# SMAI ASSIGNMENT 1

**Karan Mangla**
**201301205**

In this assignment , I have implemented the K- nearest neighbour algorithm for 4 data sets . So an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors. If k = 1, then the object is simply assigned to the class of that single nearest neighbor.
For assigning the class , we have used polling which is a majority voting i.e class of the object is the one which has occurred the maximum of times in its k-nearest neighbours .
The following pre-processing has been applied to the dataset before splitting it into the training Set and the test Set according to the k-fold :

- Normalizing of the dataset so as to counter to the domination of the euclidean distance by a single feature of the data entry .This has been done in 2 ways depending on the data set :
  - Dividing each attribute by the max value of that attribute in the dataset .
  - Each attribute is changed to (val - min[attr]) / (max[attr]-min[attr]) .
- Shuffling the elements so that the dataset is randomly distributed and the training and test data can both have all types of class elements .

I have taken 2 types of distances according to the dataset and chose the one giving more accuracy namely Euclidean and Manhattan distance .
The class that occurs the most in the polling is assigned as the class of the particular point that is being tested from the testing set .

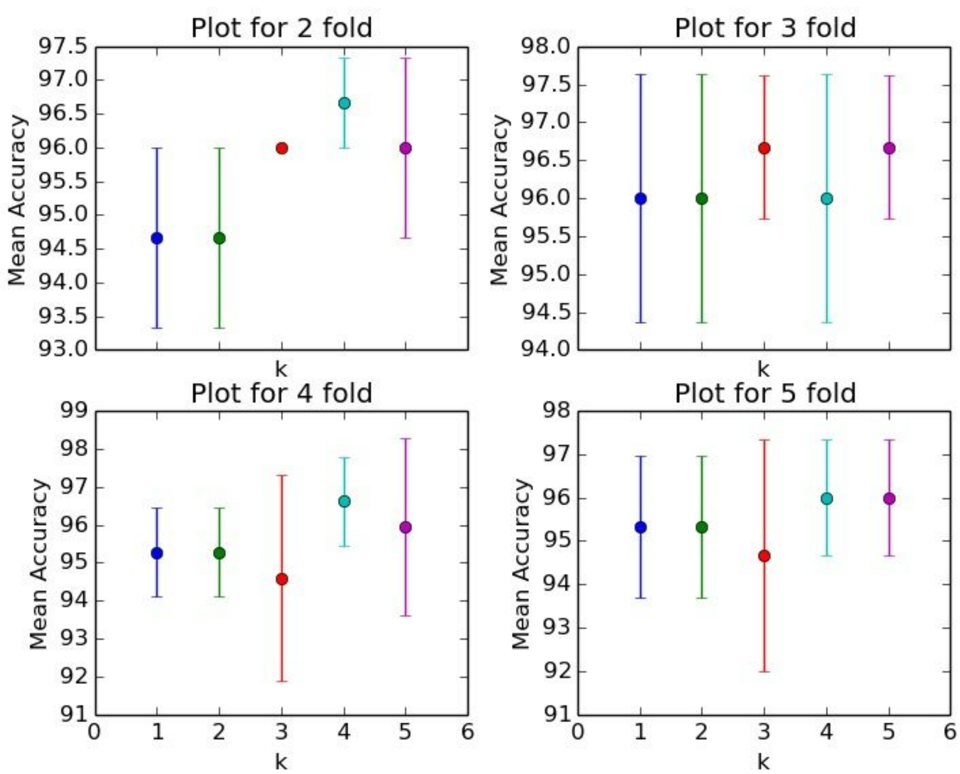
The ties are handled by maintaining a sum of the inverse of the distances of the object from the k-nearest neighbours of a particular class . So whenever a tie occurs we just look at the class which has the maximum sum of inverse distances ( as we have taken sum of inverse of distances) .

The procedure followed was that for each k and n-fold , iterate over the blocks formed , taking one as the test and the other as the training data .Take a mean of the accuracies over all the blocks for all the k values and then plot the graph .

# Results

The results of the 4 datasets are as follows :

**Iris (Contains 150 data points, 3 classes, and 4 features)**
**Distance function** : I used  Euclidean distance for this dataset to find the nearest neighbours, and then a polling was done between k nearest neighbours . The maximum repeated class in the polls is given to the test data sample .
**Tie Breaker** : The reciprocal of the distances were added for the nearest neighbours belonging to a particular class, so that more weightage was given to the points near i.e more is the sum closer are those points to the given instance and thus same class assigned .
**Feature Normalization** :Each attribute = (val[attr] - min[attr]) / (max[attr]-min[attr]) . This is done so as to reduce to domination of certain features .



