**The Project Outline**

1. **The Problem Statement**
2. **Description of Data**
3. **Application Architecture and module division**
4. **Code Explanation**
5. **Data Ingestion**
6. **Data Validation**
7. **Data Preprocessing**
8. **Model Selection**
9. **Model Tuning**
10. **Prediction**
11. **Logging Framework**
12. **Deployment**

**The Problem Statement:**

1. Do a complete EDA in the python notebook file
2. Build a solution design architecture for end to end solution starting from data ingestion to deployment with a detail documentation.
3. Deploythe end to end automated solution to GCP.
4. Create a user interface for bulk testing uploaded through excel sheet and for a single record entry both.
5. Maintain log for each and every prediction request into any database
6. Create a low level documentation for end to end solution and deployment
7. Define a retraining approach in your documentation
8. Create an end to end video of the working of the project

**To build an application which predicts restaurant ratings based on given attributes.**

**Data Set Information:**

The dataset is about bankruptcy prediction of Polish companies. The data was collected from Emerging Markets Information Service (EMIS, [Web Link]), which is a database containing information on emerging markets around the world. The bankrupt companies were analysed in the period 2000-2012, while the still operating companies were evaluated from 2007 to 2013.

Basing on the collected data five classification cases were distinguished, that depends on the forecasting period:

- **1stYear** the data contains financial rates from 1st year of the forecasting period and corresponding class label that indicates bankruptcy status after 5 years. The data contains 7027 instances (financial statements), 271 represents bankrupted companies, 6756 firms that did not bankrupt in the forecasting period.

- **2ndYear** the data contains financial rates from 2nd year of the forecasting period and corresponding class label that indicates bankruptcy status after 4 years. The data contains 10173 instances (financial statements), 400 represents bankrupted companies, 9773 firms that did not bankrupt in the forecasting period.

- **3rdYear** the data contains financial rates from 3rd year of the forecasting period and corresponding class label that indicates bankruptcy status after 3 years. The data contains 10503 instances (financial statements), 495 represents bankrupted companies, 10008 firms that did not bankrupt in the forecasting period.

- **4thYear** the data contains financial rates from 4th year of the forecasting period and corresponding class label that indicates bankruptcy status after 2 years. The data contains 9792 instances (financial statements), 515 represents bankrupted companies, 9277 firms that did not bankrupt in the forecasting period.

- **5thYear** the data contains financial rates from 5th year of the forecasting period and corresponding class label that indicates bankruptcy status after 1 year. The data contains 5910 instances (financial statements), 410 represents bankrupted companies, 5500 firms that did not bankrupt in the forecasting period.

**Attribute Information:**

**X1** net profit / total assets

**X2** total liabilities / total assets

**X3** working capital / total assets

**X4** current assets / short-term liabilities

**X5** [(cash + short-term securities + receivables - short-term liabilities) / (operating expenses - depreciation)] \* 365

**X6** retained earnings / total assets

**X7** EBIT / total assets

**X8** book value of equity / total liabilities

**X9** sales / total assets

**X10** equity / total assets

**X11** (gross profit + extraordinary items + financial expenses) / total assets

**X12** gross profit / short-term liabilities

**X13** (gross profit + depreciation) / sales

**X14** (gross profit + interest) / total assets

**X15** (total liabilities \* 365) / (gross profit + depreciation)

**X16** (gross profit + depreciation) / total liabilities

**X17** total assets / total liabilities

**X18** gross profit / total assets

**X19** gross profit / sales

**X20** (inventory \* 365) / sales

**X21** sales (n) / sales (n-1)

**X22** profit on operating activities / total assets

**X23** net profit / sales

**X24** gross profit (in 3 years) / total assets

**X25** (equity - share capital) / total assets

**X26** (net profit + depreciation) / total liabilities

**X27** profit on operating activities / financial expenses

**X28** working capital / fixed assets

**X29** logarithm of total assets

**X30** (total liabilities - cash) / sales

**X31** (gross profit + interest) / sales

**X32** (current liabilities \* 365) / cost of products sold

**X33** operating expenses / short-term liabilities

**X34** operating expenses / total liabilities

**X35** profit on sales / total assets

**X36** total sales / total assets

**X37** (current assets - inventories) / long-term liabilities

**X38** constant capital / total assets

**X39** profit on sales / sales

**X40** (current assets - inventory - receivables) / short-term liabilities

**X41** total liabilities / ((profit on operating activities + depreciation) \* (12/365))

**X42** profit on operating activities / sales

**X43** rotation receivables + inventory turnover in days

**X44** (receivables \* 365) / sales

**X45** net profit / inventory

**X46** (current assets - inventory) / short-term liabilities

**X47** (inventory \* 365) / cost of products sold

**X48** EBITDA (profit on operating activities - depreciation) / total assets

**X49** EBITDA (profit on operating activities - depreciation) / sales

**X50** current assets / total liabilities

**X51** short-term liabilities / total assets

**X52** (short-term liabilities \* 365) / cost of products sold)

**X53** equity / fixed assets

**X54** constant capital / fixed assets

**X55** working capital

**X56** (sales - cost of products sold) / sales

**X57** (current assets - inventory - short-term liabilities) / (sales - gross profit - depreciation)

**X58** total costs /total sales

**X59** long-term liabilities / equity

**X60** sales / inventory

**X61** sales / receivables

**X62** (short-term liabilities \*365) / sales

**X63** sales / short-term liabilities

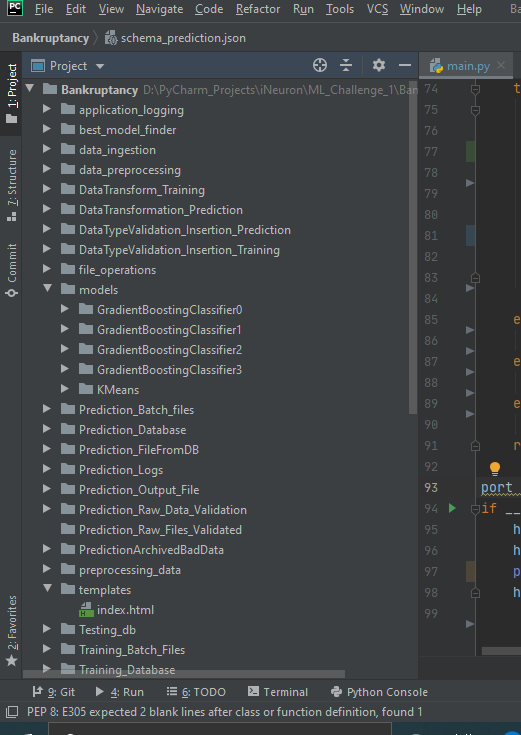
**X64** sales / fixed assets

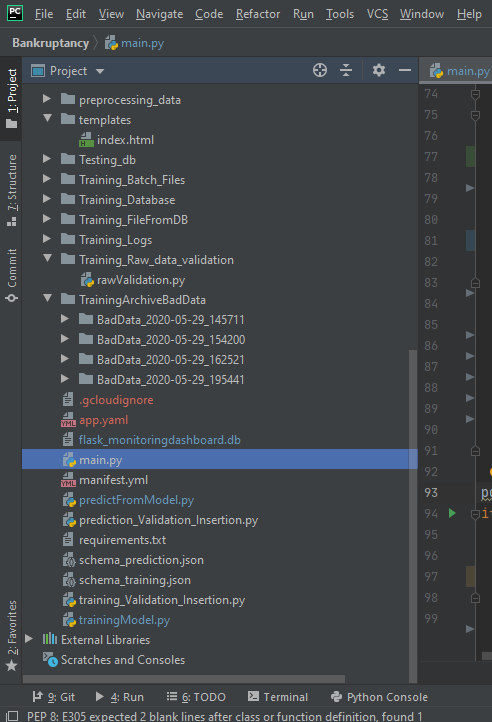
**Class -** 0 did not get bankrupt/ 1 - got bankrupt

**Archiecture Application flow and module division**



**Project Folder Structure.**





**Training**

**Schema**

Apart from training files, we also require a "schema" file from the client, which contains all the relevant information about the training files such as:

Name of the files, Length of Date value in FileName, Length of Time value in FileName, Number of Columns, Name of the Columns, and their datatype.

* Schema for training in json format

{ "SampleFileName": "bankruptancy\_08012020\_120000.csv",  
 "LengthOfDateStampInFile": 8,  
 "LengthOfTimeStampInFile": 6,  
 "NumberofColumns" : 65,  
 "ColName": {  
 "Attr1": " float",  
 "Attr2": " float",  
 "Attr3": " float",  
 "Attr4": " float",

……..

