VEHICLE LOAN PREDICTION

Text

Description automatically generated

REPORT BY,

Mrunalini Devineni

Aihan Liu

Sara Sanchez

DATS6103

Professor Amir Jafari

December 6, 2021

TABLE OF CONTENTS

|  |  |
| --- | --- |
| *Introduction* | *3* |
| *Data Description* | ***4*** |
| *EDA* | ***5*** |
| *Data Preprocessing* | ***8*** |
| *Data Modelling* | ***10*** |
| *Conclusion* | ***11*** |
| *Reference* | ***12*** |
| *Appendix* | ***13*** |

INTRODUCTION

Financial institutions incur significant losses due to the default of Vehicle Loans. This situation has led to the constricting of vehicle loan underwriting and increased vehicle loan rejection rates. These institutions also raise the need for a better credit risk scoring model. By doing this, the institutions are trying to accurately predict the Probability of loanee defaulting on a vehicle loan on the due date. In this sense, the credit scores are significant as a tool that can decide which clients can take or not a credit, considering their unique characteristics. The scoring is also a helpful tool for the clients because this can avoid accepting a loan that will not be able to be paid in the future and, consequently, prevent having problems with financial institutions.

Furthermore, the scoring tool is helpful not only for cash loans but also for mortgages and car loans, so this is the main reason that motivated us to choose this topic because it is helpful for many kinds of businesses.

To develop this scoring project, we will analyze an L&T car loan company database from Kaggle. Then we are doing some preprocessing and cleaning of the data to apply the classification models like Decision tree, Random Forest, logistic, and Gradient Boosting. Finally, we will show the results by using the Pyqt5.

The following report will be organized: first, we will describe the data set, then the methodology to be used, and finally, the results and main conclusions.

DATA DESCRIPTION

For this study, we have chosen to analyze an L&T company dataset, in charge of cars' sale from kaggle.com. This dataset is like the one that financial institution must build the scoring models that allow them to forecast the approval or rejection of customers. This dataset does not require any cleaning and is equipped to fuel the analysis of this project. The base consists of 40 variables and 233 154 observations assessing a person's attributes ranging from demographic data (date of birth, etc.) and bureau data, like the amount of loan disbursed and the asset's cost.

The following Information regarding the loan and loanee are provided in the datasets:

* **Loanee Information** (Demographic data like age, Identity proof etc.)
* **Loan Information** (Disbursal details, loan to value ratio etc.)
* **Bureau data & history** (Bureau score, number of active accounts, the status of other loans, credit history etc.)

The dataset available on Kaggle contains:

* Train.csv: a base that contains the training data with details on loan as described in the last section.
* data\_dictionary.csv: a base containing a brief description of each variable

provided in the training and test set.

EXPLORATORY DATA ANALYSIS (EDA)

From the car loan database, we will develop the Exploratory data analysis (EDA). This kind of analysis permits a better understanding of the data and builds a better model. The target variable is the “loan default,” this gives us information related to the number of persons that defaulted, and for this database, the default ratio is around 27.7%.

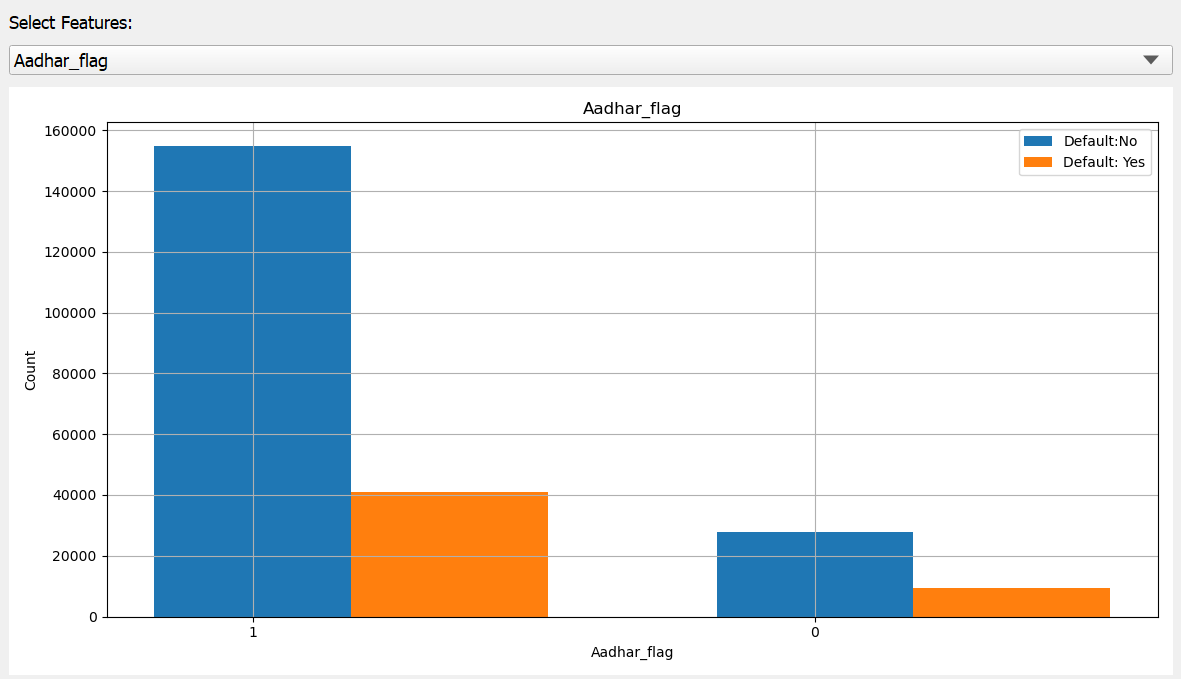
**Figure 1 – Loan\_deafult**

Chart

Description automatically generated

Another essential variable is the Aadhar flag. As this is a database from India, it is crucial to highlight that Aadhar is given to all the citizens from this country. A person who doesn’t have a citizen status will not have an Aadhar. The graph shows that people who have an Aadhar are more likely to be on default than others.

**Figure 2 – Aadhar flag**

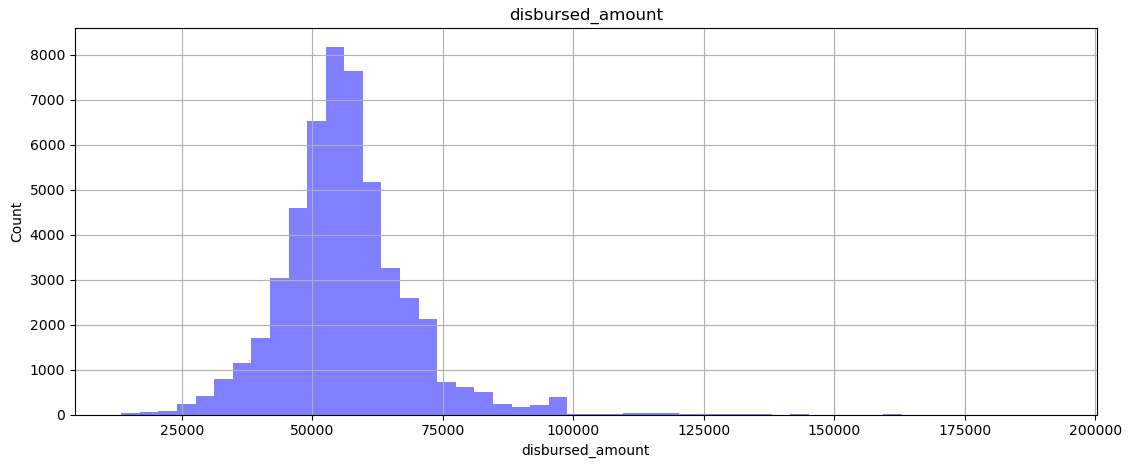
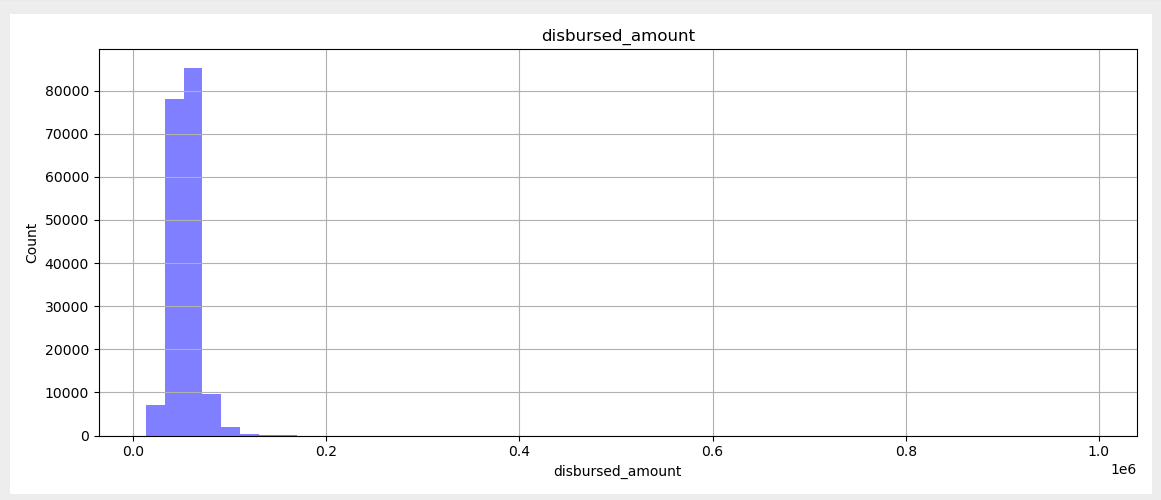


=With Aadhar

=No Aadhar

The disbursed amount is the financial institution's quantity to the borrower. Figure number three shows a significant difference in the amount between those without and those with default. Again, there is a concentration around shorter amounts. This is different from the former who have a normal distribution for the latter.

**Figure 3 – The disbursed amount**

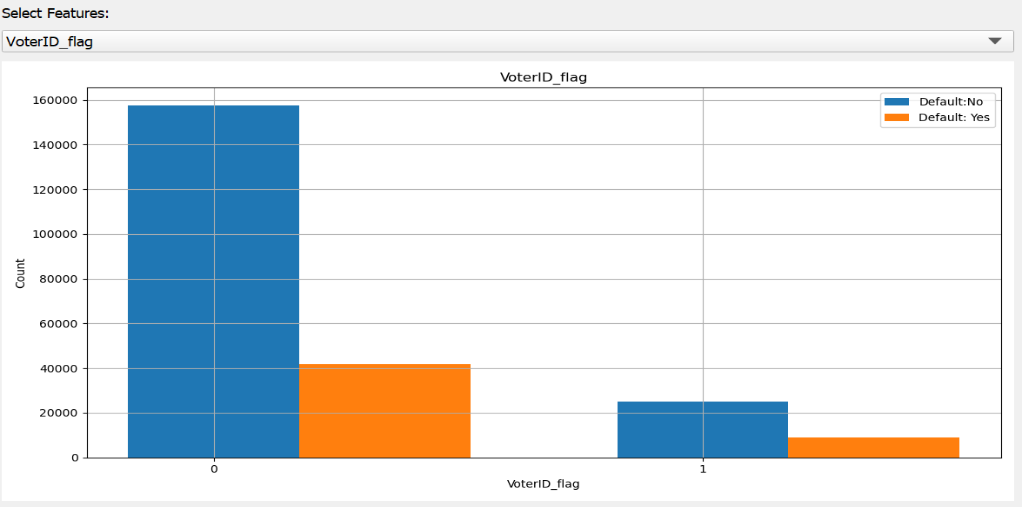


With default

Without default

From figure number four, we can analyze the voter ID flag. This variable is essential because this makes differences between profiles with and without default. As is possible to observe, most people from the database do not have a Voter ID, and this is the group with the biggest default. Therefore, it is possible that people who cannot have the biggest default could be an indicator or informality.

**Figure 4 – Voter ID**



=No Voter ID

=with VoterID

From this plot, we can observe a positive and strong correlation. Therefore, we choose to select those points with a correlation more significant than 70%. The variables that fulfill this requirement are asset cost, disbursed amount, sanctioned amount, credit history length, and average account age.

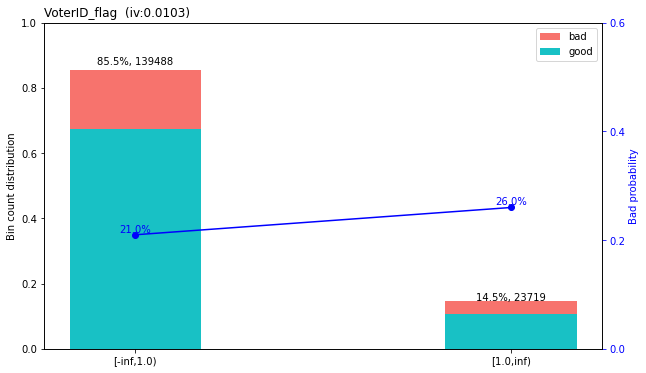
**Figure 5 – Correlation Plot**

Chart, scatter chart

Description automatically generated

The weight of the evidence plot is well known among the risk areas from financial institutions. Therefore, we can interpret this plot as 21 over 100 will default from those with a Voter ID. On the other hand, from those who do not have a Voter ID, 26 over 100 will have a default. This is a crucial plot given that confirms the importance of the variable as a predictor because those who contribute more to the model are the ones that can differentiate the profiles better.

**Figure 6 – Weight of evidence from Voter ID**



=With VoterID

=No VoterID

DATA PREPROCESSING

Data preprocessing is a technique that works for the dirty data that cannot mine directly and transform into the one that is efficiently used. Data preprocessing has several methods: data cleansing, data integration, data transformation, etc. These data processing techniques are used before data mining, significantly improving the quality of the data mining model and reducing the time required to excavate.

The following are the various preprocessing techniques that we applied to our dataset.

**Moving null values:**

There are over 5000 null values in our dataset. To prevent them from affecting the accuracy of the prediction, we remove them from the dataset.

**Format converting:**

Variables' CREDIT.HISTORY. LENGTH' and 'AVERAGE.ACCT.AGE' (credit history length and average account age) have informed date format (such as '2yrs 4mon'), we formatted and calculated them into a month.

**Date of Birth to Age:**

Converted date of birth to age.

**Categorized ID data:**

Made buckets for supplier id, branch id, manufacturer id, and State ID to reduce the levels of these features.

**One hot encoding:**

One hot encoded categorical feature such as employment type and the categorical features we generated earlier.

**Scaling:**

Scaled the features when required. We applied Standard Scaler for the dataset when we used specific algorithms related to the distance calculation, including logistic Regression.

**Balanced Data:**

We found that this data was imbalanced since the target feature-loan default was imbalanced. There were fewer defaulters than non-defaulters. To overcome this issue and improve the precision, we applied Synesthetic Minority Oversampling Technique (SMOTE) method to fix the imbalanced data. SMOTE is implemented by finding the k-nearest neighbors for minority class observations and randomly choosing one of the k-nearest neighbors. The new observations it created will be added to the group defaulter.

**Feature Selection**

Since we have over 40 features, we used Random Forest Classifier to find the essential features with a higher step score on the selection iteration. Finally, we have 25 features left in our dataset, which will be applied to the following model generation.

DATA MODELING

Classification is the process of assigning data points to predefined classes or categories. In this project, we have implemented four classification Algorithms.

**Decision tree:**

Decision Tree is a tree flowchart-like structure that divides the data into different subgroups based on conditions to classify the data. A condition is selected such that the classification is as pure as possible. At each node of the tree, a decision is made about splitting the data and getting the purest nodes. We can use different measures like Gini, entropy or misclassification error to calculate what attribute to split on. When you travel down the tree, finally, at leaf nodes, we find the labels of the data of a particular sample.

Chart

Description automatically generated

Graphical user interface, application

Description automatically generated

**Random Forest:**

For random forest, we select random features to check for the best split attribute. And we use the max voting classifier to classify the data.

Graphical user interface, chart

Description automatically generated

Graphical user interface, application

Description automatically generated

**Logistic Regression:**

Logistic Regression is a Statistical Learning technique. It is one of the Supervised Machine Learning methods used in Classification tasks. We used K-fold Cross-validation as one metric for this classification.

Graphical user interface, chart

Description automatically generated

Graphical user interface, application

Description automatically generated

**Gradient Boosting:**

Gradient boosting is a machine learning technique used in classification and Regression. It relies on the intuition that the best possible next model minimizes the overall prediction error when compared with previous models. This extraordinary ensemble learning technique combines several weak learners into strong learners. This works by each model paying attention to its predecessor’s mistakes.

The following shows the results of all the Algorithms run so far. We can see that Gradient boosting gives the best prediction with a high precision of 83.4% and an accuracy of 76.8%.

Graphical user interface, chart

Description automatically generated

Graphical user interface, application

Description automatically generated

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| METRICS | RANDOM FOREST | LOGISTIC REGRESSION | GRADIENT BOOSTING | DECISION TREE |
| Accuracy | 71.6% | 66.6% | 76.8% | 68.6% |
| F1\_score | 69.8% | 65.2% | 74.3% | 66.0% |
| Precision score | 74.5% | 68.2% | 83.4% | 71.9% |
| Recall | 65.6% | 62.4% | 67.0% | 61.0% |

CONCLUSION

We have significantly improved the accuracy and precision of predicting a loan defaulter. We also found that Gradient Boosting gives the best prediction with high accuracy of 78.43. Further, we can improve the accuracy of this data by applying PCA or other feature selection techniques. We can also use other ensemble methods to get a better result.

