DOCUMENTATION FOR APM CHALLENGE CODE

1. Setup

I chose to write my solution in Python. To run it, you will need Python 3.5, as well as current versions of the NumPy and pandas libraries. They can be installed with Python's package manager `pip` with

```
pip install numpy
```

and

```
pip install pandas
```

2. Overview

I have created three Python files:

`proximity_calculation.py`: This module contains the core logic.
`test_proximity_calculation.py`: This file contains unit tests for the public methods in `proximity_calculation`.
`solve_questions.py`: This file can be executed directly to calculate solutions to the specific questions asked.

3. Ambiguity of Bonus Question

It seems to me that the bonus question can be interpreted in two ways, so I should make clear the interpretation I assumed. The bonus question asks

> What is the maximum radius R such that the number of coordinates within a distance strictly less than R of any centroid is at most 1000?

This could be interpreted as either

> What is the maximum radius R such that the size of the set
>
> $$\{x \in \text{coordinates} \mid \exists y \in \text{centroids s.t. } distance(x, y) < R\}$$
>
> is at most 1000?

or

> What is the maximum radius R such that for all centroids y, the size of the set
>
> $$\{x \in \text{coordinates} \mid distance(x, y) < R\}$$
>
> is at most 1000?

I chose the second interpretation (though the first can also be easily answered with the code I wrote).

4. Solutions

Here are my solutions to the problems. They can be reproduced by running

```
python solve_questions.py [path to centroids file] [path to
coordinates file]
```

1. How many coordinates are within 5 meters of at least one of the K centroids? **90851**
2. How many coordinates are within 10 meters of at least one of the K centroids? **315614**
3. What is the minimum radius R such that 80% of coordinates are within R meters of at least one of the K centroids? **21.248**
4. What is the maximum radius R such that the number of coordinates within a distance strictly less than R of any centroid is at most 1000? **11.170**

5. Interface

`proximity_calculation` exposes the following methods:

`smallest_nth_proximity`:
: This function takes sets of centroid and coordinate locations as input, along with an integer n. It returns the smallest radius R such that some centroid contains at least n coordinates within radius R.

`ClosestCentroidCalculator.num_coordinates_within`:
: This function calculates the number of coordinates that are within a specified radius of any centroid.

`ClosestCentroidCalculator.min_radius_enveloping_percent`:
: This function takes a percent p as input, and it returns the smallest radius R such that a percent p of coordinates are within radius R of a centroid.

6. Performance Analysis

The main work of my solution is done in the `_nth_proximities` function in the `proximity_calculation` module. This function takes three parameters:

`X`: a numpy array with two columns
`Y`: a numpy array with two columns
`n`: an integer

The function calculates, for each row `y` in `Y`, the `n`th smallest distance from `y` to any row in `X`. This general function can be used to solve all of the questions asked.

For each row `y` in `Y`, the function performs two operations:

1. Find the distance from `y` to each element of `X`. (This is implemented in the helper method `distances`.) This requires $O(\text{length}(\texttt{X}))$ operations, since it requires a fixed number of arithmetic operations for each element of `X`. It uses $O(\text{length}(\texttt{X}))$ space.
2. Find the `n`th smallest among the distances calculated in step 1. This is done using the `heapq.nsmallest` function, which pushes each element of the set of distances into a heap of fixed size `n` and then sorts the final heap. Since each push into a heap of size `n` is done in $O(\log \texttt{n})$ time and sorting the heap takes $O(\texttt{n} \log \texttt{n})$ time, the operation terminates in $O((\text{length}(\texttt{X}) + \texttt{n}) \log \texttt{n})$ time. It requires $O(\texttt{n})$ space, for the heap of size `n`.

The time required for the entire method to run is therefore

$$O(\text{length}(\texttt{Y}) \cdot (\text{length}(\texttt{X}) + (\text{length}(\texttt{X}) + \texttt{n}) \log \texttt{n}))$$

If $\text{length}(\texttt{X}) \gg \texttt{n}$, the runtime reduces to

$$O(\text{length}(\texttt{Y}) \cdot \text{length}(\texttt{X}) \cdot (1 + \log \texttt{n})) \tag{1}$$

Since the arrays used for steps 1. and 2. can be forgotten after each iteration finishes, the amount of storage required for the entire method is

$$O(\text{length}(\texttt{Y}) + \text{length}(\texttt{X}) + \texttt{n}). \tag{2}$$

Let K be the number of centroids and N the number of coordinates. Then from (1), we see that the public methods have the following time complexities:

`ClosestCentroidCalculator.__init__`:
: This function calls `_nth_proximities` with $\texttt{n} = 1$ and then it sorts the array, so its time complexity is

$$O(NK + N \log N)$$

`num_coordinates_within`:
: This function performs a binary search on the sorted set of coordinate proximities, so its complexity is

$$O(\log N)$$

`min_radius_enveloping_percent`:
: This function does a constant number of arithmetic operations and an array lookup, so it runs in constant time.

`smallest_nth_proximity`:
: This function's main work is done in a call to `_nth_proximities` with `X` the set of coordinates and `Y` the set of centroids. Assuming $\texttt{n} \ll N$, its complexity is

$$O(NK(1 + \log \texttt{n}))$$

Each of these methods requires no significant extra storage after `_nth_proximities` runs, so if $n \ll N$, then by (2) each of these methods requires $O(N + K)$ space.