CS 4780/5780 Final Project.

Election Result Prediction for US Counties

Names and NetIDs for your group members: Aayush Chowdhry (ac2447), Adarsh Sriram (as2566)

Team Name: haitohhai

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.

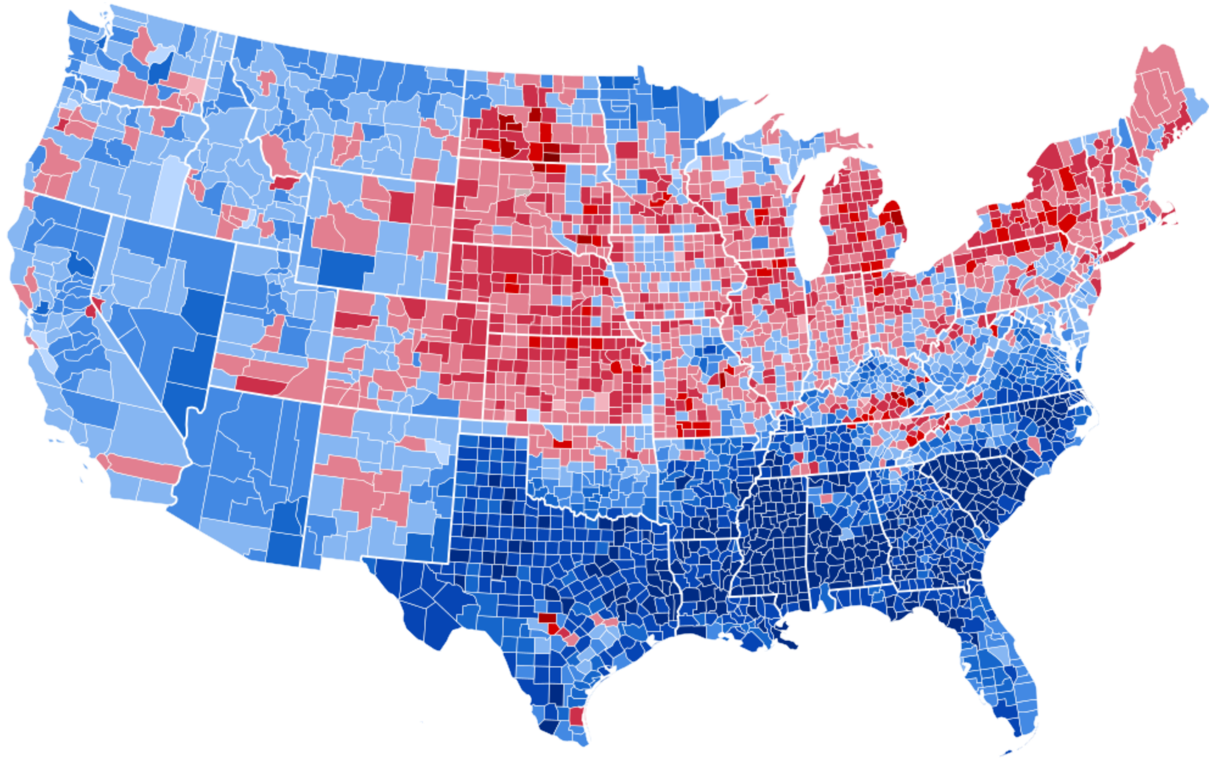
The task you will work on is forecasting election results. Economic and sociological factors have been widely used when making predictions on the voting results of US elections. Economic and sociological factors vary a lot among counties in the United States. In addition, as you may observe from the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

from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

import torch
from torch.utils.data import Dataset
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
```

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):
    """
    Compute weighted accuracy of predictions

    Parameters:
        pred: n predictions.
        true: n true labels.
    Returns:
        weighted accuracy of predictions based on number of positive and ne
    """
    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 * n
    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]:
```
# Loading training features.
df = pd.read_csv("train_2016.csv", sep=',',header=None, encoding='unicode_e
data = df.to_numpy()
data = data[1:, 1:]
ordA = ord("A")
for county in data:
    # This converts the state of the county to a unique number between 1 an
    county[0] = (ord(county[0][-2])-ordA)*26+(ord(county[0][-1])-ordA)
    county[3] = county[3].replace(",", "")
data = data.astype(np.float32)
tmp = np.copy(data[:,0])
data[:,0] = data[:,2]
data[:,2] = tmp
xTr = data[:, 2:]

