Spring 2023 CS4641/CS7641 A Homework 1 - Programming Section

Instructor: Dr. Mahdi Roozbahani

Deadline: Friday, February 10th, 11:59 pm EST

- No unapproved extension of the deadline is allowed. Submissions past our 48-hour penalized acceptance period will lead to 0 credit.
- Discussion is encouraged on Ed as part of the Q/A. However, all assignments should be done individually.
- Plagiarism is a **serious offense**. You are responsible for completing your own work. You are not allowed to copy and paste, or paraphrase, or submit materials created or published by others, as if you created the materials. All materials submitted must be your own.
- All incidents of suspected dishonesty, plagiarism, or violations of the Georgia Tech Honor Code will be subject to the institute's Academic Integrity procedures. If we observe any (even small) similarities/plagiarisms detected by Gradescope or our TAs, **WE WILL DIRECTLY REPORT ALL CASES TO OSI**, which may, unfortunately, lead to a very harsh outcome. **Consequences can be severe, e.g., academic probation or dismissal, grade penalties, a 0 grade for assignments concerned, and prohibition from withdrawing from the class.**

Instructions for the assignment

- This assignment consists of warm-up programming questions designed to get you familiar with our programming homework structure.

Using the autograder

- Grads will typically find three assignments on Gradescope and Undergrads will typically find four assignments:
  - "Assignment X Non-programming": Where you will submit the written portion of the assignment.
  - "Assignment X Programming": Where you will submit any .py files and any program outputs as required by the problem.
  - "Assignment X Programming - Bonus for All": Where you will submit any .py files and any program outputs as required for Bonus for All.
  - "Assignment X Programming - Bonus for Undergrad": Where you will submit any .py files and any program outputs as required for Bonus for Undergrand. (Undergrad Only)
- You will submit your code for the autograder in the Assignment 1 Programming section.
- We provided you .py files and we added libraries in those files please **DO NOT** remove those lines and add your code after those lines. Note that these are the only allowed libraries that you can use for the homework.
- You are allowed to make as many submissions until the deadline as you like. Additionally, note that the autograder tests each function separately, therefore it can serve as a useful tool to help you debug your code if you are not sure of what part of your implementation might have an issue.

Deliverables and Points Distribution

Q6: Programming Warm-Up [2pts total]

Deliverables:

- `warmup.py`
- `env.pkl`

Parts:

- **Setup** [1pts] - *programming*
- **Numpy** [1pts] - *programming*
  - Numpy Basics [0.5pts]
  - Broadcasting [0.5pts]

6.1 Setup [1pt]

- **Deliverable:** `env.pkl`

This notebook is tested under python 3.10.9 (https://docs.python.org/release/3.10.9/), and the corresponding packages can be downloaded from miniconda (https://docs.conda.io/en/latest/miniconda.html). You may also want to get yourself familiar with several packages:

- jupyter lab (https://jupyterlab.readthedocs.io/en/stable/): provides a web-based IDE with a built-in debugging functionality for jupyter notebooks.
- numpy (https://numpy.org/doc/1.23/): a high performance math library backed by C
- matplotlib (https://matplotlib.org/users/pyplot_tutorial.html): a python plotting libaray

Other packages you may find indispensable in machine learning (and potentially your project) are:

- scikit-learn (https://scikit-learn.org/stable/getting_started.html): provides many classical ML and data analysis algorithms
- pandas (https://pandas.pydata.org/docs/): provides many useful tools for organizing and manipulating data
- seaborn (https://seaborn.pydata.org/tutorial/introduction.html): make beatiful plots with less fidgeting in matplotlib
- plotly (https://plotly.com/python/): another great data visualization package

Please implement the functions that have "raise NotImplementedError", and after you finish the coding, please delete or comment "raise NotImplementedError".

In [12]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

import sys
import numpy as np

print('Version information')

print('python: {}'.format(sys.version))
print('numpy: {}'.format(np.__version__))


%load_ext autoreload
%autoreload 2
```

```
Version information
python: 3.10.9 (main, Jan 11 2023, 09:18:20) [Clang 14.0.6 ]
numpy: 1.23.5
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

6.1.1 Basics, Imports, and Directories

For the following part, you will need to ensure your notebook runtime is started in the correct directory. You can verify this with the following cell.

