**Phishing Website Detection by Machine Learning Techniques**

*Final project of AI & Cybersecurity Course*

**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 and deep neural nets on the dataset created 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.

*This project is worked on Google Collaboratory.*  
*The required packages for this notebook are imported when needed.*

**2. Loading Data:**

The features are extracted and store in the csv file. The working of this can be seen in the 'Phishing Website Detection\_Feature Extraction.ipynb' file.

The reulted csv file is uploaded to this notebook and stored in the dataframe.

In [0]:

*#importing basic packages*

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

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

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

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

/usr/local/lib/python3.6/dist-packages/statsmodels/tools/\_testing.py:19: FutureWarning: pandas.util.testing is deprecated. Use the functions in the public API at pandas.testing instead.

import pandas.util.testing as tm

In [0]:

*#Loading the data*

data0 **=** pd**.**read\_csv('5.urldata.csv')

data0**.**head()

Out[0]:

|  | **Domain** | **Have\_IP** | **Have\_At** | **URL\_Length** | **URL\_Depth** | **Redirection** | **https\_Domain** | **TinyURL** | **Prefix/Suffix** | **DNS\_Record** | **Web\_Traffic** | **Domain\_Age** | **Domain\_End** | **iFrame** | **Mouse\_Over** | **Right\_Click** | **Web\_Forwards** | **Label** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | graphicriver.net | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| **1** | ecnavi.jp | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| **2** | hubpages.com | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| **3** | extratorrent.cc | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| **4** | icicibank.com | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

**3. Familiarizing with Data**

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

In [0]:

*#Checking the shape of the dataset*

data0**.**shape

Out[0]:

(10000, 18)

In [0]:

*#Listing the features of the dataset*

data0**.**columns

Out[0]:

Index(['Domain', 'Have\_IP', 'Have\_At', 'URL\_Length', 'URL\_Depth',

'Redirection', 'https\_Domain', 'TinyURL', 'Prefix/Suffix', 'DNS\_Record',

'Web\_Traffic', 'Domain\_Age', 'Domain\_End', 'iFrame', 'Mouse\_Over',

'Right\_Click', 'Web\_Forwards', 'Label'],

dtype='object')

In [0]:

*#Information about the dataset*

data0**.**info()

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 10000 entries, 0 to 9999

Data columns (total 18 columns):

# Column Non-Null Count Dtype

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

0 Domain 10000 non-null object

1 Have\_IP 10000 non-null int64

2 Have\_At 10000 non-null int64

3 URL\_Length 10000 non-null int64

4 URL\_Depth 10000 non-null int64

5 Redirection 10000 non-null int64

6 https\_Domain 10000 non-null int64

7 TinyURL 10000 non-null int64

8 Prefix/Suffix 10000 non-null int64

9 DNS\_Record 10000 non-null int64

10 Web\_Traffic 10000 non-null int64

11 Domain\_Age 10000 non-null int64

12 Domain\_End 10000 non-null int64

13 iFrame 10000 non-null int64

14 Mouse\_Over 10000 non-null int64

15 Right\_Click 10000 non-null int64

16 Web\_Forwards 10000 non-null int64

17 Label 10000 non-null int64

dtypes: int64(17), object(1)