# Generating Labels
yTr = np.sign(data[:,1]-data[:, 0])
yTr = np.where(yTr==-1, 0, yTr)
assert np.sum(yTr)==225 # sanity check

# Loading test features.
testdf = pd.read_csv("test_2016_no_label.csv", sep=',',header=None, encodin
xTe = testdf.to_numpy()
FIPS = xTe[1:, 0] # Storing to later write preds file.
xTe = xTe[1:, 1:]
ordA = ord("A")
for county in xTe:
    # This converts the state of the county to a unique number between 1 an
    county[0] = (ord(county[0][-2])-ordA)*26+(ord(county[0][-1])-ordA)
    county[1] = county[1].replace(",", "")
xTe = xTe.astype(np.float32)

def preprocess(xTr, xTe):
    """
    Preproces the data by normalizing to make the training features have ze
    standard-deviation 1.

    Parameters:
        xTr: nxd training data.
        xTe: mxd testing data.
    Returns:
        nxd matrix of pre-processed (normalized) training data.
        mxd matrix of pre-processed (normalized) testing data.
    """
    ntr, d = xTr.shape
    nte, _ = xTe.shape
    m = np.mean(xTr, axis=0)
    s = np.std(xTr, axis=0)
    xTr = (xTr-m)/s
    xTe = (xTe-m)/s
    return xTr, xTe
```

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 [4]:
```
# First training algorithm: weighted kNNClassifier
def kNNClassifier(xtrain, ytrain, xtest, k):
    """
    Weighted kNN Classifer

    Parameters:
        xtrain: nxd training features.
        ytrain: n training labels.
        xtest: mxd test features (features one wants predictions of)
        k: number of neighbours to run kNN with.
    Returns:
        binary predictions for xtest after learning from given data.
    """
    xtrain, xtest = preprocess(xtrain, xtest)
    neigh = KNeighborsClassifier(n_neighbors=k, weights="distance")
    neigh.fit(xtrain, ytrain)
    return neigh.predict(xtest)

# Second training algorithm: soft-margin SVM
def SVMClassifier(xtrain, ytrain, xtest, C, kernel):
    """
    Soft Margin SVM

    Parameters:
        xtrain: nxd training features.
        ytrain: n training labels.
        xtest: mxd test features (features one wants predictions of)
        C: Weightage given to slack variables.
        Kernel: The kernel with which to run the SVM.
    Returns:
        binary predictions for xtest after learning from given data.
    """
    xtrain, xtest = preprocess(xtrain, xtest)
    sv = SVC(C=C, kernel=kernel)
    sv.fit(xtrain, ytrain)
    return sv.predict(xtest)
```

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 [5]:
```
####################################### Validation #########################
def kFoldCross(xTr, yTr, k, classifier):
    """
    kFoldCross for Estimating Prediction Error and Calculating Training Err

    Parameters:
        xTr: nxd training features.
        yTr: n training labels.
        k: number of pieces to break xTr and yTr into. 1 piece is used as v
        classifier: function that takes training features, training labels,
                    to give predictions. (the classier one is validating wi
    Returns:
        Average training loss.
        Average validation loss.
    """
    totTrain = 0; totVal = 0;
    for i in range(0, k):
        # data is split into (k-1) pieces to train and 1 piece to validate
        splitIndices = (i*len(xTr)//k, (i+1)*len(xTr)//k)
        if 0 in splitIndices:
            xt = xTr[splitIndices[1]:]
            yt = yTr[splitIndices[1]:]
        elif k in splitIndices:
            xt = xTr[:splitIndices[0]]
            yt = yTr[:splitIndices[0]]
        else:
            xt = np.concatenate((xTr[:splitIndices[0]], xTr[splitIndices[1]
            yt = np.concatenate((yTr[:splitIndices[0]], yTr[splitIndices[1]
        xv = xTr[splitIndices[0]:splitIndices[1]]
        yv = yTr[splitIndices[0]:splitIndices[1]]