**Code :**

```
import csv
import random
import math
import operator
import matplotlib.pyplot as plt
import numpy as np

def normalise(set_data):

        for x in range(len(set_data)):
           for y in range(4):
              set_data[x][y] = float(set_data[x][y])
        elements = len(set_data)
        features = len(set_data[0])
        max_stor = []
        min_stor = []
        for i in xrange(0,features-1):
                maxe = 0
                mine =100
                for j in xrange(0,elements):
                        if set_data[j][i] > maxe:
                                maxe = set_data[j][i]
                        if set_data[j][i] < mine:
                                mine = set_data[j][i]
                max_stor.append(maxe)
                min_stor.append(mine)
        for i in xrange(0,elements):
                for j in xrange(0,features-1):
                        set_data[i][j] = (set_data[i][j] -min_stor[j]) / (max_stor[j] - min_stor[j])



def loadDataset(count,fold_elements, dataset, trainingSet=[] , testSet=[]):

        for x in range(len(dataset)):
           for y in range(4):
              dataset[x][y] = float(dataset[x][y])
        #token = int(len(dataset)*split)
```

```
	for i in xrange(count,count + fold_elements):
		testSet.append(dataset[i])
	for i in xrange(0,len(dataset)):
		if i not in xrange(count,count + fold_elements):
			trainingSet.append(dataset[i])


def getResponse(neighbors):

	classVotes = {}
	invd = {}
	for x in range(len(neighbors)):
		response = neighbors[x][0][-1]
		inv = neighbors[x][1]
		if response in classVotes:
			classVotes[response] += 1
			invd[response] += inv
		else:
			classVotes[response] = 1
			invd[response] = inv
	sortedVotes = sorted(classVotes.iteritems(), key=operator.itemgetter(1),
reverse=True)
	DisVotes = sorted(invd.iteritems(), key=operator.itemgetter(1), reverse=True)
	#print sortedVotes
	if len(sortedVotes) == 1 or sortedVotes[0][1] > sortedVotes[1][1] :
		return sortedVotes[0][0]
	else :
		return DisVotes[0][0]

def getNeighbors(trainingSet, testInstance, k):
	distances = []
	length = len(testInstance)-1
	for x in range(len(trainingSet)):
		#print testInstance
		dist = euclideanDistance(testInstance, trainingSet[x], length)
		if dist > 0.0 :
			distances.append((trainingSet[x], dist, 1/dist))
		else :
```

```
                distances.append((trainingSet[x], dist, 100000)) # for bank note
1000000
        distances.sort(key=operator.itemgetter(1))
        #if distances[0][1] > 0.0 : print 1/distances[0][1]
        neighbors = []
        for x in range(k):
                neighbors.append((distances[x][0],distances[x][2]))
        return neighbors

def euclideanDistance(instance1, instance2, length):
        distance = 0
        for x in range(length):
                #print x
                distance += pow((instance1[x] - instance2[x]), 2)

        return math.sqrt(distance)



def getAccuracy(testSet, predictions):
        correct = 0
        for x in range(len(testSet)):
                if testSet[x][-1] == predictions[x]:
                        correct += 1
        return (correct/float(len(testSet))) * 100.0

def plot(karray, mean_of_chunks, stdev, axs, kfold):
        ax = axs[kfold/4, (kfold%2)]
        #       plt.figure()
        ax.errorbar(karray, mean_of_chunks      , yerr=stdev, fmt = 'o')
        ax.set_title('Plot for '+str(kfold)+' fold')
        ax.set_xlabel('k')
        ax.set_ylabel('Mean Accuracy')

if __name__ == '__main__':
        trainingSet = []
        testSet = []
        dilim = 0.50
        filename = 'iris.data'
        with open(filename, 'rb') as csvfile:
```

```
    rows = csv.reader(csvfile)
    #print rows
    dataset = list(rows)
    dataset = [ x for x in dataset if len(x) > 0]

    random.shuffle(dataset)
  #dataset has been set
  normalise(dataset)
  #print dataset
  total_elements = len(dataset)

  fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True)

  for fold in xrange(2,6):


      #mean = [0,0,0,0,0,0]
      for k in xrange(1,6):
          fold_elements = total_elements/fold
          count =0
          #accuracy =0
          accuracy = []
          mean_of_chunks = []
          stdev = []
          for index in xrange(0,fold):
              trainingSet = []
              testSet = []
              loadDataset(count,fold_elements, dataset, trainingSet,
testSet)
              #print count , fold_elements
              count += fold_elements
              #print len(testSet) , len(trainingSet)
              predictions=[]
              #k = 3

              #for k in xrange(1,6):

              for x in range(len(testSet)):
                  neighbors = getNeighbors(trainingSet, testSet[x], k)
```

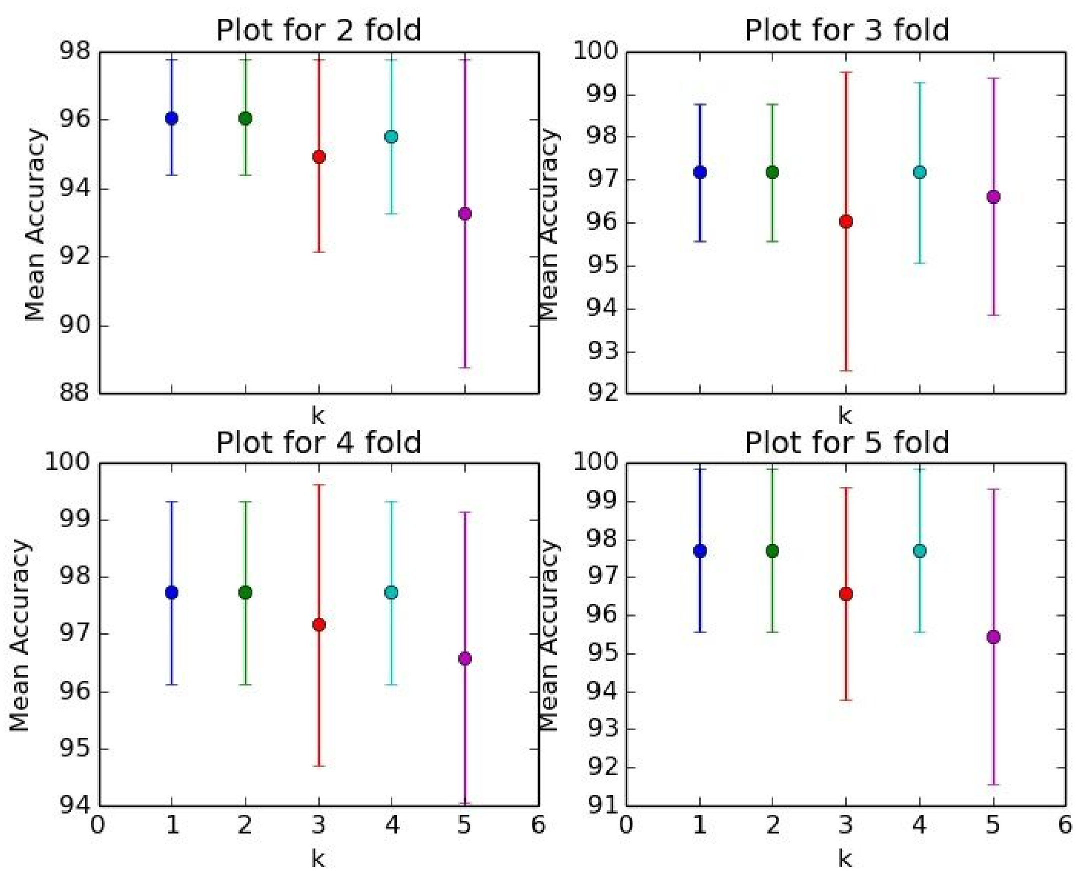
```
                    #print neighbors
                    result = getResponse(neighbors)
                    #print result
                    predictions.append(result)
               accuracy.append(getAccuracy(testSet, predictions))
          #mean[k] += accuracy
          #print accuracy/fold  , fold , k
          print accuracy
          print np.mean(accuracy)
          print np.std(accuracy)

          mean_of_chunks.append(np.mean(accuracy))
          stdev.append(np.std(accuracy))
          plot(k, mean_of_chunks, stdev, axs, fold)
plt.show()
```

**Wine (Contains 178 data points, 3 classes, and 13 features)**

**Distance function** : I used  Manhattan distance for this dataset to find the nearest neighbours, and then a polling was done between k nearest neighbours . The maximum repeated class in the polls is given to the test data sample . The error was found to reduce to certain extent .

**Tie Breaker** : The reciprocal of the distances were added for the nearest neighbours belonging to a particular class, so that more weightage was given to the points near i.e more is the sum closer are those points to the given instance and thus same class assigned .

**Feature Normalization** :Each attribute = (val[attr]) / (max[attr]) .This is done so as to reduce to domination of certain features(many have high value as compared to previous one) .



