Project - Machine Learning

(2022-2023)

TEXAS McCombs
The University of Texas at Austin
McCombs School of Business

Course Name

Post Graduate Program in
Data Science and Business Analytics

Batch Id

(PGP-DSBA-June22C)

Submitted by

Jayant Singh
Email id: jayant101169@gmail.com

Contents

DATA DICTIONARY:

1. vote: Party choice: Conservative or Labour

2. age: in years

3. economic.cond.national: Assessment of current national economic conditions, 1 to 5.

4. economic.cond.household: Assessment of current household economic conditions, 1 to 5.

5. Blair: Assessment of the Labour leader, 1 to 5.

6. Hague: Assessment of the Conservative leader, 1 to 5.

7. Europe: an 11-point scale that measures respondents' attitudes toward European integration. High scores represent ‘Eurosceptic’ sentiment.

8. political.knowledge: Knowledge of parties' positions on European integration, 0 to 3.

9. gender: female or male.

Problem 1

You are hired by one of the leading news channels CNBE who wants to analyze recent elections. This survey was conducted on 1525 voters with 9 variables. You have to build a model, to predict which party a voter will vote for on the basis of the given information, to create an exit poll that will help in predicting overall win and seats covered by a particular party. 1.1 Read the dataset. Do the descriptive statistics and do the null value condition check. Write an inference on it.¶

1.1 Read the data and do exploratory data analysis. Describe the data briefly. (Check the Data types, shape, EDA, 5 point summary). Perform Univariate, Bivariate Analysis, Multivariate

```
df= pd.read_csv('Election_Data.csv')
```

```
df.head()
```

	Unnamed: 0	vote	age	economic.cond.national	economic.cond.household	Blair	Hague	Europe	political.knowledge	g
0	1	Labour	43	3	3	4	1	2	2	
1	2	Labour	36	4	4	4	4	5	2	
2	3	Labour	35	4	4	5	2	3	2	
3	4	Labour	24	4	2	2	1	4	0	
4	5	Labour	41	2	2	1	1	6	2	

Information from the dataset

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1525 entries, 0 to 1524
Data columns (total 10 columns):
 #   Column                   Non-Null Count  Dtype 
---  ------                   --------------  ----- 
 0   Unnamed: 0               1525 non-null   int64 
 1   vote                     1525 non-null   object
 2   age                      1525 non-null   int64 
 3   economic.cond.national   1525 non-null   int64 
 4   economic.cond.household  1525 non-null   int64 
 5   Blair                    1525 non-null   int64 
 6   Hague                    1525 non-null   int64 
 7   Europe                   1525 non-null   int64 
 8   political.knowledge      1525 non-null   int64 
 9   gender                   1525 non-null   object
dtypes: int64(8), object(2)
memory usage: 119.3+ KB
```

```
df.shape
```

```
(1525, 10)
```

```
df.drop("Unnamed: 0", inplace = True, axis =1)
```

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

```
vote                       0
age                        0
economic.cond.national     0
economic.cond.household    0
Blair                      0
Hague                      0
Europe                     0
political.knowledge        0
gender                     0
dtype: int64
```

Total no of duplicate values = 8

```
dups=df.duplicated()
print("Total no of duplicate values = %d" % (dups.sum()))
df[dups]
```

Total no of duplicate values = 8

	vote	age	economic.cond.national	economic.cond.household	Blair	Hague	Europe	political.knowledge	genc
67	Labour	35	4	4	5	2	3	2	m
626	Labour	39	3	4	4	2	5	2	m
870	Labour	38	2	4	2	2	4	3	m
983	Conservative	74	4	3	2	4	8	2	fem
1154	Conservative	53	3	4	2	2	6	0	fem
1236	Labour	36	3	3	2	2	6	2	fem
1244	Labour	29	4	4	4	2	2	2	fem
1438	Labour	40	4	3	4	2	2	2	m

Total no of duplicate values = 8

```
print('Before dropping the Duplicate Values',df.shape)
df.drop_duplicates(inplace=True)
print('After dropping the Duplicate Values',df.shape)
```

```
Before dropping the Duplicate Values (1525, 9)
After dropping the Duplicate Values (1517, 9)
```

```
df.vote.value_counts()
```

```
Labour          1057
Conservative     460
Name: vote, dtype: int64
```

```
for feature in df.columns:
    if df[feature].dtype=='object':
            print(feature.upper() ," ",df[feature].nunique())
            print(df[feature].value_counts().sort_values())
```

```
VOTE   2
Conservative     460
Labour          1057
Name: vote, dtype: int64
GENDER   2
male      709
female    808
Name: gender, dtype: int64
```

```
df = df.rename(columns = {'political.knowledge': 'political_knowledge','economic.cond.national':'economi
```

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

	vote	age	economic_cond_national	economic_cond_household	Blair	Hague	Europ
count	1517.000000	1517.000000	1517.000000	1517.000000	1517.000000	1517.000000	1517.00000
mean	0.696770	54.241266	3.245221	3.137772	3.335531	2.749506	6.74027
std	0.459805	15.701741	0.881792	0.931069	1.174772	1.232479	3.29904
min	0.000000	24.000000	1.000000	1.000000	1.000000	1.000000	1.00000
25%	0.000000	41.000000	3.000000	3.000000	2.000000	2.000000	4.00000
50%	1.000000	53.000000	3.000000	3.000000	4.000000	2.000000	6.00000
75%	1.000000	67.000000	4.000000	4.000000	4.000000	4.000000	10.00000
max	1.000000	93.000000	5.000000	5.000000	5.000000	5.000000	11.00000

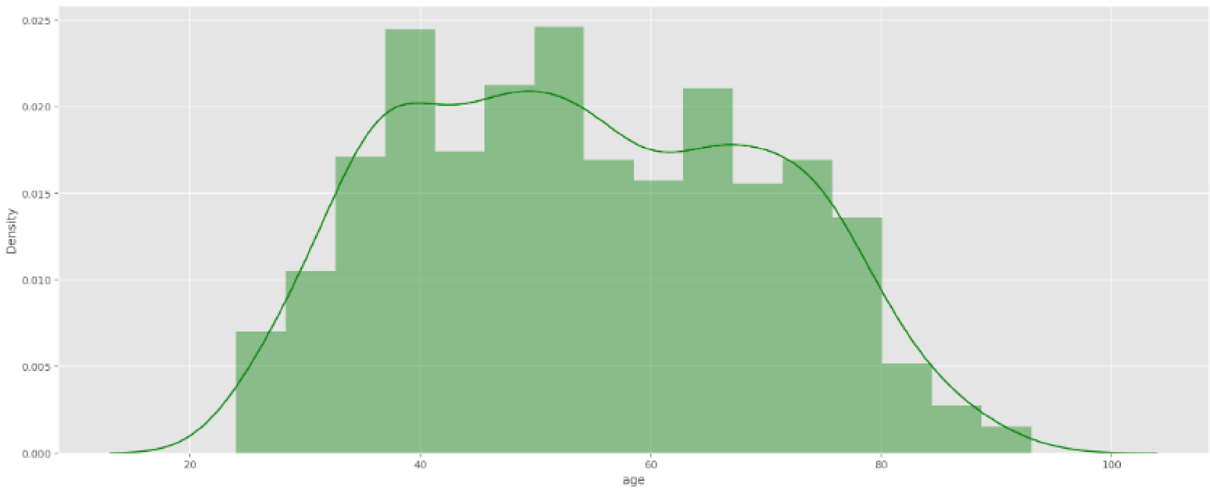
```
df.skew()
```

```
vote                      -0.857014
age                        0.139800
economic_cond_national    -0.238474
economic_cond_household   -0.144148
Blair                     -0.539514
Hague                      0.146191
Europe                    -0.141891
political_knowledge       -0.422928
gender                    -0.130929
dtype: float64
```

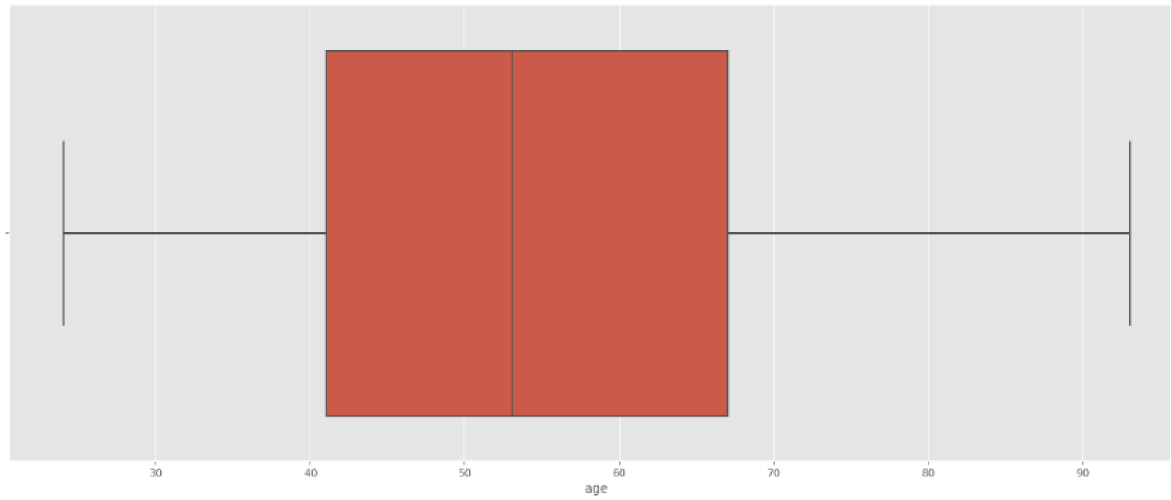
1.2 Perform Univariate and Bivariate Analysis. Do exploratory data analysis. Check for Outliers.

```
univariateAnalysis_numeric('age',20)
```

```
Description of age
------------------------------------------------

count    1517.000000
mean       54.241266
std        15.701741
min        24.000000
25%        41.000000
50%        53.000000
75%        67.000000
max        93.000000
Name: age, dtype: float64 Distribution of age
```

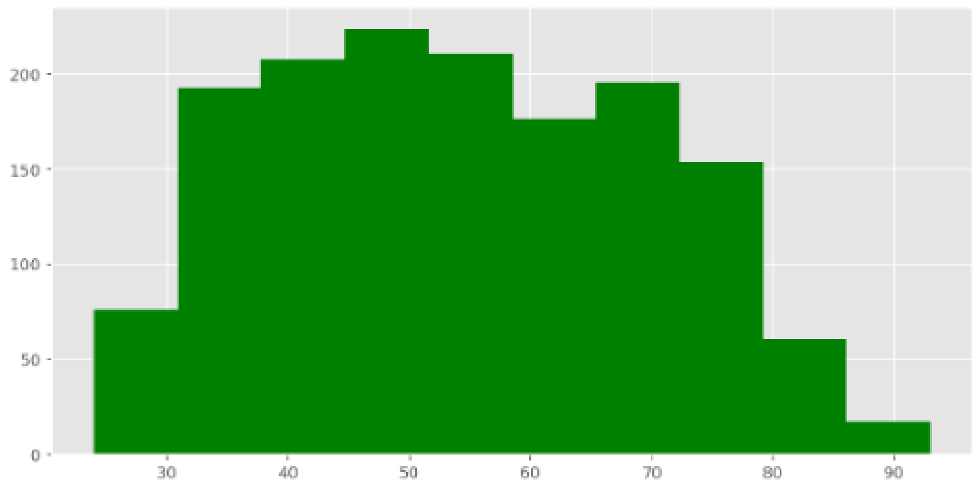



```
Skewness of age
0.1396615989084527 ----------------------------------------------------------------------------

 BoxPlot of age
 ----------------------------------------------------------------------------
```



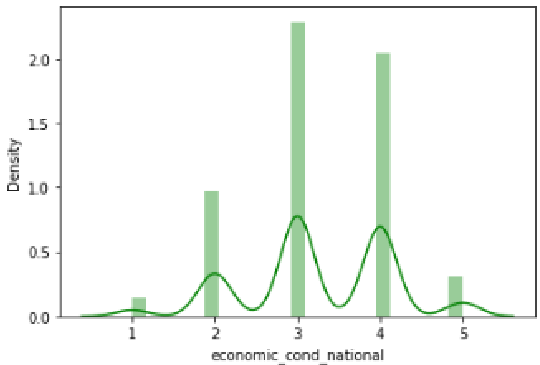

```
 Histogram of age
 ----------------------------------------------------------------------------
```

Histogram of age



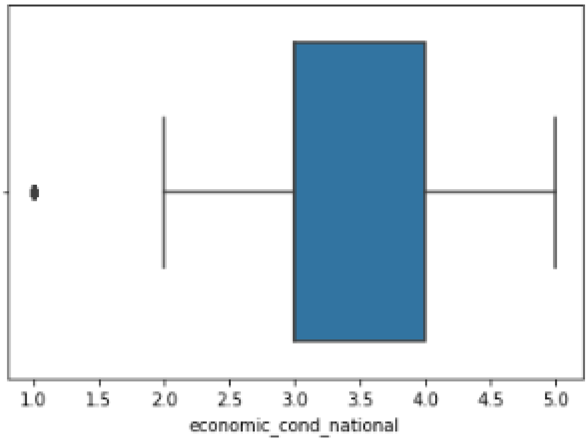

```
univariateAnalysis_numeric('economic_cond_national',20)
```

```
Description of economic_cond_national
----------------------------------------------------------------------------

count    1517.000000
mean        3.245221
std         0.881792
min         1.000000
25%         3.000000
50%         3.000000
75%         4.000000
max         5.000000
Name: economic_cond_national, dtype: float64 Distribution of economic_cond_national
----------------------------------------------------------------------------
```



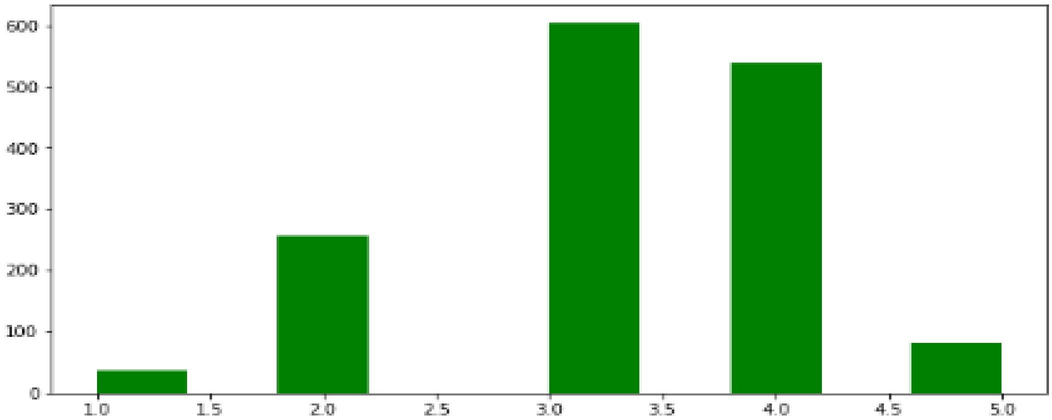

```
Skewness of economic_cond_national
-0.23823834819079348 ----------------------------------------------------------------------------
```

```
BoxPlot of economic_cond_national
----------------------------------------------------
```



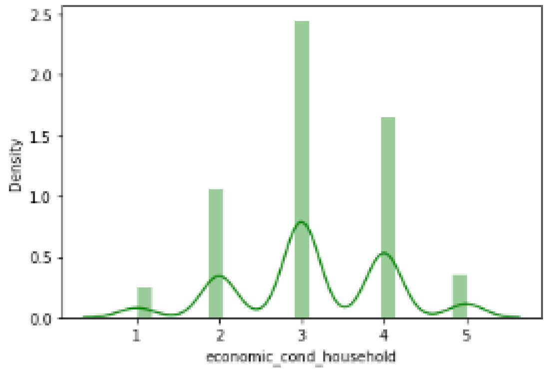

```
Histogram of economic_cond_national
```

```
Histogram of economic_cond_national
----------------------------------------------------------------------------
```

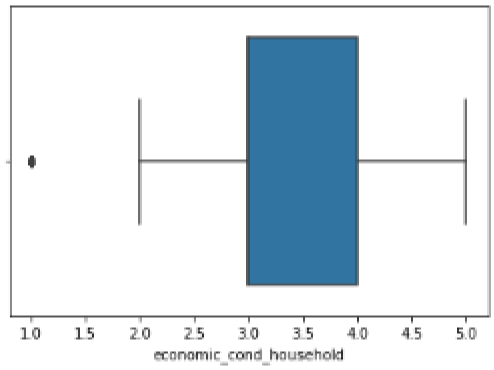


[19]:
```
univariateAnalysis_numeric('economic_cond_household',20)
```

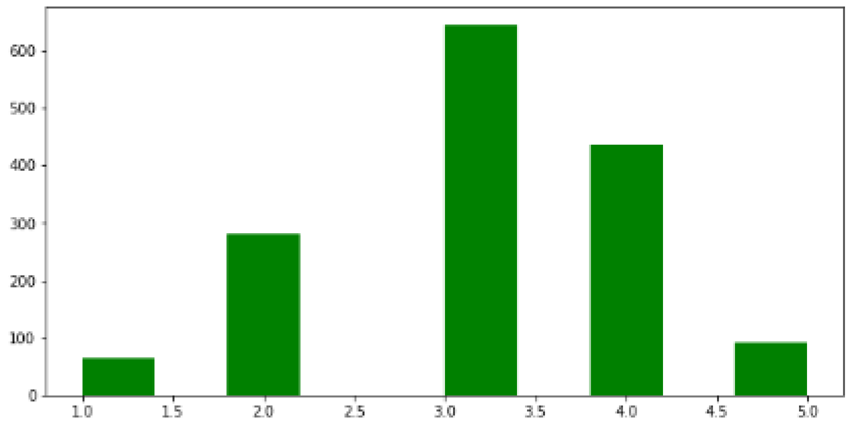
```
Description of economic_cond_household
----------------------------------------------------------------------------

count    1517.000000
mean        3.137772
std         0.931069
min         1.000000
25%         3.000000
50%         3.000000
75%         4.000000
max         5.000000
Name: economic_cond_household, dtype: float64 Distribution of economic_cond_household
----------------------------------------------------------------------------
```




```
Skewness of economic_cond_household
-0.144005097351352 ---------------------------------------------

BoxPlot of economic_cond_household
-------------------------------------------------------------
```

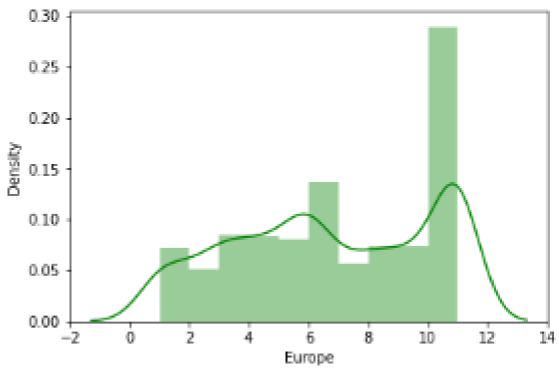



```
Histogram of economic_cond_household
```

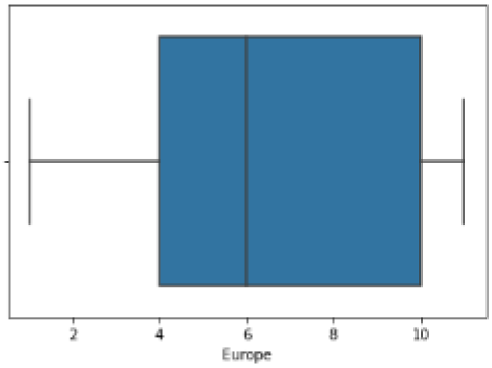


```
univariateAnalysis_numeric('Europe',20)
```

```
Description of Europe
----------------------------------------------------------------------------

count    1517.000000
mean        6.740277
std         3.299043
min         1.000000
25%         4.000000
50%         6.000000
75%        10.000000
max        11.000000
Name: Europe, dtype: float64 Distribution of Europe
----------------------------------------------------------------------------
```

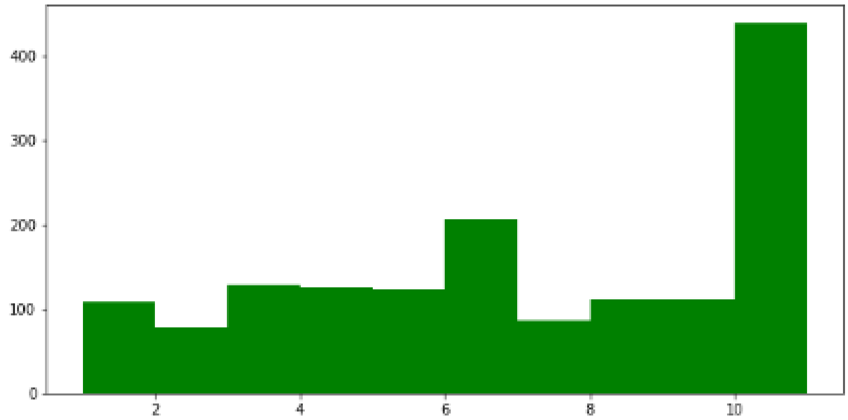



```
Skewness of Europe
-0.1417506103835579 ----------------------------------------------------------------------------


   BoxPlot of Europe
   ----------------------------------------------------------------------------
```



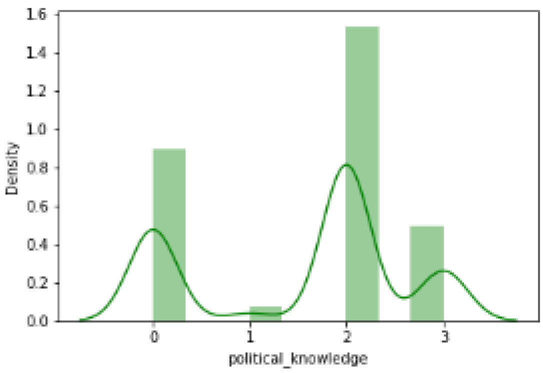

```
   Histogram of Europe
   ----------------------------------------------------------------------------
```

```
Histogram of Europe
----------------------------------------------------------------------------
```



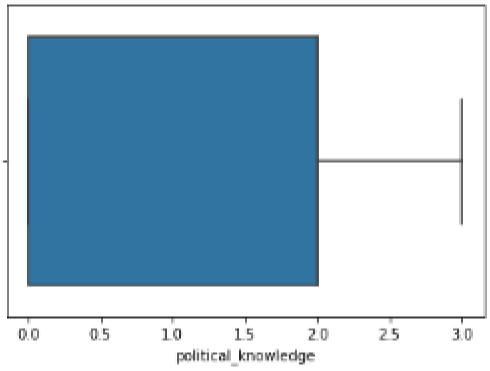

```
univariateAnalysis_numeric('political_knowledge',20)
```

```
Description of political_knowledge
----------------------------------------------------------------------------

count    1517.000000
mean        1.540541
std         1.084417
min         0.000000
25%         0.000000
50%         2.000000
75%         2.000000
max         3.000000
Name: political_knowledge, dtype: float64 Distribution of political_knowledge
----------------------------------------------------------------------------
```

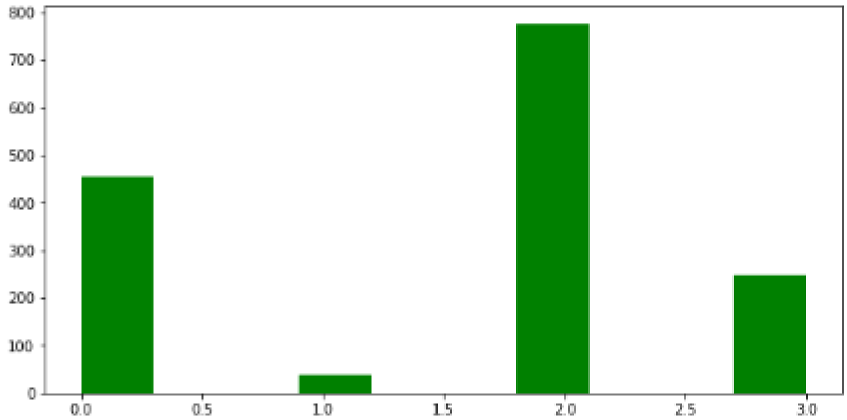



```
Skewness of political_knowledge
-0.42250931746800596 ------------------------------------------------------------------
```

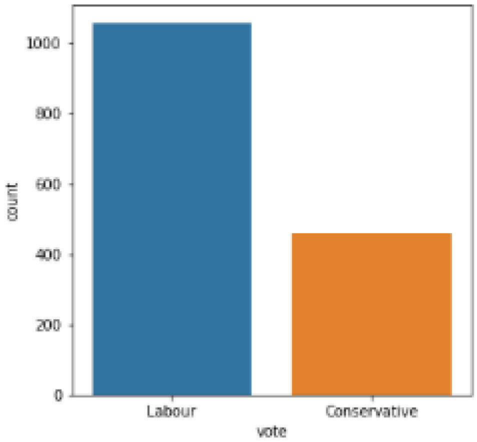
```
BoxPlot of political_knowledge
----------------------------------------------------------------------------
```



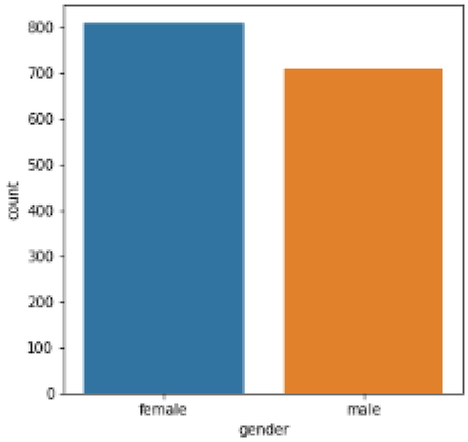

```
Histogram of political_knowledge
----------------------------------------------------------------------------
```




```
plt.figure(figsize=(5,5))
sns.countplot(df["vote"])
plt.show()
```

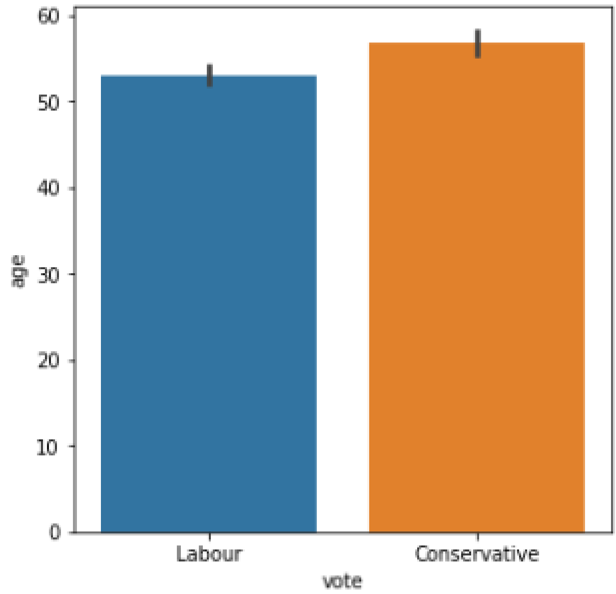


```
plt.figure(figsize=(5,5))
sns.countplot(df["gender"])
plt.show()
```



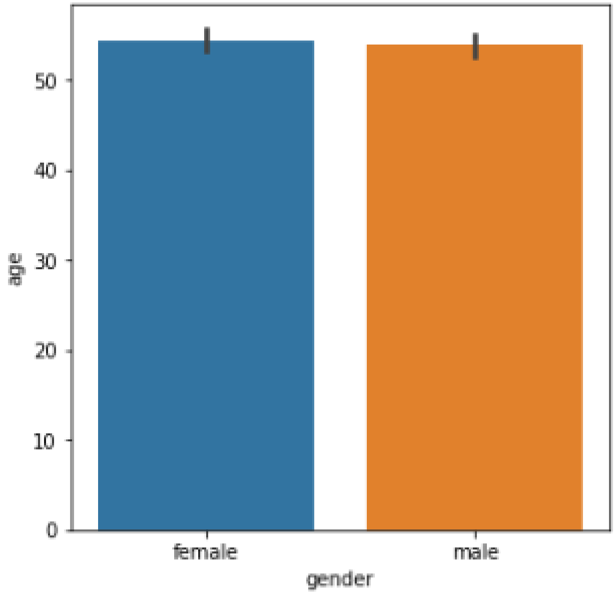

```
plt.figure(figsize=(5,5))
sns.barplot(data = df, x='vote',y='age')
```

```
<AxesSubplot:xlabel='vote', ylabel='age'>
```



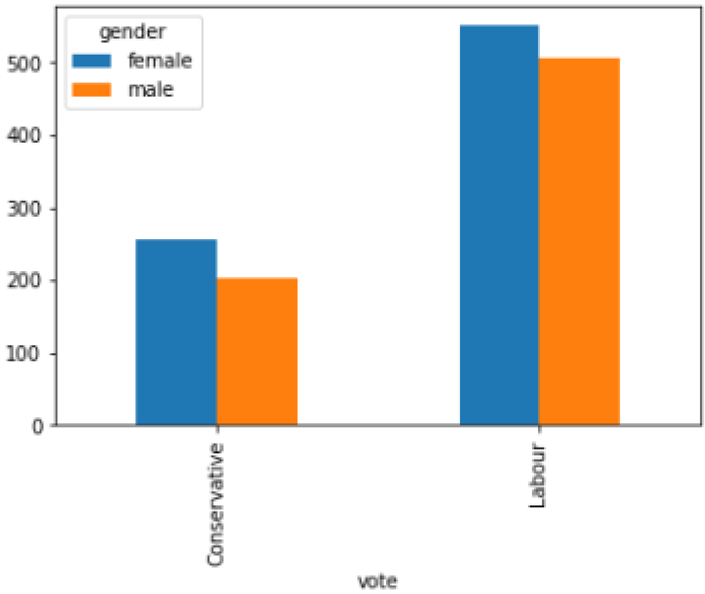
```
plt.figure(figsize=(5,5))
sns.barplot(data = df, x='gender',y='age')
```

```
<AxesSubplot:xlabel='gender', ylabel='age'>
```




```
pd.crosstab(df.vote,df.gender).plot(kind='bar')
```

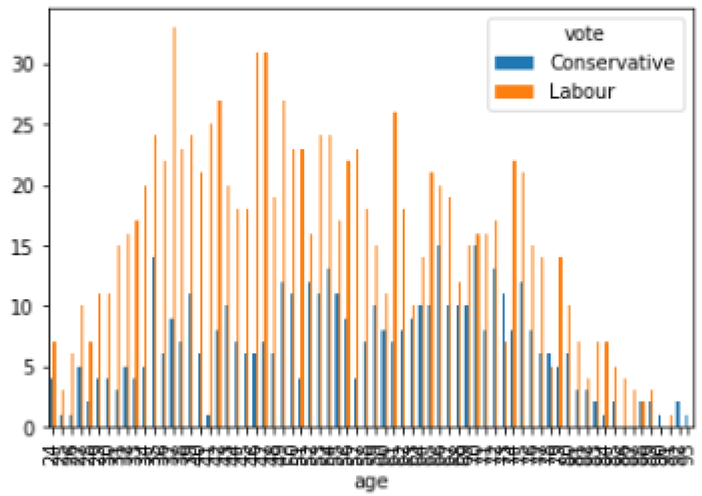
```
]: <AxesSubplot:xlabel='vote'>

<Figure size 7200x360 with 0 Axes>
```



```python
plt.figure(figsize=(1200,1200))
pd.crosstab(df.age,df.vote).plot(kind='bar')
```

```
<AxesSubplot:xlabel='age'>
```

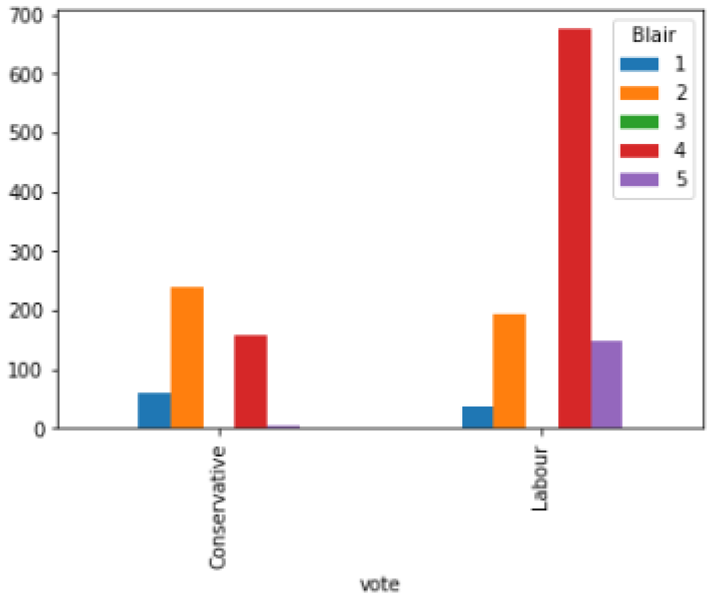
```
<Figure size 86400x86400 with 0 Axes>
```




```python
plt.figure(figsize=(100,5))
pd.crosstab(df.vote,df.Blair).plot(kind='bar')
```

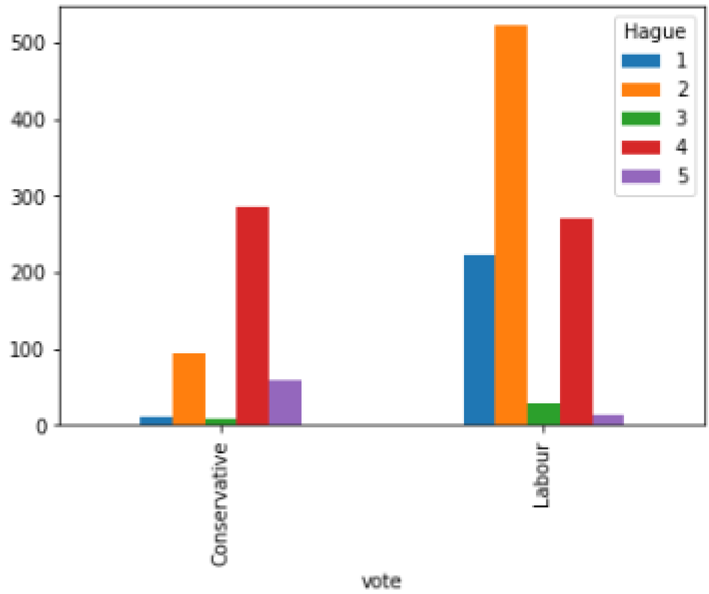
```
<AxesSubplot:xlabel='vote'>
```

```
<Figure size 7200x360 with 0 Axes>
```



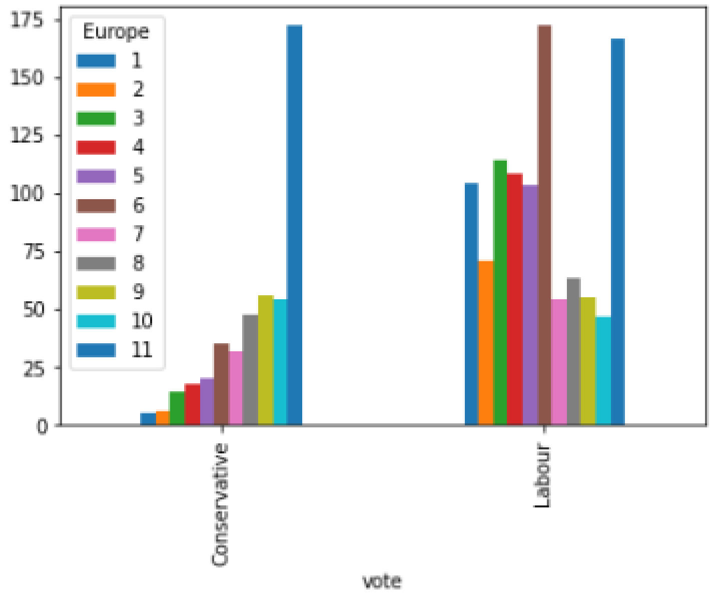
```
plt.figure(figsize=(100,10))
pd.crosstab(df.vote,df.Hague).plot(kind='bar')
```

```
<AxesSubplot:xlabel='vote'>

<Figure size 7200x720 with 0 Axes>
```



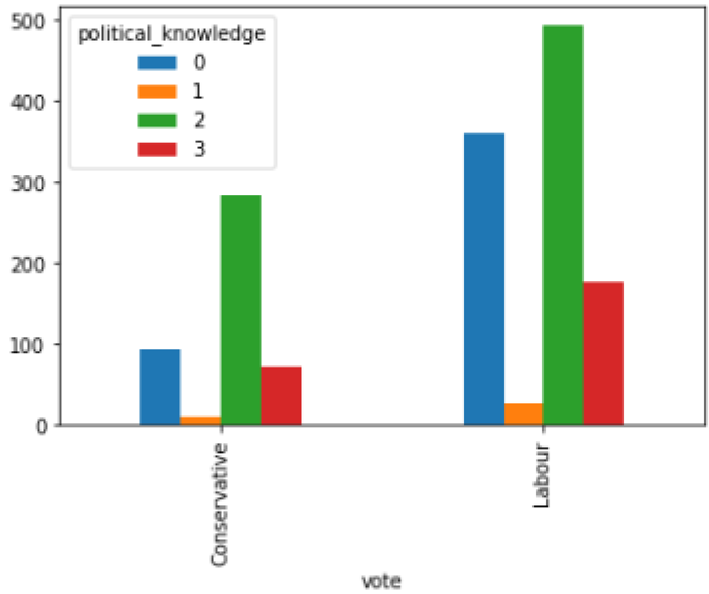

```
plt.figure(figsize=(100,10))
pd.crosstab(df.vote,df.Europe).plot(kind='bar')
```

```
<AxesSubplot:xlabel='vote'>

<Figure size 7200x720 with 0 Axes>
```



```
plt.figure(figsize=(100,10))
pd.crosstab(df.vote,df.political_knowledge).plot(kind='bar
```

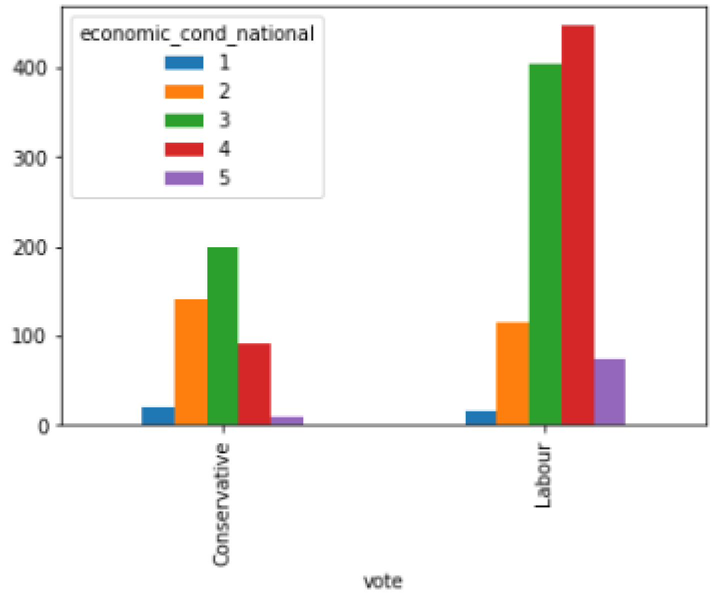
```
<AxesSubplot:xlabel='vote'>

<Figure size 7200x720 with 0 Axes>
```



```
plt.figure(figsize=(100,10))
pd.crosstab(df.vote,df.economic_cond_national).plot(kind='bar')
```

<AxesSubplot:xlabel='vote'>

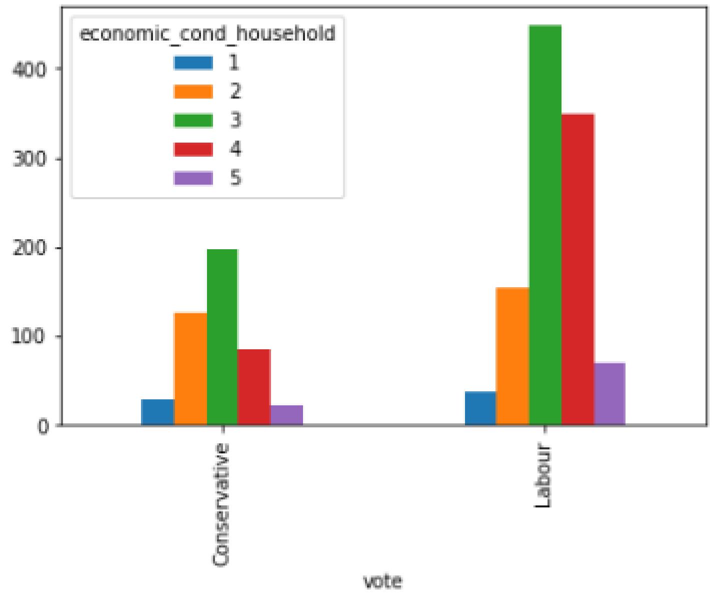
<Figure size 7200x720 with 0 Axes>



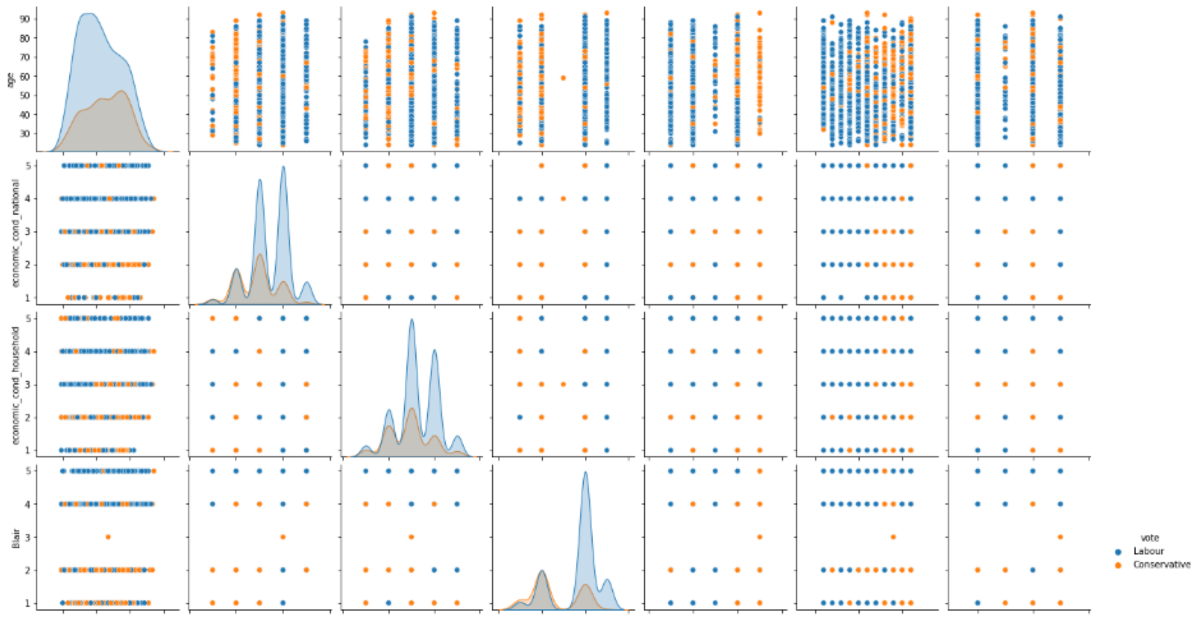

```
plt.figure(figsize=(100,10))
pd.crosstab(df.vote,df.economic_cond_household).plot(kind='bar')
```

<AxesSubplot:xlabel='vote'>

<Figure size 7200x720 with 0 Axes>

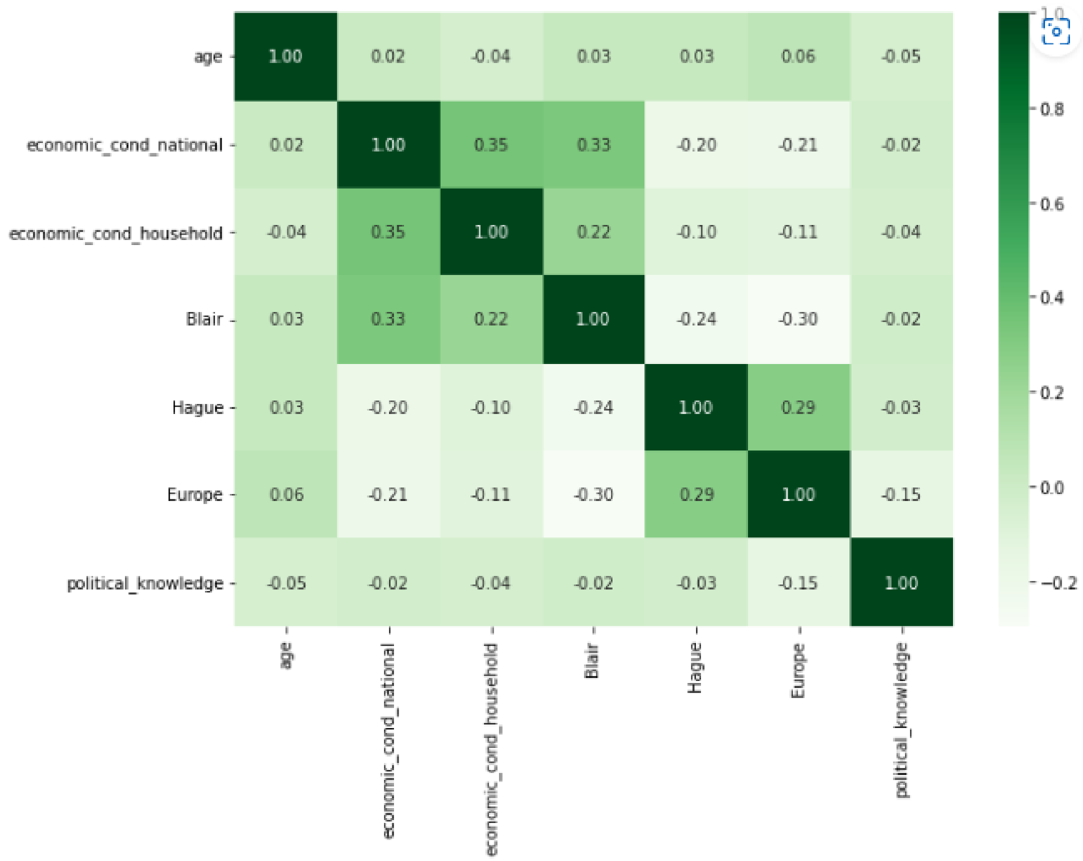


```
<seaborn.axisgrid.PairGrid at 0x1fd3e050100>
```




```
df.corr()
```

	age	economic_cond_national	economic_cond_household	Blair	Hague	Europe	political_knowledge
age	1.000000	0.018687	-0.038868	0.032084	0.031144	0.064562	-0.046598
economic_cond_national	0.018687	1.000000	0.347687	0.326141	-0.200790	-0.209150	-0.023510
economic_cond_household	-0.038868	0.347687	1.000000	0.215822	-0.100392	-0.112897	-0.038528
Blair	0.032084	0.326141	0.215822	1.000000	-0.243508	-0.295944	-0.021299
Hague	0.031144	-0.200790	-0.100392	-0.243508	1.000000	0.285738	-0.029906
Europe	0.064562	-0.209150	-0.112897	-0.295944	0.285738	1.000000	-0.151197
political_knowledge	-0.046598	-0.023510	-0.038528	-0.021299	-0.029906	-0.151197	1.000000



```
df.head(10)
```

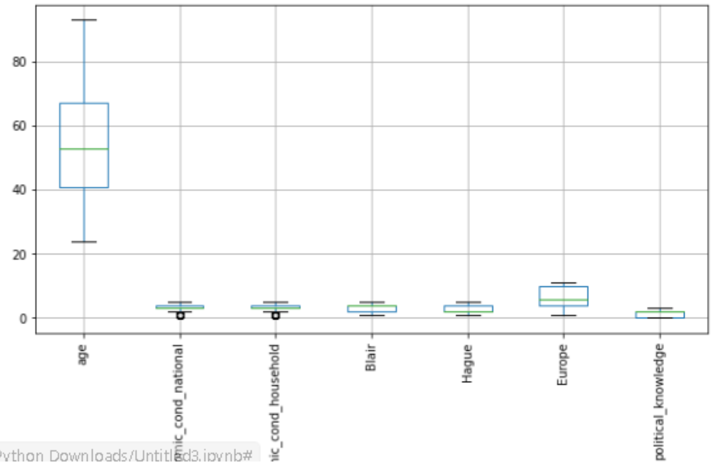
	vote	age	economic_cond_national	economic_cond_household	Blair	Hague	Europe	political_knowledge	gender
0	Labour	43	3	3	4	1	2	2	female
1	Labour	36	4	4	4	4	5	2	male
2	Labour	35	4	4	5	2	3	2	male
3	Labour	24	4	2	2	1	4	0	female
4	Labour	41	2	2	1	1	6	2	male
5	Labour	47	3	4	4	4	4	2	male
6	Labour	57	2	2	4	4	11	2	male
7	Labour	77	3	4	4	1	1	0	male
8	Labour	39	3	3	4	4	11	0	female
9	Labour	70	3	2	5	1	11	2	male

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1517 entries, 0 to 1524
Data columns (total 9 columns):
 #   Column                   Non-Null Count  Dtype 
---  ------                   --------------  ----- 
 0   vote                     1517 non-null   object
 1   age                      1517 non-null   int64 
 2   economic_cond_national   1517 non-null   int64 
 3   economic_cond_household  1517 non-null   int64 
 4   Blair                    1517 non-null   int64 
 5   Hague                    1517 non-null   int64 
 6   Europe                   1517 non-null   int64 
 7   political_knowledge      1517 non-null   int64 
 8   gender                   1517 non-null   object
dtypes: int64(7), object(2)
memory usage: 150.8+ KB
```

```
outlier = ['vote','age','economic_cond_national','economic_cond_household','Blair','Hague','Europe','political_knowledge','gender']
```

```
df[outlier].boxplot(figsize=(10,5))
plt.xticks(rotation=90)
plt.show()
```



1.3 Encode the data (having string values) for Modelling. Is Scaling necessary here or not? Data Split: Split the data into train and test (70:30).

```
X.head()
```

	age	economic_cond_national	economic_cond_household	Blair	Hague	Europe	political_knowledge	gender
0	-0.716161	-0.278185	-0.148020	0.565802	-1.419969	-1.437338	0.423832	0.936736
1	-1.162118	0.856242	0.926367	0.565802	1.014951	-0.527684	0.423832	-1.067536
2	-1.225827	0.856242	0.926367	1.417312	-0.608329	-1.134120	0.423832	-1.067536
3	-1.926617	0.856242	-1.222408	-1.137217	-1.419969	-0.830902	-1.421084	0.936736
4	-0.843577	-1.412613	-1.222408	-1.988727	-1.419969	-0.224465	0.423832	-1.067536

```
y.head()
```

```
0    1
1    1
2    1
3    1
4    1
Name: vote, dtype: int64
```

```
y_train.value_counts(1)
```

```
1    0.71065
0    0.28935
Name: vote, dtype: float64
```

```
y_test.value_counts(1)
```

```
1    0.664474
0    0.335526
Name: vote, dtype: float64
```

1.4 Apply Logistic Regression and LDA (linear discriminant analysis).

```
LogR_base_model = LogisticRegression(C=1000.0, solver = 'newton-cg')
LogR_base_model.fit(X_train, y_train)
```

```
LogisticRegression(C=1000.0, solver='newton-cg')
```

**In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.**

```
LogisticRegression(C=1000.0, solver='newton-cg')
```

```
LogisticRegression(C=1000.0, solver='newton-cg')
```

**In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.**

```
ytrain_predict_LogR_base = LogR_base_model.predict(X_train)
ytest_predict_LogR_base = LogR_base_model.predict(X_test)
```

```
ytest_predict_prob_LogR_Base=LogR_base_model.predict_proba(X_test)
pd.DataFrame(ytest_predict_prob_LogR_Base).head()
```

	0	1
0	0.424283	0.575717
1	0.148428	0.851572
2	0.007187	0.992813
3	0.836347	0.163653
4	0.068408	0.931592

```
from sklearn.metrics import roc_auc_score,roc_curve,classification_report,confusion_matrix,plot_confusion_matrix
confusion_matrix(y_train, ytrain_predict_LogR_base)
```

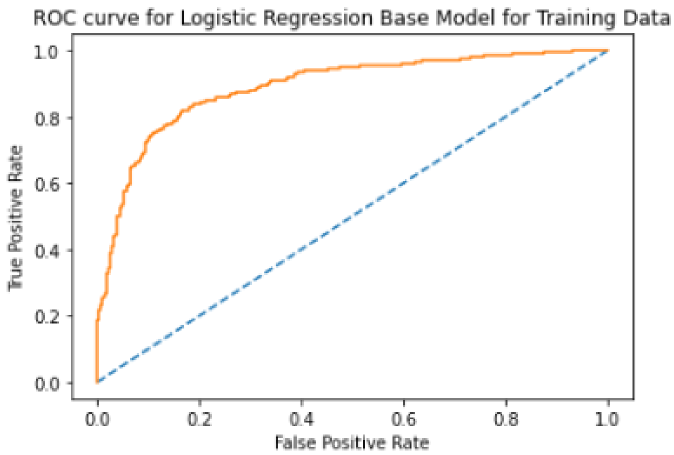
```
array([[196, 111],
       [ 68, 686]], dtype=int64)
```

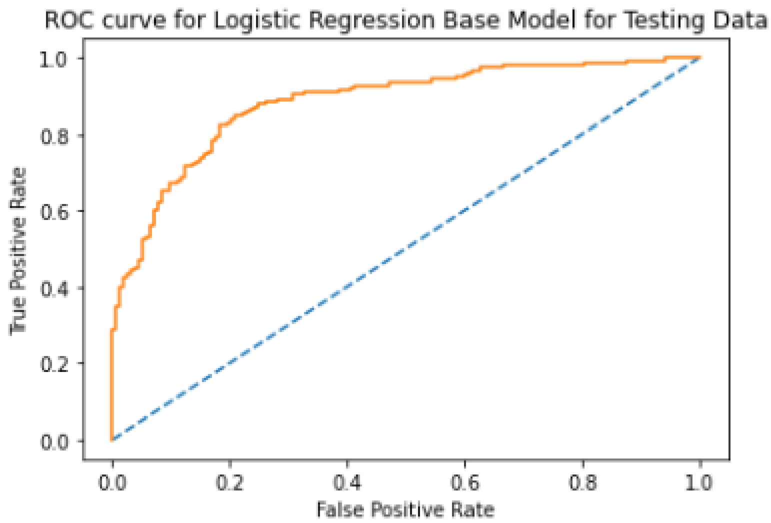
```
print(classification_report(y_train, ytrain_predict_LogR_base))
```

```
              precision    recall  f1-score   support

           0       0.74      0.64      0.69       307
           1       0.86      0.91      0.88       754

    accuracy                           0.83      1061
   macro avg       0.80      0.77      0.79      1061
weighted avg       0.83      0.83      0.83      1061
```

```
confusion_matrix(y_test, ytest_predict_LogR_base)
```

```
array([[113,  40],
       [ 35, 268]], dtype=int64)
```




ROC curve for Logistic Regression Base Model for Testing Data
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate

Regularised logistic regression

```
model_reg_LogR = LogisticRegression(solver='lbfgs',max_iter=10000,penalty='none',verbose=True,n_jobs=2,C=1.0)
model_reg_LogR.fit(X_train, y_train)
```

```
[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=2)]: Done   1 out of   1 | elapsed:    4.6s finished
```

```
LogisticRegression(max_iter=10000, n_jobs=2, penalty='none', verbose=True)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
ytrain_predict_LogR_reg = model_reg_LogR.predict(X_train)
ytest_predict_LogR_reg = model_reg_LogR.predict(X_test)
```

```
ytest_predict_prob_LogR_reg=model_reg_LogR.predict_proba(X_test)
pd.DataFrame(ytest_predict_prob_LogR_reg).head()
```

	0	1
0	0.424283	0.575717
1	0.148427	0.851573
2	0.007187	0.992813
3	0.836350	0.163650
4	0.068407	0.931593

```
confusion_matrix(y_train, ytrain_predict_LogR_reg)
```

```
array([[196, 111],
       [ 68, 686]], dtype=int64)
```

```
confusion_matrix(y_test, ytest_predict_LogR_reg)
```

```
array([[113,  40],
       [ 35, 268]], dtype=int64)
```

```
print(classification_report(y_train, ytrain_predict_LogR_reg))
```

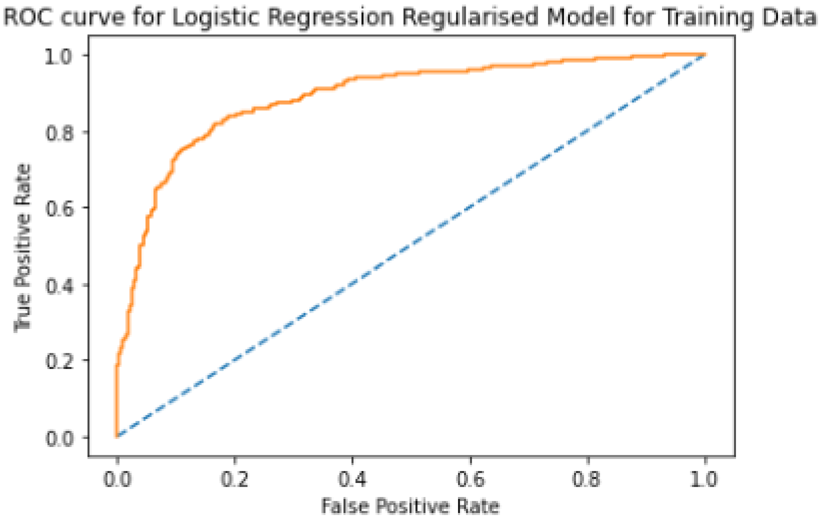
```
              precision    recall  f1-score   support

           0       0.74      0.64      0.69       307
           1       0.86      0.91      0.88       754

    accuracy                           0.83      1061
   macro avg       0.80      0.77      0.79      1061
weighted avg       0.83      0.83      0.83      1061
```

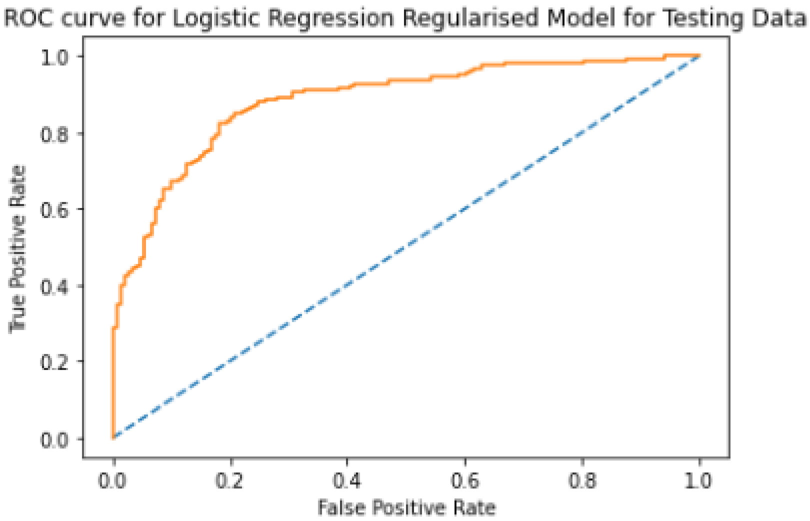
```
print(classification_report(y_train, ytrain_predict_LogR_reg))
```

```
              precision    recall  f1-score   support

           0       0.74      0.64      0.69       307
           1       0.86      0.91      0.88       754

    accuracy                           0.83      1061
   macro avg       0.80      0.77      0.79      1061
weighted avg       0.83      0.83      0.83      1061
```

```
print(classification_report(y_test, ytest_predict_LogR_reg))
```

```
              precision    recall  f1-score   support

           0       0.76      0.74      0.75       153
           1       0.87      0.88      0.88       303

    accuracy                           0.84       456
   macro avg       0.82      0.81      0.81       456
weighted avg       0.83      0.84      0.83       456
```





Applying Grid Seacrh CV on Logistic Regression

```
from sklearn.model_selection import GridSearchCV

grid={'penalty':['l1','l2','none','elasticnet'],
      'solver':['lbfgs', 'liblinear','newton-cg'],
      'tol':[0.0001,0.000001,0.001]     }
```

```
model_LogR = LogisticRegression(max_iter=100000,n_jobs=5, C=1.0)
```

```
grid_search = GridSearchCV(estimator = model_LogR, param_grid = grid, cv = 5,n_jobs=-1,scoring='f1')
```

```
grid_search.fit(X_train, y_train)
```

```
GridSearchCV(cv=5, estimator=LogisticRegression(max_iter=100000, n_jobs=5),
             n_jobs=-1,
             param_grid={'penalty': ['l1', 'l2', 'none', 'elasticnet'],
                         'solver': ['lbfgs', 'liblinear', 'newton-cg'],
                         'tol': [0.0001, 1e-06, 0.001]},
             scoring='f1')
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
print(grid_search.best_params_,'\n')
print(grid_search.best_estimator_)
```

```
{'penalty': 'l2', 'solver': 'lbfgs', 'tol': 0.0001}

LogisticRegression(max_iter=100000, n_jobs=5)
```

```
best_model_LogR = grid_search.best_estimator_
```

```
ytrain_predict_best_LogR = best_model_LogR.predict(X_train)
ytest_predict_best_LogR = best_model_LogR.predict(X_test)
```

```
ytest_predict_prob_best_LogR=best_model_LogR.predict_proba(X_test)
pd.DataFrame(ytest_predict_prob_best_LogR).head()
```

	0	1
0	0.423790	0.576210
1	0.150104	0.849896
2	0.007470	0.992530
3	0.833130	0.166870
4	0.069756	0.930244

```
confusion_matrix(y_train,ytrain_predict_best_LogR)
```

```
array([[196, 111],
       [ 68, 686]], dtype=int64)
```

```
print("Classification Report on Training Data for Logistic Regression \n\n",classification_report(y_train, ytrain_predict_best_LogR),'\
```

```
Classification Report on Training Data for Logistic Regression 

              precision    recall  f1-score   support

           0       0.74      0.64      0.69       307
           1       0.86      0.91      0.88       754

    accuracy                           0.83      1061
   macro avg       0.80      0.77      0.79      1061
weighted avg       0.83      0.83      0.83      1061
```

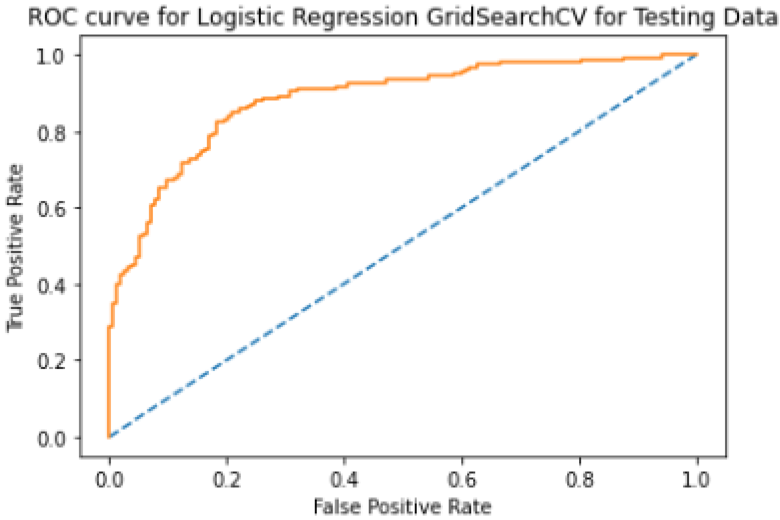
```
confusion_matrix(y_test,ytest_predict_best_LogR)
```

```
array([[111,  42],
       [ 35, 268]], dtype=int64)
```

```
print("Classification Report on Testing Data for Logistic Regression\n \n",classification_report(y_test, ytest_predict_best_LogR),'\n')
```

```
Classification Report on Testing Data for Logistic Regression

              precision    recall  f1-score   support

           0       0.76      0.73      0.74       153
           1       0.86      0.88      0.87       303

    accuracy                           0.83       456
   macro avg       0.81      0.80      0.81       456
weighted avg       0.83      0.83      0.83       456
```


ROC curve for Logistic Regression GridSearchCV for Training Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


ROC curve for Logistic Regression GridSearchCV for Testing Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

Linear Discriminant Analysis

```
clf = LinearDiscriminantAnalysis()
model_LDA=clf.fit(X_train,y_train)
```

```
pred_class_train_LDA = model_LDA.predict(X_train)
pred_class_test_LDA = model_LDA.predict(X_test)
```

```
pred_prob_train_LDA = model_LDA.predict_proba(X_train)
pred_prob_test_LDA = model_LDA.predict_proba(X_test)
```

```
train_acc_LDA = model_LDA.score(X_train,y_train)
train_acc_LDA
```

```
0.8341187558906692
```

```
print('Classification Report on Training Data for LDA\n\n',metrics.classification_report(y_train,pred_class_train_LDA),'\n')
```

```
Classification Report on Training Data for LDA

              precision    recall  f1-score   support

           0       0.74      0.65      0.69       307
           1       0.86      0.91      0.89       754

    accuracy                           0.83      1061
   macro avg       0.80      0.78      0.79      1061
weighted avg       0.83      0.83      0.83      1061
```

```
confusion_matrix(y_train, pred_class_train_LDA)
```

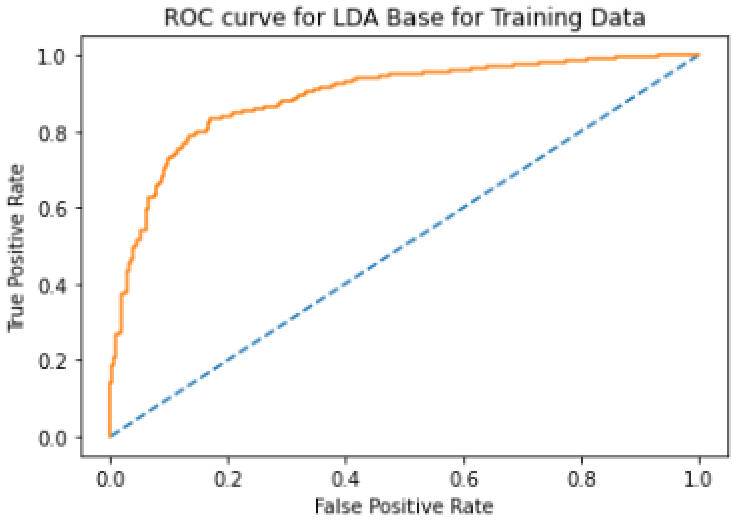
```
array([[200, 107],
       [ 69, 685]], dtype=int64)
```

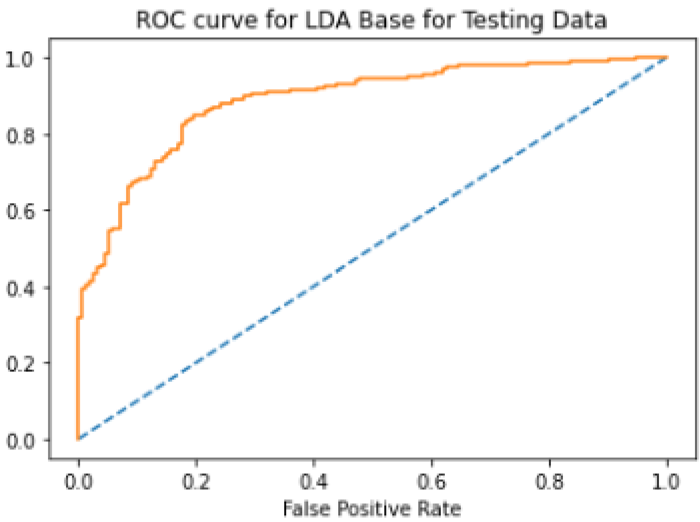
```
print('Classification Report on Testing Data for LDA\n\n',metrics.classification_report(y_test,pred_class_test_LDA),'\n')
```

```
Classification Report on Testing Data for LDA

              precision    recall  f1-score   support

           0       0.77      0.73      0.74       153
           1       0.86      0.89      0.88       303

    accuracy                           0.83       456
   macro avg       0.82      0.81      0.81       456
weighted avg       0.83      0.83      0.83       456
```

```
confusion_matrix(y_test, pred_class_test_LDA)
```

```
array([[111,  42],
       [ 34, 269]], dtype=int64)
```





Applying GridSearchCV on LDA

```
]: from sklearn.model_selection import RepeatedStratifiedKFold
   from numpy import arange
   best_model_LDA = LinearDiscriminantAnalysis()
   cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
```

```
]: grid['shrinkage'] = arange(0, 1, 0.01)
   grid = dict()
   grid['solver'] = ['svd', 'lsqr', 'eigen']
   search = GridSearchCV(best_model_LDA, grid, scoring='accuracy', cv=cv, n_jobs=-1)
   results = search.fit(X_train, y_train)
```

```
]: print(results.best_estimator_)
```

```
LinearDiscriminantAnalysis()
```

```
]: ytrain_predict = results.predict(X_train)
   ytest_predict = results.predict(X_test)
```

```
]: ytest_predict_prob=results.predict_proba(X_test)
   pd.DataFrame(ytest_predict_prob).head()
```


```
ytest_predict_prob=results.predict_proba(X_test)
pd.DataFrame(ytest_predict_prob).head()
```


	0	1
0	0.462093	0.537907
1	0.133955	0.866045
2	0.006414	0.993586
3	0.861210	0.138790
4	0.056545	0.943455

```
confusion_matrix(y_train, ytrain_predict)
```

```
array([[200, 107],
       [ 69, 685]], dtype=int64)
```

```
print("Classification Report on Training Data for LDA\n \n",classification_report(y_train, ytrain_predict),'\n');
```

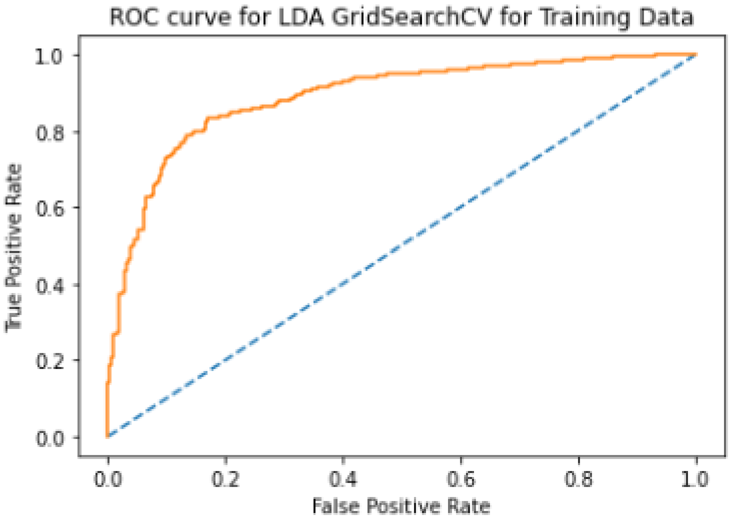
```
Classification Report on Training Data for LDA

              precision    recall  f1-score   support

           0       0.74      0.65      0.69       307
           1       0.86      0.91      0.89       754

    accuracy                           0.83      1061
   macro avg       0.80      0.78      0.79      1061
weighted avg       0.83      0.83      0.83      1061
```

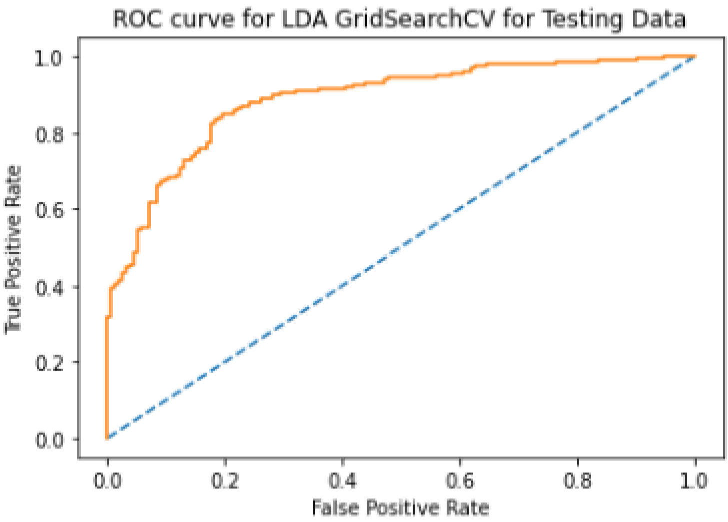
```
print("Classification Report on Testing Data for LDA\n \n",classification_report(y_test, ytest_predict),'\n');
```

```
Classification Report on Testing Data for LDA

              precision    recall  f1-score   support

           0       0.77      0.73      0.74       153
           1       0.86      0.89      0.88       303

    accuracy                           0.83       456
   macro avg       0.82      0.81      0.81       456
weighted avg       0.83      0.83      0.83       456
```

```
confusion_matrix(y_test, ytest_predict)
```

```
array([[111,  42],
       [ 34, 269]], dtype=int64)
```





1.5 Apply KNN Model and Naïve Bayes Model. Interpret the results.

Naïve Bayes Model

```
NB_model = GaussianNB()
NB_model.fit(X_train, y_train)
```

```
GaussianNB()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
y_train_predict = NB_model.predict(X_train)
y_test_predict = NB_model.predict(X_test)
```

```
print(metrics.confusion_matrix(y_train, y_train_predict))
print('Classification Report on Training Data for Gaussian NB \n\n', metrics.classification_report(y_train, y_train_predict))
```

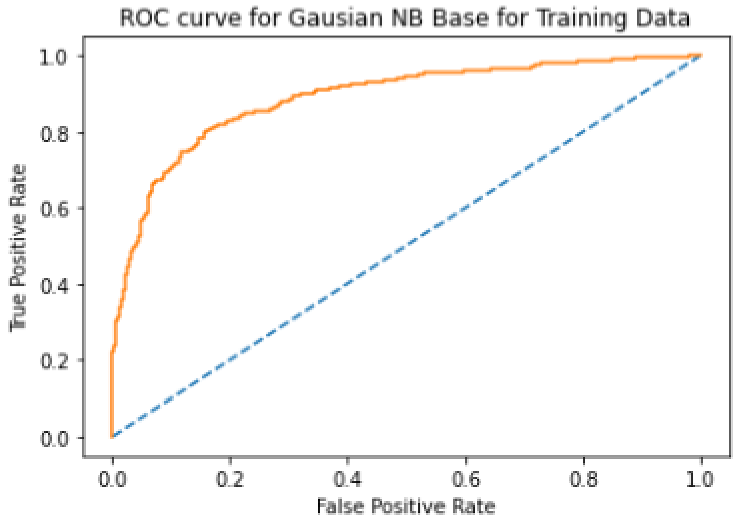
```
[[211  96]
 [ 79 675]]
Classification Report on Training Data for Gaussian NB

              precision    recall  f1-score   support

           0       0.73      0.69      0.71       307
           1       0.88      0.90      0.89       754

    accuracy                           0.84      1061
   macro avg       0.80      0.79      0.80      1061
weighted avg       0.83      0.84      0.83      1061
```

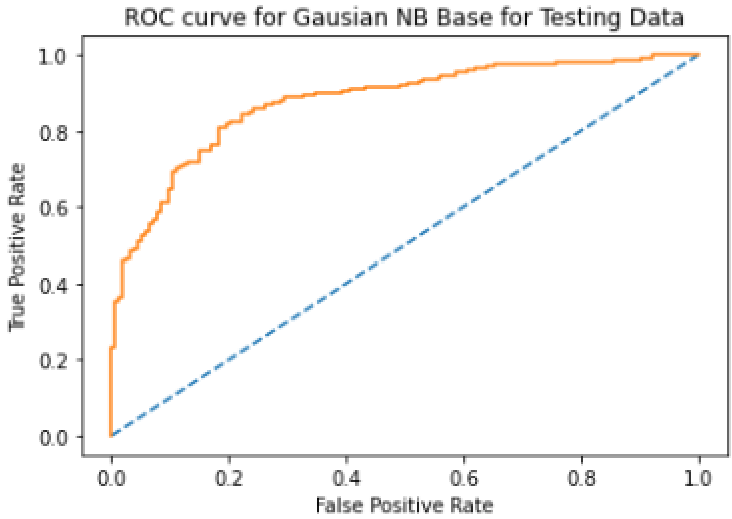
```
print(metrics.confusion_matrix(y_test, y_test_predict))
print("Classification Report on Testing Data for Gaussian NB \n\n",metrics.classification_report(y_test, y_test_predict))
```

```
[[112  41]
 [ 40 263]]
Classification Report on Testing Data for Gaussian NB

              precision    recall  f1-score   support

           0       0.74      0.73      0.73       153
           1       0.87      0.87      0.87       303

    accuracy                           0.82       456
   macro avg       0.80      0.80      0.80       456
weighted avg       0.82      0.82      0.82       456
```

```
imps = permutation_importance(NB_model, X_test, y_test)
importances = imps.importances_mean
std = imps.importances_std
indices = np.argsort(importances)[::-1]
print("Feature ranking:")
for f in range(X_test.shape[1]):
    print("%d. %s (%f)" % (f + 1, df.columns[indices[f]], importances[indices[f]]))
```

```
Feature ranking:
1. Blair (0.055263)
2. Hague (0.044298)
3. economic_cond_household (0.035088)
4. Europe (0.018860)
5. age (0.017105)
6. vote (0.007456)
7. economic_cond_national (0.003070)
8. political_knowledge (-0.003947)
```





Applying GridSearchCV on Gausian Naive Bayes

```
cv_method = RepeatedStratifiedKFold(n_splits=10,
                                    n_repeats=5,
                                    random_state=1)
```

```
from sklearn.preprocessing import PowerTransformer
params_NB = {'var_smoothing': np.logspace(0,-9, num=100)}

BestModel_NB = GridSearchCV(estimator=NB_model,
                     param_grid=params_NB,
                     cv=cv_method,
                     verbose=1,
                     scoring='accuracy')

Data_transformed = PowerTransformer().fit_transform(X_test)

BestModel_NB.fit(Data_transformed, y_test);
```

```
Fitting 50 folds for each of 100 candidates, totalling 5000 fits
```

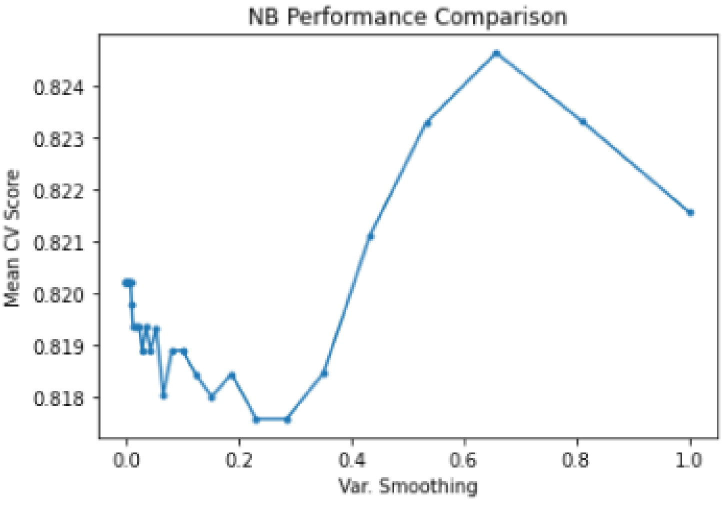
```
BestModel_NB.best_params_
```

```
{'var_smoothing': 0.657933224657568}
```

```
BestModel_NB.best_score_
```

```
0.8246280193236715
```

```
plt.plot(results_NB['var_smoothing'], results_NB['test_score'], marker = '.')
plt.xlabel('Var. Smoothing')
plt.ylabel("Mean CV Score")
plt.title("NB Performance Comparison")
plt.show()
```



```
# predict the target on the test dataset
from sklearn.metrics import accuracy_score
predict_test = BestModel_NB.predict(Data_transformed)
# Accuracy Score on test dataset
accuracy_test = accuracy_score(y_test,predict_test)
print('accuracy_score on test dataset : ', accuracy_test)
```

```
accuracy_score on test dataset :  0.831140350877193
```

```
ytrain_predict_BestModel_NB = BestModel_NB.predict(X_train)
ytest_predict_BestModel_NB = BestModel_NB.predict(X_test)
```

```
ytest_predict_prob_BestModel_NB=BestModel_NB.predict_proba(X_test)
```

```
print(classification_report(y_train, ytrain_predict_BestModel_NB),'\n');
```

```
              precision    recall  f1-score   support

           0       0.73      0.70      0.71       307
           1       0.88      0.89      0.89       754

    accuracy                           0.84      1061
   macro avg       0.80      0.79      0.80      1061
weighted avg       0.83      0.84      0.84      1061
```

```
confusion_matrix(y_train, ytrain_predict_BestModel_NB)
```

```
array([[214,  93],
       [ 81, 673]], dtype=int64)
```

```
print(classification_report(y_test, ytest_predict_BestModel_NB),'\n');
```

```
              precision    recall  f1-score   support

           0       0.75      0.75      0.75       153
           1       0.87      0.87      0.87       303

    accuracy                           0.83       456
   macro avg       0.81      0.81      0.81       456
weighted avg       0.83      0.83      0.83       456
```

```
confusion_matrix(y_test, ytest_predict_BestModel_NB)
```

```
array([[115,  38],
       [ 39, 264]], dtype=int64)
```

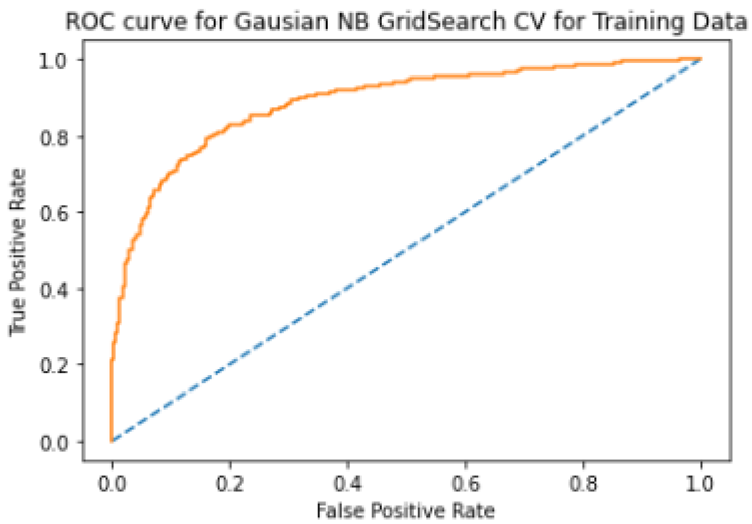
```
confusion_matrix(y_test, ytest_predict_BestModel_NB)
```

```
array([[115,  38],
       [ 39, 264]], dtype=int64)
```

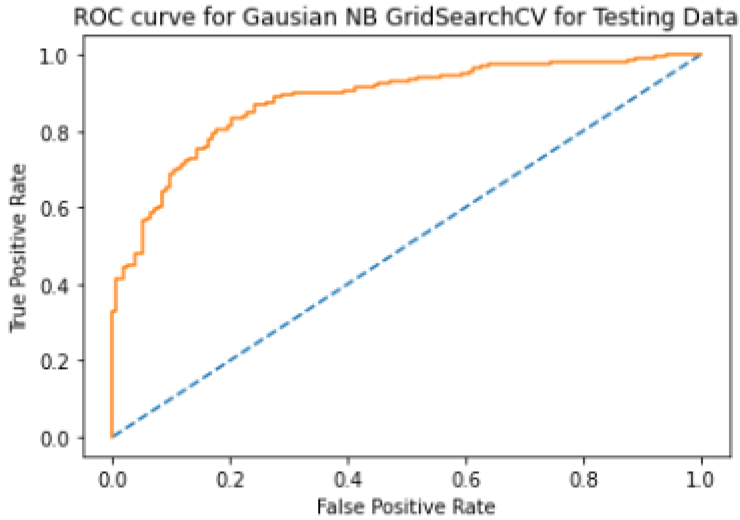
```
imps = permutation_importance(BestModel_NB, X_test, y_test)
importances = imps.importances_mean
std = imps.importances_std
indices = np.argsort(importances)[::-1]
print("Feature ranking:")
for f in range(X_test.shape[1]):
    print("%d. %s (%f)" % (f + 1, df.columns[indices[f]], importances[indices[f]]))
```

```
Feature ranking:
1. Blair (0.092544)
2. Hague (0.053509)
3. economic_cond_household (0.046491)
4. Europe (0.025877)
5. age (0.017982)
6. vote (0.015789)
7. economic_cond_national (0.003947)
8. political_knowledge (0.001316)
```

```
AUC: 0.888
```



```
AUC: 0.882
```



KNN

```python
from sklearn.neighbors import KNeighborsClassifier
```

```python
KNN_model=KNeighborsClassifier()
KNN_model.fit(X_train,y_train)
```

```
KNeighborsClassifier()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```python
## Performance Matrix on train data set
y_train_predict_KNN = KNN_model.predict(X_train)
model_score = KNN_model.score(X_train, y_train)
print(model_score)
print(metrics.confusion_matrix(y_train, y_train_predict_KNN))
print('Classification Report on Training Data for KNN \n\n',metrics.classification_report(y_train, y_train_predict_KNN))
```

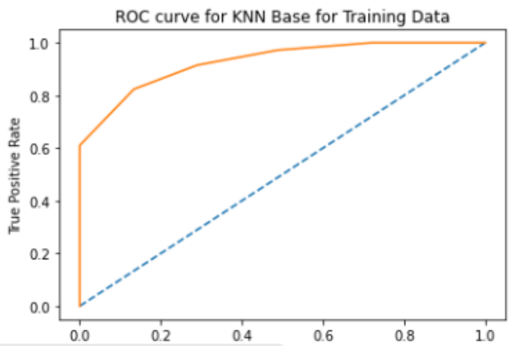
```
0.8557964184731386
[[218  89]
 [ 64 690]]
 Classification Report on Training Data for KNN 

               precision    recall  f1-score   support

           0       0.77      0.71      0.74       307
           1       0.89      0.92      0.90       754

    accuracy                           0.86      1061
   macro avg       0.83      0.81      0.82      1061
weighted avg       0.85      0.86      0.85      1061
```

```
## Performance Matrix on test data set
y_test_predict_KNN = KNN_model.predict(X_test)
model_score = KNN_model.score(X_test, y_test)
print(model_score)
print(metrics.confusion_matrix(y_test, y_test_predict_KNN))
print("Classification Report on Testing Data for KNN \n \n",metrics.classification_report(y_test, y_test_predict_KNN))
```

```
0.8245614035087719
[[105  48]
 [ 32 271]]
Classification Report on Testing Data for KNN

              precision    recall  f1-score   support

           0       0.77      0.69      0.72       153
           1       0.85      0.89      0.87       303

    accuracy                           0.82       456
   macro avg       0.81      0.79      0.80       456
weighted avg       0.82      0.82      0.82       456
```

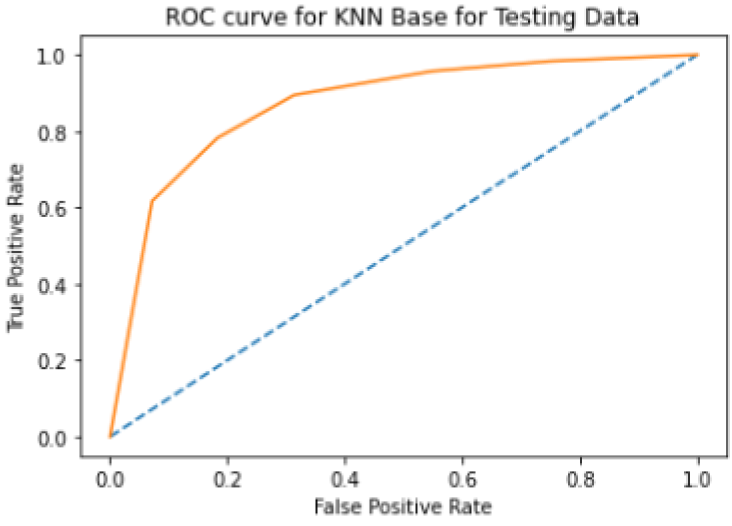
```
# predict probabilities
probs_KNN_base_train = KNN_model.predict_proba(X_train)
# keep probabilities for the positive outcome only
probs_KNN_base_train = probs_KNN_base_train[:, 1]
# calculate AUC
KNN_base_train_auc = roc_auc_score(y_train, probs_KNN_base_train)
print('AUC: %.3f' % KNN_base_train_auc)
# calculate roc curve
train_fpr_KNN_base, train_tpr_KNN_base, train_thresholds_KNN_base = roc_curve(y_train, probs_KNN_base_train)
plt.title('ROC curve for KNN Base for Training Data')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(train_fpr_KNN_base, train_tpr_KNN_base);
```

AUC: 0.927



```
AUC: 0.870
```

```
]: [<matplotlib.lines.Line2D at 0x1fd46302ee0>]
```



Applying GridSearch CV on KNN

```
params_KNN = {'n_neighbors': [1, 2, 3, 4, 5, 6, 7],
              'p': [1, 2, 5]}
# we need to define a dictionary of KNN parameters for the grid search. Here, we will consider K values between 3 and 7 and
# p values of 1 (Manhattan), 2 (Euclidean), and 5 (Minkowski).
```

```
cv_method = RepeatedStratifiedKFold(n_splits=10,
                                    n_repeats=5,
                                    random_state=1)
```

```
gs_KNN = GridSearchCV(estimator=KNeighborsClassifier(),
                      param_grid=params_KNN,
                      cv=cv_method,
                      verbose=1,  # verbose: the higher, the more messages
                      scoring='accuracy',
                      return_train_score=True)

#we pass the KNeighborsClassifier() and KNN_params as the model and the parameter dictionary into the GridSearchCV function. In additio
```

```
gs_KNN.fit(X_train, y_train)
```

```
Fitting 50 folds for each of 21 candidates, totalling 1050 fits

GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=5, n_splits=10, random_state=1),
             estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7], 'p': [1, 2, 5]},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
gs_KNN.best_params_
#To get the best parameter values, we call the best_params_ attribute.
```

```
{'n_neighbors': 7, 'p': 5}
```

```
#Redefining neighbours

params_KNN = {'n_neighbors': [6, 7,9,11,13,15],
              'p': [1, 2, 5]}
```

```
              'p': [1, 2, 5]}
```

In [187]:
```
cv_method = RepeatedStratifiedKFold(n_splits=5,
                                    n_repeats=5,
                                    random_state=1)
```

In [188]:
```
gs_KNN1 = GridSearchCV(estimator=KNeighborsClassifier(),
                      param_grid=params_KNN,
                      cv=cv_method,
                      verbose=1,  # verbose: the higher, the more messages
                      scoring='accuracy',
                      return_train_score=True)
```

In [189]:
```
gs_KNN1.fit(X_train, y_train)
```

```
Fitting 25 folds for each of 18 candidates, totalling 450 fits
```

Out[189]:
```
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=5, n_splits=5, random_state=1),
             estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [6, 7, 9, 11, 13, 15], 'p': [1, 2, 5]},
             return_train_score=True, scoring='accuracy', verbose=1)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

]:
```
gs_KNN1.best_params_
```

]:
```
{'n_neighbors': 15, 'p': 1}
```

]:
```
params_KNN = {'n_neighbors': [13,15,17,19,21],
              'p': [1, 2, 5]}
```

]:
```
cv_method = RepeatedStratifiedKFold(n_splits=3,
                                    n_repeats=5,
                                    random_state=1)
```

]:
```
gs_KNN2 = GridSearchCV(estimator=KNeighborsClassifier(),
                      param_grid=params_KNN,
                      cv=cv_method,
                      verbose=1,  # verbose: the higher, the more messages
                      scoring='accuracy',
                      return_train_score=True)
```

]:
```
gs_KNN2.fit(X_train, y_train)
```

```
Fitting 15 folds for each of 15 candidates, totalling 225 fits
```

]:
```
GridSearchCV(cv=RepeatedStratifiedKFold(n_repeats=5, n_splits=3, random_state=1),
             estimator=KNeighborsClassifier(),
             param_grid={'n_neighbors': [13, 15, 17, 19, 21], 'p': [1, 2, 5]},
             return_train_score=True, scoring='accuracy', verbose=1)
```

```
gs_KNN2.best_params_
```

```
{'n_neighbors': 17, 'p': 1}
```

```
gs_KNN.best_score_
```

```
0.8175365896667253
```

```
gs_KNN1.best_score_
```

```
0.8233687660554523
```

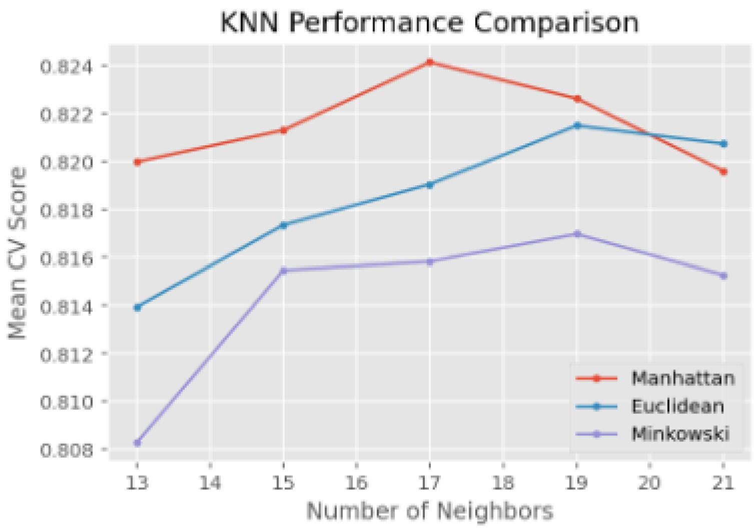
```
gs_KNN2.best_score_
```

```
0.8241294686918156
```

```
gs_KNN2.cv_results_['mean_test_score']
 #To extract more cross-validation results, we can call gs.csv_results - a dictionary consisting of run details for each fold.
```

```
array([0.81997727, 0.81394931, 0.80829692, 0.82129927, 0.81734341,
       0.81545803, 0.82412947, 0.81903992, 0.81583255, 0.82261861,
       0.8214908 , 0.81696462, 0.81960169, 0.82073644, 0.81526971])
```

	n_neighbors	p	test_score	metric
0	13	1	0.819977	Manhattan
1	13	2	0.813949	Euclidean
2	13	5	0.808297	Minkowski
3	15	1	0.821299	Manhattan
4	15	2	0.817343	Euclidean
5	15	5	0.815458	Minkowski
6	17	1	0.824129	Manhattan
7	17	2	0.819040	Euclidean
8	17	5	0.815833	Minkowski
9	19	1	0.822619	Manhattan
10	19	2	0.821491	Euclidean
11	19	5	0.816965	Minkowski
12	21	1	0.819602	Manhattan
13	21	2	0.820736	Euclidean
14	21	5	0.815270	Minkowski




```
## Performance Matrix on train data set
y_train_predict_KNN7 = gs_KNN2.predict(X_train)
model_score = gs_KNN2.score(X_train, y_train)
print(model_score)
print(metrics.confusion_matrix(y_train, y_train_predict_KNN7))
print('Classification Report on Training Data for KNN for K=17 \n\n',metrics.classification_report(y_train, y_train_predict_KNN7))
```

```
0.8388312912346843
[[202 105]
 [ 66 688]]
Classification Report on Training Data for KNN for K=17

              precision    recall  f1-score   support

           0       0.75      0.66      0.70       307
           1       0.87      0.91      0.89       754

    accuracy                           0.84      1061
   macro avg       0.81      0.79      0.80      1061
weighted avg       0.83      0.84      0.84      1061
```

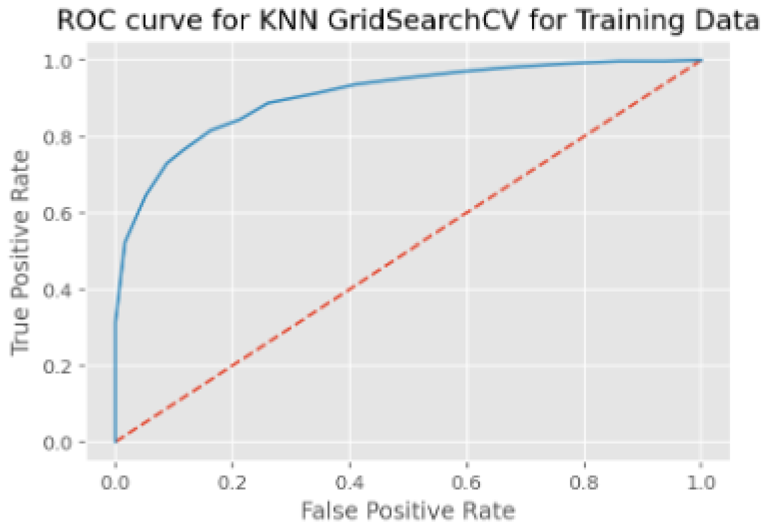
```
## Performance Matrix on test data set
y_test_predict_KNN7 = gs_KNN2.predict(X_test)
model_score = gs_KNN2.score(X_test, y_test)
print(model_score)
print(metrics.confusion_matrix(y_test, y_test_predict))
print('Classification Report on Testing Data for KNN for K=17 \n\n',metrics.classification_report(y_test, y_test_predict_KNN7))
```

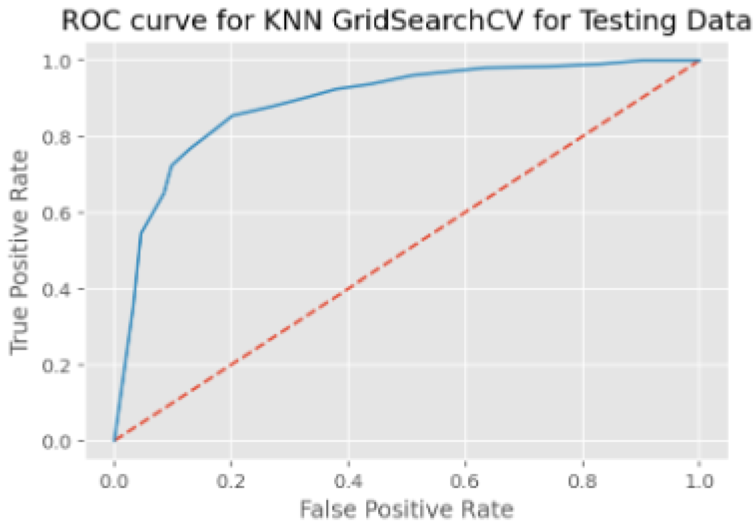
```
0.8245614035087719
[[112  41]
 [ 40 263]]
Classification Report on Testing Data for KNN for K=17

              precision    recall  f1-score   support

           0       0.78      0.67      0.72       153
           1       0.84      0.90      0.87       303

    accuracy                           0.82       456
   macro avg       0.81      0.79      0.80       456
weighted avg       0.82      0.82      0.82       456
```

```
imps = permutation_importance(gs_KNN2, X_test, y_test)
importances = imps.importances_mean
std = imps.importances_std
indices = np.argsort(importances)[::-1]
importances
print("Feature ranking:")
for f in range(X_test.shape[1]):
    print("%d. %s (%f)" % (f + 1, df.columns[indices[f]], importances[indices[f]]))
```

```
Feature ranking:
1. Blair (0.060526)
2. Europe (0.047368)
3. Hague (0.032018)
4. economic_cond_household (0.027193)
5. vote (0.009211)
6. political_knowledge (0.004825)
7. age (0.000000)
8. economic_cond_national (-0.007018)
```

```
AUC: 0.907
```


ROC curve for KNN GridSearchCV for Training Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


ROC curve for KNN GridSearchCV for Testing Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

1.6 Model Tuning, Bagging (Random Forest should be applied for Bagging), and Boosting.

Bagging

```
## Performance Matrix on train data set
y_train_predict = Bagging_model.predict(X_train)
model_score =Bagging_model.score(X_train, y_train)
print(model_score)
print(metrics.confusion_matrix(y_train, y_train_predict))
print(metrics.classification_report(y_train, y_train_predict))
```

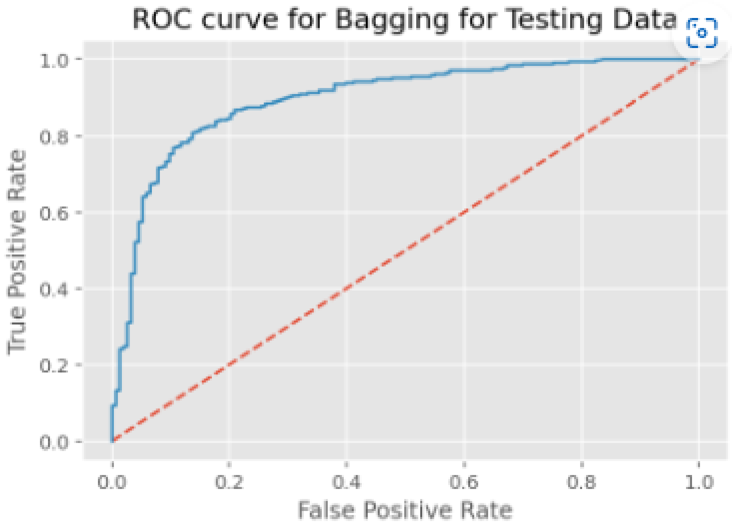
```
0.9679547596606974
[[278  29]
 [  5 749]]
              precision    recall  f1-score   support

           0       0.98      0.91      0.94       307
           1       0.96      0.99      0.98       754

    accuracy                           0.97      1061
   macro avg       0.97      0.95      0.96      1061
weighted avg       0.97      0.97      0.97      1061
```

```
## Performance Matrix on test data set
y_test_predict = Bagging_model.predict(X_test)
model_score = Bagging_model.score(X_test, y_test)
print(model_score)
print(metrics.confusion_matrix(y_test, y_test_predict))
print(metrics.classification_report(y_test, y_test_predict))
```

```
0.8289473684210527
[[104  49]
 [ 29 274]]
              precision    recall  f1-score   support

           0       0.78      0.68      0.73       153
           1       0.85      0.90      0.88       303

    accuracy                           0.83       456
   macro avg       0.82      0.79      0.80       456
weighted avg       0.83      0.83      0.83       456
```


ROC curve for Bagging for Training Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

AUC: 0.896


ROC curve for Bagging for Testing Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

Gradient Boosting

214]:
```
from sklearn.ensemble import GradientBoostingClassifier
gbcl = GradientBoostingClassifier(random_state=1)
gbcl = gbcl.fit(X_train, y_train)
```

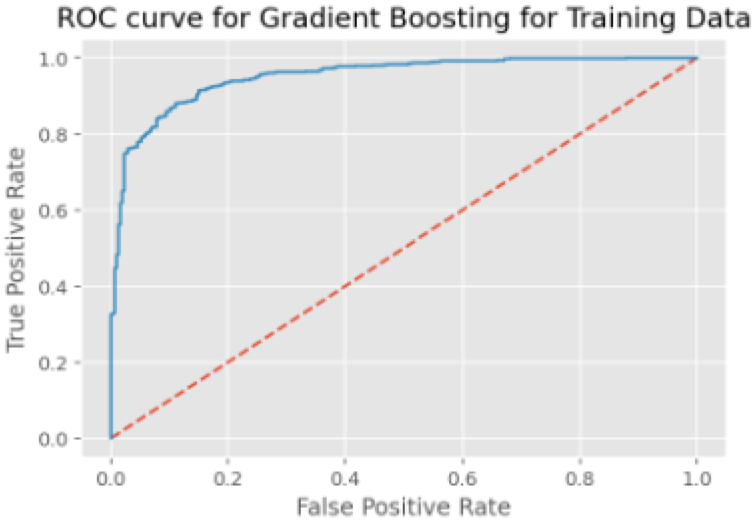
215]:
```
y_train_predict = gbcl.predict(X_train)
model_score_GraBoosting_train = gbcl.score(X_train, y_train)
print(model_score_GraBoosting_train)
print(metrics.confusion_matrix(y_train, y_train_predict))
print(metrics.classification_report(y_train, y_train_predict))
```

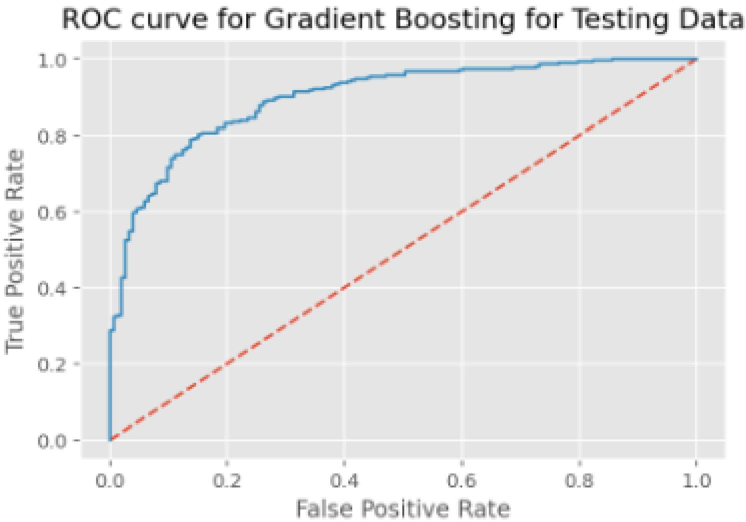
```
0.8925541941564562
[[239  68]
 [ 46 708]]
              precision    recall  f1-score   support

           0       0.84      0.78      0.81       307
           1       0.91      0.94      0.93       754

    accuracy                           0.89      1061
   macro avg       0.88      0.86      0.87      1061
weighted avg       0.89      0.89      0.89      1061
```

```
y_test_predict = gbcl.predict(X_test)
model_score_GraBoosting_test = gbcl.score(X_test, y_test)
print(model_score_GraBoosting_test)
print(metrics.confusion_matrix(y_test, y_test_predict))
print(metrics.classification_report(y_test, y_test_predict))
```

```
0.8355263157894737
[[105  48]
 [ 27 276]]
              precision    recall  f1-score   support

           0       0.80      0.69      0.74       153
           1       0.85      0.91      0.88       303

    accuracy                           0.84       456
   macro avg       0.82      0.80      0.81       456
weighted avg       0.83      0.84      0.83       456
```

AUC: 0.951


ROC curve for Gradient Boosting for Training Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0


ROC curve for Gradient Boosting for Testing Data
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

Ada Boosting

]:
```
from sklearn.ensemble import AdaBoostClassifier
ADB_model = AdaBoostClassifier(n_estimators=100,random_state=1)
ADB_model.fit(X_train,y_train)
```

]:
```
AdaBoostClassifier(n_estimators=100, random_state=1)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

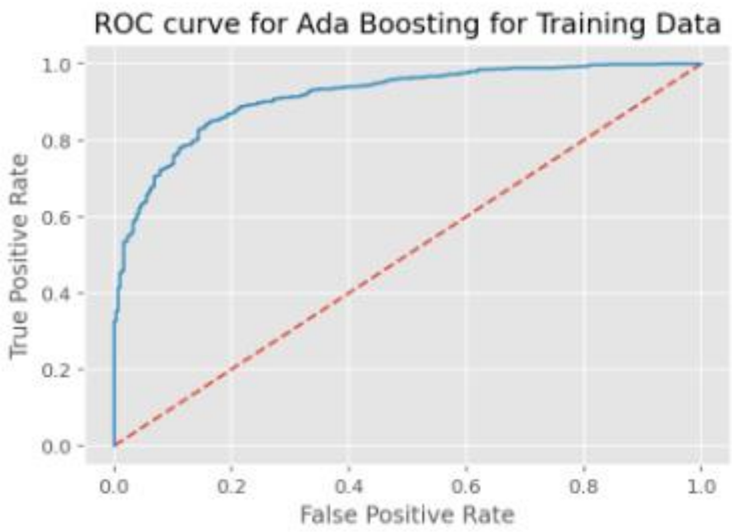
]:
```
## Performance Matrix on train data set
y_train_predict = ADB_model.predict(X_train)
model_score_AdaBoosting_train = ADB_model.score(X_train, y_train)
print(model_score_AdaBoosting_train)
print(metrics.confusion_matrix(y_train, y_train_predict))
print(metrics.classification_report(y_train, y_train_predict))
```

```
0.8501413760603205
[[214  93]
 [ 66 688]]
              precision    recall  f1-score   support

           0       0.76      0.70      0.73       307
           1       0.88      0.91      0.90       754

    accuracy                           0.85      1061
   macro avg       0.82      0.80      0.81      1061
weighted avg       0.85      0.85      0.85      1061
```

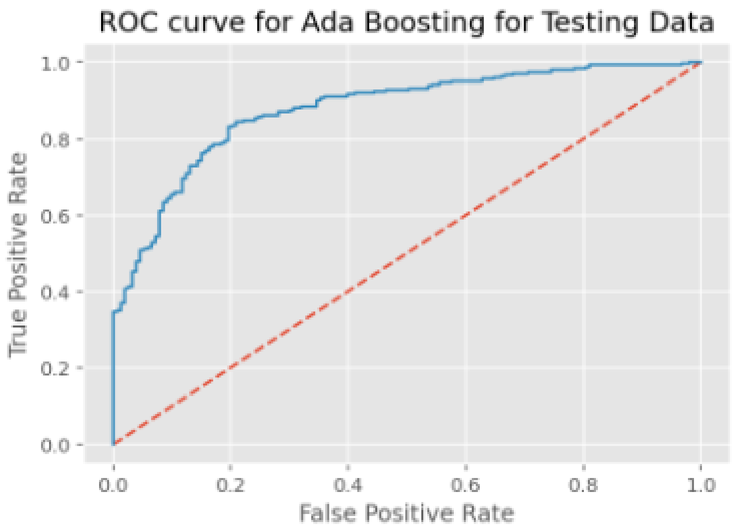
```
## Performance Matrix on test data set
y_test_predict = ADB_model.predict(X_test)
model_score_AdaBoosting_test = ADB_model.score(X_test, y_test)
print(model_score_AdaBoosting_test)
print(metrics.confusion_matrix(y_test, y_test_predict))
print(metrics.classification_report(y_test, y_test_predict))
```

```
0.8135964912280702
[[103  50]
 [ 35 268]]
              precision    recall  f1-score   support

           0       0.75      0.67      0.71       153
           1       0.84      0.88      0.86       303

    accuracy                           0.81       456
   macro avg       0.79      0.78      0.79       456
weighted avg       0.81      0.81      0.81       456
```

AUC: 0.915



AUC: 0.877



SMOTE

```
sm = SMOTE(random_state=1)
```

```
X_train_res, y_train_res = sm.fit_resample(X_train,y_train.ravel())
```

```
X_train_res.shape
```

```
(1508, 8)
```

```
y_train_res.shape
```

```
(1508,)
```

Logistic Regression with SMOTE

```
LogSMOTE_model = LogisticRegression()
```

```
LogSMOTE_model.fit(X_train_res, y_train_res)
```

```
LogisticRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
y_train_predict = LogSMOTE_model.predict(X_train_res)
model_score_LogRSMOTE_train = LogSMOTE_model.score(X_train_res, y_train_res)
print(model_score_LogRSMOTE_train)
print(metrics.confusion_matrix(y_train_res, y_train_predict))
print(metrics.classification_report(y_train_res ,y_train_predict))
```

```
0.8368700265251989
[[636 118]
 [128 626]]
              precision    recall  f1-score   support

           0       0.83      0.84      0.84       754
           1       0.84      0.83      0.84       754

    accuracy                           0.84      1508
   macro avg       0.84      0.84      0.84      1508
weighted avg       0.84      0.84      0.84      1508
```

```
## Performance Matrix on test data set
y_test_predict = LogSMOTE_model.predict(X_test)
model_score_LogRSMOTE_test = LogSMOTE_model.score(X_test, y_test)
print(model_score_LogRSMOTE_test)
print(metrics.confusion_matrix(y_test, y_test_predict))
print(metrics.classification_report(y_test, y_test_predict))
```

```
0.8114035087719298
[[127  26]
 [ 60 243]]
              precision    recall  f1-score   support

           0       0.68      0.83      0.75       153
           1       0.90      0.80      0.85       303

    accuracy                           0.81       456
   macro avg       0.79      0.82      0.80       456
weighted avg       0.83      0.81      0.82       456
```

AUC: 0.890

AUC: 0.890

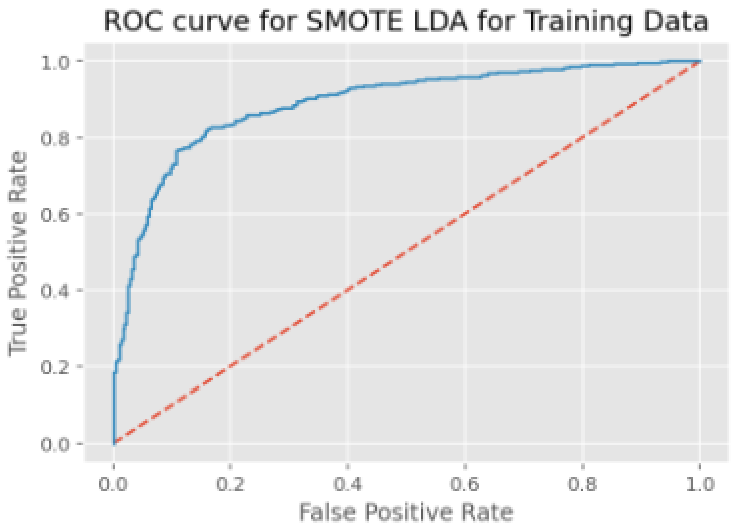
```
y_train_predict_LDASMOTE_train = LDA_smote.predict(X_train_res)
model_score_LDASMOTE_train = LDA_smote.score(X_train_res, y_train_res)
print(model_score_LDASMOTE_train)
print(metrics.confusion_matrix(y_train_res, y_train_predict_LDASMOTE_train))
print(metrics.classification_report(y_train_res ,y_train_predict_LDASMOTE_train))
```

```
0.8381962864721485
[[640 114]
 [130 624]]
              precision    recall  f1-score   support

           0       0.83      0.85      0.84       754
           1       0.85      0.83      0.84       754

    accuracy                           0.84      1508
   macro avg       0.84      0.84      0.84      1508
weighted avg       0.84      0.84      0.84      1508
```

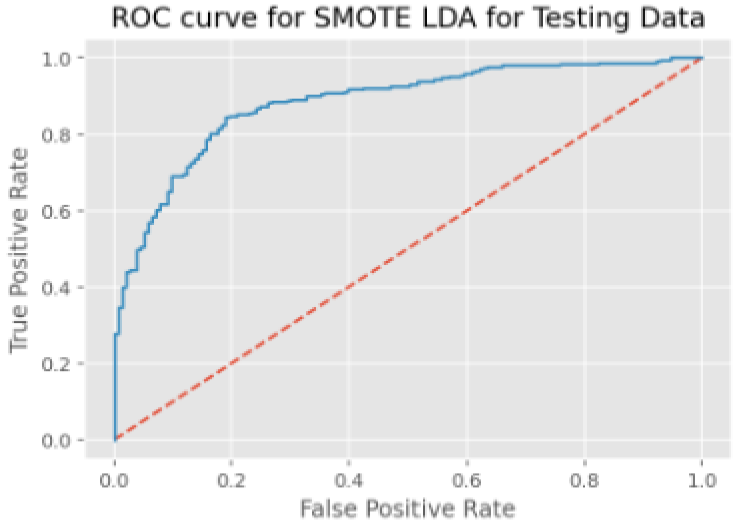
```
## Performance Matrix on test data set
y_test_predict_LDASMOTE_test = LDA_smote.predict(X_test)
model_score_LDASMOTE_test = LDA_smote.score(X_test, y_test)
print(model_score_LDASMOTE_test)
print(metrics.confusion_matrix(y_test, y_test_predict_LDASMOTE_test))
print(metrics.classification_report(y_test, y_test_predict_LDASMOTE_test))
```

```
0.8048245614035088
[[128  25]
 [ 64 239]]
              precision    recall  f1-score   support

           0       0.67      0.84      0.74       153
           1       0.91      0.79      0.84       303

    accuracy                           0.80       456
   macro avg       0.79      0.81      0.79       456
weighted avg       0.83      0.80      0.81       456
```

```
AUC: 0.890
```



AUC: 0.890



KNN with SMOTE

```
]: from sklearn.neighbors import KNeighborsClassifier

   KNN_SM_model=KNeighborsClassifier()
   KNN_SM_model.fit(X_train_res,y_train_res)
```

]: KNeighborsClassifier()

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook. On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

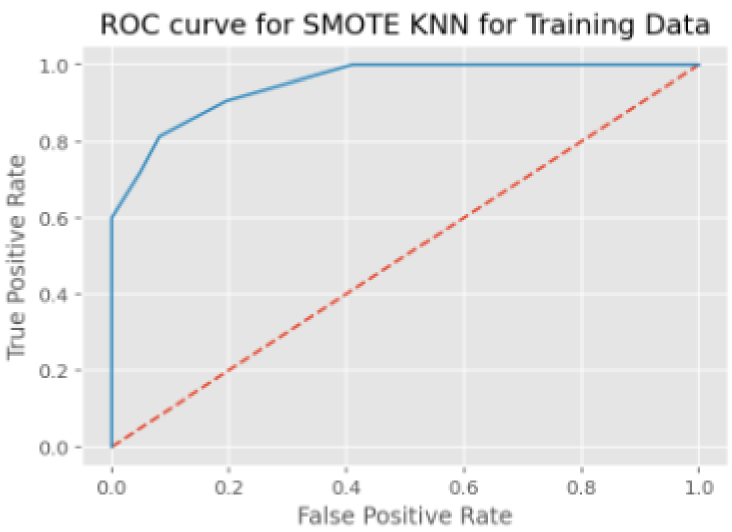
```
]: ## Performance Matrix on train data set
   y_train_predict = KNN_SM_model.predict(X_train_res)
   model_score = KNN_SM_model.score(X_train_res, y_train_res)
   print(model_score)
   print(metrics.confusion_matrix(y_train_res, y_train_predict))
   print(metrics.classification_report(y_train_res, y_train_predict))
```

```
0.8859416445623343
[[723  31]
 [141 613]]
              precision    recall  f1-score   support

           0       0.84      0.96      0.89       754
           1       0.95      0.81      0.88       754

    accuracy                           0.89      1508
   macro avg       0.89      0.89      0.89      1508
weighted avg       0.89      0.89      0.89      1508
```

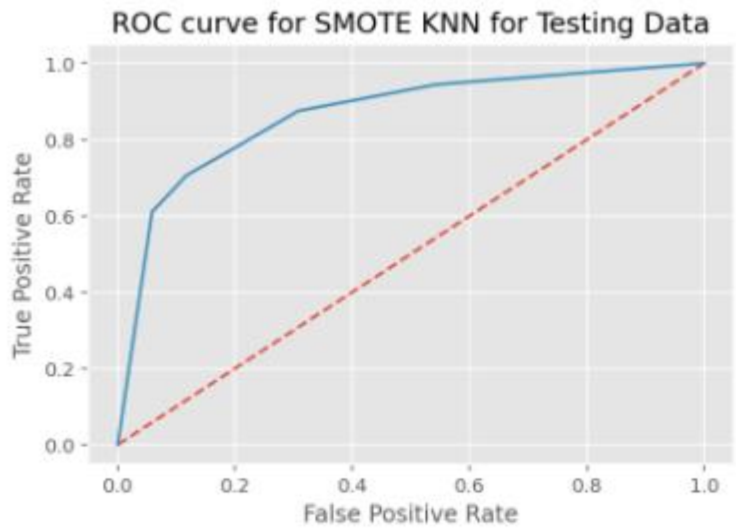
```
## Performance Matrix on test data set
y_test_predict = KNN_SM_model.predict(X_test)
model_score = KNN_SM_model.score(X_test, y_test)
print(model_score)
print(metrics.confusion_matrix(y_test, y_test_predict))
print(metrics.classification_report(y_test, y_test_predict))
```

```
0.7828947368421053
[[124  29]
 [ 70 233]]
              precision    recall  f1-score   support

           0       0.64      0.81      0.71       153
           1       0.89      0.77      0.82       303

    accuracy                           0.78       456
   macro avg       0.76      0.79      0.77       456
weighted avg       0.81      0.78      0.79       456
```

```
AUC: 0.950
```



AUC: 0.865



GNB with SMOTE

```
;0]: NB_SM_model = GaussianNB()
     NB_SM_model.fit(X_train_res, y_train_res)
```

;0]: GaussianNB()

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

```
;1]: ## Performance Matrix on train data set
     y_train_predict = NB_SM_model.predict(X_train_res)
     model_score = NB_SM_model.score(X_train_res, y_train_res)
     print(model_score)
     print(metrics.confusion_matrix(y_train_res, y_train_predict))
     print(metrics.classification_report(y_train_res, y_train_predict))
```

```
0.8348806366047745
[[634 120]
 [129 625]]
              precision    recall  f1-score   support

           0       0.83      0.84      0.84       754
           1       0.84      0.83      0.83       754

    accuracy                           0.83      1508
   macro avg       0.83      0.83      0.83      1508
weighted avg       0.83      0.83      0.83      1508
```

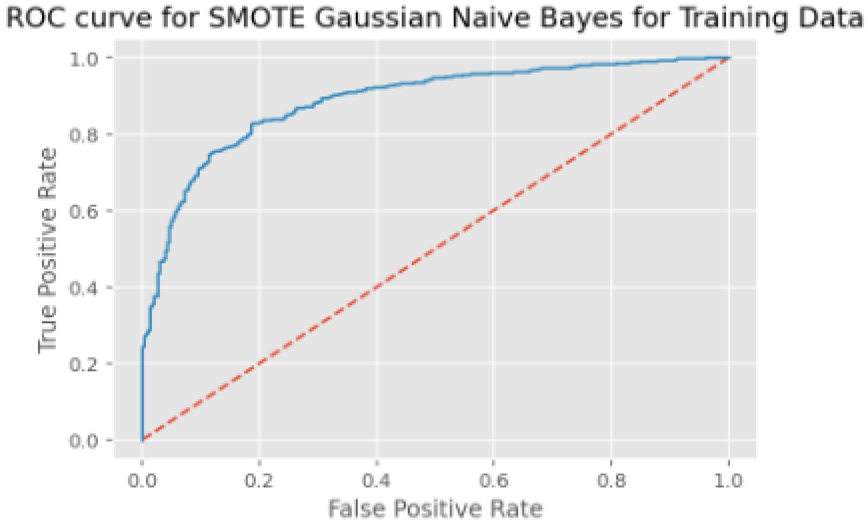
```
## Performance Matrix on test data set
y_test_predict = NB_SM_model.predict(X_test)
model_score = NB_SM_model.score(X_test, y_test)
print(model_score)
print(metrics.confusion_matrix(y_test, y_test_predict))
print(metrics.classification_report(y_test, y_test_predict))
```

```
0.8070175438596491
[[125  28]
 [ 60 243]]
              precision    recall  f1-score   support

           0       0.68      0.82      0.74       153
           1       0.90      0.80      0.85       303

    accuracy                           0.81       456
   macro avg       0.79      0.81      0.79       456
weighted avg       0.82      0.81      0.81       456
```

```
imps = permutation_importance(NB_SM_model, X_test, y_test)
importances = imps.importances_mean
std = imps.importances_std
indices = np.argsort(importances)[::-1]
print("Feature ranking:")
for f in range(X_test.shape[1]):
    print("%d. %s (%f)" % (f + 1, df.columns[indices[f]], importances[indices[f]]))
```

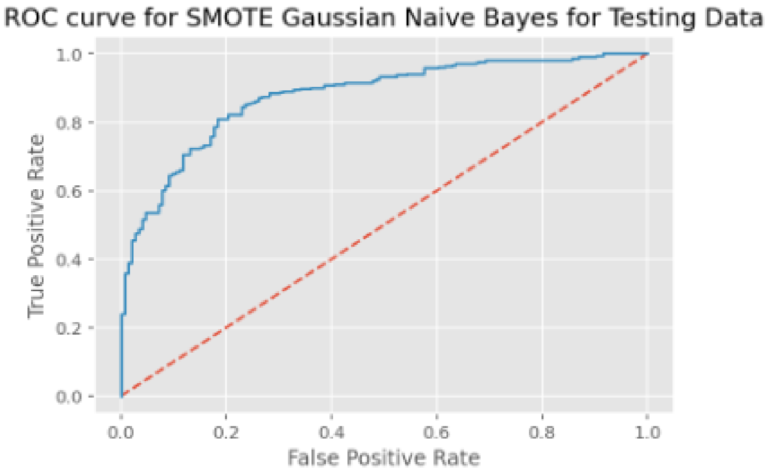
```
Feature ranking:
1. Blair (0.078509)
2. economic_cond_household (0.041667)
3. Hague (0.041228)
4. vote (0.018860)
5. age (0.015789)
6. Europe (0.013596)
7. economic_cond_national (0.008772)
8. political_knowledge (0.002193)
```

```
AUC: 0.888
```




```python
# keep probabilities for the positive outcome only
probs_nbsm_test = probs_nbsm_test[:, 1]
# calculate AUC
nbsm_test_auc = roc_auc_score(y_test, probs_nbsm_test)
print('AUC: %.3f' % nbsm_test_auc)
# calculate roc curve
test_fpr_nbsm, test_tpr_nbsm, test_thresholds_nbsm = roc_curve(y_test, probs_nbsm_test)
plt.title('ROC curve for SMOTE Gaussian Naive Bayes for Testing Data')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(test_fpr_nbsm, test_tpr_nbsm);
```

```
AUC: 0.878
```



Problem 2: In this particular project, we are going to work on the inaugural corpora from the nltk in Python. We will be looking at the following speeches of the Presidents of the United States of America:¶

President Franklin D. Roosevelt in 1941

President John F. Kennedy in 1961

President Richard Nixon in 1973

2.1 Find the number of characters, words, and sentences for the mentioned documents.¶

]:
```
print("The number of characters in 1941-Roosevelt.txt is:",len(inaugural.raw('1941-Roosevelt.txt')))
print("The number of characters in 1961-Kennedy.txt is:",len(inaugural.raw('1961-Kennedy.txt')))
print("The number of characters in 1973-Nixon.txt is:", len(inaugural.raw('1973-Nixon.txt')))
```

```
The number of characters in 1941-Roosevelt.txt is: 7571
The number of characters in 1961-Kennedy.txt is: 7618
The number of characters in 1973-Nixon.txt is: 9991
```

]:
```
print("The number of words in 1941-Roosevelt.txt is:",len(inaugural.words('1941-Roosevelt.txt')))
print("The number of words in 1961-Kennedy.txt is:",len(inaugural.words('1961-Kennedy.txt')))
print("The number of words in 1973-Nixon.txt is:", len(inaugural.words('1973-Nixon.txt')))
```

```
The number of words in 1941-Roosevelt.txt is: 1536
The number of words in 1961-Kennedy.txt is: 1546
The number of words in 1973-Nixon.txt is: 2028
```