        totTrain+= weighted_accuracy(classifier(xt, yt, xt), yt) # compute
        totVal+= weighted_accuracy(classifier(xt, yt, xv), yv) # compute va

    return totTrain/k, totVal/k # return average training and test error


####################################### Model Selection ###################

# k = 5
# bestAcc = -float("inf")
# # Generating Predictions
# for i in range(1, 11): # parameter selection for number of neighbours fro
#     print("Running "+str(k)+"-FoldCross for "+str(i)+"-NNClassifier.")
#     trainAcc, valAcc = kFoldCross(xTr, yTr, 5, lambda xt, yt, xv: kNNClas
#     print("Average Training Accuracy: "+str(trainAcc))
#     print("Average Validation Accuracy: "+str(valAcc))
#     print()
#     if valAcc>bestAcc:
#         bestAcc = valAcc
#         bestK = i

# Manually found SVM has better performance using printed validation values

k = 5 # for kfoldcross
bestAcc = -float("inf")
```

```
Cs = [50.30, 50.31, 50.32, 50.33, 50.34] # good range of C values found thr
for C in Cs:
    print("Running SVM with C "+str(C)+" and rbf kernel.")
    trainAcc, valAcc = kFoldCross(xTr, yTr, 5, lambda xt, yt, xv: SVMClassi
    print("Average Training Accuracy: "+str(trainAcc))
    print("Average Validation Accuracy: "+str(valAcc))
    print()
    if valAcc>bestAcc:
        bestAcc = valAcc
        bestC = C
```

2.4 Explanation in Words:

2.4.1 How did you preprocess the dataset and features?

We preprocessed the dataset by extracting the respective values for each county which made up the dimensions of the feature vector. We also converted the state initials to unique values from 1 to 26x26 for each state to add to the feature vector. We did the same thing for the test set. To extract the training labels, we simply assigned 0 when the GOP votes where greater than or equal to the DEM votes and 1 otherwise for each county.

As part of our classifiers, we had a preprocess function (as defined above) that we used to normalize out feature vectors in the training set. It essentially ensured that the mean each dimension of the features was 0 and the standard deviation 1. We then applied the same transformation to the test features (with the mean and sd for each dimension in the training set).

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

The first learning algorithm we used was weighted kNN classification. We chose this algorithm because the dimensions of the feature vectors was relatively low (6) and because it is likely that similar counties vote similarly.

The second learning algorithm we used was a kernelized soft margin SVM. We chose this algorithm as we thought, given similar counties would vote similarly, a kernelized SVM would be able to identify spaces in which counties vote GOP or DEM and allowing for slack would reduce the chances of overfitting.

2.4.3 How did you do the model selection?

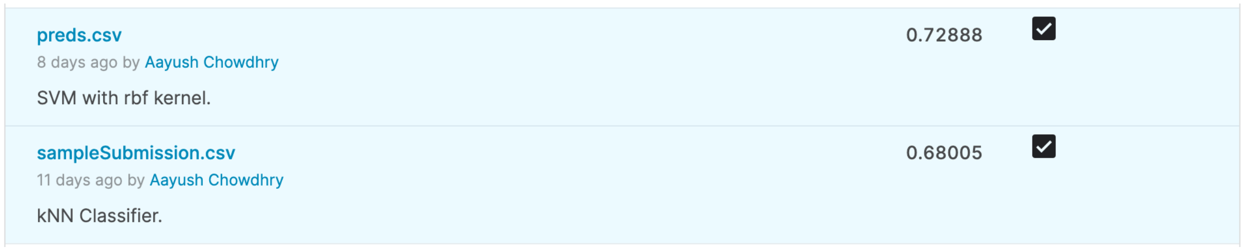
We used a kFoldCross validation where we broke the training data set into k pieces and used each piece as a validation set while training the classifier on the remaining k-1 pieces of data. We usually used a 5-foldCross validation as we found it gave a pretty accurate measure of the test accuracy.