**Code**

```
import csv
import random
import math
import operator
import matplotlib.pyplot as plt
import numpy as np

def normalise(set_data):

        for x in range(len(set_data)):
           for y in xrange(1,14):
              set_data[x][y] = float(set_data[x][y])
        elements = len(set_data)
        features = len(set_data[0])
        max_stor = []
        for i in xrange(1,features):
                maxe = 0
                for j in xrange(0,elements):
                        if set_data[j][i] > maxe:
                                maxe = set_data[j][i]
                max_stor.append(maxe)
        for i in xrange(0,elements):
                for j in xrange(1,features):
                        set_data[i][j] /= max_stor[j-1]


def loadDataset(count,fold_elements, dataset, trainingSet=[] , testSet=[]):

        for x in range(len(dataset)):
           for y in range(1,14):
              dataset[x][y] = float(dataset[x][y])
        #token = int(len(dataset)*split)
        for i in xrange(count,count + fold_elements):
                testSet.append(dataset[i])
        for i in xrange(0,len(dataset)):
                if i not in xrange(count,count + fold_elements):
                        trainingSet.append(dataset[i])
```

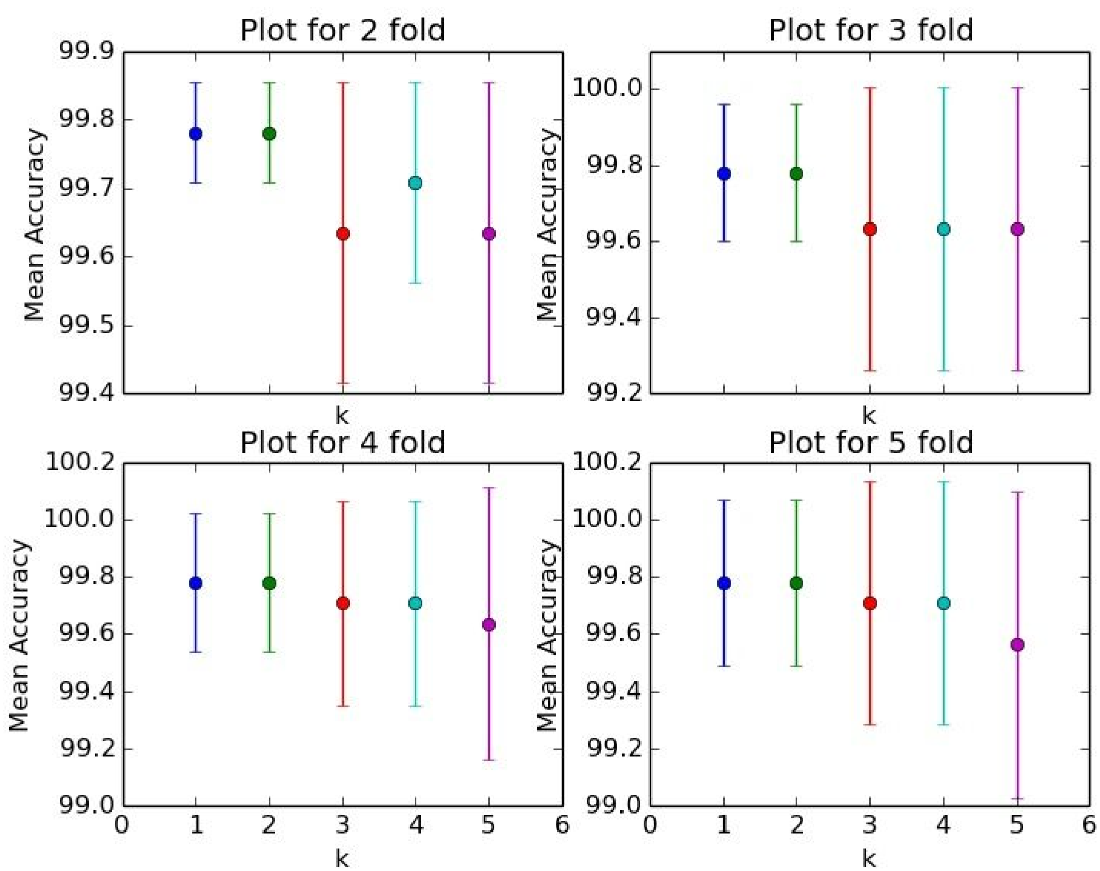
```
def getResponse(neighbors):
	classVotes = {}
	invd = {}
	for x in range(len(neighbors)):
		response = neighbors[x][0][0]
		inv = neighbors[x][1]
		if response in classVotes:
			classVotes[response] += 1
			invd[response] += inv
		else:
			classVotes[response] = 1
			invd[response] = inv
	sortedVotes = sorted(classVotes.iteritems(), key=operator.itemgetter(1),
reverse=True)
	DisVotes = sorted(invd.iteritems(), key=operator.itemgetter(1), reverse=True)
	#print sortedVotes
	if len(sortedVotes) == 1 or sortedVotes[0][1] > sortedVotes[1][1]:
		return sortedVotes[0][0]
	else :
		#print DisVotes
		#print sortedVotes
		return DisVotes[0][0]

def getNeighbors(trainingSet, testInstance, k):
	distances = []
	length = len(testInstance)

	for x in xrange(0,len(trainingSet)):
		#print testInstance
		dist = manhattanDistance(testInstance, trainingSet[x], length)
		if dist > 0.0 :
			distances.append((trainingSet[x], dist, 1/dist))
		else :
			distances.append((trainingSet[x], dist, 100000))
	distances.sort(key=operator.itemgetter(1))
	neighbors = []
	for x in range(k):
		neighbors.append((distances[x][0],distances[x][2]))
```

```
        return neighbors

def euclideanDistance(instance1, instance2, length):
        distance = 0
        for x in xrange(1,length):
                #print x
                distance += pow((instance1[x] - instance2[x]), 2)

        return math.sqrt(distance)

def manhattanDistance(instance1, instance2, length):
        distance = 0
        for x in range(1,length):
                #print x
                distance += abs(instance1[x] - instance2[x])

        return distance

def getAccuracy(testSet, predictions):
        correct = 0
        for x in range(len(testSet)):
                if testSet[x][0] == predictions[x]:
                        correct += 1
        return (correct/float(len(testSet))) * 100.0

def plot(karray, mean_of_chunks, stdev, axs, kfold):
        ax = axs[kfold/4, (kfold%2)]
        #       plt.figure()
        ax.errorbar(karray, mean_of_chunks      , yerr=stdev, fmt = 'o')
        ax.set_title('Plot for '+str(kfold)+' fold')
        ax.set_xlabel('k')
        ax.set_ylabel('Mean Accuracy')

if __name__ == '__main__':
        trainingSet = []
        testSet = []
        dilim = 0.50
        filename = 'wine.data'
        with open(filename, 'rb') as csvfile:
```

```
        rows = csv.reader(csvfile)
        #print rows
        dataset = list(rows)
        dataset = [ x for x in dataset if len(x) > 0]
        for i in xrange(0,5):
            random.shuffle(dataset)
    #dataset has been set
    #print dataset
    normalise(dataset)
    total_elements = len(dataset)
    fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True)

    for fold in xrange(2,6):



        #mean = [0,0,0,0,0,0]
        for k in xrange(1,6):
            fold_elements = total_elements/fold
            count =0
            accuracy = []
            mean_of_chunks = []
            stdev = []
            for index in xrange(0,fold):
                trainingSet = []
                testSet = []
                loadDataset(count,fold_elements, dataset, trainingSet,
testSet)
                #print count , fold_elements
                count += fold_elements
                #print len(testSet) , len(trainingSet)
                predictions=[]
                #k = 3

                #for k in xrange(1,6):

                for x in range(len(testSet)):
                    neighbors = getNeighbors(trainingSet, testSet[x], k)
                    #print neighbors
                    result = getResponse(neighbors)
```

```
                #print result
                predictions.append(result)
            accuracy.append(getAccuracy(testSet, predictions))
            #mean[k] += accuracy
        #print accuracy/fold  , fold , k
        print accuracy
        print np.mean(accuracy)
        print np.std(accuracy)
        mean_of_chunks.append(np.mean(accuracy))
        stdev.append(np.std(accuracy))
        plot(k, mean_of_chunks, stdev, axs, fold)
plt.show()
```

