CS 491/691 Project 2

Elaine Chu and Jared Manning

Fall 2019

Note: The following functions are implemented with the numpy, random, and statistics libraries.

1 Nearest Neighbors

1.1 KNN_test

Takes in training data sample features X_train, training data sample labels Y_train, test data sample features X_test, test data labels Y_test, and number of neighbors to test on, K and returns the accuracy of the classifier on the test data.

1. Initial steps
   (a) First, we add a fail-safe to return the accuracy as 0 if K is defined as 0. We also store the total number of samples and initialize the correct counter to 0.

```
def KNN_test(X_train, Y_train, X_test, Y_test, K):
    if K == 0:
        return 0
    total = len(X_test)
    correct = 0
```

2. Loop through all the samples in test data
   (a) The next step starts the bulk of the function, and begins with a for loop that iterates through all of the test data. For each test sample, empty lists are initiated to be filled later on in the algorithm. The maximum distance of the test data from a nearest neighbor, max_dist, is set to 0, and the index of the furthest nearest neighbor, max_idx, is set to NULL.

```
for sample_idx, sample in enumerate(X_test):
    dists = []
    dist_idxs = []
    max_dist = 0
    max_idx = None
```

   (b) Loop through all the samples in training data and find the neighbor (K) closest to the test sample
      i. At this step, iterate through all of the samples in training data, setting the distance, dist, to 0.

```
for idx, trainData in enumerate(X_train):
        dist = 0
```

ii. Next, calculate the Euclidean distance between each training feature to test feature and sum them all up. The square root is left out of the calculation, because it is not necessary for comparing distances in this instance.

```
for feat_idx in range(0,len(trainData)):
    dist += (sample[feat_idx] - trainData[feat_idx])
        * (sample[feat_idx] - trainData[feat_idx])
```

A. Calculated distance is then stored as dist for the idx-th sample in the training data.

iii. If the list of distances, dists, is less than the defined K, for number of nearest neighbors, add the distance of the current training sample, dist, to dists, and add the current traning sample's index, idx, to the list of distance indices, dist_idx.

```
if len(dists) < K:
    dists += [dist]
    dist_idxs += [idx]
    if dist > max_dist:
        max_dist = dist
        max_idx = idx
```

A. If the distance between the current training and test sample is greater than value of max_dist, set max_dist to dist, and set max_idx as idx. This defines the current distance between the training sample and test sample as the greatest distance between any of the current nearest neighbor training samples and the current test sample.

iv. If dist is less than max_dist, then replace max_dist from the list of distances and list of distance indices for the current test sample with dist. Reset max_dist as the largest value in the list dists and the corresponding index in dist_idxs.

```
elif dist < max_dist:
    dists.remove(max_dist)
    dist_idxs.remove(max_idx)
    dists += [dist]
    dist_idxs += [idx]
    max_dist = max(dists)
    max_idx = dist_idxs[dists.index(max_dist)]
```

3. Iterate through the K nearest neighbors and calculate the majority vote

(a) First, define the number of positive and negative votes as 0. Define the decision as NULL.

```
pos_votes = 0
neg_votes = 0
decision = None
```

(b) Next, iterate through the K nearest neighbors and calculate the majority vote to determine whether the decision is positive or negative.

```
for i in dist_idxs:
    if Y_train[i][0] == 1:
        pos_votes+=1
    else:
        neg_votes+=1
    if pos_votes > neg_votes:
    decision = 1
    else:
    decision = -1
```

i. If the number of positive votes is greater than the number of negative votes, then decision is positive, whereas the opposite returns decision as negative.

(c) If the decision matches the test label from Y_test, then add 1 to the correct counter.

```
if decision == Y_test[sample_idx][0]:
        correct+=1
```

4. Calculate the accuracy of K nearest neighbors

(a) Divide the number ofcorrect classifications by the total number of test samples. Return the accuracy.

```
return correct/total
```

1.2 Write-Up

Using the following training data, how would your algorithm classify the test points listed below with K=1, K=3, and K=5?

X1	X2	Label	K=1	K=3	K=5
1	1	1	-1	-1	1
2	1	-1	1	1	-1
0	10	1	1	1	-1
10	10	-1	-1	1	-1
5	5	1	-1	-1	-1
3	10	-1	1	-1	-1
9	4	1	1	1	1
6	2	-1	1	1	1
2	2	1	-1	-1	1
8	7	-1	1	1	1

1.3 choose_K

This function takes in training and validation data, and returns the number of K nearest neighbors that produces the highest classification accuracy for the validation data.

1. Initial steps
   (a) Set values best_acc and best_K as 0.

```
best_acc = 0
best_K = 0
```

2. Loop through all possible values of K
   (a) This block of code loops through all odd values of K and runs the previously implemented KNN_test with training and validation data as inputs, storing the accuracy as acc. Next, evaluate if accuracy is greater than best_acc. If TRUE, replace best accuracy with the current accuracy, and replace best_K with the current K value. Once finished looping, return best_K.

```
for i in range(1,len(X_train)+1):
    #if its even skip (K should be odd)
    if i%2 == 0:
        continue
    acc = KNN_test(X_train, Y_train, X_val, Y_val, i)
    if acc > best_acc:
        best_acc = acc
        best_K = i
return best_K
```

   i. All possible values of K range from 1 to the number of samples there are in the training data. K=1 results in overfitting and K=N results in underfitting, which are the most extreme cases.
   ii. Skip even values of K to avoid the possibility of a tie.

1.4 Write-Up

What is the best K value for the training data above? Answer: K=9, Accuracy = 0.7

2 Clustering

2.1 K_Means

This function takes feature vectors X and a K value as input and returns a numpy array of cluster centers C.

1. Initial steps

   (a) This initial step stores the number of samples (num_samps) as the number of rows in X. It also initializes random cluster centers by randomly choosing K index numbers of samples from X.

```
num_samps = X.shape[0]

cluster_index = random.sample(range(num_samps),K)
newlabels = []
clusters = []
old_clusters = [0] * K

for a in range(K):
    clusters.append(list(X[cluster_index[a]]))
```

      i. This step also initializes newlabels and clusters as empty lists, and old_clusters as a list K elements long, filled with 0.

      ii. After the empty clusters list is initialized, a for loop fills it with the randomly selected cluster centers.

2. Update cluster centers

   (a) This next block represents the bulk of the K_Means code within a for loop. First, update cluster centers **many** times, until the they stop changing between loops.

```
for i in range(999):
    if old_clusters == clusters:
        break
```

   (b) If clusters change between loops, replace the value of old_cluster with the last cluster centers and reinitialize clusters as an empty list. Begin a loop that iterates through all samples in X.

```
for b in range(num_samps):  # loop through samples
    samp = X[b]  # select sample
    centroid_dist = []
```

      i. Here, initiate samp as the current sample. Also initiate an empty list, centroid_dist.

   (c) Next, loop through centroids to calculate Euclidean distances (except for square-root) from cluster centers to samp. Store the distances in centroid_dist.

```
for c in range(K):  # loop through centroids
    centroid = old_clusters[c]
    samp_dist = np.square(samp - centroid)
    summed_dist = np.sum(samp_dist)
    centroid_dist.append(summed_dist)

best_clust = centroid_dist.index(min(centroid_dist))
newlabels.append(best_clust)
```

i. Outside of the loop, find the smallest distance, and declare that cluster center as the closest to samp. Store the cluster center index as the cluster label for the current sample.

3. Calculate new cluster centers

(a) This next step takes the newlabels of each sample in X and calculates new cluster centers given the samples closest to each old cluster center.

```
for d in range(K):
    index_list = np.where(np.asarray(newlabels)==d)[0]
        clusters.append(old_clusters[d])
    if len(index_list) == 0:
        clusters.append(old_clusters[d])
        clusters = sorted(clusters)
    else:
        new_mean = np.nanmean(X[index_list,], axis=0)
        clusters.append(tuple(new_mean))
        clusters = sorted(clusters)
newlabels = []
```

i. The **if statement** considers whether any of the new cluster centers have returned as 'nan'. If so, old clusters centers replace the 'nan' to prevent warnings.

ii. New cluster centers are calculated as the mean of the samples closest to the old cluster centers. The list of cluster centers, clusters, is sorted to allow for uniform comparison between loops.

iii. Outside of the for loop, newlabels is reinitialized as an empty list.

