CS 4780/5780 Final Project:

Election Result Prediction for US Counties

Names and NetIDs for your group members: Junho Kim-Lee (jk2333), Jeong Hyun Lee (jl2374)

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.

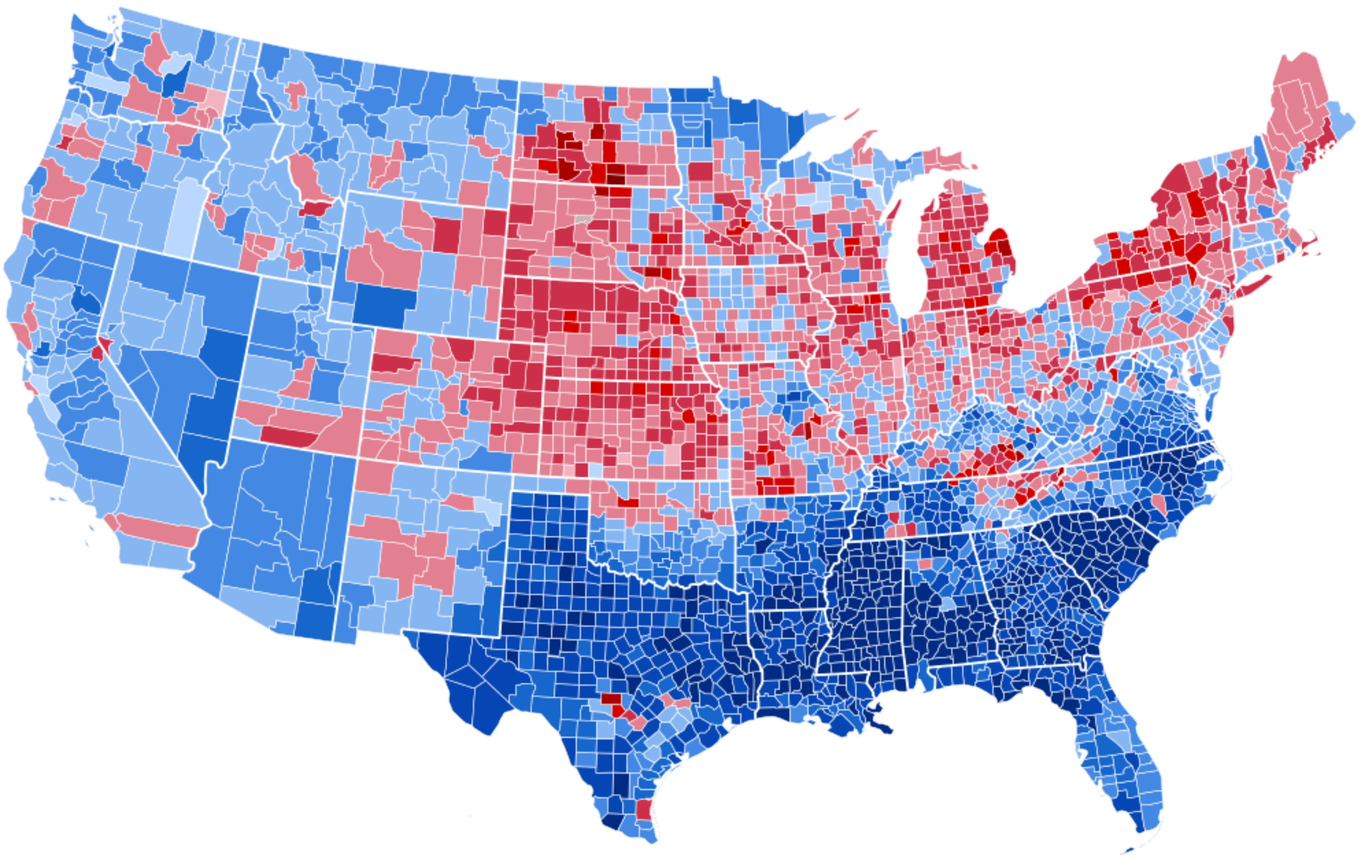
The task you will work on is forecasting election results. Economic and sociological factors have been widely used when making predictions on the voting results of US elections. Economic and sociological factors vary a lot among counties in the United States. In addition, as you may observe from the 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. </p>

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

https://pytorch.org/tutorials/ (https://pytorch.org/tutorials/)

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

```
In [ ]: import os
        import pandas as pd
        import numpy as np
        # TODO
        from sklearn import preprocessing, svm
        from sklearn.neural_network import MLPClassifier
        from sklearn.model_selection import train_test_split, KFold
        import math
        import time
```

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.

```
In [ ]: 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 the 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.

In [ ]:
```
# You may change this but we suggest loading data with the following cod
e and you may need to change
# datatypes and do necessary data transformation after loading the raw d
ata to the dataframe.

# Make sure you comment your code clearly and you may refer to these com
ments in the part 2.4

def preprocess(path, labels = True):
    df = pd.read_csv(path, sep=',',header=0, encoding='unicode_escape')

    # drop non-numerical columns
    df = df.drop('County', axis=1)

    # type conversions
    for col in df.columns:
        if col == 'DEM' or col == 'GOP':
            df[col] = df[col].astype(int)
        elif col == 'MedianIncome':
            df[col] = df[col].str.replace(",","").astype(int)
        elif col == 'FIPS':
            df[col] = df[col].astype(int)
        else:
            df[col] = df[col].astype(float)

    # replace DEM/GOP columns with a binary results array
    if labels == True:
        yTr = np.where(df['DEM'] > df['GOP'], 1, 0)
        df = df.drop('DEM', axis=1)
        df = df.drop('GOP', axis=1)
    else:
        yTr = df['FIPS']     # for carrying over FIPS ID for testing data

    # drop FIPS after potentially storing it to yTr
    df = df.drop('FIPS', axis=1)

    # normalize
    xTr = preprocessing.StandardScaler().fit_transform(df)

    return xTr, yTr
```

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.

In [ ]:
```
# Make sure you comment your code clearly and you may refer to these com
ments in the part 2.4

# create a neural network with custom inputs for hyperparameters
# outputs predicted values for given test set xTe
def neural(xTe, xTr, yTr, h, a, al, m):
    clf = MLPClassifier(
        hidden_layer_sizes = h,
        activation = a,
        alpha = al,
        max_iter = m,
        random_state = 7
    )
    clf.fit(xTr,yTr)
    yTe = clf.predict(xTe)

    return yTe

# create a SVM classification function with custom inputs for hyperparam
eters C, kernel, and gamma
# outputs predicted values for given test set xTe
def SVM(xTe, xTr, yTr, C, k, g="scale"):
    classifier = svm.SVC(C, kernel = k, gamma = g)
    classifier.fit(xTr, yTr)
    yTe = classifier.predict(xTe)
    return yTe
```

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.

In [ ]:
```
# Make sure you comment your code clearly and you may refer to these com
ments in the part 2.4
# TODO

# read in training data and return xTr, yTr, xTe, yTe with an 80/20 spli
t
def get_split_data(path):
    xTr, yTr = preprocess(path)
    return train_test_split(xTr, yTr, test_size=.2, random_state=7)
```

In [ ]:

```
# validate neural network to choose best custom parameters that leads to
highest testing accuracy.
# 80/20 split into training and validation sets.
def neural_val():

    # split into test/train/val sets
    xTr,xTe,yTr,yTe = get_split_data("train_2016.csv")
    xTr,xVal,yTr,yVal = train_test_split(xTr, yTr, test_size=.2, random_
state=7)

    # hyperparameters
    hidden_layer_sizes = np.arange(80,101,5)
    activation = ['logistic','tanh','relu']
    alpha = [.000001,.00001,.0001]
    max_iter = np.arange(1000,3001,500)

    # tracker for best neural net parameters
    best = (
        hidden_layer_sizes[0],
        activation[0],
        alpha[0],
        max_iter[0]
            )
    best_a = -1

    # validate until we have selected good parameters
    start = time.time()
    z = 1
    for h in hidden_layer_sizes:
        for a in activation:
            for al in alpha:
                for m in max_iter:
                    yVal_emp = neural(xVal, xTr, yTr, h, a, al, m,)
                    acc = weighted_accuracy(yVal_emp, yVal)
                    if acc > best_a:
                        best_a = acc
                        best = (h,a,al,m)

                # progress tracker
                print(str(math.floor(z*100/45))+"% done. Time to complet
ion: "+
                      str(round(((time.time()-start)/z)*(45-z),0)) +" se
conds.")
                z += 1
    print("Validation finished after "+str(round(time.time()-start,0))+"
seconds.")

    # test the best model with the test set
    yTe_emp = neural(xTe, xTr, yTr,best[0],best[1],best[2],best[3])
    acc = weighted_accuracy(yTe_emp, yTe)

    # print outputs
    print("Hidden Layer Size: " + str(best[0]))
    print("Activation: " + best[1])
    print("Alpha: " + str(best[2]))
    print("Max Iterations: " + str(best[3]))
    print("Validation Accuracy: " + str(round(best_a*100,2)) + "%")
    print("Test Accuracy: " + str(round(acc*100,2)) + "%")

    return acc, best
```

In [ ]:
```
# validate svm to choose best custom parameters that leads to highest te
sting accuracy.
# 10-fold cross-validation.
def svm_val():

    X_train, X_test, y_train, y_test = get_split_data("train_2016.csv")

    # hyperparameters for SVM
    parameters_C = np.arange(1, 20, 1)
    parameters_kernel = ('linear', 'poly', 'rbf', 'sigmoid')
    parameters_gamma = [1, 0.1, 0.01, 0.001, "scale", "auto"]

    bestC = 0
    bestKernel = ""
    bestGamma = 0
    bestScore = 0

    # K-fold CV that tunes three hyperparameters
    kf = KFold(n_splits=10)

    # keep track of runtime and progress
    start = time.time()
    z = 1

    # three loops for the three hyperparameters
    for C in parameters_C:
        for kernel in parameters_kernel:
            for gamma in parameters_gamma:
                scores = []
                # 10-fold CV
                for train_index, test_index in kf.split(X_train):
                    X_train_split, X_test_split = X_train[train_index],
X_train[test_index]
                    y_train_split, y_test_split = y_train[train_index],
y_train[test_index]

                    preds = SVM(X_test_split, X_train_split, y_train_spl
it, C, kernel, gamma)
                    scores.append(weighted_accuracy(preds, y_test_split
))

                # get average score across the 10-fold CV
                score = np.mean(scores)

                # record best score and its hyperparameters
                if score > bestScore:
                    bestScore = score
                    bestC = C
                    bestGamma = gamma
                    bestKernel = kernel
                # progress tracker
                print(str(math.floor(z*100/(len(parameters_C)*len(parame
ters_gamma)*len(parameters_kernel))))
                      + "% done. Time to completion: "
                      + str(round(((time.time()-start)/z)
                            * ((len(parameters_C)*len(parameters_g
amma)*len(parameters_kernel))-z),0)) +" seconds.")
                z += 1
    print("Validation finished after "+str(round(time.time()-start,0))+"
seconds.")

    print("Best weighted accuracy:", bestScore)
    print("Optimal hyperparameters: C=" + str(bestC) + ", kernel=" + bes
tKernel + ", gamma=" + str(bestGamma))

    # get test set accuracy
    y_preds_SVM = SVM(X_test, X_train, y_train, bestC, bestKernel, bestG
amma)

    acc= weighted_accuracy(y_preds_SVM, y_test)
    print("SVM accuracy", acc)

    return acc, (bestC, bestKernel, bestGamma)
```

In [ ]:

```
# validate both neural net and svm and choose the one that performs bett
er
neural_acc, neural_para = neural_val()
print("------------")
svm_acc, svm_para = svm_val()
print("------------")
if neural_acc > svm_acc:
    print("Neural network is more accurate.")
    print("Parameters:" + str(neural_para))
else:
    print("SVM is more accurate.")
    print("Parameters:" + str(svm_para))
```

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?

> Our preprocessing was done in two steps. In the first step, we read the train dataset as a Pandas dataframe and set the headers. Additionally, we first dropped any features with string values (`County`) as they are hard to use in classification. Then we formatted each column to fit the data type of the feature. Hence, for features `DEM` , `GOP` , and `FIPS` , we read their values as `int` , for `MedianIncome` , as its values were in string, we had to convert them to `int` . For the rest of the features, we converted them to `float` to accomodate for the decimals. Then, for each county, we had to deduce the winner of the election and create our labels for the training set. Hence, by comparing the numbers of `DEM` and `GOP` for each county, the label would contain the feature with the higher vote count. In our second step, we dropped `DEM` and `GOP` as they are not features provided in the test data (if so, we would have our predictions right away from the test set) and we also dropped `FIPS` as we considered county codes as not informative (they are systematically generated and hence not really affected by the outcome). Lastly, we normalized all our feature vectors so that they would all have mean of 1 and standard deviation of 1.

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

> We used SVM and a simple neural network to make the classification. We thought the SVM was a suitable learning method because first, we have a binary classification problem. We used SVM over other binary classifiers (such as Perceptron) because we had a strong sense that the data will probably not be linearly separable and hence a soft-margin SVM will work better to accommodate for the support vectors. In addition, we could kernelize the classifier using algorithms such as SVM so that our data can be better classified in case the raw distribution is not in a linearly separable shape. In defining our hyperparameters, we tested for different `C` values, different kernels, and different gamma values. We conducted a grid-search 10-fold cross validation on our training set, which then we found our best hyperparameters from our best valdiation score
>
> Our second choice of using a simple neural network was largely based on our other assumption that the data might not be completely linearly separable (and in some other form of distribution). Based on the algorithms we have learned in class, we thought the neural network was better than other non-linear classification methods such as decision trees and k-NN because we believed decisions trees would be, even with pruning measures, subject to overfitting due to our number of features and k-NN would be subject to noninformative features that may be present. Furthermore, we thought the neural network was the best choice in this regard because by customizing the number of neurons and layers, we could more easily control for overfitting and accuracy. In defining our hyperparameters, we tested for different hidden layer sizes, different activation functions, different values of alpha, and different values of maximum iterations. Due to its larger size of parameters, we did a single-fold validation, where we divided the training data into a training set, a validation set, and a test set, and we chose the hyperparameters that gave the best validation score.

2.4.3 How did you do the model selection?

In our model selection, we fed the same training set (which would further be divided into training and validation within each method) and testing set to both learning methods mentioned above (`svm_val` and `neural_val`). We chose to go with a training/testing ratio of 80/20 and we used the same random state number so our train/test split would be same for both.

For our neural network, we chose to test different parameters for our hidden layer sizes, activation function, alpha, and maximum iteration size. We wanted to test a wide range to ensure that we chose parameters that led to an accurate result without overfitting. Similarly, for our SVM, we chose to test different parameters for our C, kernel function, and alpha. For both our models, if the model selected by our validation chose a parameter that was either a min or max value, we reran the validation introducing new values for that parameter smaller than the min or greater than the max, respectively. This ensured that there were no better parameter values that we were not testing.

We then chose the model (and its hyperparameters) that gave the higher testing score to submit to Kaggle. In the end, as the above results show, our best model was the neural network with 1 hidden layer consisting of 85 neurons, using a `ReLu` activation function, an alpha of .00001 and maximum iteration number of 1500.

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

With the above model, we submitted to the Kaggle baseline and we have reached over 68% weighted accuracy (72.285% to be exact). We attach a screenshot below.
baseline_solution

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.

```
In [ ]: # Make sure you comment your code clearly and you may refer to these com
        ments in the part 3.2
        # TODO
```

In [ ]:
```
# takes a FIPS ID of a county and returns a normalized value between -1
 and 1 that measures that county's neighbors
# affinity for voting DEM/GOP. +1 indicates 100% neighbor voting for DEM
and -1 indicates 100% neighbor voting for
# GOP.
def get_neighbor_score(id):

    # get list of neighbor FIPS ID's
    df = pd.read_csv("graph.csv", sep=',',header=0, encoding='unicode_es
cape')
    df = df[df.SRC.eq(id)]
    dst = df['DST']

    # load training data
    df = pd.read_csv("train_2016.csv", sep=',',header=0, encoding='unico
de_escape')

    # create scalar geographic affinity score
    score = 0
    labeled_counties = 0
    for i in dst:
        if i != id:
            df2 = df[df.FIPS.eq(i)]
            dem_list = df2['DEM'].tolist()
            if len(dem_list) == 1:
                labeled_counties += 1
                gop_list = df2['GOP'].tolist()
                if dem_list[0] > gop_list[0]:
                    score += 1
                else:
                    score -= 1
    if labeled_counties == 0:
        normalized_score = 0
    else:
        normalized_score = score/labeled_counties

    return normalized_score
```

In [ ]:
```python
# the same as preprocess but adds a feature to all vectors, which is the
neighbor score calculated in the func above
def preprocess2(path, labels = True):
    df = pd.read_csv(path, sep=',',header=0, encoding='unicode_escape')

    # drop non-numerical columns
    df = df.drop('County', axis=1)

    # type conversions
    for col in df.columns:
        if col == 'DEM' or col == 'GOP':
            df[col] = df[col].astype(int)
        elif col == 'MedianIncome':
            df[col] = df[col].str.replace(",","").astype(int)
        elif col == 'FIPS':
            df[col] = df[col].astype(int)
        else:
            df[col] = df[col].astype(float)

    # replace DEM/GOP columns with a binary results array
    if labels == True:
        yTr = np.where(df['DEM'] > df['GOP'], 1, 0)
        df = df.drop('DEM', axis=1)
        df = df.drop('GOP', axis=1)
    else:
        yTr = df['FIPS']    # for carrying over FIPS ID for testing data

    # add geography feature
    geo = []
    for i in df['FIPS']:
        geo.append(get_neighbor_score(i))
    df['Geo'] = geo

    # drop FIPS after potentially storing it to yTr
    df = df.drop('FIPS', axis=1)

    # normalize
    xTr = preprocessing.StandardScaler().fit_transform(df)

    return xTr, yTr
```

In [ ]:
```python
# the same as neural() but drops the activation func as a parameter
def neural2(xTe, xTr, yTr, h, a, m):

    clf = MLPClassifier(
        hidden_layer_sizes = h,
        alpha = a,
        max_iter = m,
        random_state = 7
    )
    clf.fit(xTr,yTr)
    yTe = clf.predict(xTe)

    return yTe
```

In [ ]:
```
# validate new neural network using the upgraded preprocesser and withou
t activation func as a parameter
def neural_val2(xTr, yTr):
    # hyperparameters
    hidden_layer_sizes = np.arange(80,101,5)
    alpha = [.0000001,.000001,.00001]
    max_iter = np.arange(1000,3001,500)

    # split into test/train/val sets
    xTr,xTe,yTr,yTe = train_test_split(xTr, yTr, test_size=.2, random_st
ate=7)
    xTr,xVal,yTr,yVal = train_test_split(xTr, yTr, test_size=.2, random_
state=7)

    # tracker for best neural net hyperparameters
    best = (
        hidden_layer_sizes[0],
        alpha[0],
        max_iter[0],
           )
    best_a = -1

    # validate until we have selected good parameters
    start = time.time()
    z = 1
    for h in hidden_layer_sizes:
        for a in alpha:
            for m in max_iter:
                yVal_emp = neural2(xVal, xTr, yTr, h, a, m)
                acc = weighted_accuracy(yVal_emp, yVal)
                if acc > best_a:
                    best_a = acc
                    best = (h,a,m)

                # progress tracker
                print(str(math.floor(z*100/75))+"% done. Time to complet
ion: "+
                      str(int(round(((time.time()-start)/z)*(75-z),0)))
+" seconds.")
                z += 1
    print("Validation finished after "+str(round(time.time()-start,0))+"
seconds.")

    # test the best model with the test set
    yTe_emp = neural2(xTe, xTr, yTr,best[0],best[1],best[2])
    acc = weighted_accuracy(yTe_emp, yTe)

    # print outputs
    print("Hidden Layer Size: " + str(best[0]))
    print("Alpha: " + str(best[1]))
    print("Max Iterations: " + str(best[2]))
    print("Validation Accuracy: " + str(round(best_a*100,2)) + "%")
    print("Test Accuracy: " + str(round(acc*100,2)) + "%")

    return acc, best
```

```
In [ ]: # load xTr and yTr
        xTr, yTr = preprocess2("train_2016.csv")

        # validate
        neural_acc, neural_para = neural_val2(xTr,yTr)
```

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)

> For our creative solution, we were able to reach the 75% accuracy mark on the creative Kaggle competition with a score of 75.103%. Compared to part 2, we increased our accuracy by about 3%.
> creative_solution

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

> We decided to further pursue neural networks because from our previous part, it gave a much higher test accuracy than SVM. From extensive validation testing from part 2, it was clear that `ReLu` consistently produced results with the highest test accuracy, so for part 3 we fixed `ReLu` as a custom parameter of our neural network in order to expedite the validation processing time.
>
> Our main strategy for part 3 was to boost our neural network by taking advantage of geographic location. Specifically, with `graph.csv` , we knew, for any given county, all the counties that bordered it. Additionally, with our training data, we knew the voting results of many of these counties. Our thinking was that we could evaluate how a county's neighbors voted and there was a good chance that the county would vote similar to its neighbors. We wanted to create a new feature for all our data that would represent this "geographic neighbor voting affinity".
>
> Our algorithm to do this is as follows: for any county, initialize their score to be 0 and use `graph.csv` to look at all its neighboring counties. Then, for each of these counties, we check to see if we know how they voted. If we do not know how they voted, unfortunately that data point becomes useless to us. However, if we do know how they voted, if they voted DEM, we add +1 to the score, and if they voted GOP, we add -1 to the score. After tallying up all its neighbors, we then normalize this score to fit between -1 and +1. As an example, if a county has 6 neighbors and we know that 3 of them voted DEM and 2 of them voted GOP, they would have a raw score of +1. After normalization, it becomes +0.2.
>
> We ran this algorithm for each county to generate a new feature entirely in the preprocesser function. We then ran the same neural network validation algorithm (minus activation function as a custom parameter), and the model it selected was a neural network with 1 hidden layer with 90 neurons, an alpha of .000001, and a max iteration count of 2000. This model was able to get us above the 75% threshold of the Kaggle creative competition.

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.

In [ ]:
```
# basic kaggle submission

# import data
xTr, yTr = preprocess("train_2016.csv")
xTe, FIPS = preprocess("test_2016_no_label.csv", labels = False)

# predict
yTe = neural(xTe, xTr, yTr, 85, 'relu', .00001, 1500)

# write to csv
d = {'FIPS': FIPS,'Result': yTe}
df = pd.DataFrame(data=d)
df.to_csv("Submission_basic.csv",index=False)
```

In [ ]:
```
# creative kaggle submission

# import data
xTr, yTr = preprocess2("train_2016.csv")
xTe, FIPS = preprocess2("creative_csv/test_2016_no_label_creative.csv",
labels = False)

# predict
yTe = neural2(xTe, xTr, yTr, 90, .000001, 2000)

# write to csv
d = {'FIPS': FIPS,'Result': yTe}
df = pd.DataFrame(data=d)
df.to_csv("Submission_creative.csv",index=False)
```

Part 5: Resources and Literature Used

For this project, we used `pandas` and `numpy` to read and manipulate csv training and testing csv files. Additionally, we used scikit for the bulk of our machine learning algorithms. This includes `preprocessing` and `svm` from sklearn, as well as `train_test_split` and `KFold` from `sklearn.model_selection` and `MLPClassifier` from `sklearn.neural_network` .