"Attr61": " float",  
 "Attr62": " float",  
 "Attr63": " float",  
 "Attr64": " float",  
 "class" : "Integer"  
 }  
}

**Data Validation**

In this step, we perform different sets of validation on the given set of training files.

1. Name Validation- We validate the name of the files based on the given name in the schema file. We have created a regex pattern as per the name given in the schema file to use for validation. After validating the pattern in the name, we check for the length of date in the file name as well as the length of time in the file name. If all the values are as per requirement, we move such files to "Good\_Data\_Folder" else we move such files to "Bad\_Data\_Folder."
2. Number of Columns - We validate the number of columns present in the files, and if it doesn't match with the value given in the schema file, then the file is moved to "Bad\_Data\_Folder."
3. Name of Columns - The name of the columns is validated and should be the same as given in the schema file. If not, then the file is moved to "Bad\_Data\_Folder".
4. The datatype of columns - The datatype of columns is given in the schema file. This is validated when we insert the files into Database. If the datatype is wrong, then the file is moved to "Bad\_Data\_Folder".
5. Null values in columns - If any of the columns in a file have all the values as NULL or missing, we discard such a file and move it to "Bad\_Data\_Folder".

**Code Snippet**:

with open(self.schema\_path, 'r') as f:  
 dic = json.load(f)  
 f.close()  
pattern = dic['SampleFileName']  
LengthOfDateStampInFile = dic['LengthOfDateStampInFile']  
LengthOfTimeStampInFile = dic['LengthOfTimeStampInFile']  
column\_names = dic['ColName']  
NumberofColumns = dic['NumberofColumns']  
  
file = open("Training\_Logs/valuesfromSchemaValidationLog.txt", 'a+')  
message ="LengthOfDateStampInFile:: %s" %LengthOfDateStampInFile + "\t" + "LengthOfTimeStampInFile:: %s" % LengthOfTimeStampInFile +"\t " + "NumberofColumns:: %s" % NumberofColumns + "\n"  
self.logger.log(file,message)  
  
file.close()

It will return the LengthOfDateStampInFile, LengthOfTimeStampInFile, column\_names, noofcolumns

# getting the regex defined to validate filename

regex = "[ bankruptancy]+['\\_'']+[\d\_]+[\d]+\.csv"

# validating filename of prediction files

# delete the directories for good and bad data in case last run was unsuccessful and folders were not deleted.  
self.deleteExistingBadDataTrainingFolder()  
self.deleteExistingGoodDataTrainingFolder()  
  
onlyfiles = [f for f in listdir(self.Batch\_Directory)]  
try:  
 # create new directories  
 self.createDirectoryForGoodBadRawData()  
 f = open("Training\_Logs/nameValidationLog.txt", 'a+')  
 for filename in onlyfiles:  
 if (re.match(regex, filename)):  
 splitAtDot = re.split('.csv', filename)  
 splitAtDot = (re.split('\_', splitAtDot[0]))  
 if len(splitAtDot[1]) == LengthOfDateStampInFile:  
 if len(splitAtDot[2]) == LengthOfTimeStampInFile:  
 shutil.copy("Training\_Batch\_Files/" + filename, "Training\_Raw\_files\_validated/Good\_Raw")  
 self.logger.log(f,"Valid File name!! File moved to GoodRaw Folder :: %s" % filename)  
  
 else:  
 shutil.copy("Training\_Batch\_Files/" + filename, "Training\_Raw\_files\_validated/Bad\_Raw")  
 self.logger.log(f,"Invalid File Name!! File moved to Bad Raw Folder :: %s" % filename)  
 else:  
 shutil.copy("Training\_Batch\_Files/" + filename, "Training\_Raw\_files\_validated/Bad\_Raw")  
 self.logger.log(f,"Invalid File Name!! File moved to Bad Raw Folder :: %s" % filename)  
 else:  
 shutil.copy("Training\_Batch\_Files/" + filename, "Training\_Raw\_files\_validated/Bad\_Raw")  
 self.logger.log(f, "Invalid File Name!! File moved to Bad Raw Folder :: %s" % filename)  
  
 f.close()

# validating column length in the file

f = open("Training\_Logs/columnValidationLog.txt", 'a+')  
self.logger.log(f,"Column Length Validation Started!!")  
for file in listdir('Training\_Raw\_files\_validated/Good\_Raw/'):  
 csv = pd.read\_csv("Training\_Raw\_files\_validated/Good\_Raw/" + file)  
 if csv.shape[1] == NumberofColumns:  
 pass  
 else:  
 shutil.move("Training\_Raw\_files\_validated/Good\_Raw/" + file, "Training\_Raw\_files\_validated/Bad\_Raw")  
 self.logger.log(f, "Invalid Column Length for the file!! File moved to Bad Raw Folder :: %s" % file)

# validating if any column has all values missing

for file in listdir('Training\_Raw\_files\_validated/Good\_Raw/'):  
 csv = pd.read\_csv("Training\_Raw\_files\_validated/Good\_Raw/" + file)  
 count = 0  
 for columns in csv:  
 if (len(csv[columns]) - csv[columns].count()) == len(csv[columns]):  
 count+=1  
 shutil.move("Training\_Raw\_files\_validated/Good\_Raw/" + file,  
 "Training\_Raw\_files\_validated/Bad\_Raw")  
 self.logger.log(f,"Invalid Column Length for the file!! File moved to Bad Raw Folder :: %s" % file)  
 break  
 if count==0:  
 csv.rename(columns={"Unnamed: 0": "serial"}, inplace=True)  
 csv.to\_csv("Training\_Raw\_files\_validated/Good\_Raw/" + file, index=None, header=True)

# Starting Data Transforamtion!!

# replaceMissingWithNull

onlyfiles = [f for f in listdir(self.goodDataPath)]  
for file in onlyfiles:  
 csv = pandas.read\_csv(self.goodDataPath+"/" + file, low\_memory=False)  
 # replace '?' with nan  
 csv.replace('?', np.nan, inplace=True)  
 # convert object into float for all attr  
 csv.iloc[:, :-1] = csv.iloc[:, :-1].astype(np.float64)  
 # fill NULL  
 csv.fillna('NULL', inplace=True)  
  
 csv.to\_csv(self.goodDataPath+ "/" + file, index=None, header=True)

**Data Insertion in Database**

1**) Database Creation and connection -** Create a database with the given name passed. If the database is already created, open the connection to the database.

conn = sqlite3.connect(self.path+DatabaseName+'.db')

2) **Table creation in the database -** Table with name - "Good\_Data", is created in the database for inserting the files in the "Good\_Data\_Folder" based on given column names and datatype in the schema file. If the table is already present, then the new table is not created and new files are inserted in the already present table as we want training to be done on new as well as old training files.

**Code Snippet**:

# create database with given name, if present open the connection! Create table with columns given in schema

conn = self.dataBaseConnection(DatabaseName)  
c=conn.cursor()  
c.execute("SELECT count(name) FROM sqlite\_master WHERE type = 'table'AND name = 'Good\_Raw\_Data'")  
if c.fetchone()[0] ==1:  
 conn.close()  
 file = open("Training\_Logs/DbTableCreateLog.txt", 'a+')  
 self.logger.log(file, "Tables created successfully!!")  
 file.close()  
  
 file = open("Training\_Logs/DataBaseConnectionLog.txt", 'a+')  
 self.logger.log(file, "Closed %s database successfully" % DatabaseName)  
 file.close()  
  
else:  
  
 for key in column\_names.keys():  
 type = column\_names[key]  
  
 #in try block we check if the table exists, if yes then add columns to the table  
 # else in catch block we will create the table  
 try:  
 conn.execute('ALTER TABLE Good\_Raw\_Data ADD COLUMN "{column\_name}" {dataType}'.format(column\_name=key,dataType=type))  
 except:  
 conn.execute('CREATE TABLE Good\_Raw\_Data ({column\_name} {dataType})'.format(column\_name=key, dataType=type))

conn.close()

**3) Insertion of files in the table** - All the files in the "Good\_Data\_Folder" are inserted in the above-created table. If any file has invalid data type in any of the columns, the file is not loaded in the table and is moved to "Bad\_Data\_Folder".

