COMP6630: Mini Project One

By Colin Vande Vijvere

Coding the KNN algorithm

I chose to code my KNN algorithm, in the following way. I coded 2 different algorithms, one for Manhattan distance, and one for Euclidean distance. Then I coded a “KNN_tests” file, where I manipulated the test data and called my algorithms so they could perform their calculations. The code, you can see below.

KNN using Euclidean distance:

```python
import numpy as np
from collections import Counter

#Euclidian function
def euclidian_distance(x1,x2):
    return np.sqrt(np.sum((x1-x2)**2))

class KNNeuclidean:
    # neirest neighbors = 3
    def __init__(self, k=3):
        self.k = k

    # store training samples
    # X = training samples, y = training lables
    def fit(self, X, y):
        self.x_train = X
        self.y_train = y


    #this will predict
    def predict(self, X):
        #call predict for all samples in X
        predicted_labels = [self._predict(x) for x in X]
        return np.array(predicted_labels)

    def _predict(self, x):
        # compute distances
        distances = [euclidian_distance(x, x_train) for x_train in self.x_train]

        # get k neirest samples and labels,
        # sort array by distances and pick the ones from 0 -> k
        k_indices = np.argsort(distances)[:self.k]

        #get the labels of k-neirest neighbors
        k_neirest_labels = [self.y_train[i] for i in k_indices]

        # majority vote, mostcommon class label wins
        most_common = Counter(k_neirest_labels).most_common(1)
        return most_common[0][0]
```

KNN using Manhattan distance:

```
import numpy as np
from collections import Counter

#Euclidian function
def manhattan_distance(x1,x2):
    return np.abs(np.sum((x1-x2)))

class KNNmanhattan:
    # neirest neighbors = 3
    def __init__(self, k=3):
        self.k = k

    # store training samples
    # X = training samples, y = training lables
    def fit(self, X, y):
        self.x_train = X
        self.y_train = y


    #this will predict
    def predict(self, X):
        #call predict for all samples in X
        predicted_labels = [self._predict(x) for x in X]
        return np.array(predicted_labels)

    def _predict(self, x):
        # compute distances
        distances = [manhattan_distance(x, x_train) for x_train in self.x_train]

        # get k neirest samples and labels,
        # sort array by distances and pick the ones from 0 -> k
        k_indices = np.argsort(distances)[:self.k]

        #get the labels of k-neirest neighbors
        k_neirest_labels = [self.y_train[i] for i in k_indices]

        # majority vote, mostcommon class label wins
        most_common = Counter(k_neirest_labels).most_common(1)
        return most_common[0][0]
```

Knn_tests.py:

```
from knnManhattan import KNNmanhattan
from knnEuclidean import KNNeuclidean
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from libsvm.svmutil import svm_read_problem

cmap = ListedColormap(['#FF0000', '#00FF00', '#0000FF'])
k = 7

# -----------------------------------------------------------
# function to calculate accuracy of knn
# -----------------------------------------------------------
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy

# -----------------------------------------------------------
# Function To Shuffle and Split the Data
# -----------------------------------------------------------
def shuffle_split_data(X, y, ratio):
    split = np.random.rand(X.shape[0]) < ratio

    X_Train = X[split]
    y_Train = y[split]
    X_Test = X[~split]
    y_Test = y[~split]

    return X_Train, y_Train, X_Test, y_Test

