HW5 Report

Team members:

1. **Worapol Setwipatanachai**
2. **Worawich Chaiyakunapruk**

1. Page 1 (40 pts) Experiment table

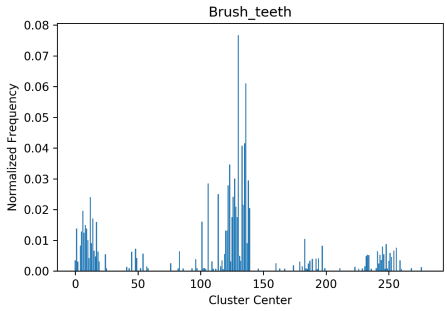
Random Forest classifier has 200 trees and 800 depth.

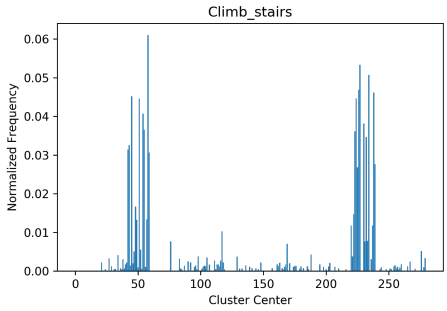
Trial	Fixed Sample Length	Overlap%	Hierarchical k-means (Layer 1, Layer 2)	Classifier Accuracy (Random Forest)
1	32	50	10, 14	68%
2	32	50	14, 10	67%
3	32	50	14, 20	72%
4	64	50	14, 10	69%
5	64	50	14, 20	69%

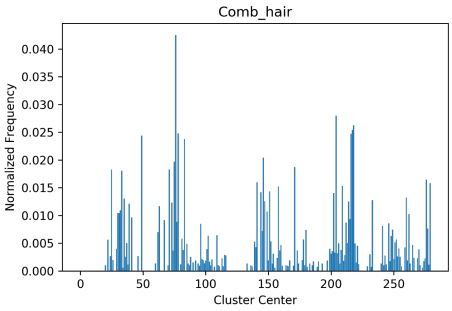
2. Page 2 (28 pts) Histograms

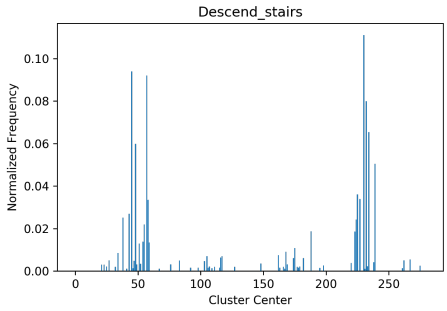
Histograms of the mean quantized vector for each activity with the K value that gives you the highest accuracy.

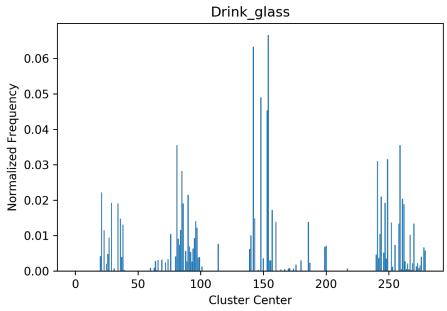
Hierarchical k-means: 1st layer - **14**, 2nd layer - **20**

