Cyber Security

DESCRIPTION

Problem Statement:

Book-My-Show will enable the ads on their website, but they are also very cautious about their user privacy and information who visit their website. Some ads URL could contain a malicious link that can trick any recipient and lead to a malware installation, freezing the system as part of a ransomware attack or revealing sensitive information. Book-My-Show now wants to analyze that whether the particular URL is prone to phishing (malicious) or not.

Dataset Details:

The input dataset contains an 11k sample corresponding to the 11k URL. Each sample contains 32 features that give a different and unique description of URL ranging from -1,0,1.

1: Phishing

0: Suspicious

1: Legitimate

The sample could be either legitimate or phishing.

Project Task: Week 1

Exploratory Data Analysis:

Each sample has 32 features ranging from -1,0,1. Explore the data using histogram, heatmaps.

Determine the number of samples present in the data, unique elements in all the features.

Check if there is any null value in any features.

Correlation of features and feature selection:

Next, we have to find if there are any correlated features present in the data. Remove the feature which might be correlated with some threshold.

Project Task: Week 2

Building Classification Model

Finally, build a robust classification system that classifies whether the URL sample is a phishing site or not.

Build classification models using a binary classifier to detect malicious or phishing URLs.

Illustrate the diagnostic ability of this binary classifier by plotting the ROC curve.

Validate the accuracy of data by the K-Fold cross-validation technique.

The final output consists of the model, which will give maximum accuracy on the validation dataset with selected attributes.

Import necessary modules

In [133…

```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')

pd.options.display.max_columns = 999

# pre-processing
from sklearn.decomposition import PCA

# import libraries for model validation
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import LeaveOneOut


# import the ML algorithm
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import LogisticRegression
from statsmodels.tools.eval_measures import rmse
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier


# import libraries for metrics and reporting
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn import metrics
from sklearn.metrics import classification_report

from sklearn.model_selection import GridSearchCV
from sklearn.feature_selection import SelectKBest
from sklearn.pipeline import make_pipeline
from sklearn.pipeline import Pipeline

# Grid searching key hyperparametres for logistic regression
from sklearn.model_selection import RepeatedStratifiedKFold
```

Load Data

In [36]:

```
import os
for dirname, _, filenames in os.walk('.\input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

```
.\input\dataset.csv
```

In [37]:

```
file=r'.\input\dataset.csv'
```

In [38]:

```
df=pd.read_csv(file,header=0)
print("Training Data has ", df.shape[0],"Rows and ",df.shape[1], "Columns", )
```

```
Training Data has  11055 Rows and  32 Columns
```

Exploratory Data Analysis

In [9]:

```
df.sample(5)
```

Out[9]:

	index	having_IPhaving_IP_Address	URLURL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix_Suffix	having_
7750	7751	-1	-1	1	1	1	-1	
4485	4486	1	-1	1	1	1	-1	
747	748	-1	-1	-1	1	-1	-1	
9628	9629	1	-1	1	1	1	-1	
6462	6463	-1	-1	1	1	1	-1	

In [10]:

```
df.dtypes
```

Out[10]:

```
index                          int64
having_IPhaving_IP_Address     int64
URLURL_Length                  int64
Shortining_Service             int64
having_At_Symbol               int64
double_slash_redirecting       int64
Prefix_Suffix                  int64
having_Sub_Domain              int64
SSLfinal_State                 int64
Domain_registeration_length    int64
Favicon                        int64
port                           int64
HTTPS_token                    int64
Request_URL                    int64
URL_of_Anchor                  int64
Links_in_tags                  int64
SFH                            int64
Submitting_to_email            int64
Abnormal_URL                   int64
Redirect                       int64
on_mouseover                   int64
RightClick                     int64
popUpWidnow                    int64
Iframe                         int64
age_of_domain                  int64
DNSRecord                      int64
web_traffic                    int64
Page_Rank                      int64
Google_Index                   int64
Links_pointing_to_page         int64
Statistical_report             int64
Result                         int64
dtype: object
```

In [11]:

```
df.describe(include='all')
```

Out[11]:

	index	having_IPhaving_IP_Address	URLURL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix_Suffix
count	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
std	3191.447947	0.949534	0.766095	0.673998	0.713598	0.671011	0.678139
min	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
50%	5528.000000	1.000000	-1.000000	1.000000	1.000000	1.000000	-1.000000
75%	8291.500000	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

In [41]:

```
df.columns
```

Out[41]:

```
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')
```

Removing Index column as this column will not have any significance in training and testing/prediction

In [42]:

```
df=df[['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']]
```

In [43]:

```
df.dtypes
```

Out[43]:

```
having_IPhaving_IP_Address     int64
URLURL_Length                  int64
Shortining_Service             int64
having_At_Symbol               int64
double_slash_redirecting       int64
Prefix_Suffix                  int64
having_Sub_Domain              int64
SSLfinal_State                 int64
Domain_registeration_length    int64
Favicon                        int64
port                           int64
HTTPS_token                    int64
Request_URL                    int64
URL_of_Anchor                  int64
Links_in_tags                  int64
SFH                            int64
Submitting_to_email            int64
Abnormal_URL                   int64
Redirect                       int64
on_mouseover                   int64
RightClick                     int64
popUpWidnow                    int64
Iframe                         int64
age_of_domain                  int64
DNSRecord                      int64
web_traffic                    int64
Page_Rank                      int64
Google_Index                   int64
Links_pointing_to_page         int64
Statistical_report             int64
Result                         int64
dtype: object
```

Each sample has 30 features ranging from -1,0,1. Explore the data using histogram, heatmaps.

Histogram

In [15]:

```
cols=['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']
for col in cols:
    plt.figure(figsize=(18,5))
    plt.hist(df[col],bins=3,color = 'green',density=False)
    plt.xlabel(col,fontsize=10)
    plt.ylabel('Count',fontsize=10)
    plt.xticks(fontsize=8)
    plt.yticks(fontsize=8)
```

Heatmap

In [16]:

```
df.columns
```

Out[16]:

```
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 [17]:

```
corr = df.corr()
corr
```