REFERENCES

[1] 2019. LT Vehicle Loan Default Prediction. (2019). <https://www.kaggle.com/mamtadhaker/lt-vehicle-loan-defaultpredictiondata_dictionary.csv>

[2] T. Cover &P. Hart Mickey Haggblade.Nearest neighbor pattern classification, IEEE Transactions on Information Theory 2013.

[3] C.J.C. Burges, “A Tutorial on Support Vector Machines for Pattern Recognition,” submitted to Data Mining and Knowledge Discovery, 1998.

[4] Lloyd, Stuart P. "Least squares quantization in PCM." Information Theory, IEEE Transactions on 28.2 (1982): 129-137.

[5] Nitesh V. Chawla, Kevin W. Bowyer. SMOTE: synthetic minority oversampling technique, Journal of Artificial Intelligence Research Archive Volume 16 Issue 1, January 2002

[6] Herve Abdi, Lynne J. Williams. Principal component analysis, WIREs Computational Statistics, 2010

[7] Aadhar, <https://uidai.gov.in/>

**APPENDIX**

1. Data Download

|  |
| --- |
|  |
|  | ''' |
|  | Please install opendatasets package first |
|  | Please |
|  | ''' |
|  |  |
|  | import opendatasets as od |
|  | od.download(r'https://www.kaggle.com/mamtadhaker/lt-vehicle-loan-default-prediction') |

1. PREPROCESSING

|  |
| --- |
| import pandas as pd |
|  | import numpy as np |
|  | import seaborn as sns |
|  | from scipy import stats |
|  | import matplotlib.pyplot as plt |
|  | import os |
|  | import re |
|  | from sklearn.model\_selection import train\_test\_split |
|  | import random |
|  |  |
|  | import scorecardpy as sc |
|  |  |
|  | # split train into train data and test data |
|  | # os.chdir(r'D:\GWU\Aihan\DATS 6103 Data Mining\Final Project\Code') |
|  |  |
|  |  |
|  | def split\_data(inpath, target\_name, test\_size): |
|  | df = pd.read\_csv(inpath) |
|  | y = df[target\_name] |
|  | #x = df1.loc[:,df1.columns!='loan\_default'] |
|  | x=df.drop(target\_name,axis=1) |
|  | # set a random seed for the data, so that we could get the same train and test set |
|  | random.seed(12345) |
|  | X\_train, X\_test, y\_train, y\_test = train\_test\_split(x, y, test\_size=test\_size, random\_state=1, stratify=y) |
|  |  |
|  | training = pd.concat([X\_train, y\_train], axis=1) |
|  | testing = pd.concat([X\_test, y\_test], axis=1) |
|  | return training, testing |
|  |  |
|  |  |
|  | class PreProcessing(): |
|  | def \_\_init\_\_(self, df): |
|  | self.Title = "Preprocessing Start" |
|  | self.df = df |
|  | # checking the null value and drop the null value |
|  | def Null\_value(self): |
|  | self.df.isnull().sum() |
|  | self.df\_new = self.df.dropna() |
|  | return self.df\_new |
|  |  |
|  | # convert the format of 'AVERAGE.ACCT.AGE' and 'CREDIT.HISTORY.LENGTH' from 'xyrs xmon' to numbers that represent month. |
|  | def find\_number(self, text): |
|  | num = re.findall(r'[0-9]+',text) |
|  | return int(num[0])\*12 + int(num[1]) |
|  |  |
|  | def comvert\_format(self, colname): |
|  | colname\_new = self.df[colname].apply(lambda x: self.find\_number(x)) |
|  | self.df[colname] = colname\_new |
|  |  |
|  |  |
|  | # convert categorical string to numbers |
|  | def convert\_cate\_to\_num(self, colname\_list): |
|  | for colname in colname\_list: |
|  | self.df[colname] = self.df[colname].astype('category') |
|  | cat\_columns = self.df.select\_dtypes(['category']).columns |
|  | self.df[cat\_columns] = self.df[cat\_columns].apply(lambda x: x.cat.codes) |
|  |  |
|  | def format\_date(self, colname\_list): |
|  | for colname in colname\_list: |
|  | self.df[colname] = pd.to\_datetime(self.df[colname], format = "%d-%m-%y",infer\_datetime\_format=True) |
|  |  |
|  | def format\_age\_disbursal(self): |
|  | self.df['Date.of.Birth'] = self.df['Date.of.Birth'].where(self.df['Date.of.Birth'] < pd.Timestamp('now'), |
|  | self.df['Date.of.Birth'] - np.timedelta64(100, 'Y')) |
|  | self.df['Age'] = (pd.Timestamp('now') - self.df['Date.of.Birth']).astype('<m8[Y]').astype(int) |
|  | self.df['Disbursal\_months'] = ((pd.Timestamp('now') - self.df['DisbursalDate']) / np.timedelta64(1, 'M')).astype(int) |
|  |  |
|  |  |
|  | def bin\_cutpoint(self, target\_name, colname\_list): |
|  | for colname in colname\_list: |
|  | bins\_disbursed\_amount = sc.woebin(self.df, y=target\_name, x=[colname]) |
|  | sc.woebin\_plot(bins\_disbursed\_amount) |
|  |  |
|  | pd.concat(bins\_disbursed\_amount) |
|  | list\_break = pd.concat(bins\_disbursed\_amount).breaks.astype('float').to\_list() |
|  | list\_break.insert(0, float('-inf')) |
|  | # list\_break |
|  |  |
|  | self.df[colname] = pd.cut(self.df[colname], list\_break) |
|  |  |
|  | def delet\_columns(self, delete\_list): |
|  | df\_new = self.df.drop(delete\_list, axis=1) |
|  | return df\_new |
|  |  |
|  | def save\_csv(self, outpath): |
|  | self.df.to\_csv(outpath) |
|  |  |
|  |  |
|  |  |
|  |  |
|  | ''' |
|  | # format the date variable |
|  | training['Date.of.Birth'] = pd.to\_datetime(training['Date.of.Birth']).dt.strftime('%d/%m/%Y') |
|  | training['DisbursalDate'] = pd.to\_datetime(training['DisbursalDate'], format = "%d-%m-%y",infer\_datetime\_format=True) |
|  | # covert Date of birth to age |
|  |  |
|  |  |
|  | def age(born): |
|  | born\_date = datetime.strptime(born, "%d/%m/%Y").date() |
|  | today = datetime.now() |
|  | return relativedelta(today, born\_date).years |
|  |  |
|  | training['Age'] = training['Date.of.Birth'].apply(age) |
|  | training['Disbursal\_months'] = ((pd.Timestamp('now') - training['DisbursalDate'])/np.timedelta64(1,'M')).astype(int) |
|  |  |
|  | ''' |
|  |  |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  | inpath = r'lt-vehicle-loan-default-prediction/train.csv' |
|  | target\_name = 'loan\_default' |
|  | outpath\_train = r'lt-vehicle-loan-default-prediction/final\_train.csv' |
|  | outpath\_test = r'lt-vehicle-loan-default-prediction/final\_test.csv' |
|  | training, testing = split\_data(inpath, target\_name, test\_size=0.3) |
|  | # checking the format of each variable |
|  | print(training.dtypes) |
|  |  |
|  | print(PreProcessing(training).Title) |
|  | df\_new = PreProcessing(training).Null\_value() |
|  |  |
|  | # There are 5375 missing value |
|  |  |
|  | PreProcessing(df\_new).comvert\_format('AVERAGE.ACCT.AGE') |
|  | PreProcessing(df\_new).comvert\_format('CREDIT.HISTORY.LENGTH') |
|  | # comvert\_format(training, 'AVERAGE.ACCT.AGE') |
|  | # comvert\_format(training, 'CREDIT.HISTORY.LENGTH') |
|  |  |
|  | PreProcessing(df\_new).convert\_cate\_to\_num(['Employment.Type', 'PERFORM\_CNS.SCORE.DESCRIPTION']) |
|  |  |
|  | # Create Age and Disbursal\_months |
|  | PreProcessing(df\_new).format\_date(['Date.of.Birth', 'DisbursalDate']) |
|  | PreProcessing(df\_new).format\_age\_disbursal() |
|  | df\_all = PreProcessing(df\_new).delet\_columns(['UniqueID', 'Date.of.Birth', 'DisbursalDate', 'PERFORM\_CNS.SCORE.DESCRIPTION', 'Employee\_code\_ID', 'Current\_pincode\_ID']) |
|  |  |
|  |  |
|  | PreProcessing(df\_all).save\_csv(outpath\_train) |
|  |  |
|  |  |
|  |  |
|  | ''' |
|  | # FINISH FOR NOW |
|  | ''' |

1. MODELLING

import pandas as pd

import xlwt

from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler

from sklearn.pipeline import make\_pipeline

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.tree import DecisionTreeClassifier, export\_graphviz

from sklearn.compose import make\_column\_transformer

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from imblearn.over\_sampling import SMOTE

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

from sklearn.ensemble import RandomForestClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.metrics import roc\_auc\_score

from sklearn.metrics import roc\_curve, auc, log\_loss, brier\_score\_loss

from sklearn.calibration import calibration\_curve

from sklearn.linear\_model import LogisticRegression

from sklearn import feature\_selection

from sklearn import metrics

from sklearn.preprocessing import label\_binarize

from sklearn.model\_selection import cross\_val\_predict

import matplotlib.pyplot as plt

import numpy as np

from sklearn.decomposition import PCA

import random

from sklearn.model\_selection import train\_test\_split

'''

############################# Final Result ##############################

################################ loading data ################################

'''

import os

# os.chdir(r'D:\GWU\Aihan\DATS 6103 Data Mining\Final Project\Code\lt-vehicle-loan-default-prediction')

# read csv

df\_original = pd.read\_csv(r"lt-vehicle-loan-default-prediction\final\_train.csv")

df\_original.shape

df\_original.info()

'''

################################ data cleaning ################################

'''

# # ## null value check

# df\_original.isnull().sum()

# ds = df\_original.dropna()

# # print("The total number of data-points after removing the rows with missing values are:", len(df))

# #

# # ## Checking for the duplicates

# ds.duplicated().sum()

df = df\_original.drop(['loan\_default'], axis=1)

y = df\_original['loan\_default']

sm = SMOTE(random\_state=0)

df, y = sm.fit\_resample(df, y)

'''

################################ Classification ################################

'''

F1 = []

model\_names =[]

scalar = StandardScaler()

X\_train\_std = scalar.fit\_transform(df) # normalizing the features

df\_temp = pd.DataFrame(X\_train\_std)

df\_temp.columns = df.columns

y = pd.DataFrame({'loan\_default': y})

X\_train, X\_test, y\_train, y\_test = train\_test\_split(df\_temp, y, test\_size=0.3, random\_state=1)

testing = pd.concat([X\_test, y\_test], axis=1)

testing.to\_csv(r"final\_test2.csv", index=False)

'''

################## Modelling, please comment this part if necessary ######################

'''

'''

######## Logistic - scale#############

'''

lr = LogisticRegression(solver='liblinear')

lr.fit(X\_train, y\_train)

y\_pred\_lr = lr.predict(X\_test)

accuracy\_testing=accuracy\_score(y\_test,y\_pred\_lr)

f1\_score\_testing = f1\_score(y\_test,y\_pred\_lr)

precision\_score\_testing = precision\_score(y\_test,y\_pred\_lr)

recall\_score\_testing = recall\_score(y\_test,y\_pred\_lr)

print('###############Logistic Regression')

print("Accuracy")

print("Testing")

print(accuracy\_testing)

print("F1\_score")

print("Testing")

print(f1\_score\_testing)

print("Precision\_score")

print("Testing")

print(precision\_score\_testing)

print("Recall")

print("Testing")

print(recall\_score\_testing)

import pickle

filename = 'lr\_finalized\_model2.sav'

pickle.dump(lr, open(filename, 'wb'))

# filename2 = 'lr\_finalized\_model2.sav'

# clf\_entropy = pickle.load(open(filename2, 'rb'))

# y\_pred\_entropy = clf\_entropy.predict(X\_test)

# accuracy\_score = accuracy\_score(y\_test, y\_pred\_entropy)

# f1\_score= f1\_score(y\_test,y\_pred\_entropy)

# precision = precision\_score(y\_test,y\_pred\_entropy)

# recall = recall\_score(y\_test,y\_pred\_entropy)

#

# print("Accuracy")

# print(accuracy\_score)

# print("F1\_score")

# print(f1\_score)

# print("Precision\_score")

# print(precision)

# print("Recall")

# print(recall)

dt = DecisionTreeClassifier(max\_depth=5,min\_samples\_leaf=0.01,criterion='gini',class\_weight='balanced',random\_state=123)

dt.fit(X\_train, y\_train)

y\_pred\_lr = dt.predict(X\_test)

accuracy\_testing=accuracy\_score(y\_test,y\_pred\_lr)

f1\_score\_testing = f1\_score(y\_test,y\_pred\_lr)

precision\_score\_testing = precision\_score(y\_test,y\_pred\_lr)

recall\_score\_testing = recall\_score(y\_test,y\_pred\_lr)

print('#################Decision Tree')

print("Accuracy")

print(accuracy\_testing)

print("F1\_score")

print(f1\_score\_testing)

print("Precision\_score")

print(precision\_score\_testing)

print("Recall")

print(recall\_score\_testing)

import pickle

# filename = 'dt\_finalized\_model2.sav'

# pickle.dump(dt, open(filename, 'wb'))

# to graph the tree

from pydotplus import graph\_from\_dot\_data

from sklearn.tree import export\_graphviz

dot\_data = export\_graphviz(dt, filled=True, rounded=True, class\_names=["No","Yes"], feature\_names=X\_test.columns, out\_file=None)

graph = graph\_from\_dot\_data(dot\_data)

graph.write\_pdf("decision\_tree\_entropy.pdf")

1. GUI

import sys,os

#os.chdir("/Users/utkarshvirendranigam/Desktop/Homework/Project")

# required\_packages=["PyQt5","re", "scipy","itertools","random","matplotlib","pandas","numpy","sklearn","pydotplus","collections","warnings","seaborn"]

#print(os.getcwd())

# for my\_package in required\_packages:

# try:

# command\_string="conda install "+ my\_package+ " --yes"

# os.system(command\_string)

# except:

# count=1

from PyQt5.QtWidgets import (QMainWindow, QApplication, QWidget, QPushButton, QAction, QComboBox, QLabel,

QGridLayout, QCheckBox, QGroupBox, QVBoxLayout, QHBoxLayout, QLineEdit, QPlainTextEdit)

from PyQt5.QtGui import QIcon

from PyQt5.QtCore import pyqtSlot, QRect

from PyQt5.QtCore import pyqtSignal

from PyQt5.QtCore import Qt

# from scipy import interp

from itertools import cycle, combinations

import random

from PyQt5.QtWidgets import QDialog, QVBoxLayout, QSizePolicy, QFormLayout, QRadioButton, QScrollArea, QMessageBox

from PyQt5.QtGui import QPixmap

from matplotlib.backends.backend\_qt5agg import FigureCanvasQTAgg as FigureCanvas

from matplotlib.backends.backend\_qt5agg import NavigationToolbar2QT as NavigationToolbar

from matplotlib.figure import Figure

import pandas as pd

import numpy as np

import pickle

from numpy.polynomial.polynomial import polyfit

from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler

from sklearn.pipeline import make\_pipeline

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.tree import DecisionTreeClassifier, export\_graphviz

from sklearn.compose import make\_column\_transformer

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

from sklearn.ensemble import RandomForestClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn.metrics import roc\_auc\_score

from sklearn.metrics import roc\_curve, auc, log\_loss, brier\_score\_loss

from sklearn.calibration import calibration\_curve

from sklearn.linear\_model import LogisticRegression

from sklearn.ensemble import GradientBoostingClassifier

from sklearn import feature\_selection

from sklearn import metrics

from sklearn.preprocessing import label\_binarize

from sklearn.model\_selection import cross\_val\_predict

# Libraries to display decision tree

from pydotplus import graph\_from\_dot\_data

import collections

from sklearn.tree import export\_graphviz

import webbrowser

import warnings

warnings.filterwarnings("ignore")

import matplotlib.pyplot as plt

from Preprocessing import PreProcessing

import random

import seaborn as sns

#%%-----------------------------------------------------------------------

import os

os.environ["PATH"] += os.pathsep + 'C:\\Program Files (x86)\\graphviz-2.38\\release\\bin'

#%%-----------------------------------------------------------------------

#::--------------------------------

# Deafault font size for all the windows

#::--------------------------------

font\_size\_window = 'font-size:18px'

class DecisionTree(QMainWindow):

#::--------------------------------------------------------------------------------

# Implementation of Random Forest Classifier using the happiness dataset

# the methods in this class are

# \_init\_ : initialize the class

# initUi : creates the canvas and all the elements in the canvas

# update : populates the elements of the canvas base on the parametes

# chosen by the user

#::---------------------------------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

super(DecisionTree, self).\_\_init\_\_()

self.Title = "Decision Tree Classifier"

self.initUi()

def initUi(self):

