model
        # Time to test the model for test set
        print(colored("Test results for test set",color = 'yellow', attrs=['bold']))
        self.pred = model_clf.predict(self.X_test)
        self.calc_metrics_class("Gradient Boosting",self.pred, y_test = self.y_test, label = 'test')


        # Let's see feature importance if called
        if feature_importance:
            self.feature_importance_info(model_clf)

        # Returning model
        return model_clf

    def logistic_regression(self):

        print(colored("Performing modelling for Logistic Regression",color = 'blue', attrs=['bold']))
        # Create logistic regression
        logistic = linear_model.LogisticRegression(max_iter = 1000)

        # Create regularization penalty space
        penalty = ['l2']

        # Create regularization hyperparameter space
        C = np.logspace(0, 4, 10)

        # Create hyperparameter options and fot it into grid search
        hyperparameters = dict(C=C, penalty=penalty)
        grid_model = GridSearchCV(estimator=logistic, param_grid=hyperparameters, cv=5, verbose=0, n_jobs=-1)
```

```
        # Fit the model and find best hyperparams
        grid_model.fit(self.X_train,self.y_train)
        print("Best parameters =", grid_model.best_params_)

        # Fit the model with best params
        model_clf = logistic.set_params(**grid_model.best_params_)
        model_clf.fit(self.X_train, self.y_train)

        # Time to test the model for test set
        print(colored("Test results for test set",color = 'yellow', attrs=['bold']))
        self.pred = model_clf.predict(self.X_test)
        self.calc_metrics_class("Logistic Regression",self.pred, y_test = self.y_test, label = 'test')

        # Returning model
        return model_clf


    def XG_Boost(self, feature_importance = False):

        print(colored("Performing modelling for XG Boost Classifier",color = 'blue', attrs=['bold']))
        # Create XGB Classifier
        xg = XGBClassifier(nthread=4, seed=42)
        model_params = {
                          'max_depth':   [75],
                          'n_estimators':  [200, 300],
                          'learning_rate': [0.01]
                          }
        # Fit the model and find best hyperparams
        grid_model = GridSearchCV(estimator=xg, param_grid=model_params, cv=5,scoring = 'accuracy', n_jobs=-1)
        grid_model.fit(self.X_train,self.y_train)

        # Fit the model with best params
        print("Best parameters =", grid_model.best_params_)
        model_clf = xg.set_params(**grid_model.best_params_)
        model_clf.fit(self.X_train, self.y_train)
```

```
        # Time to test the model
        # Time to test the model for test set
        print(colored("Test results for test set",color = 'yellow', attrs=['bold']))
        self.pred = model_clf.predict(self.X_test)
        self.calc_metrics_class("XG Boost",self.pred, y_test = self.y_test, label = 'test')

        # Let's see feature importance if called
        if feature_importance:
            self.feature_importance_info(model_clf)

        # Returning model
        return model_clf
```

```
# Intialise the model with the instance variable
c = classification(X_train,y_train,X_test,y_test,cols)
```

```
# simple model
model = c.logistic_regression()
```

```
Performing modelling for Logistic Regression
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: DataConversionWarning:

A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using rav

Best parameters = {'C': 1.0, 'penalty': 'l2'}
/usr/local/lib/python3.7/dist-packages/sklearn/utils/validation.py:993: DataConversionWarning:

A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using rav

Test results for test set
Generating the results wait for it....
              precision    recall  f1-score   support

           0       0.37      0.47      0.41     42811
           1       0.39      0.57      0.46     41190
           2       0.37      0.09      0.14     41441

    accuracy                           0.38    125442
   macro avg       0.37      0.38      0.34    125442
weighted avg       0.37      0.38      0.34    125442

The results of your model are:
```

	Model	Datatype	Precision	Recall	Accuracy	F1_score
0	Logistic Regression	test	0.488436	0.377553	0.377553	0.415035

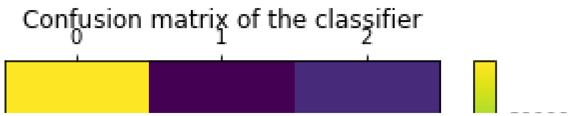
```
None
The confusion matrix is :
[[20068 19107  3636]
```

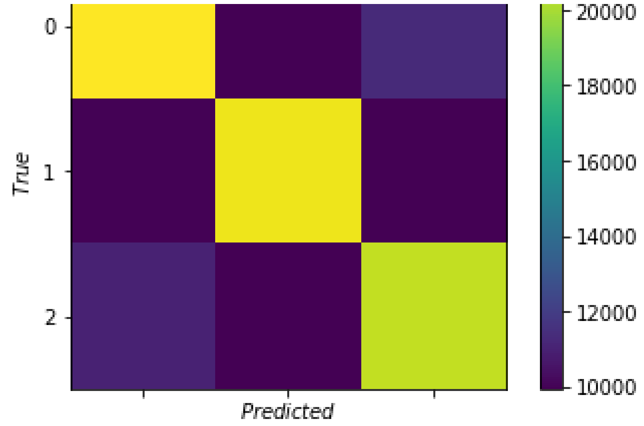
```
# Random Forest model
model = c.random_forest(feature_importance=True)
```