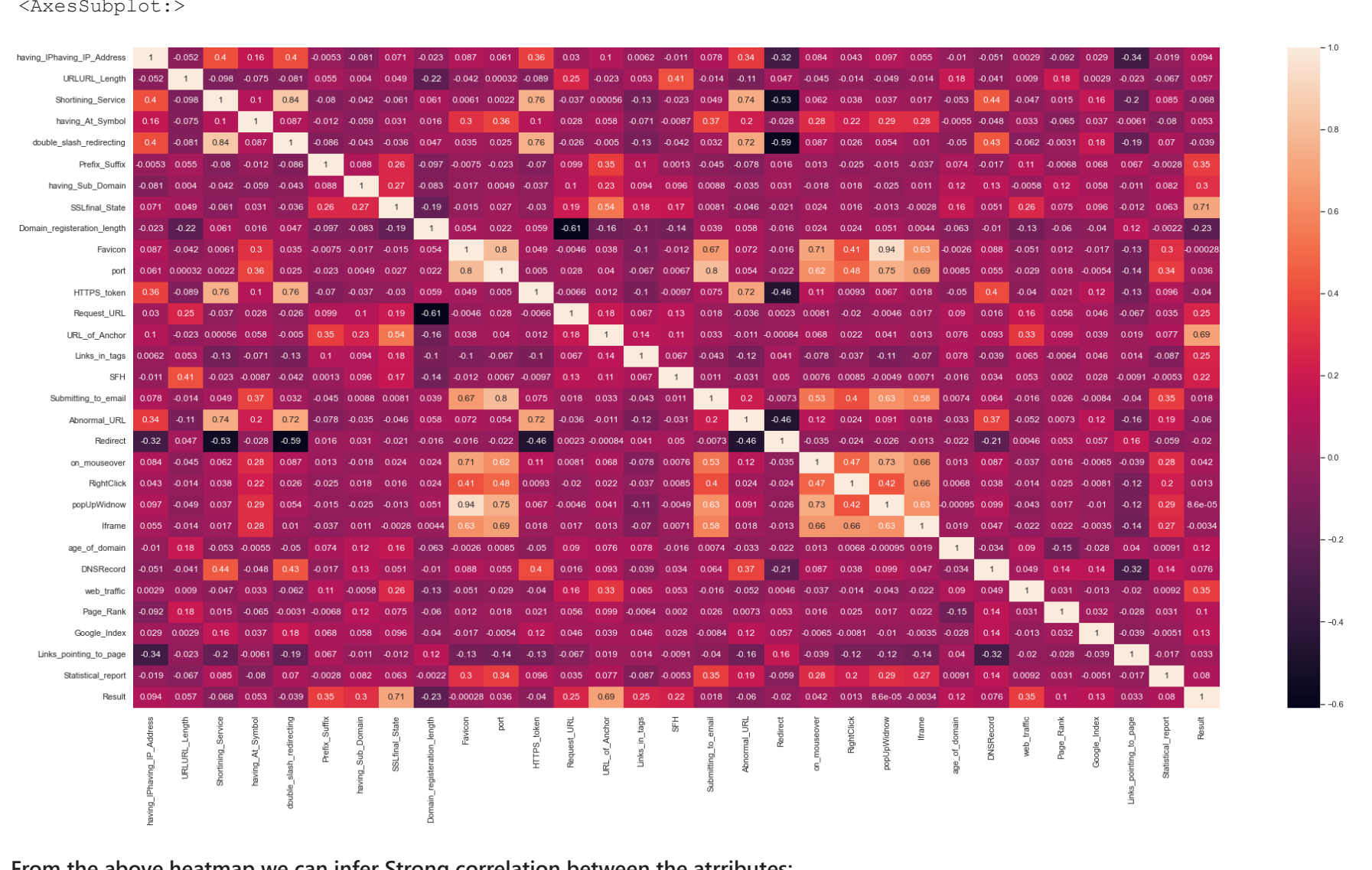
Out[17]:

	having_IPhaving_IP_Address	URLURL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix
having_IPhaving_IP_Address	1.000000	-0.052411	0.403461	0.158699	0.397389	-0.
URLURL_Length	-0.052411	1.000000	-0.097881	-0.075108	-0.081247	0.
Shortining_Service	0.403461	-0.097881	1.000000	0.104447	0.842796	-0.
having_At_Symbol	0.158699	-0.075108	0.104447	1.000000	0.086960	-0.
double_slash_redirecting	0.397389	-0.081247	0.842796	0.086960	1.000000	-0.
Prefix_Suffix	-0.005257	0.055247	-0.080471	-0.011726	-0.085590	1.
having_Sub_Domain	-0.080745	0.003997	-0.041916	-0.058976	-0.043079	0.
SSLfinal_State	0.071414	0.048754	-0.061426	0.031220	-0.036200	0.
Domain_registeration_length	-0.022739	-0.221892	0.060923	0.015522	0.047464	-0.
Favicon	0.087025	-0.042497	0.006101	0.304899	0.035100	-0.
port	0.060979	0.000323	0.002201	0.364891	0.025060	-0.
HTTPS_token	0.363534	-0.089383	0.757838	0.104561	0.760799	-0.
Request_URL	0.029773	0.246348	-0.037235	0.027909	-0.026368	0.
URL_of_Anchor	0.099847	-0.023396	0.000561	0.057914	-0.005036	0.
Links_in_tags	0.006212	0.052869	-0.133379	-0.070861	-0.125583	0.
SFH	-0.010962	0.414196	-0.022723	-0.008672	-0.041672	0.
Submitting_to_email	0.077989	-0.014457	0.049328	0.370123	0.031898	-0.
Abnormal_URL	0.336549	-0.106761	0.739290	0.203945	0.723724	-0.
Redirect	-0.321181	0.046832	-0.534530	-0.028160	-0.591478	0.
on_mouseover	0.084059	-0.045103	0.062383	0.279697	0.086635	0.
RightClick	0.042881	-0.013613	0.038118	0.219503	0.025863	-0.
popUpWidnow	0.096882	-0.049381	0.036616	0.290893	0.054463	-0.
Iframe	0.054694	-0.013838	0.016581	0.284410	0.010459	-0.
age_of_domain	-0.010446	0.179426	-0.052596	-0.005499	-0.050107	0.
DNSRecord	-0.050733	-0.040823	0.436064	-0.047872	0.431409	-0.
web_traffic	0.002922	0.008993	-0.047074	0.032918	-0.062369	0.
Page_Rank	-0.091774	0.183518	0.014591	-0.064735	-0.003132	-0.
Google_Index	0.029153	0.002902	0.155844	0.037061	0.178415	0.
Links_pointing_to_page	-0.339065	-0.022987	-0.198410	-0.006080	-0.194165	0.
Statistical_report	-0.019103	-0.067153	0.085461	-0.080357	0.070390	-0.
Result	0.094160	0.057430	-0.067966	0.052948	-0.038608	0.

