Northeastern University
CS1 – Introduction to Computer Science
Fall 2018, Derbinsky & Strange

Nearest Neighbor Search

Introduction

When analyzing data, it is often useful to find one or more data points that are closest (or most similar) to a given point. Once these *nearest neighbors* are identified, they can be used to understand groups, or *clusters*, within your data, as well as to make predictions. For example, nearest neighbors might be used to identify products similar one that a user has previously purchased, or to find films that are akin to a user's favorite; in general these constitute specific instances of "recommendation systems," which are often powered by nearest neighbors-like technology.

Importantly, depending on your problem, there are various ways to define the "closeness/similarity" of your data points. For instance, when looking for the closest available Lyft cars, a system might use straight-line (*Euclidean*) distance; however, in a grid-based city like New York, the "taxi cab" (*Manhattan*) distance[1] might be more appropriate, as it counts the number of blocks that the cabs would actually need to traverse. More abstractly, the things we use to represent the objects (data points) being compared will tend to dictate the appropriate measure of similarity.

This homework includes five problems, each of which involves you implementing a function. When combined, these five functions implement a nearest-neighbor search to be used for *classification*: that is, it will predict the label of a new data point to be the label of its nearest neighbor. For example, a program of this type that is supplied restaurant locations might predict that a new restaurant in the North End of Boston would be labeled as serving "Italian" cuisine.

[1] `https://en.wikipedia.org/wiki/Taxicab_geometry`

Expectations

To begin this assignment, download `hw3_starter.zip`, which contains a starter file and a test suite. As you answer each question, remember to comment your code & run the test suite. Also remember that passing all tests does not *guarantee* a perfect grade, but it certainly suggests that you are making good progress. Finally, remember to be very careful about capitalization, spacing, spelling, etc. – as you know by now, these tests are quite picky!

To receive credit, submit to Blackboard a **single ZIP file** that contains **only** the following file:

- `nn.py`: it will contain all your functions

You are welcome to upload as many submissions as you wish (and indeed, uploading multiple times is a good strategy for backing up your work!) – **but we will only grade the final/latest submission**. Once you have uploaded your (final) submission to Blackboard, we recommend that you download the file to make sure that you didn't accidentally submit the wrong (or blank!) file.

In general, you are not allowed to use advanced Python functionality that we have not yet covered in class (particularly if it trivializes the problem). If you have a question as to whether a particular piece of code is acceptable, please **ASK** during class, office hours, or via e-mail/(privately) Piazza.

Question 1 (`myabs`; 10 points)

In this problem you will implement a function to return the absolute value of the supplied input. For example ...

```
>>> myabs(-5)
5
>>> myabs(0)
0
>>> myabs(200.4)
200.4
>>>
```

Note that while Python has the `abs` function built-in, you are **NOT** allowed to use it in your implementation. This function will get you warmed up & will come in handy in the next problem `:)`.

Question 2 (`mymanhattan`; 20 points)

In this problem you will implement your first *distance function*[2]: given two points it returns a value (≥ 0) that represents the distance between those points. The two arguments to the function will be lists of numbers (you can assume they are the same length) and you are to implement the "Manhattan distance"[3]:

$$d(a,b) = \sum_j |a_j - b_j|$$

Meaning, the distance between two points is the sum of the absolute value of the difference between each of the points' coordinates. If it helps, think about how many blocks you would have to walk (using only right angles at corners) between two locations. For example, let's find the Manhattan distance between the points $(0,0)$ and $(3,4)$ ...

$$d((0,0),(3,4)) = |0-3| + |0-4| = 3 + 4 = 7$$

And in Python ...

```
>>> mymanhattan([0,0], [3,4])
7
```

This is a two-dimensional example, but note that this could work no matter how many coordinates (i.e., in an arbitrary number of dimensions)! For example, let's find the Manhattan distance between the points $(1,4,7,0)$ and $(-3,4,20,1)$ ...

$$d((1,4,7,0),(-3,4,20,1)) = |1--3| + |4-4| + |7-20| + |0-1| = 4 + 0 + 13 + 1 = 18$$

And in Python ...

```
>>> mymanhattan([1,4,7,0], [-3,4,20,1])
18
```

Again, you are not allowed to use the built-in `abs` function, but feel free to call your own solution to Question 1! It is also probably worth thinking for a moment about why we need to take the absolute value here.

