Final Project - Comprehensive Classifier Creation

Sam Neyhart
Jon Clark Freeman
Kevin Sayarath
ECE 471 - Professor Qi

April 23, 2017

Abstract

The objective of this final project was to integrate various classification techniques to achieve the best performance on the chosen dataset. This project was a group effort where we explored both supervised and unsupervised classificaiton techniques including MPP (all 3 cases), kNN (with different k's), BPNN (developed), decision tree, and k-means, winner-take-all, and kohonen maps. We also explored two classifier fusion techniques including majority vote fusion and naive Bayes.

Introduction

This project performed for ECE 471 is an exercise using all the classificaiton methods we have used in past projects with the additional exploration of two new fusion methods which we previously had no experience with. The dataset we are using is titled "Poker Hand Dataset," publised by Robert Cattrel et. al., released in January 2007 *link to dataset*. By experimenting with various classification techniques, we learn the best specific methodology to classify this particular dataset.

Overall the project served its purpose and was an opportunity to review and become experts with the classification methods learned during this course. We are proud to announce we did not use any supporting libraries that do heavy-lifting of core computations, i.e. all classification methods were re-written by team members for this project.

We used a github repository to collaborate on this project:
https://github.com/sneyhart/swagteam6

Team Member Contributions

Clark

Mpp
Knn
BPNN
Decision Tree
Report Writing

Sam

Winner-take-all
Kmeans
Kohonen Map
Report writing

Kevin

Normalization
Evaluation
Team organization
Report writing

Dataset

The dataset we chose consists of all possible permutations of five-card poker hands dealt from a standard deck of 52 cards. Each sample is one hand and each card is represented by two features (suit and rank), for a total of 10 predictive attributes plus one class attribute. As previously mentioned, the order of cards is unimportant to its classification, which is why there are 480 possible Royal Flush hands as compared to only 4 in the combination set.

From poker-hand.names

Attribute Information:
1) S1 Suit of card 1
   Ordinal (1-4) representing Hearts, Spades, Diamonds, Clubs
2) C1 Rank of card 1
   Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
3) S2 Suit of card 2
   Ordinal (1-4) representing Hearts, Spades, Diamonds, Clubs
4) C2 Rank of card 2
   Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
5) S3 Suit of card 3
   Ordinal (1-4) representing Hearts, Spades, Diamonds, Clubs
6) C3 Rank of card 3
   Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
7) S4 Suit of card 4
   Ordinal (1-4) representing Hearts, Spades, Diamonds, Clubs
8) C4 Rank of card 4
   Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
9) S5 Suit of card 5
   Ordinal (1-4) representing Hearts, Spades, Diamonds, Clubs
10) C5 Rank of card 5
   Numerical (1-13) representing (Ace, 2, 3, ... , Queen, King)
11) CLASS Poker Hand
   Ordinal (0-9)

Class Information:
0: Nothing in hand; not a recognized poker hand
1: One pair; one pair of equal ranks within five cards
2: Two pairs; two pairs of equal ranks within five cards
3: Three of a kind; three equal ranks within five cards
4: Straight; five cards, sequentially ranked with no gaps
5: Flush; five cards with the same suit
6: Full house; pair + different rank three of a kind
7: Four of a kind; four equal ranks within five cards
8: Straight flush; straight + flush
9: Royal flush; Ace, King, Queen, Jack, Ten + flush

N: 25,010 training, 1,000,000 testing

Technical Approach

MPP The mpp.py Python script implements the MPP algorithm using basic Python, the numpy library, and the matplotlib library. It performs MPP parametric-based classification by first calculating the mean and covariance matricies from the dataset. *Bayes/Discriminant Func*

Case 1 The features are statistically independent, and have the same variance. Geometrically, the samples fall in equal-size hyperspherical clusters. Decision boundary: hyperplane of d-1 dimension. This classification technique employs the linear discriminant function and linear machine. Additionally, when prior probabilities are the same, the discriminant function is actually measuring the minimum distance from each feature to each of the c mean vectors.

Case 2 The covariance matrices for all the classes are identical but not a scalar of identity matrix. Geometrically, the samples fall in hyperellipsoidal clusters. Decision boundary: hyperplane of d-1 dimension.

Case 3 No assumption: the covariance matrices are different for each class. Quadratic classifier. Decision boundary: hyperquadratic for 2-D Gaussian.

kNN The knn.py Python script implements the kNN algorithm using basic Python, the numpy library, and the scipy library. It performs kNN classification, or majority voting, using Euclidean distance to assign a random sample according to the majority representation of classes within the enclosing hypersphere of k nearest neighbors. We experiment with different k's and evaluate performance.

BPNN The backprop.py script implements the backpropogation algorithm using basic Python, the numpy library, and the random library. Backpropogation is a type of multilayer feedforward network that calculates the difference between unit output and expected output, "back-propogating" this delta from the output layers back toward the feeding layers. This technique causes the network to "learn" correct classifictions in a supervised architecture. The script uses a network with 2 hidden layes. Each hidden layer has 15 neurons. The input layers change based on if the PCA preprocessing has been done on the input data. The output for the network is one neuron for the classification.

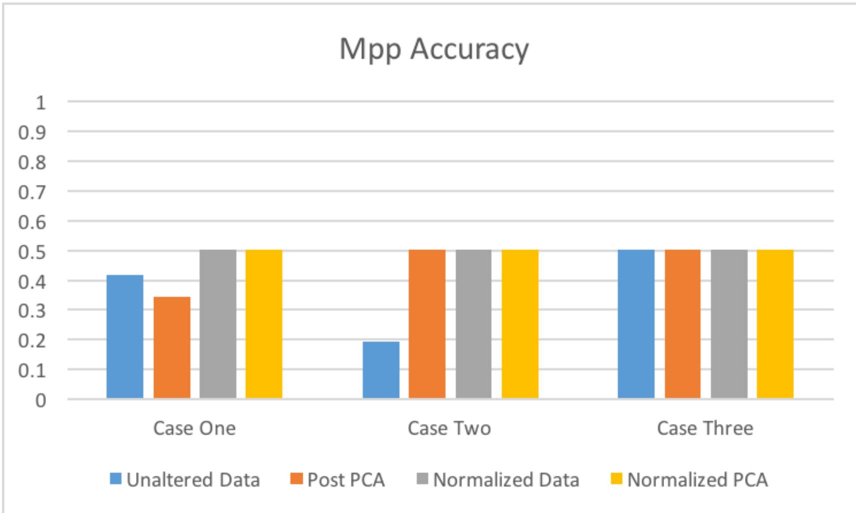
Decision Tree The dtree.py script implements the decision tree architecture using basic Python and the numpy library. The decision tree is structured to decide the various classes in an increasing faction. The highest valued class that is decided is the resulting classification of that hand. This is very efficient because the features of the data set are such that each class is made up of a specific set of cards. Thus each node will test the hand and classify it into that hand.

Kmeans The *kmeans* program was written for this project in C because we believed that python would not be able to perform the kmeans algorithm within an acceptable time span. The files *hand.c* and *hand.h* contain the definition of a hand struct as well some useful functions relating to this struct. Our kmeans program makes use of these two file in order to perform kmeans clustering on our data set. Despite having been written in C, the algorithm can take several minutes large data sets. Because of the the nature of poker hands, clustering on the raw data does not perform particularily well.

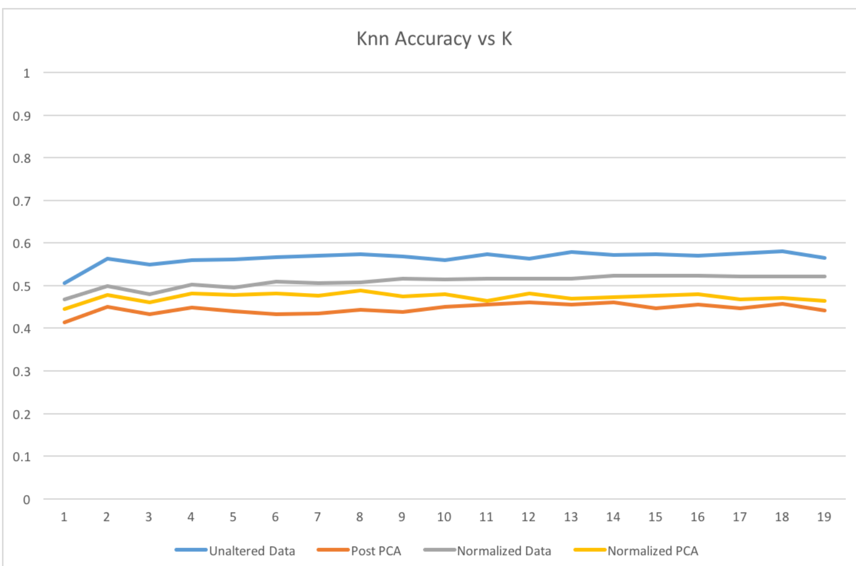
Winner-Take-All In order to perform winner-take-all clustering on our data set the program *wta* was written in C. The general structure of this program is the same as that of the kmeans program. The file input and output sections of code are reused and *wta* also makes use of the hand.c and hand.h files. The main way in which our winner-take-all program differs from our kmeans program is in the loop where the actual algorith is performed. This program is very fast (less than 5 seconds) even for our largest data set (1,000,000 data points).

Kohonen Maps In order to the perform the kohonen map algorithm on our data set the program *kohonen* was written in C. The general structure of this program is the same as that of the kmeans and winner-take-all programs. The file input and output sections of code are reused and *kohonen* also makes use of of the hand.c and hand.h files. Once again all major changes were made in the main loop where the algorithm is performed. The speed of this program is comporable to *wta*.

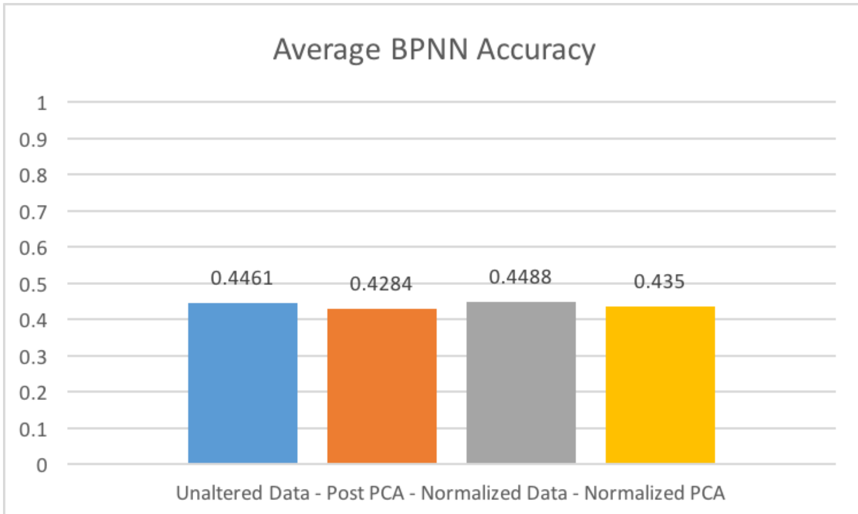
Experiments and Results



MPP

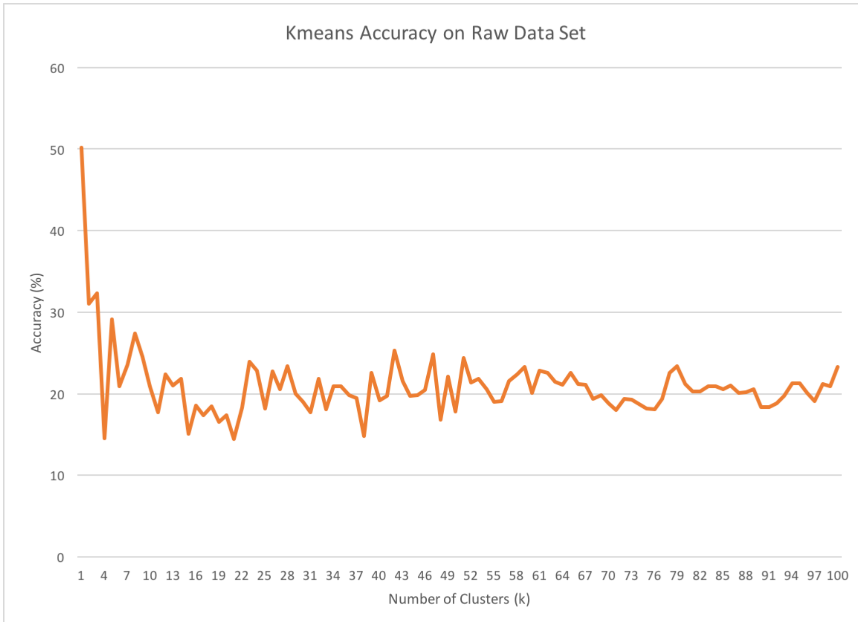


kNN

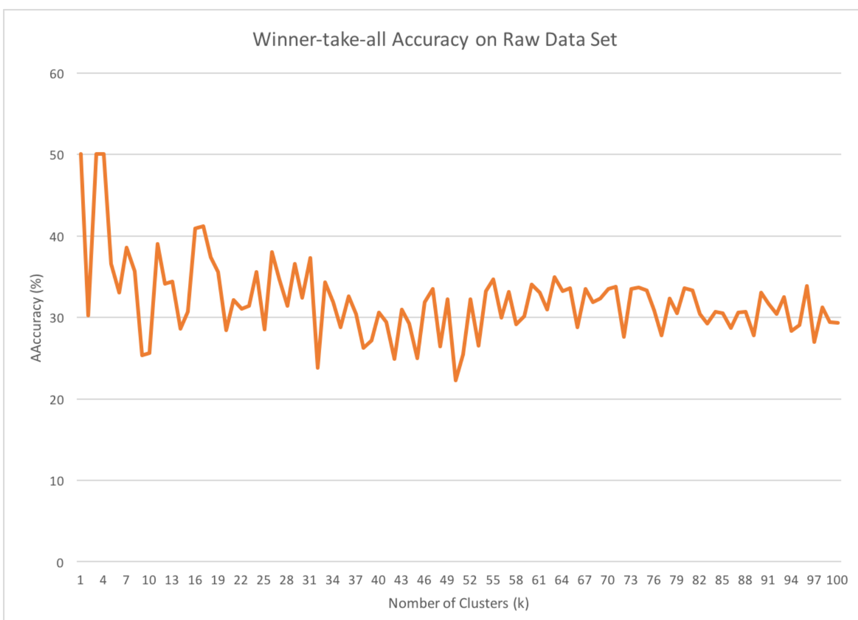


BPNN

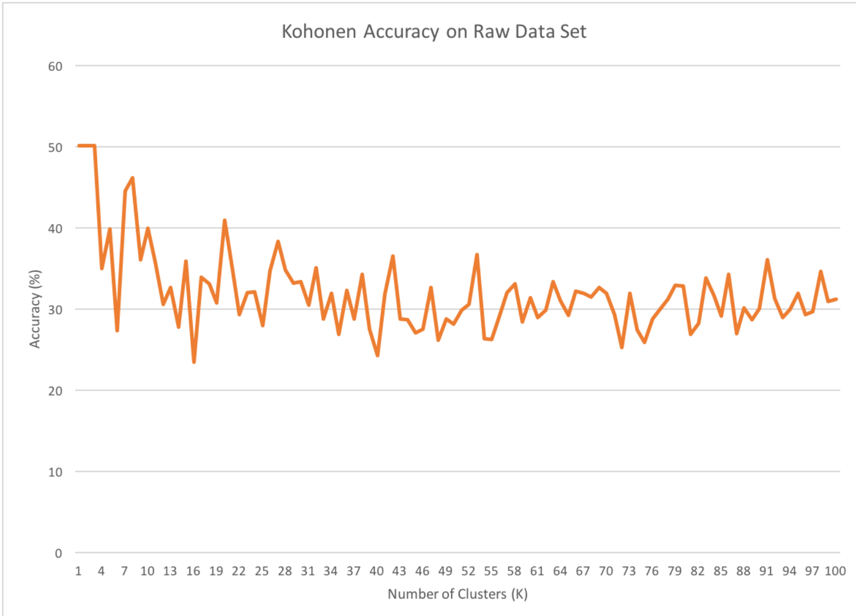
Decision Tree The Decision tree algorithm was actually able to classify with an accuracy of one hundred percent when run on the testing set. The algorithm simply ran through each of the hands and had a node that decided for each of the groups since each group is defined in such a way that is known before hand and has to be a certain combination of the cards. This lead the algorithm to be very fast as well since it only goes through the data set one time to classify each hand.