We tried training the kNN classifier with different number of neighbours and used kFoldCross validation to get the average validation accuracy. We then chose the parameter k (number of neighbours) for the classifier with greatest average kFoldCross validation accuracy and computed the test predictions with that k over the entire training set.

For the SVM, the validation process was the same but we tried training with different kernels and C values. We found that the rbf kernel worked best by manually printing values and by a manual telescopic search, found that the range of Cs above provide for the highest validation error. We then picked the best C according to the best average kFoldCross validation error and trained the SVM on the entire training set and then generated predictions on the test set.

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

Both the kNN classifier and the kernelized soft margin SVM were able to beat the 68% baseline accuracy. However, the SVM performed better than the kNN algorithm.



preds.csv 0.72888 ✓
8 days ago by Aayush Chowdhry
SVM with rbf kernel.

sampleSubmission.csv 0.68005 ✓
11 days ago by Aayush Chowdhry
kNN Classifier.

Type *Markdown* and LaTeX: α^2

Part 3: Creative Solution

3.1 Open-ended Code:

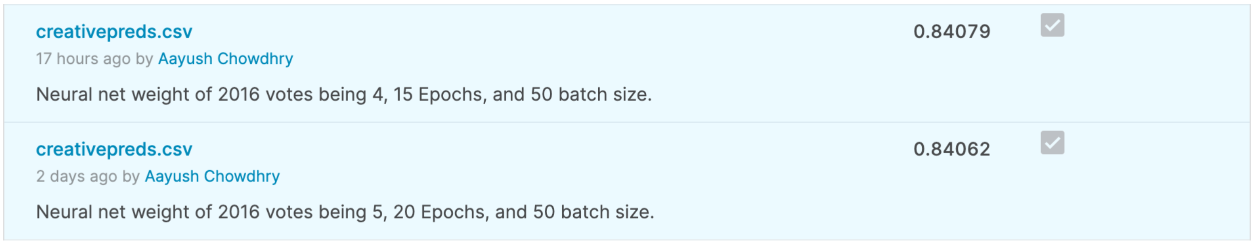
You may follow the steps in part 2 again but making innovative changes like creating new features, using new training algorithms, etc. Make sure you explain everything clearly in part 3.2. Note that reaching the 75% creative baseline is only a small portion of this part. Any creative ideas will receive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

In [6]:
```
########################################## Data Loading Creative ##########
# Loading training features from 2016.
df = pd.read_csv("./data/train_2016.csv", sep=',',header=None, encoding='un
data = df.to_numpy()
data = data[1:, 1:]
ordA = ord("A")
for county in data:
    # This converts the state of the county to a unique number between 1 an
    county[0] = (ord(county[0][-2])-ordA)*26+(ord(county[0][-1])-ordA)
    county[3] = county[3].replace(",", "")
data = data.astype(np.float32)
tmp = np.copy(data[:,0])
data[:,0] = data[:,2]
data[:,2] = tmp
xTr = data[:, 2:]

# yTr is currently difference in votes in 2016 between GOP and DEM.
yTr = data[:,1]-data[:, 0]
assert np.sum(np.where(yTr<=0, 0, 1))==225 # Sanity check

# Loading training features from 2012.
df = pd.read_csv("./data/train_2012.csv", sep=',',header=None, encoding='un
data = df.to_numpy()
data = data[1:, 1:]
ordA = ord("A")
for county in data:
    # This converts the state of the county to a unique number between 1 an
    county[0] = (ord(county[0][-2])-ordA)*26+(ord(county[0][-1])-ordA)
    county[3] = county[3].replace(",", "")
data = data.astype(np.float32)
tmp = np.copy(data[:,0])
data[:,0] = data[:,2]
data[:,2] = tmp
xTr2 = data[:, 3:] # Do not want state to be repeated hence col index is 3

# yTr is currently difference in votes in 2012 between GOP and DEM.
yTr2 = data[:,1]-data[:, 0]
assert np.sum(np.where(yTr2<=0, 0, 1))==322 # Sanity Check

