# Import library

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

from sklearn.metrics import accuracy\_score

from sklearn.linear\_model import SGDClassifier,RidgeClassifier

from sklearn.metrics import (precision\_score, recall\_score,f1\_score)

from sklearn.metrics import average\_precision\_score

from sklearn.metrics import plot\_confusion\_matrix

from sklearn.preprocessing import LabelEncoder, StandardScaler, MinMaxScaler

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

# Metrics

from sklearn.metrics import confusion\_matrix, accuracy\_score, classification\_report

from sklearn.metrics import roc\_auc\_score, roc\_curve

# Models

from sklearn.linear\_model import LogisticRegression

from sklearn.naive\_bayes import GaussianNB

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC

from sklearn import tree

# for evaluation

from sklearn import metrics

from sklearn.metrics import (confusion\_matrix, precision\_recall\_curve, auc, roc\_curve, recall\_score,

                             classification\_report, f1\_score, average\_precision\_score, precision\_recall\_fscore\_support)

from google.colab import files

uploaded = files.upload() #this is used because colab, its take data from PC and upload in colab

Importing a dataset is simple with Pandas through functions dedicated to reading the data. If our dataset is a .csv file, we can just use

df = pd.read\_csv here I used df = final word

Final = pd.read\_csv('ph (1).csv') # dataset import

Final.head()

df stands for dataframe, which is Pandas’s object similar to an Excel sheet. This nomenclature is often used in the field. The read\_csv function takes as input the path of the file we want to read. There are many other arguments that we can specify.

Pandas are .head() and .tail(). These two allow us to view an arbitrary number of rows (by default 5) from the beginning or end of the dataset. Very useful for accessing a small part of the dataframe quickly.

# Data visualization:

import matplotlib.pyplot as plt

import numpy as np

boxplot = Final.boxplot(figsize=(60,10), rot=40)

Box plot is used to view data in box in graph

# Drop Domain and Status columns

# drop column because strings in it

column = Final.drop(['Domain'], axis = 1).copy() # remove domain and status because string values and save the value in Column name

column = Final.drop(['status'], axis = 1).copy()

column.isnull().sum() # isnull().sum() returns the number of missing values in the dataset.

column.head(3)

boxplot = column.boxplot(figsize=(10,7), rot=30,color='yellow')

# **Data pre processing**

X = Final.copy() #copy our dataset into X

Y = Final["status"] #Y only show Status column

X.drop(["status" , "Domain"] , axis=1 , inplace=True) # drop status and Domain because string value

Cols = X.columns;

Y = Y == "legitimate" # remember Y = legitimate column

Scaler = StandardScaler(copy=True , with\_mean=True , with\_std=True) #Standardize features by removing the mean and scaling to unit variance. where u is the mean of the training samples or zero if with\_mean=False, and s is the standard deviation of the training samples or one if with\_std=False.

X = Scaler.fit\_transform(X) # Fit to data into X, then transform it.

# **Divide the data into test and train**

x\_train , x\_test , y\_train , y\_test = train\_test\_split(X , Y , test\_size=0.20 , random\_state=10 , shuffle=True) # 80% and 20%

#By default, the function shuffles the data (with shuffle=True ) before splitting. The random state hyperparameter in the train\_test\_split() function controls the shuffling process.

**Ridget Classifier**

[ ]

# Ridge Classifier

#This classifier first converts the target values into {-1, 1} and then treats the problem as a regression task (multi-output regression in the multiclass case).

rc = RidgeClassifier() #RC is valuse which call ridge

rc\_fit = rc.fit(x\_train , y\_train) #Fit Ridge classifier model.

y\_pred\_rc = rc\_fit.predict(x\_test) # predic the value

rc\_accuracy = accuracy\_score(y\_test, y\_pred\_rc) # for accuracy

#Print result

print ("Accuracy: " + str(accuracy\_score(y\_pred\_rc, y\_test)))

print ("Precision: " + str(precision\_score(y\_pred\_rc, y\_test)))

print ("Recall: " + str(recall\_score(y\_pred\_rc, y\_test)))

print ("F1: " + str(f1\_score(y\_pred\_rc, y\_test)))

print(classification\_report(y\_test, y\_pred\_rc))

**XGB**

from xgboost import XGBClassifier

xgb = XGBClassifier(learning\_rate=0.8,max\_depth=9)  #The maximum depth per tree. A deeper tree might increase the performance, but also the complexity and chances to overfit. The value must be an integer greater than 0. Default is 6 i used 9

xgb.fit(x\_train, y\_train) #fit the model

#predicting

y\_test\_xgb = xgb.predict(x\_test) # predict the model

y\_train\_xgb = xgb.predict(x\_train)

acc\_train\_xgb = accuracy\_score(y\_train,y\_train\_xgb)

acc\_test\_xgb = accuracy\_score(y\_test,y\_test\_xgb)

print ("Accuracy: " + str(accuracy\_score(y\_test\_xgb, y\_test)))

print ("Precision: " + str(precision\_score(y\_test\_xgb, y\_test)))

print ("Recall: " + str(recall\_score(y\_test\_xgb, y\_test)))

print ("F1: " + str(f1\_score(y\_test\_xgb, y\_test)))

**Decision Tree**

[ ]

from sklearn.tree import·DecisionTreeClassifier

dt·=·DecisionTreeClassifier(max\_depth·=·5)·#·max\_depthint,·default=None·.·The·maximum·depth·of·the·tree.·If·None,·then·nodes·are·expanded·until·all·leaves·are·pure·or·until·all·leaves·contain·less·than·min\_samples\_split·samples.