**Banknote(Contains 1372 data points, 2 classes, and 4 features)**
**Distance function** : I used  Euclidean distance for this dataset to find the nearest neighbours, and then a polling was done between k nearest neighbours . The maximum repeated class in the polls is given to the test data sample .
**Tie Breaker** : The reciprocal of the distances were added for the nearest neighbours belonging to a particular class, so that more weightage was given to the points near i.e more is the sum closer are those points to the given instance and thus same class assigned .
**Feature Normalization** :Each attribute = (val[attr]) / (max[attr]) .This is done so as to reduce to domination of certain features(many have high value as compared to previous one) .



**Code**

```
import csv
import random
import math
import operator
import matplotlib.pyplot as plt
import numpy as np

def normalise(set_data):

        for x in range(len(set_data)):
           for y in range(4):
              set_data[x][y] = float(set_data[x][y])
        elements = len(set_data)
        features = len(set_data[0])
        max_stor = []
        for i in xrange(0,features-1):
                maxe = 0
                for j in xrange(0,elements):
                        if set_data[j][i] > maxe:
                                maxe = set_data[j][i]
                max_stor.append(maxe)
        for i in xrange(0,elements):
                for j in xrange(0,features-1):
                        set_data[i][j] /= max_stor[j]



def loadDataset(count,fold_elements, dataset, trainingSet=[] , testSet=[]):

        for x in range(len(dataset)):
           for y in range(4):
              dataset[x][y] = float(dataset[x][y])
        #token = int(len(dataset)*split)
        for i in xrange(count,count + fold_elements):
                testSet.append(dataset[i])
        for i in xrange(0,len(dataset)):
                if i not in xrange(count,count + fold_elements):
                        trainingSet.append(dataset[i])
```

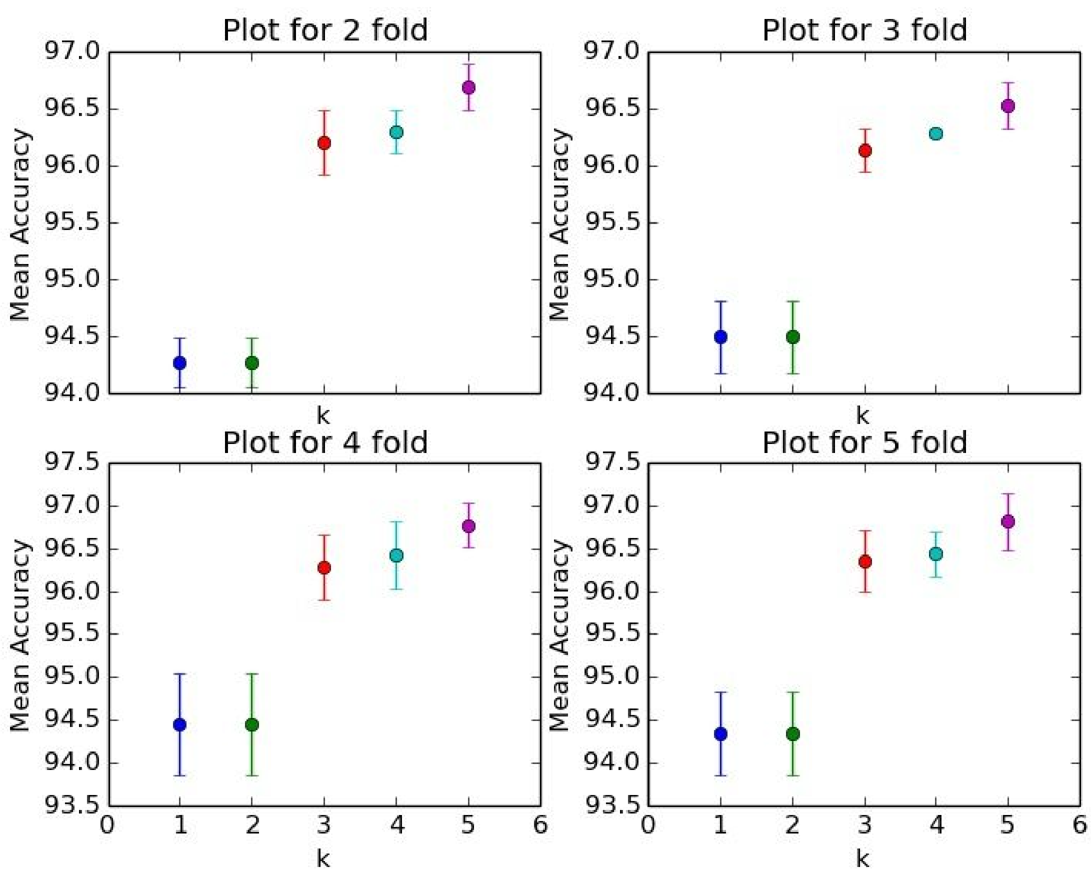
```
def getResponse(neighbors):

	classVotes = {}
	invd = {}
	for x in range(len(neighbors)):
		response = neighbors[x][0][-1]
		inv = neighbors[x][1]
		if response in classVotes:
			classVotes[response] += 1
			invd[response] += inv
		else:
			classVotes[response] = 1
			invd[response] = inv
	sortedVotes = sorted(classVotes.iteritems(), key=operator.itemgetter(1),
reverse=True)
	DisVotes = sorted(invd.iteritems(), key=operator.itemgetter(1), reverse=True)
	#print sortedVotes
	if len(sortedVotes) == 1 or sortedVotes[0][1] > sortedVotes[1][1] :
		return sortedVotes[0][0]
	else :
		return DisVotes[0][0]

def getNeighbors(trainingSet, testInstance, k):
	distances = []
	length = len(testInstance)-1
	for x in range(len(trainingSet)):
		#print testInstance
		dist = euclideanDistance(testInstance, trainingSet[x], length)
		if dist > 0.0 :
			distances.append((trainingSet[x], dist, 1/dist))
		else :