```

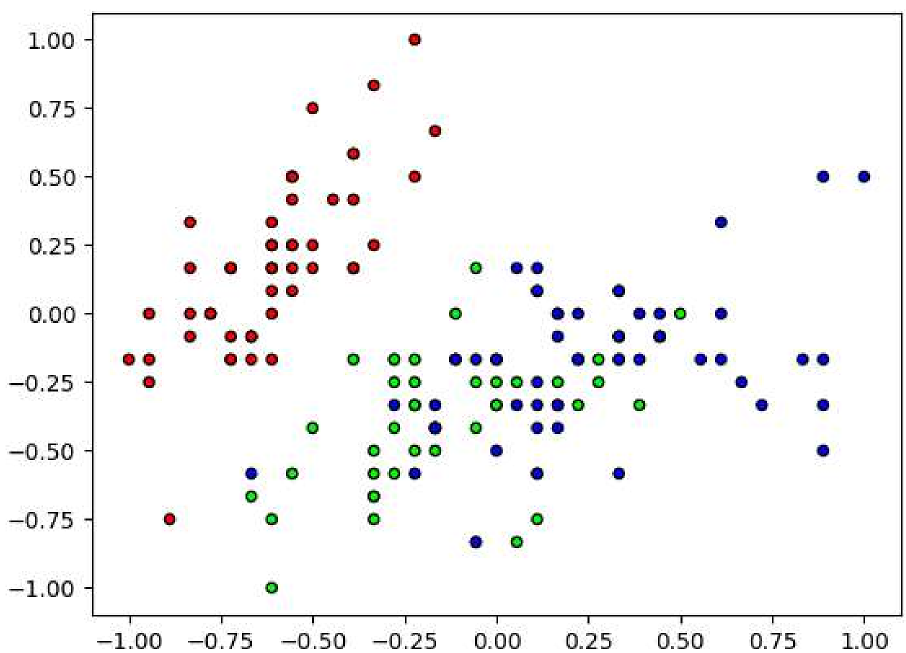
```
# -----------------------------------------------------
# formatting iris.scale data
# -----------------------------------------------------
y_raw, x_raw = svm_read_problem('iris.scale')
length_y = len(y_raw)
y = np.array(y_raw)
length_x = len(x_raw)
x = np.zeros((len(y_raw), 4))
for a in range(length_y):
    for b, c in x_raw[a].items():
        x[a][b-1] = c

x_train, y_train, x_test, y_test = shuffle_split_data(x, y, 0.6)
length_y_test = len(y_test)
```

```python
# ---------------------------------------------------------------------
# RUNNING KNN ON iris.scale USING EUCLIDEAN DISTANCE
# ---------------------------------------------------------------------
clf = KNNeuclidean(k=k)
clf.fit(x_train, y_train)
predictions = clf.predict(x_test)
print("\n")
print("==== KNN classification (euclidean distance) ====", sep='')
print("dataset: iris.scale", sep='')
print("# nearest neighbors (k): ", k, sep='')
print("total predicted: ", len(y_test), sep='')
print("correct predictions: ", np.sum(y_test == predictions), sep='')
print("accuracy: ", accuracy(y_test, predictions), sep='  ')
print("\n")


# ---------------------------------------------------------------------
# RUNNING KNN ON iris.scale USING MANHATTAN DISTANCE
# ---------------------------------------------------------------------
clf = KNNmanhattan(k=k)
clf.fit(x_train, y_train)
predictions = clf.predict(x_test)
print("==== KNN classification (manhattan distance) ====", sep='')
print("dataset: iris.scale", sep='')
print("# nearest neighbors (k): ", k, sep='')
print("total predicted: ", len(y_test), sep='')
print("correct predictions: ", np.sum(y_test == predictions), sep='')
print("accuracy: ", accuracy(y_test, predictions), sep='  ')
print("\n")

 Plotting graph
 plt.figure()
 plt.scatter(x[:, 0], x[:, 1], c=y, cmap=cmap, edgecolor='k', s=20)
 plt.show()
```

```python
# ---------------------------------------------------------------------
# FORMATTING a4a DATA + figuring out value for k
# ---------------------------------------------------------------------
y_raw, x_raw = svm_read_problem('a4aSubset')
length_y = len(y_raw)
y = np.array(y_raw)
length_x = len(x_raw)
x = np.zeros((len(y_raw), 400))
for a in range(length_y):
    for b, c in x_raw[a].items():
        x[a][b-1] = c


x_train, y_train, x_test, y_test = shuffle_split_data(x, y, 0.6)
length_y_test = len(y_test)



# ---------------------------------------------------------------------
# RUNNING KNN ON iris.scale USING EUCLIDEAN DISTANCE
# ---------------------------------------------------------------------
clf = KNNeuclidean(k=k)
clf.fit(x_train, y_train)
predictions = clf.predict(x_test)
print("\n")
print("==== KNN classification (euclidean distance) ====", sep='')
print("dataset: subset of a4a", sep='')
print("# nearest neighbors (k): ", k, sep='')
print("total predicted: ", len(y_test), sep='')
print("correct predictions: ", np.sum(y_test == predictions), sep='')
print("accuracy: ", accuracy(y_test, predictions), sep=' ')
print("\n")
```

```
# ------------------------------------------------------------------------
# RUNNING KNN ON iris.scale USING MANHATTAN DISTANCE
# ------------------------------------------------------------------------
clf = KNNmanhattan(k=k)
clf.fit(x_train, y_train)
predictions = clf.predict(x_test)
print("==== KNN classification (manhattan distance) ====", sep='')
print("dataset: subset of a4a", sep='')
print("# nearest neighbors (k): ", k, sep='')
print("total predicted: ", len(y_test), sep='')
print("correct predictions: ", np.sum(y_test == predictions), sep='')
print("accuracy: ", accuracy(y_test, predictions), sep='  ')
print("\n")

 #Plotting graph
 plt.figure()
 plt.scatter(x[:, 0], x[:, 1], c=y, cmap=cmap, edgecolor='k', s=20)
 plt.show()

