**Phishing Website Detection**

**1. Objective:**

A phishing website is a common social engineering method that mimics trustful uniform resource locators (URLs) and webpages. The objective of this project is to train machine learning models on the dataset to predict phishing websites. Both phishing and benign URLs of websites are gathered to form a dataset and from them required URL and website content-based features are extracted. The performance level of each model is measures and compared.

**2. Loading Data:**

The dataset that is given is loaded.

In [39]:

*#importing basic packages*

**import** pandas **as** pd

**import** numpy **as** np

**import** seaborn **as** sns

**import** matplotlib.pyplot **as** plt

In [40]:

*#Loading the data*

data0 **=** pd**.**read\_csv('dataset\_website.csv')

data0**.**head()

Out[40]:

|  | **index** | **having\_IPhaving\_IP\_Address** | **URLURL\_Length** | **Shortining\_Service** | **having\_At\_Symbol** | **double\_slash\_redirecting** | **Prefix\_Suffix** | **having\_Sub\_Domain** | **SSLfinal\_State** | **Domain\_registeration\_length** | **...** | **popUpWidnow** | **Iframe** | **age\_of\_domain** | **DNSRecord** | **web\_traffic** | **Page\_Rank** | **Google\_Index** | **Links\_pointing\_to\_page** | **Statistical\_report** | **Result** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1 | -1 | 1 | 1 | 1 | -1 | -1 | -1 | -1 | -1 | ... | 1 | 1 | -1 | -1 | -1 | -1 | 1 | 1 | -1 | -1 |
| **1** | 2 | 1 | 1 | 1 | 1 | 1 | -1 | 0 | 1 | -1 | ... | 1 | 1 | -1 | -1 | 0 | -1 | 1 | 1 | 1 | -1 |
| **2** | 3 | 1 | 0 | 1 | 1 | 1 | -1 | -1 | -1 | -1 | ... | 1 | 1 | 1 | -1 | 1 | -1 | 1 | 0 | -1 | -1 |
| **3** | 4 | 1 | 0 | 1 | 1 | 1 | -1 | -1 | -1 | 1 | ... | 1 | 1 | -1 | -1 | 1 | -1 | 1 | -1 | 1 | -1 |
| **4** | 5 | 1 | 0 | -1 | 1 | 1 | -1 | 1 | 1 | -1 | ... | -1 | 1 | -1 | -1 | 0 | -1 | 1 | 1 | 1 | 1 |

5 rows × 32 columns

**3. Familiarizing with Data**

In this step, few dataframe methods are used to look into the data and its features.

In [41]:

*#Checking the shape of the dataset*

data0**.**shape

Out[41]:

(11055, 32)

In [42]:

*#Listing the features of the dataset*

data0**.**columns

Out[42]:

Index(['index', 'having\_IPhaving\_IP\_Address', 'URLURL\_Length',

'Shortining\_Service', 'having\_At\_Symbol', 'double\_slash\_redirecting',

'Prefix\_Suffix', 'having\_Sub\_Domain', 'SSLfinal\_State',

'Domain\_registeration\_length', 'Favicon', 'port', 'HTTPS\_token',

'Request\_URL', 'URL\_of\_Anchor', 'Links\_in\_tags', 'SFH',

'Submitting\_to\_email', 'Abnormal\_URL', 'Redirect', 'on\_mouseover',

'RightClick', 'popUpWidnow', 'Iframe', 'age\_of\_domain', 'DNSRecord',

'web\_traffic', 'Page\_Rank', 'Google\_Index', 'Links\_pointing\_to\_page',

'Statistical\_report', 'Result'],

dtype='object')

In [43]:

*#Information about the dataset*

data0**.**info()

RangeIndex: 11055 entries, 0 to 11054

Data columns (total 32 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 index 11055 non-null int64

