CS 4780 Final Project Student Template

December 14, 2020

CS 4780/5780 Final Project:

Election Result Prediction for US Counties

Names: Peter Wu, Elena Stoeva, Sheel Yerneni

Netids: hw399, ess243, scy33

Introduction:

The final project is about conducting a real-world machine learning project on your own, with everything that is involved. Unlike in the programming projects 1-5, where we gave you all the scaffolding and you just filled in the blanks, you now start from scratch. The programming project provide templates for how to do this, and the most recent video lectures summarize some of the tricks you will need (e.g. feature normalization, feature construction). So, this final project brings realism to how you will use machine learning in the real world.

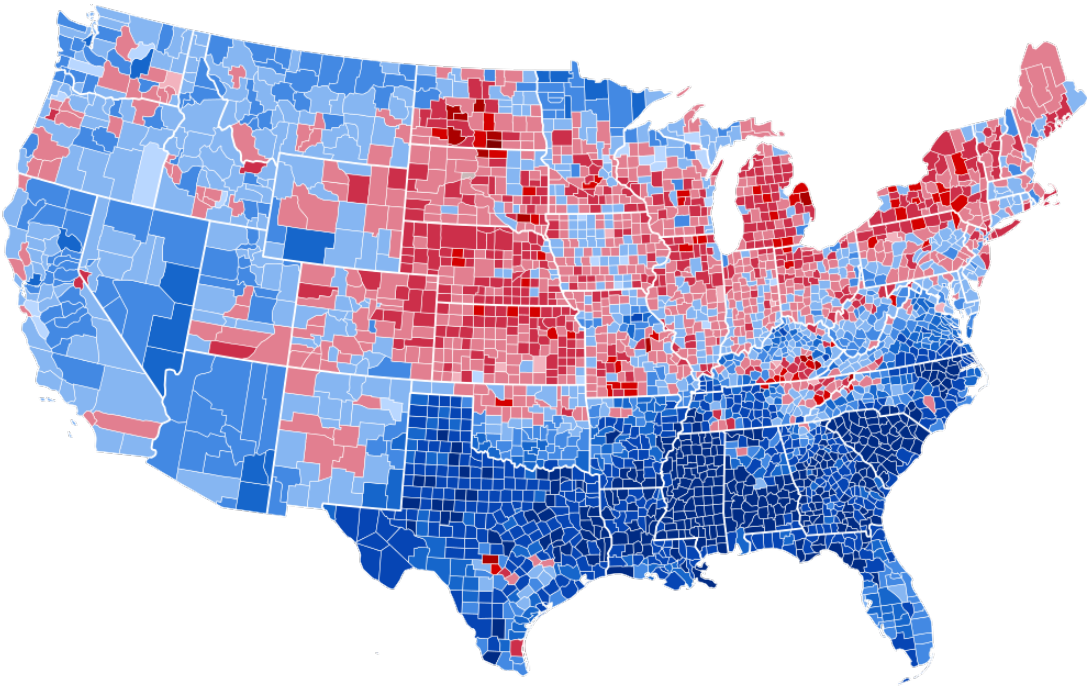
The task you will work on is forecasting election results. Economic and sociological factors have been widely used when making predictions on the voting results of US elections. Economic and sociological factors vary a lot among counties in the United States. In addition, as you may observe from the election map of recent elections, neighbor counties show similar patterns in terms of the voting results. In this project you will bring the power of machine learning to make predictions for the county-level election results using Economic and sociological factors and the geographic structure of US counties.

Your Task:

Plase read the project description PDF file carefully and make sure you write your code and answers to all the questions in this Jupyter Notebook. Your answers to the questions are a large portion of your grade for this final project. Please import the packages in this notebook and cite any references you used as mentioned in the project description. You need to print this entire Jupyter Notebook as a PDF file and submit to Gradescope and also submit the ipynb runnable version to Canvas for us to run.

Due Date:

The final project dataset and template jupyter notebook will be due on December 15th . Note that no late submissions will be accepted and you cannot use any of your unused slip days before.



Part 1: Basics

1.1 Import:

Please import necessary packages to use. Note that learning and using packages are recommended but not required for this project. Some official tutorial for suggested packacges includes:

https://scikit-learn.org/stable/tutorial/basic/tutorial.html

https://pytorch.org/tutorials/

https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html

[1]:
```
import os
import pandas as pd
import numpy as np
import locale
from sklearn.model_selection import KFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
# TODO
```

1.2 Weighted Accuracy:

Since our dataset labels are heavily biased, you need to use the following function to compute weighted accuracy throughout your training and validation process and we use this for testing on Kaggle.

[2]:
```
def weighted_accuracy(pred, true):
    assert(len(pred) == len(true))
    num_labels = len(true)
    num_pos = sum(true)
    num_neg = num_labels - num_pos
    frac_pos = num_pos/num_labels
    weight_pos = 1/frac_pos
    weight_neg = 1/(1-frac_pos)
    num_pos_correct = 0
    num_neg_correct = 0
    for pred_i, true_i in zip(pred, true):
        num_pos_correct += (pred_i == true_i and true_i == 1)
        num_neg_correct += (pred_i == true_i and true_i == 0)
    weighted_accuracy = ((weight_pos * num_pos_correct)
                         + (weight_neg * num_neg_correct))/((weight_pos *␣
 ↪num_pos) + (weight_neg * num_neg))
    return weighted_accuracy
```

Part 2: Baseline Solution

Note that your code should be commented well and in part 2.4 you can refer to your comments. (e.g. # Here is SVM, # Here is validation for SVM, etc). Also, we recommend that you do not to use 2012 dataset and the graph dataset to reach the baseline accuracy for 68% in this part, a basic solution with only 2016 dataset and reasonable model selection will be enough, it will be great if you explore thee graph and possibly 2012 dataset in Part 3.