[2] `https://en.wikipedia.org/wiki/Metric_(mathematics)`
[3] `http://mathworld.wolfram.com/TaxicabMetric.html`

Question 3 (`myhamming`; 20 points)

In this problem you are going to implement a second distance function: the Hamming distance[4]. The hamming distance between two points is simply the count of the number of coordinates that are *different*. (This may seem strange, but it can come in handy when finding similar words/names or detecting errors in communicated information.)

For example, the Hamming distance between the points $(0, 0)$ and $(3, 4)$ is 2, because both coordinates are different between the points ($0 \neq 3$ and $0 \neq 4$). And in Python ...

```
>>> myhamming([0,0], [3,4])
2
```

Similarly, the Hamming distance between the points $(1, 4, 7, 0)$ and $(-3, 4, 20, 1)$ is 3 because only the second coordinate, of four, is the same between the points (4) ...

```
>>> myhamming([1,4,7,0], [-3,4,20,1])
3
```

Question 4 (`myeuclidean`; 20 points)

In this problem you are going to implement one last distance function: Euclidean distance[5].

$$d(a, b) = \sqrt{\sum_j (a_j - b_j)^2}$$

For example, let's find the Euclidean distance between the points $(0, 0)$ and $(3, 4)$ ...

$$d((0, 0), (3, 4)) = \sqrt{(0-3)^2 + (0-4)^2} = \sqrt{9+16} = 5$$

And in Python ...

```
>>> myeuclidean([0,0], [3,4])
5.0
```

And the Euclidean distance between the points $(1, 4, 7, 0)$ and $(-3, 4, 20, 1)$ ...

$$d((1, 4, 7, 0), (-3, 4, 20, 1)) = \sqrt{(1--3)^2 + (4-4)^2 + (7-20)^2 + (0-1)^2} = \sqrt{16+0+169+1} \approx 13.638$$

And in Python ...

```
>>> myeuclidean([1,4,7,0], [-3,4,20,1])
13.638181696985855
```

Note that you are not allowed to use any modules (we'll talk about those next week!), so remember that taking the square root of a number is the same as raising it to the $\frac{1}{2}$ power.

[4] `https://en.wikipedia.org/wiki/Hamming_distance`
[5] `https://en.wikipedia.org/wiki/Euclidean_distance`

Question 5 (`nearestneighbor`; 30 points)

It's time to put all these distance functions to use! In particular we will them to 'classify' a point via its nearest neighbor. You will be supplied a point (as a list of numbers, like in the previous problems), a "community" (basically, a dataset of labeled data points), and a distance function to use – your job is to return the label of the nearest neighbor to the point according to the distance function.

Consider the following simple "community" in Python ...

```
triangle = [
    ["top", [0, 1]],
    ["bottom-left", [0, 0]],
    ["bottom-right", [2, 0]],
]
```

This list-of-lists represents a right triangle where the "top" is located at $(0, 1)$, the "bottom-left" is at $(0, 0)$ and the "bottom-right" is at $(2, 0)$. Now consider the following call to this function in Python ...

```
>>> nearestneighbor([0, 0.6], triangle, myeuclidean)
'top'
```

The new point is $(0, 0.6)$ and the distance function is Euclidean (i.e. your solution to Question 4). Now let's confirm this result ...

```
>>> myeuclidean([0, 0.6], [0, 1])
0.4
>>> myeuclidean([0, 0.6], [0, 0])
0.6
>>> myeuclidean([0, 0.6], [2, 0])
2.08806130178211
```

So we can see that $(0, 1)$ is the closest point, and so the function should return it's label ("top").

At a high level, your function should loop over all data points in the community & compute the distance to the test point, keeping track of the shortest seen so far (if two have the same distance, simply keep the first that was encountered). When you've reached the end of the list of labeled points, return the label of the closest point found.