Practicum 9 - k-means clustering, part 1

DS2001 - Computer Science Practicum
Fall 2021
November 3rd - 4th, 2021
Practicum Deadline: November 5th, 2021 at 12pm (noon!) Boston time

Next week, we will be holding in-person final project office hours/workshop time on November 10th in the following locations:
9:50am - 11:30am WVF 010
2:50pm - 4:30pm Snell Library 015
We *highly* encourage ~*all groups*~ to take advantage of these as there will not be a large amount of time during practicum hours to work on your projects! (you'll have to do the vast majority of it outside of class)

What we're practicing today:
- functions
- dicts

Handouts:
- dicts (from DS 2000)

What to do if you miss practicum this week:
- Fill out the "I'm missing class" form ASAP
- Follow up in office hours with Felix (find office hours links on the course website). **Note that this is required.**
- Come to practicum next week if you are well again. **Do not come to class if you are not feeling well.**

How far we expect you to get this week:
- We expect you to complete **Task 3**, which means attempting to write a `distance` function with parameters and a return value that make sense.

Everyone should create **one** file for the whole practicum called **kmeans.py**. We'll add to this file for practicum 10 as well!

At the end, everyone should make sure to turn in your code to gradescope **with your group's answers to the questions written as comments in your file!**

Task 0: Write your file comment, your `if __name__ == "__main__":` and import the `kmeans_utils` and `matplotlib.pyplot` modules
First, write down your name in a comment at the top of your file and a small description of what the file is, like we did last time. This is a good habit to get into for all your programs!

Task 1: Introducing the algorithm and a new data type — the tuple

Today will be part 1 of a two-part series in which you will be implementing an algorithm from machine learning called k-means clustering. This algorithm is one that figures out groups ("clusters") of points that belong together. At its basis, it is composed of a few steps:
1) graph your data
2) pick some means ("centroids") by randomly plotting k new data points (e.g. k=2 for us)
3) assign each data point to the closest centroid
4) adjust the centroids to be closer to the mean of the group of points that they "belong to"

5) Repeat steps 2 & 3 until the centroids no longer move

This is an algorithm that is used in a wide range of applications. For example, it is used to identify cancerous vs. non-cancerous cells in computational biology!

A `tuple` is a data type in python, just like how a string or an int is a data type. A `tuple` is very similar to a list, with one important difference—you cannot change an element of a `tuple`. You can index into the tuple and loop over it just like a list though!

We call tuples **immutable** data types, which is a fancy way of saying that they are just like strings and integers: you can't change just one letter or one decimal place. To change a `tuple`, you have to reassign the whole thing.

```
# make a tuple
tup = (13, 47)  # notice the curvy parentheses, instead of square brackets!
print(tup[0])  # prints 13
tup[0] = 0  # ERROR - NO REASSIGNMENT
# versus making a list
ls = [13, 47]  # notice the square brackets!
print(ls[0])  # prints 13
ls[0] = 0  # Okay!
print(ls[0])  # prints 0
```

We'll be working with 2-dimensional points in the form of tuples over the next two practicums. The main reason for this is because we would like to use our points as **keys** in a **dictionary**.

For this task, write down code that:

1) creates a dictionary where the keys are **strings** and the values are **integers**. Put at least three different keys in your dictionary.
    a) (You can pick any strings and values here — the only goal is to make a dictionary!)
2) creates a dictionary where the keys are either **lists** or **tuples** and the values are **integers**. Put at least three different keys in your dictionary.
    a) (You can pick any values to put in your dicts here — the only goal is to make a dictionary and experiment with lists and tuples!)

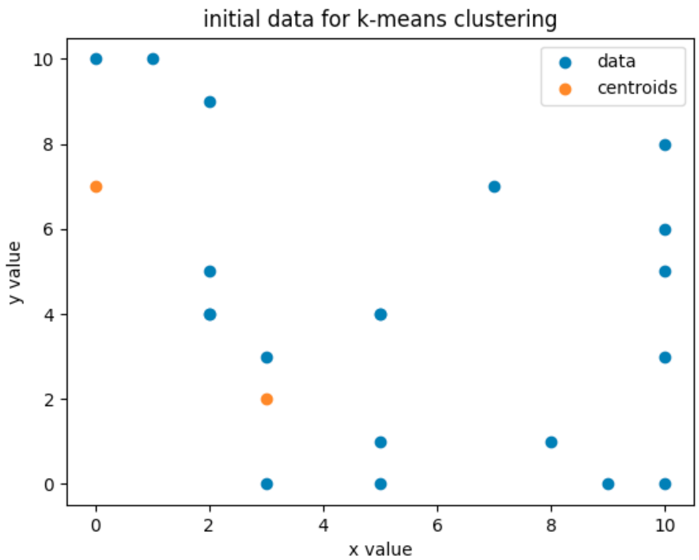
Group question: what happens when you try to make a dictionary with a list as a key mapping to an integer as a value? What about if a tuple is a key? Write down the code to do these experiments! (if they cause an error, leave them in your practicum, just comment them out)

Task 2: Generate points, then graph them on a scatter plot

Next, use the provided functions in the `kmeans_utils` module to do the following:

1) generate two different lists of points, one with 20 points (this will be the **data** that you are clustering), and one with two different points (these will be your starting **centroids**).
2) using matplotlib, make a scatter plot with the data as one color and the centroids as a different color.
    a) Hint! you'll want to make use of the `get_column` function to help with this. Remember that `plt.scatter` takes parameters like:
    `plt.scatter(x_values, y_values)`
    you can also use:
    `plt.plot(x_values, y_values, ".") # for points`

Example output:
```
data: [(0, 10), (10, 5), (9, 0), (5, 1), (10, 3), (10, 0), (1, 10), (2, 4), (2, 5), (3, 0), (5, 4), (7, 7), (5, 4), (8, 1), (10, 6), (2, 4), (2, 9), (3, 3), (10, 8), (5, 0)]
centroids: [(3, 2), (0, 7)]
```



Task 3: Calculate the distance between two data points
Write a function that will calculate the Euclidean distance between two 2-dimensional points. The function signature is up to you! (remember, you're doing a calculation here, so you should probably return the value that you calculate)

Test your function out by calling it, passing in two points that you know the distance between. Make sure that this function works if you pass in two points from your lists of data or your centroids themselves!

Example output:
```
Testing distance function
distance from (0, 0) to (1, 1): 1.4142135623730951
distance from (-10, 0) to (10, 0): 20.0
first point in the data list: (0, 10)
first point in the centroids list: (3, 2)
distance between them: DISPLAY DISTANCE CALCULATED HERE
```

Task 4: Find the closest point
Write a function that answers the question "given a single point and a list of centroids (2-d points), which centroid (2-d point) is the target point closest to?". Again, function name and signature are up to you, but you'll want your return value to be the point (a tuple) that is closest to the target point!

Test your function out by calling it, passing in a single point and your list of centroids. Make sure to test it with different values!

Example output:
```
Testing closest function
closest centroid to (0, 0): (3, 2)
```

```
closest centroid to (10, 10): (0, 7)
```

Group question: how might having this function help with the overall k-means algorithm?

Task 5: Count the number of points "closest" to each cluster centroid

As our final task of "part 1" of implementing this algorithm, we'll be counting up how many of our data points match with each centroid. For this task, we'll need to make a dictionary mapping centroids (tuples) to integers (the count of how many points "belong" to this centroid.

Example output:

```
counts of points assigned to clusters
{(0, 7): 4, (3, 2): 16}
```

If you finish...

- work on your final project!
- think about how you would assign each point to the centroid it belongs to, rather than just counting up how many points belong with which one