K-means



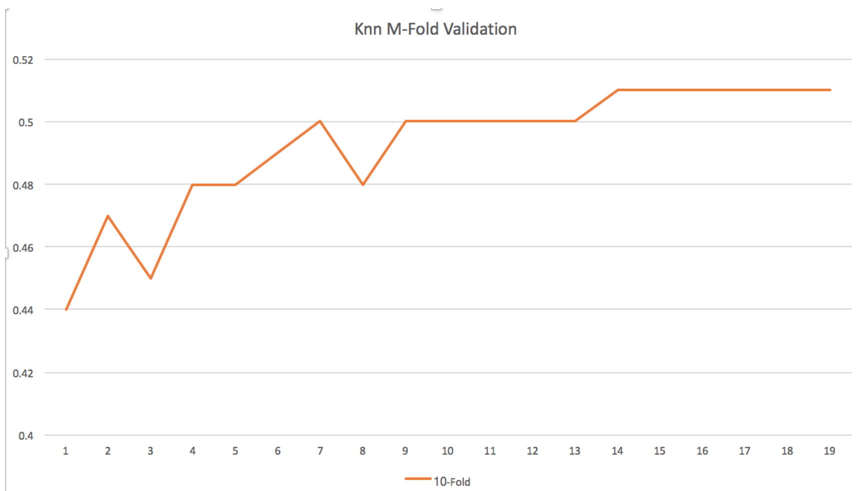
Winner-Take-All



Kohonen Maps

Evaluation

Evalution was performed using 10-fold cross-validation for the kNN classifier. Evaluation is performed in evaluation.py using standard Python language, with the support of the math, random, numpy, and subprocess libraries. This script performes the m-fold cross-validation algorithm by dividing the dataset into m disjoint sets and iterates over them using the 'leave-one-out' methodology, where one set is preserved as the testing (or validaiton) set and the remaining m-1 sets are combined to become the new training set. Normalization is performed anew on each of the m iterations for the training and testing sets separately.



kNN with 10-fold validation has a similar profile to the previous kNN trials performed with generally lower accuracy due to near complete mitigation of training/testing cycle bias.

Discussion

This project was very interesting to perform. Employing all of the classifiers that the course has covered proved to be a difficult yet enjoyable exercise. Using python was a good choice for most of the programs because of the simplicity of programming in Python as well as the ever useful libraries provided in the language. The clustering programs were written in c because of the size of the data set. C performed much faster than Python would have thus the group was able to run more tesst to glean a reasonable understanding of how well the programs performed. This was a very rewarding project to work on for the team.

Appendix

normalize.py

```
# Normalizaiton
# Input/Output file names defined in main funciton
# Running simply normalizes data and writes to output file
# Handy data separation techniques used

import math
import numpy as np
from numpy import linalg as LA

DEBUG = 0

def getInputSeparated(fname):
        #fname = 'poker-hand-training-true.data'
        file = open(fname, 'r')                              # read file
        #with open(fname) as f:
                #lines = f.read().splitlines
        lines = file.readlines()
        file.close()
        classData = []
        class0Data = []                                      # can probably do this m
        class1Data = []
        class2Data = []
        class3Data = []
        class4Data = []
        class5Data = []
        class6Data = []
        class7Data = []
        class8Data = []
        class9Data = []
        for line in lines:                                   # read elements
                line = line.strip()
                splitLine = line.split(',')
                classIndex = len(splitLine)-1    # max index of splitLine
                #print splitLine
                #print 'Class Index: {}'.format(classIndex)
                #print splitLine[classIndex]
                sampleData = []
                #sampleData.append(int(splitLine[:lenIndexSplitLine]))
                for j in range(classIndex):
                        #print j
                        sampleData.append(float(splitLine[j]))
                # Separate by classes
```

```
        if (splitLine[classIndex] == '0'):
            #print 'appending 0'
            class0Data.append(sampleData)
        elif(splitLine[classIndex] == '1'):
            class1Data.append(sampleData)
        elif(splitLine[classIndex] == '2'):
            class2Data.append(sampleData)
        elif(splitLine[classIndex] == '3'):
            class3Data.append(sampleData)
        elif(splitLine[classIndex] == '4'):
            class4Data.append(sampleData)
        elif(splitLine[classIndex] == '5'):
            class5Data.append(sampleData)
        elif(splitLine[classIndex] == '6'):
            class6Data.append(sampleData)
        elif(splitLine[classIndex] == '7'):
            class7Data.append(sampleData)
        elif(splitLine[classIndex] == '8'):
            class8Data.append(sampleData)
        elif(splitLine[classIndex] == '9'):
            class9Data.append(sampleData)
    classData.append(np.asarray(class0Data))
    classData.append(np.asarray(class1Data))
    classData.append(np.asarray(class2Data))
    classData.append(np.asarray(class3Data))
    classData.append(np.asarray(class4Data))
    classData.append(np.asarray(class5Data))
    classData.append(np.asarray(class6Data))
    classData.append(np.asarray(class7Data))
    classData.append(np.asarray(class8Data))
    classData.append(np.asarray(class9Data))
    return np.asarray(classData)

def test_getInputSeparated():
    print 'Entered test_getInputSeparated()'
    classData = getInputSeparated('poker-hand-training-true.txt')
    i = -1
    for classSamples in classData:
        i += 1
        print 'Class {} data:'.format(i)
        for classSample in classSamples:
            print classSample

def test_getInputSeparated2():
    print 'Entered test_getInputSeparated2()'
    classData = getInputSeparated('poker-hand-training-true.txt')
```

```
        print classData

def weave(classData):
        weaved = []
        largestClassSampCount = 0
        # find the length of the longest list (the number of samples in the larg
        for classSamples in classData:
                numClassSamples = len(classSamples)
                if numClassSamples > largestClassSampCount:
                        largestClassSampCount = numClassSamples
        # weave samples across all classes
        # Kevin - I can explain this, just ask me
        for i in range(largestClassSampCount):
                for classSamples in classData:
                        if (i < len(classSamples)):
                                weaved.append(classSamples[i])
        return weaved

def test_weave():
        fName_tr = 'poker-hand-training-true.data'
        # Get training data
        X0 = getInputSeparated(fName_tr)
        X = weave(X0)
        for samp in X:
                print samp

def getColumnMeans(X):
        return np.mean(X, axis=0)

def getColumnStddevs(X):
        return np.std(X, axis=0)

def getColMeansSeparated(_XS):
        colMeansSep = []
        for classSamples in _XS:
                colMeansSep.append(getColumnMeans(classSamples))
        return np.asarray(colMeansSep)

def getColumnStddevsSeparated(_XS):
        colMeansSep = []
        for classSamples in _XS:
                colMeansSep.append(getColumnStddevs(classSamples))
        return np.asarray(colMeansSep)

def normalize(X, colMeans, colStddevs):
        nX = []
```

```
        for sample in X:
            nRow = sample
            for i in range(len(nRow)):
                nRow[i] = (sample[i] - colMeans[i])/colStddevs[i]
            nX.append(nRow)
        return np.asarray(nX)

def normalizeSep(XS, colMeans_all_tr, colStddevs_all_tr):
        nXS = []
        for classSamples in XS:
            nXClass = normalize(classSamples, colMeans_all_tr, colStddevs_al
            nXS.append(nXClass)
        return np.asarray(nXS)

def rebuild(nXS, fname):
        nOriginalData = open(fname, 'w')
        i = -1
        for nClassSamples in nXS:
            i += 1
            for nSamp in nClassSamples:
                nSampString = ''
                for feature in nSamp:
                    nSampString += str(feature) + ', '
                nSampString += str(i) + '\n'
                nOriginalData.write(nSampString)
                #print nSampString
        nOriginalData.close()

def main():
        if DEBUG: print 'Entered main()'
        #fName_IN = 'poker-hand-training-true.data'
        fName_IN = 'poker-hand-testing.data'
        #fName_OUT = 'poker-hand-training-true-normalized.data'
        fName_OUT = 'poker-hand-testing-normalized.data'

        # 'X' denotes original dataset
        # 'XS' denotes original dataset separated by class (hand type)
        XS = getInputSeparated(fName_IN)
        #if DEBUG: print 'XS: \n{}'.format(XS)
        X = weave(XS)

        if DEBUG: print 'X:\n{}\n\n\n'.format(X)
        if DEBUG: print 'XS:\n{}\n\n\n'.format(XS)

        # Compute un-normalized training mu & sigma for normalization
        colMeans_all_tr = getColumnMeans(X)
```

```
        colStddevs_all_tr = getColumnStddevs(X)

        # 'nX' denotes original dataset - normalized
        # 'nXS' denotes original dataset, separated - normalized
        nXS = normalizeSep(XS, colMeans_all_tr, colStddevs_all_tr)
        nX = weave(nXS)                                    # nX is both classes com

        if DEBUG: print 'nX:\n{}\n\n\n'.format(nX)
        #if DEBUG:
        print 'nXS:\n{}'.format(nXS)

        rebuild(nXS, fName_OUT)


main()
```