#::-----------------------------------------------------------------

# Create the canvas and all the element to create a dashboard with

# all the necessary elements to present the results from the algorithm

# The canvas is divided using a grid loyout to facilitate the drawing

# of the elements

#::-----------------------------------------------------------------

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.main\_widget = QWidget(self)

self.layout = QGridLayout(self.main\_widget)

self.groupBox1 = QGroupBox('Decision Tree Features')

self.groupBox1Layout= QGridLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.feature0 = QCheckBox(features\_list[0],self)

self.feature1 = QCheckBox(features\_list[1],self)

self.feature2 = QCheckBox(features\_list[2], self)

self.feature3 = QCheckBox(features\_list[3], self)

self.feature4 = QCheckBox(features\_list[4],self)

self.feature5 = QCheckBox(features\_list[5],self)

self.feature6 = QCheckBox(features\_list[6], self)

self.feature7 = QCheckBox(features\_list[7], self)

self.feature8 = QCheckBox(features\_list[8], self)

self.feature9 = QCheckBox(features\_list[9], self)

self.feature10 = QCheckBox(features\_list[10], self)

self.feature11 = QCheckBox(features\_list[11], self)

self.feature12 = QCheckBox(features\_list[12], self)

self.feature13 = QCheckBox(features\_list[13], self)

self.feature14 = QCheckBox(features\_list[14], self)

self.feature15 = QCheckBox(features\_list[15], self)

self.feature16 = QCheckBox(features\_list[16], self)

self.feature17 = QCheckBox(features\_list[17], self)

self.feature18 = QCheckBox(features\_list[18], self)

self.feature19 = QCheckBox(features\_list[19], self)

self.feature20 = QCheckBox(features\_list[20], self)

self.feature21 = QCheckBox(features\_list[21], self)

self.feature22 = QCheckBox(features\_list[22], self)

self.feature23 = QCheckBox(features\_list[23], self)

self.feature24 = QCheckBox(features\_list[24], self)

self.feature25 = QCheckBox(features\_list[25], self)

self.feature26 = QCheckBox(features\_list[26], self)

self.feature27 = QCheckBox(features\_list[27], self)

self.feature28 = QCheckBox(features\_list[28], self)

self.feature29 = QCheckBox(features\_list[29], self)

self.feature30 = QCheckBox(features\_list[30], self)

self.feature31 = QCheckBox(features\_list[31], self)

self.feature32 = QCheckBox(features\_list[32], self)

self.feature33 = QCheckBox(features\_list[33], self)

self.feature34 = QCheckBox(features\_list[34], self)

self.feature35 = QCheckBox(features\_list[35], self)

self.feature0.setChecked(True)

self.feature1.setChecked(True)

self.feature2.setChecked(True)

self.feature3.setChecked(True)

self.feature4.setChecked(True)

self.feature5.setChecked(True)

self.feature6.setChecked(True)

self.feature7.setChecked(True)

self.feature8.setChecked(True)

self.feature9.setChecked(True)

self.feature10.setChecked(True)

self.feature11.setChecked(True)

self.feature12.setChecked(True)

self.feature13.setChecked(True)

self.feature14.setChecked(True)

self.feature15.setChecked(True)

self.feature16.setChecked(True)

self.feature17.setChecked(True)

self.feature18.setChecked(True)

self.feature19.setChecked(True)

self.feature20.setChecked(True)

self.feature21.setChecked(True)

self.feature22.setChecked(True)

self.feature23.setChecked(True)

self.feature24.setChecked(True)

self.feature25.setChecked(True)

self.feature26.setChecked(True)

self.feature27.setChecked(True)

self.feature28.setChecked(True)

self.feature29.setChecked(True)

self.feature30.setChecked(True)

self.feature31.setChecked(True)

self.feature32.setChecked(True)

self.feature33.setChecked(True)

self.feature34.setChecked(True)

self.feature35.setChecked(True)

self.lblPercentTest = QLabel('Percentage for Test :')

self.lblPercentTest.adjustSize()

self.txtPercentTest = QLineEdit(self)

self.txtPercentTest.setText("30")

self.lblMaxDepth = QLabel('Maximun Depth :')

self.txtMaxDepth = QLineEdit(self)

self.txtMaxDepth.setText("3")

self.btnExecute = QPushButton("Run Model")

self.btnExecute.setGeometry(QRect(60, 500, 75, 23))

self.btnExecute.clicked.connect(self.update)

self.btnDTFigure = QPushButton("View Tree")

self.btnDTFigure.setGeometry(QRect(60, 500, 75, 23))

self.btnDTFigure.clicked.connect(self.view\_tree)

# We create a checkbox for each feature

self.groupBox1Layout.addWidget(self.feature0, 0, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature1, 0, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature2, 1, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature3, 1, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature4, 2, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature5, 2, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature6, 3, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature7, 3, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature8, 4, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature9, 4, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature10, 5, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature11, 5, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature12, 6, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature13, 6, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature14, 7, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature15, 7, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature16, 8, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature17, 8, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature18, 9, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature19, 9, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature20, 10, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature21, 10, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature22, 11, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature23, 11, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature24, 12, 0, 1, 1)

self.groupBox1Layout.addWidget(self.lblPercentTest, 19, 0, 1, 1)

self.groupBox1Layout.addWidget(self.txtPercentTest, 19, 1, 1, 1)

self.groupBox1Layout.addWidget(self.lblMaxDepth, 20, 0, 1, 1)

self.groupBox1Layout.addWidget(self.txtMaxDepth, 20, 1, 1, 1)

self.groupBox1Layout.addWidget(self.btnExecute, 21, 0, 1, 1)

self.groupBox1Layout.addWidget(self.btnDTFigure, 21, 1, 1, 1)

self.groupBox2 = QGroupBox('Measurements:')

self.groupBox2Layout = QVBoxLayout()

self.groupBox2.setLayout(self.groupBox2Layout)

# self.groupBox2.setMinimumSize(400, 100)

self.current\_model\_summary = QWidget(self)

self.current\_model\_summary.layout = QFormLayout(self.current\_model\_summary)

self.txtCurrentAccuracy = QLineEdit()

self.txtCurrentPrecision = QLineEdit()

self.txtCurrentRecall = QLineEdit()

self.txtCurrentF1score = QLineEdit()

self.current\_model\_summary.layout.addRow('Accuracy:', self.txtCurrentAccuracy)

self.current\_model\_summary.layout.addRow('Precision:', self.txtCurrentPrecision)

self.current\_model\_summary.layout.addRow('Recall:', self.txtCurrentRecall)

self.current\_model\_summary.layout.addRow('F1 Score:', self.txtCurrentF1score)

self.groupBox2Layout.addWidget(self.current\_model\_summary)

self.groupBox3 = QGroupBox('Other Models Accuracy:')

self.groupBox3Layout = QVBoxLayout()

self.groupBox3.setLayout(self.groupBox3Layout)

self.other\_models = QWidget(self)

self.other\_models.layout = QFormLayout(self.other\_models)

self.txtAccuracy\_lr = QLineEdit()

self.txtAccuracy\_gb = QLineEdit()

self.txtAccuracy\_rf = QLineEdit()

self.other\_models.layout.addRow('Logistic:', self.txtAccuracy\_lr)

self.other\_models.layout.addRow('Random Forest:', self.txtAccuracy\_rf)

self.other\_models.layout.addRow('Gradient Boosting:', self.txtAccuracy\_gb)

self.groupBox3Layout.addWidget(self.other\_models)

#::-------------------------------------

# Graphic 1 : Confusion Matrix

#::-------------------------------------

self.fig = Figure()

self.ax1 = self.fig.add\_subplot(111)

self.axes=[self.ax1]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.groupBoxG1 = QGroupBox('Confusion Matrix')

self.groupBoxG1Layout= QVBoxLayout()

self.groupBoxG1.setLayout(self.groupBoxG1Layout)

self.groupBoxG1Layout.addWidget(self.canvas)

#::---------------------------------------------

# Graphic 2 : ROC Curve

#::---------------------------------------------

self.fig2 = Figure()

self.ax2 = self.fig2.add\_subplot(111)

self.axes2 = [self.ax2]

self.canvas2 = FigureCanvas(self.fig2)

self.canvas2.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas2.updateGeometry()

self.groupBoxG2 = QGroupBox('ROC Curve')

self.groupBoxG2Layout = QVBoxLayout()

self.groupBoxG2.setLayout(self.groupBoxG2Layout)

self.groupBoxG2Layout.addWidget(self.canvas2)

#::-------------------------------------------

# Graphic 3 : Importance of Features

#::-------------------------------------------

self.fig3 = Figure()

self.ax3 = self.fig3.add\_subplot(111)

self.axes3 = [self.ax3]

self.canvas3 = FigureCanvas(self.fig3)

self.canvas3.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas3.updateGeometry()

self.groupBoxG3 = QGroupBox('Importance of Features')

self.groupBoxG3Layout = QVBoxLayout()

self.groupBoxG3.setLayout(self.groupBoxG3Layout)

self.groupBoxG3Layout.addWidget(self.canvas3)

#::--------------------------------------------

# Graphic 4 : ROC Curve by class

#::--------------------------------------------

self.fig4 = Figure()

self.ax4 = self.fig4.add\_subplot(111)

self.axes4 = [self.ax4]

self.canvas4 = FigureCanvas(self.fig4)

self.canvas4.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas4.updateGeometry()

self.groupBoxG4 = QGroupBox('ROC Curve by Class')

self.groupBoxG4Layout = QVBoxLayout()

self.groupBoxG4.setLayout(self.groupBoxG4Layout)

self.groupBoxG4Layout.addWidget(self.canvas4)

#::-------------------------------------------------

# End of graphs

#::-------------------------------------------------

self.layout.addWidget(self.groupBox1, 0, 0, 3, 2)

self.layout.addWidget(self.groupBoxG1, 0, 2, 1, 1)

self.layout.addWidget(self.groupBoxG3, 0, 3, 1, 1)

self.layout.addWidget(self.groupBoxG2, 1, 2, 1, 1)

self.layout.addWidget(self.groupBoxG4, 1, 3, 1, 1)

self.layout.addWidget(self.groupBox2, 2, 2, 1, 1)

self.layout.addWidget(self.groupBox3, 2, 3, 1, 1)

self.setCentralWidget(self.main\_widget)

self.resize(1800, 1200)

self.show()

def update(self):

'''

Random Forest Classifier

We pupulate the dashboard using the parametres chosen by the user

The parameters are processed to execute in the skit-learn Random Forest algorithm

then the results are presented in graphics and reports in the canvas

:return:None

'''

# processing the parameters

self.list\_corr\_features = pd.DataFrame([])

if self.feature0.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[0]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[0]]],axis=1)

if self.feature1.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[1]]],axis=1)

if self.feature2.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[2]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[2]]],axis=1)

if self.feature3.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[3]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[3]]],axis=1)

if self.feature4.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[4]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[4]]],axis=1)

if self.feature5.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[5]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[5]]],axis=1)

if self.feature6.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[6]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[6]]],axis=1)

if self.feature7.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[7]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[7]]],axis=1)

if self.feature8.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[8]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[8]]],axis=1)

if self.feature9.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[9]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[9]]],axis=1)

if self.feature10.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[10]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[10]]], axis=1)

if self.feature11.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[11]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[11]]], axis=1)

if self.feature12.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[12]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[12]]], axis=1)

if self.feature13.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[13]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[13]]], axis=1)

if self.feature14.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[14]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[14]]], axis=1)

if self.feature15.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[15]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[15]]], axis=1)

if self.feature16.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[16]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[16]]], axis=1)

if self.feature17.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[17]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[17]]], axis=1)

if self.feature18.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[18]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[18]]], axis=1)

if self.feature19.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[19]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[19]]], axis=1)

if self.feature20.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[20]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[20]]],axis=1)

if self.feature21.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[21]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[21]]],axis=1)

if self.feature22.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[22]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[22]]],axis=1)

if self.feature23.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[23]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[23]]],axis=1)

if self.feature24.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[24]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[24]]],axis=1)

if self.feature25.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[25]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[25]]],axis=1)

if self.feature26.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[26]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[26]]],axis=1)

if self.feature27.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[27]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[27]]],axis=1)

if self.feature28.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[28]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[28]]],axis=1)

if self.feature29.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[29]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[29]]],axis=1)

if self.feature30.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[30]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[30]]], axis=1)

if self.feature31.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[31]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[31]]], axis=1)

if self.feature32.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[32]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[32]]], axis=1)

if self.feature33.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[33]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[33]]], axis=1)

if self.feature34.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[34]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[34]]], axis=1)

if self.feature35.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[35]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[35]]], axis=1)

vtest\_per = float(self.txtPercentTest.text())

vmax\_depth = float(self.txtMaxDepth.text())

# Clear the graphs to populate them with the new information

self.ax1.clear()

self.ax2.clear()

self.ax3.clear()

self.ax4.clear()

# self.txtResults.clear()

# self.txtResults.setUndoRedoEnabled(False)

vtest\_per = vtest\_per / 100

# -----------------------------------------------------------------------

filename = 'dt\_finalized\_model2.sav'

self.clf\_entropy = pickle.load(open(filename, 'rb'))

y\_test = y

X\_test= X

#

# scalar = StandardScaler()

# X\_test = scalar.fit\_transform(X)

# predicton on test using entropy

y\_pred\_entropy = self.clf\_entropy.predict(X\_test)

# confusion matrix for RandomForest

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_entropy)

# accuracy score

self.ff\_accuracy\_score = accuracy\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentAccuracy.setText(str(self.ff\_accuracy\_score))

# precision score

self.ff\_precision\_score = precision\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentPrecision.setText(str(self.ff\_precision\_score))

# recall score

self.ff\_recall\_score = recall\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentRecall.setText(str(self.ff\_recall\_score))

# f1\_score

self.ff\_f1\_score = f1\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentF1score.setText(str(self.ff\_f1\_score))

#::------------------------------------

## Ghaph1 :

## Confusion Matrix

#::------------------------------------

class\_names1 = ['', 'No', 'Yes']

self.ax1.matshow(conf\_matrix, cmap=plt.cm.get\_cmap('Blues', 14))

self.ax1.set\_yticklabels(class\_names1)

self.ax1.set\_xticklabels(class\_names1, rotation=90)

self.ax1.set\_xlabel('Predicted label')

self.ax1.set\_ylabel('True label')

for i in range(len(class\_names)):

for j in range(len(class\_names)):

y\_pred\_score = self.clf\_entropy.predict\_proba(X\_test)

self.ax1.text(j, i, str(conf\_matrix[i][j]))

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

#::----------------------------------------

## Graph 2 - ROC Curve

#::----------------------------------------

y\_test\_bin = pd.get\_dummies(y\_test).to\_numpy()

n\_classes = y\_test\_bin.shape[1]

# From the sckict learn site

# https://scikit-learn.org/stable/auto\_examples/model\_selection/plot\_roc.html

fpr = dict()

tpr = dict()

roc\_auc = dict()

for i in range(n\_classes):

fpr[i], tpr[i], \_ = roc\_curve(y\_test\_bin[:, i], y\_pred\_score[:, i])

roc\_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area

fpr["micro"], tpr["micro"], \_ = roc\_curve(y\_test\_bin.ravel(), y\_pred\_score.ravel())

roc\_auc["micro"] = auc(fpr["micro"], tpr["micro"])

lw = 2

self.ax2.plot(fpr[1], tpr[1], color='darkorange',

lw=lw, label='ROC curve (area = %0.2f)' % roc\_auc[1])

self.ax2.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

self.ax2.set\_xlim([0.0, 1.0])

self.ax2.set\_ylim([0.0, 1.05])

self.ax2.set\_xlabel('False Positive Rate')

self.ax2.set\_ylabel('True Positive Rate')

self.ax2.set\_title('ROC Curve Random Forest')

self.ax2.legend(loc="lower right")

self.fig2.tight\_layout()

self.fig2.canvas.draw\_idle()

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

# Graph - 3 Feature Importances

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

# get feature importances

importances = self.clf\_entropy.feature\_importances\_

# convert the importances into one-dimensional 1darray with corresponding df column names as axis labels

f\_importances = pd.Series(importances, self.list\_corr\_features.columns)

# sort the array in descending order of the importances, only show the first 10

f\_importances.sort\_values(ascending=False, inplace=True)

f\_importances = f\_importances[0:10]

X\_Features = f\_importances.index

y\_Importance = list(f\_importances)

self.ax3.barh(X\_Features, y\_Importance)

self.ax3.set\_aspect('auto')

# show the plot

self.fig3.tight\_layout()

self.fig3.canvas.draw\_idle()

#::-----------------------------------------------------

# Graph 4 - ROC Curve by Class

#::-----------------------------------------------------

str\_classes = ['No','Yes']

colors = cycle(['magenta', 'darkorange'])

for i, color in zip(range(n\_classes), colors):

self.ax4.plot(fpr[i], tpr[i], color=color, lw=lw,

label='{0} (area = {1:0.2f})'