1 having\_IPhaving\_IP\_Address 11055 non-null int64

2 URLURL\_Length 11055 non-null int64

3 Shortining\_Service 11055 non-null int64

4 having\_At\_Symbol 11055 non-null int64

5 double\_slash\_redirecting 11055 non-null int64

6 Prefix\_Suffix 11055 non-null int64

7 having\_Sub\_Domain 11055 non-null int64

8 SSLfinal\_State 11055 non-null int64

9 Domain\_registeration\_length 11055 non-null int64

10 Favicon 11055 non-null int64

11 port 11055 non-null int64

12 HTTPS\_token 11055 non-null int64

13 Request\_URL 11055 non-null int64

14 URL\_of\_Anchor 11055 non-null int64

15 Links\_in\_tags 11055 non-null int64

16 SFH 11055 non-null int64

17 Submitting\_to\_email 11055 non-null int64

18 Abnormal\_URL 11055 non-null int64

19 Redirect 11055 non-null int64

20 on\_mouseover 11055 non-null int64

21 RightClick 11055 non-null int64

22 popUpWidnow 11055 non-null int64

23 Iframe 11055 non-null int64

24 age\_of\_domain 11055 non-null int64

25 DNSRecord 11055 non-null int64

26 web\_traffic 11055 non-null int64

27 Page\_Rank 11055 non-null int64

28 Google\_Index 11055 non-null int64

29 Links\_pointing\_to\_page 11055 non-null int64

30 Statistical\_report 11055 non-null int64

31 Result 11055 non-null int64

dtypes: int64(32)

memory usage: 2.7 MB

**4. Visualizing the data**

Few plots and graphs are displayed to find how the data is distributed and the how features are related to each other.

In [44]:

*#Plotting the data distribution*

data0**.**hist(bins **=** 50,figsize **=** (15,15))

plt**.**show()

In [45]:

*#Correlation heatmap*

plt**.**figure(figsize**=**(15,13))

sns**.**heatmap(data0**.**corr())

plt**.**show()

**5. Data Preprocessing & EDA**

Here, we clean the data by applying data preprocesssing techniques and transform the data to use it in the models.

In [46]:

data0**.**describe()

Out[46]:

|  | **index** | **having\_IPhaving\_IP\_Address** | **URLURL\_Length** | **Shortining\_Service** | **having\_At\_Symbol** | **double\_slash\_redirecting** | **Prefix\_Suffix** | **having\_Sub\_Domain** | **SSLfinal\_State** | **Domain\_registeration\_length** | **...** | **popUpWidnow** | **Iframe** | **age\_of\_domain** | **DNSRecord** | **web\_traffic** | **Page\_Rank** | **Google\_Index** | **Links\_pointing\_to\_page** | **Statistical\_report** | **Result** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | ... | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 | 11055.000000 |
| **mean** | 5528.000000 | 0.313795 | -0.633198 | 0.738761 | 0.700588 | 0.741474 | -0.734962 | 0.063953 | 0.250927 | -0.336771 | ... | 0.613388 | 0.816915 | 0.061239 | 0.377114 | 0.287291 | -0.483673 | 0.721574 | 0.344007 | 0.719584 | 0.113885 |
| **std** | 3191.447947 | 0.949534 | 0.766095 | 0.673998 | 0.713598 | 0.671011 | 0.678139 | 0.817518 | 0.911892 | 0.941629 | ... | 0.789818 | 0.576784 | 0.998168 | 0.926209 | 0.827733 | 0.875289 | 0.692369 | 0.569944 | 0.694437 | 0.993539 |
| **min** | 1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | ... | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 |
| **25%** | 2764.500000 | -1.000000 | -1.000000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | -1.000000 | -1.000000 | -1.000000 | ... | 1.000000 | 1.000000 | -1.000000 | -1.000000 | 0.000000 | -1.000000 | 1.000000 | 0.000000 | 1.000000 | -1.000000 |
| **50%** | 5528.000000 | 1.000000 | -1.000000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | 0.000000 | 1.000000 | -1.000000 | ... | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | 1.000000 | 0.000000 | 1.000000 | 1.000000 |
| **75%** | 8291.500000 | 1.000000 | -1.000000 | 1.000000 | 1.000000 | 1.000000 | -1.000000 | 1.000000 | 1.000000 | 1.000000 | ... | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |
| **max** | 11055.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | ... | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 |