```

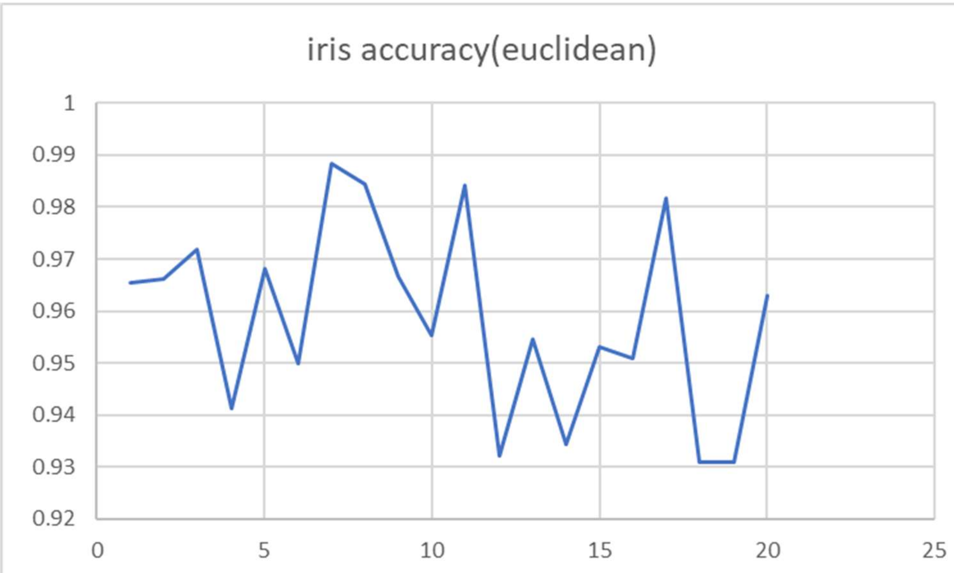
Note: if you've look closely, you can see that I've used matplotlib to visualize my data, here is the graph it created for the iris dataset. (I chose to exclude the a4a graph since there is not much to see)