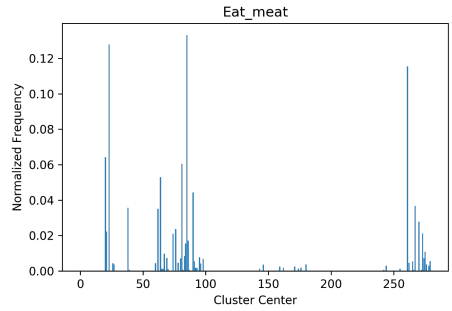


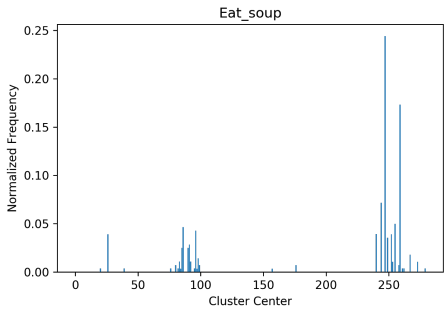


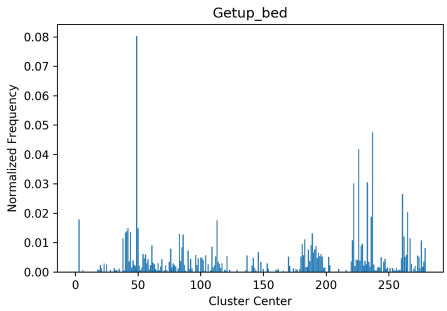


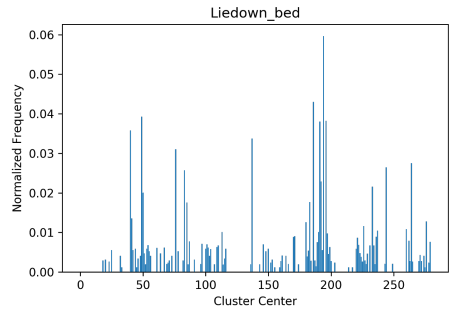


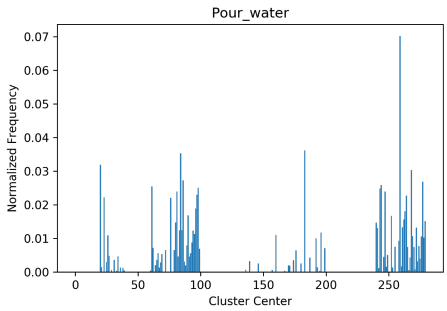


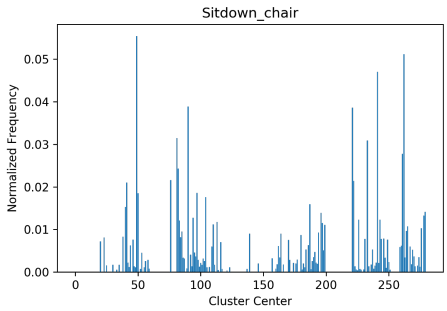


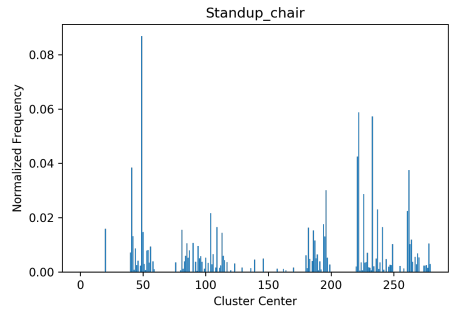


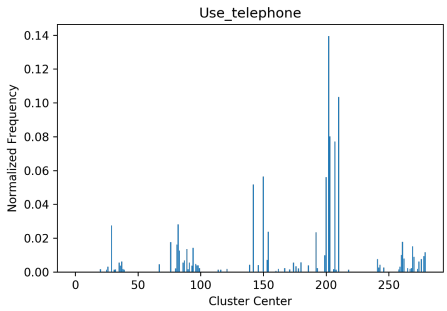


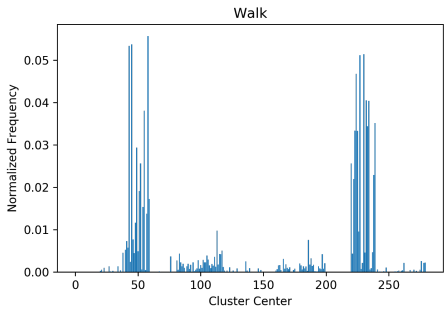












3. Page 3 (22 pts) Confusion matrix

Highest accuracy was achieved at Trial 3 conditions.

Random Forest: 200 Trees, 800 Depth

Test 3		Predict														
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	Accuracy
Actual	1	2	0	1	0	0	0	0	0	0	0	1	0	0	0	50%
	2	0	28	0	4	0	0	0	0	0	0	0	0	0	2	82%
	3	0	0	10	0	0	0	0	0	0	0	0	0	0	0	100%
	4	0	2	0	12	0	0	0	0	0	0	0	0	0	0	86%
	5	0	0	0	0	31	0	0	0	0	2	0	0	0	0	94%
	6	0	0	0	0	0	2	0	0	0	0	0	0	0	0	100%
	7	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0%
	8	0	0	0	0	6	0	0	21	0	0	3	4	0	0	62%
	9	0	0	1	0	0	0	0	2	0	1	5	0	0	0	0%
	10	0	0	0	0	0	0	0	0	0	26	7	0	0	0	79%
	11	0	0	0	0	0	0	0	0	0	0	28	5	0	0	85%
	12	0	1	0	1	0	0	0	1	0	0	7	24	0	0	71%
	13	0	0	0	0	1	0	0	0	0	1	0	0	2	0	50%
	14	0	9	0	0	0	0	0	0	0	0	0	2	0	22	67%

1	2	3	4	5	6	7
Brush_teeth	Climb_stairs	Comb_hair	Descend_Stairs	Drink_glass	Eat_meat	Eat_soup

8	9	10	11	12	13	14
Getup_bed	Liedown_bed	Pour_water	Sitdown_chair	Standup_chair	Use_telephone	Walk

4. Page 4 (10 pts) A screenshot of your code

i) Segmentation of the vector

```
# Chop up data into chunks with specified overlap and size

def prepDataKM(data, siz=32, overlap=0):

#     Iterate over all activities

    for i in range(len(data)):

#         Iterate over all files

        for j in range(len(data[i])):

#         Ravel file and turn it into [1,rav.size] numpy array

            rav = np.asarray([np.ravel(data[i][j])])

#             If size of ravelled data is not divisible by size of chunks,
#             collect the leftover data by taking the last 32 rows of data from the back

            if rav.size%(siz*3) != 0:
                ind = math.floor((rav.size-siz*3*overlap)/((1-overlap)*3*siz))
                t = np.asarray([np.asarray(rav[0,int(i*siz*3*(1-overlap)):int(siz*3*(1+i*(1-overlap)))]) for i in range(ind)])
                t = np.concatenate((t,[rav[0,-siz*3:]]))
                if i==0 and j==0:
                    chunks = t.copy()
                else:
                    chunks = np.concatenate((chunks,t))
            else:
                ind = math.floor((rav.size-siz*3*overlap)/((1-overlap)*3*siz))
                t = np.asarray([np.asarray(rav[0,int(i*siz*3*(1-overlap)):int(siz*3*(1+i*(1-overlap)))]) for i in range(ind)])
                if i==0 and j==0:
                    chunks = t.copy()
                else:
                    chunks = np.concatenate((chunks,t))
    return chunks
```

ii) K-means

```
# Returns k-means centers based on input data and number of clusters

def kmeansCluster0(chunks, clusters0):
    kmeans = MiniBatchKMeans(n_clusters = clusters0).fit(chunks)
    centers0 = kmeans.cluster_centers_
    return centers0

# Returns k-means centers AND cluster indices based on input data and number of clusters

def kmeansCluster1(chunks, clusters1):
    kmeans = MiniBatchKMeans(n_clusters = clusters1).fit(chunks)
    ind1 = kmeans.labels_
    return ind1

# Returns cluster centers after 2nd layer k-means.
# kmeansCluster2 takes in the original data set, center indices from 1st layer, and # of clusters in 2nd layer
# Returns ndarray of all centers

def kmeansCluster2(chunks, ind1, clusters2):
    centers2 = np.asarray([kmeansCluster0(chunks[ind1==i], clusters2) for i in range(np.amax(ind1)+1)])
    centers2 = np.ravel(centers2).reshape(int(centers2.size/96),96)
    return centers2

# Hierarchical k-means
def hkmeans(chunks, lay1, lay2):
    ind1 = kmeansCluster1(chunks, clusters1 = lay1)
    centers2 = kmeansCluster2(chunks, ind1, clusters2 = lay2)
    return centers2
```

iii) Generating the histogram & Classifier

```
# Use Random Forest to obtain VQ for each file
# Returns normalized ndarray for histogram
# ccenters -> (x,y) & act -> (z,y) i.e. they have the same # of features

def randomForestVQ(ccenters, labels, act, tree = 100, depth=300, histogram = True):
#     clf = KNeighborsClassifier(n_neighbors = tree, n_jobs = -1)
    clf = RandomForestClassifier(n_estimators = tree, max_depth = depth, n_jobs=-1)
    clf.fit(ccenters, labels)
    pre = clf.predict(act)
    if histogram:
        VQ = np.histogram(pre, bins = range(np.amax(labels)+2), density = False)
        n_VQ = VQ[0]/np.sum(VQ[0])
        return n_VQ
    else:
        return pre

# Generate mean histograms
def meanHistogram(data):
    meanHist = []
    for i in range(len(data)):
        meanH = np.asarray(data[i])
        meanH = np.sum(meanH, axis=0)/meanH.shape[0]
        meanHist.append(meanH)
    return meanHist
```

5. Page 5+ Screenshots of all your source code.

```
import numpy as np
from sklearn.cluster import MiniBatchKMeans
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
import os
import math
import time

def getData(act_dir, filenames):
    data = [np.genfromtxt(act_dir + '/' + filenames[i]) for i in range(len(filenames))]
    return data

# Chop up data into chunks with specified overlap and size

def prepDataKM(data, siz=32, overlap=0):

#     Iterate over all activities

    for i in range(len(data)):

#         Iterate over all files

        for j in range(len(data[i])):

#         Ravel file and turn it into [1,rav.size] numpy array

            rav = np.asarray([np.ravel(data[i][j])])

#             If size of ravelled data is not divisible by size of chunks,
#             collect the leftover data by taking the last 32 rows of data from the back

            if rav.size%(siz*3) != 0:
                ind = math.floor((rav.size-siz*3*overlap)/((1-overlap)*3*siz))
                t =
np.asarray([np.asarray(rav[0,int(i*siz*3*(1-overlap)):int(siz*3*(1+i*(1-overlap)))]) for i in
range(ind)])
                t = np.concatenate((t,[rav[0,-siz*3:]]))
                if i==0 and j==0:
                    chunks = t.copy()
                else:
                    chunks = np.concatenate((chunks,t))
            else:
                ind = math.floor((rav.size-siz*3*overlap)/((1-overlap)*3*siz))
                t =
np.asarray([np.asarray(rav[0,int(i*siz*3*(1-overlap)):int(siz*3*(1+i*(1-overlap)))]) for i in
range(ind)])
```

```
                if i==0 and j==0:
                    chunks = t.copy()
                else:
                    chunks = np.concatenate((chunks,t))
    return chunks

# Returns k-means centers based on input data and number of clusters

def kmeansCluster0(chunks, clusters0):
    kmeans = MiniBatchKMeans(n_clusters = clusters0).fit(chunks)
    centers0 = kmeans.cluster_centers_
    return centers0

# Returns k-means centers AND cluster indices based on input data and number of clusters

def kmeansCluster1(chunks, clusters1):
    kmeans = MiniBatchKMeans(n_clusters = clusters1).fit(chunks)
    ind1 = kmeans.labels_
    return ind1

# Returns cluster centers after 2nd layer k-means.
# kmeansCluster2 takes in the original data set, center indices from 1st layer, and # of
clusters in 2nd layer
# Returns ndarray of all centers

def kmeansCluster2(chunks, ind1, clusters2):
    centers2 = np.asarray([kmeansCluster0(chunks[ind1==i], clusters2) for i in
range(np.amax(ind1)+1)])
    centers2 = np.ravel(centers2).reshape(int(centers2.size/96),96)
    return centers2

# Hierarchical k-means
def hkmeans(chunks, lay1, lay2):
    ind1 = kmeansCluster1(chunks, clusters1 = lay1)
    centers2 = kmeansCluster2(chunks, ind1, clusters2 = lay2)
    return centers2

# Returns a list (activity) of ndarray (chopped file)

def prepDataTestTrain(activity, siz=32, overlap=0):
#     Iterate over all .txt files
    chunks_list = []

    for j in range(len(activity)):

#         Ravel file and turn it into [1,rav.size] numpy array

        rav = np.asarray([np.ravel(activity[j])])
```

```
#             If size of ravelled data is not divisible by size of chunks,
#             collect the leftover data by taking the last 32 rows of data from the back

        if rav.size%(siz*3) != 0:
            ind = math.floor(int((rav.size-siz*3*overlap)/((1-overlap)*3*siz)))
            t =
np.asarray([np.asarray(rav[0,int(i*siz*3*(1-overlap)):int(siz*3*(1+i*(1-overlap)))]) for i in
range(ind)])
            t = np.concatenate((t,[rav[0,-siz*3:]]))
            chunks_list.append(t)
        else:
            ind = math.floor(int((rav.size-siz*3*overlap)/((1-overlap)*3*siz)))
            t =
np.asarray([np.asarray(rav[0,int(i*siz*3*(1-overlap)):int(siz*3*(1+i*(1-overlap)))]) for i in
range(ind)])
            chunks_list.append(t)
    return chunks_list

# Combine two training fractions and transform each activity into ndarray with specified size
and overlap
# Returns training_list[i][j] accesses activity i file j

def prepTraining(frac1, frac2, siz = 32, overlap = 0):
    training_list = [prepDataTestTrain(frac1[i]+frac2[i], siz, overlap) for i in
range(len(frac1))]
    return training_list

# Transform each activity into ndarray with specified size and overlap
# Returns test_list[i][j] accesses activity i file j

def prepTest(frac1, siz = 32, overlap = 0):
    test_list = [prepDataTestTrain(frac1[i], siz, overlap) for i in range(len(frac1))]
    return test_list

# Use Random Forest to obtain VQ for each file
# Returns normalized ndarray for histogram
# ccenters -> (x,y) & act -> (z,y) i.e. they have the same # of features

def randomForestVQ(ccenters, labels, act, tree = 100, depth=300, histogram = True):
#     clf = KNeighborsClassifier(n_neighbors = tree, n_jobs = -1)
    clf = RandomForestClassifier(n_estimators = tree, max_depth = depth, n_jobs=-1)
    clf.fit(ccenters, labels)
    pre = clf.predict(act)
    if histogram:
        VQ = np.histogram(pre, bins = range(np.amax(labels)+2), density = False)
        n_VQ = VQ[0]/np.sum(VQ[0])
        return n_VQ
    else:
        return pre
```

```
# Generate mean histograms
def meanHistogram(data):
    meanHist = []
    for i in range(len(data)):
        meanH = np.asarray(data[i])
        meanH = np.sum(meanH, axis=0)/meanH.shape[0]
        meanHist.append(meanH)
    return meanHist

# Convert from data[j][i] to ndarray in which each row represents a file

def flattenFolders(data):
    flat_labels = []
    for sub in data:
        for items in sub:
            flat_labels.append(items)
    return np.asarray(flat_labels)

# Generate labels from data[j][i] for Random Forest classifier

def generateLabels(trainHist):
    flat_labels = []
    labels = [np.full((len(trainHist[i])), i) for i in range(len(trainHist))]
    for sub in labels:
        for items in sub:
            flat_labels.append(items)
    return np.asarray(flat_labels)