			distances.append((trainingSet[x], dist, 1000000)) # for bank note
1000000
	distances.sort(key=operator.itemgetter(1))
	#if distances[0][1] > 0.0 : print 1/distances[0][1]
	neighbors = []
	for x in range(k):
```

```
			neighbors.append((distances[x][0],distances[x][2]))
		return neighbors

def euclideanDistance(instance1, instance2, length):
		distance = 0
		for x in range(length):
			#print x
			distance += pow((instance1[x] - instance2[x]), 2)

		return math.sqrt(distance)

def manhattanDistance(instance1, instance2, length):
		distance = 0
		for x in range(length):
			#print x
			distance += abs(instance1[x] - instance2[x])

		return distance


def getAccuracy(testSet, predictions):
		correct = 0
		for x in range(len(testSet)):
			if testSet[x][-1] == predictions[x]:
				correct += 1
		return (correct/float(len(testSet))) * 100.0

def plot(karray, mean_of_chunks, stdev, axs, kfold):
		ax = axs[kfold/4, (kfold%2)]
		#	plt.figure()
		ax.errorbar(karray, mean_of_chunks	, yerr=stdev, fmt = 'o')
		ax.set_title('Plot for '+str(kfold)+' fold')
		ax.set_xlabel('k')
		ax.set_ylabel('Mean Accuracy')

if __name__ == '__main__':
		trainingSet = []
		testSet = []
		dilim = 0.50
```

```
filename = 'data_banknote_authentication.txt'
with open(filename, 'rb') as csvfile:
    rows = csv.reader(csvfile)
    #print rows
    dataset = list(rows)
    dataset = [ x for x in dataset if len(x) > 0]

    random.shuffle(dataset)
#dataset has been set
normalise(dataset)
#print dataset
total_elements = len(dataset)

fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True)

for fold in xrange(2,6):


        #mean = [0,0,0,0,0,0]
        for k in xrange(1,6):
                fold_elements = total_elements/fold
                count =0
                #accuracy =0
                accuracy = []
                mean_of_chunks = []
                stdev = []
                for index in xrange(0,fold):
                        trainingSet = []
                        testSet = []
                        loadDataset(count,fold_elements, dataset, trainingSet,
testSet)
                        #print count , fold_elements
                        count += fold_elements
                        #print len(testSet) , len(trainingSet)
                        predictions=[]
                        #k = 3

                        #for k in xrange(1,6):
```

```
        for x in range(len(testSet)):
            neighbors = getNeighbors(trainingSet, testSet[x], k)
            #print neighbors
            result = getResponse(neighbors)
            #print result
            predictions.append(result)
        accuracy.append(getAccuracy(testSet, predictions))
    #mean[k] += accuracy
    #print accuracy/fold  , fold , k
    print accuracy
    print np.mean(accuracy)
    print np.std(accuracy)

    mean_of_chunks.append(np.mean(accuracy))
    stdev.append(np.std(accuracy))
    plot(k, mean_of_chunks, stdev, axs, fold)
plt.show()
```

