In [1]:
```
#library and data imports
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
from sklearn.naive_bayes import GaussianNB
from scipy.cluster.hierarchy import linkage, fcluster
from sklearn.cluster import DBSCAN
import geopandas as gp
import shapely
import shapefile
import plotly.figure_factory as ff
import plotly

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import plotly.figure_factory as ff
#demographics_test = pd.read_csv('demographics_test.csv')
merged_train = pd.read_csv('merged_train.csv')
X = merged_train[['State','County','FIPS','Total Population', 'Percent White, not Hispanic or Latino',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 65 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree', 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']]



Y = merged_train[['Democratic','Republican','Party']]
```

1. Partition the merged dataset into a training set and a validation set using the holdout method or the cross-validation method. How did you partition the dataset?

In [2]:
```
x_train, x_test, y_train, y_test = train_test_split(X, Y, train_size=.75, test_size=0.25, random_state=0)
```

2. Standardize the training set and the validation set.

In [3]:
```
scaler = StandardScaler()
scaler.fit(x_train[['Total Population', 'Percent White, not Hispanic or Latin
o',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic o
r Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 6
5 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree'
, 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']])
x_train_scaled = scaler.transform(x_train[['Total Population', 'Percent White,
not Hispanic or Latino',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic o
r Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 6
5 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree'
, 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']])
x_test_scaled = scaler.transform(x_test[['Total Population', 'Percent White, n
ot Hispanic or Latino',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic o
r Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 6
5 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree'
, 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']])
#print(x_train_scaled)
```

3. Build a linear regression model to predict the number of votes cast for the Democratic party in each county. Consider multiple combinations of predictor variables.Compute evaluation metrics for the validation set and report your results.

In [4]:
```
#Simple linear regression using 'Population' as predictor to predict Democrati
c votes.

from sklearn import linear_model
import numpy

n = len(x_train) #Number of observations in the training set

model = linear_model.LinearRegression()
fitted_model_D = model.fit(X = x_train_scaled[:, 0].reshape(-1, 1), y = y_trai
n['Democratic'])


predicted = fitted_model_D.predict(x_test_scaled[:, 0].reshape(-1, 1))

corr_coef = numpy.corrcoef(predicted,y_test['Democratic'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 1 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)


#print(x_train.info())
```

```
Model coefficient [74711.50206856]
R_squared value:  0.9436415220931651
Adjusted R_squared value:  0.9435784812901373
```

In [5]:
```
#Simple linear regression using 'Percent Less than High School Degree' as pred
ictor to predict Democratic votes.

model = linear_model.LinearRegression()
fitted_model_D = model.fit(X = x_train_scaled[:, 10].reshape(-1, 1), y = y_tra
in['Democratic'])

predicted = fitted_model_D.predict(x_test_scaled[:, 10].reshape(-1, 1))

corr_coef = numpy.corrcoef(predicted,y_test['Democratic'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 1 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [-8137.73810376]
R_squared value:  0.022734480001930003
Adjusted R_squared value:  0.02164134183638411
```

In [6]:
```
#Multiple linear regression using "Population", "Median Household Income" as p
redictor to predict Democratic votes.

model = linear_model.LinearRegression()
fitted_model_D = model.fit(X = x_train_scaled[:, [0,8]], y = y_train['Democrat
ic'])

predicted = fitted_model_D.predict(x_test_scaled[:, [0,8]])


corr_coef = numpy.corrcoef(predicted,y_test['Democratic'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 2 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [73067.37334453  6279.76422366]
R_squared value:  0.939337563328897
Adjusted R_squared value:  0.939201701208693
```

In [7]:
```
#Multiple linear regression using all predictor to predict Democratic votes.

model = linear_model.LinearRegression()
fitted_model_D = model.fit(X = x_train_scaled, y = y_train['Democratic'])

predicted = fitted_model_D.predict(x_test_scaled)

corr_coef = numpy.corrcoef(predicted,y_test['Democratic'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - len(x_train.columns) - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [ 69224.38708039  -3209.1591268   -1023.23488454  -6931.147
08179
   3973.74580741    194.19056985  -5299.5676761   -1853.22320472
   1471.25963216   1467.0213699    4037.7699931  -10519.02638282
   -158.13004477]
R_squared value:  0.9338361960241593
Adjusted R_squared value:  0.9326318491941099
```

In [8]:
```
#Multiple linear regression using "Population", "Median Household Income","Per
cent white, not hispanic or latino",
#"Percent Less than Bachelor's degree" as predictor to predict Democratic vote
s.

model = linear_model.LinearRegression()
fitted_model_D = model.fit(X = x_train_scaled[:, [0,1,8,11]], y = y_train['Dem
ocratic'])

predicted = fitted_model_D.predict(x_test_scaled[:, [0,1,8,11]])

corr_coef = numpy.corrcoef(predicted,y_test['Democratic'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 4 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [71012.84796525  -345.05366382  1157.04687807 -8608.1704282
6]
R_squared value:  0.947734113056962
Adjusted R_squared value:  0.9474994738338731
```

What is the best performing linear regression model? What is the performance of the model? How did you select the variables of the model?

Answer: The best performing linear Regression model is Multiple linear Regression model using "Population", "Median Household Income","Percent white, not hispanic or latino", "Percent Less than Bachelor's degree" as predictor. The model perform well with these four predictors with adjusted R square value = 0.947. Selection of the varible is consistant with Project 1 conclusion and also on present analysis as we see here the adjusted R square value decreases if we consider all variables as predictors.

Build a linear regression model to predict the number of votes cast for the Republican party in each county. Consider multiple combinations of predictor variables.Compute evaluation metrics for the validatiRepublicanon set and report your results.

In [9]:
```
#Simple linear regression using 'Population' as predictor to predict Republica
n votes.

from sklearn import linear_model
import numpy

n = len(x_train) #Number of observations in the training set

model = linear_model.LinearRegression()
fitted_model_R = model.fit(X = x_train_scaled[:, 0].reshape(-1, 1), y = y_trai
n['Republican'])

predicted = fitted_model_R.predict(x_test_scaled[:, 0].reshape(-1, 1))

corr_coef = numpy.corrcoef(predicted,y_test['Republican'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 1 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [45306.87897032]
R_squared value:  0.6718468162068597
Adjusted R_squared value:  0.6714797544800217
```

In [10]:
```
#Simple linear regression using 'Percent Less than High School Degree' as pred
ictor to predict Republican votes.

model = linear_model.LinearRegression()
fitted_model_R = model.fit(X = x_train_scaled[:, 10].reshape(-1, 1), y = y_tra
in['Republican'])

predicted = fitted_model_R.predict(x_test_scaled[:, 10].reshape(-1, 1))

corr_coef = numpy.corrcoef(predicted,y_test['Republican'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 1 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [-6381.7748349]
R_squared value:  0.03593599340559725
Adjusted R_squared value:  0.03485762203356779
```

In [11]:
```
#Multiple linear regression using "Population", "Median Household Income" as p
redictor to predict Republican votes.

model = linear_model.LinearRegression()
fitted_model_R = model.fit(X = x_train_scaled[:, [0,8]], y = y_train['Republic
an'])

predicted = fitted_model_R.predict(x_test_scaled[:, [0,8]])


corr_coef = numpy.corrcoef(predicted,y_test['Republican'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 2 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [44042.16950014  4830.56902305]
R_squared value:  0.6841236214388341
Adjusted R_squared value:  0.6834161715428404
```

In [12]:
```
#Multiple linear regression using "Population", "Median Household Income","Per
cent white, not hispanic or latino",
#"Percent Less than Bachelor's degree" as predictor to predict Republican vote
s.

model = linear_model.LinearRegression()
fitted_model_R = model.fit(X = x_train_scaled[:, [0,1,8,11]], y = y_train['Rep
ublican'])

predicted = fitted_model_R.predict(x_test_scaled[:, [0,1,8,11]])

corr_coef = numpy.corrcoef(predicted,y_test['Republican'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - 4 - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [44609.62027579  3068.87458444  3337.02252553 -2140.8068834
6]
R_squared value:  0.6837837434980034
Adjusted R_squared value:  0.6823641418975455
```

In [13]:
```
#Multiple linear regression using all predictor to predict Republican votes.

model = linear_model.LinearRegression()
fitted_model_R = model.fit(X = x_train_scaled, y = y_train['Republican'])

predicted = fitted_model_R.predict(x_test_scaled)

corr_coef = numpy.corrcoef(predicted,y_test['Republican'])[1, 0]
R_squared = corr_coef ** 2

adj_R_squared = 1 - ((1 - R_squared)*(n - 1)/(n - len(x_train.columns) - 1))

print("Model coefficient",model.coef_)
print("R_squared value: ",R_squared)
print("Adjusted R_squared value: ",adj_R_squared)
```

```
Model coefficient [45467.5097118   1769.95034533 -3141.42063749  1167.1732340
2
 -6463.65917143 -1121.73432851  -955.67013341  2580.74056065
  5910.97457236  2037.10575397  3530.42010898 -3156.11275644
 -5992.05181735]
R_squared value:  0.7239014362949742
Adjusted R_squared value:  0.7188757514038702
```

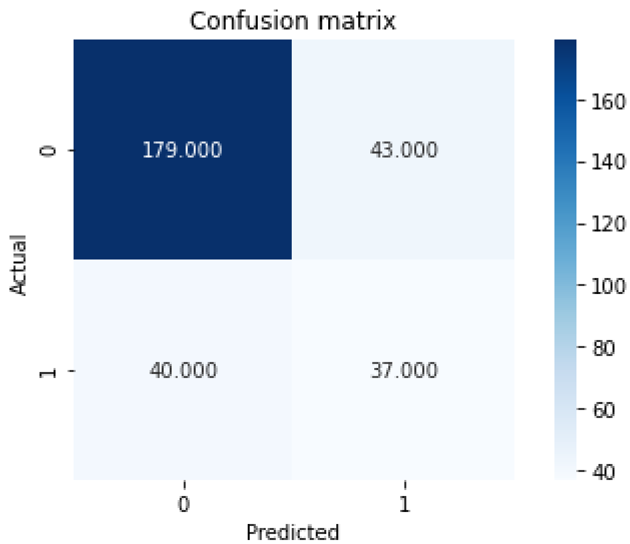
What is the best performing linear regression model? What is the performance of the model? How did you select the variables of the model?

Answer: The best performing linear Regression model while prediction Republican votes is Multiple linear Regression model using all veriables as predictor. The model does not perform too well with maximum adjusted R square value = 0.719. All the variables are selected for the model as it gives the best adjusted R square value.

Task 4

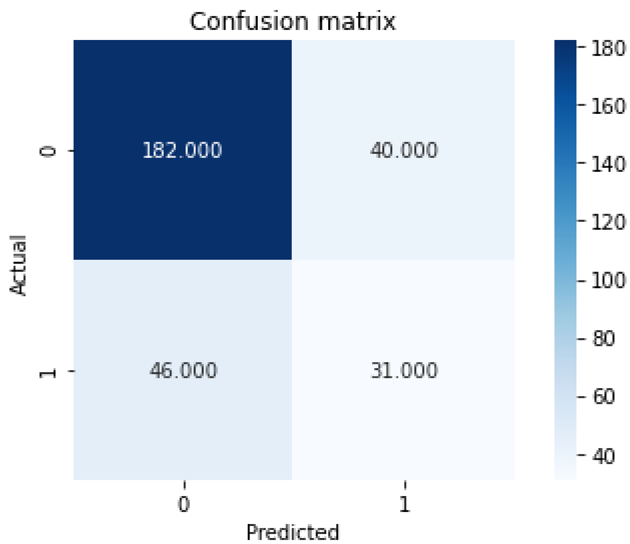
In [14]:
```
#Using decision tree classifier to classify each county as either Democratic o
r Republican

'''
First run with no random state and using variables "Total Population",
"Percent White, not Hispanic or Latino", and "Percent Rural"

'''

classifier = DecisionTreeClassifier(criterion = "entropy", random_state = 0)
print("Decision tree classifier with no random state and variables 'Total Popu
lation', 'Percent White', and 'Percent Rural'\n")
classifier.fit(x_train_scaled[:, [0,1, 12]], y_train['Party'])

print("Number of decision tree nodes: ", len(classifier.tree_.__getstate__()[
'nodes']))

#predicting Party labels for the test set using decision tree classifier
y_predicted = classifier.predict(x_test_scaled[:, [0,1, 12]])
conf_matrix = metrics.confusion_matrix(y_test['Party'], y_predicted)

# print confusion matrix
sns.heatmap(conf_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.
cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion matrix')
plt.tight_layout()


# print out Evaluation metrics for valudation set
accuracy = metrics.accuracy_score(y_test['Party'], y_predicted)
error = 1 - metrics.accuracy_score(y_test['Party'], y_predicted)
precision = metrics.precision_score(y_test['Party'], y_predicted, average = No
ne)
recall = metrics.recall_score(y_test['Party'], y_predicted, average = None)
F1_score = metrics.f1_score(y_test['Party'], y_predicted, average = None)

print("Accuracy: ", accuracy)
print("Error: ", error)
print("Precision: ", precision)
print("Recall: ", recall)
print("F1 score: ", F1_score)
print("\n")
```

```
Decision tree classifier with no random state and variables 'Total Populatio
n', 'Percent White', and 'Percent Rural'

Number of decision tree nodes:  347
Accuracy:  0.7224080267558528
Error:  0.2775919732441472
Precision:  [0.8173516 0.4625    ]
Recall:  [0.80630631 0.48051948]
F1 score:  [0.81179138 0.47133758]
```



In [15]:
```
#Decision tree continued
'''
Second run using random state and using variables 'Median Household Income',
 'Percent Unemployed',
'Percent Less than High School Degree', 'Percent Less than Bachelor's Degree'

'''
classifier = DecisionTreeClassifier(criterion = "entropy", random_state = 1)
print("Decision tree classifier with random state and variables 'Median Househ
old Income', 'Percent Unemployed', 'Percent Less than High School Degree' and
 'Percent Less than Bachelor's Degree'\n")
classifier.fit(x_train_scaled[:, [8,9, 10, 12]], y_train['Party'])

print("Number of decision tree nodes: ", len(classifier.tree_.__getstate__()[
'nodes']))

#predicting Party labels for the test set using decision tree classifier
y_predicted = classifier.predict(x_test_scaled[:, [8,9, 10, 12]])
conf_matrix = metrics.confusion_matrix(y_test['Party'], y_predicted)

# print confusion matrix
sns.heatmap(conf_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.
cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion matrix')
plt.tight_layout()


# print out Evaluation metrics for valudation set
accuracy = metrics.accuracy_score(y_test['Party'], y_predicted)
error = 1 - metrics.accuracy_score(y_test['Party'], y_predicted)
precision = metrics.precision_score(y_test['Party'], y_predicted, average = No
ne)
recall = metrics.recall_score(y_test['Party'], y_predicted, average = None)
F1_score = metrics.f1_score(y_test['Party'], y_predicted, average = None)

print("Accuracy: ", accuracy)
print("Error: ", error)
print("Precision: ", precision)
print("Recall: ", recall)
print("F1 score: ", F1_score)
print("\n")
```

```
Decision tree classifier with random state and variables 'Median Household In
come', 'Percent Unemployed', 'Percent Less than High School Degree' and 'Perc
ent Less than Bachelor's Degree'

Number of decision tree nodes:  309
Accuracy:  0.7123745819397993
Error:  0.2876254180602007
Precision:  [0.79824561 0.43661972]
Recall:  [0.81981982 0.4025974 ]
F1 score:  [0.80888889 0.41891892]
```

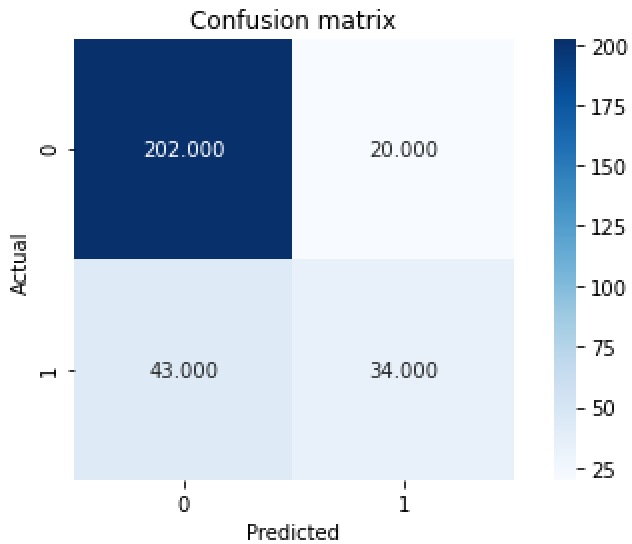


For our first run the decision tree classifier predicts party with precision of 46.25%, recall of 48.05%, and F1 Score of 46.13%. Our second run (with the random state) predicts party with precision of 43.66%, recall of 40.26%, and F1 Score of 41.89%.

The first classifier does a better job predicting party.

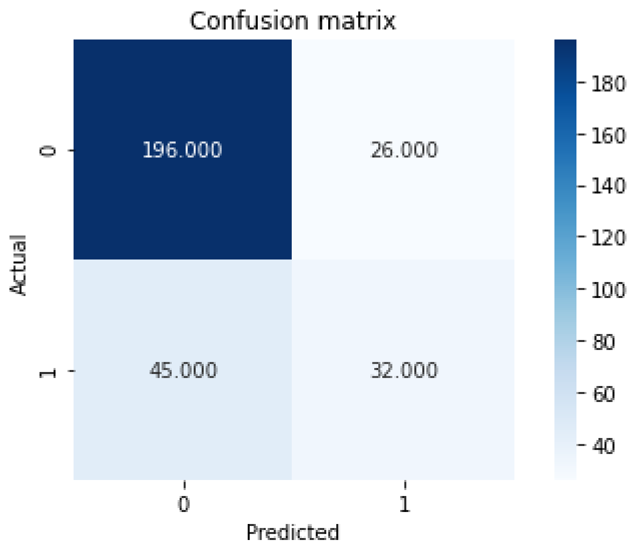
In [16]:
```
#Using a Naive Bayes classifier to classify each county as either Democratic o
r Republican


'''
First run using variables "Total Population", "Percent White, not Hispanic or
 Latino", and "Percent Rural"
There are no parameters for Gaussian NB other than var_smoothing which we opte
d out from using
'''

#no parameters for Gaussian NB
classifier = GaussianNB()
print("Naive Bayes classifier with variables 'Total Population', 'Percent Whit
e', and 'Percent Rural'\n")
classifier.fit(x_train_scaled[:, [0,1, 12]], y_train['Party'])

#predicting Party labels for the test set using NB
y_predicted = classifier.predict(x_test_scaled[:, [0,1, 12]])
conf_matrix = metrics.confusion_matrix(y_test['Party'], y_predicted)

# print confusion matrix
sns.heatmap(conf_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.
cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion matrix')
plt.tight_layout()


# print out Evaluation metrics for valudation set
accuracy = metrics.accuracy_score(y_test['Party'], y_predicted)
error = 1 - metrics.accuracy_score(y_test['Party'], y_predicted)
precision = metrics.precision_score(y_test['Party'], y_predicted, average = No
ne)
recall = metrics.recall_score(y_test['Party'], y_predicted, average = None)
F1_score = metrics.f1_score(y_test['Party'], y_predicted, average = None)

print("Accuracy: ", accuracy)
print("Error: ", error)
print("Precision: ", precision)
print("Recall: ", recall)
print("F1 score: ", F1_score)
print("\n")
```

```
Naive Bayes classifier with variables 'Total Population', 'Percent White', an
d 'Percent Rural'

Accuracy:  0.7892976588628763
Error:  0.21070234113712372
Precision:  [0.8244898  0.62962963]
Recall:  [0.90990991 0.44155844]
F1 score:  [0.86509636 0.51908397]
```



In [17]:
```
#NB continued
'''
Second run using variables 'Median Household Income', 'Percent Unemployed',
'Percent Less than High School Degree', 'Percent Less than Bachelor's Degree'
'''

#no parameters for Gaussian NB
classifier = GaussianNB()
print("Naive Bayes classifier with variables 'Median Household Income', 'Percent Unemployed', 'Percent Less than High School Degree' and 'Percent Less than Bachelor's Degree'\n")
classifier.fit(x_train_scaled[:, [8,9, 10, 12]], y_train['Party'])

#predicting Party labels for the test set using NB
y_predicted = classifier.predict(x_test_scaled[:, [8,9, 10, 12]])
conf_matrix = metrics.confusion_matrix(y_test['Party'], y_predicted)



# print confusion matrix
sns.heatmap(conf_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion matrix')
plt.tight_layout()


# print out Evaluation metrics for valudation set
accuracy = metrics.accuracy_score(y_test['Party'], y_predicted)
error = 1 - metrics.accuracy_score(y_test['Party'], y_predicted)
precision = metrics.precision_score(y_test['Party'], y_predicted, average = None)
recall = metrics.recall_score(y_test['Party'], y_predicted, average = None)
F1_score = metrics.f1_score(y_test['Party'], y_predicted, average = None)

print("Accuracy: ", accuracy)
print("Error: ", error)
print("Precision: ", precision)
print("Recall: ", recall)
print("F1 score: ", F1_score)
print("\n")
```

```
Naive Bayes classifier with variables 'Median Household Income', 'Percent Une
mployed', 'Percent Less than High School Degree' and 'Percent Less than Bache
lor's Degree'

Accuracy:  0.7625418060200669
Error:  0.23745819397993306
Precision:  [0.81327801 0.55172414]
Recall:  [0.88288288 0.41558442]
F1 score:  [0.84665227 0.47407407]
```



For our first run the Naive Bayes classifier predicts party with precision of 62.96%, recall of 44.15%, and F1 Score of 51.91%. Our second run predicts party with precision of 55.17%, recall of 41.55%, and F1 Score of 47.41%.

The first classifier does a better job predicting party.

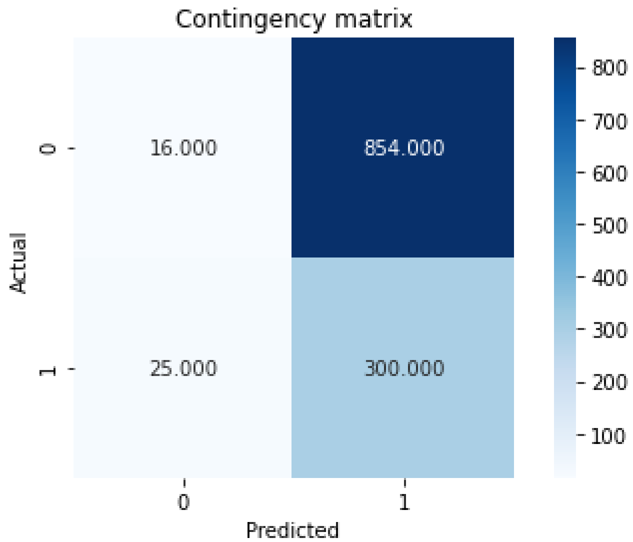
What is the best performing classification model? What is the performance of the model? How did you select the parameters of the model? How did you select the variables of the model?

Answer: The best performing classification model is the Naive Bayes classifier using variables 'Total Population', 'Percent White', and 'Percent Rural'. It had an accuracy of 78.93% and the most True Positive values out of all 4 classifiers used. It also was the best in terms of evaluation metrics with precision of 62.96%, recall of 44.15%, and F1 Score of 51.91%. The variables were selected because they gave the best indicator of a Republican county (a rural place with smaller populations and a white demographic). The model had no parameters because we did not want to use var_smoothing and there was nothing else to change.

Task 5

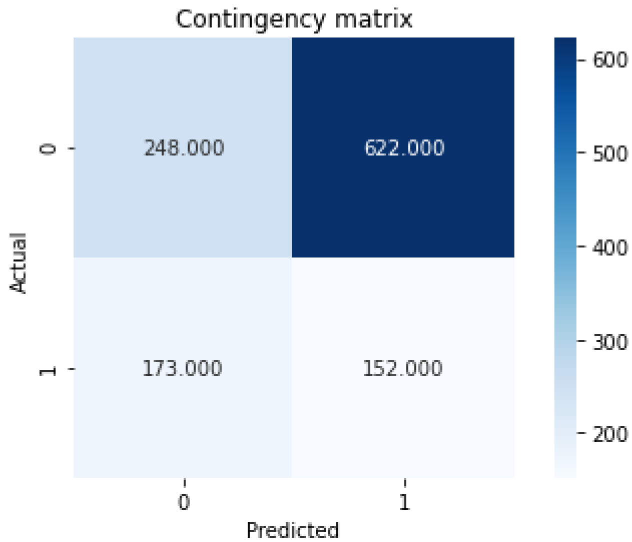
In [18]:
```
# Clustering the counties and evaluating the clusters found using Hierarchical
clustering

'''
First run: Hierarchical clustering with complete linkage and using variables
 'Percent White, not Hispanic or Latino',
'Percent Black, not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percen
t Foreign Born'
'''

X_cluster = merged_train[['Percent White, not Hispanic or Latino', 'Percent Bl
ack, not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percent Foreign B
orn']]
scaler = StandardScaler()
scaler.fit(X_cluster)
X_scaled_c = scaler.transform(X_cluster)

print("Hierarchical clustering with complete linkage, euclidean distance metri
c and using variables 'Percent White, not Hispanic or Latino','Percent Black,
 not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percent Foreign Born'
\n")


# we can use Y_Party from previous tests
clustering = linkage(X_scaled_c, method = "complete", metric = "euclidean")
clusters = fcluster(clustering, 2, criterion = 'maxclust')

# print contingency matrix

cont_matrix = metrics.cluster.contingency_matrix(Y['Party'], clusters)
sns.heatmap(cont_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.
cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Contingency matrix')
plt.tight_layout()

# print out unsupervised and supervised evaluation metrics
adjusted_rand_index = metrics.adjusted_rand_score(Y['Party'], clusters)
silhouette_coefficient = metrics.silhouette_score(X_scaled_c, clusters, metric
= "euclidean")
print("Supervised metric: ", adjusted_rand_index)
print("Unsupervised metric: ", silhouette_coefficient)
```

```
Hierarchical clustering with complete linkage, euclidean distance metric and
using variables 'Percent White, not Hispanic or Latino','Percent Black, not H
ispanic or Latino', 'Percent Hispanic or Latino', 'Percent Foreign Born'

Supervised metric:  0.05057355502877359
Unsupervised metric:  0.64412187209008
```



In [19]:
```
# Hierarchical clustering continued

'''
Second run: Hierarchical clustering with complete linkage and using variables
 'Percent Female',
'Percent Age 29 and Under', 'Percent Age 65 and Older', 'Median Household Inco
me', 'Percent Unemployed'
'''

X_cluster = merged_train[['Percent Female', 'Percent Age 29 and Under', 'Perce
nt Age 65 and Older', 'Median Household Income', 'Percent Unemployed']]
scaler = StandardScaler()
scaler.fit(X_cluster)
X_scaled_c = scaler.transform(X_cluster)

print("Hierarchical clustering with complete linkage, jaccard distance metric
 and using variables 'Percent Female', 'Percent Age 29 and Under', 'Percent Ag
e 65 and Older', 'Median Household Income', 'Percent Unemployed'\n")


# we can use Y_Party from previous tests
clustering = linkage(X_scaled_c, method = "complete", metric = "cosine")
clusters = fcluster(clustering, 2, criterion = 'maxclust')

# print contingency matrix
cont_matrix = metrics.cluster.contingency_matrix(Y['Party'], clusters)
sns.heatmap(cont_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.
cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Contingency matrix')
plt.tight_layout()

# print out unsupervised and supervised evaluation metrics
adjusted_rand_index = metrics.adjusted_rand_score(Y['Party'], clusters)
silhouette_coefficient = metrics.silhouette_score(X_scaled_c, clusters, metric
= "cosine")
print("Supervised metric: ", adjusted_rand_index)
print("Unsupervised metric: ", silhouette_coefficient)
```

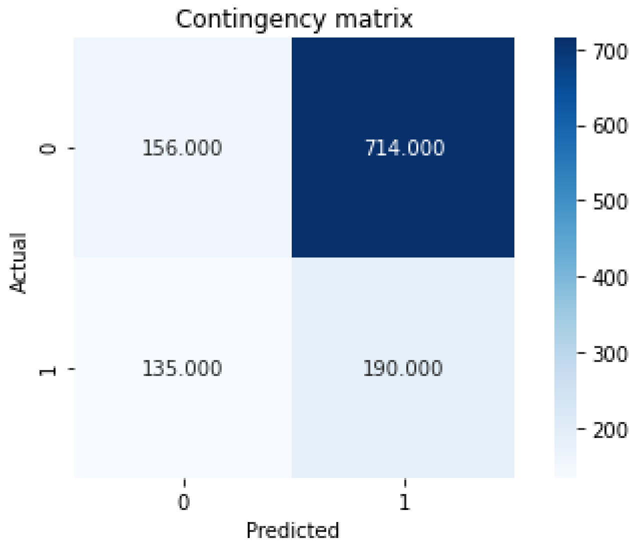
```
Hierarchical clustering with complete linkage, jaccard distance metric and us
ing variables 'Percent Female', 'Percent Age 29 and Under', 'Percent Age 65 a
nd Older', 'Median Household Income', 'Percent Unemployed'

Supervised metric:  0.09223396339355758
Unsupervised metric:  0.37214725289194917
```



The first Heirarchical clustering had the best performance with an unsupervised metric of 64.41%

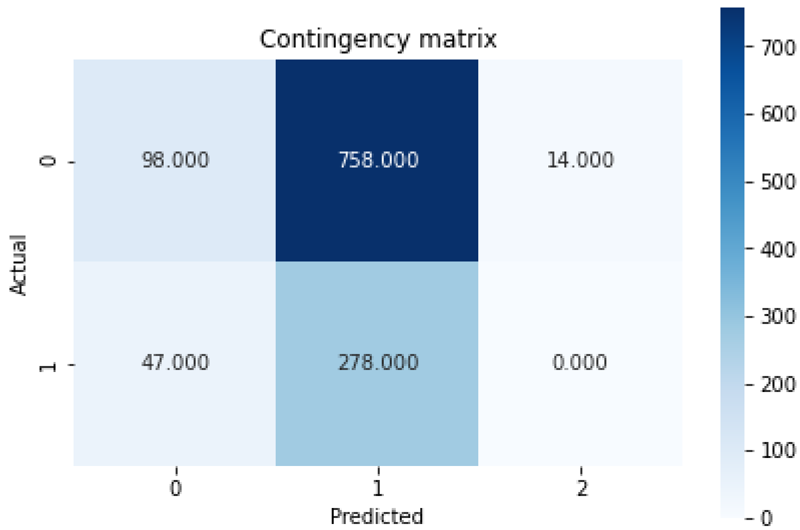
In [20]:
```
# Clustering using DBScan

'''
First run: DBSCAN with eps = .5, min_samples = 15 and using variables 'Percent
White, not Hispanic or Latino',
'Percent Black, not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percen
t Foreign Born'
'''

X_cluster = merged_train[['Percent White, not Hispanic or Latino', 'Percent Bl
ack, not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percent Foreign B
orn']]
scaler = StandardScaler()
scaler.fit(X_cluster)
X_scaled_c = scaler.transform(X_cluster)

print("DBSCAN clustering with eps = .5, min_samples = 15 and using variables
 'Percent White, not Hispanic or Latino','Percent Black, not Hispanic or Latin
o', 'Percent Hispanic or Latino', 'Percent Foreign Born'\n")

clustering = DBSCAN(eps = .5, min_samples = 15, metric = "euclidean").fit(X_sc
aled_c)
clusters = clustering.labels_

# print contingency matrix
cont_matrix = metrics.cluster.contingency_matrix(Y['Party'], clusters)
sns.heatmap(cont_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.
cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Contingency matrix')
plt.tight_layout()

# print out unsupervised and supervised evaluation metrics
adjusted_rand_index = metrics.adjusted_rand_score(Y['Party'], clusters)
silhouette_coefficient = metrics.silhouette_score(X_scaled_c, clusters, metric
= "euclidean")
print("Supervised metric: ", adjusted_rand_index)
print("Unsupervised metric: ", silhouette_coefficient)
```

```
DBSCAN clustering with eps = .5, min_samples = 15 and using variables 'Percen
t White, not Hispanic or Latino','Percent Black, not Hispanic or Latino', 'Pe
rcent Hispanic or Latino', 'Percent Foreign Born'

Supervised metric:  0.12908114450483138
Unsupervised metric:  0.5786850122739186
```



In [21]:
```
# DBSCAN continued

'''
Second run: DBSCAN with eps = 1, min_samples = 10 and using variables 'Percent Female',
'Percent Age 29 and Under', 'Percent Age 65 and Older', 'Median Household Income', 'Percent Unemployed'
'''


X_cluster = merged_train[['Percent Female', 'Percent Age 29 and Under', 'Percent Age 65 and Older', 'Median Household Income', 'Percent Unemployed']]
scaler = StandardScaler()
scaler.fit(X_cluster)
X_scaled_c = scaler.transform(X_cluster)

print("DBSCAN clustering with eps = 1, min_samples = 10 and using variables 'Percent Female', 'Percent Age 29 and Under', 'Percent Age 65 and Older', 'Median Household Income', 'Percent Unemployed'\n")


clustering = DBSCAN(eps = 1, min_samples = 10, metric = "euclidean").fit(X_scaled_c)
clusters = clustering.labels_

# print contingency matrix
cont_matrix = metrics.cluster.contingency_matrix(Y['Party'], clusters)
sns.heatmap(cont_matrix, annot = True, fmt = ".3f", square = True, cmap = plt.cm.Blues)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Contingency matrix')
plt.tight_layout()

# print out unsupervised and supervised evaluation metrics
adjusted_rand_index = metrics.adjusted_rand_score(Y['Party'], clusters)
silhouette_coefficient = metrics.silhouette_score(X_scaled_c, clusters, metric = "euclidean")
print("Supervised metric: ", adjusted_rand_index)
print("Unsupervised metric: ", silhouette_coefficient)
```

```
DBSCAN clustering with eps = 1, min_samples = 10 and using variables 'Percent
Female', 'Percent Age 29 and Under', 'Percent Age 65 and Older', 'Median Hous
ehold Income', 'Percent Unemployed'

Supervised metric:  0.008381757165266535
Unsupervised metric:  0.2213462398681104
```

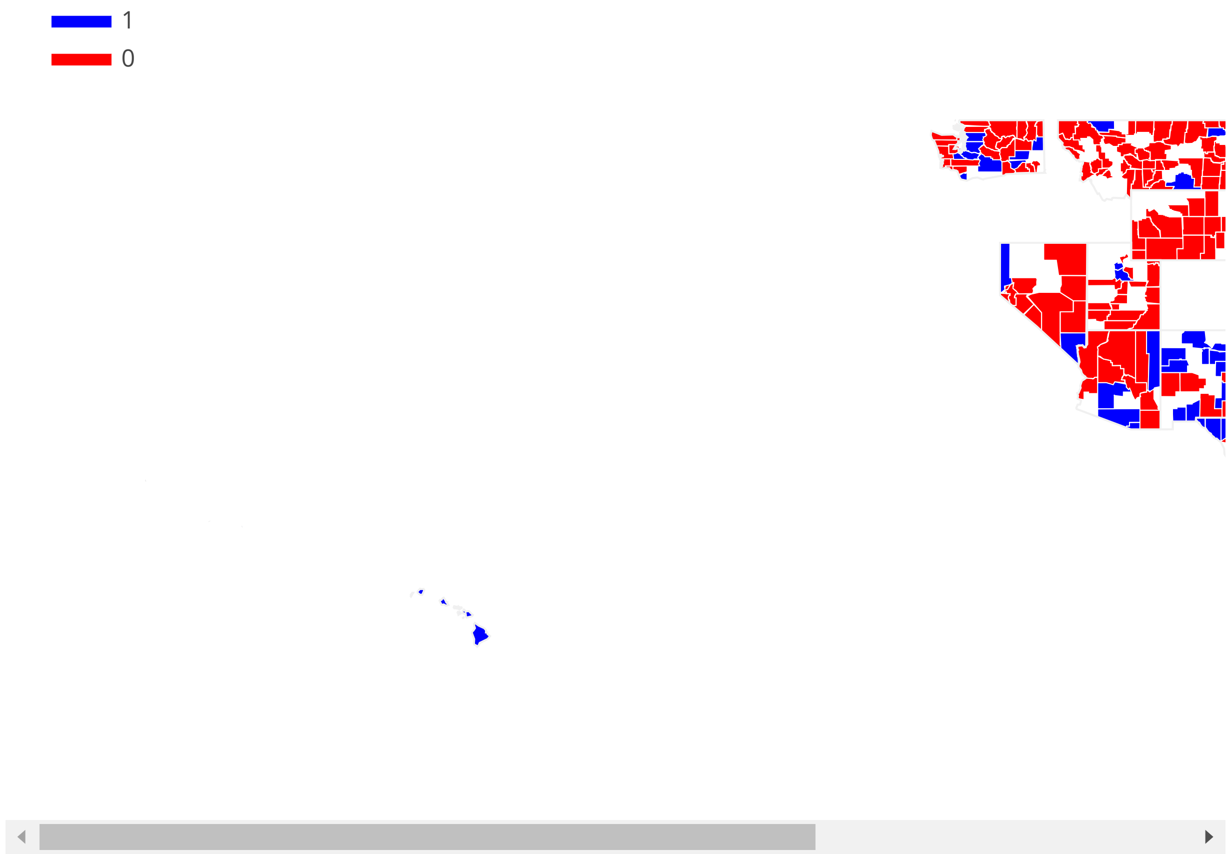


The first DBSCAN clustering had the better performance with an unsupervised metric of 57.87%.

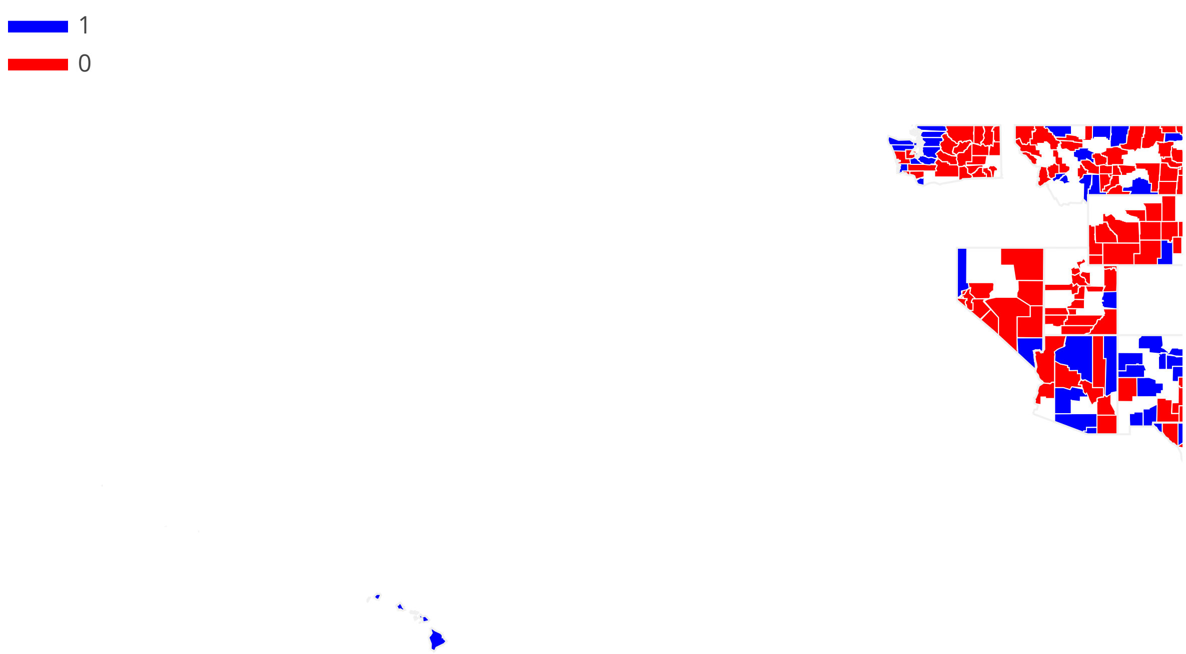
What is the best performing clustering model? What is the performance of the model? How did you select the parameters of model? How did you select the variables of the model?

Answer: The best performing model was Heirarchical clustering with complete linkage and variables 'Percent White, not Hispanic or Latino', 'Percent Black, not Hispanic or Latino', 'Percent Hispanic or Latino', 'Percent Foreign Born'. It had an unsupervised metric of 64.41% and supervised metric of 5.06. The parameters selected were complete linkage and euclidean distance because single linkage did not accurately cluster the data. The variables were selected because they all describe the demographic of a given county and provide a rough estimation of diversity.

In [22]:
```
#no parameters for Gaussian NB
classifier = GaussianNB()
classifier.fit(x_train_scaled[:, [0,1, 12]],y_train['Party'] )
scaler.fit(x_train[['Total Population', 'Percent White, not Hispanic or Latin
o',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic o
r Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 6
5 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree'
, 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']])

merged_train_scaled = scaler.transform(merged_train[['Total Population', 'Perc
ent White, not Hispanic or Latino',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic o
r Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 6
5 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree'
, 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']])

#predicting Party labels for the test set using NB
y_predicted = classifier.predict(merged_train_scaled[:, [0,1, 12]])
merged_train['PartyPredicted'] = y_predicted
states = list(set(merged_train['State'].tolist()))
values = merged_train['PartyPredicted'].tolist()
fips = merged_train['FIPS'].tolist()
colorscale = ['rgb(255.0, 0.0, 0.0)', 'rgb(0.0, 0.0, 255.0)']

fig = ff.create_choropleth(
    fips=fips, values=values,colorscale=colorscale,
    scope=states, county_outline={'color': 'rgb(255,255,255)', 'width': 0.5},
    legend_title='Party by County Predicted'

)
fig.update_layout(
    legend_x = 0,
    annotations = {'x': -0.12, 'xanchor': 'left'}
)

fig.layout.template = None
fig.show()
```


1
0

In [23]:
```
states = list(set(merged_train['State'].tolist()))
values = merged_train['Party'].tolist()
fips = merged_train['FIPS'].tolist()
colorscale = ['rgb(255.0, 0.0, 0.0)', 'rgb(0.0, 0.0, 255.0)']

fig = ff.create_choropleth(
    fips=fips, values=values,colorscale=colorscale,
    scope=states, county_outline={'color': 'rgb(255,255,255)', 'width': 0.5},
    legend_title='Party by County'

)
fig.update_layout(
    legend_x = 0,
    annotations = {'x': -0.12, 'xanchor': 'left'}
)

fig.layout.template = None
fig.show()
```



Task 7: Use your best performing regression and classification models to predict the number of votes cast for the Democratic party in each county, the number of votes cast for the Republican party in each county, and the party (Democratic or Republican) of each county for the test dataset (demographics_test.csv). Save the output in a single CSV file. For the expected format of the output, see sample_output.csv.

In [24]:
```
election_demographic = pd.read_csv('demographics_test.csv')
election_demographic.head()


scaler = StandardScaler()
scaler.fit(x_train[['Total Population', 'Percent White, not Hispanic or Latin
o',
                  'Percent Black, not Hispanic or Latino', 'Percent Hispanic o
r Latino', 'Percent Foreign Born',
                  'Percent Female', 'Percent Age 29 and Under', 'Percent Age 6
5 and Older', 'Median Household Income',
                  'Percent Unemployed', 'Percent Less than High School Degree'
, 'Percent Less than Bachelor\'s Degree',
                  'Percent Rural']])


x_test = election_demographic.select_dtypes(include = [numpy.int64,numpy.float
64])
x_test = x_test.iloc[:,1:14]
x_test.info()
x_test_scaled = scaler.transform(x_test)
x_test_scaled_df = pd.DataFrame(x_test_scaled,index = x_test.index,columns=x_t
est.columns)

y_pred_democratic = fitted_model_D.predict(x_test_scaled_df[['Total Populatio
n','Median Household Income',
                                                             'Percent White, not
Hispanic or Latino',
                                                             'Percent Less than
 Bachelor\'s Degree']])
election_demographic['Democratic'] = y_pred_democratic


# #"Population", "Median Household Income","Percent white, not hispanic or lat
ino", "Percent Less than Bachelor's degree"
y_pred_republican = fitted_model_R.predict(x_test_scaled_df)

election_demographic['Republican'] = y_pred_republican

election_demographic.head()


classifier = GaussianNB()
classifier.fit(x_train_scaled[:, [0,1, 12]],y_train['Party'] )
y_predicted = classifier.predict(x_test_scaled_df[['Total Population','Percent
White, not Hispanic or Latino', 'Percent Rural']])
election_demographic['Party'] = y_predicted


sample_output = election_demographic[['State','County','Democratic','Republica
n','Party']]
sample_output.head()

numaric_data = sample_output._get_numeric_data()
numaric_data[numaric_data < 0] = 0
sample_output.head()
```

```
sample_output.to_csv("output.csv")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 13 columns):
 #   Column                                 Non-Null Count  Dtype  
---  ------                                 --------------  -----  
 0   Total Population                       400 non-null    int64  
 1   Percent White, not Hispanic or Latino  400 non-null    float64
 2   Percent Black, not Hispanic or Latino  400 non-null    float64
 3   Percent Hispanic or Latino             400 non-null    float64
 4   Percent Foreign Born                   400 non-null    float64
 5   Percent Female                         400 non-null    float64
 6   Percent Age 29 and Under               400 non-null    float64
 7   Percent Age 65 and Older               400 non-null    float64
 8   Median Household Income                400 non-null    int64  
 9   Percent Unemployed                     400 non-null    float64
 10  Percent Less than High School Degree   400 non-null    float64
 11  Percent Less than Bachelor's Degree    400 non-null    float64
 12  Percent Rural                          400 non-null    float64
dtypes: float64(11), int64(2)
memory usage: 40.8 KB
```

In [ ]:

In [ ]: