Lab 4

Last time, we worked towards a fully functional k-means. Today, we are going to look at a few issues concerning k-means:

1. Discuss what *Stopping Conditions* are and how to use them
2. How to use the `sklearn` implementation for k-means
3. How do we decide if we have "good" clusters?

Today we will continue working with our Smith Students data. Please create a copy and place it in your Lab04 folder. Then create the `justtwo` dataframe with just *coffee* and *sleep* variables.

In [ ]:

```
# Package Import block
#    What packages do you need?



# New import
from sklearn.cluster import KMeans
```

In [ ]:

```
# Import your data
students =


# Create your subset
justtwo =
```

Stopping Conditions

As a warm-up, let's start with a few questions about stopping conditions:

1. What are stopping conditions?
2. Why do we care about them?
3. Why are stopping conditions necessary for k-means?

- Write down at least 3 thoughts/ideas/questions about stopping conditions
- Pick at least 1 to share with the class

Wait here for your fellow ML explorers

(For HW2, you will be asked to share your full implementation of k-means, with an explanation of your stopping conditions.)

`sklearn`

There are several important python packages for doing machine learning, but `sklearn` (or *scikit-learn*) is one of the most powerful. Unlike `numpy` , `pandas` , and `scipy` which serve to be used generally for numerical computations, data wrangling, and scientific computing, respectively, the `sklearn` package is specifically for applying machine learning algorithms to data. The implementations in `sklearn` seek to be optimized and this direct usage is considered to be "off the shelf".

In this course, we will first program our own version of each algorithm and then switch to using the optimized versions. In a sense `sklearn` becomes our "check" after we have a deeper understanding of each method.

Note The phrase "off the shelf" refers to the analogy of going to a store and buying something that is designed to just work with little tinkering from the customer. When someone says they used "off the shelf" *method* they mean that they did not modify the *method* implementation.

k-means in `sklearn`

To use k-means in `sklearn` , there are two steps:

1. Setting up how the k-means will function, including:
    - How many clusters (i.e. the k).
    - How you would like the clusters to be selected
    - The starting random state
    - The maximum number of iterations through loop of assigning clusters and finding the centers
2. Applying the k-means you set up to your data

This might seem odd, and perhaps you are wondering why *scikit-learn* can't do this in one step. I would argue that in fact, this two-step procedure mirrors how we as humans approach coding k-means cold. In fact, note that during the last class, 1) we developed our k-means algorithm and then 2) we applied that code to our dataset.

In the below code block, we have one possible setting of `KMeans()` from `sklearn` .

- What *type* is the output and what information is contained within `km_alg` ?
- What are the various inputs (called *parameters* in the helpfiles) doing?

In [ ]:

```
# Step one: Set up the k-means
km_alg = KMeans(n_clusters=2, init="random",random_state = 1, max_iter = 200)
```

In [ ]:

```
# Code block for further discovery
```

Now that we have our k-means set up, we *fit* it to our data. There are a few ways to fit data using k-means. In this course, we will use `.fit()` and `.predict()` most often.

Using `.fit()`

This first fitting applies k-means to the data. What *type* is the output and what information is contained within `fit1` ?

In [ ]:

```
# Step 2: Apply the set k-means to the data using .fit()
fit1 = km_alg.fit(justtwo_np)
```

In [ ]:

```
# Code block for further discovery
```

It's not immediately obvious how to access the 1) cluster assignments, nor 2) the centers for each cluster. The result of `.fit()` wraps this information inside a class style object. We can use `.labels_` and `.cluster_centers_` to access this information:

In [ ]:

```
print("Labels\n", fit1.labels_)

print("\n Cluster Centers \n",fit1.cluster_centers_)
```

Using `.predict()`

If we discover more data that we want to compute the cluster label for **after** applying k-means, we can use `.predict()` on that point. For example, say your friend drinks an average of 1 cup of coffee and sleeps for 8 hours per night, which cluster do they belong to?

In [ ]:

```
new_friend_label = km_alg.predict(np.array([[1,2]]))
print(new_friend_label)
```

Plotting our results

Each time we apply a machine learning method, we should evaluate the results. One of the fastest and more visceral ways to do this is by plotting our results. Adapting the visualizations from this example, we proceed as follows:

In [ ]:

```
labels = fit1.labels_
centers = fit1.cluster_centers_

# Plot and color the points according to their label
plt.scatter(justtwo_np[:,0], justtwo_np[:,1], c=labels, s=50, cmap="spring")
# Add the cluster centers on top
plt.scatter(centers[:, 0], centers[:, 1], c='black', marker="x", s=200, alpha
```

Is this what we expect? Check in with your local ML explorers after you make your next plots

In class, we discussed how we expected to see different clusters. We are likely seeing this due to the scale of the data. So you may decide that you need to scale the different variables so that they are on the same size of axis.