''.format(str\_classes[i], roc\_auc[i]))

self.ax4.plot([0, 1], [0, 1], 'k--', lw=lw)

self.ax4.set\_xlim([0.0, 1.0])

self.ax4.set\_ylim([0.0, 1.05])

self.ax4.set\_xlabel('False Positive Rate')

self.ax4.set\_ylabel('True Positive Rate')

self.ax4.set\_title('ROC Curve by Class')

self.ax4.legend(loc="lower right")

# show the plot

self.fig4.tight\_layout()

self.fig4.canvas.draw\_idle()

#::-----------------------------------------------------

# Other Models Comparison

#::-----------------------------------------------------

filename2 = 'lr\_finalized\_model2.sav'

self.other\_clf\_lr = pickle.load(open(filename2, 'rb'))

y\_pred\_lr = self.other\_clf\_lr.predict(X\_test)

self.accuracy\_lr = accuracy\_score(y\_test, y\_pred\_lr) \* 100

self.txtAccuracy\_lr.setText(str(self.accuracy\_lr))

filename3 = 'rf\_finalized\_model2.sav'

self.other\_clf\_rf = pickle.load(open(filename3, 'rb'))

y\_pred\_rf = self.other\_clf\_rf.predict(X\_test)

self.accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf) \* 100

self.txtAccuracy\_rf.setText(str(self.accuracy\_rf))

filename4 = 'gb\_finalized\_model2.sav'

self.other\_clf\_gb = pickle.load(open(filename4, 'rb'))

y\_pred\_gb = self.other\_clf\_gb.predict(X\_test)

self.accuracy\_gb = accuracy\_score(y\_test, y\_pred\_gb) \* 100

self.txtAccuracy\_gb.setText(str(self.accuracy\_gb))

def view\_tree(self):

'''

Executes the graphviz to create a tree view of the information

then it presents the graphic in a pdf formt using webbrowser

:return:None

'''

webbrowser.open\_new(r'decision\_tree\_entropy.pdf')

class RandomForest(QMainWindow):

#::--------------------------------------------------------------------------------

# Implementation of Random Forest Classifier using the happiness dataset

# the methods in this class are

# \_init\_ : initialize the class

# initUi : creates the canvas and all the elements in the canvas

# update : populates the elements of the canvas base on the parametes

# chosen by the user

#::---------------------------------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

super(RandomForest, self).\_\_init\_\_()

self.Title = "Random Forest Classifier"

self.initUi()

def initUi(self):

#::-----------------------------------------------------------------

# Create the canvas and all the element to create a dashboard with

# all the necessary elements to present the results from the algorithm

# The canvas is divided using a grid loyout to facilitate the drawing

# of the elements

#::-----------------------------------------------------------------

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.main\_widget = QWidget(self)

self.layout = QGridLayout(self.main\_widget)

self.groupBox1 = QGroupBox('Random Forest Features')

self.groupBox1Layout= QGridLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.feature0 = QCheckBox(features\_list[0], self)

self.feature1 = QCheckBox(features\_list[1], self)

self.feature2 = QCheckBox(features\_list[2], self)

self.feature3 = QCheckBox(features\_list[3], self)

self.feature4 = QCheckBox(features\_list[4], self)

self.feature5 = QCheckBox(features\_list[5], self)

self.feature6 = QCheckBox(features\_list[6], self)

self.feature7 = QCheckBox(features\_list[7], self)

self.feature8 = QCheckBox(features\_list[8], self)

self.feature9 = QCheckBox(features\_list[9], self)

self.feature10 = QCheckBox(features\_list[10], self)

self.feature11 = QCheckBox(features\_list[11], self)

self.feature12 = QCheckBox(features\_list[12], self)

self.feature13 = QCheckBox(features\_list[13], self)

self.feature14 = QCheckBox(features\_list[14], self)

self.feature15 = QCheckBox(features\_list[15], self)

self.feature16 = QCheckBox(features\_list[16], self)

self.feature17 = QCheckBox(features\_list[17], self)

self.feature18 = QCheckBox(features\_list[18], self)

self.feature19 = QCheckBox(features\_list[19], self)

self.feature20 = QCheckBox(features\_list[20], self)

self.feature21 = QCheckBox(features\_list[21], self)

self.feature22 = QCheckBox(features\_list[22], self)

self.feature23 = QCheckBox(features\_list[23], self)

self.feature24 = QCheckBox(features\_list[24], self)

self.feature25 = QCheckBox(features\_list[25], self)

self.feature26 = QCheckBox(features\_list[26], self)

self.feature27 = QCheckBox(features\_list[27], self)

self.feature28 = QCheckBox(features\_list[28], self)

self.feature29 = QCheckBox(features\_list[29], self)

self.feature30 = QCheckBox(features\_list[30], self)

self.feature31 = QCheckBox(features\_list[31], self)

self.feature32 = QCheckBox(features\_list[32], self)

self.feature33 = QCheckBox(features\_list[33], self)

self.feature34 = QCheckBox(features\_list[34], self)

self.feature35 = QCheckBox(features\_list[35], self)

self.feature0.setChecked(True)

self.feature1.setChecked(True)

self.feature2.setChecked(True)

self.feature3.setChecked(True)

self.feature4.setChecked(True)

self.feature5.setChecked(True)

self.feature6.setChecked(True)

self.feature7.setChecked(True)

self.feature8.setChecked(True)

self.feature9.setChecked(True)

self.feature10.setChecked(True)

self.feature11.setChecked(True)

self.feature12.setChecked(True)

self.feature13.setChecked(True)

self.feature14.setChecked(True)

self.feature15.setChecked(True)

self.feature16.setChecked(True)

self.feature17.setChecked(True)

self.feature18.setChecked(True)

self.feature19.setChecked(True)

self.feature20.setChecked(True)

self.feature21.setChecked(True)

self.feature22.setChecked(True)

self.feature23.setChecked(True)

self.feature24.setChecked(True)

self.feature25.setChecked(True)

self.feature26.setChecked(True)

self.feature27.setChecked(True)

self.feature28.setChecked(True)

self.feature29.setChecked(True)

self.feature30.setChecked(True)

self.feature31.setChecked(True)

self.feature32.setChecked(True)

self.feature33.setChecked(True)

self.feature34.setChecked(True)

self.feature35.setChecked(True)

self.lblPercentTest = QLabel('Percentage for Test :')

self.lblPercentTest.adjustSize()

self.txtPercentTest = QLineEdit(self)

self.txtPercentTest.setText("30")

self.lblMaxDepth = QLabel('Maximun Depth :')

self.txtMaxDepth = QLineEdit(self)

self.txtMaxDepth.setText("3")

self.btnExecute = QPushButton("Run Model")

self.btnExecute.setGeometry(QRect(60, 500, 75, 23))

self.btnExecute.clicked.connect(self.update)

# We create a checkbox for each feature

self.groupBox1Layout.addWidget(self.feature0, 0, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature1, 0, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature2, 1, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature3, 1, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature4, 2, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature5, 2, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature6, 3, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature7, 3, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature8, 4, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature9, 4, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature10, 5, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature11, 5, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature12, 6, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature13, 6, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature14, 7, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature15, 7, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature16, 8, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature17, 8, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature18, 9, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature19, 9, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature20, 10, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature21, 10, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature22, 11, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature23, 11, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature24, 12, 0, 1, 1)

self.groupBox1Layout.addWidget(self.lblPercentTest, 19, 0, 1, 1)

self.groupBox1Layout.addWidget(self.txtPercentTest, 19, 1, 1, 1)

self.groupBox1Layout.addWidget(self.lblMaxDepth, 20, 0, 1, 1)

self.groupBox1Layout.addWidget(self.txtMaxDepth, 20, 1, 1, 1)

self.groupBox1Layout.addWidget(self.btnExecute, 21, 0, 1, 1)

self.groupBox2 = QGroupBox('Measurements:')

self.groupBox2Layout = QVBoxLayout()

self.groupBox2.setLayout(self.groupBox2Layout)

# self.groupBox2.setMinimumSize(400, 100)

self.current\_model\_summary = QWidget(self)

self.current\_model\_summary.layout = QFormLayout(self.current\_model\_summary)

self.txtCurrentAccuracy = QLineEdit()

self.txtCurrentPrecision = QLineEdit()

self.txtCurrentRecall = QLineEdit()

self.txtCurrentF1score = QLineEdit()

self.current\_model\_summary.layout.addRow('Accuracy:', self.txtCurrentAccuracy)

self.current\_model\_summary.layout.addRow('Precision:', self.txtCurrentPrecision)

self.current\_model\_summary.layout.addRow('Recall:', self.txtCurrentRecall)

self.current\_model\_summary.layout.addRow('F1 Score:', self.txtCurrentF1score)

self.groupBox2Layout.addWidget(self.current\_model\_summary)

self.groupBox3 = QGroupBox('Other Models Accuracy:')

self.groupBox3Layout = QVBoxLayout()

self.groupBox3.setLayout(self.groupBox3Layout)

self.other\_models = QWidget(self)

self.other\_models.layout = QFormLayout(self.other\_models)

self.txtAccuracy\_lr = QLineEdit()

self.txtAccuracy\_gb = QLineEdit()

self.txtAccuracy\_dt = QLineEdit()

self.other\_models.layout.addRow('Logistic:', self.txtAccuracy\_lr)

self.other\_models.layout.addRow('Gradient Boosting:', self.txtAccuracy\_gb)

self.other\_models.layout.addRow('Decision tree:', self.txtAccuracy\_dt)

self.groupBox3Layout.addWidget(self.other\_models)

#::-------------------------------------

# Graphic 1 : Confusion Matrix

#::-------------------------------------

self.fig = Figure()

self.ax1 = self.fig.add\_subplot(111)

self.axes=[self.ax1]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.groupBoxG1 = QGroupBox('Confusion Matrix')

self.groupBoxG1Layout= QVBoxLayout()

self.groupBoxG1.setLayout(self.groupBoxG1Layout)

self.groupBoxG1Layout.addWidget(self.canvas)

#::---------------------------------------------

# Graphic 2 : ROC Curve

#::---------------------------------------------

self.fig2 = Figure()

self.ax2 = self.fig2.add\_subplot(111)

self.axes2 = [self.ax2]

self.canvas2 = FigureCanvas(self.fig2)

self.canvas2.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas2.updateGeometry()

self.groupBoxG2 = QGroupBox('ROC Curve')

self.groupBoxG2Layout = QVBoxLayout()

self.groupBoxG2.setLayout(self.groupBoxG2Layout)

self.groupBoxG2Layout.addWidget(self.canvas2)

#::-------------------------------------------

# Graphic 3 : Importance of Features

#::-------------------------------------------

self.fig3 = Figure()

self.ax3 = self.fig3.add\_subplot(111)

self.axes3 = [self.ax3]

self.canvas3 = FigureCanvas(self.fig3)

self.canvas3.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas3.updateGeometry()

self.groupBoxG3 = QGroupBox('Importance of Features')

self.groupBoxG3Layout = QVBoxLayout()

self.groupBoxG3.setLayout(self.groupBoxG3Layout)

self.groupBoxG3Layout.addWidget(self.canvas3)

#::--------------------------------------------

# Graphic 4 : ROC Curve by class

#::--------------------------------------------

self.fig4 = Figure()

self.ax4 = self.fig4.add\_subplot(111)

self.axes4 = [self.ax4]

self.canvas4 = FigureCanvas(self.fig4)

self.canvas4.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas4.updateGeometry()

self.groupBoxG4 = QGroupBox('ROC Curve by Class')

self.groupBoxG4Layout = QVBoxLayout()

self.groupBoxG4.setLayout(self.groupBoxG4Layout)

self.groupBoxG4Layout.addWidget(self.canvas4)

#::-------------------------------------------------

# End of graphs

#::-------------------------------------------------

self.layout.addWidget(self.groupBox1, 0, 0, 3, 2)

self.layout.addWidget(self.groupBoxG1, 0, 2, 1, 1)

self.layout.addWidget(self.groupBoxG3, 0, 3, 1, 1)

self.layout.addWidget(self.groupBoxG2, 1, 2, 1, 1)

self.layout.addWidget(self.groupBoxG4, 1, 3, 1, 1)

self.layout.addWidget(self.groupBox2, 2, 2, 1, 1)

self.layout.addWidget(self.groupBox3, 2, 3, 1, 1)

self.setCentralWidget(self.main\_widget)

self.resize(1800, 1200)

self.show()

def update(self):

'''

Random Forest Classifier

We pupulate the dashboard using the parametres chosen by the user

The parameters are processed to execute in the skit-learn Random Forest algorithm

then the results are presented in graphics and reports in the canvas

:return:None

'''

# processing the parameters

self.list\_corr\_features = pd.DataFrame([])

if self.feature0.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[0]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[0]]],axis=1)

if self.feature1.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[1]]],axis=1)

if self.feature2.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[2]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[2]]],axis=1)

if self.feature3.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[3]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[3]]],axis=1)

if self.feature4.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[4]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[4]]],axis=1)

if self.feature5.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[5]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[5]]],axis=1)

if self.feature6.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[6]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[6]]],axis=1)

if self.feature7.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[7]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[7]]],axis=1)

if self.feature8.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[8]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[8]]],axis=1)

if self.feature9.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[9]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[9]]],axis=1)

if self.feature10.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[10]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[10]]], axis=1)

if self.feature11.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[11]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[11]]], axis=1)

if self.feature12.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[12]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[12]]], axis=1)

if self.feature13.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[13]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[13]]], axis=1)

if self.feature14.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[14]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[14]]], axis=1)

if self.feature15.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[15]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[15]]], axis=1)

if self.feature16.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[16]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[16]]], axis=1)

if self.feature17.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[17]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[17]]], axis=1)

if self.feature18.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[18]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[18]]], axis=1)

if self.feature19.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[19]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[19]]], axis=1)

if self.feature20.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[20]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[20]]],axis=1)

if self.feature21.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[21]]],axis=1)

if self.feature22.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[22]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[22]]],axis=1)

if self.feature23.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[23]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[23]]],axis=1)

if self.feature24.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[24]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[24]]],axis=1)

if self.feature25.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[25]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[25]]],axis=1)

if self.feature26.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[26]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[26]]],axis=1)

if self.feature27.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[27]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[27]]],axis=1)

if self.feature28.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[28]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[28]]],axis=1)

if self.feature29.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[29]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[29]]],axis=1)

if self.feature30.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[30]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[30]]], axis=1)

if self.feature31.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[31]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[31]]], axis=1)

if self.feature32.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[32]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[32]]], axis=1)

if self.feature33.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[33]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[33]]], axis=1)

if self.feature34.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[34]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[34]]], axis=1)

if self.feature35.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[35]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[35]]], axis=1)

vtest\_per = float(self.txtPercentTest.text())

vmax\_depth = float(self.txtMaxDepth.text())

# Clear the graphs to populate them with the new information

self.ax1.clear()

self.ax2.clear()

self.ax3.clear()

self.ax4.clear()

# self.txtResults.clear()

# self.txtResults.setUndoRedoEnabled(False)

vtest\_per = vtest\_per / 100

filename = 'rf\_finalized\_model2.sav'

self.clf\_entropy = pickle.load(open(filename, 'rb'))

y\_test = y

X\_test = X[features\_list]

# -----------------------------------------------------------------------

# predicton on test using entropy

y\_pred\_entropy = self.clf\_entropy.predict(X\_test)

# confusion matrix for RandomForest

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_entropy)

# accuracy score

self.ff\_accuracy\_score = accuracy\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentAccuracy.setText(str(self.ff\_accuracy\_score))

# precision score

self.ff\_precision\_score = precision\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentPrecision.setText(str(self.ff\_precision\_score))

# recall score

self.ff\_recall\_score = recall\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentRecall.setText(str(self.ff\_recall\_score))

# f1\_score

self.ff\_f1\_score = f1\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentF1score.setText(str(self.ff\_f1\_score))

#::------------------------------------

## Ghaph1 :

## Confusion Matrix

#::------------------------------------

class\_names1 = ['', 'No', 'Yes']

self.ax1.matshow(conf\_matrix, cmap=plt.cm.get\_cmap('Blues', 14))

self.ax1.set\_yticklabels(class\_names1)

self.ax1.set\_xticklabels(class\_names1, rotation=90)

self.ax1.set\_xlabel('Predicted label')

self.ax1.set\_ylabel('True label')

for i in range(len(class\_names)):

for j in range(len(class\_names)):

y\_pred\_score = self.clf\_entropy.predict\_proba(X\_test)

self.ax1.text(j, i, str(conf\_matrix[i][j]))

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

#::----------------------------------------

## Graph 2 - ROC Curve

#::----------------------------------------

y\_test\_bin = pd.get\_dummies(y\_test).to\_numpy()

n\_classes = y\_test\_bin.shape[1]

# From the sckict learn site

# https://scikit-learn.org/stable/auto\_examples/model\_selection/plot\_roc.html

fpr = dict()

tpr = dict()

roc\_auc = dict()

for i in range(n\_classes):

fpr[i], tpr[i], \_ = roc\_curve(y\_test\_bin[:, i], y\_pred\_score[:, i])

roc\_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area

fpr["micro"], tpr["micro"], \_ = roc\_curve(y\_test\_bin.ravel(), y\_pred\_score.ravel())

roc\_auc["micro"] = auc(fpr["micro"], tpr["micro"])

lw = 2

self.ax2.plot(fpr[1], tpr[1], color='darkorange',

lw=lw, label='ROC curve (area = %0.2f)' % roc\_auc[1])

self.ax2.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

self.ax2.set\_xlim([0.0, 1.0])

self.ax2.set\_ylim([0.0, 1.05])

self.ax2.set\_xlabel('False Positive Rate')

self.ax2.set\_ylabel('True Positive Rate')

self.ax2.set\_title('ROC Curve Random Forest')

self.ax2.legend(loc="lower right")

self.fig2.tight\_layout()

self.fig2.canvas.draw\_idle()