mpp.py

```
##################
# Jon Clark Freeman, Sam Neyhart, Kevin Sayarath
# mpp.py classifies using the three cases of mpp
#################

import matplotlib.pyplot as plt
import numpy as np

#function that returns the likelihood for a given [x, y], mean, and covariance m
def cmv(x,mean,cov,prior):
    pre_top = 1;
    pre_bottom = (2 * np.pi) * np.sqrt(np.linalg.det(cov))
    post_matrix_mult = np.dot(np.dot(np.transpose(np.subtract(np.array(x), mean)
    post_scalar = -1.0/2.0
    return (pre_top/pre_bottom) * np.exp(post_scalar * post_matrix_mult) * prior

#function that returns Case 1 Discriminant
def case_1(x, mean,covariance,prior):
    return -np.dot(np.subtract(np.array(x), mean),np.transpose(np.subtract(np.ar

#function that returns Case 2 Discriminant
def case_2(x, mean, cov, prior):
    return -np.dot(np.dot(np.transpose(np.subtract(np.array(x), mean)), np.linal

#initialization of priors and glasses
priors = [0.501177,0.422569,0.047539,0.021128,0.003925,0.001965,0.001441,0.00024
classes = [[] for i in range(0,10)]

# Handle file manipulation
lines = [np.array((file_line.rstrip('\n').split(','))).astype(float) for file_li
lines = np.array(lines)

#handle getting covariance matrix and means
cov_lines = np.delete(lines.T, lines.T.shape[0] - 1,0)
for line in lines:
    classes[int(line[-1])].append(np.delete(line,(line.shape[0]-1),0))
classes = np.array([np.array(classer) for classer in classes])
means = [np.mean(row) for row in classes]
covariance = np.cov(cov_lines)

testing_lines = [np.array((file_line.rstrip('\n').split(','))).astype(float) for
testing_lines = np.array(testing_lines)
tests = np.delete(testing_lines.T, testing_lines.T.shape[0] - 1,0).T
actual_classes = testing_lines.T[-1]
```

```python

choices = []

for t,test in enumerate(tests):
    max_disc = -1
    max_val = -1000000
    for i in range(0,len(means)):
        res = cmv(test, means[i],covariance,priors[i])
        if res > max_val:
            max_disc = i
            max_val = res
    choices.append(max_disc)

corrects = 0
total = 0
for i in range(0, len(choices)):
    if choices[i] == int(actual_classes[i]):
        corrects = corrects + 1
    total = total + 1
fusout = open("fusion-files-majority/mpp.data",'w')
for choice in choices:
    fusout.write("{}\n".format(int(choice)))
```

knn.py

```
import numpy as np
from scipy import spatial as sp

#Preprocessing
lines = []
tlines = []
data = []
groups = []
classes = []
dtp = []
knn_res = []
corrects = [0] * 20
#file IO

lines = [np.array((file_line.rstrip('\n').split(','))).astype(float) for file_li
lines = np.array(lines)
training = np.delete(lines.T, lines.T.shape[0] - 1,0).T
res = lines.T[-1]

testing_lines = [np.array((file_line.rstrip('\n').split(','))).astype(float) for
testing_lines = np.array(testing_lines)
tests = np.delete(testing_lines.T, testing_lines.T.shape[0] - 1,0).T
actual_classes = testing_lines.T[-1]


#testing each unit in the testing set
fusout = open("fusion-files-majority/knn_un.data",'w')
for u,unit in enumerate(tests):
    if u % 150 == 0:
        print "{}%".format(u/150*10)
    dis = []
    cords = unit
    actual = actual_classes[u]
    for point in training:
        dis.append(np.linalg.norm(point - cords))
    idx = np.argsort(dis, axis=0)
    dis = np.array(dis)[idx]
    results = np.array(res)[idx]
    counts = [0,0,0,0,0,0,0,0,0,0]

#going through each k value after each distance is calculated
    for k in range(1,20):
        mmax = -1
        mindex = -1
```

```
        counts[int(results[k-1])] = counts[int(results[k-1])] + 1

        for ind,count in enumerate(counts):
            if mmax < count:
                mmax = count
                mindex = ind
        if k == 17:
            fusout.write("{}\n".format(mindex))
        if(actual == mindex):
            corrects[k] += 1
# counting the correct guesses per each k value
corrects = [(correct * 1.0)/len(tests) for correct in corrects]
knn_res.append(corrects)
knn_res_tp = np.transpose(knn_res)
knn_res_tp = [np.mean(col) for col in knn_res_tp]
knn_res = np.transpose(knn_res_tp)
#print out the results for each k value up to the max k value
for i, k in enumerate(knn_res):
    print "{},{}".format(i, k)
```