Accuracy of KNN in relation with the k value

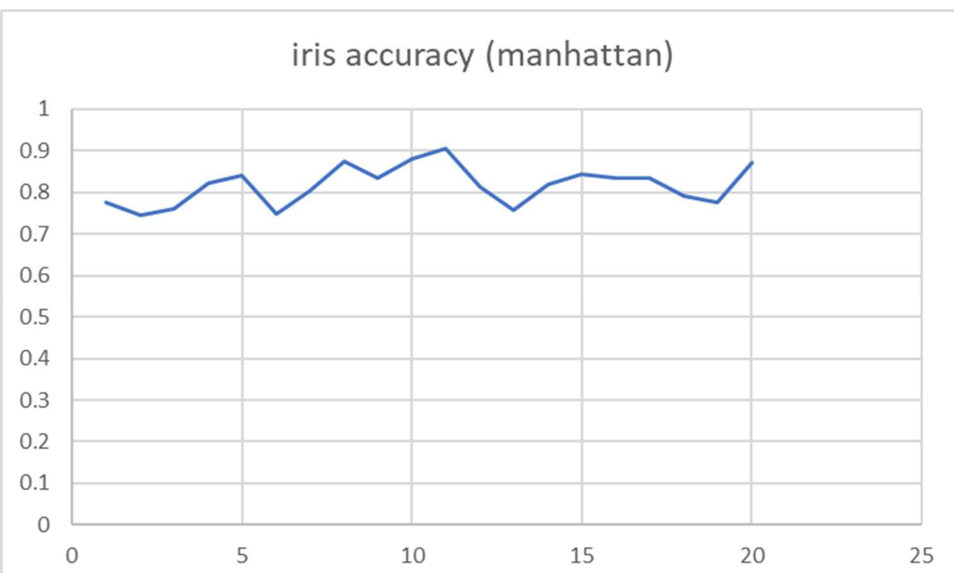
Below you can see graphs that measure the accuracy of my KNN algorithm over the values of k from (0-20). The accuracy number is calculated by running the algorithm 5 times with the same value of k, and taking the average of those accuracy numbers.

KNN accuracy using Euclidean distance on the iris dataset:



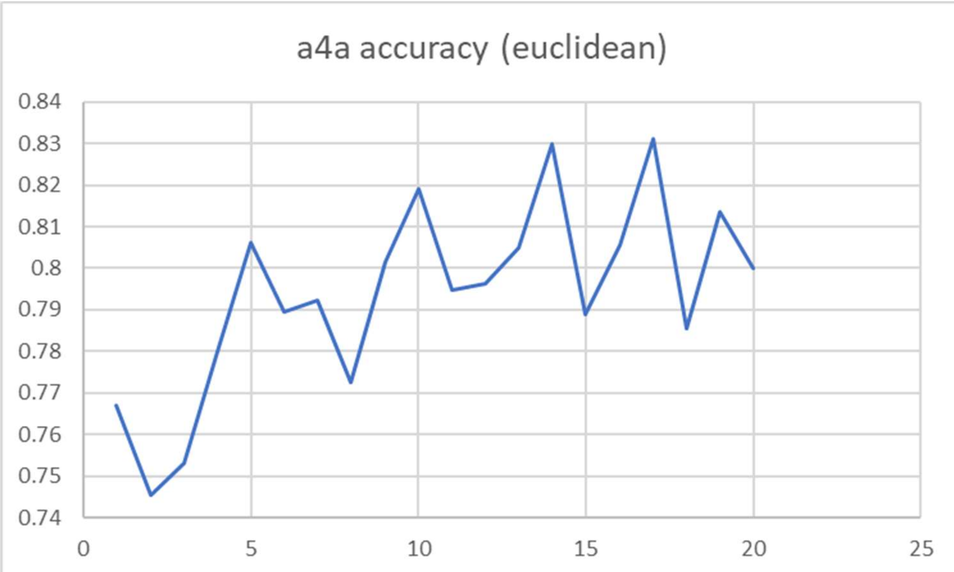
K= 7 resulted in the best average accuracy, which was roughly equal to 98.8%

KNN accuracy using Euclidean distance on the iris dataset:



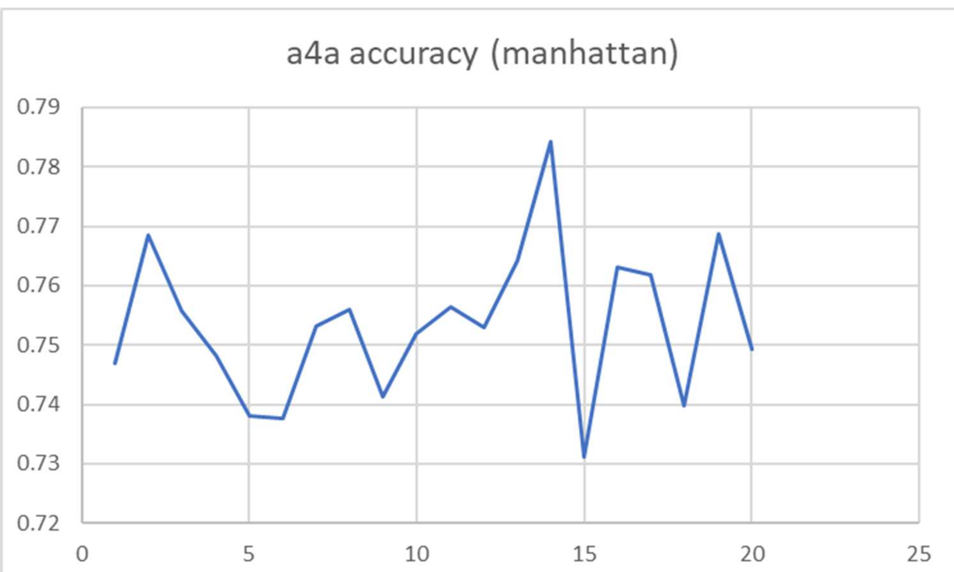
K= 11 resulted in the best average accuracy, which was roughly equal to 90.4%

KNN accuracy using Euclidean distance on a subsample of the a4a dataset:



K= 17 resulted in the best average accuracy, which was roughly equal to 83.1%

KNN accuracy using Manhattan distance on a subsample of the a4a dataset:



K= 14 resulted in the best average accuracy, which was roughly equal to 78.4%

Takeaways from analyzing the data

As I expected before running tests. Using Euclidean distance turns out to be quite a bit more accurate than using Manhattan distance as our distance metric. This makes sense since Euclidean distance is the “true” distance from one point to another.

Perceptron

For my perceptron, I went a similar route as my KNN algorithm. I coded the algorithm and then I created a separate file called “perceptron_tests”. In that file, I formatted the data and ran my algorithm on the data.

Perceptron.py code:

```
import numpy as np


class Perceptron():

    def __init__(self, alpha=0.2, n_iter=100):
        self.alpha = alpha
        self.n_iter = n_iter

    # Store training samples
    # X = training samples, y = training labels
    def fit(self, X, y):
        self.w = np.zeros(X.shape[1])
        self.b = 0
        self.errors_ = []
        for _ in range(self.n_iter):
            errors = 0
            for xi, yi in zip(X, y):
                update = self.alpha * (yi - self.predict(xi))
                self.w += update * xi
                self.b += update
                errors += int(update != 0.0)

            if errors == 0:
                break
            self.errors_.append(errors)
        return self

    # predict functions:
    def predict(self, x):
        return np.where(np.dot(x, self.w) + self.b <= 0.0, -1, 1)

    def predict_iris(self, x):
        result = np.dot(x, self.w) + self.b
        return np.where(result <= 0.0, -1, result)
```

Perceptron_tests.py code:

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from libsvm.svmutil import svm_read_problem
from Perceptron import Perceptron


# -------------------------------------------------------
# function to calculate accuracy of knn
# -------------------------------------------------------
def accuracy(y_true, y_pred):
    accuracy = np.round(y_true * 100 / len(y_pred))
    return accuracy

# -------------------------------------------------------
# Function To Shuffle and Split the Data
# -------------------------------------------------------
def shuffle_split_data(X, y, ratio):
    split = np.random.rand(X.shape[0]) < ratio

    X_Train = X[split]
    y_Train = y[split]
    X_Test = X[~split]
    y_Test = y[~split]

    return X_Train, y_Train, X_Test, y_Test
```

```python
# ---------------------------------------------------
# Reading in data
# ---------------------------------------------------
y_raw, x_raw = svm_read_problem('a4a')

dataset = np.zeros((len(y_raw), 123))

for i in range(len(y_raw)):
    line = x_raw[i]
    for k, v in line.items():
        dataset[i][k - 1] = v

y = np.array(y_raw)

perceptron = Perceptron(0.2, 300)
perceptron.fit(dataset, y)


y_test, x_test = svm_read_problem('a4a.t')
dataset_test = np.zeros((len(y_test), 123))

for i in range(len(y_test)):
    line = x_test[i]
    for k, v in line.items():
        dataset_test[i][k - 1] = v

y_test = np.array(y_test)


predicted = np.zeros(len(y_test))
for i in range(len(y_test)):
    predicted[i] = perceptron.predict(dataset_test[i])
```