memory usage: 1.4+ MB

**4. Visualizing the data**

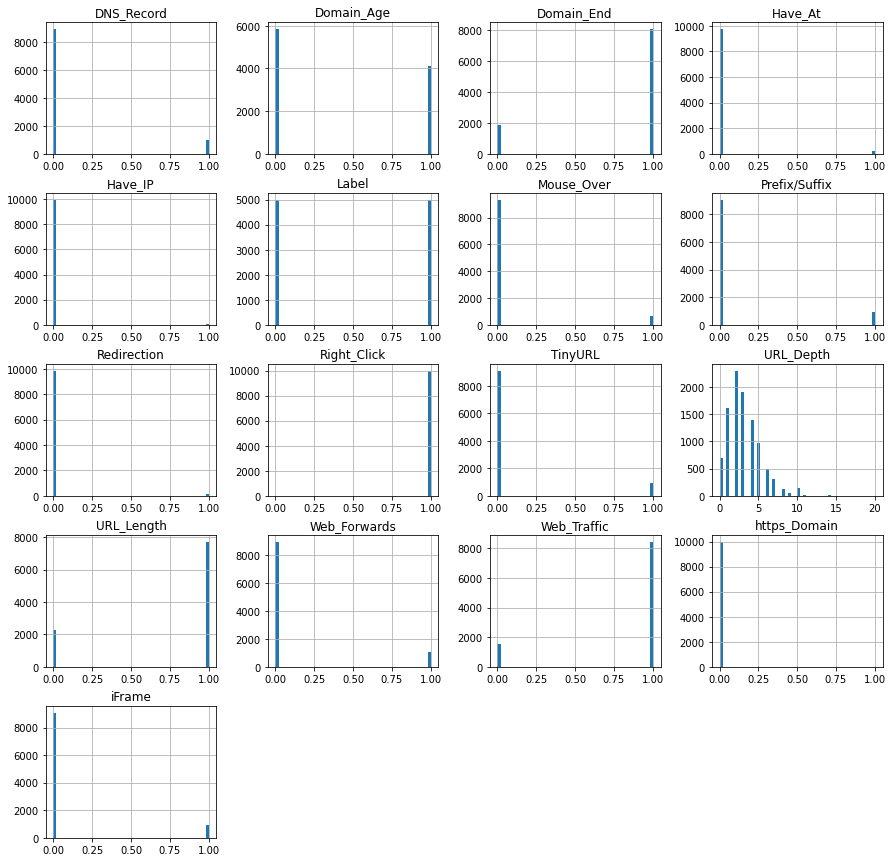
Few plots and graphs are displayed to find how the data is distributed and the how features are related to each other.

In [0]:

*#Plotting the data distribution*

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

plt**.**show()



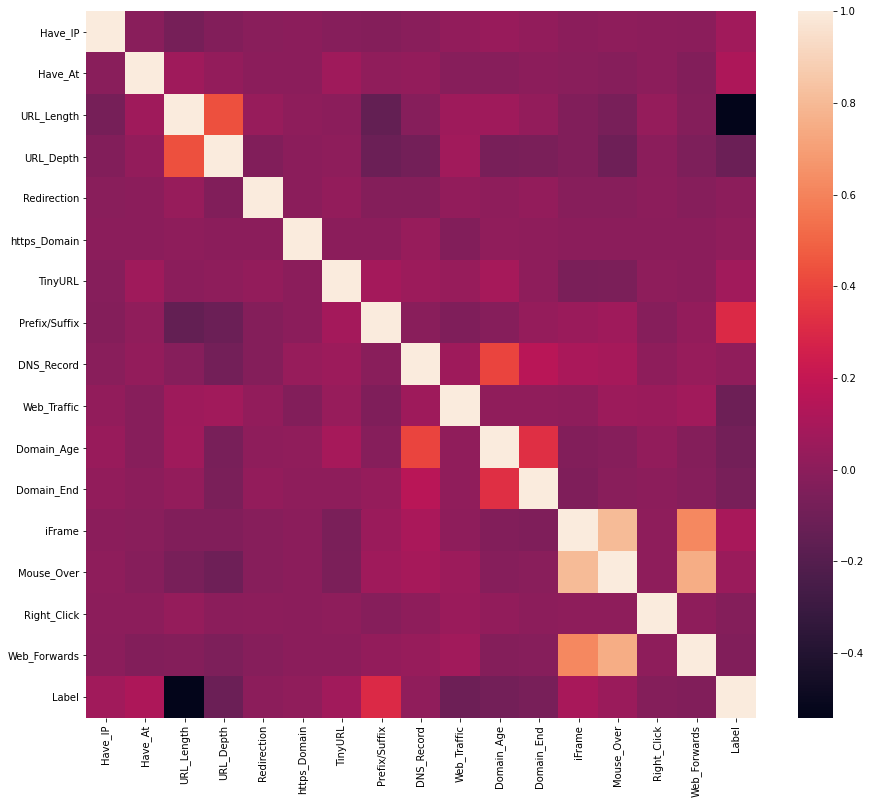
In [0]:

*#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 [0]:

data0**.**describe()

Out[0]:

|  | **Have\_IP** | **Have\_At** | **URL\_Length** | **URL\_Depth** | **Redirection** | **https\_Domain** | **TinyURL** | **Prefix/Suffix** | **DNS\_Record** | **Web\_Traffic** | **Domain\_Age** | **Domain\_End** | **iFrame** | **Mouse\_Over** | **Right\_Click** | **Web\_Forwards** | **Label** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **count** | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.000000 | 10000.0000 | 10000.000000 | 10000.00000 | 10000.00000 | 10000.000000 | 10000.000000 |
| **mean** | 0.005500 | 0.022600 | 0.773400 | 3.072000 | 0.013500 | 0.000200 | 0.090300 | 0.093200 | 0.100800 | 0.845700 | 0.413700 | 0.8099 | 0.090900 | 0.06660 | 0.99930 | 0.105300 | 0.500000 |
| **std** | 0.073961 | 0.148632 | 0.418653 | 2.128631 | 0.115408 | 0.014141 | 0.286625 | 0.290727 | 0.301079 | 0.361254 | 0.492521 | 0.3924 | 0.287481 | 0.24934 | 0.02645 | 0.306955 | 0.500025 |
| **min** | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.0000 | 0.000000 | 0.00000 | 0.00000 | 0.000000 | 0.000000 |
| **25%** | 0.000000 | 0.000000 | 1.000000 | 2.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 1.000000 | 0.000000 | 1.0000 | 0.000000 | 0.00000 | 1.00000 | 0.000000 | 0.000000 |
| **50%** | 0.000000 | 0.000000 | 1.000000 | 3.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 1.000000 | 0.000000 | 1.0000 | 0.000000 | 0.00000 | 1.00000 | 0.000000 | 0.500000 |
| **75%** | 0.000000 | 0.000000 | 1.000000 | 4.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 1.000000 | 1.000000 | 1.0000 | 0.000000 | 0.00000 | 1.00000 | 0.000000 | 1.000000 |
| **max** | 1.000000 | 1.000000 | 1.000000 | 20.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.000000 | 1.0000 | 1.000000 | 1.00000 | 1.00000 | 1.000000 | 1.000000 |

The above obtained result shows that the most of the data is made of 0's & 1's except 'Domain' & 'URL\_Depth' columns. The Domain column doesnt have any significance to the machine learning model training. So dropping the *'Domain'* column from the dataset.

In [0]:

*#Dropping the Domain column*

data **=** data0**.**drop(['Domain'], axis **=** 1)**.**copy()

This leaves us with 16 features & a target column. The *'URL\_Depth'* maximum value is 20. According to my understanding, there is no necessity to change this column.

In [0]:

*#checking the data for null or missing values*

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

Out[0]:

Have\_IP 0

Have\_At 0

URL\_Length 0

URL\_Depth 0

Redirection 0

https\_Domain 0

TinyURL 0

Prefix/Suffix 0

DNS\_Record 0

Web\_Traffic 0

Domain\_Age 0

Domain\_End 0

iFrame 0

Mouse\_Over 0

Right\_Click 0

Web\_Forwards 0

Label 0

dtype: int64

In the feature extraction file, the extracted features of legitmate & phishing url datasets are just concatenated without any shuffling. This resulted in top 5000 rows of legitimate url data & bottom 5000 of phishing url data.

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

In [0]:

*# 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[0]:

|  | **Have\_IP** | **Have\_At** | **URL\_Length** | **URL\_Depth** | **Redirection** | **https\_Domain** | **TinyURL** | **Prefix/Suffix** | **DNS\_Record** | **Web\_Traffic** | **Domain\_Age** | **Domain\_End** | **iFrame** | **Mouse\_Over** | **Right\_Click** | **Web\_Forwards** | **Label** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| **1** | 0 | 0 | 1 | 5 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |
| **2** | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| **3** | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| **4** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 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 [0]:

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

y **=** data['Label']

X **=** data**.**drop('Label',axis**=**1)

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

Out[0]:

((10000, 16), (10000,))

In [0]:

*# 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[0]:

((8000, 16), (2000, 16))

**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
* Multilayer Perceptrons
* XGBoost
* Autoencoder Neural Network
* Support Vector Machines

In [0]:

*#importing packages*

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

In [0]:

*# 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 [0]:

*# 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[0]:

DecisionTreeClassifier(ccp\_alpha=0.0, class\_weight=None, criterion='gini',

max\_depth=5, max\_features=None, max\_leaf\_nodes=None,

min\_impurity\_decrease=0.0, min\_impurity\_split=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, presort='deprecated',

random\_state=None, splitter='best')

In [0]:

*#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 [0]:

*#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: 0.810

Decision Tree: Accuracy on test Data: 0.826

In [0]:

*#checking the feature improtance in the model*

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

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

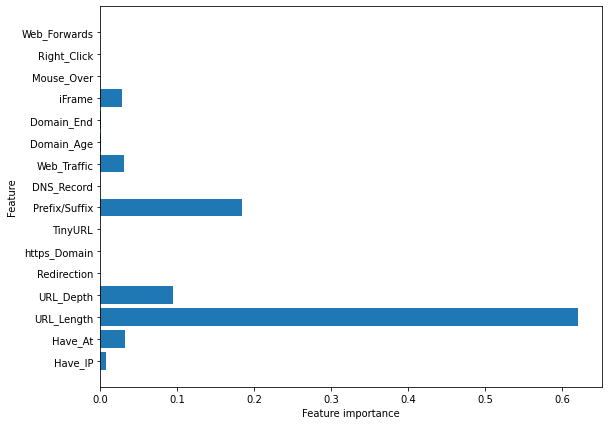
plt**.**barh(range(n\_features), tree**.**feature\_importances\_, align**=**'center')

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

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

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

plt**.**show()



**Storing the results:**

In [0]:

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

*#Caution: Execute only once to avoid duplications.*

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 [0]:

*# 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[0]:

RandomForestClassifier(bootstrap=True, ccp\_alpha=0.0, class\_weight=None,

criterion='gini', max\_depth=5, max\_features='auto',

max\_leaf\_nodes=None, max\_samples=None,

min\_impurity\_decrease=0.0, min\_impurity\_split=None,

min\_samples\_leaf=1, min\_samples\_split=2,

min\_weight\_fraction\_leaf=0.0, n\_estimators=100,

n\_jobs=None, oob\_score=False, random\_state=None,

verbose=0, warm\_start=False)

In [0]:

*#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 [0]:

*#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: 0.814

Random forest: Accuracy on test Data: 0.834

In [0]:

*#checking the feature improtance in the model*

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

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

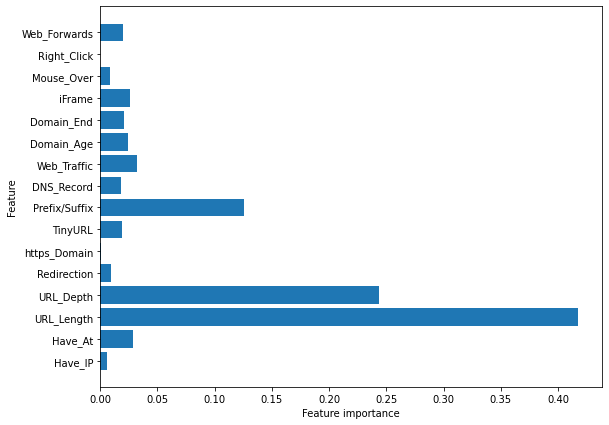
plt**.**barh(range(n\_features), forest**.**feature\_importances\_, align**=**'center')

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

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

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

plt**.**show()



**Storing the results:**

In [0]:

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

*#Caution: Execute only once to avoid duplications.*

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

**7.3. Multilayer Perceptrons (MLPs): Deep Learning**

Multilayer perceptrons (MLPs) are also known as (vanilla) feed-forward neural networks, or sometimes just neural networks. Multilayer perceptrons can be applied for both classification and regression problems.

MLPs can be viewed as generalizations of linear models that perform multiple stages of processing to come to a decision.

In [0]:

*# Multilayer Perceptrons model*

**from** sklearn.neural\_network **import** MLPClassifier

*# instantiate the model*

mlp **=** MLPClassifier(alpha**=**0.001, hidden\_layer\_sizes**=**([100,100,100]))

*# fit the model*

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

Out[0]:

MLPClassifier(activation='relu', alpha=0.001, batch\_size='auto', beta\_1=0.9,

beta\_2=0.999, early\_stopping=False, epsilon=1e-08,

hidden\_layer\_sizes=[100, 100, 100], learning\_rate='constant',

learning\_rate\_init=0.001, max\_fun=15000, max\_iter=200,

momentum=0.9, n\_iter\_no\_change=10, nesterovs\_momentum=True,

power\_t=0.5, random\_state=None, shuffle=True, solver='adam',

tol=0.0001, validation\_fraction=0.1, verbose=False,

warm\_start=False)

In [0]:

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

y\_test\_mlp **=** mlp**.**predict(X\_test)

y\_train\_mlp **=** mlp**.**predict(X\_train)

**Performance Evaluation:**

In [0]:

*#computing the accuracy of the model performance*

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

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

print("Multilayer Perceptrons: Accuracy on training Data: {:.3f}"**.**format(acc\_train\_mlp))

print("Multilayer Perceptrons: Accuracy on test Data: {:.3f}"**.**format(acc\_test\_mlp))

Multilayer Perceptrons: Accuracy on training Data: 0.859

Multilayer Perceptrons: Accuracy on test Data: 0.863

**Storing the results:**

In [0]:

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

*#Caution: Execute only once to avoid duplications.*

storeResults('Multilayer Perceptrons', acc\_train\_mlp, acc\_test\_mlp)

**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 [0]:

*#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[0]:

XGBClassifier(base\_score=0.5, booster='gbtree', colsample\_bylevel=1,

colsample\_bynode=1, colsample\_bytree=1, gamma=0,

learning\_rate=0.4, max\_delta\_step=0, max\_depth=7,

min\_child\_weight=1, missing=None, n\_estimators=100, n\_jobs=1,

nthread=None, objective='binary:logistic', random\_state=0,

reg\_alpha=0, reg\_lambda=1, scale\_pos\_weight=1, seed=None,

silent=None, subsample=1, verbosity=1)

In [0]:

*#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 [0]:

*#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: 0.866

XGBoost : Accuracy on test Data: 0.864

**Storing the results:**

In [0]:

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

*#Caution: Execute only once to avoid duplications.*

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

**7.5. Autoencoder Neural Network**

An auto encoder is a neural network that has the same number of input neurons as it does outputs. The hidden layers of the neural network will have fewer neurons than the input/output neurons. Because there are fewer neurons, the auto-encoder must learn to encode the input to the fewer hidden neurons. The predictors (x) and output (y) are exactly the same in an auto encoder.

In [0]:

*#importing required packages*

**import** keras

**from** keras.layers **import** Input, Dense

**from** keras **import** regularizers

**import** tensorflow **as** tf

**from** keras.models **import** Model

**from** sklearn **import** metrics

Using TensorFlow backend.

In [0]:

*#building autoencoder model*

input\_dim **=** X\_train**.**shape[1]

encoding\_dim **=** input\_dim

input\_layer **=** Input(shape**=**(input\_dim, ))

encoder **=** Dense(encoding\_dim, activation**=**"relu",

activity\_regularizer**=**regularizers**.**l1(10e-4))(input\_layer)

encoder **=** Dense(int(encoding\_dim), activation**=**"relu")(encoder)

encoder **=** Dense(int(encoding\_dim**-**2), activation**=**"relu")(encoder)

code **=** Dense(int(encoding\_dim**-**4), activation**=**'relu')(encoder)

decoder **=** Dense(int(encoding\_dim**-**2), activation**=**'relu')(code)

decoder **=** Dense(int(encoding\_dim), activation**=**'relu')(encoder)

decoder **=** Dense(input\_dim, activation**=**'relu')(decoder)

autoencoder **=** Model(inputs**=**input\_layer, outputs**=**decoder)

autoencoder**.**summary()

Model: "model\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) (None, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 16) 272

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_2 (Dense) (None, 16) 272

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 14) 238

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_6 (Dense) (None, 16) 240

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_7 (Dense) (None, 16) 272

=================================================================

Total params: 1,294

Trainable params: 1,294

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [0]:

*#compiling the model*

autoencoder**.**compile(optimizer**=**'adam',

loss**=**'binary\_crossentropy',

metrics**=**['accuracy'])

*#Training the model*

history **=** autoencoder**.**fit(X\_train, X\_train, epochs**=**10, batch\_size**=**64, shuffle**=True**, validation\_split**=**0.2)

Train on 6400 samples, validate on 1600 samples

Epoch 1/10

6400/6400 [==============================] - 0s 51us/step - loss: 1.3997 - accuracy: 0.7132 - val\_loss: -0.3941 - val\_accuracy: 0.7890

Epoch 2/10

6400/6400 [==============================] - 0s 24us/step - loss: -0.4269 - accuracy: 0.7821 - val\_loss: -0.5190 - val\_accuracy: 0.7812

Epoch 3/10

6400/6400 [==============================] - 0s 24us/step - loss: -1.0514 - accuracy: 0.7908 - val\_loss: -1.3147 - val\_accuracy: 0.8149

Epoch 4/10

6400/6400 [==============================] - 0s 24us/step - loss: -1.3118 - accuracy: 0.8200 - val\_loss: -1.3532 - val\_accuracy: 0.8128

Epoch 5/10

6400/6400 [==============================] - 0s 25us/step - loss: -1.3789 - accuracy: 0.8168 - val\_loss: -1.4710 - val\_accuracy: 0.8190

Epoch 6/10

6400/6400 [==============================] - 0s 25us/step - loss: -1.4435 - accuracy: 0.8187 - val\_loss: -1.5160 - val\_accuracy: 0.8204