backprop.py

```
### backprop.py
### Jon Clark Freeman
### learns and tests XOR logic
import numpy as np
import numpy.random as rando
import random

class BPNN:
    #initialize the network
    def __init__(self, network):
        self.weights = []

        #weights for hidden and input
        for i in range(1, len(network) - 1):
            self.weights.append(2*rando.random((network[i-1] + 1, network[i] + 1

        self.weights.append(2*rando.random((network[i] + 1, network[i+1])) - 1)

    def activation(self, x):
        return np.tanh(x)

    def activation_deriv(self, x):
        return 1.0 - x**2

    def train(self, inp, target,learning_rate=0.01, trials=10000):
        #Add bias units as a col of ones on the inputs
        ones = np.atleast_2d(np.ones(inp.shape[0]))
        x = np.concatenate((ones.T, inp), axis=1)
        for j in range(trials):
            #choose a random input to train with
            i = np.random.randint(inp.shape[0])
            activations = [x[i]]

            #calculating the activations for each of the layers
            for k in range(len(self.weights)):
                dot_value = np.dot(activations[k], self.weights[k])
                act = self.activation(dot_value)
                activations.append(act)

            # calculating error from outputs and targets
            error = target[i] - activations[-1]
            deltas = [error * self.activation_deriv(activations[-1])]

            # Calculating deltas starting at the hidden layer
```

```
        for k in range(len(activations) - 2, 0, -1):
            deltas.append(deltas[-1].dot(self.weights[k].T)*self.activation_
        deltas.reverse()

        # updating the weights using the deltas, activations
        for i in range(len(self.weights)):
            layer = np.atleast_2d(activations[i])
            delta = np.atleast_2d(deltas[i])
            self.weights[i] += learning_rate * layer.T.dot(delta)

    #test on all of the inputs
    def test(self, inp):
        retval = np.concatenate((np.ones(1).T, np.array(inp)), axis=1)
        for i in range(0, len(self.weights)):
            retval = self.activation(np.dot(retval, self.weights[i]))
        return retval

#code to test and train using the class above
nn = BPNN([10,15,15,1])

lines = [np.array((file_line.rstrip('\n').split(','))).astype(float) for file_li
lines = np.array(lines)
training = np.delete(lines.T, lines.T.shape[0] - 1,0).T
res = lines.T[-1]

testing_lines = [np.array((file_line.rstrip('\n').split(','))).astype(float) for
testing_lines = np.array(testing_lines)
tests = np.delete(testing_lines.T, testing_lines.T.shape[0] - 1,0).T
actual_classes = testing_lines.T[-1]

nn.train(training, res)
corrects = 0.0
total = 0.0
fusout = open("fusion-files-majority/bpnn.data",'w')
for t,inpu in enumerate(tests):
    total = total + 1
    fusout.write("{}\n".format(int(round(nn.test(inpu)[0]))))
    if int(round(nn.test(inpu)[0])) == actual_classes[t]:
        corrects = corrects + 1
print "Accuracy - {}".format(corrects/total)
```

dtree.py

```
'''
dtree.py
decision tree classifier
has 9 nodes to check for each possible hand
'''
#file IO
import numpy as np
testing_lines = [np.array((file_line.rstrip('\n').split(','))).astype(int) for file_line in
testing_lines = np.array(testing_lines)
tests = np.delete(testing_lines.T, testing_lines.T.shape[0] - 1,0).T
actual_classes = testing_lines.T[-1]

#initialize
corrects = 0.0
total = 0.0

#each testing
for t,test in enumerate(tests):
    #get info sorted into suits and ranks
    total = total + 1
    ranks = []
    suits = []
    for i in range(0,10,2):
        suits.append(test[i])
        ranks.append(test[i+1])
    matches = [0]
    idx = np.argsort(ranks, axis=0)
    ranks = np.array(ranks)[idx]
    suits = np.array(suits)[idx]
    ranks = ranks.tolist()
    suits = suits.tolist()

    #is pair?
    ph = 0
    for val in ranks:
        if ranks.count(val) == 2:
            ph = 1
    if ph == 1:
        matches.append(1)

    #is two pair?
    ph = 0
    first = 0
    for val in ranks:
```

```
    if ranks.count(val) == 2:
        if first == 0:
            first = val
        elif first != val:
            ph = 1
if ph == 1:
    matches.append(2)

#is three of a kind?
ph = 0
for val in ranks:
    if ranks.count(val) == 3:
        ph = 1
if ph == 1:
    matches.append(3)

#is straight?
ph = 1
if ranks.count(1) == 1 and ranks[-1] == 13:
    for i,val in enumerate(ranks):
        if i != 0 and i != 1:
            if ranks[i] - ranks[i - 1] != 1:
                ph = 0
else:
    for i,val in enumerate(ranks):
        if i != 0:
            if ranks[i] - ranks[(i - 1)] != 1:
                ph = 0
if ph == 1:
    matches.append(4)


#is flush?
if suits.count(suits[0]) == 5:
    matches.append(5)

#is full house?
if 1 in matches and 3 in matches:
    matches.append(6)

#is four of a kind?
ph = 0
for val in ranks:
    if ranks.count(val) == 4:
        ph = 1
if ph == 1:
```

```
        matches.append(7)

    #is straight flush?
    if 4 in matches and 5 in matches:
        matches.append(8)

    #is royal flush?
    if 8 in matches and ranks[0] == 1 and ranks[-1] == 13:
        matches.append(9)

    #checking result
    result = matches[-1]
    if result == actual_classes[t]:
        corrects = corrects + 1
print "Accuracy {}".format(corrects/total)
```

kmeans.c

```
/*
 * kmeans.c
 * Sam Neyhart
 * This file contains the kmeans program used in the final project of
 * ECE 471.
 * USAGE: kmeans filename.data nclusters
 */

#include<string.h>
#include<time.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include"hand.h"


static const int ITERATIONS = 1; //How many times to go through data
static const int NCLASSES = 10;
static const int NCARDS = 10;
static int NCLUSTERS;
static int NLINES = 0;

int main(int argc, char ** argv)
{
        //----------------File IO----------------
        FILE * in;
        in = fopen(argv[1],"r");
        if (argc != 3) return 0;
        NCLUSTERS = atoi(argv[2]);
        //Get numLines
        char buf[255];
        while(fgets(buf,255,in) != NULL){
                NLINES++;
        }
        rewind(in);
        //create arrays
        hand * data = malloc(NLINES*sizeof(hand));
        hand * mu = malloc(NCLUSTERS*sizeof(hand));
        int * clas = malloc(NLINES*sizeof(int));
        //fill data
        for(int i = 0; i < NLINES; i++){
                char * tmp;
                fgets(buf,255,in);
                tmp = strtok(buf,",");
```

```c
        for(int j = 0; j < NCARDS; j++){
            data[i].cards[j] = atof(tmp);
            tmp = strtok(NULL,",");
        }
        clas[i]=atoi(tmp);
    }
    //randomly fill mu
    time_t t;
    srand((unsigned)time(&t));
    for(int i = 0; i < NCLUSTERS; i++){
        int tmp = rand() % NLINES;
        for(int j = 0; j < NCARDS; j++){
            mu[i].cards[j] = data[tmp].cards[j];
        }
    }

    //------------Perform Kmeans-------------
    int *clus1 = malloc(NLINES*sizeof(int));
    int *clus2 = malloc(NLINES*sizeof(int));
    int flip = 0;
    int done = 0;
    int it = 0;
    while (done==0){
        //printf("iteration %d\n",it);
        if (flip==0){
            cluster(data,mu,NLINES,NCLUSTERS,clus1);
            avg(mu, data, clus1,NLINES,NCLUSTERS);
            flip++;
        }else{
            cluster(data,mu,NLINES,NCLUSTERS,clus2);
            avg(mu, data, clus2,NLINES,NCLUSTERS);
            flip--;
        }
        if (clus_equal(clus1,clus2,NLINES))
            done++;
        it++;
    }

    //-------Deterimine Accuracy-------
    hand *avgclas = calloc(NCLASSES,sizeof(hand));
    hand *avgclus = calloc(NCLUSTERS,sizeof(hand));
    avg(avgclas, data, clas, NLINES, NCLASSES);
    avg(avgclus, data, clus1, NLINES, NCLUSTERS);
    printf(
            "-------------------------------For Number of clusters %d"
            "-------------------------------\n",NCLUSTERS);
```

```c
		printf("Number of Iterations for Kmeans: %d\n", it);
		printf("Avg Clusters:\n");
		for(int i = 0; i < NCLUSTERS; i++){
			for(int j = 0; j < NCARDS; j++){
				printf("%lf ", avgclus[i].cards[j]);
			}
			printf("\n");
		}
		printf("\nAvg Classes:\n");
		for(int i = 0; i < NCLASSES; i++){
			for(int j = 0; j < NCARDS; j++){
				printf("%lf ", avgclas[i].cards[j]);
			}
			printf("\n");
		}
		double acc;
		acc = accuracy(NLINES,NCLUSTERS,NCLASSES, clus1, clas, avgclus, avgclas);
		printf("\nOverall Accuracy is %lf\n\n", acc);

		//----------Cleanup----------
		free(data);
		free(mu);
		free(clas);
		free(avgclas);
		free(avgclus);
		free(clus1);
		free(clus2);
		fclose(in);
}
```