dt.fit(x\_train,·y\_train)·#Build·a·decision·tree·classifier·from·the·training·set·(X,·y)

y\_test\_dt·=·dt.predict(x\_test)·#predict·

y\_train\_dt·=·dt.predict(x\_train)

acc\_train\_dt·=·accuracy\_score(y\_train,y\_train\_dt)

acc\_test\_dt·=·accuracy\_score(y\_test,y\_test\_dt)

print·("Accuracy:·"·+·str(accuracy\_score(y\_test\_dt,·y\_test)))

print·("Precision:·"·+·str(precision\_score(y\_test\_dt,·y\_test)))

print·("Recall:·"·+·str(recall\_score(y\_test\_dt,·y\_test)))

#PRint·result

print·("F1:·"·+·str(f1\_score(y\_test\_dt,·y\_test)))

**MLP**

[ ]

#·MLP

from·sklearn.neural\_network·import·MLPClassifier

from·sklearn.datasets·import·make\_classification

mlp·=·MLPClassifier(alpha=0.0000000001,··hidden\_layer\_sizes=(100·,·100·,·100·,·100)·,·max\_iter=100·,·random\_state=50)·#hidden\_layer\_sizes·:·This·parameter·allows·us·to·set·the·number·of·layers·and·the·number·of·nodes·we·wish·to·have·in·the·Neural·Network·Classifier

mlp.fit(x\_train,·y\_train)·#fit·the·model

y\_test\_mlp·=·mlp.predict(x\_test)

y\_train\_mlp·=·mlp.predict(x\_train)

print·("Accuracy:·"·+·str(accuracy\_score(y\_test\_mlp,·y\_test)))

print·("Precision:·"·+·str(precision\_score(y\_test\_mlp,·y\_test)))

print·("Recall:·"·+·str(recall\_score(y\_test\_mlp,·y\_test)))

print·("F1:·"·+·str(f1\_score(y\_test\_mlp,·y\_test)))

**Random forest**

[ ]

from sklearn.ensemble import RandomForestClassifier

forest = RandomForestClassifier(max\_depth=5) #max\_depth: The number of splits that each decision tree is allowed to make. If the number of splits is too low, the model underfits the data and if it is too high the model overfits.

forest.fit(x\_train, y\_train)

#predicting

y\_test\_forest = forest.predict(x\_test)

y\_train\_forest = forest.predict(x\_train)

print ("Accuracy: " + str(accuracy\_score(y\_test\_forest, y\_test)))

print ("Precision: " + str(precision\_score(y\_test\_forest, y\_test)))

print ("Recall: " + str(recall\_score(y\_test\_forest, y\_test)))

print ("F1: " + str(f1\_score(y\_test\_forest, y\_test)))

SVM

from sklearn.svm import SVC

svm = SVC(kernel='linear', C=1.0, random\_state=15) #Random state ensures that the splits that you generate are reproducible.

svm.fit(x\_train, y\_train) #fit the model

y\_test\_svm = svm.predict(x\_test) #predic the model

#result

print ("Accuracy: " + str(accuracy\_score(y\_test\_svm, y\_test)))

print ("Precision: " + str(precision\_score(y\_test\_svm, y\_test)))

print ("Recall: " + str(recall\_score(y\_test\_svm, y\_test)))

print ("F1: " + str(f1\_score(y\_test\_svm, y\_test)))

**ANN**

[ ]

input\_shape = [x\_train.shape[1]]

print("Input shape is ", input\_shape) #input shape for learning



Input shape is [59]

[ ]

from tensorflow import keras #Keras layers and models are fully compatible with pure-TensorFlow tensors, and as a result, Keras makes a great model definition add-on

from tensorflow.keras import layers

#A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor

model = keras.Sequential([

    layers.BatchNormalization(input\_shape=input\_shape), #Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1

    layers.Dense(512, activation='relu'), #ReLU is half rectified, not open all

    layers.BatchNormalization(),

    layers.Dropout(0.3), #The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting.

    layers.Dense(512, activation='relu'), #a dense layer is a layer that is deeply connected with its preceding layer which means the neurons of the layer are connected to every neuron of its preceding layer.

    layers.BatchNormalization(),

    layers.Dropout(0.3),

    layers.Dense(1, activation='sigmoid'), # It can take any value from –infinity to +infinity yet its output is always between 0 and 1.

])

[ ]

····optimizer='adam',·#Optimizer·that·implements·the·Adam·algorithm.·Adam·optimization·is·a·stochastic·gradient·descent·method·that·is·based·on·adaptive·estimation·of·first-order·and·second-order·moments.

····loss='binary\_crossentropy',·#Used·as·a·loss·function·for·binary·classification·model.·The·binary\_crossentropy·function·computes·the·cross-entropy·loss·between·true·labels·and·predicted·labels.

····metrics=['binary\_accuracy'],

·#Compile·defines·the·loss·function,·the·optimizer·and·the·metrics.·That's·all.·It·has·nothing·to·do·with·the·weights·and·you·can·compile·a·model·as·many·times·as·you·want·without·causing·any·problem·to·pretrained·weights

model.compile(

)



[ ]

early\_stopping = keras.callbacks.EarlyStopping( #EarlyStopping is a callback used while training neural networks, which provides us the advantage of using a large number of training epochs and stopping the training once the model's performance stops improving on the validation Dataset.

    patience=20,

    min\_delta=0.01,

    restore\_best\_weights=True,

)

history = model.fit(

    x\_train, y\_train,

    batch\_size=512,

    epochs=200, #i give 512 for better accuracy

    callbacks=[early\_stopping],

)

history\_df = pd.DataFrame(history.history)

history\_df.loc[0:, ['binary\_accuracy']].plot()

   #show in graph history\_df is define to store result

Ann = print(( history\_df['binary\_accuracy'].max())) # select max accuracy from 200 iteration