# insert csv files in the table

for file in onlyfiles:  
 try:  
 with open(goodFilePath+'/'+file, "r") as f:  
 next(f)  
 reader = csv.reader(f, delimiter="\n")  
 for line in enumerate(reader):  
 for list\_ in (line[1]):  
 try:  
 conn.execute('INSERT INTO Good\_Raw\_Data values ({values})'.format(values=(list\_)))  
 self.logger.log(log\_file," %s: File loaded successfully!!" % file)  
 conn.commit()  
 except Exception as e:  
 raise e

**Model Training**

1. **Data Export from Db –**

The data in a stored database is exported as a CSV file to be used for model training.

# export data in table to csvfile

self.fileFromDb = 'Training\_FileFromDB/'  
self.fileName = 'InputFile.csv'  
log\_file = open("Training\_Logs/ExportToCsv.txt", 'a+')  
try:  
 conn = self.dataBaseConnection(Database)  
 sqlSelect = "SELECT \* FROM Good\_Raw\_Data"  
 cursor = conn.cursor()  
  
 cursor.execute(sqlSelect)  
  
 results = cursor.fetchall()  
 # Get the headers of the csv file  
 headers = [i[0] for i in cursor.description]  
  
 #Make the CSV ouput directory  
 if not os.path.isdir(self.fileFromDb):  
 os.makedirs(self.fileFromDb)  
  
 # Open CSV file for writing.  
 csvFile = csv.writer(open(self.fileFromDb + self.fileName, 'w', newline=''),delimiter=',', lineterminator='\r\n',quoting=csv.QUOTE\_ALL, escapechar='\\')  
  
 # Add the headers and data to the CSV file.  
 csvFile.writerow(headers)  
 csvFile.writerows(results)  
  
 self.logger.log(log\_file, "File exported successfully!!!")  
 log\_file.close()

**2) Data Preprocessing**

1. Check for null values in the columns. If present, impute the null values using the Simple imputer with strategy mean.

imputer = SimpleImputer(missing\_values=np.nan, strategy='mean')  
self.new\_array = imputer.fit\_transform(self.data) # impute the missing values  
  
# convert the nd-array returned in the step above to a Dataframe  
self.new\_data=pd.DataFrame(data=self.new\_array, columns=self.data.columns)

2. Drop The Duplicate value if it is present in dataset

# drop the duplicates values  
data.drop\_duplicates(inplace = True)

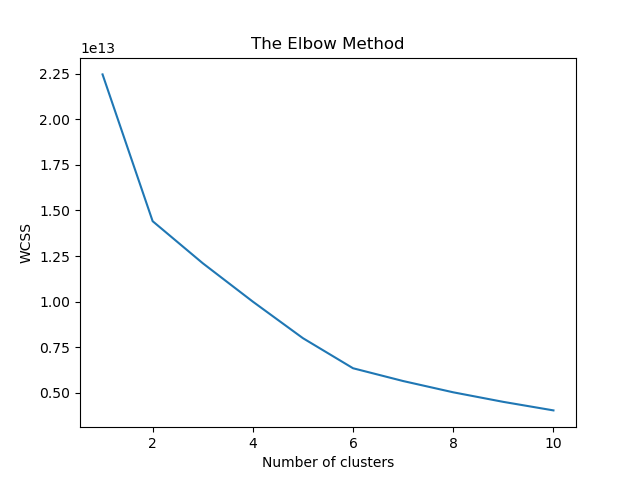
3. Droping the numerical outliers values

# drop the numerical\_outliers values  
# Constrains will contain `True` or `False` depending on if it is a value below the threshold.  
constrains = self.data.iloc[:, :-1].select\_dtypes(include=[np.number]) \  
 .apply(lambda x: np.abs(stats.zscore(x)) < z\_thresh) \  
 .all(axis=1)  
  
# Drop (inplace) values set to be rejected  
self.data.drop(self.data.index[~constrains], inplace=True)

**3) Clustering –**

a) KMeans algorithm is used to create clusters in the preprocessed data. The optimum number of clusters is selected by plotting the elbow plot, and for the dynamic selection of the number of clusters, we are using "KneeLocator" function. The idea behind clustering is to implement different algorithms

wcss=[] # initializing an empty list  
try:  
 for i in range (1,11):  
 kmeans=KMeans(n\_clusters=i,init='k-means++',random\_state=42) # initializing the KMeans object  
 kmeans.fit(data) # fitting the data to the KMeans Algorithm  
 wcss.append(kmeans.inertia\_)  
 plt.plot(range(1,11),wcss) # creating the graph between WCSS and the number of clusters  
 plt.title('The Elbow Method')  
 plt.xlabel('Number of clusters')  
 plt.ylabel('WCSS')  
 #plt.show()  
 plt.savefig('preprocessing\_data/K-Means\_Elbow.PNG') # saving the elbow plot locally  
 # finding the value of the optimum cluster programmatically  
 self.kn = KneeLocator(range(1, 11), wcss, curve='convex', direction='decreasing')  
 self.logger\_object.log(self.file\_object, 'The optimum number of clusters is: '+str(self.kn.knee)+' . Exited the elbow\_plot method of the KMeansClustering class')  
 return self.kn.knee



Here I got the number of cluster 4 that can be seen through log files in ModelTrainingLog or in model folder i.e. number of model has been created.

b) To train data in different clusters. The Kmeans model is trained over preprocessed data and the model is saved for further use in prediction.

# Divide the data into clusters and save model

self.kmeans = KMeans(n\_clusters=number\_of\_clusters, init='k-means++', random\_state=42)  
#self.data = self.data[~self.data.isin([np.nan, np.inf, -np.inf]).any(1)]  
self.y\_kmeans=self.kmeans.fit\_predict(data) # divide data into clusters  
  
self.file\_op = file\_methods.File\_Operation(self.file\_object,self.logger\_object)  
self.save\_model = self.file\_op.save\_model(self.kmeans, 'KMeans') # saving the KMeans model to directory  
 # passing 'Model' as the functions need three parameters  
  
self.data['Cluster']=self.y\_kmeans # create a new column in dataset for storing the cluster information

**4) Model Selection –**

After clusters are created, we find the best model for each cluster. We are using Three algorithms, "Random forest Classifier ", “Gradient boosting Classifier” and “Xtreme boosting Classifier”. For each cluster, the algorithms are passed with the best parameters derived from GridSearch. We calculate the Rsquared scores for both models and select the model with the best score. Similarly, the model is selected for each cluster. All the models for every cluster are saved for use in prediction.

1. **Random forest Classifier**

# get\_best\_params\_for\_random\_forest

# initializing with different combination of parameters  
self.param\_grid = {"n\_estimators": [10, 50, 100, 130],  
 "criterion": ['gini', 'entropy'],  
 "max\_depth": range(2, 4, 1),  
 "max\_features": ['auto', 'log2']}  
  
# Creating an object of the Grid Search class  
self.grid = GridSearchCV(estimator=self.clf, param\_grid=self.param\_grid, cv=5, verbose=3)  
# finding the best parameters  
self.grid.fit(train\_x, train\_y)  
# extracting the best parameters  
self.criterion = self.grid.best\_params\_['criterion']  
self.max\_depth = self.grid.best\_params\_['max\_depth']  
self.max\_features = self.grid.best\_params\_['max\_features']  
self.n\_estimators = self.grid.best\_params\_['n\_estimators']  
# creating a new model with the best parameters  
self.clf = RandomForestClassifier(n\_estimators=self.n\_estimators, criterion=self.criterion,  
 max\_depth=self.max\_depth, max\_features=self.max\_features)  
# training the mew model  
self.clf.fit(train\_x, train\_y)  
self.logger\_object.log(self.file\_object,  
 'Random Forest best params: ' + str(  
 self.grid.best\_params\_) + '. Exited the get\_best\_params\_for\_random\_forest method of the Model\_Finder class')  
return self.clf

1. **Gradient boosting Classifier**

# get\_best\_params\_for\_GradientBoostingClassifier

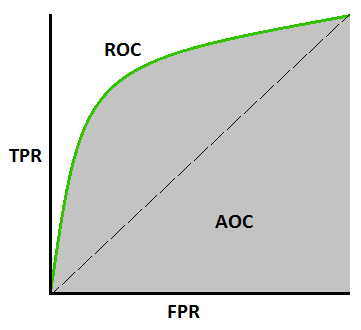
# initializing with different combination of parameters  
self.param\_grid\_gbc = {  
 "n\_estimators": [100, 200],  
 "max\_depth": range(5, 10, 3),  
 'learning\_rate': [0.1, 0.01, 0.5],  
 'min\_samples\_split': [i for i in range(10, 31, 15)],  
 'min\_samples\_leaf': [i for i in range(10, 31, 15)]  
}  
  