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

# Graph - 3 Feature Importances

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

# get feature importances

importances = self.clf\_entropy.feature\_importances\_

# convert the importances into one-dimensional 1darray with corresponding df column names as axis labels

f\_importances = pd.Series(importances, self.list\_corr\_features.columns)

# sort the array in descending order of the importances, only show the first 10

f\_importances.sort\_values(ascending=False, inplace=True)

f\_importances = f\_importances[0:10]

X\_Features = f\_importances.index

y\_Importance = list(f\_importances)

self.ax3.barh(X\_Features, y\_Importance)

self.ax3.set\_aspect('auto')

# show the plot

self.fig3.tight\_layout()

self.fig3.canvas.draw\_idle()

#::-----------------------------------------------------

# Graph 4 - ROC Curve by Class

#::-----------------------------------------------------

str\_classes = ['No','Yes']

colors = cycle(['magenta', 'darkorange'])

for i, color in zip(range(n\_classes), colors):

self.ax4.plot(fpr[i], tpr[i], color=color, lw=lw,

label='{0} (area = {1:0.2f})'

''.format(str\_classes[i], roc\_auc[i]))

self.ax4.plot([0, 1], [0, 1], 'k--', lw=lw)

self.ax4.set\_xlim([0.0, 1.0])

self.ax4.set\_ylim([0.0, 1.05])

self.ax4.set\_xlabel('False Positive Rate')

self.ax4.set\_ylabel('True Positive Rate')

self.ax4.set\_title('ROC Curve by Class')

self.ax4.legend(loc="lower right")

# show the plot

self.fig4.tight\_layout()

self.fig4.canvas.draw\_idle()

#::-----------------------------------------------------

# Other Models Comparison

#::-----------------------------------------------------

filename2 = 'lr\_finalized\_model2.sav'

self.other\_clf\_lr = pickle.load(open(filename2, 'rb'))

y\_pred\_lr = self.other\_clf\_lr.predict(X\_test)

self.accuracy\_lr = accuracy\_score(y\_test, y\_pred\_lr) \* 100

self.txtAccuracy\_lr.setText(str(self.accuracy\_lr))

filename3 = 'dt\_finalized\_model2.sav'

self.other\_clf\_dt = pickle.load(open(filename3, 'rb'))

y\_pred\_dt = self.other\_clf\_dt.predict(X\_test)

self.accuracy\_dt = accuracy\_score(y\_test, y\_pred\_dt) \* 100

self.txtAccuracy\_dt.setText(str(self.accuracy\_dt))

filename4 = 'gb\_finalized\_model2.sav'

self.other\_clf\_gb = pickle.load(open(filename4, 'rb'))

y\_pred\_gb = self.other\_clf\_gb.predict(X\_test)

self.accuracy\_gb = accuracy\_score(y\_test, y\_pred\_gb) \* 100

self.txtAccuracy\_gb.setText(str(self.accuracy\_gb))

class LogisticReg(QMainWindow):

#::--------------------------------------------------------------------------------

# Implementation of Random Forest Classifier using the happiness dataset

# the methods in this class are

# \_init\_ : initialize the class

# initUi : creates the canvas and all the elements in the canvas

# update : populates the elements of the canvas base on the parametes

# chosen by the user

#::---------------------------------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

super(LogisticReg, self).\_\_init\_\_()

self.Title = "Logistic Regression Classifier"

self.initUi()

def initUi(self):

#::-----------------------------------------------------------------

# Create the canvas and all the element to create a dashboard with

# all the necessary elements to present the results from the algorithm

# The canvas is divided using a grid loyout to facilitate the drawing

# of the elements

#::-----------------------------------------------------------------

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.main\_widget = QWidget(self)

self.layout = QGridLayout(self.main\_widget)

self.groupBox1 = QGroupBox('Logistic Regression Features')

self.groupBox1Layout= QGridLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.feature0 = QCheckBox(features\_list[0], self)

self.feature1 = QCheckBox(features\_list[1], self)

self.feature2 = QCheckBox(features\_list[2], self)

self.feature3 = QCheckBox(features\_list[3], self)

self.feature4 = QCheckBox(features\_list[4], self)

self.feature5 = QCheckBox(features\_list[5], self)

self.feature6 = QCheckBox(features\_list[6], self)

self.feature7 = QCheckBox(features\_list[7], self)

self.feature8 = QCheckBox(features\_list[8], self)

self.feature9 = QCheckBox(features\_list[9], self)

self.feature10 = QCheckBox(features\_list[10], self)

self.feature11 = QCheckBox(features\_list[11], self)

self.feature12 = QCheckBox(features\_list[12], self)

self.feature13 = QCheckBox(features\_list[13], self)

self.feature14 = QCheckBox(features\_list[14], self)

self.feature15 = QCheckBox(features\_list[15], self)

self.feature16 = QCheckBox(features\_list[16], self)

self.feature17 = QCheckBox(features\_list[17], self)

self.feature18 = QCheckBox(features\_list[18], self)

self.feature19 = QCheckBox(features\_list[19], self)

self.feature20 = QCheckBox(features\_list[20], self)

self.feature21 = QCheckBox(features\_list[21], self)

self.feature22 = QCheckBox(features\_list[22], self)

self.feature23 = QCheckBox(features\_list[23], self)

self.feature24 = QCheckBox(features\_list[24], self)

self.feature25 = QCheckBox(features\_list[25], self)

self.feature26 = QCheckBox(features\_list[26], self)

self.feature27 = QCheckBox(features\_list[27], self)

self.feature28 = QCheckBox(features\_list[28], self)

self.feature29 = QCheckBox(features\_list[29], self)

self.feature30 = QCheckBox(features\_list[30], self)

self.feature31 = QCheckBox(features\_list[31], self)

self.feature32 = QCheckBox(features\_list[32], self)

self.feature33 = QCheckBox(features\_list[33], self)

self.feature34 = QCheckBox(features\_list[34], self)

self.feature35 = QCheckBox(features\_list[35], self)

self.feature0.setChecked(True)

self.feature1.setChecked(True)

self.feature2.setChecked(True)

self.feature3.setChecked(True)

self.feature4.setChecked(True)

self.feature5.setChecked(True)

self.feature6.setChecked(True)

self.feature7.setChecked(True)

self.feature8.setChecked(True)

self.feature9.setChecked(True)

self.feature10.setChecked(True)

self.feature11.setChecked(True)

self.feature12.setChecked(True)

self.feature13.setChecked(True)

self.feature14.setChecked(True)

self.feature15.setChecked(True)

self.feature16.setChecked(True)

self.feature17.setChecked(True)

self.feature18.setChecked(True)

self.feature19.setChecked(True)

self.feature20.setChecked(True)

self.feature21.setChecked(True)

self.feature22.setChecked(True)

self.feature23.setChecked(True)

self.feature24.setChecked(True)

self.feature25.setChecked(True)

self.feature26.setChecked(True)

self.feature27.setChecked(True)

self.feature28.setChecked(True)

self.feature29.setChecked(True)

self.feature30.setChecked(True)

self.feature31.setChecked(True)

self.feature32.setChecked(True)

self.feature33.setChecked(True)

self.feature34.setChecked(True)

self.feature35.setChecked(True)

self.lblPercentTest = QLabel('Percentage for Test :')

self.lblPercentTest.adjustSize()

self.txtPercentTest = QLineEdit(self)

self.txtPercentTest.setText("30")

self.btnExecute = QPushButton("Run Model")

self.btnExecute.setGeometry(QRect(60, 500, 75, 23))

self.btnExecute.clicked.connect(self.update)

# We create a checkbox for each feature

self.groupBox1Layout.addWidget(self.feature0, 0, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature1, 0, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature2, 1, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature3, 1, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature4, 2, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature5, 2, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature6, 3, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature7, 3, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature8, 4, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature9, 4, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature10, 5, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature11, 5, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature12, 6, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature13, 6, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature14, 7, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature15, 7, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature16, 8, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature17, 8, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature18, 9, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature19, 9, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature20, 10, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature21, 10, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature22, 11, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature23, 11, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature24, 12, 0, 1, 1)

self.groupBox1Layout.addWidget(self.lblPercentTest, 19, 0, 1, 1)

self.groupBox1Layout.addWidget(self.txtPercentTest, 19, 1, 1, 1)

self.groupBox1Layout.addWidget(self.btnExecute, 21, 0, 1, 1)

self.groupBox2 = QGroupBox('Measurements:')

self.groupBox2Layout = QVBoxLayout()

self.groupBox2.setLayout(self.groupBox2Layout)

# self.groupBox2.setMinimumSize(400, 100)

self.current\_model\_summary = QWidget(self)

self.current\_model\_summary.layout = QFormLayout(self.current\_model\_summary)

self.txtCurrentAccuracy = QLineEdit()

self.txtCurrentPrecision = QLineEdit()

self.txtCurrentRecall = QLineEdit()

self.txtCurrentF1score = QLineEdit()

self.current\_model\_summary.layout.addRow('Accuracy:', self.txtCurrentAccuracy)

self.current\_model\_summary.layout.addRow('Precision:', self.txtCurrentPrecision)

self.current\_model\_summary.layout.addRow('Recall:', self.txtCurrentRecall)

self.current\_model\_summary.layout.addRow('F1 Score:', self.txtCurrentF1score)

self.groupBox2Layout.addWidget(self.current\_model\_summary)

self.groupBox3 = QGroupBox('Other Models Accuracy:')

self.groupBox3Layout = QVBoxLayout()

self.groupBox3.setLayout(self.groupBox3Layout)

self.other\_models = QWidget(self)

self.other\_models.layout = QFormLayout(self.other\_models)

self.txtAccuracy\_dt = QLineEdit()

self.txtAccuracy\_gb = QLineEdit()

self.txtAccuracy\_rf = QLineEdit()

self.other\_models.layout.addRow('Decision Tree:', self.txtAccuracy\_dt)

self.other\_models.layout.addRow('Gradient Boosting:', self.txtAccuracy\_gb)

self.other\_models.layout.addRow('Random Forest:', self.txtAccuracy\_rf)

self.groupBox3Layout.addWidget(self.other\_models)

#::-------------------------------------

# Graphic 1 : Confusion Matrix

#::-------------------------------------

self.fig = Figure()

self.ax1 = self.fig.add\_subplot(111)

self.axes=[self.ax1]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.groupBoxG1 = QGroupBox('Confusion Matrix')

self.groupBoxG1Layout= QVBoxLayout()

self.groupBoxG1.setLayout(self.groupBoxG1Layout)

self.groupBoxG1Layout.addWidget(self.canvas)

#::---------------------------------------------

# Graphic 2 : ROC Curve

#::---------------------------------------------

self.fig2 = Figure()

self.ax2 = self.fig2.add\_subplot(111)

self.axes2 = [self.ax2]

self.canvas2 = FigureCanvas(self.fig2)

self.canvas2.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas2.updateGeometry()

self.groupBoxG2 = QGroupBox('ROC Curve')

self.groupBoxG2Layout = QVBoxLayout()

self.groupBoxG2.setLayout(self.groupBoxG2Layout)

self.groupBoxG2Layout.addWidget(self.canvas2)

#::-------------------------------------------

# Graphic 3 : k-fold Cross validation

#::-------------------------------------------

self.fig3 = Figure()

self.ax3 = self.fig3.add\_subplot(111)

self.axes3 = [self.ax3]

self.canvas3 = FigureCanvas(self.fig3)

self.canvas3.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas3.updateGeometry()

self.groupBoxG3 = QGroupBox('K-fold cross validation')

self.groupBoxG3Layout = QVBoxLayout()

self.groupBoxG3.setLayout(self.groupBoxG3Layout)

self.groupBoxG3Layout.addWidget(self.canvas3)

#::--------------------------------------------

# Graphic 4 : ROC Curve by class

#::--------------------------------------------

self.fig4 = Figure()

self.ax4 = self.fig4.add\_subplot(111)

self.axes4 = [self.ax4]

self.canvas4 = FigureCanvas(self.fig4)

self.canvas4.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas4.updateGeometry()

self.groupBoxG4 = QGroupBox('ROC Curve by Class')

self.groupBoxG4Layout = QVBoxLayout()

self.groupBoxG4.setLayout(self.groupBoxG4Layout)

self.groupBoxG4Layout.addWidget(self.canvas4)

#::-------------------------------------------------

# End of graphs

#::-------------------------------------------------

self.layout.addWidget(self.groupBox1, 0, 0, 3, 2)

self.layout.addWidget(self.groupBoxG1, 0, 2, 1, 1)

self.layout.addWidget(self.groupBoxG3, 0, 3, 1, 1)

self.layout.addWidget(self.groupBoxG2, 1, 2, 1, 1)

self.layout.addWidget(self.groupBoxG4, 1, 3, 1, 1)

self.layout.addWidget(self.groupBox2, 2, 2, 1, 1)

self.layout.addWidget(self.groupBox3, 2, 3, 1, 1)

self.setCentralWidget(self.main\_widget)

self.resize(1800, 1200)

self.show()

def update(self):

'''

Random Forest Classifier

We pupulate the dashboard using the parametres chosen by the user

The parameters are processed to execute in the skit-learn Random Forest algorithm

then the results are presented in graphics and reports in the canvas

:return:None

'''

# processing the parameters

self.list\_corr\_features = pd.DataFrame([])

if self.feature0.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[0]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[0]]],axis=1)

if self.feature1.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[1]]],axis=1)

if self.feature2.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[2]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[2]]],axis=1)

if self.feature3.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[3]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[3]]],axis=1)

if self.feature4.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[4]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[4]]],axis=1)

if self.feature5.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[5]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[5]]],axis=1)

if self.feature6.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[6]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[6]]],axis=1)

if self.feature7.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[7]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[7]]],axis=1)

if self.feature8.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[8]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[8]]],axis=1)

if self.feature9.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[9]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[9]]],axis=1)

if self.feature10.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[10]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[10]]], axis=1)

if self.feature11.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[11]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[11]]], axis=1)

if self.feature12.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[12]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[12]]], axis=1)

if self.feature13.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[13]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[13]]], axis=1)

if self.feature14.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[14]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[14]]], axis=1)

if self.feature15.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[15]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[15]]], axis=1)

if self.feature16.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[16]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[16]]], axis=1)

if self.feature17.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[17]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[17]]], axis=1)

if self.feature18.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[18]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[18]]], axis=1)

if self.feature19.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[19]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[19]]], axis=1)

if self.feature20.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[20]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[20]]],axis=1)

if self.feature21.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[21]]],axis=1)

if self.feature22.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[22]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[22]]],axis=1)

if self.feature23.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[23]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[23]]],axis=1)

if self.feature24.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[24]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[24]]],axis=1)

if self.feature25.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[25]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[25]]],axis=1)

if self.feature26.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[26]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[26]]],axis=1)

if self.feature27.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[27]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[27]]],axis=1)

if self.feature28.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[28]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[28]]],axis=1)

if self.feature29.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[29]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[29]]],axis=1)

if self.feature30.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[30]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[30]]], axis=1)

if self.feature31.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[31]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[31]]], axis=1)

if self.feature32.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[32]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[32]]], axis=1)

if self.feature33.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[33]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[33]]], axis=1)

if self.feature34.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[34]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[34]]], axis=1)

if self.feature35.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[35]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[35]]], axis=1)

vtest\_per = float(self.txtPercentTest.text())

# Clear the graphs to populate them with the new information

self.ax1.clear()

self.ax2.clear()

self.ax3.clear()

self.ax4.clear()

# self.txtResults.clear()

# self.txtResults.setUndoRedoEnabled(False)

vtest\_per = vtest\_per / 100

filename = 'lr\_finalized\_model2.sav'

self.clf\_entropy = pickle.load(open(filename, 'rb'))

y\_test = y

X\_test = X[features\_list]

# -----------------------------------------------------------------------

# predicton on test using entropy

y\_pred\_entropy = self.clf\_entropy.predict(X\_test)

# confusion matrix for RandomForest

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_entropy)

# accuracy score

self.ff\_accuracy\_score = accuracy\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentAccuracy.setText(str(self.ff\_accuracy\_score))

# precision score

self.ff\_precision\_score = precision\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentPrecision.setText(str(self.ff\_precision\_score))

# recall score

self.ff\_recall\_score = recall\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentRecall.setText(str(self.ff\_recall\_score))

# f1\_score

self.ff\_f1\_score = f1\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentF1score.setText(str(self.ff\_f1\_score))