In [18]:

```
sns.set(rc={'figure.figsize':(32,16)})
sns.heatmap(corr, annot = True)
```

Out[18]: <AxesSubplot:>



From the above heatmap we can infer Strong correlation between the atrributes:

In [19]:

```
corr[corr<1].unstack().sort_values(ascending = False).drop_duplicates().head(20)
```

Out[19]:

```
popUpWidnow               Favicon                     0.939633
Shortining_Service        double_slash_redirecting    0.842796
Favicon                   port                        0.803834
Submitting_to_email       port                        0.799088
double_slash_redirecting  HTTPS_token                 0.760799
HTTPS_token               Shortining_Service          0.757838
popUpWidnow               port                        0.748517
Abnormal_URL              Shortining_Service          0.739290
popUpWidnow               on_mouseover                0.733629
double_slash_redirecting  Abnormal_URL                0.723724
HTTPS_token               Abnormal_URL                0.716287
SSLfinal_State            Result                      0.714741
Favicon                   on_mouseover                0.706179
Result                    URL_of_Anchor               0.692935
Iframe                    port                        0.687044
Favicon                   Submitting_to_email         0.668317
Iframe                    on_mouseover                0.659478
                          RightClick                  0.655863
Submitting_to_email       popUpWidnow                 0.629462
popUpWidnow               Iframe                      0.629406
dtype: float64
```

Determine the number of samples present in the data, unique elements in all the features.

In [20]:

```
print("Training Data has ", df.shape[0],"Rows and ",df.shape[1], "Columns", )
```

```
Training Data has  11055 Rows and  31 Columns
```

In [21]:

```
# display distinct value for each feature
for col_name in df.columns:
 print('The No of unique values in {' +col_name+ '} are: ', df[col_name].nunique() , ' Unique Values are: ', df
 #print(df[col_name].unique())
```

```
The No of unique values in {having_IPhaving_IP_Address} are:  2  Unique Values are:  [-1  1]
The No of unique values in {URLURL_Length} are:  3  Unique Values are:  [ 1  0 -1]
The No of unique values in {Shortining_Service} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {having_At_Symbol} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {double_slash_redirecting} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Prefix_Suffix} are:  2  Unique Values are:  [-1  1]
The No of unique values in {having_Sub_Domain} are:  3  Unique Values are:  [-1  0  1]
The No of unique values in {SSLfinal_State} are:  3  Unique Values are:  [-1  1  0]
The No of unique values in {Domain_registeration_length} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Favicon} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {port} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {HTTPS_token} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Request_URL} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {URL_of_Anchor} are:  3  Unique Values are:  [-1  0  1]
The No of unique values in {Links_in_tags} are:  3  Unique Values are:  [ 1 -1  0]
The No of unique values in {SFH} are:  3  Unique Values are:  [-1  1  0]
The No of unique values in {Submitting_to_email} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Abnormal_URL} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Redirect} are:  2  Unique Values are:  [0 1]
The No of unique values in {on_mouseover} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {RightClick} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {popUpWidnow} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {Iframe} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {age_of_domain} are:  2  Unique Values are:  [-1  1]
The No of unique values in {DNSRecord} are:  2  Unique Values are:  [-1  1]
The No of unique values in {web_traffic} are:  3  Unique Values are:  [-1  0  1]
The No of unique values in {Page_Rank} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Google_Index} are:  2  Unique Values are:  [ 1 -1]
The No of unique values in {Links_pointing_to_page} are:  3  Unique Values are:  [ 1  0 -1]
The No of unique values in {Statistical_report} are:  2  Unique Values are:  [-1  1]
The No of unique values in {Result} are:  2  Unique Values are:  [-1  1]
```

Check if there is any null value in any features.

In [22]:

```
df.isnull().sum().sum()
```

Out[22]: 0

In [23]:

```
NAs= pd.concat([df.isnull().sum()],axis=1,keys=['Count of NANs'])
NAs.sample(5)
```

Out[23]:

	Count of NANs
SSLfinal_State	0
DNSRecord	0
having_At_Symbol	0
Prefix_Suffix	0
Shortining_Service	0

In [24]:

```
NAs[NAs.sum(axis=1)>0]
```

Out[24]:

Count of NANs

It is good that there is no missing values or no null values

Now let us have a look at the data type of all the variables present in the training dataset

In [25]:

```
# Now let us have a look at the data type of all the variables present in the training dataset
dtype_data=df.dtypes.reset_index()
dtype_data.columns = ["Count", "Column Type"]
dtype_data.groupby("Column Type").aggregate('count').reset_index()
```

Out[25]:

	Column Type	Count
0	int64	31

Code to replace missing values in any future dataset for categorical columns with mode

In [26]:

```
for column in df.columns:
    mode = df[column].mode()
    df[column] = df[column].fillna(mode)
```

Check and remove duplicate Values

In [44]:

```
df
```

Out[44]:

	having_IPhaving_IP_Address	URLURL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix_Suffix	having_Sub_D
0	-1	1	1	1	-1	-1	
1	1	1	1	1	1	-1	
2	1	0	1	1	1	-1	
3	1	0	1	1	1	-1	
4	1	0	-1	1	1	-1	
...	...	...	...	...	...	...	
11050	1	-1	1	-1	1	1	
11051	-1	1	1	-1	-1	-1	
11052	1	-1	1	1	1	-1	
11053	-1	-1	1	1	1	-1	
11054	-1	-1	1	1	1	-1	

11055 rows × 31 columns

In [45]:

```
# Selecting duplicate rows except first occurrence based on all columns
duplicate = df[df.duplicated()]
duplicate
```

Out[45]:

	having_IPhaving_IP_Address	URLURL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix_Suffix	having_Sub_D
180	1	-1	1	1	1	-1	
203	1	-1	1	1	1	-1	
323	1	-1	1	1	1	-1	
364	1	-1	1	1	1	-1	
381	1	-1	1	1	1	-1	
...	...	...	...	...	...	...	
11047	-1	-1	1	1	1	-1	
11050	1	-1	1	-1	1	1	
11051	-1	1	1	-1	-1	-1	
11052	1	-1	1	1	1	-1	
11053	-1	-1	1	1	1	-1	

5206 rows × 31 columns

So there are 5206 duplicate values which needs to be removed

In [46]:

```
# dropping duplicate values
df.drop_duplicates(keep='first',inplace=True)
df
```

Out[46]:

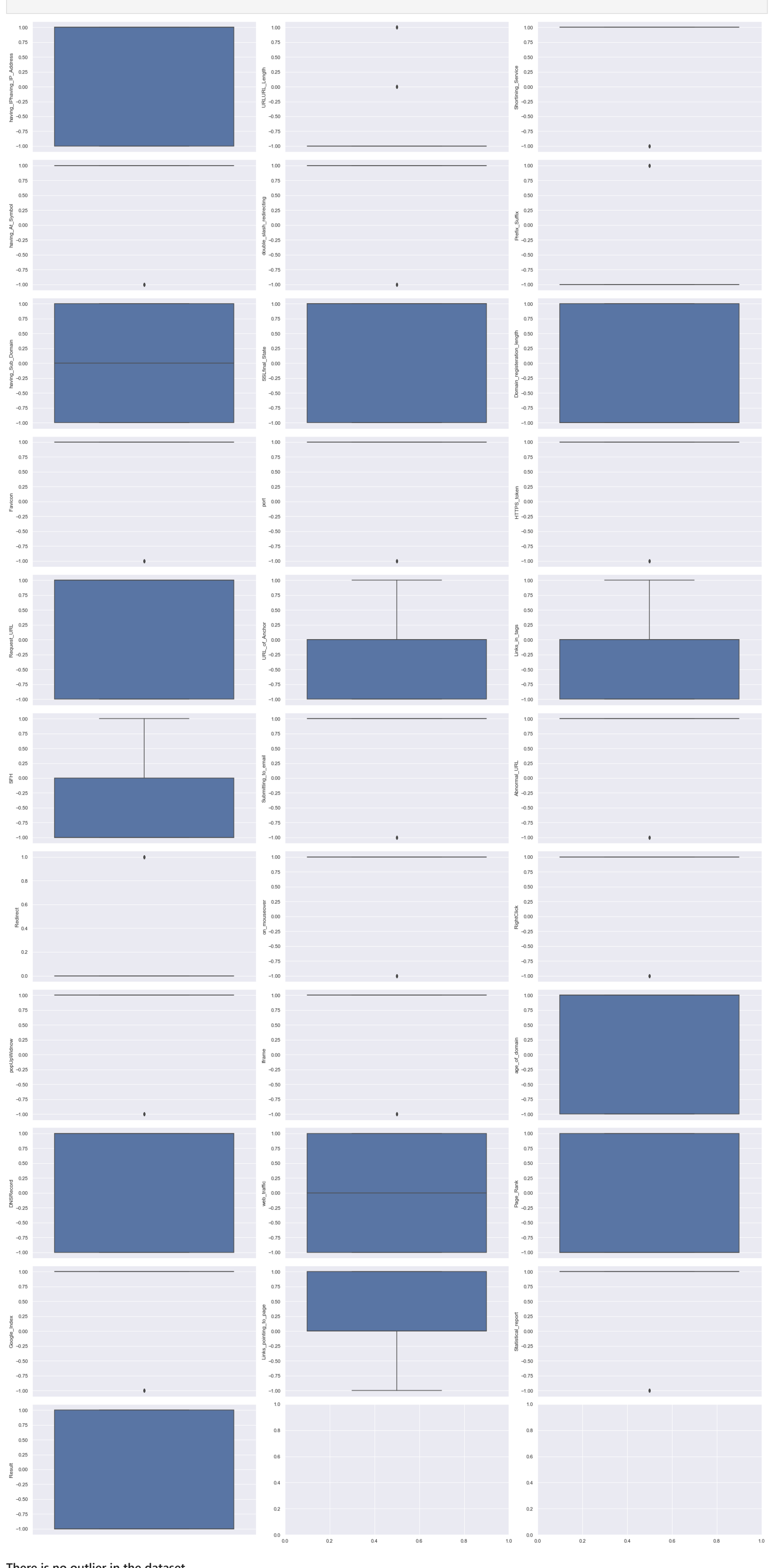
	having_IPhaving_IP_Address	URLURL_Length	Shortining_Service	having_At_Symbol	double_slash_redirecting	Prefix_Suffix	having_Sub_D
0	-1	1	1	1	-1	-1	
1	1	1	1	1	1	-1	
2	1	0	1	1	1	-1	
3	1	0	1	1	1	-1	
4	1	0	-1	1	1	-1	
...	...	...	...	...	...	...	
11037	1	-1	-1	1	-1	-1	
11045	1	-1	1	1	1	-1	
11048	1	-1	1	1	1	-1	
11049	-1	-1	1	1	-1	-1	
11054	-1	-1	1	1	1	-1	

5849 rows × 31 columns

Remove Outlier if any

In [66]:

```
names = list(df.columns)
f, axes = plt.subplots(round(len(names)/3)+1, 3,figsize=(24, 48))
y = 0
for name in names:
    i, j = divmod(y, 3)
    sns.boxplot(y=df[name], ax=axes[i, j])
    y = y + 1

plt.tight_layout()
plt.show()
```



There is no outlier in the dataset

In [87]:

```
names = list(df.columns)
for name in names:
    # IQR
    Q1 = np.percentile(df[name], 25,interpolation = 'midpoint')
    Q3 = np.percentile(df[name], 75,interpolation = 'midpoint')
    IQR = Q3 - Q1
    upper = Q3 +1.5*IQR
    lower = Q1 - 1.5*IQR
    # Upper bound
    upperbound = np.where(df[name] >= (Q3+1.5*IQR))
    # Lower bound
    lowerbound = np.where(df[name] <= (Q1-1.5*IQR))
    ''' Removing the Outliers '''
    if len(upperbound)>1:
        df_out.drop(upper[0], inplace = True)
    if len(lowerbound)>1:
        df_out.drop(lower[0], inplace = True)
```

Correlation of features and feature selection:

Find if there are any correlated features present in the data. Remove the feature which might be correlated with some threshold.

In [26]:

```
correlated_df=corr[corr<1].unstack().sort_values(ascending = False).drop_duplicates().head(20)
correlated_df
```

Out[26]:

```
popUpWidnow               Favicon                     0.939633
Shortining_Service        double_slash_redirecting    0.842796
Favicon                   port                        0.803834
Submitting_to_email       port                        0.799088
double_slash_redirecting  HTTPS_token                 0.760799
HTTPS_token               Shortining_Service          0.757838
popUpWidnow               port                        0.748517
Abnormal_URL              Shortining_Service          0.739290
popUpWidnow               on_mouseover                0.733629
double_slash_redirecting  Abnormal_URL                0.723724
HTTPS_token               Abnormal_URL                0.716287
SSLfinal_State            Result                      0.714741
Favicon                   on_mouseover                0.706179
Result                    URL_of_Anchor               0.692935
Iframe                    port                        0.687044
Favicon                   Submitting_to_email         0.668317
Iframe                    on_mouseover                0.659478
                          RightClick                  0.655863
Submitting_to_email       popUpWidnow                 0.629462
popUpWidnow               Iframe                      0.629406
dtype: float64
```

In [27]:

```
corr_threshold=.8
correlated_df[correlated_df>corr_threshold]
```

Out[27]:

```
popUpWidnow         Favicon                     0.939633
Shortining_Service  double_slash_redirecting    0.842796
Favicon             port                        0.803834
dtype: float64
```

It seems that due to strong correlation with Favicon we can remove popUpWidnow. However will not remove any column now.