# Creating an object of the Grid Search class  
self.grid = GridSearchCV(self.gbc, self.param\_grid\_gbc, verbose=3, cv=5)  
# finding the best parameters  
self.grid.fit(train\_x, train\_y)  
  
# extracting the best parameters  
# self.criterion = self.grid.best\_params\_['criterion']  
self.max\_depth = self.grid.best\_params\_['max\_depth']  
self.n\_estimators = self.grid.best\_params\_['n\_estimators']  
self.learning\_rate = self.grid.best\_params\_['learning\_rate']  
self.min\_samples\_split = self.grid.best\_params\_['min\_samples\_split']  
self.min\_samples\_leaf = self.grid.best\_params\_['min\_samples\_leaf']  
  
# creating a new model with the best parameters  
self.gbc = GradientBoostingClassifier(random\_state=10, learning\_rate=self.learning\_rate,  
 max\_depth=self.max\_depth, n\_estimators=self.n\_estimators,  
 min\_samples\_split=self.min\_samples\_split, min\_samples\_leaf=self.min\_samples\_leaf)  
# training the mew model  
self.gbc.fit(train\_x, train\_y)  
self.logger\_object.log(self.file\_object,  
 'GradientBoostingClassifier best params: ' + str(  
 self.grid.best\_params\_) + '. Exited the get\_best\_params\_for\_GradientBoostingClassifier method of the Model\_Finder class')  
return self.gbc

1. **Xtreme boosting Classifier**
2. # initializing with different combination of parameters  
   self.param\_grid\_xgboost = {  
    "n\_estimators": [400, 200, 300],  
    "criterion": ['gini', 'entropy'],  
    "max\_depth": range(5, 10, 2),  
    "objective": ['binary:logistic', 'binary:logitraw'],  
    'learning\_rate': [0.1, 0.01, 0.5]  
     
     
   }  
   # Creating an object of the Grid Search class  
   self.grid= GridSearchCV(XGBClassifier(),self.param\_grid\_xgboost, verbose=3,cv=5)  
   # finding the best parameters  
   self.grid.fit(train\_x, train\_y)  
     
   # extracting the best parameters  
   self.criterion = self.grid.best\_params\_['criterion']  
   self.max\_depth = self.grid.best\_params\_['max\_depth']  
   self.n\_estimators = self.grid.best\_params\_['n\_estimators']  
   self.objective = self.grid.best\_params\_['objective']  
   self.learning\_rate = self.grid.best\_params\_['learning\_rate']  
     
   # creating a new model with the best parameters  
   self.xgb = XGBClassifier(objective=self.objective, learning\_rate=self.learning\_rate, criterion=self.criterion, max\_depth=self.max\_depth,n\_estimators= self.n\_estimators, n\_jobs=-1 )  
   # training the mew model  
   self.xgb.fit(train\_x, train\_y)  
   self.logger\_object.log(self.file\_object,  
    'XGBoost best params: ' + str(  
    self.grid.best\_params\_) + '. Exited the get\_best\_params\_for\_xgboost method of the Model\_Finder class')  
   return self.xgb

**AUC-ROC Curve**

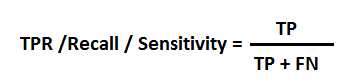
AUC - ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents degree or measure of separability. It tells how much model is capable of distinguishing between classes. Higher the AUC, better the model is at predicting 0s as 0s and 1s as 1s. By analogy, Higher the AUC, better the model is at distinguishing between patients with disease and no disease.

The ROC curve is plotted with TPR against the FPR where TPR is on y-axis and FPR is on the x-axis.

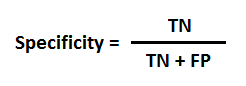


**Defining the terms used in AUC And ROC Curve**

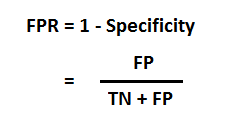
1. **TPR (True Positive Rate)/Recall / Sensitivity**



1. **Specificity**



1. **FPR**



self.xgboost= self.get\_best\_params\_for\_xgboost(train\_x,train\_y)  
self.prediction\_xgboost = self.xgboost.predict(test\_x) # Predictions using the XGBoost Model  
  
if len(test\_y.unique()) == 1: #if there is only one label in y, then roc\_auc\_score returns error. We will use accuracy in that case  
 self.xgboost\_score = accuracy\_score(test\_y, self.prediction\_xgboost)  
 self.logger\_object.log(self.file\_object, 'Accuracy for XGBoost:' + str(self.xgboost\_score)) # Log AUC  
else:  
 self.xgboost\_score = roc\_auc\_score(test\_y, self.prediction\_xgboost) # AUC for XGBoost  
 self.logger\_object.log(self.file\_object, 'AUC for XGBoost:' + str(self.xgboost\_score)) # Log AUC  
  
# create best model for GradientBoostingClassifier  
self.gbc = self.get\_best\_params\_for\_GradientBoostingClassifier(train\_x, train\_y)  
self.prediction\_gbc = self.gbc.predict(test\_x) # Predictions using the GradientBoostingClassifier Model  
  
if len(test\_y.unique()) == 1: # if there is only one label in y, then roc\_auc\_score returns error. We will use accuracy in that case  
 self.gbc\_score = accuracy\_score(test\_y, self.prediction\_gbc)  
 self.logger\_object.log(self.file\_object, 'Accuracy for GradientBoostingClassifier:' + str(self.gbc\_score)) # Log AUC  
else:  
 self.gbc\_score = roc\_auc\_score(test\_y, self.prediction\_gbc) # AUC for XGBoost  
 self.logger\_object.log(self.file\_object, 'AUC for GradientBoostingClassifier:' + str(self.gbc\_score)) # Log AUC  
  
# create best model for Random Forest  
self.random\_forest = self.get\_best\_params\_for\_random\_forest(train\_x, train\_y)  
self.prediction\_random\_forest = self.random\_forest.predict(test\_x) # prediction using the Random Forest Algorithm  
  
if len(test\_y.unique()) == 1: # if there is only one label in y, then roc\_auc\_score returns error. We will use accuracy in that case  
 self.random\_forest\_score = accuracy\_score(test\_y, self.prediction\_random\_forest)  
 self.logger\_object.log(self.file\_object, 'Accuracy for RF:' + str(self.random\_forest\_score))  
else:  
 self.random\_forest\_score = roc\_auc\_score(test\_y,  
 self.prediction\_random\_forest) # AUC for Random Forest  
 self.logger\_object.log(self.file\_object, 'AUC for RF:' + str(self.random\_forest\_score))  
  
#comparing the three models  
if(self.gbc\_score < self.xgboost\_score and self.random\_forest\_score < self.xgboost\_score):  
 return 'XGBoost',self.xgboost  
elif(self.gbc\_score < self.random\_forest\_score and self.random\_forest\_score > self.xgboost\_score):  
 return 'RandomForest', self.random\_forest  
else:  
 return 'GradientBoostingClassifier',self.gbc\_score

**Prediction Data Description**

Client will send the data in multiple set of files in batches at a given location. Data will contain restaurants ratings in 64 columns.( bankruptancy\_08012020\_120000.csv)

Apart from prediction files, we also require a "schema" file from client which contains all the relevant information about the training files such as:

Name of the files, Length of Date value in FileName, Length of Time value in FileName, Number of Columns, Name of the Columns and their datatype

* Schema for Prediction in json format

{ "SampleFileName": "bankruptancy\_08012020\_120000.csv",  
 "LengthOfDateStampInFile": 8,  
 "LengthOfTimeStampInFile": 6,  
 "NumberofColumns" : 64,  
 "ColName": {  
 "Attr1": " float",  
 "Attr2": " float",  
 "Attr3": " float",  
 "Attr4": " float",

……..

"Attr61": " float",  
 "Attr62": " float",  
 "Attr63": " float",  
 "Attr64": " float"  
 }  
}

**Data Validation**

In this step, we perform different sets of validation on the given set of training files.

1. Name Validation- We validate the name of the files based on the given name in the schema file. We have created a regex pattern as per the name given in the schema file to use for validation. After validating the pattern in the name, we check for the length of date in the file name as well as the length of time in the file name. If all the values are as per requirement, we move such files to "Good\_Data\_Folder" else we move such files to "Bad\_Data\_Folder."
2. Number of Columns - We validate the number of columns present in the files, and if it doesn't match with the value given in the schema file, then the file is moved to "Bad\_Data\_Folder."
3. Name of Columns - The name of the columns is validated and should be the same as given in the schema file. If not, then the file is moved to "Bad\_Data\_Folder".
4. The datatype of columns - The datatype of columns is given in the schema file. This is validated when we insert the files into Database. If the datatype is wrong, then the file is moved to "Bad\_Data\_Folder".
5. Null values in columns - If any of the columns in a file have all the values as NULL or missing, we discard such a file and move it to "Bad\_Data\_Folder".