#::------------------------------------

## Ghaph1 :

## Confusion Matrix

#::------------------------------------

class\_names1 = ['', 'No', 'Yes']

self.ax1.matshow(conf\_matrix, cmap=plt.cm.get\_cmap('Blues', 14))

self.ax1.set\_yticklabels(class\_names1)

self.ax1.set\_xticklabels(class\_names1, rotation=90)

self.ax1.set\_xlabel('Predicted label')

self.ax1.set\_ylabel('True label')

for i in range(len(class\_names)):

for j in range(len(class\_names)):

y\_pred\_score = self.clf\_entropy.predict\_proba(X\_test)

self.ax1.text(j, i, str(conf\_matrix[i][j]))

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

#::----------------------------------------

## Graph 2 - ROC Curve

#::----------------------------------------

y\_test\_bin = pd.get\_dummies(y\_test).to\_numpy()

n\_classes = y\_test\_bin.shape[1]

# From the sckict learn site

# https://scikit-learn.org/stable/auto\_examples/model\_selection/plot\_roc.html

fpr = dict()

tpr = dict()

roc\_auc = dict()

for i in range(n\_classes):

fpr[i], tpr[i], \_ = roc\_curve(y\_test\_bin[:, i], y\_pred\_score[:, i])

roc\_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area

fpr["micro"], tpr["micro"], \_ = roc\_curve(y\_test\_bin.ravel(), y\_pred\_score.ravel())

roc\_auc["micro"] = auc(fpr["micro"], tpr["micro"])

lw = 2

self.ax2.plot(fpr[1], tpr[1], color='darkorange',

lw=lw, label='ROC curve (area = %0.2f)' % roc\_auc[1])

self.ax2.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

self.ax2.set\_xlim([0.0, 1.0])

self.ax2.set\_ylim([0.0, 1.05])

self.ax2.set\_xlabel('False Positive Rate')

self.ax2.set\_ylabel('True Positive Rate')

self.ax2.set\_title('ROC Curve Logistic Regression')

self.ax2.legend(loc="lower right")

self.fig2.tight\_layout()

self.fig2.canvas.draw\_idle()

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

# Graph - cross validation

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

# get cross validation

score2=cross\_val\_score(self.clf\_entropy,X\_test,cv=5,y=y\_test,scoring='accuracy',n\_jobs=-1)

# repeats=range(1,15)

# results=list()

# for r in repeats:

self.ax3.boxplot(score2)

self.ax3.set\_aspect('auto')

# show the plot

self.fig3.tight\_layout()

self.fig3.canvas.draw\_idle()

#::-----------------------------------------------------

# Graph 4 - ROC Curve by Class

#::-----------------------------------------------------

str\_classes = ['No','Yes']

colors = cycle(['magenta', 'darkorange'])

for i, color in zip(range(n\_classes), colors):

self.ax4.plot(fpr[i], tpr[i], color=color, lw=lw,

label='{0} (area = {1:0.2f})'

''.format(str\_classes[i], roc\_auc[i]))

self.ax4.plot([0, 1], [0, 1], 'k--', lw=lw)

self.ax4.set\_xlim([0.0, 1.0])

self.ax4.set\_ylim([0.0, 1.05])

self.ax4.set\_xlabel('False Positive Rate')

self.ax4.set\_ylabel('True Positive Rate')

self.ax4.set\_title('ROC Curve by Class')

self.ax4.legend(loc="lower right")

# show the plot

self.fig4.tight\_layout()

self.fig4.canvas.draw\_idle()

#::-----------------------------------------------------

# Other Models Comparison

#::-----------------------------------------------------

filename2 = 'dt\_finalized\_model2.sav'

self.other\_clf\_dt = pickle.load(open(filename2, 'rb'))

y\_pred\_dt = self.other\_clf\_dt.predict(X\_test)

self.accuracy\_dt = accuracy\_score(y\_test, y\_pred\_dt) \* 100

self.txtAccuracy\_dt.setText(str(self.accuracy\_dt))

filename3 = 'rf\_finalized\_model2.sav'

self.other\_clf\_rf = pickle.load(open(filename3, 'rb'))

y\_pred\_rf = self.other\_clf\_rf.predict(X\_test)

self.accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf) \* 100

self.txtAccuracy\_rf.setText(str(self.accuracy\_rf))

filename4 = 'gb\_finalized\_model2.sav'

self.other\_clf\_gb = pickle.load(open(filename4, 'rb'))

y\_pred\_gb = self.other\_clf\_gb.predict(X\_test)

self.accuracy\_gb = accuracy\_score(y\_test, y\_pred\_gb) \* 100

self.txtAccuracy\_gb.setText(str(self.accuracy\_gb))

class GradientBoosting(QMainWindow):

#::--------------------------------------------------------------------------------

# Implementation of Random Forest Classifier using the happiness dataset

# the methods in this class are

# \_init\_ : initialize the class

# initUi : creates the canvas and all the elements in the canvas

# update : populates the elements of the canvas base on the parametes

# chosen by the user

#::---------------------------------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

super(GradientBoosting, self).\_\_init\_\_()

self.Title = "Gradient Boosting Classifier"

self.initUi()

def initUi(self):

#::-----------------------------------------------------------------

# Create the canvas and all the element to create a dashboard with

# all the necessary elements to present the results from the algorithm

# The canvas is divided using a grid loyout to facilitate the drawing

# of the elements

#::-----------------------------------------------------------------

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.main\_widget = QWidget(self)

self.layout = QGridLayout(self.main\_widget)

self.groupBox1 = QGroupBox('Gradient Boosting Features')

self.groupBox1Layout= QGridLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.feature0 = QCheckBox(features\_list[0], self)

self.feature1 = QCheckBox(features\_list[1], self)

self.feature2 = QCheckBox(features\_list[2], self)

self.feature3 = QCheckBox(features\_list[3], self)

self.feature4 = QCheckBox(features\_list[4], self)

self.feature5 = QCheckBox(features\_list[5], self)

self.feature6 = QCheckBox(features\_list[6], self)

self.feature7 = QCheckBox(features\_list[7], self)

self.feature8 = QCheckBox(features\_list[8], self)

self.feature9 = QCheckBox(features\_list[9], self)

self.feature10 = QCheckBox(features\_list[10], self)

self.feature11 = QCheckBox(features\_list[11], self)

self.feature12 = QCheckBox(features\_list[12], self)

self.feature13 = QCheckBox(features\_list[13], self)

self.feature14 = QCheckBox(features\_list[14], self)

self.feature15 = QCheckBox(features\_list[15], self)

self.feature16 = QCheckBox(features\_list[16], self)

self.feature17 = QCheckBox(features\_list[17], self)

self.feature18 = QCheckBox(features\_list[18], self)

self.feature19 = QCheckBox(features\_list[19], self)

self.feature20 = QCheckBox(features\_list[20], self)

self.feature21 = QCheckBox(features\_list[21], self)

self.feature22 = QCheckBox(features\_list[22], self)

self.feature23 = QCheckBox(features\_list[23], self)

self.feature24 = QCheckBox(features\_list[24], self)

self.feature25 = QCheckBox(features\_list[25], self)

self.feature26 = QCheckBox(features\_list[26], self)

self.feature27 = QCheckBox(features\_list[27], self)

self.feature28 = QCheckBox(features\_list[28], self)

self.feature29 = QCheckBox(features\_list[29], self)

self.feature30 = QCheckBox(features\_list[30], self)

self.feature31 = QCheckBox(features\_list[31], self)

self.feature32 = QCheckBox(features\_list[32], self)

self.feature33 = QCheckBox(features\_list[33], self)

self.feature34 = QCheckBox(features\_list[34], self)

self.feature35 = QCheckBox(features\_list[35], self)

self.feature0.setChecked(True)

self.feature1.setChecked(True)

self.feature2.setChecked(True)

self.feature3.setChecked(True)

self.feature4.setChecked(True)

self.feature5.setChecked(True)

self.feature6.setChecked(True)

self.feature7.setChecked(True)

self.feature8.setChecked(True)

self.feature9.setChecked(True)

self.feature10.setChecked(True)

self.feature11.setChecked(True)

self.feature12.setChecked(True)

self.feature13.setChecked(True)

self.feature14.setChecked(True)

self.feature15.setChecked(True)

self.feature16.setChecked(True)

self.feature17.setChecked(True)

self.feature18.setChecked(True)

self.feature19.setChecked(True)

self.feature20.setChecked(True)

self.feature21.setChecked(True)

self.feature22.setChecked(True)

self.feature23.setChecked(True)

self.feature24.setChecked(True)

self.feature25.setChecked(True)

self.feature26.setChecked(True)

self.feature27.setChecked(True)

self.feature28.setChecked(True)

self.feature29.setChecked(True)

self.feature30.setChecked(True)

self.feature31.setChecked(True)

self.feature32.setChecked(True)

self.feature33.setChecked(True)

self.feature34.setChecked(True)

self.feature35.setChecked(True)

self.lblPercentTest = QLabel('Percentage for Test :')

self.lblPercentTest.adjustSize()

self.txtPercentTest = QLineEdit(self)

self.txtPercentTest.setText("30")

self.btnExecute = QPushButton("Run Model")

self.btnExecute.setGeometry(QRect(60, 500, 75, 23))

self.btnExecute.clicked.connect(self.update)

# We create a checkbox for each feature

self.groupBox1Layout.addWidget(self.feature0, 0, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature1, 0, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature2, 1, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature3, 1, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature4, 2, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature5, 2, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature6, 3, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature7, 3, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature8, 4, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature9, 4, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature10, 5, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature11, 5, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature12, 6, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature13, 6, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature14, 7, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature15, 7, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature16, 8, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature17, 8, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature18, 9, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature19, 9, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature20, 10, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature21, 10, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature22, 11, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature23, 11, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature24, 12, 0, 1, 1)

self.groupBox1Layout.addWidget(self.lblPercentTest, 19, 0, 1, 1)

self.groupBox1Layout.addWidget(self.txtPercentTest, 19, 1, 1, 1)

self.groupBox1Layout.addWidget(self.btnExecute, 21, 0, 1, 1)

self.groupBox2 = QGroupBox('Measurements:')

self.groupBox2Layout = QVBoxLayout()

self.groupBox2.setLayout(self.groupBox2Layout)

# self.groupBox2.setMinimumSize(400, 100)

self.current\_model\_summary = QWidget(self)

self.current\_model\_summary.layout = QFormLayout(self.current\_model\_summary)

self.txtCurrentAccuracy = QLineEdit()

self.txtCurrentPrecision = QLineEdit()

self.txtCurrentRecall = QLineEdit()

self.txtCurrentF1score = QLineEdit()

self.current\_model\_summary.layout.addRow('Accuracy:', self.txtCurrentAccuracy)

self.current\_model\_summary.layout.addRow('Precision:', self.txtCurrentPrecision)

self.current\_model\_summary.layout.addRow('Recall:', self.txtCurrentRecall)

self.current\_model\_summary.layout.addRow('F1 Score:', self.txtCurrentF1score)

self.groupBox2Layout.addWidget(self.current\_model\_summary)

self.groupBox3 = QGroupBox('Other Models Accuracy:')

self.groupBox3Layout = QVBoxLayout()

self.groupBox3.setLayout(self.groupBox3Layout)

self.other\_models = QWidget(self)

self.other\_models.layout = QFormLayout(self.other\_models)

self.txtAccuracy\_lr = QLineEdit()

self.txtAccuracy\_dt = QLineEdit()

self.txtAccuracy\_rf = QLineEdit()

self.other\_models.layout.addRow('Decision Tree:', self.txtAccuracy\_dt)

self.other\_models.layout.addRow('Logistic Regression:', self.txtAccuracy\_lr)

self.other\_models.layout.addRow('Random Forest:', self.txtAccuracy\_rf)

self.groupBox3Layout.addWidget(self.other\_models)

#::-------------------------------------

# Graphic 1 : Confusion Matrix

#::-------------------------------------

self.fig = Figure()

self.ax1 = self.fig.add\_subplot(111)

self.axes=[self.ax1]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.groupBoxG1 = QGroupBox('Confusion Matrix')

self.groupBoxG1Layout= QVBoxLayout()

self.groupBoxG1.setLayout(self.groupBoxG1Layout)

self.groupBoxG1Layout.addWidget(self.canvas)

#::---------------------------------------------

# Graphic 2 : ROC Curve

#::---------------------------------------------

self.fig2 = Figure()

self.ax2 = self.fig2.add\_subplot(111)

self.axes2 = [self.ax2]

self.canvas2 = FigureCanvas(self.fig2)

self.canvas2.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas2.updateGeometry()

self.groupBoxG2 = QGroupBox('ROC Curve')

self.groupBoxG2Layout = QVBoxLayout()

self.groupBoxG2.setLayout(self.groupBoxG2Layout)

self.groupBoxG2Layout.addWidget(self.canvas2)

#::-------------------------------------------

# Graphic 3 : k-fold Cross validation

#::-------------------------------------------

self.fig3 = Figure()

self.ax3 = self.fig3.add\_subplot(111)

self.axes3 = [self.ax3]

self.canvas3 = FigureCanvas(self.fig3)

self.canvas3.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas3.updateGeometry()

self.groupBoxG3 = QGroupBox('K-fold cross validation')

self.groupBoxG3Layout = QVBoxLayout()

self.groupBoxG3.setLayout(self.groupBoxG3Layout)

self.groupBoxG3Layout.addWidget(self.canvas3)

#::--------------------------------------------

# Graphic 4 : ROC Curve by class

#::--------------------------------------------

self.fig4 = Figure()

self.ax4 = self.fig4.add\_subplot(111)

self.axes4 = [self.ax4]

self.canvas4 = FigureCanvas(self.fig4)

self.canvas4.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)

self.canvas4.updateGeometry()

self.groupBoxG4 = QGroupBox('ROC Curve by Class')

self.groupBoxG4Layout = QVBoxLayout()

self.groupBoxG4.setLayout(self.groupBoxG4Layout)

self.groupBoxG4Layout.addWidget(self.canvas4)

#::-------------------------------------------------

# End of graphs

#::-------------------------------------------------

self.layout.addWidget(self.groupBox1, 0, 0, 3, 2)

self.layout.addWidget(self.groupBoxG1, 0, 2, 1, 1)

self.layout.addWidget(self.groupBoxG3, 0, 3, 1, 1)

self.layout.addWidget(self.groupBoxG2, 1, 2, 1, 1)

self.layout.addWidget(self.groupBoxG4, 1, 3, 1, 1)

self.layout.addWidget(self.groupBox2, 2, 2, 1, 1)

self.layout.addWidget(self.groupBox3, 2, 3, 1, 1)

self.setCentralWidget(self.main\_widget)

self.resize(1800, 1200)

self.show()

def update(self):

'''

Random Forest Classifier

We pupulate the dashboard using the parametres chosen by the user

The parameters are processed to execute in the skit-learn Random Forest algorithm

then the results are presented in graphics and reports in the canvas

:return:None

'''

# processing the parameters

self.list\_corr\_features = pd.DataFrame([])

if self.feature0.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[0]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[0]]],axis=1)

if self.feature1.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[1]]],axis=1)

if self.feature2.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[2]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[2]]],axis=1)

if self.feature3.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[3]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[3]]],axis=1)

if self.feature4.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[4]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[4]]],axis=1)

if self.feature5.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[5]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[5]]],axis=1)

if self.feature6.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[6]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[6]]],axis=1)

if self.feature7.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[7]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[7]]],axis=1)

if self.feature8.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[8]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[8]]],axis=1)

if self.feature9.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[9]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[9]]],axis=1)

if self.feature10.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[10]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[10]]], axis=1)

if self.feature11.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[11]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[11]]], axis=1)

if self.feature12.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[12]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[12]]], axis=1)

if self.feature13.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[13]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[13]]], axis=1)

if self.feature14.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[14]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[14]]], axis=1)

if self.feature15.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[15]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[15]]], axis=1)

if self.feature16.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[16]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[16]]], axis=1)

if self.feature17.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[17]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[17]]], axis=1)

if self.feature18.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[18]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[18]]], axis=1)

if self.feature19.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[19]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[19]]], axis=1)

if self.feature20.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[20]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[20]]],axis=1)

if self.feature21.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[21]]],axis=1)

if self.feature22.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[22]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[22]]],axis=1)

if self.feature23.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[23]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[23]]],axis=1)

if self.feature24.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[24]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[24]]],axis=1)

if self.feature25.isChecked():

if len(self.list\_corr\_features)==0:

self.list\_corr\_features = df[features\_list[25]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[25]]],axis=1)

if self.feature26.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[26]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[26]]],axis=1)

if self.feature27.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[27]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[27]]],axis=1)

if self.feature28.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[28]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[28]]],axis=1)

if self.feature29.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[29]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[29]]],axis=1)

if self.feature30.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[30]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[30]]], axis=1)

if self.feature31.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[31]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[31]]], axis=1)

if self.feature32.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[32]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[32]]], axis=1)

if self.feature33.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[33]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[33]]], axis=1)

if self.feature34.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[34]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[34]]], axis=1)

if self.feature35.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[35]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[35]]], axis=1)

vtest\_per = float(self.txtPercentTest.text())

# Clear the graphs to populate them with the new information

self.ax1.clear()

self.ax2.clear()

self.ax3.clear()

self.ax4.clear()

# self.txtResults.clear()

# self.txtResults.setUndoRedoEnabled(False)

vtest\_per = vtest\_per / 100

filename = 'gb\_finalized\_model2.sav'

self.clf\_entropy = pickle.load(open(filename, 'rb'))

y\_test = y

X\_test = X[features\_list]