wta.c

```c
/*
 * wta.c
 * Sam Neyhart
 * This file contains the wta program used in the final project of
 * ECE 471.
 * USAGE: wta filename.data nclusters
 */

#include<string.h>
#include<time.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include"hand.h"


static const int ITERATIONS = 20; //How many times to go through data
static const int NCLASSES = 10;
static const int NCARDS = 10;
static int NCLUSTERS;
static int NLINES = 0;

int main(int argc, char ** argv)
{
        //----------File IO----------
        FILE * in;
        in = fopen(argv[1],"r");
        if (argc != 3) return 0;
        NCLUSTERS = atoi(argv[2]);
        //Get numLines
        char buf[255];
        while(fgets(buf,255,in) != NULL){
                NLINES++;
        }
        rewind(in);
        //create arrays
        hand * data = malloc(NLINES*sizeof(hand));
        hand * mu = malloc(NCLUSTERS*sizeof(hand));
        int * clas = malloc(NLINES*sizeof(int));
        //fill data
        for(int i = 0; i < NLINES; i++){
                char * tmp;
                fgets(buf,255,in);
                tmp = strtok(buf,",");
```

```
			for(int j = 0; j < NCARDS; j++){
				data[i].cards[j] = atof(tmp);
				tmp = strtok(NULL,",");
			}
			clas[i]=atoi(tmp);
		}
		//randomly fill mu
		time_t t;
		srand((unsigned)time(&t));
		for(int i = 0; i < NCLUSTERS; i++){
			int tmp = rand() % NLINES;
			for(int j = 0; j < NCARDS; j++){
				mu[i].cards[j] = data[tmp].cards[j];
			}
		}

		//----------------Perform WTA---------------
		for(int i = 0; i < ITERATIONS; i++){
			for(int j=0; j<NLINES;j++){
				update_mu(&data[j],mu,NCLUSTERS);
			}
		}

		//--------Deterimine Accuracy--------
		int *clus = malloc(NLINES*sizeof(int));
		cluster(data,mu,NLINES,NCLUSTERS,clus);
		hand *avgclas = calloc(NCLASSES,sizeof(hand));
		hand *avgclus = calloc(NCLUSTERS,sizeof(hand));
		avg(avgclas, data, clas, NLINES, NCLASSES);
		avg(avgclus, data, clus, NLINES, NCLUSTERS);
		printf(
				"------------------------------------------------For Number of clusters %d"
				"------------------------------------------------\n",NCLUSTERS);
		printf("Avg Clusters:\n");
		for(int i = 0; i < NCLUSTERS; i++){
			for(int j = 0; j < NCARDS; j++){
				printf("%lf ",avgclus[i].cards[j]);
			}
			printf("\n");
		}
		printf("\nAvg Classes:\n");
		for(int i = 0; i < NCLASSES; i++){
			for(int j = 0; j < NCARDS; j++){
				printf("%lf ",avgclas[i].cards[j]);
			}
			printf("\n");
```

```
        }
        double acc;
        acc = accuracy(NLINES,NCLUSTERS,NCLASSES,clus,clas,avgclus,avgclas);
        printf("\nOverall Accuracy is %lf\n\n",acc);

        //------------Cleanup------------
        free(data);
        free(mu);
        free(clas);
        free(clus);
        fclose(in);
}
```

kohonen.c

```
/*
 * kohonen.c
 * Sam Neyhart
 * This file contains the kohonen program used in the final project of
 * ECE 471.
 * USAGE: kohonen filename.data nclusters
 */

#include<string.h>
#include<time.h>
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include"hand.h"


static const int ITERATIONS = 6000000; //How many times to go through data
static const int NCLASSES = 10;
static const int NCARDS = 10;
static int NCLUSTERS;
static int NLINES = 0;

int main(int argc, char ** argv)
{
        //------------File IO-----------
        FILE * in;
        in = fopen(argv[1],"r");
        if (argc != 3) return 0;
        NCLUSTERS = atoi(argv[2]);
        //Get numLines
        char buf[255];
        while(fgets(buf,255,in) != NULL){
                NLINES++;
        }
        rewind(in);
        //create arrays
        hand * data = malloc(NLINES*sizeof(hand));
        hand * mu = malloc(NCLUSTERS*sizeof(hand));
        int * clas = malloc(NLINES*sizeof(int));
        //fill data
        for(int i = 0; i < NLINES; i++){
                char * tmp;
                fgets(buf,255,in);
                tmp = strtok(buf,",");
```

```
                for(int j = 0; j < NCARDS; j++){
                        data[i].cards[j] = atof(tmp);
                        tmp = strtok(NULL,",");
                }
                clas[i]=atoi(tmp);
        }
        //randomly fill mu
        time_t t;
        srand((unsigned)time(&t));
        for(int i = 0; i < NCLUSTERS; i++){
                int tmp = rand() % NLINES;
                for(int j = 0; j < NCARDS; j++){
                        mu[i].cards[j] = data[tmp].cards[j];
                }
        }

        //----------Perform Kohonen----------
        for(int i = 0; i < ITERATIONS; i++){
                kohonen_mu(&data[i%NLINES],mu,NCLUSTERS,i,ITERATIONS);
        }

        //------Deterimine Accuracy------
        int *clus = malloc(NLINES*sizeof(int));
        cluster(data,mu,NLINES,NCLUSTERS,clus);
        hand *avgclas = calloc(NCLASSES,sizeof(hand));
        hand *avgclus = calloc(NCLUSTERS,sizeof(hand));
        avg(avgclas, data, clas, NLINES, NCLASSES);
        avg(avgclus, data, clus, NLINES, NCLUSTERS);
        printf(
                        "------------------------------For Number of clusters %d"
                        "------------------------------\n",NCLUSTERS);
        printf("Avg Clusters:\n");
        for(int i = 0; i < NCLUSTERS; i++){
                for(int j = 0; j < NCARDS; j++){
                        printf("%lf ",avgclus[i].cards[j]);
                }
                printf("\n");
        }
        printf("\nAvg Classes:\n");
        for(int i = 0; i < NCLASSES; i++){
                for(int j = 0; j < NCARDS; j++){
                        printf("%lf ",avgclas[i].cards[j]);
                }
                printf("\n");
        }
        double acc;
```

```
        acc = accuracy(NLINES,NCLUSTERS,NCLASSES,clus,clas,avgclus,avgclas);
        printf("\nOverall Accuracy is %lf\n\n",acc);

        //------------Cleanup------------
        free(data);
        free(mu);
        free(clas);
        free(clus);
        fclose(in);
}
```