```python
# ---------------------------------------------------------------
# Calculating and printing out results
# ---------------------------------------------------------------
print('\n')
print('======= a4a =======')
print('Total predicted: ', len(predicted), sep='')
correct_num = np.sum(predicted == y_test)
print('Total correct: ', correct_num, sep='')
print('Accuracy = ', accuracy(correct_num, predicted), '%', sep='')

y_raw, x_raw = svm_read_problem('iris.scale')

y = np.array(y_raw)
x = np.zeros((len(y_raw), 4))
for i in range(len(y_raw)):
    line = x_raw[i]
    for k, v in line.items():
        x[i][k - 1] = v

x_train, y_train, x_test, y_test = shuffle_split_data(x, y, 0.3)

y_train1 = np.copy(y_train)
y_train1[y_train1 != 1] = -1
y_train1[y_train1 != -1] = 1

y_train2 = np.copy(y_train)
y_train2[y_train2 != 2] = -1
y_train2[y_train2 != -1] = 1

y_train3 = np.copy(y_train)
y_train3[y_train3 != 3] = -1
y_train3[y_train3 != -1] = 1

perceptron_iris1 = Perceptron(0.2, 1000)
perceptron_iris1.fit(x_train, y_train1)
perceptron_iris2 = Perceptron(0.2, 1000)
perceptron_iris2.fit(x_train, y_train2)
perceptron_iris3 = Perceptron(0.2, 1000)
perceptron_iris3.fit(x_train, y_train3)
```

```python
predicted = np.zeros(len(y_test))
for i in range(len(y_test)):
    pred = np.zeros(3)
    pred[0] = perceptron_iris1.predict(x_test[i])
    pred[1] = perceptron_iris2.predict(x_test[i])
    pred[2] = perceptron_iris3.predict(x_test[i])
    predicted[i] = np.argmax(pred) + 1

print('\n')
print('======= iris =======')
print('Total predicted: ', len(predicted), sep='')
correct_num = np.sum(predicted == y_test)
print('Total correct: ', correct_num, sep='')
print('Accuracy = ', accuracy(correct_num, predicted), '%', sep='')
```

```python
# ---------------------------------------------------------
#   Creating Graphs
# ---------------------------------------------------------
weights = perceptron.w
top_index = np.argsort(-np.abs(weights))[:20]
plt.figure()
plt.bar(list(map(str, top_index)), weights[top_index])
plt.xlabel('Feature Number')
plt.ylabel('Weight')
plt.title('Top 20 Weights in a4a Dataset')
plt.show(block=False)

weights = perceptron_iris1.w
top_index = np.argsort(-np.abs(weights))
plt.figure()
plt.bar(list(map(str, top_index)), weights[top_index])
plt.xlabel('Feature Number')
plt.ylabel('Weight')
plt.title('Weights in iris Dataset (Class 1)')
plt.show(block=False)

weights = perceptron_iris2.w
top_index = np.argsort(-np.abs(weights))
plt.figure()
plt.bar(list(map(str, top_index)), weights[top_index])
plt.xlabel('Feature Number')
plt.ylabel('Weight')
plt.title('Weights in a4a Dataset (Class 2)')
plt.show(block=False)