**Code Snippet**:

with open(self.schema\_path, 'r') as f:  
 dic = json.load(f)  
 f.close()  
pattern = dic['SampleFileName']  
LengthOfDateStampInFile = dic['LengthOfDateStampInFile']  
LengthOfTimeStampInFile = dic['LengthOfTimeStampInFile']  
column\_names = dic['ColName']  
NumberofColumns = dic['NumberofColumns']  
  
file = open("Training\_Logs/valuesfromSchemaValidationLog.txt", 'a+')  
message ="LengthOfDateStampInFile:: %s" %LengthOfDateStampInFile + "\t" + "LengthOfTimeStampInFile:: %s" % LengthOfTimeStampInFile +"\t " + "NumberofColumns:: %s" % NumberofColumns + "\n"  
self.logger.log(file,message)  
  
file.close()

It will return the LengthOfDateStampInFile, LengthOfTimeStampInFile, column\_names, noofcolumns

# getting the regex defined to validate filename

regex = "[ bankruptancy]+['\\_'']+[\d\_]+[\d]+\.csv"

# validating filename of prediction files

self.deleteExistingBadDataTrainingFolder()  
self.deleteExistingGoodDataTrainingFolder()  
self.createDirectoryForGoodBadRawData()  
onlyfiles = [f for f in listdir(self.Batch\_Directory)]  
try:  
 f = open("Prediction\_Logs/nameValidationLog.txt", 'a+')  
 for filename in onlyfiles:  
 if (re.match(regex, filename)):  
 splitAtDot = re.split('.csv', filename)  
 splitAtDot = (re.split('\_', splitAtDot[0]))  
 if len(splitAtDot[1]) == LengthOfDateStampInFile:  
 if len(splitAtDot[2]) == LengthOfTimeStampInFile:  
 shutil.copy(self.Batch\_Directory + '/' + filename,  
 "Prediction\_Raw\_Files\_Validated/Good\_Raw")  
 self.logger.log(f, "Valid File name!! File moved to GoodRaw Folder :: %s" % filename)  
  
 else:  
 shutil.copy(self.Batch\_Directory + '/' + filename, "Prediction\_Raw\_Files\_Validated/Bad\_Raw")  
 self.logger.log(f, "Invalid File Name!! File moved to Bad Raw Folder :: %s" % filename)  
 else:  
 shutil.copy(self.Batch\_Directory + '/' + filename, "Prediction\_Raw\_Files\_Validated/Bad\_Raw")  
 self.logger.log(f, "Invalid File Name!! File moved to Bad Raw Folder :: %s" % filename)  
 else:  
 shutil.copy(self.Batch\_Directory + '/' + filename, "Prediction\_Raw\_Files\_Validated/Bad\_Raw")  
 self.logger.log(f, "Invalid File Name!! File moved to Bad Raw Folder :: %s" % filename)  
  
 f.close()

# validating if any column has all values missing

f = open("Prediction\_Logs/missingValuesInColumn.txt", 'a+')  
self.logger.log(f, "Missing Values Validation Started!!")  
  
for file in listdir('Prediction\_Raw\_Files\_Validated/Good\_Raw/'):  
 csv = pd.read\_csv("Prediction\_Raw\_Files\_Validated/Good\_Raw/" + file, low\_memory=False)  
 count = 0  
 for columns in csv:  
 if (len(csv[columns]) - csv[columns].count()) == len(csv[columns]):  
 count+=1  
 shutil.move("Prediction\_Raw\_Files\_Validated/Good\_Raw/" + file,  
 "Prediction\_Raw\_Files\_Validated/Bad\_Raw")  
 self.logger.log(f,"Invalid Column Length for the file!! File moved to Bad Raw Folder :: %s" % file)  
 break  
 if count==0:  
 csv.rename(columns={"Unnamed: 0": "Index"}, inplace=True)  
 csv.to\_csv("Prediction\_Raw\_Files\_Validated/Good\_Raw/" + file, index=None, header=True)

csv.rename(columns={"Unnamed: 0": "serial"}, inplace=True)  
 csv.to\_csv("Training\_Raw\_files\_validated/Good\_Raw/" + file, index=None, header=True)

# Starting Data Transforamtion!!

# replaceMissingWithNull

log\_file = open("Prediction\_Logs/dataTransformLog.txt", 'a+')  
onlyfiles = [f for f in listdir(self.goodDataPath)]  
for file in onlyfiles:  
 csv = pandas.read\_csv(self.goodDataPath + "/" + file, low\_memory=False)  
 # replace '?' with nan  
 csv.replace('?', np.nan, inplace=True)  
 # convert object into float for all attr  
 csv.iloc[:, :] = csv.iloc[:, :].astype(np.float64)  
 # fill NULL  
 csv.fillna('NULL', inplace=True)  
  
  
 csv.to\_csv(self.goodDataPath+ "/" + file, index=None, header=True)

**Data Insertion in Database**

**1) Database Creation and connection -** Create a database with the given name passed. If the database is already created, open the connection to the database.

conn = sqlite3.connect(self.path+DatabaseName+'.db')

**2) Table creation in the database -** Table with name - "Good\_Data", is created in the database for inserting the files in the "Good\_Data\_Folder" based on given column names and datatype in the schema file. If the table is already present, then the new table is not created and new files are inserted in the already present table as we want training to be done on new as well as old training files.

**Code Snippet**:

# create database with given name, if present open the connection! Create table with columns given in schema

conn = self.dataBaseConnection(DatabaseName)  
conn.execute('DROP TABLE IF EXISTS Good\_Raw\_Data;')  
  
for key in column\_names.keys():  
 type = column\_names[key]  
  
 # we will remove the column of string datatype before loading as it is not needed for training  
 #in try block we check if the table exists, if yes then add columns to the table  
 # else in catch block we create the table  
 try:  
   
 conn.execute('ALTER TABLE Good\_Raw\_Data ADD COLUMN "{column\_name}" {dataType}'.format(column\_name=key,dataType=type))  
 except:  
 conn.execute('CREATE TABLE Good\_Raw\_Data ({column\_name} {dataType})'.format(column\_name=key, dataType=type))  
  
conn.close()

**3) Insertion of files in the table -** All the files in the "Good\_Data\_Folder" are inserted in the above-created table. If any file has invalid data type in any of the columns, the file is not loaded in the table and is moved to "Bad\_Data\_Folder".

# insert csv files in the table

for file in onlyfiles:  
 try:  
 with open(goodFilePath+'/'+file, "r") as f:  
 next(f)  
 reader = csv.reader(f, delimiter="\n")  
 for line in enumerate(reader):  
 for list\_ in (line[1]):  
 try:  
 conn.execute('INSERT INTO Good\_Raw\_Data values ({values})'.format(values=(list\_)))  
 self.logger.log(log\_file," %s: File loaded successfully!!" % file)  
 conn.commit()  
 except Exception as e:  
 raise e

**Prediction**

**1) Data Export from Db -** The data in the stored database is exported as a CSV file to be used for prediction.

conn = self.dataBaseConnection(Database)  
sqlSelect = "SELECT \* FROM Good\_Raw\_Data"  
cursor = conn.cursor()  
  
cursor.execute(sqlSelect)  
  
results = cursor.fetchall()  
  
#Get the headers of the csv file  
headers = [i[0] for i in cursor.description]  
  
#Make the CSV ouput directory  
if not os.path.isdir(self.fileFromDb):  
 os.makedirs(self.fileFromDb)  
  
# Open CSV file for writing.  
csvFile = csv.writer(open(self.fileFromDb + self.fileName, 'w', newline=''),delimiter=',', lineterminator='\r\n',quoting=csv.QUOTE\_ALL, escapechar='\\')  
  
# Add the headers and data to the CSV file.  
csvFile.writerow(headers)  
csvFile.writerows(results)

**2) Data Preprocessing**

1. Check for null values in the columns. If present, impute the null values using the Simple imputer with strategy mean.

imputer = SimpleImputer(missing\_values=np.nan, strategy='mean')  
self.new\_array = imputer.fit\_transform(self.data) # impute the missing values  
  