8 rows × 32 columns

From data distribution graph and correlation matrix, we can conclude that the following features do not have much impact on the result:

* having\_Sub\_Domain
* Domain\_registeration\_length
* Favicon
* HTTPS\_token
* Request\_URL
* URL\_of\_Anchor
* Links\_in\_tags
* Submitting\_to\_email
* Redirect
* on\_mouseover
* RightClick
* age\_of\_domain
* web\_traffic
* Page\_Rank
* Google\_Index
* Links\_pointing\_to\_page

In [47]:

*#Removing the features which do not have much impact on Result*

data**=**data0**.**iloc[:,[0,1,2,3,4,5,6,8,11,16,18,22,23,25,30]]

data**.**head()

Out[47]:

|  | **index** | **having\_IPhaving\_IP\_Address** | **URLURL\_Length** | **Shortining\_Service** | **having\_At\_Symbol** | **double\_slash\_redirecting** | **Prefix\_Suffix** | **SSLfinal\_State** | **port** | **SFH** | **Abnormal\_URL** | **popUpWidnow** | **Iframe** | **DNSRecord** | **Statistical\_report** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1 | -1 | 1 | 1 | 1 | -1 | -1 | -1 | 1 | -1 | -1 | 1 | 1 | -1 | -1 |
| **1** | 2 | 1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | -1 | 1 | 1 | 1 | -1 | 1 |
| **2** | 3 | 1 | 0 | 1 | 1 | 1 | -1 | -1 | 1 | -1 | -1 | 1 | 1 | -1 | -1 |
| **3** | 4 | 1 | 0 | 1 | 1 | 1 | -1 | -1 | 1 | -1 | 1 | 1 | 1 | -1 | 1 |
| **4** | 5 | 1 | 0 | -1 | 1 | 1 | -1 | 1 | 1 | -1 | 1 | -1 | 1 | -1 | 1 |

In [48]:

data**.**info()

RangeIndex: 11055 entries, 0 to 11054

Data columns (total 15 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 index 11055 non-null int64

1 having\_IPhaving\_IP\_Address 11055 non-null int64

2 URLURL\_Length 11055 non-null int64

3 Shortining\_Service 11055 non-null int64

4 having\_At\_Symbol 11055 non-null int64

5 double\_slash\_redirecting 11055 non-null int64

6 Prefix\_Suffix 11055 non-null int64

7 SSLfinal\_State 11055 non-null int64

8 port 11055 non-null int64

9 SFH 11055 non-null int64

10 Abnormal\_URL 11055 non-null int64

11 popUpWidnow 11055 non-null int64

12 Iframe 11055 non-null int64

13 DNSRecord 11055 non-null int64

14 Statistical\_report 11055 non-null int64

dtypes: int64(15)

memory usage: 1.3 MB

This leaves us with 14 features & a target column.

In [49]:

*#checking the data for null or missing values*

data**.**isnull()**.**sum()

Out[49]:

index 0

having\_IPhaving\_IP\_Address 0

URLURL\_Length 0

Shortining\_Service 0

having\_At\_Symbol 0

double\_slash\_redirecting 0

Prefix\_Suffix 0

SSLfinal\_State 0

port 0

SFH 0

Abnormal\_URL 0

popUpWidnow 0

Iframe 0

DNSRecord 0

Statistical\_report 0

dtype: int64

To even out the distribution while splitting the data into training & testing sets, we need to shuffle it. This evades the case of overfitting while model training.