**Twonorm(Contains 7400 data points, 2 classes, and 20 features)**
**Distance function** : I used Euclidean distance for this dataset to find the nearest neighbours, and then a polling was done between k nearest neighbours . The maximum repeated class in the polls is given to the test data sample .
**Tie Breaker** : The reciprocal of the distances were added for the nearest neighbours belonging to a particular class, so that more weightage was given to the points near i.e more is the sum closer are those points to the given instance and thus same class assigned .
**Feature Normalization** :Each attribute = (val[attr]) / (max[attr]) .This is done so as to reduce to domination of certain features(many have high value as compared to previous one) .



**Code**

```
import csv
import random
import math
import operator
import matplotlib.pyplot as plt
import numpy as np

def normalise(set_data):

        for x in range(len(set_data)):
           for y in range(20):
              set_data[x][y] = float(set_data[x][y])
        elements = len(set_data)
        features = len(set_data[0])
        max_stor = []
        for i in xrange(0,features-1):
                maxe = 0
                for j in xrange(0,elements):
                        if set_data[j][i] > maxe:
                                maxe = set_data[j][i]
                max_stor.append(maxe)
        for i in xrange(0,elements):
                for j in xrange(0,features-1):
                        set_data[i][j] /= max_stor[j]



def loadDataset(count,fold_elements, dataset, trainingSet=[] , testSet=[]):

        for x in range(len(dataset)):
           for y in range(20):
              dataset[x][y] = float(dataset[x][y])
        #token = int(len(dataset)*split)
        for i in xrange(count,count + fold_elements):
                testSet.append(dataset[i])
        for i in xrange(0,len(dataset)):
                if i not in xrange(count,count + fold_elements):
                        trainingSet.append(dataset[i])
```

```
def getResponse(neighbors):

    classVotes = {}
    invd = {}
    for x in range(len(neighbors)):
        response = neighbors[x][0][-1]
        inv = neighbors[x][1]
        if response in classVotes:
            classVotes[response] += 1
            invd[response] += inv
        else:
            classVotes[response] = 1
            invd[response] = inv
    sortedVotes = sorted(classVotes.iteritems(), key=operator.itemgetter(1),
reverse=True)
    DisVotes = sorted(invd.iteritems(), key=operator.itemgetter(1), reverse=True)
    #print sortedVotes
    if len(sortedVotes) == 1 or sortedVotes[0][1] > sortedVotes[1][1] :
        return sortedVotes[0][0]
    else :
        return DisVotes[0][0]

def getNeighbors(trainingSet, testInstance, k):
    distances = []
    length = len(testInstance)-1
    for x in range(len(trainingSet)):
        #print testInstance
        dist = euclideanDistance(testInstance, trainingSet[x], length)
        if dist > 0.0 :
            distances.append((trainingSet[x], dist, 1/dist))
        else :
            distances.append((trainingSet[x], dist, 100000)) # for bank note
1000000
    distances.sort(key=operator.itemgetter(1))
    #if distances[0][1] > 0.0 : print 1/distances[0][1]
    neighbors = []
    for x in range(k):
```

```
            neighbors.append((distances[x][0],distances[x][2]))
    return neighbors

def euclideanDistance(instance1, instance2, length):
    distance = 0
    for x in range(length):
        #print x
        distance += pow((instance1[x] - instance2[x]), 2)

    return math.sqrt(distance)

def getAccuracy(testSet, predictions):
    correct = 0
    for x in range(len(testSet)):
        if testSet[x][-1] == predictions[x]:
            correct += 1
    return (correct/float(len(testSet))) * 100.0

def plot(karray, mean_of_chunks, stdev, axs, kfold):
    ax = axs[kfold/4, (kfold%2)]
    #       plt.figure()
    ax.errorbar(karray, mean_of_chunks       , yerr=stdev, fmt = 'o')
    ax.set_title('Plot for '+str(kfold)+' fold')
    ax.set_xlabel('k')
    ax.set_ylabel('Mean Accuracy')

if __name__ == '__main__':
    trainingSet = []
    testSet = []
    dilim = 0.50
    filename = 'Twonorm.data'
    with open(filename, 'rb') as csvfile:
       rows = csv.reader(csvfile)
       dataset = []
       for i in rows:
           #print i[0].split(" ")
           temp = [ x for x in i[0].split(" ") if len(x) > 0]
           dataset.append(temp)
       #dataset = list(rows)
```

```
    dataset = [ x for x in dataset if len(x) > 0]

    random.shuffle(dataset)
  #dataset has been set
  #
  normalise(dataset)
  #print dataset
  total_elements = len(dataset)

  fig, axs = plt.subplots(nrows=2, ncols=2, sharex=True)

  for fold in xrange(2,6):


      #mean = [0,0,0,0,0,0]
      for k in xrange(1,6):
          fold_elements = total_elements/fold
          count =0
          #accuracy =0
          accuracy = []
          mean_of_chunks = []
          stdev = []
          for index in xrange(0,fold):
              trainingSet = []
              testSet = []
              loadDataset(count,fold_elements, dataset, trainingSet,
testSet)
              #print count , fold_elements
              count += fold_elements
              #print len(testSet) , len(trainingSet)
              predictions=[]
              #k = 3

              #for k in xrange(1,6):

              for x in range(len(testSet)):
                  neighbors = getNeighbors(trainingSet, testSet[x], k)
                  #print neighbors
                  result = getResponse(neighbors)
```

```
                #print result
                predictions.append(result)
            accuracy.append(getAccuracy(testSet, predictions))
        #mean[k] += accuracy
        #print accuracy/fold  , fold , k
        print accuracy
        print np.mean(accuracy)
        print np.std(accuracy)

        mean_of_chunks.append(np.mean(accuracy))
        stdev.append(np.std(accuracy))
        plot(k, mean_of_chunks, stdev, axs, fold)
plt.show()
```