def exportHistograms(Hist, act_names):
    for i in range(14):
        fig, ax = plt.subplots()
        plt.bar(range(len(meanHistogram(Hist)[i])),meanHistogram(Hist)[i])
        plt.ylabel('Normalized Frequency')
        plt.xlabel('Cluster Center')
        plt.title(act_names[i])
        plt.savefig(act_names[i]+'.png', dpi = 300, format='png', transparent=True)
        plt.show()

def exportConfusionMatrix(confusion):
    np.savetxt('Confusion Matrix.txt', confusion, fmt='%4d', delimiter=',', newline='\n')

data_cwd = os.getcwd() + '/HMP_Dataset'
act_names = os.listdir(path=data_cwd)

act_dir = [data_cwd + '/' + act_names[i] for i in range(len(act_names))]
```

```
filenames = [os.listdir(path=act_dir[i]) for i in range(len(act_dir))]

# data contains all data from the folder HMP_Dataset
# len(data) = 14 objects
# data[i][j] accesses file j of activity i

data = [getData(act_dir[i], filenames[i]) for i in range(len(act_dir))]
labels = act_names

# Each fraction has 1/3 file of each category in it
# frac1[i][j] accesses file j of activity i

frac1 = [data[i][:round(len(data[i])/3)] for i in range(len(data))]
frac2 = [data[i][round(len(data[i])/3):2*round(len(data[i])/3)] for i in range(len(data))]
frac3 = [data[i][2*round(len(data[i])/3):] for i in range(len(data))]

# Creating chunks of data
chunks = prepDataKM(data, 32, 0.5)

# Hierarchical k-means
centers2 = hkmeans(chunks, lay1 = 10, lay2 = 14)

def predict(train1, train2, test, siz=32, overlap=0.5, tree=200, dep=800):

    start_time = time.time()
    print(siz,'_14,20_',tree,',',dep)
#     Preparing training and testing dataset
    training1 = prepTraining(train1, train2, siz, overlap)
    test1 = prepTest(test, siz, overlap)
    print("Finished Data Preparations...")

#     Get training and testing histograms
    labCen2 = range(centers2.shape[0])
    TrainHist1 = [[randomForestVQ(centers2, labCen2, training1[j][i], tree, dep, True) for i
in range(len(training1[j]))] for j in range(len(training1))]
    print("Finished Generating Training Histograms...")

    TestHist1 = [[randomForestVQ(centers2, labCen2, test1[j][i], tree, dep, True) for i in
range(len(test1[j]))] for j in range(len(test1))]
    print("Finished Generating Testing Histograms...")

#     Compare with training histogram
    lab = generateLabels(TrainHist1)
    trainRF = flattenFolders(TrainHist1)
    pred = [[np.asarray([randomForestVQ(trainRF, lab, [TestHist1[j][i]], tree, dep, True)])
for i in range(len(TestHist1[j]))] for j in range(len(TestHist1))]
    print("Finished Predicting...")

#     Generating a confusion matrix
```

```
    a = [np.ravel(pred[i]).reshape(len(pred[i]),14) for i in range(len(pred))]
    confusion = np.asarray([np.sum(a[i], axis=0) for i in range(len(a))])

#     Plot confusion matrix
    fig, ax = plt.subplots()
    im = ax.imshow(confusion)

    ax.set_xticks(np.arange(14))
    ax.set_yticks(np.arange(14))

    ax.set_xticklabels(np.arange(14)+1)
    ax.set_yticklabels(np.arange(14)+1)
    fig.show()

#     Accuracy
    acc = [confusion[i][i] for i in range(len(confusion))]
    acc_percentage = np.sum(acc)*100/np.sum(confusion)
    print('Accuracy: ', acc_percentage, '%')

    print("Executed in --- %s seconds ---" % (time.time() - start_time))

#     Export Resutls
    exportHistograms(TrainHist1, act_names)
    exportConfusionMatrix(confusion)
    print("Finished Exporting Files")

predict(frac3, frac2, frac1)
```