]:
```
print("The number of sentences in 1941-Roosevelt.txt is:",len(inaugural.sents('1941-Roosevelt.txt')))
print("The number of sentences in 1961-Kennedy.txt is:",len(inaugural.sents('1961-Kennedy.txt')))
print("The number of sentences in 1973-Nixon.txt is:", len(inaugural.sents('1973-Nixon.txt')))
```

```
The number of sentences in 1941-Roosevelt.txt is: 68
The number of sentences in 1961-Kennedy.txt is: 52
The number of sentences in 1973-Nixon.txt is: 69
```

2.2 Remove all the stopwords from the three speeches

```
all_words=list(inaugural.words('1973-Nixon.txt'))
stop_words =stopwords.words('english') + list(string.punctuation)
clean_all_words = (x.lower() for x in all_words)
clean_all_words_N = [word for word in clean_all_words if word not in stop_words]
```

```
clean_all_words_n = [word for word in clean_all_words_N if word.isalpha()]
```

```
print("The number of words after cleaning in 1973-Nixon.txt are:",(len(clean_all_words_n)))
```

```
The number of words after cleaning in 1973-Nixon.txt are: 832
```



```
print("The number of words after cleaning in 1973-Nixon.txt are:",(len(clean_all_words_n)))
```

```
The number of words after cleaning in 1973-Nixon.txt are: 832
```



```
all_words=list(inaugural.words('1941-Roosevelt.txt'))
stop_words =stopwords.words('english') + list(string.punctuation)
clean_all_words = (x.lower() for x in all_words)
clean_all_words_R = [word for word in clean_all_words if word not in stop_words]
```

```
clean_all_words_r = [word for word in clean_all_words_R if word.isalpha()]
```

```
print("The number of words after cleaning in 1941-Roosevelt.txt are:",(len(clean_all_words_r)))
```

```
The number of words after cleaning in 1941-Roosevelt.txt are: 627
```

```
all_words=list(inaugural.words('1961-Kennedy.txt'))
stop_words =stopwords.words('english') + list(string.punctuation)
clean_all_words = (x.lower() for x in all_words)
clean_all_words_K = [word for word in clean_all_words if word not in stop_words]
```

```
clean_all_words_k = [word for word in clean_all_words_K if word.isalpha()]
```

```
print("The number of words after cleaning in 1961-Kennedy.txt are:",(len(clean_all_words_k)))
```

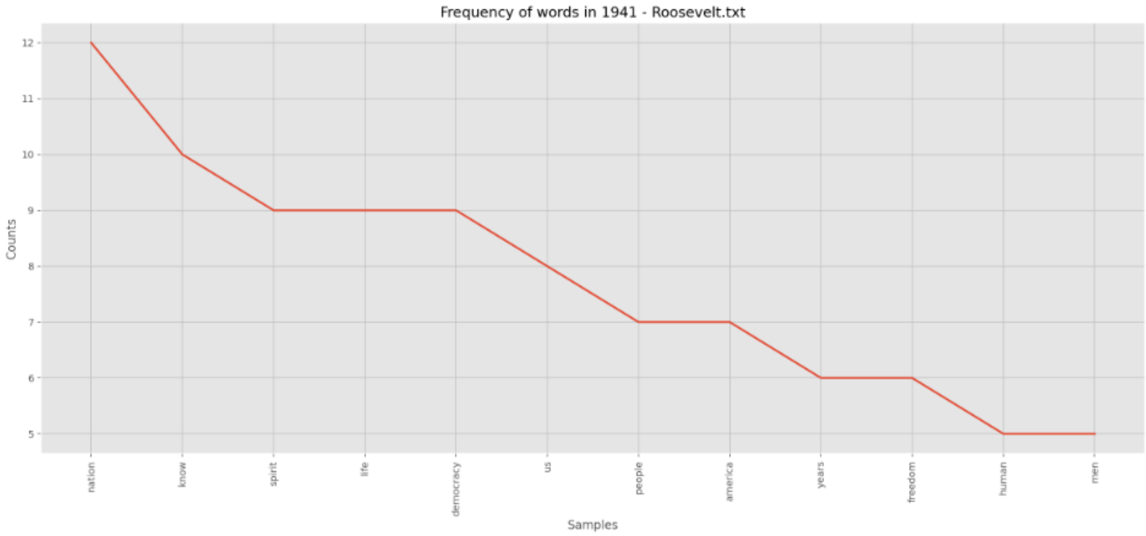
```
The number of words after cleaning in 1961-Kennedy.txt are: 692
```

2.3 Which word occurs the most number of times in his inaugural address for each president? Mention the top three words.

```
clean_all_words_freq_R = nltk.FreqDist(clean_all_words_r)
clean_all_words_freq_R.most_common(4)
```

```
[('nation', 12), ('know', 10), ('spirit', 9), ('life', 9)]
```

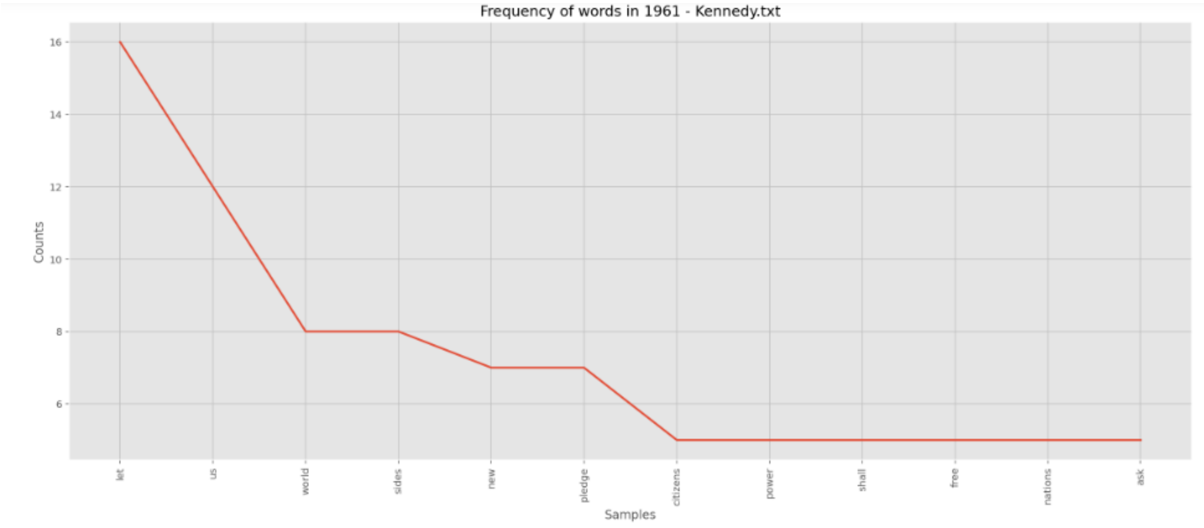
```
plt.title('Frequency of words in 1941 - Roosevelt.txt')
clean_all_words_freq_R.plot(12);
```




```
clean_all_words_freq_K = nltk.FreqDist(clean_all_words_k)
clean_all_words_freq_K.most_common(4)
```

```
[('let', 16), ('us', 12), ('world', 8), ('sides', 8)]
```

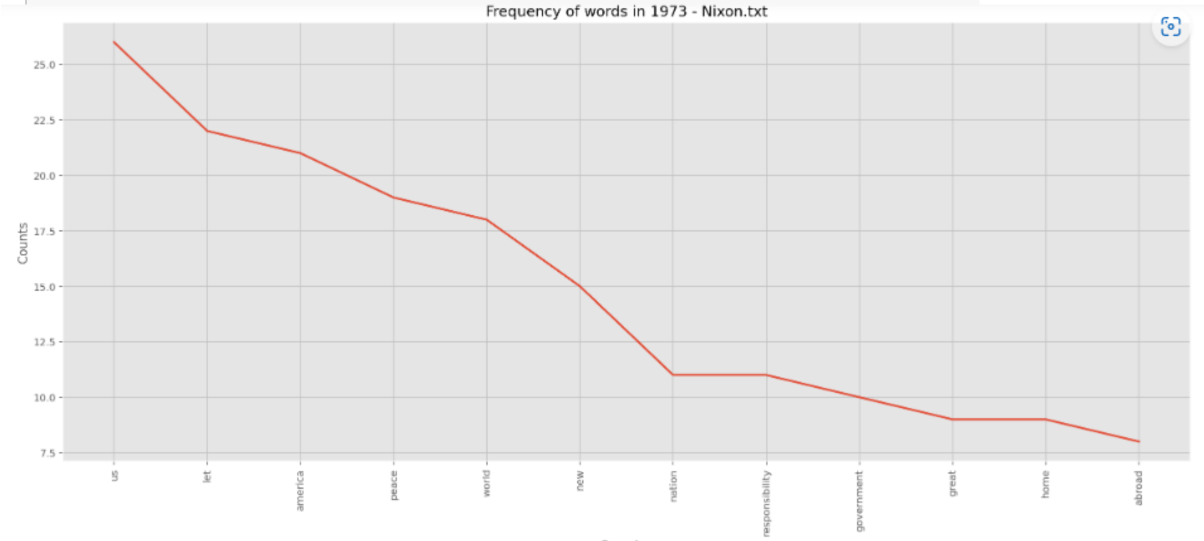
```
plt.title('Frequency of words in 1961 - Kennedy.txt')
clean_all_words_freq_K.plot(12);
```




```
clean_all_words_freq_N = nltk.FreqDist(clean_all_words_n)
clean_all_words_freq_N.most_common(4)
```

```
[('us', 26), ('let', 22), ('america', 21), ('peace', 19)]
```

```
plt.title('Frequency of words in 1973 - Nixon.txt')
clean_all_words_freq_N.plot(12);
```

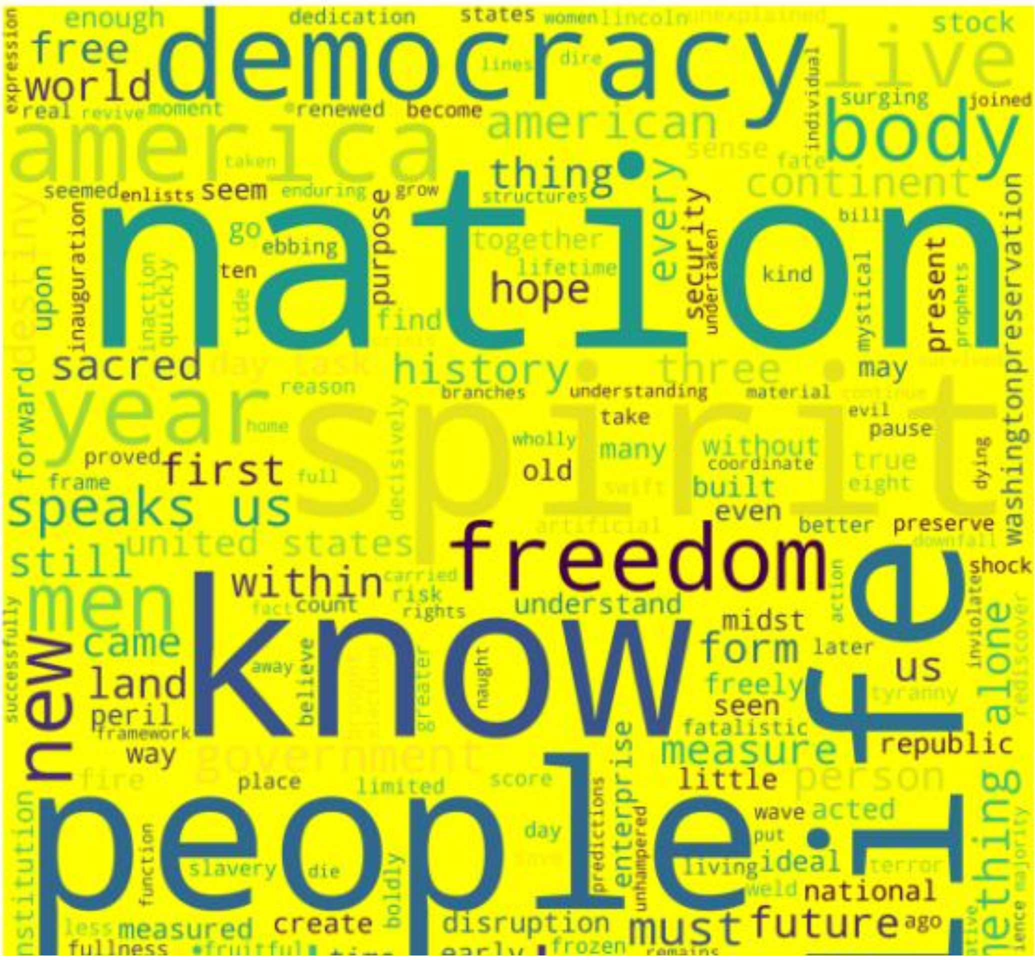


2.4 Plot the word cloud of each of the speeches of the variable.

```
: wc_a = ' '.join(clean_all_words_r)
```

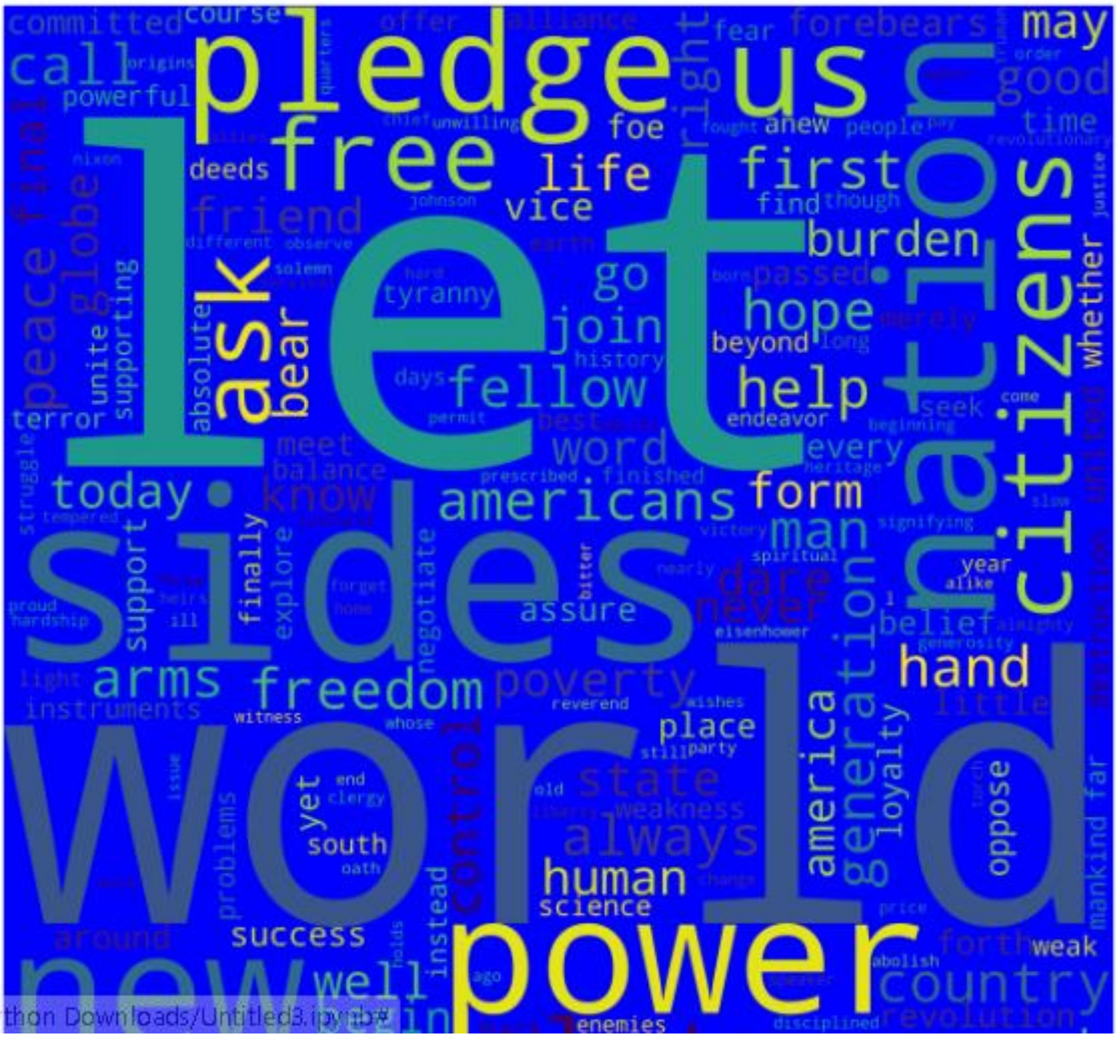
```
: # Word Cloud
  from wordcloud import WordCloud
  wordcloud = WordCloud(width = 3000, height = 3000,
                  background_color ='yellow',
                  min_font_size = 10, random_state=100).generate(wc_a)

  # plot the WordCloud image
  plt.figure(figsize = (8, 8), facecolor = None)
  plt.imshow(wordcloud)
  plt.axis("off")
  plt.xlabel('Word Cloud')
  plt.tight_layout(pad = 0)

  print("Word Cloud for Inaugural-Roosevelt")
  plt.show()
```

```
Word Cloud for Inaugural-Roosevelt
```



```python
wc_b = ' '.join(clean_all_words_k)
```

```python
# Word Cloud
from wordcloud import WordCloud
wordcloud = WordCloud(width = 3000, height = 3000,
                background_color ='blue',
                min_font_size = 10, random_state=100).generate(wc_b)

# plot the WordCloud image
plt.figure(figsize = (8, 8), facecolor = None)
plt.imshow(wordcloud)
plt.axis("off")
plt.xlabel('Word Cloud')
plt.tight_layout(pad = 0)

print("Word Cloud for Inaugural-Kennedy")
plt.show()
```



```
wc_c = ' '.join(clean_all_words_n)
```

```
# Word Cloud
from wordcloud import WordCloud
wordcloud = WordCloud(width = 3000, height = 3000,
                background_color ='black',
                min_font_size = 10, random_state=100).generate(wc_c)

# plot the WordCloud image
plt.figure(figsize = (8, 8), facecolor = None)
plt.imshow(wordcloud)
plt.axis("off")
plt.xlabel('Word Cloud')
plt.tight_layout(pad = 0)

print("Word Cloud for Inaugural-Nixon")
plt.show()
```

```
Word Cloud for Inaugural-Nixon
```