# convert the nd-array returned in the step above to a Dataframe  
self.new\_data=pd.DataFrame(data=self.new\_array, columns=self.data.columns)

1. **Clustering –**

KMeans model created during training is loaded, and clusters for the preprocessed prediction data is predicted.

kmeans=file\_loader.load\_model('KMeans')  
clusters=kmeans.predict(data\_scaled)#drops the first column for cluster prediction  
data\_scaled['clusters']=clusters  
clusters=data\_scaled['clusters'].unique()

1. **Prediction –**

Based on the cluster number, the respective model is loaded and is used to predict the data for that cluster.

for i in clusters:  
 cluster\_data= data\_scaled[data\_scaled['clusters']==i]  
 cluster\_data = cluster\_data.drop(['clusters'],axis=1)  
 model\_name = file\_loader.find\_correct\_model\_file(i)  
 model = file\_loader.load\_model(model\_name)  
 result = []  
for val in (model.predict(data)):  
 result.append(val)

result = pandas.DataFrame(result,columns=['Predictions'])

**Once the prediction is made for all the clusters**, the predictions along with the original names before label encoder are saved in a CSV file at a given location and the location is returned to the client.

result.to\_csv("Prediction\_Output\_File/Predictions.csv", header=True, mode='a+') #appends result to prediction file

**Some Important files**

1. **requirements.txt** file consists of all the packages that you need to deploy the app in the cloud.

**Pip install -r requirements.txt**

1. **main.py** is the entry point of our application, where the flask server starts.

@app.route("/", methods=['GET'])  
@cross\_origin()  
def home():  
 return render\_template('index.html')  
  
@app.route("/predict", methods=['POST'])  
@cross\_origin()  
def predictRouteClient():  
 try:  
 # when we are sending the request in json used in postman  
 if request.json is not None:  
 path = request.json['filepath']  
 # Validation Part  
 pred\_val = Pred\_Validation(path) #object initialization  
 pred\_val.prediction\_validation() #calling the prediction\_validation function  
  
 # Training Part  
 pred = Prediction(path) #object initialization  
 # predicting for dataset present in database  
 path, json\_predictions = pred.predictionFromModel()  
  
 return Response("Prediction File created at !!!" + str(path) + 'and few of the predictions are ' + str(  
 json.loads(json\_predictions)))  
  
 # When we are sending the request in the form used in UserInterface (UI) part  
 elif request.form is not None:  
 path = request.form['filepath']  
 # Validation Part  
 pred\_val = Pred\_Validation(path) #object initialization  
 pred\_val.prediction\_validation() #calling the prediction\_validation function  
  
 # Training Part  
 pred = Prediction(path) #object initialization  
 # predicting for dataset present in database  
 path, json\_predictions = pred.predictionFromModel()  
 return Response("Prediction File created at !!!" + str(path) + 'and few of the predictions are ' +  
 str(json.loads(json\_predictions)))  
  
 except ValueError:  
 return Response("Error Occurred2! %s" %ValueError)  
 except KeyError:  
 return Response("Error Occurred3! %s" %KeyError)  
 except Exception as e:  
 return Response("Error Occurred4! %s" %e)  
  
  
  
@app.route("/train", methods=['POST'])  
@cross\_origin()  
def trainRouteClient():  
  
 try:  
 if request.json['folderPath'] is not None:  
 path = request.json['folderPath']  
 # Validation Part  
 train\_valObj = TrainValidation(path) #object initialization  
 train\_valObj.train\_validation()#calling the training\_validation function  
  
 # Training Part  
 trainModelObj = TrainModel() #object initialization  
 trainModelObj.trainingModel() #training the model for the files in the table  
  
 except ValueError:  
 return Response("Error Occurred! %s" % ValueError)  
 except KeyError:  
 return Response("Error Occurred! %s" % KeyError)  
 except Exception as e:  
 return Response("Error Occurred! %s" % e)  
 return Response("Training successfull!!")

1. This is the **predictionFromModel.py** file where the predictions take place based on the data we are giving input to the model.

class Prediction:  
  
 def \_\_init\_\_(self, path):  
 # path = r"Prediction\_Batch\_Files"  
 self.file\_object = open("Prediction\_Logs/Prediction\_Log.txt", 'a+')  
 self.log\_writer = logger.App\_Logger()  
 self.pred\_data\_val = Prediction\_Data\_validation(path)  
  
 def predictionFromModel(self):  
  
 try:  
 self.pred\_data\_val.deletePredictionFile() #deletes the existing prediction file from last run!  
 self.log\_writer.log(self.file\_object,'Start of Prediction')  
 data\_getter=data\_loader\_prediction.Data\_Getter\_Pred(self.file\_object,self.log\_writer)  
 data=data\_getter.get\_data()

1. **manifest.yml**:- This file contains the instance configuration, app name, and build pack language.

---  
applications:  
- name: restaurants  
 memory: 2GB  
 disk\_quota: 2GB  
 random-route: true  
 parameters:  
 memory: 2GB  
 buildpack: 'python\_buildpack'

1. **app.yaml** :- It contains the entry point of the app.

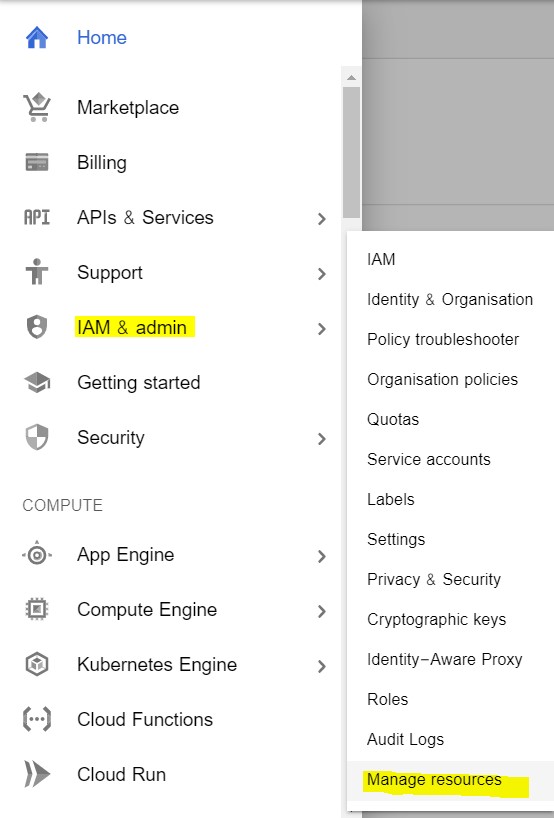
runtime: python37

**Deployment to G-cloud**

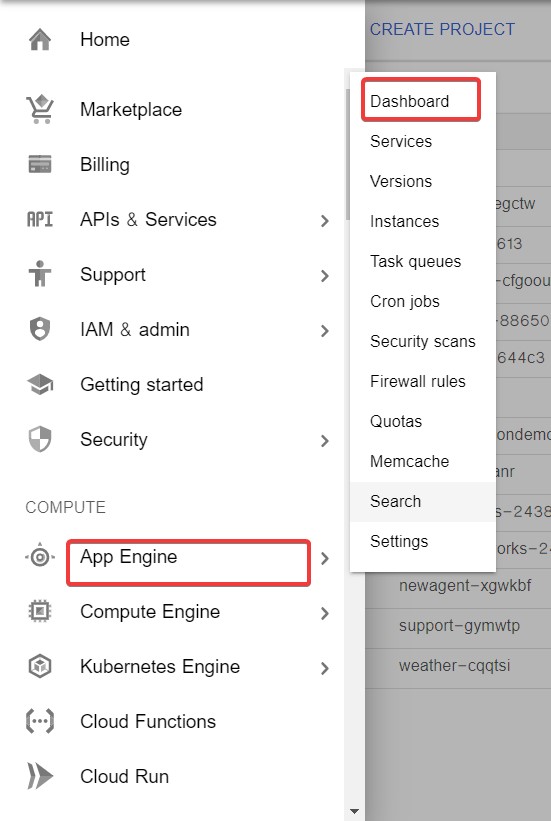
* Go to <https://cloud.google.com/>and create an account if already haven’t created one.

Then go to the console of your account.