Check columns with 0 std dev.These 0 std dev features are constant.Since these features don't vary, it will have no effect on the model performance

In [91]:

```
cols_to_drop=[]
for each_col in df.drop(['Result'],axis=1).columns:
    if df[each_col].dtypes !='O': # not an object col
        if df[each_col].std() == 0.00:
            stats= each_col + ' { ' \
            ' std : ' + str(df[each_col].std()) + \
            ' Min : ' + str(df[each_col].min()) + \
            ' Max : ' + str(df[each_col].max()) + " } "
            print(stats)
            cols_to_drop.append(each_col)
print('Number of features with 0 std : {}'.format(len(cols_to_drop)))
```

```
Number of features with 0 std : 0
```

Encoding like LabelEncoding, One hot encoding not applicable as the dataset does not have any categorical attribute

Standardization & Normalization

All the values in the dataset are either -1,0,1, so scaling like Min Max Scalar, Robust Scalar,Standard Scalar is also not applicable

PCA

In [92]:

```
df.columns
```

Out[92]:

```
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 [102…

```
# split data table into data X and class labels y

X = df.iloc[:,0:30].values
y = df.iloc[:,30].values
```

In [103…

```
# We did not specify the number of components in the constructor.
# Hence, all 30 of the features in the feature set will be returned for both the training and
# test sets.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA()
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [104…

```
# The PCA class contains explained_variance_ratio_ which returns the variance caused by each of the
# principal components.
explained_variance = pca.explained_variance_ratio_
explained_variance
```

Out[104…

```
array([0.16229804, 0.11443812, 0.10203687, 0.0671858 , 0.06106969,
       0.0560188 , 0.04255811, 0.03820977, 0.03532811, 0.03271733,
       0.03023573, 0.02768732, 0.02645797, 0.02473476, 0.02288508,
       0.02097777, 0.01994106, 0.01705505, 0.01557411, 0.01381939,
       0.01303067, 0.01147583, 0.00943692, 0.00867916, 0.0064517 ,
       0.0049197 , 0.00448683, 0.00428824, 0.0038861 , 0.00211594])
```

first 5 principal component is responsible for 16.22% + 11.44% + 10.2% + 6.7% + 6.1% = 50.66% variance in the dataset.

Collectively we can say that only 50.66% percent of the classification information

contained in the feature set is captured by the first 5 principal components.

Other 25 principal components also have almost equal influence varying from 5%,4%,3%,2%,1% and less than 1%.

Let us first try with all components

In [109…

```
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[557  58]
 [ 33 522]]
Accuracy :  0.9222222222222223
```

It can be seen from the output that with all feature, the random forest algorithm is able to correctly predict 1096 out of 1170 instances, resulting in 93.67% accuracy.

PCA with 10 principal components

In [110…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=10)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [111…

```
# The PCA class contains explained_variance_ratio_ which returns the variance caused by each of the
# principal components.
explained_variance = pca.explained_variance_ratio_
explained_variance
```

Out[111…

```
array([0.16229804, 0.11443812, 0.10203687, 0.0671858 , 0.06106967,
       0.05601877, 0.04255792, 0.03820968, 0.03532691, 0.03271592])
```

In [112…

```
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[557  58]
 [ 33 522]]
Accuracy :  0.9222222222222223
```

Decrease in accuracy can be seen with 10 principal components

PCA with 5 principal components

In [116…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=5)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [117…

```
# The PCA class contains explained_variance_ratio_ which returns the variance caused by each of the
# principal components.
explained_variance = pca.explained_variance_ratio_
explained_variance
```

Out[117…

```
array([0.16229804, 0.11443809, 0.10203686, 0.06718553, 0.06106601])
```

In [118…

```
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[558  57]
 [ 35 520]]
Accuracy :  0.9213675213675213
```

Accuracy dropped with 5 principal components

PCA with 15 principal components

In [119…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=15)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [120…

```
# The PCA class contains explained_variance_ratio_ which returns the variance caused by each of the
# principal components.
explained_variance = pca.explained_variance_ratio_
explained_variance
```

Out[120…

```
array([0.16229804, 0.11443812, 0.10203687, 0.0671858 , 0.06106969,
       0.0560188 , 0.04255811, 0.03820977, 0.03532811, 0.03271732,
       0.03023573, 0.02768732, 0.02645794, 0.02473474, 0.02288507])
```

In [121…

```
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[556  59]
 [ 31 524]]
Accuracy :  0.9230769230769231
```

Accuracy still decreased with 15 principal components compared with all attributes together

In [122…

```
# Define a pipeline to search for the best combination of PCA truncation
# and classifier regularization.
pca = PCA()
# set the tolerance to a large value to make the example faster
classifier = LogisticRegression()
pipe = Pipeline(steps=[("pca", pca), ("classifier", classifier)])
```

In [123…

```
# Parameters of pipelines can be set using '__' separated parameter names:
param_grid = {
    "pca__n_components": [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
}
search = GridSearchCV(pipe, param_grid, n_jobs=-1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
search.fit(X_train, y_train)
print("Best parameter (CV score=%0.3f):" % search.best_score_)
print(search.best_params_)
```

```
Best parameter (CV score=0.916):
{'pca__n_components': 26}
```