hand.c

```c
/*
 * hand.c
 * Sam Neyhart
 * This contains the hand struct and some related functions
 * used in the final project of ECE 471. The usage of these
 * functions is documented in hand.h
 */

#include"hand.h"
#include<math.h>
#include<stdio.h>

static const double EPSILON = 0.01;
static const double EPSILON_MAX = 0.05;

int clus_equal(int *c1, int *c2,int NLINES)
{
        for(int i=0; i<NLINES;i++){
                if(c1[i]!=c2[i]){
                        return 0;
                }
        }
        return 1;
}

void avg(hand *avgc, hand *data, int *c, int nl, int nc)
{
        int count[nc];
        for(int i=0; i<nc; i++)
                count[i]=0;
        for(int i=0; i<nc; i++){
                for(int j=0; j<10; j++){
                        avgc[i].cards[j] = 0;
                }
        }
        for(int i=0; i<nl; i++){
                count[c[i]]++;
                for(int j=0; j<10; j++){
                        avgc[c[i]].cards[j]+= data[i].cards[j];
                }
        }
        for(int i=0; i<nc; i++){
                for(int j=0; j<10; j++){
                        avgc[i].cards[j]/=count[i];
```

```c
                        if (count[i]==0)
                                avgc[i].cards[j]=-1;
                }
        }
}

double accuracy(int NLINES, int nclu, int nc, int *clus, int *clas, hand *avgclu
{
        int truec[nclu];
        int count = 0;
        printf("\n");
        for(int i=0; i<nclu; i++){
                truec[i] = closest(&avgclus[i], avgclas, nc);
                printf("Cluster %d is class %d\n",i,truec[i]);
        }
        for(int i=0; i<NLINES; i++){
                count+=(truec[clus[i]]==clas[i]);
        }
        return count/(float)NLINES;
}

void kohonen_mu(hand *h1, hand *mu, int NCLUSTERS, int k, int kmax)
{
        int close = closest(h1, mu, NCLUSTERS);
        hand win = mu[close];
        for(int i=0; i<NCLUSTERS; i++){
                hand tmp = vdiff(h1,&mu[i]);
                double e = EPSILON_MAX * pow(EPSILON/EPSILON_MAX,k/(double)kmax)
                double d = exp(-1*dist(&win,&mu[i]));
                tmp = scale(&tmp,e*d);
                for(int j=0; j < sizeof(tmp.cards)/sizeof(tmp.cards[0]); j++){
                        mu[i].cards[j]+=tmp.cards[j];
                }
        }
}

void update_mu(hand *h1, hand *mu, int nc)
{
        int close = closest(h1, mu, nc);
        hand tmp = vdiff(h1, &mu[close]);
        tmp = scale(&tmp, EPSILON);
        for(int i = 0; i < sizeof(tmp.cards)/sizeof(tmp.cards[0]); i++)
                mu[close].cards[i] += tmp.cards[i];
}

hand vdiff(hand *h1, hand *h2)
```

```
{
	hand out = *h1;
	for(int i = 0; i < sizeof(out.cards)/sizeof(out.cards[0]); i++)
		out.cards[i] = h1->cards[i] - h2->cards[i];
	return out;
}

hand scale(hand *h1, double s)
{
	hand out = *h1;
	for(int i = 0; i < sizeof(out.cards)/sizeof(out.cards[0]); i++)
		out.cards[i]*=s;
	return out;
}

double dist(hand *h1, hand *h2)
{
	double dist = 0;
	for(int i = 0; i<sizeof(h1->cards)/sizeof(h1->cards[0]); i++){
		dist += pow(fabs(h1->cards[i]-h2->cards[i]),2);
	}
	return sqrt(dist);
}

int closest(hand *h1, hand *mu, int NCLUSTERS)
{
	double small = -1;
	int closest = -1;
	for(int i = 0; i < NCLUSTERS; i++){
		int tmp = dist(h1, &(mu[i]));
		if(small == -1 || small > tmp){
			small = tmp;
			closest = i;
		}
	}
	return closest;
}

void cluster(hand *data, hand *mu, int nl, int nc, int *clus)
{
	for(int i=0; i<nl; i++){
		clus[i] = closest(&data[i],mu,nc);
	}
}
```

hand.h

```
/*
 * hand.h
 * Sam Neyhart
 * This contains the definition of the hand struct
 * and the declarations of the relevant functions
 * used in the final project of ECE 471.
 */

#ifndef _hand
#define _hand
typedef struct{
        double cards[10];
}hand;

//returns a hand which is h1 times s
hand scale(hand *h1, double s);

//returns the euclidean distance betwen h1 and h2
double dist(hand *h1, hand *h2);

//returns a hand which is h1-h2
hand vdiff(hand *h1, hand *h2);

//updates mu in the winner-take-all algorithm based on h1
void update_mu(hand *h1, hand *mu, int nc);

//updates all of the kohonen map centers based on h1
void kohonen_mu(hand *h1,hand *mu, int NCLUSTERS,int k,int kmax);

//returns the index of the hand in mu which is closest to h1
int closest(hand *h1, hand *mu, int nc);

//fills the int array cluster with the index of the cluster (mu)
//which each data point belongs to.
void cluster(hand *data, hand *mu, int nl, int nc, int *cluster);

//modifies the hands at location avgc to be averages of the data
//in each cluster
void avg(hand *avgc, hand *data, int *c, int nl, int nc);

//determines the accuracy of the clusters
double accuracy(int NLINES, int nclu, int nc, int *clus, int *clas, hand *avgclu

//determines if two clusters are the same
```

```
int clus_equal(int *c1, int *c2,int NLINES);
#endif
```