# Generate final training set.
xTr = np.hstack((xTr, xTr2)) # Stack 2016 features and 2012 features horizo
# Generate labels. Weight 2016 votes more as they are the lastest ones and
weight2016 = 4 # Manually set after validation.
yTr = np.sign(weight2016*yTr + yTr2)
yTr = np.where(yTr==-1, 0, yTr)
yTr = yTr.reshape((len(yTr), 1))


# Loading test features from 2016 and 2012 similar to training features.
testdf = pd.read_csv("./data/test_2016_no_label.csv", sep=',',header=None,
xTe = testdf.to_numpy()
FIPS = xTe[1:, 0]
xTe = xTe[1:, 1:]
ordA = ord("A")
for county in xTe:
    county[0] = (ord(county[0][-2])-ordA)*26+(ord(county[0][-1])-ordA)
    county[1] = county[1].replace(",", "")
xTe = xTe.astype(np.float32)

testdf = pd.read_csv("./data/test_2012_no_label.csv", sep=',',header=None,
xTe2 = testdf.to_numpy()
xTe2 = xTe2[1:, 1:]
ordA = ord("A")
for county in xTe2:
    county[0] = (ord(county[0][-2])-ordA)*26+(ord(county[0][-1])-ordA)
    county[1] = county[1].replace(",", "")
xTe2 = xTe2.astype(np.float32)
xTe = np.hstack((xTe,xTe2[:,1:])) # xTe2 is sliced as state isn't repeated

def preprocess(xTr, xTe):
    """
    Preproces the data by normalizing to make the training features have ze
    standard-deviation 1.

    Parameters:
        xTr: nxd training data.
        xTe: mxd testing data.
    Returns:
        nxd matrix of pre-processed (normalized) training data.
        mxd matrix of pre-processed (normalized) testing data.
    """
    ntr, d = xTr.shape
    nte, _ = xTe.shape
    m = np.mean(xTr, axis=0)
    s = np.std(xTr, axis=0)
    xTr = (xTr-m)/s
    xTe = (xTe-m)/s
    return xTr, xTe


######################################### Creative Classifier (Neural Net)

# Create custom dataset for training and test data.
class NPDataset(Dataset):
    def __init__(self, x, y=None):
        """
        Constructor for Dataset.

        Parameters:
            x: nxd features.
            optional y: n training labels if x is training data set.
        """
        # Where the initial logic happens like reading a csv, doing data au
        assert y is None or len(x)==len(y)
        self.xdata = x; self.ydata = y

    def __len__(self):
        """
        Overwrite python's len function to return the count of samples (an
        """
        return len(self.xdata)

    def __getitem__(self, idx):
        """
        Function to fetch data at index.

        Parameters:
            idx: index to fetch data from.
        Returns:
            feature vector and coressponding label at idx if dataset is tra
            feature vector at idx if dataset is test set (y is None)
        """
        if self.ydata is None:
            return self.xdata[idx]
        return self.xdata[idx], self.ydata[idx]

# Define network (3 hidden layers of 20, 20 and 10 neurons respectively. Re
class Net(nn.Module):
    def __init__(self, d):
        """
        Initialize most NN layers here.

        Parameters:
            d: dimension of the feature vectors.
        """
        super(Net, self).__init__()
        self.fc1 = nn.Linear(d, 20)
        self.fc2 = nn.Linear(20, 20)
        self.fc3 = nn.Linear(20, 10)
        self.fc4 = nn.Linear(10, 2) # output layer is 2 for cross entropy l

    def forward(self, x):
        """
        Defines in what order the input x is forwarded through all the NN l

        Parameters:
            x: nxd input vectors.
        Returns:
            nxd matrix generated after passing x through layers of neural n
        """
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x =  F.relu(self.fc3(x))
        x =  self.fc4(x)
        return x

    def predict(self,x):
        """
        Generate binary predictions.