In [50]:

*# shuffling the rows in the dataset so that when splitting the train and test set are equally distributed*

data **=** data**.**sample(frac**=**1)**.**reset\_index(drop**=True**)

data**.**head()

Out[50]:

|  | **index** | **having\_IPhaving\_IP\_Address** | **URLURL\_Length** | **Shortining\_Service** | **having\_At\_Symbol** | **double\_slash\_redirecting** | **Prefix\_Suffix** | **SSLfinal\_State** | **port** | **SFH** | **Abnormal\_URL** | **popUpWidnow** | **Iframe** | **DNSRecord** | **Statistical\_report** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 402 | 1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | -1 | 1 | -1 | 1 | -1 | 1 |
| **1** | 1403 | 1 | -1 | 1 | 1 | 1 | -1 | 1 | 1 | -1 | -1 | 1 | 1 | -1 | -1 |
| **2** | 10081 | 1 | -1 | 1 | -1 | 1 | 1 | 1 | 1 | -1 | -1 | 1 | 1 | 1 | 1 |
| **3** | 2069 | -1 | -1 | -1 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| **4** | 621 | 1 | -1 | 1 | -1 | 1 | 1 | 1 | 1 | -1 | 1 | 1 | 1 | 1 | 1 |

From the above execution, it is clear that the data doesnot have any missing values.

By this, the data is throughly preprocessed & is ready for training.

**6. Splitting the Data**

In [51]:

*# Sepratating & assigning features and target columns to X & y*

X**=**data**.**iloc[:,:14]

y**=**data**.**iloc[:,14]

X**.**shape, y**.**shape

Out[51]:

((11055, 14), (11055,))

In [52]:

*# Splitting the dataset into train and test sets: 80-20 split*

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

X\_train, X\_test, y\_train, y\_test **=** train\_test\_split(X, y, test\_size **=** 0.2, random\_state **=** 12)

X\_train**.**shape, X\_test**.**shape

Out[52]:

((8844, 14), (2211, 14))

**7. Machine Learning Models & Training**

From the dataset above, it is clear that this is a supervised machine learning task. There are two major types of supervised machine learning problems, called classification and regression.

This data set comes under classification problem, as the input URL is classified as phishing (1) or legitimate (0). The supervised machine learning models (classification) considered to train the dataset in this notebook are:

* Decision Tree
* Random Forest
* XGBoost
* Support Vector Machines

In [53]:

*#importing packages*

**from** sklearn.metrics **import** accuracy\_score

In [54]:

*# Creating holders to store the model performance results*

ML\_Model **=** []

acc\_train **=** []

acc\_test **=** []

*#function to call for storing the results*

**def** storeResults(model, a,b):

ML\_Model**.**append(model)

acc\_train**.**append(round(a, 3))

acc\_test**.**append(round(b, 3))

**7.1. Decision Tree Classifier**

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision. Learning a decision tree means learning the sequence of if/else questions that gets us to the true answer most quickly.

In the machine learning setting, these questions are called tests (not to be confused with the test set, which is the data we use to test to see how generalizable our model is). To build a tree, the algorithm searches over all possible tests and finds the one that is most informative about the target variable.

In [55]:

*# Decision Tree model*

**from** sklearn.tree **import** DecisionTreeClassifier

*# instantiate the model*

tree **=** DecisionTreeClassifier(max\_depth **=** 5)

*# fit the model*

tree**.**fit(X\_train, y\_train)

Out[55]:

DecisionTreeClassifier(max\_depth=5)

In [56]:

*#predicting the target value from the model for the samples*

y\_test\_tree **=** tree**.**predict(X\_test)

y\_train\_tree **=** tree**.**predict(X\_train)

**Performance Evaluation:**

In [57]:

*#computing the accuracy of the model performance*

acc\_train\_tree **=** accuracy\_score(y\_train,y\_train\_tree)

acc\_test\_tree **=** accuracy\_score(y\_test,y\_test\_tree)

print("Decision Tree: Accuracy on training Data: {:.3f}"**.**format(acc\_train\_tree))

print("Decision Tree: Accuracy on test Data: {:.3f}"**.**format(acc\_test\_tree))