weights = perceptron_iris3.w
top_index = np.argsort(-np.abs(weights))
plt.figure()
plt.bar(list(map(str, top_index)), weights[top_index])
plt.xlabel('Feature Number')
plt.ylabel('Weight')
plt.title('Weights in a4a Dataset (Class 3)')
plt.show()
```

Accuracy of the Perceptron algorithm

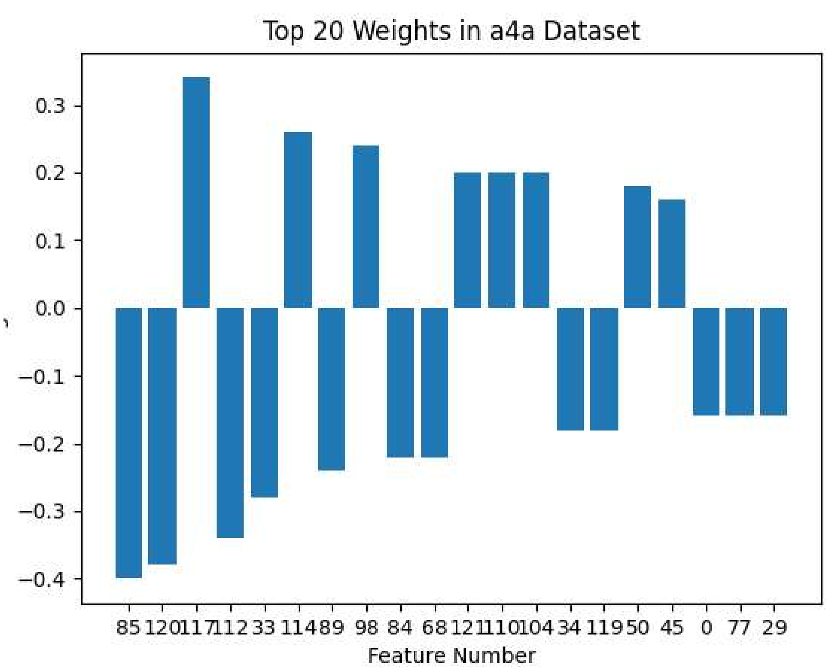
While playing around with the data I've found that changing the learning rate is what had the biggest impact on the accuracy of my predictions. A learning rate of 0.01 worked very well on the iris dataset, this gave me an accuracy of 78%. The same goes for the a4a dataset where I had 82% accuracy when it came to my predictions. I struggled a lot with the iris dataset and I think it was because my training set was on the small side, I could not increase it by much since the entire dataset is quite small.

Highest accuracy (terminal screenshot):

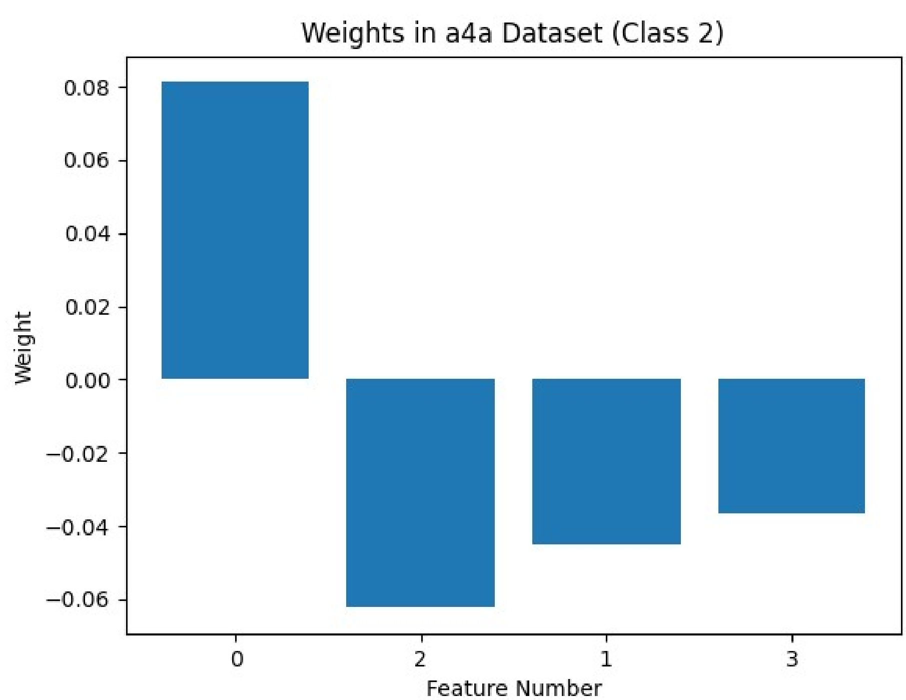
```
======= a4a =======
Total predicted: 27780
Total correct: 22792
Accuracy = 82.0%