evaluation.py

```
# Normalizaiton
# Input/Output file names defined in main funciton
# Running simply normalizes data and writes to output file
# Handy data separation techniques used

import math
import random
import numpy as np
from numpy import linalg as LA
from subprocess import call

DEBUG = 0
CLASSINDEX = 10                    # zero-indexed

def getInputSeparated(fname):
        #fname = 'poker-hand-training-true.data'
        file = open(fname, 'r')                         # read file
        #with open(fname) as f:
                #lines = f.read().splitlines
        lines = file.readlines()
        file.close()
        classData = []
        class0Data = []                                         # can probably d
        class1Data = []
        class2Data = []
        class3Data = []
        class4Data = []
        class5Data = []
        class6Data = []
        class7Data = []
        class8Data = []
        class9Data = []
        for line in lines:                                      # read elements
                line = line.strip()
                splitLine = line.split(',')
                classIndex = len(splitLine)-1   # max index of splitLine
                #print splitLine
                #print 'Class Index: {}'.format(classIndex)
                #print splitLine[classIndex]
                sampleData = []
                #sampleData.append(int(splitLine[:lenIndexSplitLine]))
                for j in range(classIndex):
                        #print j
                        sampleData.append(float(splitLine[j]))
```

```
			# Separate by classes
			if (splitLine[classIndex] == '0'):
				#print 'appending 0'
				class0Data.append(sampleData)
			elif(splitLine[classIndex] == '1'):
				class1Data.append(sampleData)
			elif(splitLine[classIndex] == '2'):
				class2Data.append(sampleData)
			elif(splitLine[classIndex] == '3'):
				class3Data.append(sampleData)
			elif(splitLine[classIndex] == '4'):
				class4Data.append(sampleData)
			elif(splitLine[classIndex] == '5'):
				class5Data.append(sampleData)
			elif(splitLine[classIndex] == '6'):
				class6Data.append(sampleData)
			elif(splitLine[classIndex] == '7'):
				class7Data.append(sampleData)
			elif(splitLine[classIndex] == '8'):
				class8Data.append(sampleData)
			elif(splitLine[classIndex] == '9'):
				class9Data.append(sampleData)
		classData.append(np.asarray(class0Data))
		classData.append(np.asarray(class1Data))
		classData.append(np.asarray(class2Data))
		classData.append(np.asarray(class3Data))
		classData.append(np.asarray(class4Data))
		classData.append(np.asarray(class5Data))
		classData.append(np.asarray(class6Data))
		classData.append(np.asarray(class7Data))
		classData.append(np.asarray(class8Data))
		classData.append(np.asarray(class9Data))
		return np.asarray(classData)

def test_getInputSeparated():
		print 'Entered test_getInputSeparated()'
		classData = getInputSeparated('poker-hand-training-true.txt')
		nTotal = 0
		i = -1
		for classSamples in classData:
			i += 1
			print 'Class {} data:'.format(i)
			for classSample in classSamples:
				print classSample

def test_getInputSeparated2():
```

```
        print 'Entered_test_getInputSeparated2()'
        classData = getInputSeparated('poker-hand-training-true.txt')
        print classData

# aggregates a 2-D array into a single numpy array
def weave(classData, classification=0):
        weaved = []
        largestClassSampCount = 0
        # find the length of the longest list (the number of samples in the larg
        for classSamples in classData:
                numClassSamples = len(classSamples)
                if numClassSamples > largestClassSampCount:
                        largestClassSampCount = numClassSamples
        # weave samples across all classes
        # Kevin - I can explain this, just ask me
        c = 0
        samp = []
        for i in range(largestClassSampCount):
                c = -1
                for j in range(len(classData)):
                        c += 1
                        classSamples = classData[j]
                        if (i < len(classSamples)):
                                samp = []
                                for feature in classSamples[i]:
                                        samp.append(feature)
                                if (classification): samp.append(c)
                                weaved.append(samp)
        return np.asarray(weaved)

def test_weave():
        fName_tr = 'poker-hand-training-true.data'
        # Get training data
        X0 = getInputSeparated(fName_tr)
        X = weave(X0, 1)
        for samp in X:
                print samp

def getColumnMeans(X):
        return np.mean(X, axis=0)

def getColumnStddevs(X):
        return np.std(X, axis=0)

def getColMeansSeparated(_XS):
        colMeansSep = []
```

```
    for classSamples in _XS:
        colMeansSep.append(getColumnMeans(classSamples))
    return np.asarray(colMeansSep)

def getColumnStddevsSeparated(_XS):
    colMeansSep = []
    for classSamples in _XS:
        colMeansSep.append(getColumnStddevs(classSamples))
    return np.asarray(colMeansSep)

# Normalize the given dataset
def normalize(X, colMeans, colStddevs):
    nX = []
    for sample in X:
        nRow = sample
        for i in range(len(nRow)):
            nRow[i] = (sample[i] - colMeans[i])/colStddevs[i]
        nX.append(nRow)
    return np.asarray(nX)

# Normalize each separate class
def normalizeSep(XS, colMeans_all_tr, colStddevs_all_tr):
    nXS = []
    for classSamples in XS:
        nXClass = normalize(classSamples, colMeans_all_tr, colStddevs_al
        nXS.append(nXClass)
    return np.asarray(nXS)

# Rebuild original data file with normalized samples (classes are not normalized
def rebuild(nXS, fname):
    nOriginalData = open(fname, 'w')
    i = -1
    for nClassSamples in nXS:
        i += 1
        for nSamp in nClassSamples:
            nSampString = ''
            for feature in nSamp:
                nSampString += str(feature) + ', '
            nSampString += str(i) + '\n'
            nOriginalData.write(nSampString)
            #print nSampString
    nOriginalData.close()

# Verify correct sample classificaiton
def verify(sample, classification):
    if (sample[CLASSINDEX] == classification):
```

```
                return 1
        else:
                return 0
        #return sample[CLASSINDEX] == classification

# Leave specified set out (stored as an empty list)
def combineSetsExcept(sets, exclude):
        combined = []
        i = -1
        for st in sets:
                i += 1
                if i == exclude:
                        combined.append([])
                        continue
                combined.append(st)
        return np.asarray(combined)

def combineSetsExcept2(sets, exclude):
        combined = []
        sampWithClass = []
        i = -1
        for st in sets:
                i += 1
                if i == exclude: continue
                for sample in st:
                        sampWithClass = sample
                        sampWithClass.append(i)
                        combined.append(sampWithClass)
        return np.asarray(combined)

# Get m randomly assigned sets
def getMSets(XS, m):
        mSets = []
        for mth in range(m):                                          # init n
                mSets.append([])
        #print 'len(XS):\n{}'.format(len(XS))
        X = weave(XS, 1)
        n = len(X)

        # Randomly divide dataset into m sets
        randIs = random.sample(range(n), n)                     # get random ind
        numRIs = len(randIs)
        mMaxSize = (n / m) + n % m                                    # max m
        ri = -1
        for _ in range(mMaxSize):                                     # iterat
                for mthSet in range(m):                               # iterat
```

```
            if ((ri + 1) >= numRIs):
                break
            else:
                ri += 1
            mSets[mthSet].append(X[ri])                                   # assign
    return mSets

def test_getMSets():
    fName_tr = 'poker-hand-training-true.data'
    # Get training data
    XS = getInputSeparated(fName_tr)
    mSets = getMSets(XS, 10)

    mSetSize = 0
    mSetSizeCombined = 0
    i = -1
    for mthSet in mSets:
        i += 1
        mSetSize = len(mthSet)
        mSetSizeCombined += mSetSize
        print '{}th set:\nSize: {}'.format(i, mSetSize)
        for samp in mthSet:
            print samp
        print
    print 'mSetSizeCombined: {}'.format(mSetSizeCombined)

# Placeholder for WIP code
def mFoldValid_devSup():
    print '{}th set:'.format(mthSet)
    for samp in trSet:
        print samp
    print '\n\n'

# m = n?  See project requirements
def mFoldValid(XS, m):
    print 'Entered mFoldValid()'
    fName_OUT = 'm-foldValid.txt'
    mSets = getMSets(XS, 10)

    # go through sets
    for mthSet in range(len(mSets)):
        te = mSets[mthSet]
        trS = combineSetsExcept(mSets, mthSet)                            # trS is
        tr = weave(trS)
        # Normalize tr set
        trColMeans = getColumnMeans(tr)
```

```
        trColStddevs = getColumnStddevs(tr)
        nTr = normalize(tr, trColMeans, trColStddevs)
        nTrS = normalizeSep(trS, trColMeans, trColStddevs)
        # Normalize te set
        teColMeans = getColumnMeans(te)
        teColStddevs = getColumnStddevs(te)
        nTe = normalize(trSet, teColMeans, teColStddevs)
        # CLASSIFY using tr
        #rebuild(trS, fName_OUT)
        #call(['knn.py'])
        # TEST using te
        #call(['knn.py'])

def test_mFoldValid():
    print 'Entered test_mFoldValid()'
    fName_tr = 'poker-hand-training-true.data'
    XS = getInputSeparated(fName_tr)
    mFoldValid(XS, 10)

def main():
    if DEBUG: print 'Entered main()'
    fName_IN = 'poker-hand-training-true.data'
    #fName_IN = 'poker-hand-testing.data'
    fName_OUT = 'poker-hand-training-true-normalized.data'
    #fName_OUT = 'poker-hand-testing-normalized.data'

    # 'X' denotes original dataset
    # 'XS' denotes original dataset separated by class (hand type)
    XS = getInputSeparated(fName_IN)
    #if DEBUG: print 'XS: \n{}'.format(XS)
    X = weave(XS)

    if DEBUG: print 'X:\n{}\n\n\n'.format(X)
    if DEBUG: print 'XS:\n{}\n\n\n'.format(XS)

    # Compute un-normalized training mu & sigma for normalization
    colMeans_all_tr = getColumnMeans(X)
    colStddevs_all_tr = getColumnStddevs(X)

    # 'nX' denotes original dataset - normalized
    # 'nXS' denotes original dataset, separated - normalized
    nXS = normalizeSep(XS, colMeans_all_tr, colStddevs_all_tr)
    nX = weave(nXS)                              # nX is both classes con

    if DEBUG: print 'nX:\n{}\n\n\n'.format(nX)
    #if DEBUG:
```

```
        print 'nXS:\n{}'.format(nXS)

        # Create normalized file
        #rebuild(nXS, fName_OUT)

        # M-fold validation
        mfoldValid(XS, 10)



main()
#test_weave()
#test_getMSets()
#test_mFoldValid()
```