2.1 Preprocessing and Feature Extraction:

Given the training dataset and graph information, you need to correctly preprocess the dataset (e.g. feature normalization). For baseline solution in this part, you might not need to introduce extra features to reach the baseline test accuracy.

[3]:
```
# Load Data
train = pd.read_csv("train_2016.csv")

# Create Output Column
output = train["DEM"] > train["GOP"]
train["Output"] = output.astype(int)

# Save mean and std of  median income, migrarate, birthrate, deathrate and␣
 ↪bachelorrate
train["MedianIncome"] = train["MedianIncome"].str.replace(',', '').astype(float)
mean_MI = train["MedianIncome"].mean()
sd_MI = train["MedianIncome"].std()

train["MigraRate"] = train["MigraRate"].astype(float)
mean_MR = train["MigraRate"].mean()
sd_MR = train["MigraRate"].std()
```

```
train["BirthRate"] = train["BirthRate"].astype(float)
mean_BR = train["BirthRate"].mean()
sd_BR = train["BirthRate"].std()

train["DeathRate"] = train["DeathRate"].astype(float)
mean_DR = train["DeathRate"].mean()
sd_DR = train["DeathRate"].std()

train["BachelorRate"] = train["BachelorRate"].astype(float)
mean_BC = train["BachelorRate"].mean()
sd_BC = train["BachelorRate"].std()

train["UnemploymentRate"] = train["UnemploymentRate"].astype(float)
mean_UR = train["UnemploymentRate"].mean()
sd_UR = train["UnemploymentRate"].std()

# Standardize median income, migrarate, birthrate, deathrate and bachelorrate
train["MedianIncome"] = (train["MedianIncome"] - mean_MI)/sd_MI
train["MigraRate"] = (train["MigraRate"] - mean_MR)/sd_MR
train["BirthRate"] = (train["BirthRate"] - mean_BR)/sd_BR
train["DeathRate"] = (train["DeathRate"] - mean_DR)/sd_DR
train["BachelorRate"] = (train["BachelorRate"] - mean_BC)/sd_BC
train["UnemploymentRate"] = (train["UnemploymentRate"] - mean_UR)/sd_UR

# Function to standardize the test data in same manner as training data
def mod_test(test, mean_MI, sd_MI, mean_MR, sd_MR, mean_BR, sd_BR, mean_DR,␣
↪sd_DR, mean_BC, sd_BC, mean_UR, sd_UR):
    test["MedianIncome"] = test["MedianIncome"].str.replace(',', '').
↪astype(float)
    test["MigraRate"] = test["MigraRate"].astype(float)
    test["BirthRate"] = test["BirthRate"].astype(float)
    test["DeathRate"] = test["DeathRate"].astype(float)
    test["BachelorRate"] = test["BachelorRate"].astype(float)
    test["UnemploymentRate"] = test["UnemploymentRate"].astype(float)

    test["MedianIncome"] = (test["MedianIncome"] - mean_MI)/sd_MI
    test["MigraRate"] = (test["MigraRate"] - mean_MR)/sd_MR
    test["BirthRate"] = (test["BirthRate"] - mean_BR)/sd_BR
    test["DeathRate"] = (test["DeathRate"] - mean_DR)/sd_DR
    test["BachelorRate"] = (test["BachelorRate"] - mean_BC)/sd_BC
    test["UnemploymentRate"] = (test["UnemploymentRate"] - mean_UR)/sd_UR

    return test
```

2.2 Use At Least Two Training Algorithms from class:

You need to use at least two training algorithms from class. You can use your code from previous projects or any packages you imported in part 1.1.

[4]:
```
# Make sure you comment your code clearly and you may refer to these comments␣
↪in the part 2.4

#Approach 1: Random Forest (extension of decisions trees learned in class)

def rf(train_x, test_x, train_y, test_y, n_tree, max_fea, min_leaf):
    ran_f = RandomForestClassifier(n_estimators = n_tree, max_features =␣
↪max_fea, min_samples_leaf = min_leaf,
                                   random_state=0)
    ran_f.fit(train_x, train_y)
    pred = ran_f.predict(test_x)
    return weighted_accuracy(pred, test_y)

#Approach 2: SVM with a regularization parameter and a kernel
# This function creates an SVM model given the training data and the parameters␣
↪c and ker
# Then it predicts the examples in test_x and computes the accuracy given their␣
↪correct labels test_y
def svm(train_x, test_x, train_y, test_y, c, ker):
    SVM_Model = SVC(C = c, kernel = ker, gamma='scale')
    SVM_Model.fit(train_x, train_y)
    pred = SVM_Model.predict(test_x)
    return weighted_accuracy(pred, test_y)
```

2.3 Training, Validation and Model Selection:

You need to split your data to a training set and validation set or performing a cross-validation for model selection.

[5]:
```
# Five Fold Cross-validation for hyper-parameter selection
kf = KFold(n_splits=5, shuffle = True, random_state = 0)
kf.get_n_splits(train)

# Testing all combinations for hyper-parameters: number of trees, max features␣
↪and minimum leaf split
no_tree = [300, 400, 500]
max_fea = [3, 4, 5]
min_leaf = [10, 25, 40]
fea_ls = []
acc_ls = []

# For each combination, record average of five-fold cross validation accuracy␣
↪in acc_ls
for nt in no_tree:
    for mf in max_fea:
        for ml in min_leaf:
            print((nt, mf, ml))
```

```
            fea_ls.append((nt, mf, ml))
            cv_acc = []
            for train_ind, test_ind in kf.split(train):
                X_train, X_test = train.iloc[:,4:10].iloc[train_ind,:], train.
↪iloc[:,4:10].iloc[test_ind,:]
                y_train, y_test = train.iloc[:,10][train_ind], train.iloc[:
↪,10][test_ind]
                pred_acc = rf(X_train, X_test, y_train, y_test, nt, mf, ml)
                cv_acc.append(pred_acc)
            acc_ls.append(sum(cv_acc)/len(cv_acc))
```

```
(300, 3, 10)
(300, 3, 25)
(300, 3, 40)
(300, 4, 10)
(300, 4, 25)
(300, 4, 40)
(300, 5, 10)
(300, 5, 25)
(300, 5, 40)
(400, 3, 10)
(400, 3, 25)
(400, 3, 40)
(400, 4, 10)
(400, 4, 25)
(400, 4, 40)
(400, 5, 10)
(400, 5, 25)
(400, 5, 40)
(500, 3, 10)
(500, 3, 25)
(500, 3, 40)
(500, 4, 10)
(500, 4, 25)
(500, 4, 40)
(500, 5, 10)
(500, 5, 25)
(500, 5, 40)
```

[6]:
```
# Select max features
print(max(acc_ls))
ind = acc_ls.index(max(acc_ls))
print(fea_ls[ind])
nt = fea_ls[ind][0]
mf = fea_ls[ind][1]
ml = fea_ls[ind][2]
```

```
final_rf = RandomForestClassifier(n_estimators = nt, max_features = mf,␣
↪min_samples_leaf = ml,
                                  random_state=0)
final_rf.fit(train.iloc[:,4:10], train.iloc[:,10])
```

```
0.7047923052462925
(400, 5, 10)
```

[6]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features=5, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=10, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=400,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False)
```

[7]:
```
# Training, Validation, and Model Selection for approach 2

# Five Fold Cross-validation for hyper-parameter selection
kf = KFold(n_splits=5, shuffle = True, random_state = 0)
kf.get_n_splits(train)

# We test polynomial, Radial basis function, and sigmoid kernel
# We test C values from 3 to 7
kernel = ['poly', 'rbf', 'sigmoid']
C= range(3, 8)

# Accumulate the accuracies and the corresponding pairs of parameters
accuracy = []
parameters=[]

# For each pair of a kernel and C, we record average of five-fold cross␣
↪validation accuracy
for kr in kernel:
    for c in C:
        print(kr,c)
        cv_acc = []
        parameters.append((kr,c))
        for train_ind, test_ind in kf.split(train):
            X_train, X_test = train.iloc[:,4:10].iloc[train_ind,:], train.iloc[:
↪,4:10].iloc[test_ind,:]
            y_train, y_test = train.iloc[:,10][train_ind], train.iloc[:
↪,10][test_ind]
            pred_acc = svm(X_train, X_test, y_train, y_test, c, kr)
            cv_acc.append(pred_acc)
        accuracy.append(sum(cv_acc)/len(cv_acc))
```

```
poly 3
poly 4
poly 5
poly 6
poly 7
rbf 3
rbf 4
rbf 5
rbf 6
rbf 7
sigmoid 3
sigmoid 4
sigmoid 5
sigmoid 6
sigmoid 7
```

[8]:
```
# Select kernel and C that maximize the accuracy of the SVM model

print(max(accuracy))
ind = accuracy.index(max(accuracy))
print(parameters[ind])
ker = parameters[ind][0]
c = parameters[ind][1]

# Based on the selected parameters, create an SVM model that will be used to␣
 ↪predict the election results
final_svm = SVC(C = c, kernel = ker, gamma='scale')
final_svm.fit(train.iloc[:,4:10], train.iloc[:,10])
```

```
0.7139413251357022
('rbf', 6)
```

[8]:
```
SVC(C=6, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma='scale', kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

2.4 Explanation in Words:

You need to answer the following questions in the markdown cell after this cell:

2.4.1 How did you preprocess the dataset and features?

2.4.2 Which two learning methods from class did you choose and why did you made the choices?

2.4.3 How did you do the model selection?

2.4.4 Does the test performance reach a given baseline 68% performanc? (Please include a screenshot of Kaggle Submission)

2.4.1 We loaded the data from the train_2016.csv file into a data frame, train, using the Pandas

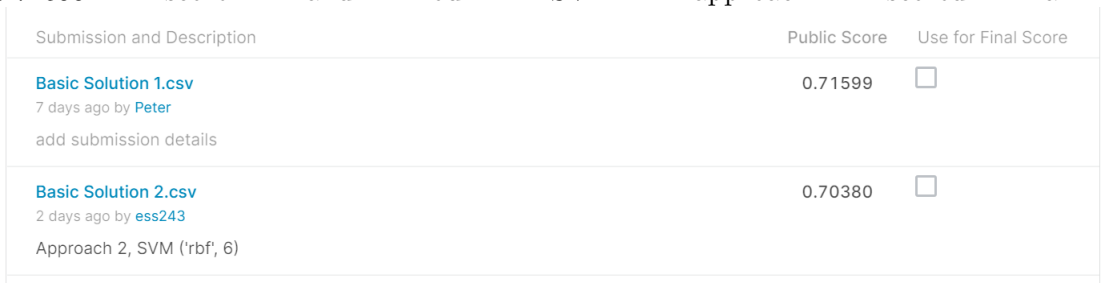
library. We created an additional output column in our data frame that has a 1 if the democratic party had a win over the grand old party (republican party) in that and a 0 otherwise using a simple comparison operator for each index. We have 6 attributes for each county, median income, migration rate, birth rate, death rate, bachelor rate and unemployment rate. We standardize each of these columns by calculating the mean and standard deviation of each column and set the column values to their Z values using the formula: z = (x-)/ which is essentially how many standard deviations from the mean the value is. Now we have put these 6 different variables on the same scale which is critical for comparing values between different types of variables. For instance, if we didn't standardize our data, the median income would be significantly higher than any of the rates in that it is not a percentage, but this finding hinders finding relevant relationships within our data. We store the mean and standard deviation for each variable to use for standardizing test set. Then we set the test set to type float so we can do the same standardization on our test set to ensure our hypothesis trained on the training data will be consistent with the test set.

2.4.2 For our first approach we chose the Random Forest method which is an extension of the decision tree method we learned in class. The Random Forest method has advantage over decision tree as Random Forest is a bagging algorithm that reduces overfitting through using an ensemble of trees. We implemented it using the Scikit-learn library. We chose this because as a decision tree algorithm, it is less influenced by outliers. It also does not make assumptions about the underlying distribution of our data which is useful given that we do not know anything about the distribution of our data across counties. It is also good for reducing overfitting given that we don't know whether median income, migration rate, birth rate, death rate and bachelor rate are useful for determining the majority party, thus our random forest can avoid fitting to features that aren't useful. Since we are only interested in the 0,1 voting classification and not the degree to which each feature influences the voting outcome, we chose Random Forest over other regression models. For our second approach we chose the SVM method with regularization and a kernel. We also implemented it using the Scikit-learn library. We chose this method as our secondary approach because it is a good complement to our Random Forest method given that it uses all the features and accounts for the degree to which the outcome is influenced by each feature. Furthermore, it works easily with this 2-class problem and it allows for a flexible decision boundary, maximizing the margin to distinguish red and blue counties while also allowing for counties that overlap with others in features but report different outcomes. And we can find an appropriate C value to reduce the influence of these outliers.

2.4.3 For our Random Forest model selection, we used a five fold cross-validation for hyper parameter selection. We split our training data into 5 groups such that we would train a hypothesis on 80% of our dataset and validate it on the other 20% of our dataset. We do this such that each group takes turns as a validation set. Furthermore, when training on each group, we test different numbers of trees: 300, 400, 500. For overfitting prevention implementation we test various early stopping measures setting the max features to: 3, 4, 5 and test different minimum sample requirements for creating a leaf node: 10, 25, 40. We use a triple nested loop to iterate through all the different combinations of these Random Forest parameters and add the accuracy of each combination in each sample split to an accuracy list as well as the current random forest parameters used to a features list. After running through all our groups, we choose the combination of Random Forest features that performs with the maximum accuracy by checking our accuracy list for the maximum value and get the index of this value. Then we look up the values of this index in our parameters list to select our best Random forest features for this cross validation. Then we train a new Random Forest hypothesis using these parameters on the entire training set (all 5 groups) to get our final hypothesis. We do a similar five fold cross-validation for our SVM approach, but instead of tree

parameters, we test different kernel functions: polynomial, rbf, sigmoid with different C values: 3, 4, 5, 6, 7.

2.4.4 Yes, our Random Forest approach performed with a 0.71599 score and our SVM approach scored a 0.70380.



Part 3: Creative Solution

3.1 Open-ended Code:

You may follow the steps in part 2 again but making innovative changes like creating new features, using new training algorithms, etc. Make sure you explain everything clearly in part 3.2. Note that reaching the 75% creative baseline is only a small portion of this part. Any creative ideas will receive most points as long as they are reasonable and clearly explained.

[9]:
```
# Making Use of Neighboring FIPS file, creates a dictionary mapping each county␣
↪to set of all of its neighbors
neighbor_data = pd.read_csv("graph.csv")
neigh_dic = {}
for i in range(neighbor_data.shape[0]):
    # If neighbor is itself, pass
    if neighbor_data.iloc[i,0] == neighbor_data.iloc[i,1]:
        continue
    if neighbor_data.iloc[i,0] in neigh_dic:
        neigh_dic[neighbor_data.iloc[i, 0]].append(neighbor_data.iloc[i, 1])
    else:
        neigh_dic[neighbor_data.iloc[i, 0]] = [neighbor_data.iloc[i, 1]]
    if neighbor_data.iloc[i,1] in neigh_dic:
        neigh_dic[neighbor_data.iloc[i, 1]].append(neighbor_data.iloc[i, 0])
    else:
        neigh_dic[neighbor_data.iloc[i, 1]] = [neighbor_data.iloc[i, 0]]
```

[10]:
```
# Computes KNN algorithm using gaussian kernel
import scipy
import random
# Return gaussian kernel value between train inputs and test inputs
def gaussian_kernel(test_inp, train_inp):
    ker_dist = np.array([0]*train_inp.shape[0])
    pred_val = np.array([0]*train_inp.shape[0])
    for i in range(train_inp.shape[0]):
        ker_dist[i] = np.exp(-np.sum((test_inp.iloc[4:10] - train_inp.iloc[i, 4:
↪10])**2)/2)
```

```
        if train_inp.iloc[i, 10] == 0:
            pred_val[i] = -1
        else:
            pred_val[i] = 1
    final = np.sign(np.sum(ker_dist*pred_val)/np.sum(ker_dist))
    if final >= 0:
        return 1
    else:
        return 0

#Five Fold Cross Validation
kf = KFold(n_splits=5, shuffle = True, random_state = 0)
kf.get_n_splits(train)
cv_acc = []
for train_ind, test_ind in kf.split(train):
    X_train, X_test = train.iloc[train_ind], train.iloc[test_ind]
    pred = []
    train_uniq = X_train.iloc[:,0].unique()
    for tst in range(len(test_ind)):
        fips_test = X_test.iloc[tst, 0]
        # if no neighbor, random guess
        if fips_test not in neigh_dic:
            pred.append(random.randint(0, 1))
            continue
        neigh_test = neigh_dic[fips_test]
        # Check if the neighbors are in training
        train_neigh = []
        for ng in neigh_test:
            if ng in train_uniq:
                train_neigh.append(ng)
        # If no neighbor in training, random prediction
        if len(train_neigh) == 0:
            pred.append(random.randint(0, 1))
            continue
        # Compute kernel distance between training and testing
        tp_train = X_train[X_train["FIPS"].isin(train_neigh)]
        pred.append(gaussian_kernel(X_test.iloc[tst,:], tp_train))
    cv_acc.append(weighted_accuracy(pred, X_test.iloc[:, 10]))
print(cv_acc)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:14:
RuntimeWarning: invalid value encountered in long_scalars

C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:14:
RuntimeWarning: invalid value encountered in sign
```

```
[0.5093362409704238, 0.498964464791803, 0.5094264892268694, 0.5180668473351401,
0.5331662489557226]
```

[11]:
```
# Train a random forest algorithm using train_x and train_y corresponding␣
↪hyper-parameters
# Utilizes out-of-bag testing for prediction, return weighted accuracy between␣
↪prediction and true label
def rf_oob(train_x, train_y, n_tree, max_fea, max_depth):
    ran_f = RandomForestClassifier(n_estimators = n_tree, max_features =␣
↪max_fea, max_depth = max_depth,
                                   random_state=0, oob_score = True)
    ran_f.fit(train_x, train_y)
    pred = []
    for i in range(ran_f.oob_decision_function_.shape[0]):
        if ran_f.oob_decision_function_[i, 0] > ran_f.oob_decision_function_[i,␣
↪1]:
            pred.append(0)
        else:
            pred.append(1)
    return weighted_accuracy(pred, train_y)
```

[12]:
```
# Random Forest with new features
train_2016 = pd.read_csv("train_2016.csv")
train_2012 = pd.read_csv("train_2012.csv")

#Convert features to floats
train_2016["MedianIncome"] = train_2016["MedianIncome"].str.replace(',', '').
↪astype(float)
train_2016["MigraRate"] = train_2016["MigraRate"].astype(float)
train_2016["BirthRate"] = train_2016["BirthRate"].astype(float)
train_2016["DeathRate"] = train_2016["DeathRate"].astype(float)
train_2016["BachelorRate"] = train_2016["BachelorRate"].astype(float)
train_2016["UnemploymentRate"] = train_2016["UnemploymentRate"].astype(float)
train_2012["MedianIncome"] = train_2012["MedianIncome"].str.replace(',', '').
↪astype(float)
train_2012["MigraRate"] = train_2012["MigraRate"].astype(float)
train_2012["BirthRate"] = train_2012["BirthRate"].astype(float)
train_2012["DeathRate"] = train_2012["DeathRate"].astype(float)
train_2012["BachelorRate"] = train_2012["BachelorRate"].astype(float)
train_2012["UnemploymentRate"] = train_2012["UnemploymentRate"].astype(float)

# Create new features based on the difference between 2016 and 2012
train_2016["MI Diff"] = train_2016.iloc[:, 4] - train_2012.iloc[:, 4]
train_2016["MR Diff"] = train_2016.iloc[:, 5] - train_2012.iloc[:, 5]
train_2016["BR Diff"] = train_2016.iloc[:, 6] - train_2012.iloc[:, 6]
train_2016["DR Diff"] = train_2016.iloc[:, 7] - train_2012.iloc[:, 7]
train_2016["BC Diff"] = train_2016.iloc[:, 8] - train_2012.iloc[:, 8]
```

```
train_2016["UR Diff"] = train_2016.iloc[:, 9] - train_2012.iloc[:, 9]

# Create feature for percentage of neighboring countries that voted democrat
output = train_2016["DEM"] > train_2016["GOP"]
train_2016["Output"] = output.astype(int)
nei_pct_dem = []
for i in range(train_2016.shape[0]):
    # if no neighbors in graph.csv, set default to 0.5
    if train_2016.iloc[i,0] not in neigh_dic:
        nei_pct_dem.append(0.5)
        continue
    # Find all neighbors in training set
    tp_neigh = neigh_dic[train_2016.iloc[i,0]]
    neigh_df = train_2016[train_2016["FIPS"].isin(tp_neigh)]
    # If no neighbors in training set, set default to 0.5
    if neigh_df.size == 0:
        nei_pct_dem.append(0.5)
    else:
        nei_pct_dem.append(np.sum(neigh_df.iloc[:, 16])/neigh_df.shape[0])
train_2016["Neighbor Percent Democratic"] = nei_pct_dem

# Standardize features
mean_fea = []
sd_fea = []
for i in range(12):
    mean_fea.append(train_2016.iloc[:, i+4].mean())
    sd_fea.append(train_2016.iloc[:, i+4].std())
    train_2016.iloc[:, i+4] = (train_2016.iloc[:, i+4] - mean_fea[i])/sd_fea[i]

# Function to standardize testing features
def mod_test_crea(test_2016, test_2012, mean_fea, sd_fea, neigh_dic,␣
 ↪train_2016):
    #Convert features to floats
    test_2016["MedianIncome"] = test_2016["MedianIncome"].str.replace(',', '').
 ↪astype(float)
    test_2016["MigraRate"] = test_2016["MigraRate"].astype(float)
    test_2016["BirthRate"] = test_2016["BirthRate"].astype(float)
    test_2016["DeathRate"] = test_2016["DeathRate"].astype(float)
    test_2016["BachelorRate"] = test_2016["BachelorRate"].astype(float)
    test_2016["UnemploymentRate"] = test_2016["UnemploymentRate"].astype(float)
    test_2012["MedianIncome"] = test_2012["MedianIncome"].str.replace(',', '').
 ↪astype(float)
    test_2012["MigraRate"] = test_2012["MigraRate"].astype(float)
    test_2012["BirthRate"] = test_2012["BirthRate"].astype(float)
    test_2012["DeathRate"] = test_2012["DeathRate"].astype(float)
    test_2012["BachelorRate"] = test_2012["BachelorRate"].astype(float)
    test_2012["UnemploymentRate"] = test_2012["UnemploymentRate"].astype(float)
```

```
    # Create new features based on the difference between 2016 and 2012
    test_2016["MI Diff"] = test_2016.iloc[:, 2] - test_2012.iloc[:, 2]
    test_2016["MR Diff"] = test_2016.iloc[:, 3] - test_2012.iloc[:, 3]
    test_2016["BR Diff"] = test_2016.iloc[:, 4] - test_2012.iloc[:, 4]
    test_2016["DR Diff"] = test_2016.iloc[:, 5] - test_2012.iloc[:, 5]
    test_2016["BC Diff"] = test_2016.iloc[:, 6] - test_2012.iloc[:, 6]
    test_2016["UR Diff"] = test_2016.iloc[:, 7] - test_2012.iloc[:, 7]

    for i in range(12):
        test_2016.iloc[:, i+2] = (test_2016.iloc[:, i+2] - mean_fea[i])/
↪sd_fea[i]

    # Create final input column based on percentage of neighbors that voted␣
↪democratic in 2016
    nei_pct_dem = []
    for i in range(test_2016.shape[0]):
        if test_2016.iloc[i,0] not in neigh_dic:
            nei_pct_dem.append(0.5)
            continue
        tp_neigh = neigh_dic[test_2016.iloc[i,0]]
        neigh_df = train_2016[train_2016["FIPS"].isin(tp_neigh)]
        if neigh_df.size == 0:
            nei_pct_dem.append(0.5)
        else:
            nei_pct_dem.append(np.sum(neigh_df.iloc[:, 16])/neigh_df.shape[0])
    test_2016["Neighbor Percent Democratic"] = nei_pct_dem

    return test_2016
```

[13]:
```
# Random Forest with out of bag testing
no_tree = [300, 500, 700]
max_fea = [6, 8, 10]
max_depth = [6, 8, 10]
fea_ls = []
acc_ls = []

# Grid search through all combinations of hyper-parameter settings to find␣
↪optimal setting
fea_col = [*range(4, 16), 17]
for nt in no_tree:
    for mf in max_fea:
        for md in max_depth:
            fea_ls.append((nt, mf, md))
            oob_sc = rf_oob(train_2016.iloc[:,fea_col], train_2016.iloc[:, 16],␣
↪nt, mf, md)
            acc_ls.append(oob_sc)
```

```
# Select hyper-parameter setting with best cross-validation performance
print(max(acc_ls))
ind = acc_ls.index(max(acc_ls))
print(fea_ls[ind])
nt = fea_ls[ind][0]
mf = fea_ls[ind][1]
md = fea_ls[ind][2]

# Retrain random forest with optimal hyper-parameter settings on full training␣
 ↪data
final_rf2 = RandomForestClassifier(n_estimators = nt, max_features = mf,␣
 ↪max_depth = md,
                                   random_state=0)
final_rf2.fit(train_2016.iloc[:,fea_col], train_2016.iloc[:,16])
```

```
0.7527234753550545
(500, 8, 10)
```

[13]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=10, max_features=8, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=500,
                       n_jobs=None, oob_score=False, random_state=0, verbose=0,
                       warm_start=False)
```

[14]:
```
# Using Cross Validation for hyper-parameter selection for Gradient Boosted␣
 ↪Machine
from sklearn.ensemble import GradientBoostingClassifier

# Initialize five fold cross-validation
kf = KFold(n_splits=5, shuffle = True, random_state = 0)
kf.get_n_splits(train_2016)

# Hyper-parameters will be selected iteratively; starting with number of␣
 ↪estimators
number_of_estimators = [200, 300, 400, 500]
acc_ls = []
for ne in number_of_estimators:
    cv_acc = []
    gb_f = GradientBoostingClassifier(n_estimators = ne, random_state = 0)
    for train_ind, test_ind in kf.split(train_2016):
        X_train, X_test = train_2016.iloc[:,fea_col].iloc[train_ind,:],␣
 ↪train_2016.iloc[:,fea_col].iloc[test_ind,:]
        y_train, y_test = train_2016.iloc[:,16][train_ind], train_2016.iloc[:
 ↪,16][test_ind]
```

```
        gb_f.fit(X_train, y_train)
        pred = gb_f.predict(X_test)
        cv_acc.append(weighted_accuracy(pred, y_test))
    acc_ls.append(sum(cv_acc)/len(cv_acc))
ne = number_of_estimators[acc_ls.index(max(acc_ls))]

#Next, select best hyper-parameter for learning rate
learning_rate = [0.01, 0.03, 0.05, 0.01, 0.1, 0.2]
acc_ls = []
for lr in learning_rate:
    cv_acc = []
    gb_f = GradientBoostingClassifier(n_estimators = ne, learning_rate = lr,␣
 ↪random_state = 0)
    for train_ind, test_ind in kf.split(train_2016):
        X_train, X_test = train_2016.iloc[:,fea_col].iloc[train_ind,:],␣
 ↪train_2016.iloc[:,fea_col].iloc[test_ind,:]
        y_train, y_test = train_2016.iloc[:,16][train_ind], train_2016.iloc[:
 ↪,16][test_ind]
        gb_f.fit(X_train, y_train)
        pred = gb_f.predict(X_test)
        cv_acc.append(weighted_accuracy(pred, y_test))
    acc_ls.append(sum(cv_acc)/len(cv_acc))
lr = learning_rate[acc_ls.index(max(acc_ls))]

#Next, select best hyper-parameter for subsampling rate
subsample = [0.5, 0.6, 0.7, 0.8, 0.9]
acc_ls = []
for ss in subsample:
    cv_acc = []
    gb_f = GradientBoostingClassifier(n_estimators = ne, learning_rate = lr,␣
 ↪subsample = ss, random_state = 0)
    for train_ind, test_ind in kf.split(train_2016):
        X_train, X_test = train_2016.iloc[:,fea_col].iloc[train_ind,:],␣
 ↪train_2016.iloc[:,fea_col].iloc[test_ind,:]
        y_train, y_test = train_2016.iloc[:,16][train_ind], train_2016.iloc[:
 ↪,16][test_ind]
        gb_f.fit(X_train, y_train)
        pred = gb_f.predict(X_test)
        cv_acc.append(weighted_accuracy(pred, y_test))
    acc_ls.append(sum(cv_acc)/len(cv_acc))
ss = subsample[acc_ls.index(max(acc_ls))]

#Next, select best hyper-parameter for max depth
max_depth = [4, 6, 8, 10, 12]
acc_ls = []
for ml in max_depth:
```

```
    cv_acc = []
    gb_f = GradientBoostingClassifier(n_estimators = ne, learning_rate = lr,␣
↪subsample = ss,
                                      max_depth = ml, random_state = 0)
    for train_ind, test_ind in kf.split(train_2016):
        X_train, X_test = train_2016.iloc[:,fea_col].iloc[train_ind,:],␣
↪train_2016.iloc[:,fea_col].iloc[test_ind,:]
        y_train, y_test = train_2016.iloc[:,16][train_ind], train_2016.iloc[:
↪,16][test_ind]
        gb_f.fit(X_train, y_train)
        pred = gb_f.predict(X_test)
        cv_acc.append(weighted_accuracy(pred, y_test))
    acc_ls.append(sum(cv_acc)/len(cv_acc))
ml = max_depth[acc_ls.index(max(acc_ls))]

#Next, select best hyper-parameter for min_samples_split
min_samples_split = [10, 25, 50, 75, 100]
acc_ls = []
for mss in min_samples_split:
    cv_acc = []
    gb_f = GradientBoostingClassifier(n_estimators = ne, learning_rate = lr,␣
↪subsample = ss,
                                      max_depth = ml, min_samples_split = mss,␣
↪random_state = 0)
    for train_ind, test_ind in kf.split(train_2016):
        X_train, X_test = train_2016.iloc[:,fea_col].iloc[train_ind,:],␣
↪train_2016.iloc[:,fea_col].iloc[test_ind,:]
        y_train, y_test = train_2016.iloc[:,16][train_ind], train_2016.iloc[:
↪,16][test_ind]
        gb_f.fit(X_train, y_train)
        pred = gb_f.predict(X_test)
        cv_acc.append(weighted_accuracy(pred, y_test))
    acc_ls.append(sum(cv_acc)/len(cv_acc))
mss = min_samples_split[acc_ls.index(max(acc_ls))]

#Next, select best hyper-parameter for min_samples_split
max_features = [4, 6, 8, 10, 12]
acc_ls = []
for mf in max_features:
    cv_acc = []
    gb_f = GradientBoostingClassifier(n_estimators = ne, learning_rate = lr,␣
↪subsample = ss,
                                      max_depth = ml, min_samples_split = mss,␣
↪max_features = mf,
                                      random_state = 0)
    for train_ind, test_ind in kf.split(train_2016):
```

```
        X_train, X_test = train_2016.iloc[:,fea_col].iloc[train_ind,:],␣
 ↪train_2016.iloc[:,fea_col].iloc[test_ind,:]
        y_train, y_test = train_2016.iloc[:,16][train_ind], train_2016.iloc[:
 ↪,16][test_ind]
        gb_f.fit(X_train, y_train)
        pred = gb_f.predict(X_test)
        cv_acc.append(weighted_accuracy(pred, y_test))
    acc_ls.append(sum(cv_acc)/len(cv_acc))
mf = max_features[acc_ls.index(max(acc_ls))]

# Train final GBM based on selected tuning parameters on full training data
gb_f = GradientBoostingClassifier(n_estimators = ne, learning_rate = lr,␣
 ↪subsample = ss,
                                  max_depth = ml, min_samples_split = mss,␣
 ↪max_features = mf, random_state = 0)
gb_f.fit(train_2016.iloc[:, fea_col], train_2016.iloc[:, 16])
```

[14]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=6,
                           max_features=8, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=75,
                           min_weight_fraction_leaf=0.0, n_estimators=500,
                           n_iter_no_change=None, presort='auto',
                           random_state=0, subsample=0.7, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

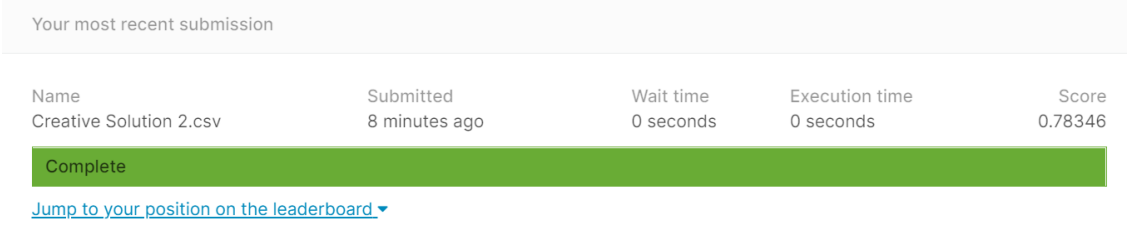
3.2 Explanation in Words:

You need to answer the following questions in a markdown cell after this cell:

3.2.1 How much did you manage to improve performance on the test set compared to part 2? Did you reach the 75% accuracy for the test in Kaggle? (Please include a screenshot of Kaggle Submission)

3.2.2 Please explain in detail how you achieved this and what you did specifically and why you tried this.

3.2.1 We managed to improve performance on the test set to 0.78346 from 0.716 from part 2. We managed to reach the 75 percent accuracy for the test in Kaggle.



3.2.2 For the creative part, the first thing we looked at is using graph.csv to identify the nearest

neighbors between counties. This is because lookng at the election map, we see large clusters of blue and red counties. These clusters seem to indicate counties tend to vote similar to its neighbors. After creating a dictionary of all neighbors for each county, we tried using a kernalized nearest neighbor algorithm using weights based on the Gaussian kernalized distance between standardized features (median income, migration rate, birth rate, death rate, bachelor rate and unemployment rate). However, the preliminary results shows this is close to random in out-of-sample testing so we gave up on that idea pretty quickly.

Second, we tried creating an additional feature using graph.csv and information from 2012. For each county, we created a new feature "neighbor percent democrat" that gives the percentage of neighbors in the training set that voted democrat in 2016 for each county. If a county has no neighbor in the training set, then we set the default value for "neighbor percent democrat" to be 0.5, as to be non-informative. This feature will give us information on how the neighbors of county voted. Next, we added 6 additional features for each county based on changes in median income, migration rate, birth rate, death rate, bachelor rate and unemployment rate between 2012 to 2016. Once again, we standardized these 6 features. These new features can give us more information about the counties and should improve our algorithm's performance. As a result, we decided to try running random forest once again as random forest seems to work well on the basic data. With the new added features, we tuned three hyper-parameters (number of trees, max depth and max features) using out-of-bag testing. For Random Forest, it has been shown that using out-of-bag testing is equivalent to using a validation set the size of the training set. Out-of-bag testing is also shown to be more efficient. After finding the optimal parameters, we tested the performance of the testing set and received a score of 0.756. While this satisfied the requirement, we believe we can do better.

Third, to improve upon the performance from Random Forest, we decided to try a more powerful learner in Gradient Boosted Machine (GBM). GBM utilizes boosting to combine weak learners to produce a strong learner that can learn highly complex patterns in the data. For hyper-parameter selection, we used a five fold cross-validation for hyper parameter selection. We split our training data into 5 groups such that we would train a hypothesis on 80% of our dataset and validate it on the other 20% of our dataset. We do this such that each group takes turns as a validation set. We tuned six parameters in the GBM algorithm: number of estimators, learning rate, sub-sampling rate, max depth, minimum samples split and max features. The parameters are tuned iteratively one after another as there are too many combinations to do a full grid-search. After finding the optimal set of parameters, we trained GBM on the full training data and tested on the test data. We were able to achieve a final result of 0.783 in testing accuracy.

Part 4: Kaggle Submission

You need to generate a prediction CSV using the following cell from your trained model and submit the direct output of your code to Kaggle. The CSV shall contain TWO column named exactly "FIPS" and "Result" and 1555 total rows excluding the column names, "FIPS" column shall contain FIPS of counties with same order as in the test_2016_no_label.csv while "Result" column shall contain the 0 or 1 prdicaitons for corresponding columns. A sample predication file can be downloaded from Kaggle.

[15]:
```
# You may use pandas to generate a dataframe with FIPS and your predictions␣
↪first
# and then use to_csv to generate a CSV file.
```

```
test = pd.read_csv("test_2016_no_label.csv")
test = mod_test(test, mean_MI, sd_MI, mean_MR, sd_MR, mean_BR, sd_BR, mean_DR,␣
↪sd_DR, mean_BC, sd_BC, mean_UR, sd_UR)

final_pred = final_rf.predict(test.iloc[:,2:8])
dic = {"FIPS": test["FIPS"], "Result": final_pred}
final_output = pd.DataFrame(dic)
final_output.to_csv("Basic Solution 1.csv", index=False)

final_pred2 = final_svm.predict(test.iloc[:,2:8])
dic2 = {"FIPS": test["FIPS"], "Result": final_pred2}
final_output2 = pd.DataFrame(dic2)
final_output2.to_csv("Basic Solution 2.csv", index=False)
```

[16]:
```
# Creative Solution
test_2016 = pd.read_csv("test_2016_no_label.csv")
test_2012 = pd.read_csv("test_2012_no_label.csv")
test_2016 = mod_test_crea(test_2016, test_2012, mean_fea, sd_fea, neigh_dic,␣
↪train_2016)
#final_pred = final_rf2.predict(test_2016.iloc[:,2:15])
final_pred = gb_f.predict(test_2016.iloc[:,2:15])
dic = {"FIPS": test_2016["FIPS"], "Result": final_pred}
final_output = pd.DataFrame(dic)
final_output.to_csv("Creative Solution 2.csv", index=False)
```

Part 5: Resources and Literature Used

We used the following packages from the Scikit-learn library:

KFold: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html
(for training, validation and model selection)

RandomForestClassifier: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClass
(for Basic Solution 1)

SVC: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC
(for Basic Solution 2)

GradientBoostingClassifier: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostin
(for Creative Solution)