        Parameters:
            x: nxd input vectors.
        Returns:
            n binary predictions for coressponding features.
        """
        pred = F.softmax(self.forward(x), dim=1) #Apply softmax to output.
        return torch.max(pred, 1)[1] #Pick the class with maximum weight

# Train neural net.
def trainNetwork(xtrain, ytrain):
    """
    Train neural network.

    Parameters:
        xtrain: nxd training features.
        ytrain: n training labels.
    Returns:
        trained neural network.
    """
    # Initialize network
    n, d = xtrain.shape
    net = Net(d)

    # Define loss function. Using weighted cross entropy loss due to imbala
    # Weighted based on number of positive and negative training labels.
    num_labels = len(ytrain)
    num_pos = sum(ytrain)
    frac_pos = num_pos/num_labels
    weight_pos = 1/frac_pos
    weight_zer = 1/(1-frac_pos)
    criterion = nn.CrossEntropyLoss(weight=torch.Tensor([weight_zer, weight

    # Define optimizer.
    optimizer = optim.Adam(net.parameters(), lr=0.01)

    # Initialize datasets and wrap in data loader.
    deepDataSet = NPDataset(xtrain, ytrain)
    dataloader = DataLoader(dataset = deepDataSet, batch_size = 50, shuffle

    # best combination of batch size, learning rate and number of epochs wa
    # by printing out training and validation accuracy and actively avoidin
    e_losses = []
    num_epochs = 15
    for e in range(num_epochs):
        e_losses += oneEpoch(net, optimizer, criterion, dataloader) # Train
    # plt.plot(e_losses)
    # plt.show()
    return net # return the model after training

def oneEpoch(net, optimizer, criterion, dataloader):
    """
    Trains one epoch of given neural network.

    Parameters:
        net: the neural network to train.
        optimizer: the optimizer to use.
        criterion: the loss function to apply.
        dataloader: wrapper of the training data.
    Returns:
        list of losser at each iteration for mini-batches in the epoch.
    """
    net.train()
    losses = []
    for bid, (input, target) in enumerate(dataloader):
        optimizer.zero_grad()
        output = net(input)
        loss = criterion(output, target.squeeze().long())
        loss.backward()
        optimizer.step()
        losses.append(loss.data.numpy())
    return losses

# Generate prediction (final classifier)
def deepnnClassifier(xt, yt, xv):
    """
    Neural Network (Deep Learner).

    Parameters:
        xt: nxd training features.
        yt: n training labels.
        xv: mxd test features (features one wants predictions of)
    Returns:
        binary predictions for xv after learning from given data.
    """
    xt, xv = preprocess(xt, xv)
    net = trainNetwork(xt, yt, xv)
    net.eval()
    preds = net.predict(torch.from_numpy(xv).type(torch.FloatTensor)).detac
    return preds.astype(np.int)


########################################## Validation #####################

def weighted_accuracy(pred, true):
    """
    Compute weighted accuracy of predictions

    Parameters:
        pred: n predictions.
        true: n true labels.
    Returns:
        weighted accuracy of predictions based on number of positive and ne
    """
    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 * n
    return weighted_accuracy

def kFoldCross(xTr, yTr, k, classifier):
    """
    kFoldCross for Estimating Prediction Error and Calculating Training Err

    Parameters:
        xTr: nxd training features.
        yTr: n training labels.
        k: number of pieces to break xTr and yTr into. 1 piece is used as v
        classifier: function that takes training features, training labels,
                    to give predictions. (the classier one is validating wi
    Returns:
        Average training loss.
        Average validation loss.
    """
    totTrain = 0; totVal = 0;
    for i in range(0, k):
        # data is split into (k-1) pieces to train and 1 piece to validate
        splitIndices = (i*len(xTr)//k, (i+1)*len(xTr)//k)
        if 0 in splitIndices:
            xt = xTr[splitIndices[1]:]
            yt = yTr[splitIndices[1]:]
        elif k in splitIndices:
            xt = xTr[:splitIndices[0]]
            yt = yTr[:splitIndices[0]]
        else:
            xt = np.concatenate((xTr[:splitIndices[0]], xTr[splitIndices[1]
            yt = np.concatenate((yTr[:splitIndices[0]], yTr[splitIndices[1]
        xv = xTr[splitIndices[0]:splitIndices[1]]
        yv = yTr[splitIndices[0]:splitIndices[1]]