Decision Tree: Accuracy on training Data: 0.929

Decision Tree: Accuracy on test Data: 0.923

In [58]:

*#checking the feature improtance in the model*

plt**.**figure(figsize**=**(9,7))

n\_features **=** X\_train**.**shape[1]

plt**.**barh(range(n\_features), tree**.**feature\_importances\_, align**=**'center')

plt**.**yticks(np**.**arange(n\_features), X\_train**.**columns)

plt**.**xlabel("Feature importance")

plt**.**ylabel("Feature")

plt**.**show()

**Storing the results:**

In [59]:

*#storing the results. The below mentioned order of parameter passing is important.*

storeResults('Decision Tree', acc\_train\_tree, acc\_test\_tree)

**7.2. Random Forest Classifier**

Random forests for regression and classification are currently among the most widely used machine learning methods.A random forest is essentially a collection of decision trees, where each tree is slightly different from the others. The idea behind random forests is that each tree might do a relatively good job of predicting, but will likely overfit on part of the data.

If we build many trees, all of which work well and overfit in different ways, we can reduce the amount of overfitting by averaging their results. To build a random forest model, you need to decide on the number of trees to build (the n\_estimators parameter of RandomForestRegressor or RandomForestClassifier). They are very powerful, often work well without heavy tuning of the parameters, and don’t require scaling of the data.

In [60]:

*# Random Forest model*

**from** sklearn.ensemble **import** RandomForestClassifier

*# instantiate the model*

forest **=** RandomForestClassifier(max\_depth**=**5)

*# fit the model*

forest**.**fit(X\_train, y\_train)

Out[60]:

RandomForestClassifier(max\_depth=5)

In [61]:

*#predicting the target value from the model for the samples*

y\_test\_forest **=** forest**.**predict(X\_test)

y\_train\_forest **=** forest**.**predict(X\_train)

**Performance Evaluation:**

In [62]:

*#computing the accuracy of the model performance*

acc\_train\_forest **=** accuracy\_score(y\_train,y\_train\_forest)

acc\_test\_forest **=** accuracy\_score(y\_test,y\_test\_forest)

print("Random forest: Accuracy on training Data: {:.3f}"**.**format(acc\_train\_forest))

print("Random forest: Accuracy on test Data: {:.3f}"**.**format(acc\_test\_forest))

Random forest: Accuracy on training Data: 0.916

Random forest: Accuracy on test Data: 0.908

In [63]:

*#checking the feature improtance in the model*

plt**.**figure(figsize**=**(9,7))

n\_features **=** X\_train**.**shape[1]

plt**.**barh(range(n\_features), forest**.**feature\_importances\_, align**=**'center')

plt**.**yticks(np**.**arange(n\_features), X\_train**.**columns)

plt**.**xlabel("Feature importance")

plt**.**ylabel("Feature")

plt**.**show()

**Storing the results:**

In [64]:

*#storing the results. The below mentioned order of parameter passing is important.*

storeResults('Random Forest', acc\_train\_forest, acc\_test\_forest)

**7.4. XGBoost Classifier**

XGBoost is one of the most popular machine learning algorithms these days. XGBoost stands for eXtreme Gradient Boosting. Regardless of the type of prediction task at hand; regression or classification. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance.

In [65]:

*#XGBoost Classification model*

**from** xgboost **import** XGBClassifier

*# instantiate the model*

xgb **=** XGBClassifier(learning\_rate**=**0.4,max\_depth**=**7)

*#fit the model*

xgb**.**fit(X\_train, y\_train)

Out[65]:

XGBClassifier(learning\_rate=0.4, max\_depth=7)

In [66]:

*#predicting the target value from the model for the samples*

y\_test\_xgb **=** xgb**.**predict(X\_test)

y\_train\_xgb **=** xgb**.**predict(X\_train)

**Performance Evaluation:**

In [67]:

*#computing the accuracy of the model performance*

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

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

print("XGBoost: Accuracy on training Data: {:.3f}"**.**format(acc\_train\_xgb))

print("XGBoost : Accuracy on test Data: {:.3f}"**.**format(acc\_test\_xgb))

XGBoost: Accuracy on training Data: 0.965

XGBoost : Accuracy on test Data: 0.923

**Storing the results:**

In [68]:

*#storing the results. The below mentioned order of parameter passing is important.*

storeResults('XGBoost', acc\_train\_xgb, acc\_test\_xgb)

**7.6. Support Vector Machines**

In machine learning, support-vector machines (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier.

In [69]:

*#Support vector machine model*

**from** sklearn.svm **import** SVC

*# instantiate the model*

svm **=** SVC(kernel**=**'linear', C**=**1.0, random\_state**=**12)

*#fit the model*

svm**.**fit(X\_train, y\_train)

Out[69]:

SVC(kernel='linear', random\_state=12)

In [70]:

*#predicting the target value from the model for the samples*

y\_test\_svm **=** svm**.**predict(X\_test)

y\_train\_svm **=** svm**.**predict(X\_train)

**Performance Evaluation:**

In [71]:

*#computing the accuracy of the model performance*

acc\_train\_svm **=** accuracy\_score(y\_train,y\_train\_svm)

acc\_test\_svm **=** accuracy\_score(y\_test,y\_test\_svm)

print("SVM: Accuracy on training Data: {:.3f}"**.**format(acc\_train\_svm))

print("SVM : Accuracy on test Data: {:.3f}"**.**format(acc\_test\_svm))

SVM: Accuracy on training Data: 0.899

SVM : Accuracy on test Data: 0.892

**Storing the results:**

In [72]:

*#storing the results. The below mentioned order of parameter passing is important.*

storeResults('SVM', acc\_train\_svm, acc\_test\_svm)

**8. Comparision of Models**

To compare the models performance, a dataframe is created. The columns of this dataframe are the lists created to store the results of the model.

In [73]:

*#creating dataframe*

results **=** pd**.**DataFrame({ 'ML Model': ML\_Model,

'Train Accuracy': acc\_train,

'Test Accuracy': acc\_test})

results

Out[73]:

|  | **ML Model** | **Train Accuracy** | **Test Accuracy** |
| --- | --- | --- | --- |
| **0** | Decision Tree | 0.929 | 0.923 |
| **1** | Random Forest | 0.916 | 0.908 |
| **2** | XGBoost | 0.965 | 0.923 |
| **3** | SVM | 0.899 | 0.892 |

In [74]:

*#Sorting the datafram on accuracy*

results**.**sort\_values(by**=**['Test Accuracy', 'Train Accuracy'], ascending**=False**)

Out[74]:

|  | **ML Model** | **Train Accuracy** | **Test Accuracy** |
| --- | --- | --- | --- |
| **2** | XGBoost | 0.965 | 0.923 |
| **0** | Decision Tree | 0.929 | 0.923 |
| **1** | Random Forest | 0.916 | 0.908 |
| **3** | SVM | 0.899 | 0.892 |

For the above comparision, it is clear that the **XGBoost Classifier** works well with this dataset.

So, saving the model for future use.

In [75]:

*# save XGBoost model to file*

**import** pickle

pickle**.**dump(xgb, open("XGBoostClassifier.pickle.dat", "wb"))

**Testing the saved model:**

In [76]:

*# load model from file*

loaded\_model **=** pickle**.**load(open("XGBoostClassifier.pickle.dat", "rb"))

loaded\_model

Out[76]:

XGBClassifier(learning\_rate=0.4, max\_depth=7, missing=nan)

**9. References**

* <https://machinelearningmastery.com/save-gradient-boosting-models-xgboost-python/>