CMPUT 328 Fall 2019 Assignment 1

k-Nearest Neighbors [worth 5.7% of the total weight]:

Implement *k*-nearest neighbors classification algorithm on the MNIST dataset in PyTorch. The *k*-nearest neighbor algorithm works as follows:

- First, choose a distance function (for example: Euclidean distance, Manhattan distance, cosine similarity...) on the input space.
- For every test data point, find *k* points in the train set that have closest distances to this point. These are called *k*-nearest neighbors of it. Classification result of the test data point is determined by a majority vote of its neighbor.

You will need to vary value of ***k*** and choose the distance function to reach optimal performance. Usually, the Euclidean distance is good enough for most tasks.

NOTE: A correct implementation of this algorithm will have an accuracy of around 94->96% on the **test** set.

ADDITIONAL MATERIALS:

https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

DUE DATE: The due date is **Friday, September 20 at 11:55 pm**. The assignment is to be submitted online on eclass. For late submission rules, please check the course information on eclass.

COLLABORATION POLICY: This must be your own work. Do not share or look at the code of other people (whether they are inside or outside the class). Do not search for or copy code from the Internet. You can talk to others that are in the class about solution ideas but **not so detailed that you are verbally sharing or hearing about or seeing the code**. You must **cite** whom you talked to in the comments of your programs.

SUBMISSION: You need to submit one file: ***knn.py*** on eclass.

First, you need to download both files in Assignment 1 Resources on eclass.

The first file named *Assignment_1_notebook.ipynb* is a Jupyter notebook for you to work on. Upload this file to Google Colab to use it. This file contains the template code for this assignment and a function with signature `def knn(x_train, y_train, x_test, n_classes, device)` which you need to implement.

After you finish implementing this function, you can run the main loop in the notebook to get your result (including run time and accuracy) like these for Tesla K80 and Tesla T4 GPUs on Colab:

```
Dimension of dataset:
Train: (60000, 784) (60000,)
Test: (1000, 784) (1000,)
Running...
Training on GPU: Tesla K80
Correct Predict: 963/1000 total         Accuracy: 0.963000      Time: 6.951531
OrderedDict([   ('correct_predict', 963),
                ('accuracy', 0.963),
                ('score', 100.0),
                ('run_time', 6.951531206000254)])
```

```
Dimension of dataset:
Train: (60000, 784) (60000,)
Test: (1000, 784) (1000,)
Running...
Training on GPU: Tesla T4
Correct Predict: 963/1000 total         Accuracy: 0.963000      Time: 3.433709
OrderedDict([   ('correct_predict', 963),
                ('accuracy', 0.963),
                ('score', 100.0),
                ('run_time', 3.4337091210000494)])
```

Once you are sure your result is satisfactory, move on to the next step with the second file.

This file is a zip file that contains two files within: *knn.py* and *main.py*. The important file here is *knn.py*. Copy and paste the `knn` function you implemented from your notebook into the *knn.py* file. Now you can submit this *knn.py* file onto e-class.

The file *main.py* is provided just in case you want to run your code from a command line and not in the notebook. This file is not required for this assignment and you should **not** submit it.

Again, *knn.py* should be the only file you submit. Do not submit pdf or doc or other incorrectly formatted files.

MARKING:

25% Marks: Describe your program for TAs (**Sep 16– Sep 20**)

TAs can select five random questions based on your code, results, and kNN algorithm. The time to present will be in your lab section in the week **when this lab is due**. Note that you must present this part to your TA in your designated lab section. *You will not get mark for this part if you don't present in your own lab section.*

75% Marks: A correct implementation of this algorithm will have an accuracy of around **94 -> 96%** and an execute time around **7 seconds** on the test set. Your mark will be determined by these criteria:

- **Accuracy**: determines your base score:
  - Below 84%: 0 points
  - 84% to 94%: from 0 to 100 points, scaled linearly. For example, 85% will give you 10 points, 89% will give you 50 points.
  - More than 94%: 100 points

- **Run time**: Your final score also depends on the run time. You will retain or lose some of your base score depending on your run time. Note that this runtime constraint is less strict than the expected run time of a correct implementation, which is **7 seconds**. This runtime was obtained using **Tesla K80** GPU on Colab which will also be used to evaluate your solutions.

  - Less than 12 seconds: You get 100% of base score
  - More than 12 seconds but less than 24 seconds: You get 50% of base score
  - More than 24 seconds: You don't get any points

  Please note that Colab sometimes provides **Tesla T4** that is much faster and has a runtime of **3.4 seconds** so you can scale your own runtimes appropriately. Also, the runtime is usually higher during the first run and decreases on subsequent runs. The time from **second run** will be used for evaluation.