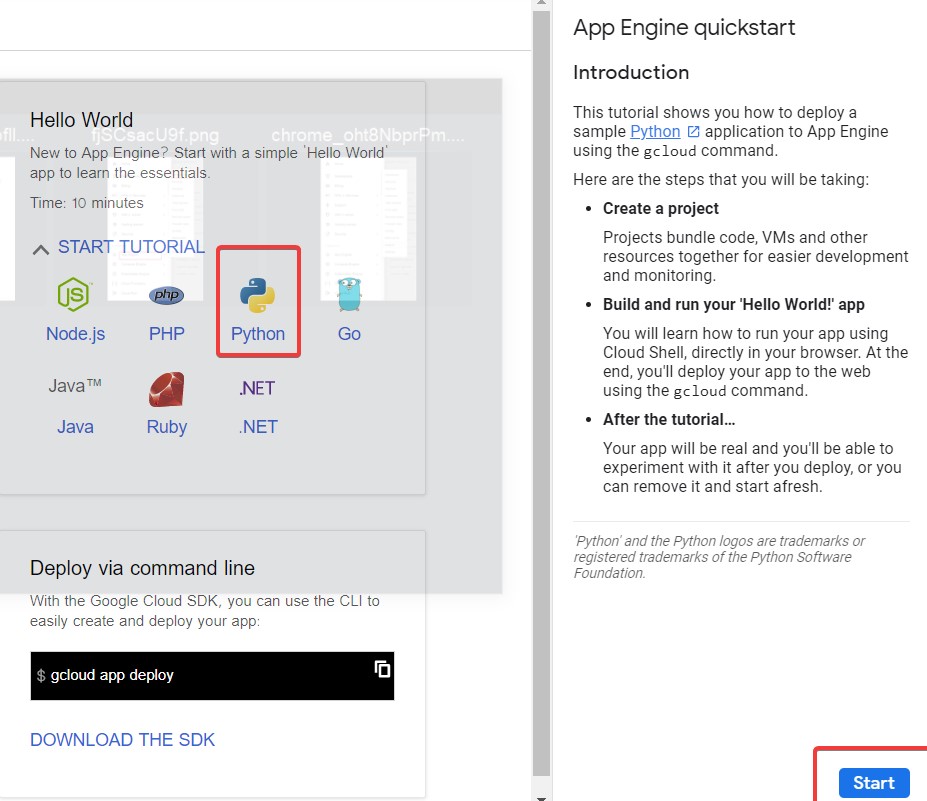
* Go to *IAM and admin*(highlighted) and click *manage resources*.



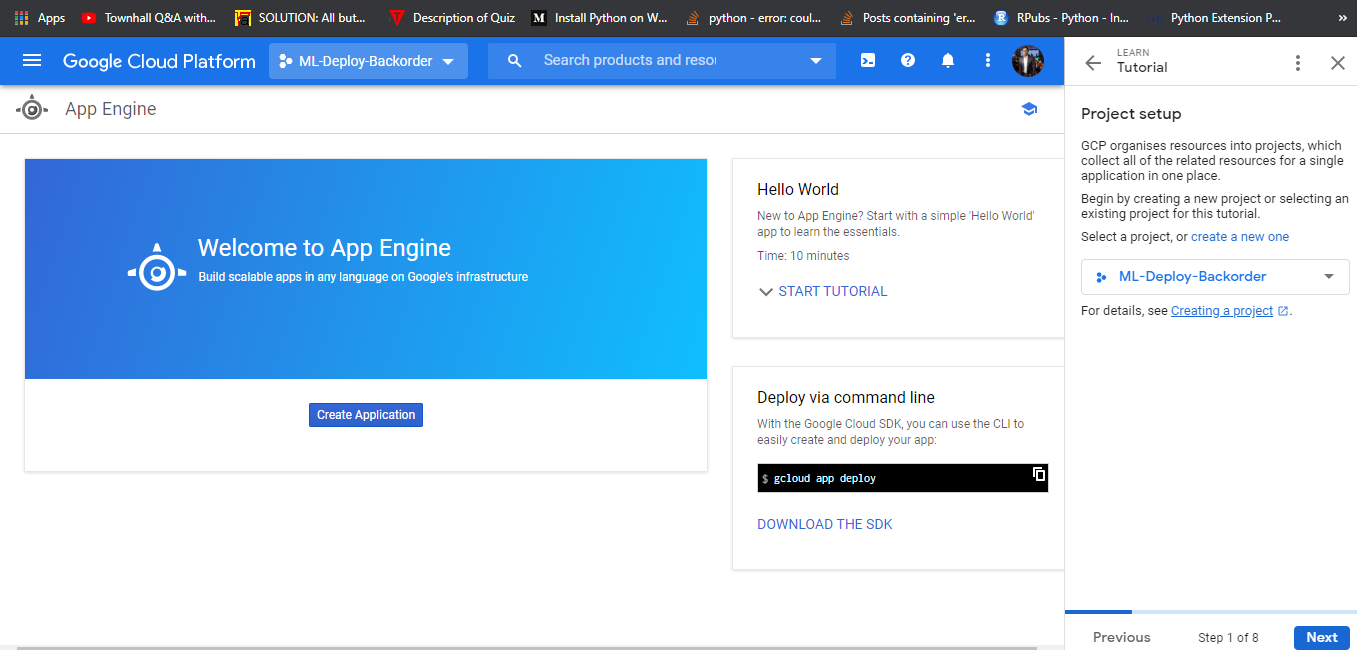
* Click *CREATE PROJECT* to create a new project for deployment.
* Once the project gets created, select *App Engine* and select *Dashboard.*



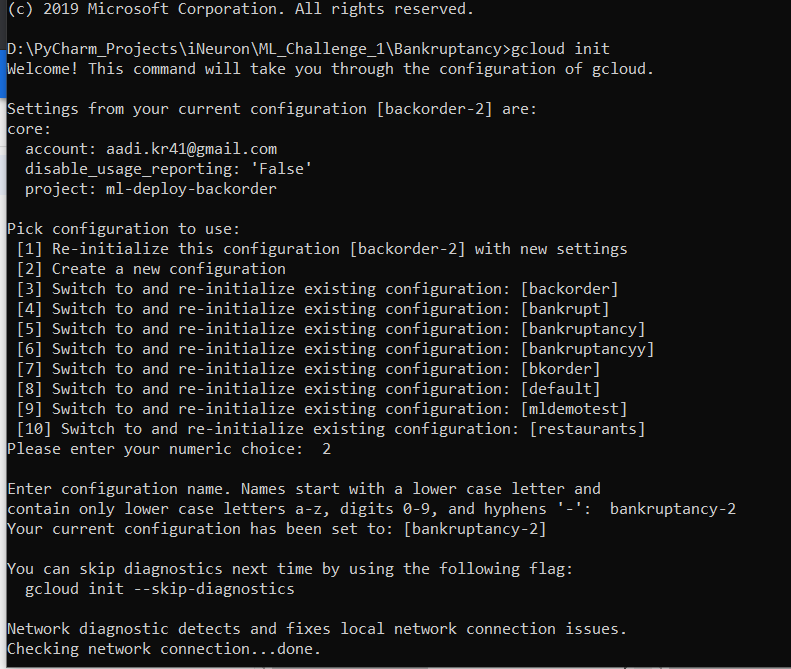
* Go to <https://dl.google.com/dl/cloudsdk/channels/rapid/GoogleCloudSDKInstaller.exe>to download the google cloud SDK to your machine.
* Click *Start Tutorial* on the screen and select Python app and click start.



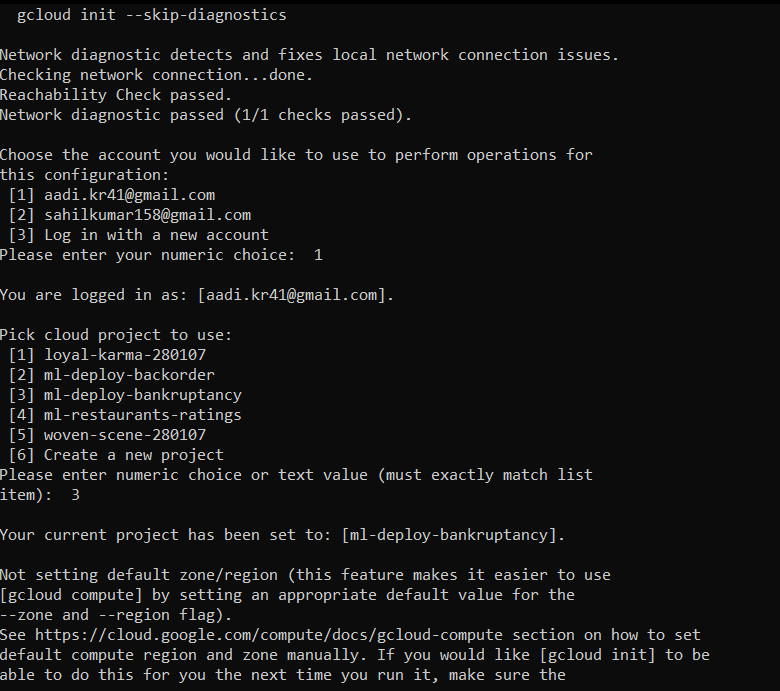
* Check whether the correct project name i.e ML-Deploy-Backorder is displayed and then click next.