Scaling variables

There are two big commonly used ways to scale variables: normalizing and standardizing. For this lab, we will *normalize* all our variables which will place them within between 0 and 1.

To *normalize* variables, we need to find the minimum and maximum values for each one. Then we use them as follows:

$$Var_{norm} = \frac{Var - Var_{min}}{Var_{max} - Var_{min}}$$

Instead of looping over all the rows for each variable, we can do this computation without a `for` loop.

In [ ]:

```
# Normalizing Coffee variable

coffee = justtwo_np[:,0]
mx = np.max(coffee)
mn = np.min(coffee)

coffee_norm = (coffee - mn)/(mx - mn)
coffee_norm = np.around(coffee_norm, decimals = 2)
```

In [ ]:

```
# Normalize the sleep variable
```

In [ ]:

```
# Check your work with a scatter plot
```

In [ ]:

```
# If you are happy with your plot, then create a new numpy array justtwo_norm

justtwo_norm = np.stack((coffee_norm, sleep_norm),axis=-1)
```

Aside - Here we have to use `stack()` because our arrays are 1-D arrays and not 2-D arrays. (If you run `.shape()` on them, only one number comes out.) If we had 2-D arrays, we would use `concatenate` with syntax like:

```
out_np = np.concatenate((twoD_array1, twoD_array2),axis=1)
```

k-means + scaled variables

The next "check" is whether our scaling did what we expect. Using `sklearn`, deploy kmeans on the normalized data and plot your results. Did you get what you expected?

Check in with your local ML explorers after you make your next plots

In [ ]:

```
# Run kmeans on your normalized data
```

In [ ]:

```
# Plot the results
```

"Good" Clusters

When we have grouped our data into clusters, we would like to know if these are "good" clusters. We might want to look at the:

- The spread of the cluster
- How separated the clusters are from each other

Within Cluster Sum of Squares

A common measure for clusters is the *within cluster sum of squares* which is the total distance between each data point and the cluster center. Using `cdist` and `np.sum` within a `for` loop, we can compute this total. Below is just a bit of code to get you started:

In [ ]:

```
# Within cluster sum of squares

# Compute the following for each cluster:

    cluster_spread = distance.cdist(cluster_points, cluster_center, 'euclidea
    cluster_total = np.sum(cluster_spread)


# Add all the cluster_totals together
```

Question - As k increases, what happens to the *within cluster sum of squares?*

Cluster separation

In cluster separation, we seek to understand the average distance of points from one cluster to the nearest cluster. Again, we use `cdist` but this time we are comparing the points within one cluster to another cluster. We have to compute an average and repeat this process for each cluster. This computation may involve a nested `for` loop. Again, the below code block has a bit of code to get you started:

In [ ]:

```
# Cluster separation

# Compute the following for each cluster:
cluster_sep_mat = np.zeros((num_clusts, num_clusts))
for :
    for :
        cluster_ij = distance.cdist(cluster_points, ???, 'euclidean')
        cluster_sep_mat[i,j] = np.sum(cluster_ij)/???

    # For each cluster determine which is the closest average

# Determine either the average or total of the closest averages
```

Question - As k increases, what happens to *cluster separation?*

Which way is "right"?

In machine learning in general, but in unsupervised in particular, there is often not one right, perfect, or even universally accepted way to evaluate an algorithm. Here a careful eye on 1) your data, 2) your goals, and 3) the overall context of your data and goals is collectively important.

Final Thoughts

To finish up this lab, play with different values for k and decide what the correct value is for this dataset is. Create a post to **#lab_submission** channel on slack stating what you think the correct value for k is with 1) your thinking and 2) a plot of the resulting clustering.

If your have questions from this lab, post them to #lab_questions with the same preamble (i.e. starting with **Lab4**). If you have the same question, please use one of the emoji's to upvote the question. If you would like to answer someone's question, please use the thread function. This will tie your answer to their question.

Resources Consulted

1. *Python Machine Learning*
2. k-means helpfile in sklearn
3. In Depth: k-Means Clustering
4. Colormaps in matplotlib (ie. `cmap`)
5. Scatter Helpfile
6. Standardize or Normalize? — Examples in Python
7. Concatenation of 2 1D `numpy` Arrays Along 2nd Axis
8. `stack()` helpfile
9. `concate()` helpfile
10. Interpret all statistics and graphs for Cluster K-Means