```
Performing modelling for Random forest
/usr/local/lib/python3.7/dist-packages/joblib/externals/loky/process_executor.py:705: UserWarning:

A worker stopped while some jobs were given to the executor. This can be caused by a too short worker timeout or by a memory le

/usr/local/lib/python3.7/dist-packages/sklearn/model_selection/_search.py:926: DataConversionWarning:

A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using rave

Best parameters = {'max_features': 0.1, 'min_samples_split': 2, 'n_estimators': 75}
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:123: DataConversionWarning:

A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples,), for example using rave

Test results for test set
Generating the results wait for it....
              precision    recall  f1-score   support

           0       0.51      0.50      0.50     42811
           1       0.52      0.52      0.52     41190
           2       0.49      0.49      0.49     41441

    accuracy                           0.50    125442
   macro avg       0.50      0.50      0.50    125442
weighted avg       0.50      0.50      0.50    125442

The results of your model are:
```

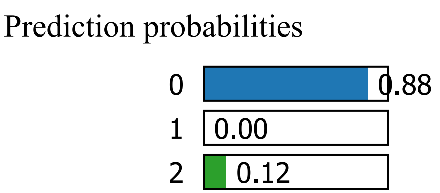
	Model	Datatype	Precision	Recall	Accuracy	F1_score
0	Logistic Regression	test	0.488436	0.377553	0.377553	0.415035
1	Random Forest	test	0.503786	0.503811	0.503811	0.503795

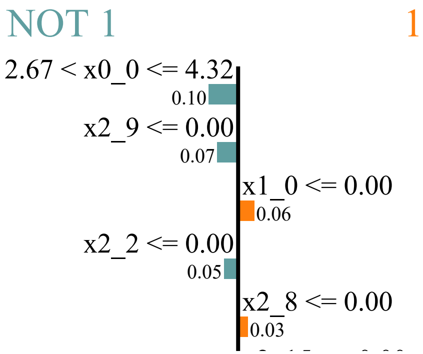
```
None
The confusion matrix is :
[[21519  9964 11328]
 [10023 21229  9938]
 [11034  9956 20451]]
```






```
The feature importance is :
   variable  importance
0      x0_0    0.996678
1      x0_1    0.000249
6      x1_3    0.000149
7      x1_4    0.000148
8      x1_5    0.000145
4      x1_1    0.000138
9      x1_6    0.000136
10     x2_0    0.000131
5      x1_2    0.000129
22    x2_12    0.000098
33    x2_23    0.000093
16     x2_6    0.000092
26    x2_16    0.000092
32    x2_22    0.000087
18     x2_8    0.000085

The feature importance viz for data index  0 is:
```





Feature	Value
x0_0	4.26
x2_9	0.00
x1_0	0.00
x2_2	0.00
x2_8	0.00
x2_15	0.00
x2_14	1.00

x2_15 <= 0.00
0.03

x0_2 0.00
x2_19 0.00

Final conclusion

0.03

- Based on our recommendation from the algorithm we have created a **base model** which would assist a rider to drive towards highest ride value.
- Deployment of the model can be briefly done with the help of REST API's with the help of POST and GET request. We can create an end point with SWAGGER where the user can input the data. Based on the preprocessing of features we can predict the models response. The models response can then be rendered as request and delivered to the application interface. For database(DB) POSTGRE can be used and commplete deployment can be done in Heroku.
- For **Randomised A/B testing** the trained data can be tested against the data with most ride value. The prediction given by the model can be considered in **Control Group** which can be tested against the data in **Target Group.** These two groups will each contain 50-50% of our data.

The null hypothesis defined in our case would be :

1. **H0:** The model predicts and diverts the rider towards highest ride value.
2. **H1:** The highest ride value is not the same as the model predicted.

Once we test our model in real world data we can do a test of significance. If it lies within a confidence interval we can accept our null hypothesis. If it rejects our null hypothesis then our model's prediction isn't true we need to tune and train our model again.