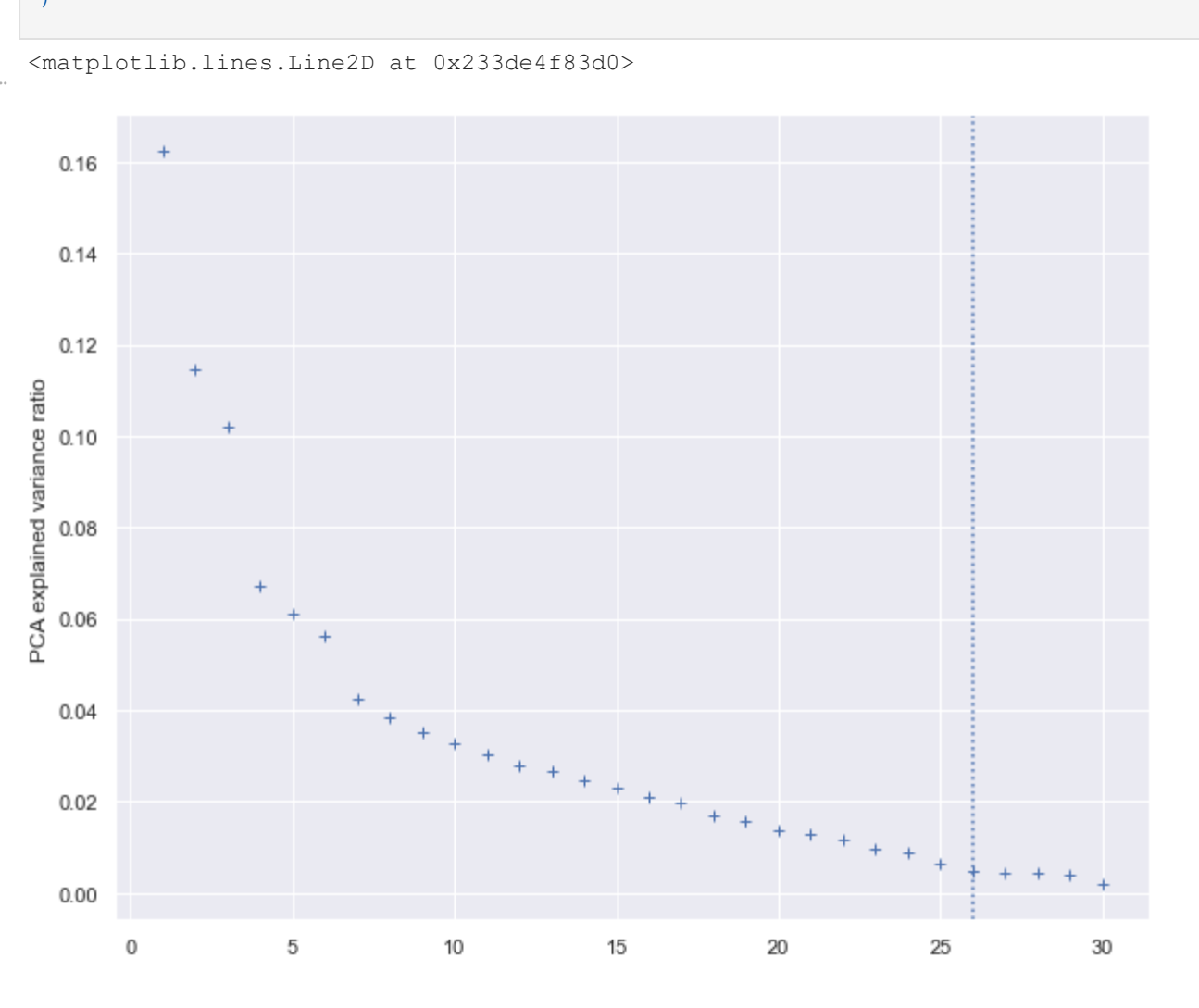
So if we use 26 principal components then the accuracy seems best

In [124…

```
pca.fit(X_train)
fig, (ax0) = plt.subplots(nrows=1, sharex=True, figsize=(10, 8))
ax0.plot(
    np.arange(1, pca.n_components_ + 1), pca.explained_variance_ratio_, "+", linewidth=2
)
ax0.set_ylabel("PCA explained variance ratio")

ax0.axvline(
    search.best_estimator_.named_steps["pca"].n_components,
    linestyle=":",
    label="n_components chosen",
)
```

Out[124…

```
<matplotlib.lines.Line2D at 0x233de4f83d0>
```



PCA with 26 principal components

In [125…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=26)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [126…

```
# The PCA class contains explained_variance_ratio_ which returns the variance caused by each of the
# principal components.
explained_variance = pca.explained_variance_ratio_
explained_variance
```

Out[126…

```
array([0.16229804, 0.11443812, 0.10203687, 0.0671858 , 0.06106969,
       0.0560188 , 0.04255811, 0.03820977, 0.03532811, 0.03271733,
       0.03023573, 0.02768732, 0.02645797, 0.02473476, 0.02288508,
       0.02097777, 0.01994106, 0.01705505, 0.01557411, 0.01381939,
       0.01303067, 0.01147583, 0.00943692, 0.00867916, 0.0064517 ,
       0.0049197 ])
```

In [127…

```
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[570  45]
 [ 27 528]]
Accuracy :  0.9384615384615385
```

Accuracy increased with 26 principal components compared with all components

In [128…

```
n_component=range(1,30)
n_component
```

Out[128…

```
range(1, 30)
```

In [129…

```
for n_component in range(1,31):

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
    pca = PCA(n_components=n_component)
    X_train = pca.fit_transform(X_train)
    X_test = pca.transform(X_test)
    # Training and Making Predictions
    classifier = LogisticRegression()
    classifier.fit(X_train, y_train)

    # Predicting the Test set results
    y_pred = classifier.predict(X_test)

    cm = confusion_matrix(y_test, y_pred)
    print(cm)
    print('Accuracy for pca no of component: ' + str(n_component) + " is ",metrics.accuracy_score(y_test, y_pre
```

```
[[487 128]
 [433 122]]
Accuracy for pca no of component: 1 is  0.5205128205128206
[[481 134]
 [322 233]]
Accuracy for pca no of component: 2 is  0.6102564102564103
[[465 150]
 [126 429]]
Accuracy for pca no of component: 3 is  0.764102564102564
[[489 126]
 [116 439]]
Accuracy for pca no of component: 4 is  0.7931623931623931
[[558  57]
 [ 35 520]]
Accuracy for pca no of component: 5 is  0.9213675213675213
[[555  60]
 [ 32 523]]
Accuracy for pca no of component: 6 is  0.9213675213675213
[[555  60]
 [ 32 523]]
Accuracy for pca no of component: 7 is  0.9213675213675213
[[555  60]
 [ 32 523]]
Accuracy for pca no of component: 8 is  0.9213675213675213
[[556  59]
 [ 33 522]]
Accuracy for pca no of component: 9 is  0.9213675213675213
[[556  59]
 [ 33 522]]
Accuracy for pca no of component: 10 is  0.9213675213675213
[[556  59]
 [ 33 522]]
Accuracy for pca no of component: 11 is  0.9213675213675213
[[557  58]
 [ 31 524]]
Accuracy for pca no of component: 12 is  0.923931623931624
[[557  58]
 [ 31 524]]
Accuracy for pca no of component: 13 is  0.923931623931624
[[556  59]
 [ 34 521]]
Accuracy for pca no of component: 14 is  0.9205128205128205
[[556  59]
 [ 31 524]]
Accuracy for pca no of component: 15 is  0.9230769230769231
[[561  54]
 [ 30 525]]
Accuracy for pca no of component: 16 is  0.9282051282051282
[[562  53]
 [ 25 530]]
Accuracy for pca no of component: 17 is  0.9333333333333333
[[569  46]
 [ 29 526]]
Accuracy for pca no of component: 18 is  0.9358974358974359
[[570  45]
 [ 28 527]]
Accuracy for pca no of component: 19 is  0.9376068376068376
[[569  46]
 [ 26 529]]
Accuracy for pca no of component: 20 is  0.9384615384615385
[[568  47]
 [ 26 529]]
Accuracy for pca no of component: 21 is  0.9376068376068376
[[567  48]
 [ 26 529]]
Accuracy for pca no of component: 22 is  0.9367521367521368
[[566  49]
 [ 28 527]]
Accuracy for pca no of component: 23 is  0.9341880341880342
[[567  48]
 [ 30 525]]
Accuracy for pca no of component: 24 is  0.9333333333333333
[[568  47]
 [ 30 525]]
Accuracy for pca no of component: 25 is  0.9341880341880342
[[570  45]
 [ 27 528]]
Accuracy for pca no of component: 26 is  0.9384615384615385
[[570  45]
 [ 28 527]]
Accuracy for pca no of component: 27 is  0.9376068376068376
[[569  46]
 [ 26 529]]
Accuracy for pca no of component: 28 is  0.9384615384615385
[[568  47]
 [ 27 528]]
Accuracy for pca no of component: 29 is  0.9367521367521368
[[568  47]
 [ 27 528]]
Accuracy for pca no of component: 30 is  0.9367521367521368
```