4. Return best cluster centers

(a) This step outside of all for loops returns the best cluster centers, only computing after cluster centers stop changing within the overarching for loop from **Step 2**.

```
C = clusters
C = np.array(C)
return C
```

i. This step also converts the output into a numpy array, as asked.

2.2 Write-Up

Test your function on the following training data, with K=2 and K=3. Plot the clusters in different colors and label the cluster centers.

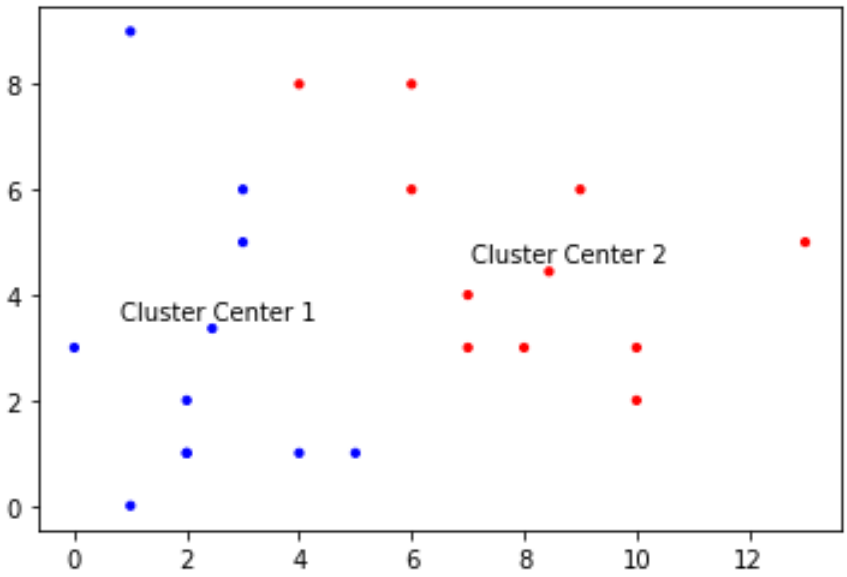

Figure 1. The above figure represents the classifications of training data using K_Means, given K=2.

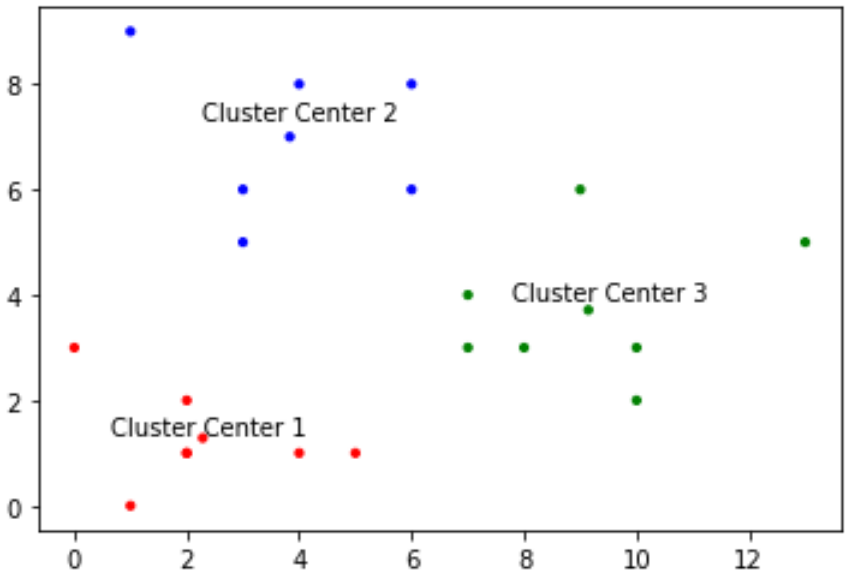

Figure 2. The above figure represents the classifications of training data using K_Means, given K=3.

Note: These plots are representative of just one of the outputs given by K_Means. Because the initial clusters vary each time, resulting cluster centers can vary.

2.3 K_Means_better

This function takes feature vectors X and a K value as input and returns a numpy array of cluster centers, C.

1. Initial Steps
   (a) First, initiate an empty list of clusters, cluster_list

```
cluster_list = []
```

2. Calculate cluster centers
   (a) Here, calculate cluster centers **many** times, append cluster centers to cluster_list.

```
for i in range(1000):
    C = K_Means(X,K)
    C = [tuple(s) for s in C]
    cluster_list.append(C)
```

3. Return the majority rules cluster
   (a) Here, convert individual entries of cluster_list to tuples to allow for comparison of cluster centers. Find the most common cluster center, and store as mode_cluster. Convert mode_cluster to a numpy array as required. Return mode_cluster.

```
final_list = [tuple(t) for t in cluster_list]

mode_cluster = stat.mode(final_list)
mode_cluster = np.array(mode_cluster)

return mode_cluster
```

2.4 Write-Up

Test your function on the following training data, with K=2 and K=3. Plot the clusters in different colors and label the cluster centers.

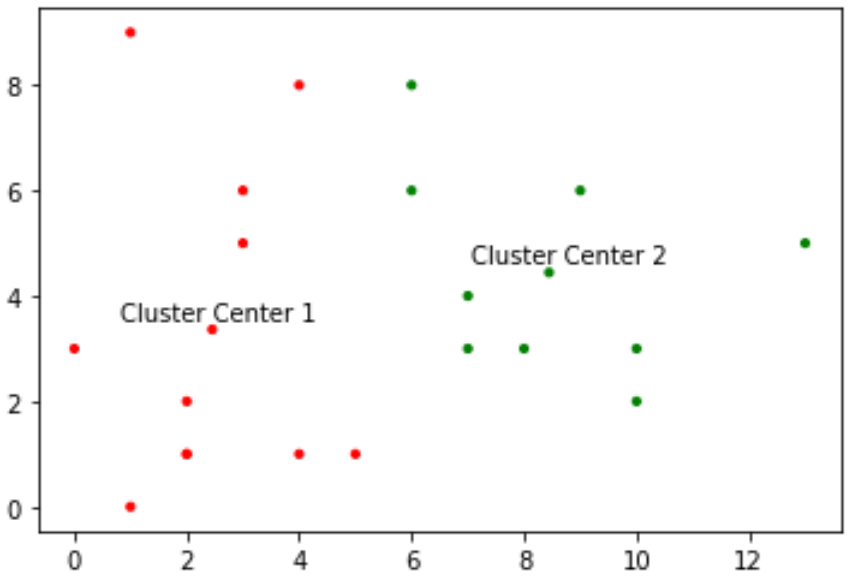

Figure 3. The above figure represents the classifications of training data using K_Means_better, given K=2.

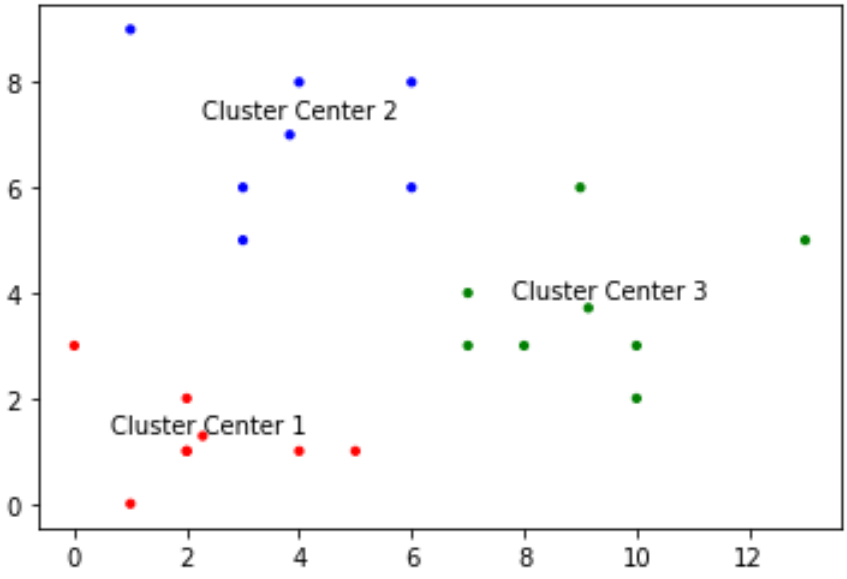

Figure 4. The above figure represents the classifications of training data using K_Means_better, given K=3.

3 Perceptron

3.1 perceptron_train

This function takes training data features X and labels Y, and returns [w,b], a list containing a list of weights, w and the bias, b.

1. Initial steps

   (a) First, initialize weights, w, as a vector of zeros and bias, b, as 0. Set variable updated as TRUE.

```
w = [0] * len(X[0])
b = 0
updated = True
```

   (a) Update weights, w, and bias, b

      i. Next, run a for loop until all samples are correctly classified without updating.

```
while updated:
    updated = False

    for sample_idx, x in enumerate(X):
        a = b
        for feat_idx, feat in enumerate(x):
                a += w[feat_idx] * feat

        if a * Y[sample_idx][0] <= 0:
                updated = True
                for feat_idx, feat in enumerate(x):
                        w[feat_idx] += feat * Y[sample_idx][0]
                b += Y[sample_idx][0]
    return [w,b]
```

      i. To do so, the code continues to run as long as updated = TRUE.

      ii. Variable a is calculated for each sample. If a and label Y are not the same sign, then the weights and bias are updated.

3.2 perceptron_test

This function takes test data features X_test and labels Y_test, weight vector w, and bias, b, and returns the accuracy of the perceptron on the test data.

1. Initial steps
   (a) First, set total as the number of samples in X_test, and set the counter, correct, as 0.

```
total = len(X_test)
correct = 0
```

2. Calculate a for each sample
   (a) Next, this bulk of code calculates a for each sample using the passed in weights, w, and bias, b. If a is greater than 1, the sample decision as a 1, otherwise as a -1.

```
for idx, x in enumerate(X_test):
    decision = None
    a = b
    for feat_idx,feat in enumerate(x):
        a += w[feat_idx] * feat

    if a > 0:
        decision = 1
    else:
        decision = -1
```

   (a) Calculate accuracy of perceptron
      i. If the perceptron classification, decision, is the same as the sample label, Y_test, then add 1 to the correct counter. Return the accuracy at the end, which is correct / total.

```
        if decision == Y_test[idx][0]:
                correct+=1
return correct/total
```

3.3 Write-Up

Train your perceptron on the following dataset. Using the w and b you get, plot the decision boundary.

w = [2.0, 4.0] and b = 2
decision boundary = -0.5x – 0.5
Accuracy = 1.0

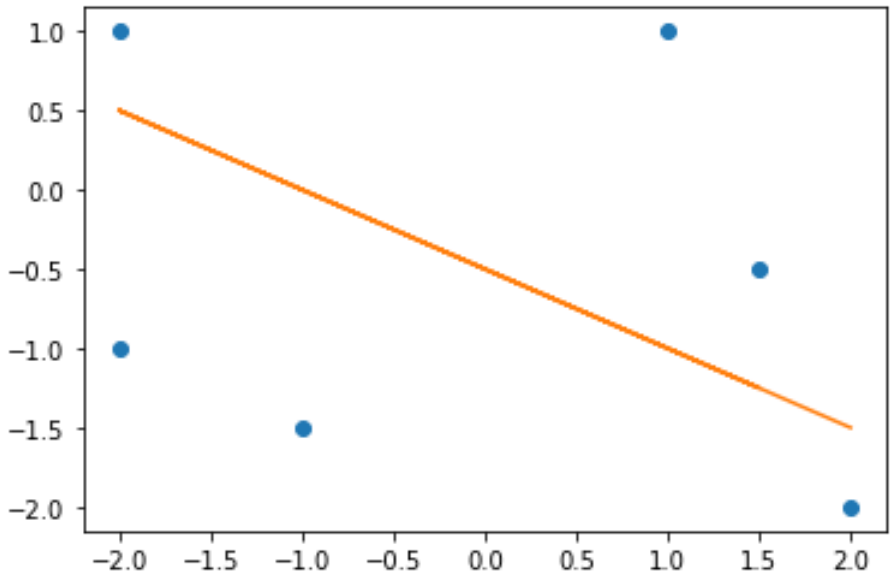

Figure 5. The above figure represents the decision boundary using w and b from perceptron_train.