# -----------------------------------------------------------------------

# predicton on test using entropy

y\_pred\_entropy = self.clf\_entropy.predict(X\_test)

# confusion matrix for RandomForest

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_entropy)

# accuracy score

self.ff\_accuracy\_score = accuracy\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentAccuracy.setText(str(self.ff\_accuracy\_score))

# precision score

self.ff\_precision\_score = precision\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentPrecision.setText(str(self.ff\_precision\_score))

# recall score

self.ff\_recall\_score = recall\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentRecall.setText(str(self.ff\_recall\_score))

# f1\_score

self.ff\_f1\_score = f1\_score(y\_test, y\_pred\_entropy) \* 100

self.txtCurrentF1score.setText(str(self.ff\_f1\_score))

#::------------------------------------

## Ghaph1 :

## Confusion Matrix

#::------------------------------------

class\_names1 = ['', 'No', 'Yes']

self.ax1.matshow(conf\_matrix, cmap=plt.cm.get\_cmap('Blues', 14))

self.ax1.set\_yticklabels(class\_names1)

self.ax1.set\_xticklabels(class\_names1, rotation=90)

self.ax1.set\_xlabel('Predicted label')

self.ax1.set\_ylabel('True label')

for i in range(len(class\_names)):

for j in range(len(class\_names)):

y\_pred\_score = self.clf\_entropy.predict\_proba(X\_test)

self.ax1.text(j, i, str(conf\_matrix[i][j]))

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

#::----------------------------------------

## Graph 2 - ROC Curve

#::----------------------------------------

y\_test\_bin = pd.get\_dummies(y\_test).to\_numpy()

n\_classes = y\_test\_bin.shape[1]

# From the sckict learn site

# https://scikit-learn.org/stable/auto\_examples/model\_selection/plot\_roc.html

fpr = dict()

tpr = dict()

roc\_auc = dict()

for i in range(n\_classes):

fpr[i], tpr[i], \_ = roc\_curve(y\_test\_bin[:, i], y\_pred\_score[:, i])

roc\_auc[i] = auc(fpr[i], tpr[i])

# Compute micro-average ROC curve and ROC area

fpr["micro"], tpr["micro"], \_ = roc\_curve(y\_test\_bin.ravel(), y\_pred\_score.ravel())

roc\_auc["micro"] = auc(fpr["micro"], tpr["micro"])

lw = 2

self.ax2.plot(fpr[1], tpr[1], color='darkorange',

lw=lw, label='ROC curve (area = %0.2f)' % roc\_auc[1])

self.ax2.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')

self.ax2.set\_xlim([0.0, 1.0])

self.ax2.set\_ylim([0.0, 1.05])

self.ax2.set\_xlabel('False Positive Rate')

self.ax2.set\_ylabel('True Positive Rate')

self.ax2.set\_title('ROC Curve Gradient Boosting')

self.ax2.legend(loc="lower right")

self.fig2.tight\_layout()

self.fig2.canvas.draw\_idle()

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

# Graph - cross validation

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

# get cross validation

score2=cross\_val\_score(self.clf\_entropy,X\_test,cv=5,y=y\_test,scoring='accuracy',n\_jobs=-1)

# repeats=range(1,15)

# results=list()

# for r in repeats:

self.ax3.boxplot(score2)

self.ax3.set\_aspect('auto')

# show the plot

self.fig3.tight\_layout()

self.fig3.canvas.draw\_idle()

#::-----------------------------------------------------

# Graph 4 - ROC Curve by Class

#::-----------------------------------------------------

str\_classes = ['No','Yes']

colors = cycle(['magenta', 'darkorange'])

for i, color in zip(range(n\_classes), colors):

self.ax4.plot(fpr[i], tpr[i], color=color, lw=lw,

label='{0} (area = {1:0.2f})'

''.format(str\_classes[i], roc\_auc[i]))

self.ax4.plot([0, 1], [0, 1], 'k--', lw=lw)

self.ax4.set\_xlim([0.0, 1.0])

self.ax4.set\_ylim([0.0, 1.05])

self.ax4.set\_xlabel('False Positive Rate')

self.ax4.set\_ylabel('True Positive Rate')

self.ax4.set\_title('ROC Curve by Class')

self.ax4.legend(loc="lower right")

# show the plot

self.fig4.tight\_layout()

self.fig4.canvas.draw\_idle()

#::-----------------------------------------------------

# Other Models Comparison

#::-----------------------------------------------------

filename2 = 'lr\_finalized\_model2.sav'

self.other\_clf\_lr = pickle.load(open(filename2, 'rb'))

y\_pred\_lr = self.other\_clf\_lr.predict(X\_test)

self.accuracy\_lr = accuracy\_score(y\_test, y\_pred\_lr) \* 100

self.txtAccuracy\_lr.setText(str(self.accuracy\_lr))

filename3 = 'rf\_finalized\_model2.sav'

self.other\_clf\_rf = pickle.load(open(filename3, 'rb'))

y\_pred\_rf = self.other\_clf\_rf.predict(X\_test)

self.accuracy\_rf = accuracy\_score(y\_test, y\_pred\_rf) \* 100

self.txtAccuracy\_rf.setText(str(self.accuracy\_rf))

filename4 = 'dt\_finalized\_model2.sav'

self.other\_clf\_dt = pickle.load(open(filename4, 'rb'))

y\_pred\_dt = self.other\_clf\_dt.predict(X\_test)

self.accuracy\_dt = accuracy\_score(y\_test, y\_pred\_dt) \* 100

self.txtAccuracy\_dt.setText(str(self.accuracy\_dt))

class TargetDistribution(QMainWindow):

#::---------------------------------------------------------

# This class crates a canvas with a plot to show the distribution

# from each feature in the dataset with the target variables

# methods

# \_init\_

# update

#::---------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

#::--------------------------------------------------------

# Crate a canvas with the layout to draw a dotplot

# The layout sets all the elements and manage the changes

# made on the canvas

#::--------------------------------------------------------

super(TargetDistribution, self).\_\_init\_\_()

self.Title = "EDA: Variable Distribution"

self.main\_widget = QWidget(self)

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.fig = Figure()

self.ax = self.fig.add\_subplot(111)

self.axes = [self.ax]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding,

QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.dropdown1 = QComboBox()

self.featuresList = numerical.copy()

self.dropdown1.addItems(self.featuresList)

self.dropdown1.currentIndexChanged.connect(self.update)

self.label = QLabel("A plot:")

self.layout = QGridLayout(self.main\_widget)

self.layout.addWidget(QLabel("Select Features:"), 0, 0, 1, 1)

self.layout.addWidget(self.dropdown1, 0, 1, 1, 1)

self.filter\_data = QWidget(self)

self.filter\_data.layout = QGridLayout(self.filter\_data)

self.filter\_data.layout.addWidget(QLabel("Choose Data Filter:"), 0, 0, 1, 1)

self.filter\_radio\_button = QRadioButton("All Data")

self.filter\_radio\_button.setChecked(True)

self.filter\_radio\_button.filter = "All\_Data"

self.set\_Filter = "All\_Data"

self.filter\_radio\_button.toggled.connect(self.onFilterClicked)

self.filter\_data.layout.addWidget(self.filter\_radio\_button, 0, 1, 1, 1)

self.filter\_radio\_button = QRadioButton("Loan Default: Yes")

self.filter\_radio\_button.filter = 1

self.filter\_radio\_button.toggled.connect(self.onFilterClicked)

self.filter\_data.layout.addWidget(self.filter\_radio\_button, 0, 2, 1, 1)

self.filter\_radio\_button = QRadioButton("Loan Default: No")

self.filter\_radio\_button.filter = 0

self.filter\_radio\_button.toggled.connect(self.onFilterClicked)

self.filter\_data.layout.addWidget(self.filter\_radio\_button, 0, 3, 1, 1)

self.btnCreateGraph = QPushButton("Show Distribution")

self.btnCreateGraph.clicked.connect(self.update)

self.groupBox1 = QGroupBox('Distribution')

self.groupBox1Layout = QVBoxLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.groupBox1Layout.addWidget(self.canvas)

self.layout.addWidget(self.filter\_data, 1, 0, 2, 2)

self.layout.addWidget(self.btnCreateGraph, 0, 3, 2, 2)

self.layout.addWidget(self.groupBox1, 3, 0, 5, 5)

self.setCentralWidget(self.main\_widget)

self.resize(1200, 700)

self.show()

def onFilterClicked(self):

self.filter\_radio\_button = self.sender()

if self.filter\_radio\_button.isChecked():

self.set\_Filter = self.filter\_radio\_button.filter

self.update()

def update(self):

#::--------------------------------------------------------

# This method executes each time a change is made on the canvas

# containing the elements of the graph

# The purpose of the method es to draw a dot graph using the

# score of happiness and the feature chosen the canvas

#::--------------------------------------------------------

colors = ["b", "r", "g", "y", "k", "c"]

self.ax.clear()

cat1 = self.dropdown1.currentText()

if (self.set\_Filter == 1 or self.set\_Filter == 0):

self.filtered\_data = df\_orig.copy()

self.filtered\_data = self.filtered\_data[self.filtered\_data["loan\_default"] == self.set\_Filter]

else:

self.filtered\_data = df\_orig.copy()

self.ax.hist(self.filtered\_data[cat1], bins=50, facecolor='blue', alpha=0.5)

self.ax.set\_title(cat1)

self.ax.set\_xlabel(cat1)

self.ax.set\_ylabel("Count")

self.ax.grid(True)

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

del cat1

del self.filtered\_data

class TargetCount(QMainWindow):

#::---------------------------------------------------------

# This class crates a canvas with a plot to show the distribution

# from each feature in the dataset with the target variables

# methods

# \_init\_

# update

#::---------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

#::--------------------------------------------------------

# Crate a canvas with the layout to draw a dotplot

# The layout sets all the elements and manage the changes

# made on the canvas

#::--------------------------------------------------------

super(TargetCount, self).\_\_init\_\_()

self.Title = "EDA: Variable Distribution"

self.main\_widget = QWidget(self)

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.fig = Figure()

self.ax = self.fig.add\_subplot(111)

self.axes = [self.ax]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding,

QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.dropdown1 = QComboBox()

self.featuresList = categorical.copy()

self.dropdown1.addItems(self.featuresList)

self.dropdown1.currentIndexChanged.connect(self.update)

self.label = QLabel("A plot:")

self.layout = QGridLayout(self.main\_widget)

self.layout.addWidget(QLabel("Select Features:"))

self.layout.addWidget(self.dropdown1)

self.layout.addWidget(self.canvas)

self.setCentralWidget(self.main\_widget)

self.resize(1200, 700)

self.show()

# def get\_bar\_dict(self, cat1, level\_list):

# count\_yes = []

# count\_no = []

# for level in level\_list:

# count\_no.append(len(df\_orig[(df\_orig[cat1] == level) & (df\_orig[target] == 0)]))

# count\_yes.append(len(df\_orig[(df\_orig[cat1] == level) & (df\_orig[target] == 1)]))

# return count\_no, count\_yes

def update(self):

#::--------------------------------------------------------

# This method executes each time a change is made on the canvas

# containing the elements of the graph

# The purpose of the method es to draw a dot graph using the

# score of happiness and the feature chosen the canvas

#::--------------------------------------------------------

colors = ["b", "r", "g", "y", "k", "c"]

self.ax.clear()

cat1 = self.dropdown1.currentText()

df\_pick = df\_orig[cat1]

level\_list = list(df\_pick.unique())

count\_yes = []

count\_no = []

for level in level\_list:

count\_no.append(len(df\_orig[(df\_orig[cat1] == level) & (df\_orig[target] == 0)]))

count\_yes.append(len(df\_orig[(df\_orig[cat1] == level) & (df\_orig[target] == 1)]))

all\_width = 0.7

width = all\_width / 2

onset = width / 2

x1, x2 = [x - onset for x in range(len(level\_list))], [x + onset for x in range(len(level\_list))]

self.ax.bar(x1, count\_no, align='edge', width=width, label='Default:No')

self.ax.bar(x2, count\_yes, align='edge', width=width, label='Default: Yes')

self.ax.set\_xticks(range(len(level\_list)))

self.ax.set\_xticklabels(level\_list)

self.ax.legend()

self.ax.set\_title(cat1)

self.ax.set\_xlabel(cat1)

self.ax.set\_ylabel("Count")

self.ax.grid(True)

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

del cat1

class CorrelationPlot(QMainWindow):

#;:-----------------------------------------------------------------------

# This class creates a canvas to draw a correlation plot

# It presents all the features plus the happiness score

# the methods for this class are:

# \_init\_

# initUi

# update

#::-----------------------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

#::--------------------------------------------------------

# Initialize the values of the class

#::--------------------------------------------------------

super(CorrelationPlot, self).\_\_init\_\_()

self.Title = 'Correlation Plot'

self.initUi()

def initUi(self):

#::--------------------------------------------------------------

# Creates the canvas and elements of the canvas

#::--------------------------------------------------------------

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.main\_widget = QWidget(self)

self.layout = QVBoxLayout(self.main\_widget)

self.groupBox1 = QGroupBox('Correlation Plot Features')

self.groupBox1Layout= QGridLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.feature0 = QCheckBox(features\_list[0], self)

self.feature1 = QCheckBox(features\_list[1], self)

self.feature2 = QCheckBox(features\_list[2], self)

self.feature3 = QCheckBox(features\_list[3], self)

self.feature4 = QCheckBox(features\_list[4], self)

self.feature5 = QCheckBox(features\_list[5], self)

self.feature6 = QCheckBox(features\_list[6], self)

self.feature7 = QCheckBox(features\_list[7], self)

self.feature8 = QCheckBox(features\_list[8], self)

self.feature9 = QCheckBox(features\_list[9], self)

self.feature10 = QCheckBox(features\_list[10], self)

self.feature11 = QCheckBox(features\_list[11], self)

self.feature12 = QCheckBox(features\_list[12], self)

self.feature13 = QCheckBox(features\_list[13], self)

self.feature14 = QCheckBox(features\_list[14], self)

self.feature15 = QCheckBox(features\_list[15], self)

self.feature16 = QCheckBox(features\_list[16], self)

self.feature17 = QCheckBox(features\_list[17], self)

self.feature18 = QCheckBox(features\_list[18], self)

self.feature19 = QCheckBox(features\_list[19], self)

self.feature20 = QCheckBox(features\_list[20], self)

self.feature21 = QCheckBox(features\_list[21], self)

self.feature22 = QCheckBox(features\_list[22], self)

self.feature23 = QCheckBox(features\_list[23], self)

self.feature0.setChecked(True)

self.feature1.setChecked(True)

self.feature2.setChecked(True)

self.feature3.setChecked(True)

self.feature4.setChecked(True)

self.feature5.setChecked(True)

self.feature6.setChecked(True)

self.feature7.setChecked(True)

self.feature8.setChecked(True)

self.feature9.setChecked(True)

self.feature10.setChecked(True)

self.feature11.setChecked(True)

self.feature12.setChecked(True)

self.feature13.setChecked(True)

self.feature14.setChecked(True)

self.feature15.setChecked(True)

self.feature16.setChecked(True)

self.feature17.setChecked(True)

self.feature18.setChecked(True)

self.feature19.setChecked(True)

self.feature20.setChecked(True)

self.feature21.setChecked(True)

self.feature22.setChecked(True)

self.feature23.setChecked(True)

self.btnExecute = QPushButton("Create Plot")

self.btnExecute.clicked.connect(self.update)

self.groupBox1Layout.addWidget(self.feature0, 0, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature1, 0, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature2, 0, 2, 1, 1)

self.groupBox1Layout.addWidget(self.feature3, 0, 3, 1, 1)

self.groupBox1Layout.addWidget(self.feature4, 0, 4, 1, 1)

self.groupBox1Layout.addWidget(self.feature5, 0, 5, 1, 1)

self.groupBox1Layout.addWidget(self.feature6, 0, 6, 1, 1)

self.groupBox1Layout.addWidget(self.feature7, 0, 7, 1, 1)

self.groupBox1Layout.addWidget(self.feature8, 1, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature9, 1, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature10, 1, 2, 1, 1)

self.groupBox1Layout.addWidget(self.feature11, 1, 3, 1, 1)

self.groupBox1Layout.addWidget(self.feature12, 1, 4, 1, 1)

self.groupBox1Layout.addWidget(self.feature13, 1, 5, 1, 1)

self.groupBox1Layout.addWidget(self.feature14, 1, 6, 1, 1)

self.groupBox1Layout.addWidget(self.feature15, 1, 7, 1, 1)

self.groupBox1Layout.addWidget(self.feature16, 2, 0, 1, 1)

self.groupBox1Layout.addWidget(self.feature17, 2, 1, 1, 1)

self.groupBox1Layout.addWidget(self.feature18, 2, 2, 1, 1)

self.groupBox1Layout.addWidget(self.feature19, 2, 3, 1, 1)

self.groupBox1Layout.addWidget(self.feature20, 2, 4, 1, 1)

self.groupBox1Layout.addWidget(self.feature21, 2, 5, 1, 1)

self.groupBox1Layout.addWidget(self.feature22, 2, 6, 1, 1)

self.groupBox1Layout.addWidget(self.feature23, 2, 7, 1, 1)

self.groupBox1Layout.addWidget(self.btnExecute,5,3,1,1)

self.fig = Figure()

self.ax1 = self.fig.add\_subplot(111)

self.axes=[self.ax1]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding,

QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.groupBox2 = QGroupBox('Correlation Plot')

self.groupBox2Layout= QVBoxLayout()

self.groupBox2.setLayout(self.groupBox2Layout)

self.groupBox2Layout.addWidget(self.canvas)

self.layout.addWidget(self.groupBox1)

self.layout.addWidget(self.groupBox2)

self.setCentralWidget(self.main\_widget)

self.resize(1500, 1400)

self.show()

self.update()

def update(self):