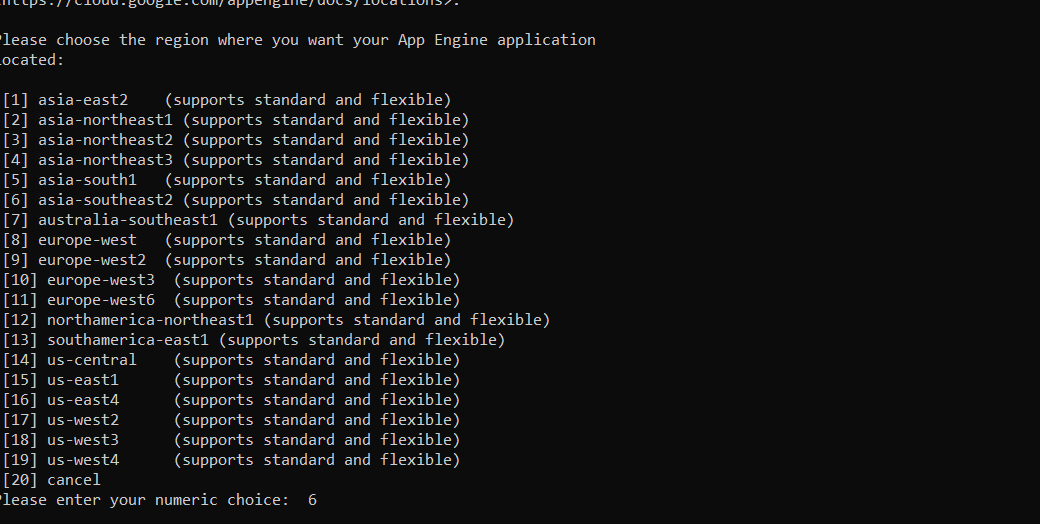
* Create a file ‘app.yaml’ and put ‘runtime: python37’ in that file.
* Create a ‘requirements.txt’ file by opening the command prompt/anaconda prompt, navigate to the project folder and enter the command ‘pip freeze > requirements.txt’. It is recommended to use separate environments for different projects.
* Your python application file should be called ‘main.py’. It is a GCP specific requirement.
* Open command prompt window, navigate to the project folder and enter the command *gcloud init* to initialise the gcloud context.
* It asks you to select from the list of available projects.



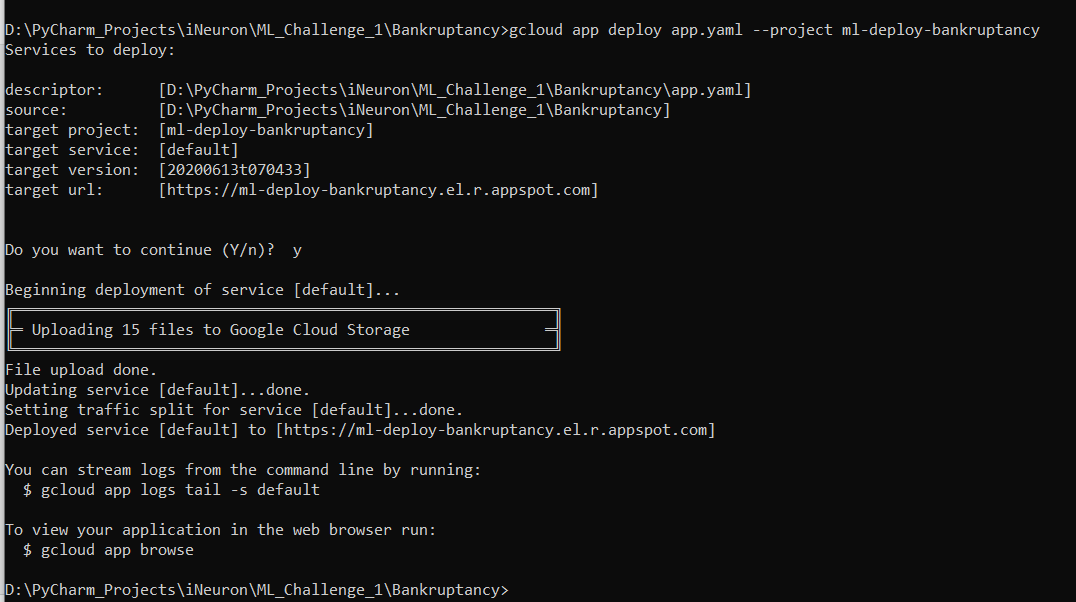
* I selected 2 option to create new configuration as bankruptancy-2.
* Select email id in which want to deploy the project.



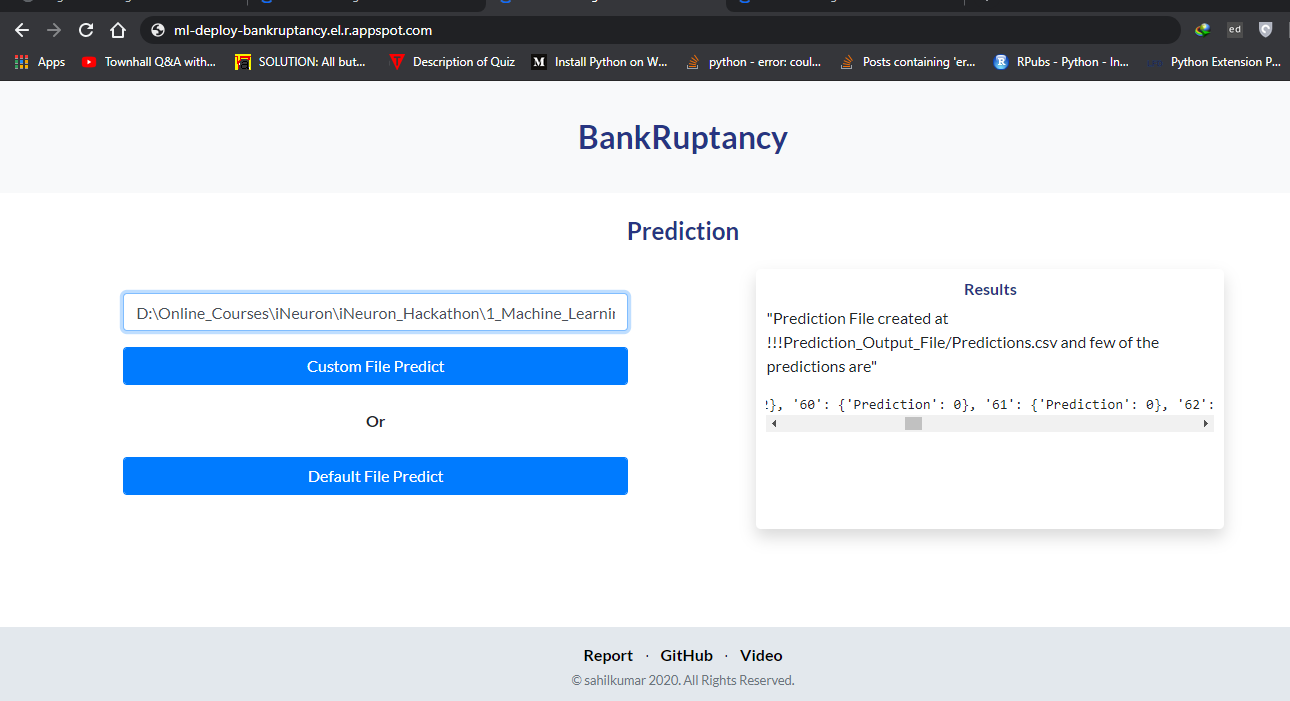
* Once the project name is selected i.e ml-deploy-bankruptancy , enter the command gcloud app deploy app.yaml --project ml-deploy-bankruptancy.
* After executing the above command, GCP will ask you to enter the region for your application. Choose the appropriate one.



* I selected 6 option as asia-southeast2.
* GCP will ask for the services to be deployed. Enter ‘y’ to deploy the services.



* And then it will give you the link for your app, and the deployed app looks like:



* To save money, go to App Engine > settings and disable your app.