In [ ]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

# RUN ME #
import os
os.getcwd()
```

In the cell below, import the `PackageUtils` class from `utils.py` located in the `utilities` folder.

In [14]:
```
# YOUR CODE HERE #


```

In the cell below, call the `get_packages` method of the PackageUtils class to see what packages are installed in this notebook's runtime environment.

In [ ]:
```
# YOUR CODE HERE #


```

6.1.2 Local Testing & Debugging

Optional local tests using a small toy dataset are sometimes provided to aid in debugging. The local tests are all stored in localtests.py

The autograder is the final arbiter

- There are no points associated with passing or failing the local tests, you must still pass the autograder to get points.
- It is possible to fail the local test and pass the autograder.
  - The autograder may have tolerances to account for minor implementation differences.
  - The reverse is also true, as the autograder may cover a larger number of corner cases.
- **You do not need to pass both local and autograder tests to get points, passing the Gradescope autograder is sufficient for credit.**

Work smarter, not harder

- Read the stack trace carefully. Often it will tell you exactly what's wrong.
- Understand what the local-test is doing. That way you can develop your own tests.
- Grow beyond the print statement: embrace a debugger. Jupyter-lab has a built in debugger (https://jupyterlab.readthedocs.io/en/stable/user/debugger.html) which allows you to look at data types, set breakpoints, and examine variables. If using a different IDE, look up your IDE's documentation on how to setup a proper debugger.
- Develop incrementally and test frequently, both locally and on Gradescope. Waiting to complete the whole class before testing can make it hard to isolate errors.

For this problem perform the following in the cell below:

- import `WarmupTests` from the `localtests.py` in the `utilities` folder.
- Run the cell and submit `env.pkl`

In [16]:
```
import unittest

# YOUR CODE HERE #
# import WarmupTests from the localtests.py in the utilities folder.



# END YOUR CODE ABOVE #




# Do not change below #
unittest.main(argv=['ignored', 'WarmupTests.test_get_packages'], verbosity=1, exit=False)
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK

Passed test_get_packages
```

Out[16]: <unittest.main.TestProgram at 0x7fe37060d3f0>

6.2 Numpy Basics [0.5pt]

The following exercise will familiarize you with the basics of working with Numpy and navigating the numpy documentation (https://numpy.org/doc/1.23/reference/index.html)

In `warmup.py` you will implement several "one-liners" using functions provided by numpy. No points will be awarded on Gradescope for any use of for loops or list comprehensions. Implement the following functions in `warmup.py` :

- indices_of_k
- argmax_1d
- mean_rows
- sum_squares

You may test your implementation with the below local tests. These local tests only checks the returned values of your implementation and does not check whether your implementation uses loops. Gradescope will check to make sure your implementation does not use loops (for, while, or list comprehensions).

In [17]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################
import unittest
from utilities.localtests import WarmupTests
unittest.main(argv=[''], verbosity=1, exit=False)
```

```
.....
----------------------------------------------------------------------
Ran 5 tests in 0.002s

OK

Correct Values for argmax_1d
Passed test_get_packages
Correct Values for indices_of_k
Correct Values for mean_rows
Correct Values for sum_squares
```

Out[17]: <unittest.main.TestProgram at 0x7fe340f331c0>

6.3 Broadcasting [0.5pt]

One of the simplest and most common similarity metrics in ML is the Manhattan Distance or taxicab-distance (https://en.wikipedia.org/wiki/Taxicab_geometry). The function below takes two lists of N points in D dimensional space (NxD numpy arrays) and computes the Manhattan distance between every possible pair of points. *Hint: you can use this to try creating your own unittests*

Unfortunately such an implementation is too slow for a large dataset. In `fast_manhattan` leverage the broadcasting properties of numpy to create a faster version in a single line.

In [18]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

import numpy as np
def slow_manhattan(x, y):
    """
    Args:
        x: N x D numpy array
        y: M x D numpy array
     Return:
        dist: N x M numpy array, where dist[i, j] is the Manhattan distance between
        x[i, :] and y[j, :]
    """
    dist = np.empty((x.shape[0], y.shape[0]))
    for i in range(x.shape[0]):
        for j in range(y.shape[0]):
            d = 0
            for k in range(x.shape[1]):
                d += abs(x[i][k]-y[j][k])
            dist[i][j] = d
    return dist
```

Let's test the speed of this naive implementation:

In [19]:
```
%%timeit

##############################
### DO NOT CHANGE THIS CELL ###
##############################

x = np.random.rand(100, 3)
y = np.random.rand(100, 3)
d = slow_manhattan(x, y)
```

```
16.7 ms ± 86.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Compare this with the vectorized implementation:

In [20]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

import warmup
```

In [21]:
```
%%timeit

##############################
### DO NOT CHANGE THIS CELL ###
##############################

x = np.random.rand(100, 3)
y = np.random.rand(100, 3)
d = warmup.fast_manhattan(x, y)
```

```
268 µs ± 3.36 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Finally for `multiple_choice` answer the following by returning the correct integer value:

Which of the following best describes the space and time complexity of `slow_manhattan` compared to `fast_manhattan`:

- return 0 if: `fast_manhattan` has lower space and time complexity.
- return 1 if: `slow_manhattan` has lower space complexity and the same time complexity.
- return 2 if: `fast_manhattan` has higher space complexity and lower time complexity.
- return 3 if: Both are about the same in space and time complexity.

In [2]:
```
##############################
### DO NOT CHANGE THIS CELL ###
##############################

warmup.multiple_choice()
```

```
0, 1, 2, or 3
```