Epoch 7/10

6400/6400 [==============================] - 0s 25us/step - loss: -1.4951 - accuracy: 0.8215 - val\_loss: -1.5601 - val\_accuracy: 0.8240

Epoch 8/10

6400/6400 [==============================] - 0s 23us/step - loss: -1.5208 - accuracy: 0.8192 - val\_loss: -1.5912 - val\_accuracy: 0.8236

Epoch 9/10

6400/6400 [==============================] - 0s 25us/step - loss: -1.5044 - accuracy: 0.8140 - val\_loss: -1.5868 - val\_accuracy: 0.8191

Epoch 10/10

6400/6400 [==============================] - 0s 25us/step - loss: -1.5554 - accuracy: 0.8214 - val\_loss: -1.6153 - val\_accuracy: 0.8205

**Performance Evaluation:**

In [0]:

acc\_train\_auto **=** autoencoder**.**evaluate(X\_train, X\_train)[1]

acc\_test\_auto **=** autoencoder**.**evaluate(X\_test, X\_test)[1]

print('\nAutoencoder: Accuracy on training Data: {:.3f}' **.**format(acc\_train\_auto))

print('Autoencoder: Accuracy on test Data: {:.3f}' **.**format(acc\_test\_auto))

8000/8000 [==============================] - 0s 18us/step

2000/2000 [==============================] - 0s 20us/step

Autoencoder: Accuracy on training Data: 0.819

Autoencoder: Accuracy on test Data: 0.818

**Storing the results:**

In [0]:

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

*#Caution: Execute only once to avoid duplications.*

storeResults('AutoEncoder', acc\_train\_auto, acc\_test\_auto)

**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 [0]:

*#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[0]:

SVC(C=1.0, break\_ties=False, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='scale', kernel='linear',

max\_iter=-1, probability=False, random\_state=12, shrinking=True, tol=0.001,

verbose=False)

In [0]:

*#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 [0]:

*#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: 0.798

SVM : Accuracy on test Data: 0.818

**Storing the results:**

In [0]:

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

*#Caution: Execute only once to avoid duplications.*

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 [0]:

*#creating dataframe*

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

'Train Accuracy': acc\_train,

'Test Accuracy': acc\_test})

results

Out[0]:

|  | **ML Model** | **Train Accuracy** | **Test Accuracy** |
| --- | --- | --- | --- |
| **0** | Decision Tree | 0.810 | 0.826 |
| **1** | Random Forest | 0.814 | 0.834 |
| **2** | Multilayer Perceptrons | 0.858 | 0.863 |
| **3** | XGBoost | 0.866 | 0.864 |
| **4** | AutoEncoder | 0.819 | 0.818 |
| **5** | SVM | 0.798 | 0.818 |

In [0]:

*#Sorting the datafram on accuracy*

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

Out[0]:

|  | **ML Model** | **Train Accuracy** | **Test Accuracy** |
| --- | --- | --- | --- |
| **3** | XGBoost | 0.866 | 0.864 |
| **2** | Multilayer Perceptrons | 0.858 | 0.863 |
| **1** | Random Forest | 0.814 | 0.834 |
| **0** | Decision Tree | 0.810 | 0.826 |
| **4** | AutoEncoder | 0.819 | 0.818 |
| **5** | SVM | 0.798 | 0.818 |

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

So, saving the model for future use.

In [0]:

*# save XGBoost model to file*

**import** pickle

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

**Testing the saved model:**

In [0]:

*# load model from file*

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

loaded\_model

Out[0]:

XGBClassifier(base\_score=0.5, booster='gbtree', colsample\_bylevel=1,

colsample\_bynode=1, colsample\_bytree=1, gamma=0,

learning\_rate=0.4, max\_delta\_step=0, max\_depth=7,

min\_child\_weight=1, missing=nan, n\_estimators=100, n\_jobs=1,

nthread=None, objective='binary:logistic', random\_state=0,

reg\_alpha=0, reg\_lambda=1, scale\_pos\_weight=1, seed=None,

silent=None, subsample=1, verbosity=1)

**9. References**

* <https://blog.keras.io/building-autoencoders-in-keras.html>
* <https://en.wikipedia.org/wiki/Autoencoder>
* <https://mc.ai/a-beginners-guide-to-build-stacked-autoencoder-and-tying-weights-with-it/>
* <https://github.com/shreyagopal/t81_558_deep_learning/blob/master/t81_558_class_14_03_anomaly.ipynb>
* <https://machinelearningmastery.com/save-gradient-boosting-models-xgboost-python/>