The final mark for this part will be computed as: (final score / 100) * 75%.

What is *k*-Nearest Neighbors

The *k*-NN algorithm belongs to the family of instance-based, competitive learning and lazy learning algorithms. Instance-based algorithms are those algorithms that model the problem using data instances (or rows) to make predictive decisions. The *k*-NN algorithm is an extreme form of instance-based methods because all training observations are retained as part of the model.

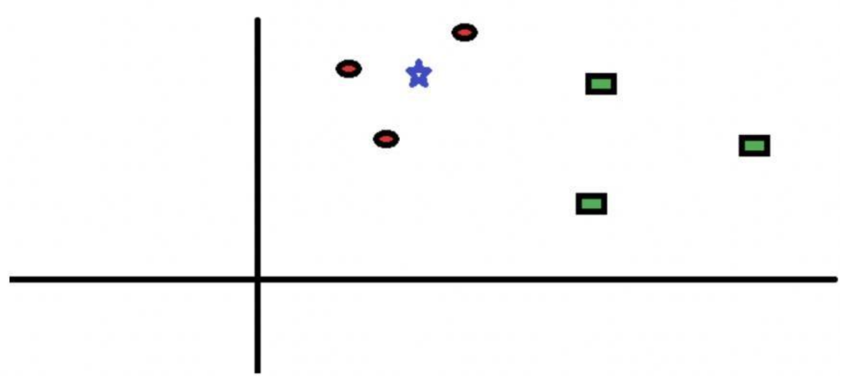
It is a competitive learning algorithm because it internally uses competition between model elements (data instances) to make a predictive decision. The objective similarity measure between data instances causes each data instance to compete to “win” or be most similar to a given unseen data instance and contribute to a prediction.

Lazy learning refers to the fact that the algorithm does not build a model until the time that a prediction is required. It is lazy because it only does work at the last second. This has the benefit of only including data relevant to the unseen data, called a localized model. A disadvantage is that it can be computationally expensive to repeat the same or similar searches over larger training datasets.

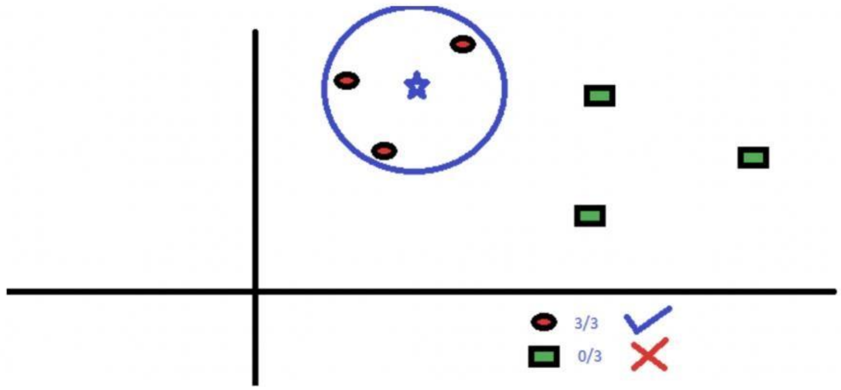
Finally, *k*-NN is powerful because it does not assume anything about the data, other than a distance measure can be calculated consistently between any two instances. As such, it is called non-parametric or non-linear as it does not assume a functional form.

How does the *k*-NN algorithm work?

Let’s take a simple case to understand this algorithm. Following is a spread of red circles (RC) and green squares (GS):



You intend to find out the class of the blue star (BS). BS can either be RC or GS and nothing else. The *k* in *k*-NN algorithm is the number of nearest neighbors we wish to take a vote from. Let’s say *k* = 3, and now, we will make a circle with BS as a center, just as big as to enclose only three data points on the plane. Refer to the following diagram for more details:



The three closest points to the BS are all RC. Hence, with the reasonable confidence level, we can say that the BS should belong to the class RC. Here, the choice became very obvious as all three votes from the closest neighbor went to RC. The parameter *k* is constant, and the choice of it is very crucial in this algorithm.

How to implement k-Nearest Neighbors

Breaking it Down – Pseudo Code of KNN

We can implement a *k*-NN model by following the below steps:

- Convert data from numpy arrays to pytorch tensors
- Choose a distance function and a value for *k*
- For each test image:
    - Calculate the distance between the test image and all training images.
    - Find indices of *k* training images with the smallest distances
    - Get classes of the corresponding training images
    - Find the most frequent class among these *k* classes
        - Represent classes as one-hot vectors and stack into a $k \times 10$ array
        - Compute column-wise sum of this array
        - Take the column with the maximum sum
    - Return the predicted class