#::------------------------------------------------------------

# Populates the elements in the canvas using the values

# chosen as parameters for the correlation plot

#::------------------------------------------------------------

self.ax1.clear()

self.list\_corr\_features = pd.DataFrame([])

if self.feature0.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[0]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[0]]], axis=1)

if self.feature1.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[1]]], axis=1)

if self.feature2.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[2]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[2]]], axis=1)

if self.feature3.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[3]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[3]]], axis=1)

if self.feature4.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[4]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[4]]], axis=1)

if self.feature5.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[5]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[5]]], axis=1)

if self.feature6.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[6]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[6]]], axis=1)

if self.feature7.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[7]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[7]]], axis=1)

if self.feature8.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[8]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[8]]], axis=1)

if self.feature9.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[9]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[9]]], axis=1)

if self.feature10.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[10]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[10]]], axis=1)

if self.feature11.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[11]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[11]]], axis=1)

if self.feature12.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[12]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[12]]], axis=1)

if self.feature13.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[13]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[13]]], axis=1)

if self.feature14.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[14]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[14]]], axis=1)

if self.feature15.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[15]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[15]]], axis=1)

if self.feature16.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[16]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[16]]], axis=1)

if self.feature17.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[17]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[17]]], axis=1)

if self.feature18.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[18]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[18]]], axis=1)

if self.feature19.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[19]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[19]]], axis=1)

if self.feature20.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[20]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[20]]], axis=1)

if self.feature21.isChecked():

if len(self.list\_corr\_features) == 20:

self.list\_corr\_features = df[features\_list[1]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[21]]], axis=1)

if self.feature22.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[22]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[22]]], axis=1)

if self.feature23.isChecked():

if len(self.list\_corr\_features) == 0:

self.list\_corr\_features = df[features\_list[23]]

else:

self.list\_corr\_features = pd.concat([self.list\_corr\_features, df[features\_list[23]]], axis=1)

vsticks = ["dummy"]

vsticks1 = list(self.list\_corr\_features.columns)

vsticks1 = vsticks + vsticks1

res\_corr = self.list\_corr\_features.corr()

self.ax1.matshow(res\_corr, cmap= plt.cm.get\_cmap('Blues', 14))

self.ax1.set\_yticklabels(vsticks1)

self.ax1.set\_xticklabels(vsticks1,rotation = 90)

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

class TargetRelationship(QMainWindow):

#::---------------------------------------------------------

# This class crates a canvas with a plot to show the relation

# from each feature in the dataset with the target variables

# methods

# \_init\_

# update

#::---------------------------------------------------------

send\_fig = pyqtSignal(str)

def \_\_init\_\_(self):

super(TargetRelationship, self).\_\_init\_\_()

self.Title = "Boxplot of Categorical vs Numerical Variables"

self.main\_widget = QWidget(self)

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.fig = Figure()

self.ax1 = self.fig.add\_subplot(111)

self.axes = [self.ax1]

self.canvas = FigureCanvas(self.fig)

self.canvas.setSizePolicy(QSizePolicy.Expanding,

QSizePolicy.Expanding)

self.canvas.updateGeometry()

self.dropdown1 = QComboBox()

self.featuresList = numerical.copy()

self.dropdown1.addItems(self.featuresList)

self.dropdown1.currentIndexChanged.connect(self.update)

self.label = QLabel("A plot:")

self.dropdown2 = QComboBox()

self.featuresList = categorical.copy()

self.dropdown2.addItems(self.featuresList)

self.dropdown2.currentIndexChanged.connect(self.update)

self.label = QLabel("A plot:")

self.layout = QGridLayout(self.main\_widget)

self.layout.addWidget(QLabel("Select a Numerical Features:"), 0, 0, 1, 1)

self.layout.addWidget(self.dropdown1, 1, 0, 1, 1)

self.layout.addWidget(QLabel("Select a Categorical Features:"), 0, 1, 1, 1)

self.layout.addWidget(self.dropdown2, 1, 1, 1, 1)

self.groupBox1 = QGroupBox('Distribution')

self.groupBox1Layout = QVBoxLayout()

self.groupBox1.setLayout(self.groupBox1Layout)

self.groupBox1Layout.addWidget(self.canvas)

self.layout.addWidget(self.groupBox1, 3, 0, 5, 5)

self.setCentralWidget(self.main\_widget)

self.resize(1500, 1200)

self.show()

self.update()

def update(self):

self.ax1.clear()

cat1 = self.dropdown1.currentText()

cat2 = self.dropdown2.currentText()

df2 = df\_orig[[cat1, cat2]]

my\_pt = pd.pivot\_table(df2, index=df2.index, columns=cat2, values=cat1, aggfunc=np.sum)

my\_pt = pd.DataFrame(my\_pt.to\_records())

my\_pt = my\_pt.drop(columns=['index'])

my\_np = my\_pt.values

mask = ~np.isnan(my\_np)

box\_result = [d[m] for d, m in zip(my\_np.T, mask.T)]

class\_names\_x = my\_pt.columns.values.tolist()

self.ax1.boxplot(box\_result)

# X\_1 = df\_orig[cat2]

# y\_1 = df\_orig[cat1]

# for j, value2 in enumerate(X\_1.unique()):

# df\_orig.loc[df\_orig[cat2] == value2].plot(kind="box", x=cat2, y=cat1, ax=self.ax1,label=value2)

vtitle = cat2 + "vrs "+ cat1

self.ax1.set\_title(vtitle)

self.ax1.set\_xlabel(cat2)

self.ax1.set\_ylabel(cat1)

self.ax1.set\_xticklabels(class\_names\_x)

self.ax1.grid(True)

self.fig.tight\_layout()

self.fig.canvas.draw\_idle()

del cat1

class PlotCanvas(FigureCanvas):

#::----------------------------------------------------------

# creates a figure on the canvas

# later on this element will be used to draw a histogram graph

#::----------------------------------------------------------

def \_\_init\_\_(self, parent=None, width=5, height=4, dpi=100):

fig = Figure(figsize=(width, height), dpi=dpi)

FigureCanvas.\_\_init\_\_(self, fig)

self.setParent(parent)

FigureCanvas.setSizePolicy(self,

QSizePolicy.Expanding,

QSizePolicy.Expanding)

FigureCanvas.updateGeometry(self)

def plot(self):

self.ax = self.figure.add\_subplot(111)

class CanvasWindow(QMainWindow):

#::----------------------------------

# Creates a canvaas containing the plot for the initial analysis

#;;----------------------------------

def \_\_init\_\_(self, parent=None):

super(CanvasWindow, self).\_\_init\_\_(parent)

self.left = 200

self.top = 200

self.Title = 'Distribution'

self.width = 500

self.height = 500

self.initUI()

def initUI(self):

self.setWindowTitle(self.Title)

self.setStyleSheet(font\_size\_window)

self.setGeometry(self.left, self.top, self.width, self.height)

self.m = PlotCanvas(self, width=5, height=4)

self.m.move(0, 30)

class App(QMainWindow):

#::-------------------------------------------------------

# This class creates all the elements of the application

#::-------------------------------------------------------

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.left = 100

self.top = 100

self.Title = 'Vehicle Loan Prediction'

self.setWindowIcon(QIcon("logo.png"))

self.width = 1000

self.height = 500

self.initUI()

def initUI(self):

#::-------------------------------------------------

# Creates the manu and the items

#::-------------------------------------------------

self.setWindowTitle(self.Title)

self.setGeometry(self.left, self.top, self.width, self.height)

#::-----------------------------

# Create the menu bar

# and three items for the menu, File, EDA Analysis and ML Models

#::-----------------------------

mainMenu = self.menuBar()

mainMenu.setStyleSheet('background-color: lightblue')

fileMenu = mainMenu.addMenu('File')

EDAMenu = mainMenu.addMenu('EDA Analysis')

MLModelMenu = mainMenu.addMenu('ML Models')

#::--------------------------------------

# Exit application

# Creates the actions for the fileMenu item

#::--------------------------------------

exitButton = QAction(QIcon('enter.png'), 'Exit', self)

exitButton.setShortcut('Ctrl+Q')

exitButton.setStatusTip('Exit application')

exitButton.triggered.connect(self.close)

fileMenu.addAction(exitButton)

#::----------------------------------------

# EDA analysis

# Creates the actions for the EDA Analysis item

# Initial Assesment : Histogram about the level of happiness in 2017

# Happiness Final : Presents the correlation between the index of happiness and a feature from the datasets.

# Correlation Plot : Correlation plot using all the dims in the datasets

#::----------------------------------------

EDA1Button = QAction(QIcon('analysis.png'),'Variable Distribution', self)

EDA1Button.setStatusTip('Variable distribution against Loan Default')

EDA1Button.triggered.connect(self.EDA1)

EDAMenu.addAction(EDA1Button)

EDA3Button = QAction(QIcon('analysis.png'), 'Variable Counts', self)

EDA3Button.setStatusTip('Categorical Variable Counts')

EDA3Button.triggered.connect(self.EDA3)

EDAMenu.addAction(EDA3Button)

EDA4Button = QAction(QIcon('analysis.png'), 'Correlation Plot', self)

EDA4Button.setStatusTip('Features Correlation Plot')

EDA4Button.triggered.connect(self.EDA4)

EDAMenu.addAction(EDA4Button)

EDA5Button = QAction(QIcon('analysis.png'), 'Variable Relationship', self)

EDA5Button.setStatusTip('Boxplot of Loan Default')

EDA5Button.triggered.connect(self.EDA5)

EDAMenu.addAction(EDA5Button)

#::--------------------------------------------------

# ML Models for prediction

# There are two models

# Decision Tree

# Random Forest

#::--------------------------------------------------

# Decision Tree Model

#::--------------------------------------------------

MLModel1Button = QAction(QIcon(), 'Decision Tree Entropy', self)

MLModel1Button.setStatusTip('ML algorithm with Entropy ')

MLModel1Button.triggered.connect(self.MLDT)

#::------------------------------------------------------

# Random Forest Classifier

#::------------------------------------------------------

MLModel2Button = QAction(QIcon(), 'Random Forest Classifier', self)

MLModel2Button.setStatusTip('Random Forest Classifier ')

MLModel2Button.triggered.connect(self.MLRF)

MLModel3Button = QAction(QIcon(), 'Logistic Regression Classifier', self)

MLModel3Button.setStatusTip('Logistic Regression Classifier ')

MLModel3Button.triggered.connect(self.MLLR)

MLModel4Button = QAction(QIcon(), 'Gradient Boosting Classifier', self)

MLModel4Button.setStatusTip('Gradient Boosting Classifier ')

MLModel4Button.triggered.connect(self.MLGB)

MLModelMenu.addAction(MLModel1Button)

MLModelMenu.addAction(MLModel2Button)

MLModelMenu.addAction(MLModel3Button)

MLModelMenu.addAction(MLModel4Button)

self.dialogs = list()

def EDA1(self):

#::------------------------------------------------------

# Creates the histogram

# The X variable contains the happiness.score

# X was populated in the method data\_happiness()

# at the start of the application

#::------------------------------------------------------

dialog = TargetDistribution()

self.dialogs.append(dialog)

dialog.show()

def EDA3(self):

#::------------------------------------------------------

# Creates the histogram

# The X variable contains the happiness.score

# X was populated in the method data\_happiness()

# at the start of the application

#::------------------------------------------------------

dialog = TargetCount()

self.dialogs.append(dialog)

dialog.show()

def EDA4(self):

#::----------------------------------------------------------

# This function creates an instance of the CorrelationPlot class

#::----------------------------------------------------------

dialog = CorrelationPlot()

self.dialogs.append(dialog)

dialog.show()

def EDA5(self):

#::----------------------------------------------------------

# This function creates an instance of the CorrelationPlot class

#::----------------------------------------------------------

dialog = TargetRelationship()

self.dialogs.append(dialog)

dialog.show()

def MLDT(self):

#::-----------------------------------------------------------

# This function creates an instance of the DecisionTree class

# This class presents a dashboard for a Decision Tree Algorithm

# using the happiness dataset

#::-----------------------------------------------------------

dialog = DecisionTree()

self.dialogs.append(dialog)

dialog.show()

def MLRF(self):

# #::-------------------------------------------------------------

# # This function creates an instance of the Random Forest Classifier Algorithm

# # using the happiness dataset

# #::-------------------------------------------------------------

dialog = RandomForest()

self.dialogs.append(dialog)

dialog.show()

def MLLR(self):

# #::-------------------------------------------------------------

# # This function creates an instance of the Random Forest Classifier Algorithm

# # using the happiness dataset

# #::-------------------------------------------------------------

dialog = LogisticReg()

self.dialogs.append(dialog)

dialog.show()

def MLGB(self):

# #::-------------------------------------------------------------

# # This function creates an instance of the Random Forest Classifier Algorithm

# # using the happiness dataset

# #::-------------------------------------------------------------

dialog = GradientBoosting()

self.dialogs.append(dialog)

dialog.show()

def main():

#::-------------------------------------------------

# Initiates the application

#::-------------------------------------------------

app = QApplication(sys.argv)

app.setStyle('Fusion')

ex = App()

ex.show()

ex.showMaximized()

sys.exit(app.exec\_())

def data\_loan():

#::--------------------------------------------------

# Loads the dataset 2017.csv ( Index of happiness and esplanatory variables original dataset)

# Loads the dataset final\_happiness\_dataset (index of happiness

# and explanatory variables which are already preprocessed)

# Populates X,y that are used in the classes above

#::--------------------------------------------------

global loan

global X

global y

global X\_test

global y\_test

global features\_list

global class\_names

global target

global categorical

global numerical

global df\_orig

global df

df\_orig = pd.read\_csv(r'lt-vehicle-loan-default-prediction/train.csv')

# df = pd.read\_csv(r'lt-vehicle-loan-default-prediction/UItry.csv')

df=pd.read\_csv(r'lt-vehicle-loan-default-prediction/final\_test2.csv')

target = 'loan\_default'

X = df.drop([target], axis=1)

y= df[target].fillna(0)

columns = X.columns.tolist()

# indexes = [0,1,2,3,4,5,6,8,9,10,13,14,15,16,17,18,19,26,28,30,31,32,33,34,35]

# features\_list=['index', 'disbursed\_amount', 'asset\_cost', 'ltv', 'branch\_id', 'manufacturer\_id', 'Employment.Type', 'MobileNo\_Avl\_Flag', 'Aadhar\_flag', 'PAN\_flag', 'Passport\_flag', 'PERFORM\_CNS.SCORE', 'PRI.NO.OF.ACCTS', 'PRI.ACTIVE.ACCTS', 'PRI.OVERDUE.ACCTS', 'PRI.CURRENT.BALANCE', 'PRI.SANCTIONED.AMOUNT', 'SEC.DISBURSED.AMOUNT', 'SEC.INSTAL.AMT', 'DELINQUENT.ACCTS.IN.LAST.SIX.MONTHS', 'AVERAGE.ACCT.AGE', 'CREDIT.HISTORY.LENGTH', 'NO.OF\_INQUIRIES', 'Age', 'Disbursal\_months']

# features\_list = [columns[i+1] for i in indexes]

features\_list = columns

class\_names = ['No', 'Yes']

categorical = ['Employment.Type', 'PERFORM\_CNS.SCORE.DESCRIPTION', 'AVERAGE.ACCT.AGE' \

'Aadhar\_flag', 'PAN\_flag', 'VoterID\_flag', 'Driving\_flag', 'Passport\_flag','loan\_default']

numerical = ['disbursed\_amount', 'asset\_cost','PERFORM\_CNS.SCORE', 'PRI.NO.OF.ACCTS', 'PRI.ACTIVE.ACCTS', 'PRI.OVERDUE.ACCTS', \

'PRI.CURRENT.BALANCE', 'PRI.SANCTIONED.AMOUNT', 'PRI.DISBURSED.AMOUNT', 'SEC.NO.OF.ACCTS', 'SEC.ACTIVE.ACCTS', \

'SEC.OVERDUE.ACCTS', 'SEC.CURRENT.BALANCE','SEC.SANCTIONED.AMOUNT', 'SEC.DISBURSED.AMOUNT', 'PRIMARY.INSTAL.AMT', \

'SEC.INSTAL.AMT', 'NEW.ACCTS.IN.LAST.SIX.MONTHS', 'DELINQUENT.ACCTS.IN.LAST.SIX.MONTHS','NO.OF\_INQUIRIES'\

]

if \_\_name\_\_ == '\_\_main\_\_':

#::------------------------------------

# First reads the data then calls for the application

#::------------------------------------

data\_loan()

main()