======= iris =======
Total predicted: 23
Total correct: 18
Accuracy = 78.0%
```

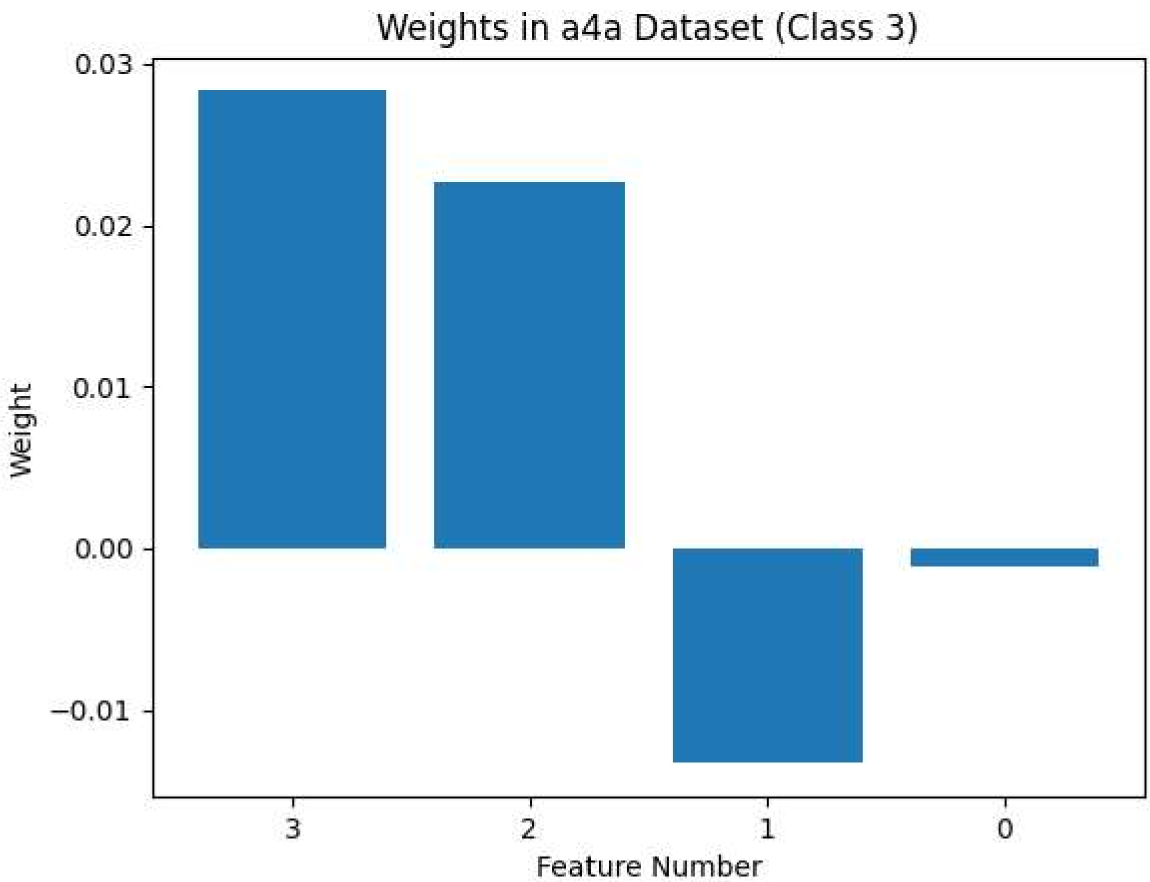
Weights of the perceptron algorithm

Below is a picture of the top 20 attributes in the a4a dataset. Note: While about 50% of the weights are negative numbers, we look at the absolute value of the number and not the general value.



Below are some of the other graphs I've created, that are not required for our project specifications.




Weights in a4a Dataset (Class 3)
Weight
0.03
0.02
0.01
0.00
−0.01
3
2
1
0
Feature Number