It seems if we keep no of components as 20 & 26 then the accuracy is 93.84%.

Still we would go for 26 components as prescribed by gridsearch as True positive is better (570) if we use 26 components.

Perform Logistic Regression

In [130…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=26)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[570  45]
 [ 27 528]]
Accuracy :  0.9384615384615385
```

Illustrate the diagnostic ability of this binary classifier by plotting the ROC curve.

What Are ROC Curves?

A useful tool when predicting the probability of a binary outcome is the Receiver Operating Characteristic curve or ROC curve.

It is a plot of the false positive rate (x-axis) versus the true positive rate (y-axis) for a number of different candidate threshold values between 0.0 and 1.0. Put another way, it plots the false alarm rate versus the hit rate.

The true positive rate is calculated as the number of true positives divided by the sum of the number of true positives and the number of false negatives. It describes how good the model is at predicting the positive class when the actual outcome is positive.

True Positive Rate = True Positives / (True Positives + False Negatives)

The true positive rate is also referred to as sensitivity.

Sensitivity = True Positives / (True Positives + False Negatives)

The false positive rate is calculated as the number of false positives divided by the sum of the number of false positives and the number of true negatives. It is also called the false alarm rate as it summarizes how often a positive class is predicted when the actual outcome is negative.

False Positive Rate = False Positives / (False Positives + True Negatives) The false positive rate is also referred to as the inverted specificity where specificity is the total number of true negatives divided by the sum of the number of true negatives and false positives. Specificity = True Negatives / (True Negatives + False Positives) False Positive Rate = 1 - Specificity

The ROC curve is a useful tool for a few reasons:

The curves of different models can be compared directly in general or for different thresholds. The area under the curve (AUC) can be used as a summary of the model skill. The shape of the curve contains a lot of information, including what we might care about most for a problem, the expected false positive rate, and the false negative rate.

To make this clear:

Smaller values on the x-axis of the plot indicate lower false positives and higher true negatives. Larger values on the y-axis of the plot indicate higher true positives and lower false negatives.

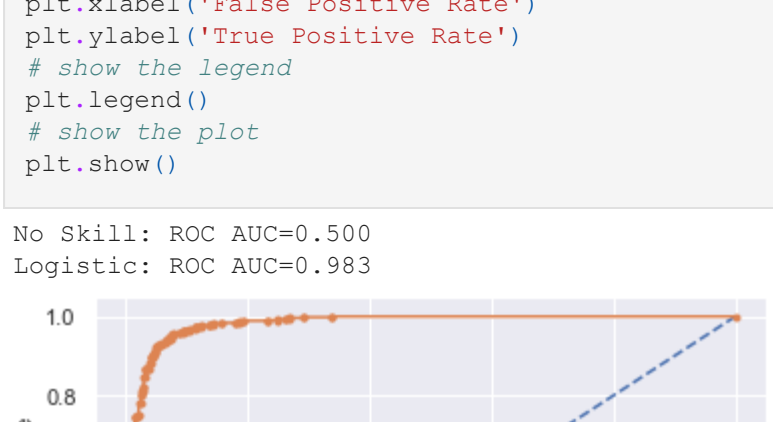
In [147…

```
from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
```

In [150…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=26)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
# Training and Making Predictions
classifier = LogisticRegression()
classifier.fit(X_train, y_train)


# generate a no skill prediction (majority class)
ns_probs = [0 for _ in range(len(y_test))]
# predict probabilities
lr_probs = classifier.predict_proba(X_test)

# keep probabilities for the positive outcome only
lr_probs = lr_probs[:, 1]

# calculate scores
ns_auc = roc_auc_score(y_test, ns_probs)
lr_auc = roc_auc_score(y_test, lr_probs)

# summarize scores
print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('Logistic: ROC AUC=%.3f' % (lr_auc))

# calculate roc curves
ns_fpr, ns_tpr, _ = roc_curve(y_test, ns_probs)
lr_fpr, lr_tpr, _ = roc_curve(y_test, lr_probs)

# plot the roc curve for the model
plt.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
plt.plot(lr_fpr, lr_tpr, marker='.', label='Logistic')

# axis labels
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
# show the legend
plt.legend()
# show the plot
plt.show()
```

```
No Skill: ROC AUC=0.500
Logistic: ROC AUC=0.983
```



As we can observe from the ROC curve the x-axis values are Smaller which indicate lower false positives and higher true negatives.

Also y-axis values are > .9 which shows higher true positives and lower false negatives.

Below are the list of parameters to do hyper parameter tuning for logistic regression

Solver

solver in ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga']

Regularization (penalty)

penalty in ['none', 'l1', 'l2', 'elasticnet'] Note: not all solvers support all regularization terms.

The C parameter controls the penality strength, which can also be effective.

C in [100, 10, 1.0, 0.1, 0.01]

In [132…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.01, random_state=17)
pca = PCA(n_components=26)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

In [136…

```
%%time
# define models and parameters
model = LogisticRegression()
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l2']
c_values = [100, 10, 1.0, 0.1, 0.01]
# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',error_score=(
grid_result = grid_search.fit(X_train, y_train)
# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.921704 using {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920265 (0.010267) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920265 (0.010267) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.920265 (0.010267) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
0.920956 (0.009818) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920956 (0.009818) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.920898 (0.009835) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.920841 (0.009962) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920841 (0.009962) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.920841 (0.009962) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.921704 (0.009400) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.921704 (0.009400) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.921589 (0.009560) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.918423 (0.010829) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.918423 (0.010829) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.918192 (0.010576) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
Wall time: 19.6 s
```

So 'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg' is providing best accuracy score of 92.17%

Adding l1 penalty as well

In [137…

```
%%time
# define models and parameters
model = LogisticRegression()
solvers = ['newton-cg', 'lbfgs', 'liblinear']
penalty = ['l1','l2']
c_values = [100, 10, 1.0, 0.1, 0.01]
# define grid search
grid = dict(solver=solvers,penalty=penalty,C=c_values)

cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

grid_search = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv, scoring='accuracy',error_score=(
grid_result = grid_search.fit(X_train, y_train)

# summarize results
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']

for mean, stdev, param in zip(means, stds, params):
    print("%f (%f) with: %r" % (mean, stdev, param))
```

```
Best: 0.921704 using {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 100, 'penalty': 'l1', 'solver': 'lbfgs'}
0.920495 (0.010281) with: {'C': 100, 'penalty': 'l1', 'solver': 'liblinear'}
0.920265 (0.010267) with: {'C': 100, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920265 (0.010267) with: {'C': 100, 'penalty': 'l2', 'solver': 'lbfgs'}
0.920265 (0.010267) with: {'C': 100, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 10, 'penalty': 'l1', 'solver': 'lbfgs'}
0.920783 (0.009848) with: {'C': 10, 'penalty': 'l1', 'solver': 'liblinear'}
0.920956 (0.009818) with: {'C': 10, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920956 (0.009818) with: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
0.920898 (0.009835) with: {'C': 10, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'lbfgs'}
0.920725 (0.010005) with: {'C': 1.0, 'penalty': 'l1', 'solver': 'liblinear'}
0.920841 (0.009962) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'newton-cg'}
0.920841 (0.009962) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'lbfgs'}
0.920841 (0.009962) with: {'C': 1.0, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'lbfgs'}
0.920380 (0.010509) with: {'C': 0.1, 'penalty': 'l1', 'solver': 'liblinear'}
0.921704 (0.009400) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg'}
0.921704 (0.009400) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
0.921589 (0.009560) with: {'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'newton-cg'}
0.000000 (0.000000) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'lbfgs'}
0.901900 (0.013232) with: {'C': 0.01, 'penalty': 'l1', 'solver': 'liblinear'}
0.918423 (0.010829) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'newton-cg'}
0.918423 (0.010829) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'lbfgs'}
0.918192 (0.010576) with: {'C': 0.01, 'penalty': 'l2', 'solver': 'liblinear'}
Wall time: 59.9 s
```

Still 'C': 0.1, 'penalty': 'l2', 'solver': 'newton-cg' is providing best accuracy score of 92.17%

In [138…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
pca = PCA(n_components=26)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
# Training and Making Predictions
classifier = LogisticRegression(C=0.1, penalty= 'l2', solver= 'newton-cg')
classifier.fit(X_train, y_train)

# Predicting the Test set results
y_pred = classifier.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
print(cm)
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```

```
[[570  45]
 [ 26 529]]
Accuracy :  0.9393162393162393
```

Validate the accuracy of data by the K-Fold cross-validation technique.

In [142…

```
# import libraries for model validation
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_validate
```

In [145…

```
kf= KFold(n_splits=11,shuffle=True,random_state=20)
```

In [146…

```
%%time
scoring = {'accuracy': 'accuracy'}
# instantiate the lin reg model
classifier = LogisticRegression(C=0.1, penalty= 'l2', solver= 'newton-cg')
scores= cross_validate(estimator=classifier, X=X_train, y=y_train, cv=kf, scoring=scoring, return_train_score=T

print('Score keys : \n', scores.keys())
print('Training accuracy : {}'.format(scores['train_accuracy'].mean()))
print('Testing accuracy : {}'.format(scores['test_accuracy'].mean()))
```

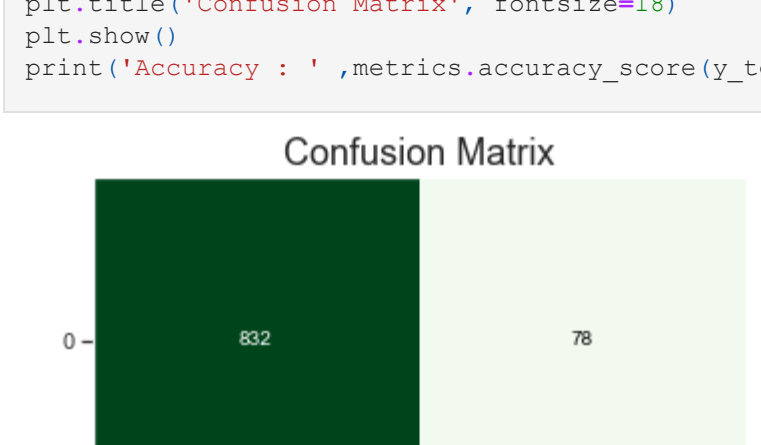
```
Score keys :
 dict_keys(['fit_time', 'score_time', 'test_accuracy', 'train_accuracy'])
Training accuracy : 0.9192776096010751
Testing accuracy : 0.9172905525846702
Wall time: 677 ms
```

The KFold cross validation result shows the training and testing scores are almost similar, no overfitting.

Final Logistic Regression Model

In [155…

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=7)
pca = PCA(n_components=26)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
# Training and Making Predictions
lr = LogisticRegression(C=0.1, penalty= 'l2', solver= 'newton-cg')
lr.fit(X_train, y_train)

# Predicting the Test set results
y_pred = lr.predict(X_test)

cm = confusion_matrix(y_test, y_pred)
from mlxtend.plotting import plot_confusion_matrix

fig, ax = plot_confusion_matrix(conf_mat=cm, figsize=(8, 6), cmap=plt.cm.Greens)
plt.xlabel('Predictions', fontsize=18)
plt.ylabel('Actuals', fontsize=18)
plt.title('Confusion Matrix', fontsize=18)
plt.show()
print('Accuracy : ' ,metrics.accuracy_score(y_test, y_pred))
```




```
Accuracy :  0.9253561253561253
```

In [157…

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

          -1       0.94      0.91      0.93       910
           1       0.91      0.94      0.92       845

    accuracy                           0.93      1755
   macro avg       0.93      0.93      0.93      1755
weighted avg       0.93      0.93      0.93      1755
```

In [ ]: