CS 4780/5780 Final Project:

Election Result Prediction for US Counties

Names and NetIDs for your group members:

Rohan Lewis, rl447

Haashim Hussain Shah, hhs66

Hudson Fernandes, haf48

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.

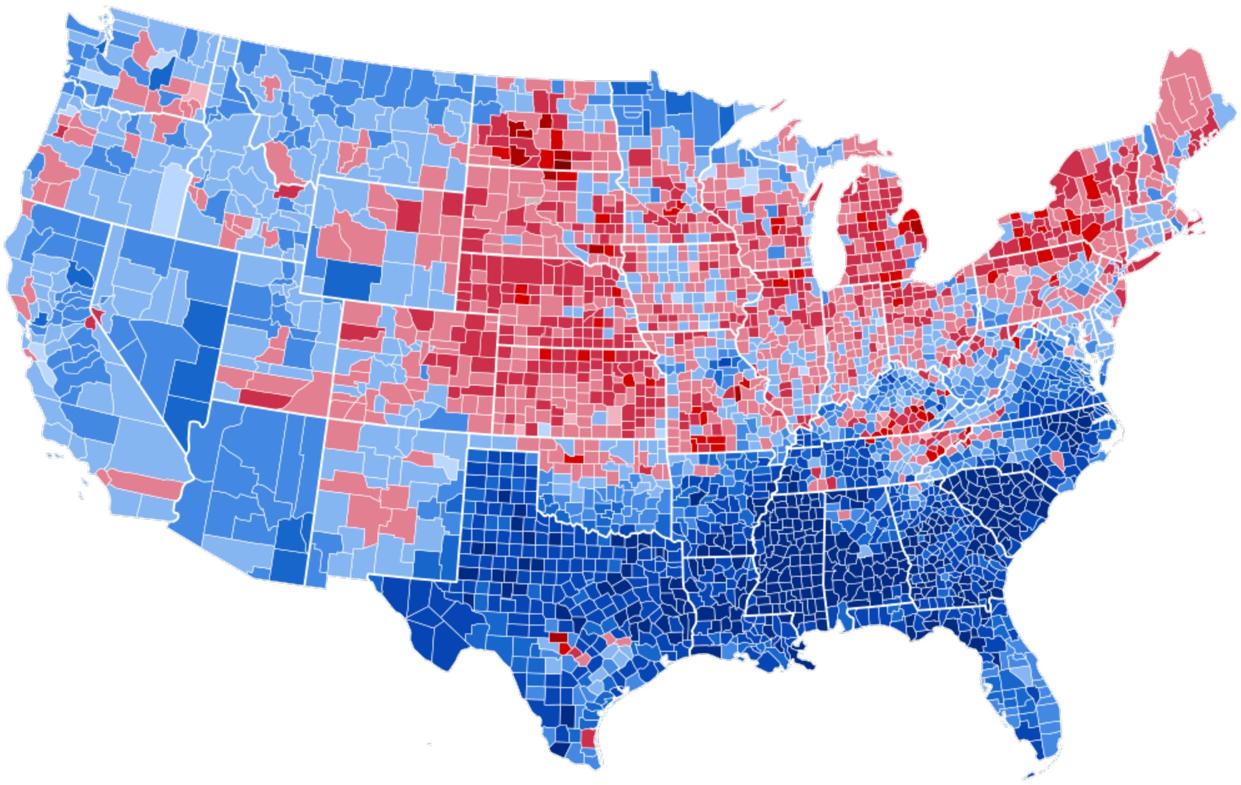
The task you will work on is forecasting election results. Economic and sociological factors have been widely used when making predictions on the voting results of US elections. Economic and sociological factors vary a lot among counties in the United States. In addition, as you may observe from the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

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

In [1]:

```
import os
import pandas as pd
import numpy as np
from IPython.display import Image
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
import itertools
import copy
```

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 [2]:

```
def weighted_accuracy(true, pred):
    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
    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.

In [3]:
```
# Note: thousands parameter needed so that MedianIncome column treated as an
train_2016 = pd.read_csv('./train_2016.csv', thousands=',')
test_2016 = pd.read_csv('./test_2016_no_label.csv', thousands=',')
test_2016.head()
```

Out[3]:

	FIPS	County	MedianIncome	MigraRate	BirthRate	DeathRate	BachelorRate	Unemploy
0	17059	Gallatin County, IL	39634	-10.6	10.8	12.5	9.9	
1	6103	Tehama County, CA	40585	1.8	12.8	10.4	15.5	
2	42047	Elk County, PA	49274	-9.3	9.7	13.0	18.6	
3	47147	Robertson County, TN	58487	7.4	12.7	9.7	18.6	
4	39039	Defiance County, OH	52210	-5.3	11.1	10.2	16.7	

The functions in this section do not mutate train_2016 or test_2016

In [4]:
```
# drop columns that will not be used in feature space
def drop_cols(data, is_train):
    cols = ['FIPS', 'County', 'DEM', 'GOP'] if is_train else ['FIPS','County'
    return data.drop(columns=cols).to_numpy()
```

In [5]:
```
# returns a binary array where an entry of 1 is a Democratic county, 0 is Rep
def get_winners(train_data):
    return (0 + (train_data["DEM"] > train_data["GOP"])).to_numpy()
```

In [6]:
```
# standardizes features of np_data given mean and std
def standardize_features(np_data, mean, std):
    return np.divide(np.subtract(np_data, mean), std)
```

In [7]:
```
# Drops FIPS and County and uses standardization.
# Returns:
#     Y_train: array of labels (1 for Democrat, 0 for Republican)
#     X_train: matrix of standardized training instances
#     X_test: matrix of standardized testing instances
def representation1(train_df, test_df):
    Y_train = get_winners(train_df)
    X_train_pure = drop_cols(train_df, True)
    mean_X_train = np.mean(X_train_pure, axis=0)
    std_X_train = np.std(X_train_pure, axis=0)
    X_train = standardize_features(X_train_pure, mean_X_train, std_X_train)
    X_test_pure = drop_cols(test_df, False)
    X_test = standardize_features(X_test_pure, mean_X_train, std_X_train)

    return Y_train, X_train, X_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.

In [8]:
```
# Model 1: KNN
# Inputs: X is a 2D np array of training instances, Y is a 1D array of corres
# Returns:
#     clf: knn model trained on X,Y with different hyperparameter values thro
def knn(X, Y, grid):
    clf = GridSearchCV(estimator=KNeighborsClassifier(), param_grid=grid, n_j
    clf.fit(X, Y)
    return clf
```

In [9]:
```
# Model 2: Kernelized SVM
# Inputs: X is a 2D np array of training instances, Y is a 1D array of corres
# Returns:
#     clf: SVM model trained on X,Y with different hyperparameter values thro
def svm(X, Y, grid):
    clf = GridSearchCV(estimator=SVC(), param_grid=grid, n_jobs=-1, scoring=m
    clf.fit(X, Y)
    return clf
```

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 [10]:
```
# Using representation 1
Y_train, X_train, X_test = representation1(train_2016, test_2016)
```

In [11]:
```
# Model 1: KNN
# Cross validation with hyperparameter grid search in KNN
# p = 1 is Manhattan distance, p = 2 is Euclidean distance, p = 3-10 is Minko
ps = [i for i in range(1, 11)]
# classify new data based on proximity to k neighbors
ks = [i for i in range(1, 6)]
knn_grid = {'p': ps, 'n_neighbors': ks}
knn_classifier = knn(X_train, Y_train, knn_grid)
best_p = knn_classifier.best_estimator_.get_params()['p']
best_k = knn_classifier.best_estimator_.get_params()['n_neighbors']
best_score = knn_classifier.best_score_
print(f'Optimal parameters are k={best_k}, and p={best_p}, with a score of {b
```

```
Optimal parameters are k=3, and p=2, with a score of 0.71046783625731
```

In [ ]:
```
# Model 2: Kernelized SVM
# Cross validation with hyperparameter grid search in SVM
# C = regularization parameter
# kernel = kernel type to be used in the algorithm
# degree = degree of polynomial kernel function
# gamma = kernel coefficient for 'rbf', 'poly', and 'sigmoid'
# coef0 = independent term in kernel function, only significant in 'poly' and
svm_grid = {'C': [i for i in range(1, 10)],
       'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
       'degree': [i for i in range(1, 5)],
       'gamma': [i for i in np.arange(0.1, 1, 0.1)],
       'coef0': [i for i in range(1, 6)]}
svm_classifier = svm(X_train, Y_train, svm_grid)
```

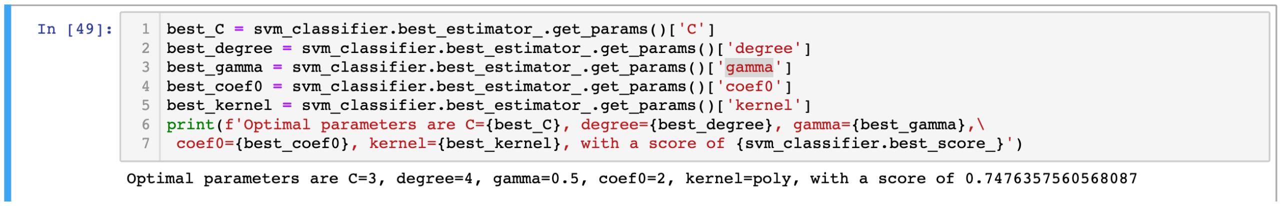
In [24]:
```
# extract and print out optimal hyperparameters from Grid Search object
def get_best_svm_params(svm_grid):
    best_C = svm_grid.best_estimator_.get_params()['C']
    best_degree = svm_grid.best_estimator_.get_params()['degree']
    best_gamma = svm_grid.best_estimator_.get_params()['gamma']
    best_coef0 = svm_grid.best_estimator_.get_params()['coef0']
    best_kernel = svm_grid.best_estimator_.get_params()['kernel']
    print(f'Optimal parameters are C={best_C}, degree={best_degree}, gamma={b
 coef0={best_coef0}, kernel={best_kernel}, with a score of {svm_grid.best_sco
```

In [ ]:
```
get_best_svm_params(svm_classifier)
```

In [12]:

```
# Grid search takes a very long time, screenshot of output
Image("SVM Classifier Grid Search.png")
```

Out[12]:



In [49]:

```
best_C = svm_classifier.best_estimator_.get_params()['C']
best_degree = svm_classifier.best_estimator_.get_params()['degree']
best_gamma = svm_classifier.best_estimator_.get_params()['gamma']
best_coef0 = svm_classifier.best_estimator_.get_params()['coef0']
best_kernel = svm_classifier.best_estimator_.get_params()['kernel']
print(f'Optimal parameters are C={best_C}, degree={best_degree}, gamma={best_gamma},\
 coef0={best_coef0}, kernel={best_kernel}, with a score of {svm_classifier.best_score_}')
```

```
Optimal parameters are C=3, degree=4, gamma=0.5, coef0=2, kernel=poly, with a score of 0.7476357560568087
```

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?

The data was first loaded from the CSV file into DataFrames. Initially, we had trouble reading in the 'MedianIncome' column as a numerical type because of the commas, so we specified a 'thousands' parameter when reading in the CSV files into DataFrames.

The winner of each county was determined via the vote counts from the 'DEM' and 'GOP' columns and Y_train was created to keep track of the binary labels (1 for Democrat, 0 for Republican).

We then had to come up with the relevant feature representation which would take the respective DataFrames and turn them into X_train and X_test. First, we decided that we would drop FIPS and County from our set of relevant features. FIPS is just an ID number and does not give us any information, and the textual county name is difficult to convert into a feature for a basic solution. So, we initially dropped those columns.

We also dropped the 'DEM' and 'GOP' columns from X_train, as these columns would not even be available for X_test (since these columns alone tell us the corresponding labels).

The remaining features were then standardized to have mean 0 and unit variance. So each resulting instance became a 6 dimensional vector with label 1 or 0.

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

The first learning method used was K-Nearest Neighbor (KNN). KNN seemed like a natural and simple choice as we hypothesized that counties with "similar" features may tend to vote the same way and be clustered together.

The second learning method used was a kernelized Support Vector Machine (SVM). As we learned from lecture and the projects, SVM is a very natural approach to a binary classification problem. In this case, we hypothesized that we could try to find a decision boundary to separate Democratic counties from Republican ones, and the various kernels would allow us to try out both linear and non-linear decision boundaries. This made us think that a kernelized SVM approach could be robust to fitting all kinds of data.

2.4.3 How did you do the model selection?

Model selection was done using a grid search for hyperparameters and 5-fold cross validation (via sklearn). We chose to use cross validation in order to get the most out of our limited training data, and a grid search to tune the hyperparameters.

For KNN the hyperparameter search included k, the number of nearest neighbors to consider, and p, the power parameter for the Minkowski metric for all p >= 3 (p = 1 represents Manhattan distance, p = 2 represents Euclidean distance). The optimal parameters found were k=3 and p=2.

For the kernelized SVM the hyperparameter search was for C (the regularization parameter), kernel(kernel type to be used in the algorithm), degree (degree of polynomial kernel function), gamma (kernel coefficient for 'rbf', 'poly', and 'sigmoid'), and coef0(independent term in kernel function, only significant in 'poly' and 'sigmoid').

The optimal combination was C=3, degree=4, gamma=0.5, coef0=2, and kernel=poly.

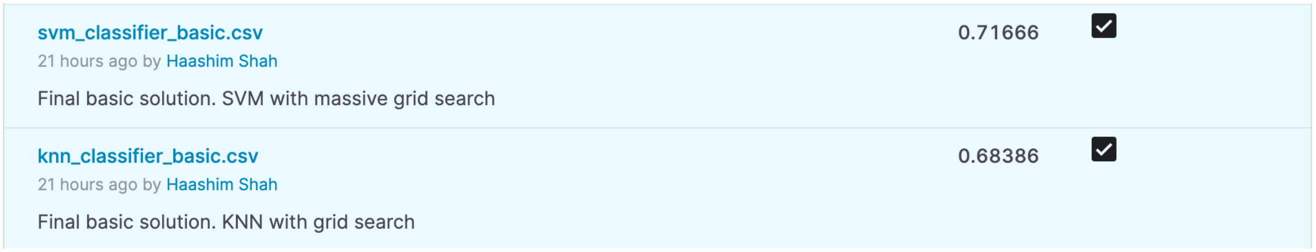
2.4.4 Does the test performance reach a given baseline 68% performance? (Please include a screenshot of Kaggle Submission)

The KNN solution reaches 0.68386 accuracy and the SVM solution reaches 0.71666 accuracy. So they both pass the Basic Solution.

(Screenshot of Kaggle Submission below)

In [13]:
```
Image("Kaggle Submission Basic.png")
```

Out[13]:



svm_classifier_basic.csv 0.71666 ✓
21 hours ago by Haashim Shah
Final basic solution. SVM with massive grid search

knn_classifier_basic.csv 0.68386 ✓
21 hours ago by Haashim Shah
Final basic solution. KNN with grid search

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.

3.1.1 Expansion to 12 features

In [14]:

```
train_2012 = pd.read_csv('./train_2012.csv', thousands=',')
test_2012 = pd.read_csv('./test_2012_no_label.csv', thousands=',')
```

In [15]:

```
# drops columns that won't be in the feature space, renames the columns of th
def prep(data, case):
    if case < 2:
        if case == 0: #2012 train data
            cols = ['County', 'DEM', 'GOP']
        else: #2012 test data
            cols = ['County']
        temp = data.drop(columns=cols)
        out = temp.rename(columns={'FIPS' : 'FIPS','MedianIncome' : 'I','Migr
          'BirthRate' : 'Bi', 'DeathRate' : 'D','BachelorRate' : 'Ba','Unem
    else:
        if case == 2: #2016 train data
            cols = ['County', 'DEM', 'GOP']
        else: #2016 test data
            cols = ['County']
        out = data.drop(columns=cols)
    return out
```

In [16]:

```
# creates 6 new features from differences between 2012 and 2016 data, drops r
def trend_features(df):
    df['dIncome'] = df['MedianIncome'] - df['I']
    df['dMigra'] = df['MigraRate'] - df['M']
    df['dBirth'] = df['BirthRate'] - df['Bi']
    df['dDeath'] = df['DeathRate'] - df['D']
    df['dBachelor'] = df['BachelorRate'] - df['Ba']
    df['dUnemployed'] = df['UnemploymentRate'] - df['U']
    return df.drop(columns = ['I','M','Bi','D','Ba','U','FIPS'])
```

In [17]:
```
# Inputs: train and test CSV data
# Returns:
#     numpy arrays for training and testing, 6 given features in 2016, 6 feat
def makefeatures(tr12,ts12,tr16,ts16):
    training = trend_features(pd.concat([prep(tr12,0),prep(tr16,2)], axis = 1
    testing = trend_features(pd.concat([prep(ts12,1),prep(ts16,3)], axis = 1,

    return training.to_numpy(), testing.to_numpy()
```

In [18]:
```
# standardizes features of data on its mean and std
def standardize(data):
    mean = np.mean(data, axis=0)
    stddev = np.std(data, axis=0)
    return np.divide(np.subtract(data, mean), stddev)
```

In [19]:
```
# returns the class weight for the DEM class
def get_class_weight(train_data):
    d = len(train_data) / np.sum(train_data)
    return [d]
```

In [20]:
```
# training data output has 12 features, rather than the 6 we had before
# ytr is a boolean array recording whether each county in the training set vo
# std_train and std_test are numpy arrays of training and testing featue vect
def creativerep(train12, test12, train16, test16):
    tr12 = copy.deepcopy(train12)
    ts12 = copy.deepcopy(test12)
    tr16 = copy.deepcopy(train16)
    ts16 = copy.deepcopy(test16)
    ytr = get_winners(train16)
    weight = get_class_weight(ytr)
    xtrain, xtest = makefeatures(tr12,ts12,tr16,ts16)
    std_train = standardize(xtrain)
    std_test = standardize(xtest)
    return ytr,std_train,std_test, weight
```

In [21]:
```
Y_tr , X_tr, X_ts, weights = creativerep(train_2012,test_2012,train_2016,test_
```

3.1.2 Expanded SVM Grid Search

In [25]:
```
# Trying out a couple higher degree polynomials for SVM as they seemed promis
# training model on all 12 features.
svm_expanded_grid = {'C': [2, 3, 4, 5],
       'degree': [i for i in range(5, 11)],
       'gamma': [0.5, 1],
       'coef0': [1, 2],
       'kernel': ['poly']}
svm_expanded_grid = svm(X_tr, Y_tr, svm_expanded_grid)
#svm_expanded_grid = svm(X_train, Y_train, svm_expanded_grid)
```

In [26]:
```
get_best_svm_params(svm_expanded_grid)
```

```
Optimal parameters are C=2, degree=5, gamma=0.5, coef0=2, kernel=poly, with a
score of 0.7193984962406016
```

3.1.3 Neural Network

In [27]:
```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

In [28]:
```
# generates tensors for later use
class_weights = torch.FloatTensor(weights)

[Y_train, Y_val] = np.split(Y_tr, [1300])
[X_train, X_val] = np.split(X_tr, [1300])

xtrTensor = torch.from_numpy(X_tr)
ytrTensor = torch.from_numpy(Y_tr)
ytrTensor = ytrTensor.type(torch.LongTensor)

xvalTensor = torch.from_numpy(X_val)
yvalTensor = torch.from_numpy(Y_val)
yvalTensor = yvalTensor.type(torch.LongTensor)

xtsTensor = torch.from_numpy(X_ts)
ytsTensor = torch.from_numpy(Y_tr)

print(class_weights)
```

```
tensor([6.9111])
```

In [29]:
```
# Inputs: train_labels: 1D Numpy array of training labels
# Returns:
#     ngop: number of GOP-voting counties
#     wgop: GOP class weight as as defined in weighted_accuracy
#     ndem: number of DEM-voting counties
#     wdem: DEM class weight as defined in weighted_accuracy
def voter_stats(train_labels):
    ndem = np.sum(train_labels)
    ngop = len(train_labels)-ndem
    wdem = (ndem + ngop)/ndem
    wgop = (ndem + ngop)/ngop
    return ngop, wgop, ndem, wdem
```

In [30]:
```
ng, wg, nd, wd = voter_stats(Y_train)
ngv, _, ndv, _ = voter_stats(Y_val)
print(ng, nd)
```

```
1106 194
```

In [31]:
```
# Inputs: predictions, true labels, number of counties voting for each party,
# Returns: weighted accuracy for a minibatch of samples
def iter_weighted_accuracy(preds, true, ng, wg, nd, wd):
    d_correct = 0
    g_correct = 0
    for pred_i, true_i in zip(preds, true):
        d_correct += 0 + (pred_i == true_i and true_i == 1)
        g_correct += 0 + (pred_i == true_i and true_i == 0)
    return (wg * g_correct + wd * d_correct) / (wg * ng + wd * nd)
```

In [32]:
```
dataset = torch.utils.data.TensorDataset(xtrTensor,ytrTensor)
valset = torch.utils.data.TensorDataset(xvalTensor,yvalTensor)
testset = torch.utils.data.TensorDataset(xtsTensor, ytsTensor)

trainloader = torch.utils.data.DataLoader(dataset, batch_size=4,shuffle=True)
validationloader = torch.utils.data.DataLoader(valset, batch_size=4,shuffle=T
testloader = torch.utils.data.DataLoader(testset, batch_size=1,shuffle=False)
```

In [33]:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(12, 8)
        self.fc2 = nn.Linear(8, 4)
        self.fc3 = nn.Linear(4, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
net = Net()
print(net)
```

```
Net(
  (fc1): Linear(in_features=12, out_features=8, bias=True)
  (fc2): Linear(in_features=8, out_features=4, bias=True)
  (fc3): Linear(in_features=4, out_features=1, bias=True)
)
```

In [34]:

```
criterion = nn.BCEWithLogitsLoss(weight = class_weights)
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

In [35]:
```
training_acc = []
validation_acc = []

num_epochs = 13
for epoch in range(num_epochs):  # loop over the dataset multiple times
    weightedacc = 0
    for i, data in enumerate(trainloader):
        # get the inputs; data is a list of [xtr, ytr]
        inputs, labels = data
        labels = labels.unsqueeze(1)

        # zero the parameter gradients
        optimizer.zero_grad()
        outputs = net(inputs.float())
        labels = labels.type_as(outputs)

        # calculate loss and perform backpropagation
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

    with torch.no_grad():
        w_acc = 0
        for data in trainloader:
            inputs, labels = data
            outputs = net(inputs.float())
            predicted = np.where(outputs.data > 0.5, 1, 0)
            w_acc += iter_weighted_accuracy(labels.numpy(), predicted, ng, wg

        w_acc_v = 0
        for data in validationloader:
            inputs, labels = data
            outputs = net(inputs.float())
            predicted = np.where(outputs.data > 0.5, 1, 0)
            w_acc_v += iter_weighted_accuracy(labels.numpy(), predicted, ngv,
        print('Epoch: %d Train Error: %d Validation Error: %d' \
              % (epoch + 1, w_acc * 100, w_acc_v * 100))
        training_acc.append(w_acc)
        validation_acc.append(w_acc)
print('Finished Training')
```

```
Epoch: 1 Train Error: 60 Validation Error: 55
Epoch: 2 Train Error: 60 Validation Error: 55
Epoch: 3 Train Error: 60 Validation Error: 55
Epoch: 4 Train Error: 64 Validation Error: 58
Epoch: 5 Train Error: 86 Validation Error: 81
Epoch: 6 Train Error: 89 Validation Error: 84
Epoch: 7 Train Error: 92 Validation Error: 85
Epoch: 8 Train Error: 84 Validation Error: 82
Epoch: 9 Train Error: 81 Validation Error: 73
Epoch: 10 Train Error: 85 Validation Error: 80
Epoch: 11 Train Error: 86 Validation Error: 80
Epoch: 12 Train Error: 83 Validation Error: 76
Epoch: 13 Train Error: 85 Validation Error: 81
Finished Training
```

In [36]:

```
# returns predictions from neural net
def make_predictions(net, testloader):
    with torch.no_grad():
        preds = []
        for data in testloader:
            inputs, _ = data
            outputs = net(inputs.float())
            predicted = np.where(outputs.data > 0.5, 1, 0)
            preds.append(predicted[0][0])
    return preds
p = make_predictions(net, testloader)
```

3.1.4 SVM with Balanced Weight Class and Grid Search

In [ ]:

```
def svm_creative(X, Y, grid):
    clf = GridSearchCV(estimator=SVC(class_weight='balanced'), param_grid=gri
    clf.fit(X, Y)
    return clf
svm_creative_classifier = svm_creative(X_train, Y_train, svm_grid)
```

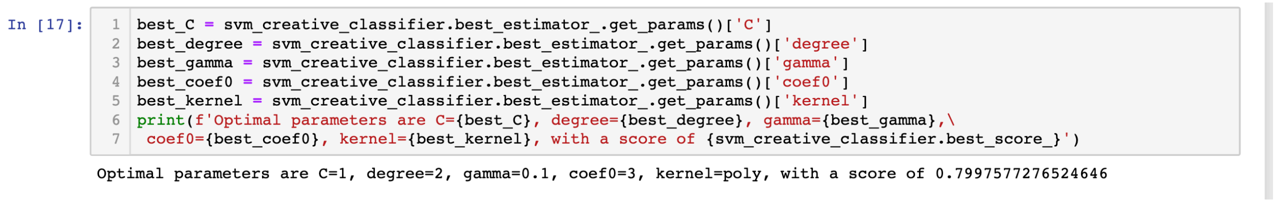
In [ ]:

```
get_best_svm_params(svm_creative_classifier)
```

In [37]:

```
# Grid search takes a very long time, screenshot of output
Image("SVM Balanced Grid Search.png")
```

Out[37]:



In [17]:

```
best_C = svm_creative_classifier.best_estimator_.get_params()['C']
best_degree = svm_creative_classifier.best_estimator_.get_params()['degree']
best_gamma = svm_creative_classifier.best_estimator_.get_params()['gamma']
best_coef0 = svm_creative_classifier.best_estimator_.get_params()['coef0']
best_kernel = svm_creative_classifier.best_estimator_.get_params()['kernel']
print(f'Optimal parameters are C={best_C}, degree={best_degree}, gamma={best_gamma},\
 coef0={best_coef0}, kernel={best_kernel}, with a score of {svm_creative_classifier.best_score_}')
```

```
Optimal parameters are C=1, degree=2, gamma=0.1, coef0=3, kernel=poly, with a score of 0.7997577276524646
```

3.2 Explanation in Words:

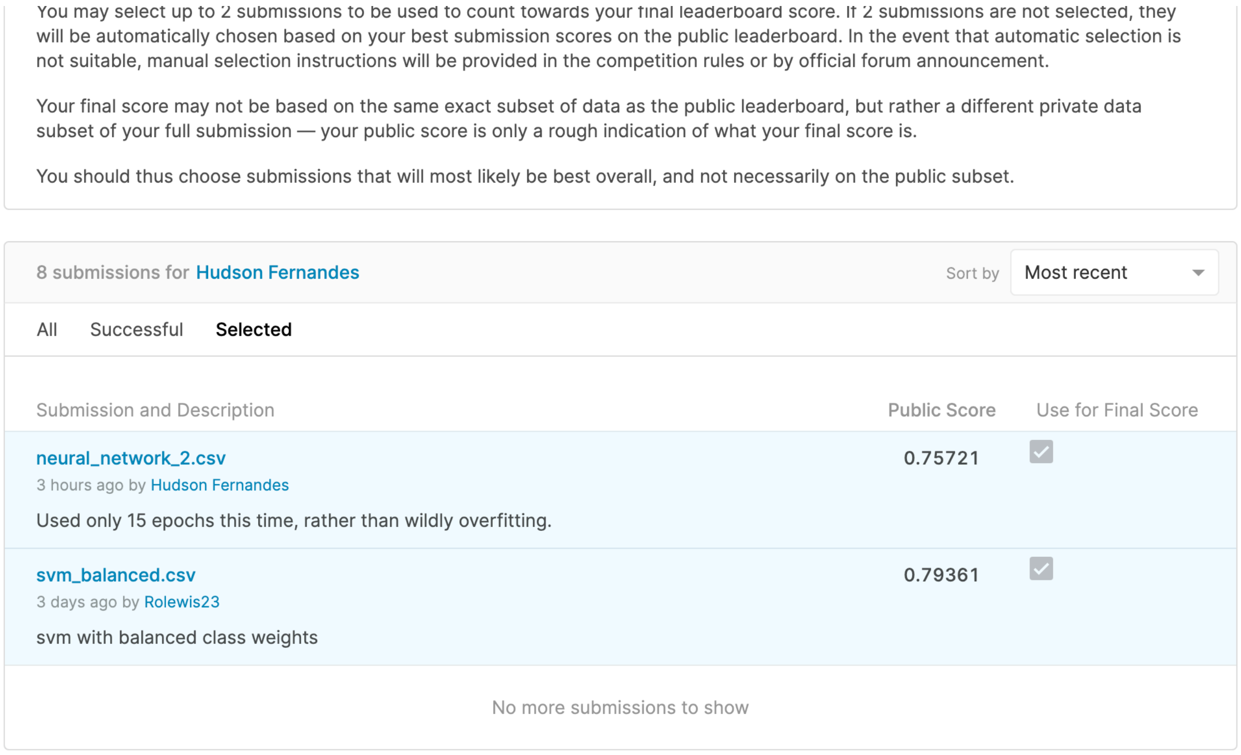
3.2.1 How much did you manage to improve performance on the test set compared to part 2? Did you reach the 75% accuracy for the test in Kaggle? (Please include a screenshot of Kaggle Submission)

We managed to improve performance by ~8% compared to part 2. From 71.67% in part 2, we managed to improve performance to 79.36% accuracy for the creative test set in Kaggle for the SVM with Balanced Class Weights, and ~4% to 75.72% accuracy for the creative test set in Kaggle for the Neural Network.

In [38]:

```python
Image("Kaggle Submission Creative.png")
```

Out[38]:



You may select up to 2 submissions to be used to count towards your final leaderboard score. If 2 submissions are not selected, they will be automatically chosen based on your best submission scores on the public leaderboard. In the event that automatic selection is not suitable, manual selection instructions will be provided in the competition rules or by official forum announcement.

Your final score may not be based on the same exact subset of data as the public leaderboard, but rather a different private data subset of your full submission — your public score is only a rough indication of what your final score is.

You should thus choose submissions that will most likely be best overall, and not necessarily on the public subset.

8 submissions for Hudson Fernandes

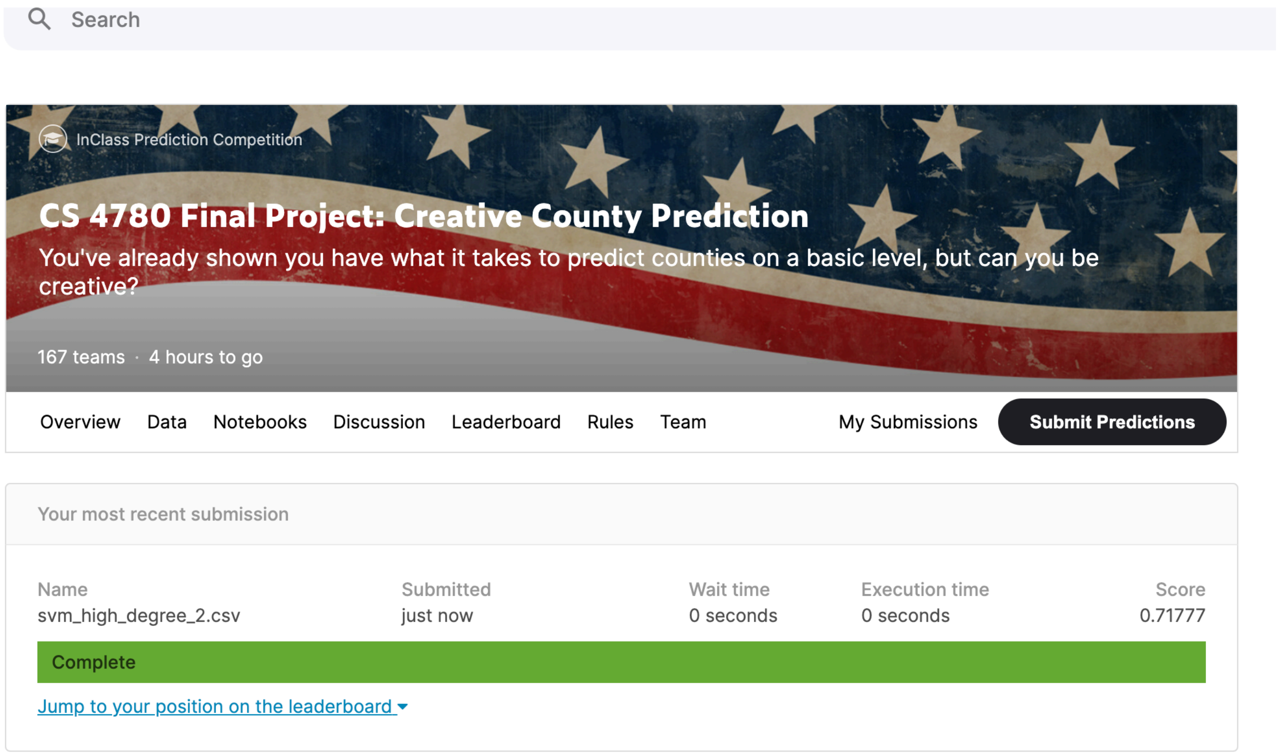
Submission and Description	Public Score	Use for Final Score
neural_network_2.csv 3 hours ago by Hudson Fernandes Used only 15 epochs this time, rather than wildly overfitting.	0.75721	✓
svm_balanced.csv 3 days ago by Rolewis23 svm with balanced class weights	0.79361	✓

No more submissions to show

In [39]:

```python
# Our first creative attempt did not hit the 75% threshold
Image("Kaggle Submission High Degree SVM.png")
```

Out[39]:


Search

InClass Prediction Competition

CS 4780 Final Project: Creative County Prediction

You've already shown you have what it takes to predict counties on a basic level, but can you be creative?

167 teams · 4 hours to go

Your most recent submission

Name	Submitted	Wait time	Execution time	Score
svm_high_degree_2.csv	just now	0 seconds	0 seconds	0.71777

Complete

Jump to your position on the leaderboard

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

Attempt 1: SVM with Expanded Grid Search

First it was noticed that the Kernelized SVM seemed to be getting better performance via high degree polynomials during the large grid search we conducted during the basic solution. To this effect we redid a smaller grid search but this time with high degree polynomials.

The resulting "optimal solution" got a smaller validation score than what we had before, and sure enough only achieved a 0.67613 accuracy on Kaggle. So higher degree polynomials were not effective.

It was time to add more features for our ML algorithms to learn with. To this effect, we crafted 6 additional features from the 2012 data, representing the trend in each previously existing feature from 2012. Unfortunately, some of these trend features likely only have subtle effects on voting data, as our SVM achieved just 0.71939 accuracy on the dataset.

Attempt 2: Neural Network

Seeking to better leverage our additional features, we constructed a simple feed-forward network with Pytorch taking all 12 features as inputs. We chose weights of 12 - 8 - 4 - 1 in an attempt to provide enough nodes for the model to meaningfully interpret all 12 features while mitigating overfitting. We used BCEWithLogitsLoss, as it provides a sigmoid smoothing of binary cross entropy loss (which was a natural choice for binary classification). We then trained our net on a training set comprised of the first 1300 counties in the provided training data, with the rest used as a validation sample. In training our neural network, we found that the network continued to decrease its training accuracy long past the validation accuracy had stopped decreasing. When we trained the network for 150 epochs, it reached a training accuracy of 94% while reaching just 71% on the Kaggle dataset. We then limited the training to 15 epochs to combat overfitting, which resulted in a test accuracy of 75%. Despite this modest success, we concluded that our neural network was unlikely to meet the desired 80% benchmark.

Attempt 3 (Selected): SVM with Balanced Class Weights

Having abandoned our neural network approach, we turned back to our initial SVM. This time we weighted samples of each class in an inversely proportional manner, rather than training with even weights. Meaning we trained with different C parameters for examples of different classes, and those parameters were inversely proportional to the number of examples in that class. In essence this penalized the model for allowing Democrat examples to get close to the margin as it is costly to misclassify a Democrat county. This modification substantially improved our prediction results, which makes intuitive sense as our previous training had not properly reflected weighted_accuracy. Having modified our SVM in this manner, we applied a grid search like the one used in our basic solution and achieved an error of 0.7939

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 [ ]:
```
def gen_CSV(classifier, test_df, csv_path, X_test):
    df = pd.DataFrame(test_df, columns=["FIPS"])
    df["Result"] = classifier.predict(X_test)
    df.to_csv(csv_path, index=False, header=True)

def gen_nn_CSV(net, test_df, csv_path, testloader):
    dfnn = pd.DataFrame(test_2016, columns=["FIPS"])
    dfnn["Result"] = make_predictions(net, testloader)
    dfnn.to_csv(csv_path, index=False, header=True)

# Basic
gen_CSV(knn_classifier, test_2016, "knn_classifier_basic.csv", X_test)
gen_CSV(svm_classifier, test_2016, "knn_classifier_basic.csv", X_test)

# Creative
gen_CSV(svm_expanded_grid, test_2016, "svm_high_degree.csv", X_ts)
gen_nn_CSV(net, test_2016, "neural_network.csv", testloader)
gen_CSV(svm_)
# gen_CSV(svm_creative1, test_2016, "svm_high_degree.csv", X_test)
```

Part 5: Resources and Literature Used

scikit-learn KNN Documentation

scikit-learn SVM Documentation

scikit-learn GridSearchCV Documentation

Deep Learning with PyTorch: A 60 Minue Blitz