        totTrain+= weighted_accuracy(classifier(xt, yt, xt), yt) # compute 
        totVal+= weighted_accuracy(classifier(xt, yt, xv), yv) # compute va

    return totTrain/k, totVal/k # return average training and test error

# Running validation after manually tuning hyperparameters such as learning
# batch size, loss criterion, optimization method etc.
trainAcc, valAcc = kFoldCross(xTr, yTr, 5, deepnnClassifier)
print("Average Training Accuracy: "+str(trainAcc))
print("Average Validation Accuracy: "+str(valAcc))
```

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)

With this creative solution, we managed to increase our test accuracy from a maximum of 72.888% to a max of 84.079% which was an improvement of 11.191% . We consistently achieved test accuracy higher that 75% with our highest two submission being greater that 84%.



creativepreds.csv 0.84079 ✓
17 hours ago by Aayush Chowdhry
Neural net weight of 2016 votes being 4, 15 Epochs, and 50 batch size.

creativepreds.csv 0.84062 ✓
2 days ago by Aayush Chowdhry
Neural net weight of 2016 votes being 5, 20 Epochs, and 50 batch size.

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

We achieved this by doing multiple things while building our classifier:

1) Although the preprocessing for the feature vectors was the same as described in the basic solution, we also extracted data for the counties from 2012 follwing a similar process but excluding the aforemention mapping of the state initials (as we already had that feature from 2016 data).

2) We changed how we calculated the labels for counties since we couldn't include the 2012 votes in the training features as the 2012 test set also didn't contain any votes. Since we were predicting the labels for 2016 elections in the counties, we weighted the difference in votes for 2016 more than that in 2012 in the training set. This weight became another hyperparameter to tune. We then calculated the label by saying if (weight2016*(GOP2016-DEM2016)+(GOP2012-DEM2012)) >= 0 then the label is 0, otherwise, it is 1.

3) We used deep learning to classify the samples instead of using kNN or SVM.

4) In the neural network, we used a cross entropy loss function such that the network is trained to predict the probabilities of seeing the binary labels (1 or 0) on the feature. We then made the binary prediction depending on which probability was higher.

5) We weighted the cross entropy loss according to the number of 1s and 0s in the true labels of the training set since we had imbalanced data.

6) We used kFoldCross (similar to that in the basic solution) to validate our models using different hyper parameters manually (by printing validation error).

7) Using kFoldCross validation, we found that a neural network with 3 hidden layers of 20, 20 and 10 neurons, a batch size of 50, with epochs from 10-25, and the Adam optimizer with initial learning rate of 0.01 provided for the best validation accuracy. We also found that the best validation accuracy came with initially computing the training labels by weighing 2016 votes by 2-5. We then used combinations of these hyperparameters (as there is some randomness in neural networks) to train our neural network on the entire training set and generate predictions on the test set. The best predictions gave us an accuracy of over 84% but the training accuracy was consistenly above 80%.

Type *Markdown* and LaTeX: α^2

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 [7]:
```
# Basic solution.
preds = SVMClassifier(xTr, yTr, xTe, bestC, "rbf").astype(np.int)
preddf = pd.DataFrame(data=preds, index=FIPS, columns=["Result"])
preddf.index.name = "FIPS"
preddf.to_csv("preds.csv")

# Creative Solution
preds = deepnnClassifier(xTr, yTr, xTe)
preddf = pd.DataFrame(data=preds, index=FIPS, columns=["Result"])
preddf.index.name = "FIPS"
preddf.to_csv("creativepreds.csv")
```

Part 5: Resources and Literature Used

https://medium.com/@prudhvirajnitjsr/simple-classifier-using-pytorch-37fba175c25c (https://medium.com/@prudhvirajnitjsr/simple-classifier-using-pytorch-37fba175c25c)

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

https://pytorch.org/docs/stable/index.html (https://pytorch.org/docs/stable/index.html)

https://scikit-learn.org/stable/modules/classes.html (https://scikit-learn.org/stable/modules/classes.html)

Type *Markdown* and LaTeX: α^2