A3: Parallel KNN

Due Wednesday by 5pm **Points** 10

Note: There is a small bug in the starter code. There should be a break statement after the following two lines (line 62 in classifier.c)

```
switch(opt) {
    case 'v':
        verbose = 1;
        break;  // ADD THIS LINE
```

Introduction

So far we've seen two machine learning protocols: nearest neighbor classification and decision trees. We saw that KNN classification tends to be accurate but slow, whereas the binary decision tree classifier was a little less accurate but much faster. In this assignment we will aim to mitigate the speed issues of KNN by parallelizing the code to run over multiple processes which can be distributed over the the cores in your processor. In practice, this can lead to speedups of between up to a factor of 2 (for an Intel Ice Lake 2 Core CPU) or up to a factor of 64 (AMD Threadripper 64 Core CPU) depending on the details of your processor. The reason for these speedups is that testing a KNN classifier is embarrassingly parallel, which allows us to perform the classification independently for each of the vectors in the test set. This makes ML tasks such as the KNN classification problem you considered in A1 a near ideal problem to use to demonstrate your new-found ability to use system calls to take advantage of multi-processing within the operating system.

This assignment will build on work that you did in A1 and A2. You will use the same KNN algorithm as A1, but will use multiple processes to compute the classification of the test images in parallel.

There is one further way that we will build on the previous assignments: we will write a KNN classifier that allows us to dynamically change the distance measure used to define the "closeness" of two images through the use of function pointers.

Program Details

The program you will write takes several command line arguments. Some of these arguments have default values. See the starter code for exactly how to use these arguments:

- The value k of the kNN algorithm

- The number of children that the parent process will create to classify the images in the training set.

- An string that specifies the distance metric that your code will use.

- An option "-v" (verbose) to allow printing of extra debugging information. You can see how this option is used in the starter code, and can add additional print statements. Be careful to ensure that any additional debugging statements you add enhance the readability of your program.

- The name of the file containing the training set of images

- The name of the file containing the test set of images

The only output that the program will produce (when -v is not used) is the accuracy achieved by kNN when classifying the images in the test data set.

Distance Measures in K-Nearest Neighbor Classification

The standard distance measure used in nearest-neighbor classification is the Euclidean distance, which reads

$$d(x,y) = \sqrt{\sum_{j=0}^{D-1}(x_j - y_j)^2} \quad .$$

In the case of our images, the values x[j] and y[j] correspond to pixels from the data sets that either take the value 0 or 255. This distance measure is well motivated geometrically since it gives the distance between two points in a (Euclidean) D-dimensional space. However, for our data sets, it's unclear whether this distance measure truly reflects the notion of what it means for two images to be closely related. Conceptually the most direct way to investigate this is to change the Euclidean distance to another distance measure that better reflects the geometry of the training data set.

You will be comparing the performance of KNN classifiers using the Euclidean distance and an alternative distance metric, known as the cosine distance.

The cosine distance is motivated using the standard expression for the dot-product between two vectors

$$x \cdot y = \|x\|\|y\| \cos(\theta) \quad ,$$

where the ||x|| and ||y|| terms refer the length of the vectors (using Pythagoras). The cosine distance uses the above formula to observe that the angle theta can be used as a measure of the similarity between two vectors. Specifically, assume x,y are vectors of non-negative real numbers. Then if x=y, we find that theta is 0. In contrast, if x and y are orthogonal then theta=pi/2. This means that theta itself, which is a measure of the angle between two vectors, can be used as a measure of the degree of similarity between two vectors. Specifically,

$$d_{cos}(x,y) = \frac{2}{\pi}\cos^{-1}\left(\frac{\sum_{j=0}^{D-1} x_j y_j}{\sqrt{\sum_{j=0}^{D-1} x_j^2}\sqrt{\sum_{j=0}^{D-1} y_j^2}}\right)$$

The factor of 2/pi is introduced just to make the cosine distance (in our case) bounded between 0 and 1, but doesn't serve any deeper purpose for us.

Our aim is to use function pointers to write general code that allows you to swap out the distance function used without having to rewrite any of the code you wrote for the classifier. This is important in ML applications because it is important to play with different distance metrics (or feature maps) when tuning classifiers like KNN and so engineering code so that it becomes easy to test such changes rapidly and repeatably is essential for developing a good classifier for a dataset, making this refactoring exercise an excellent demonstration of the power of function pointers.

The Format of the Input Data

The two files that comprise the training and the test data sets are in the same binary format as in A2. Several datasets can be found on teach.cs in `/u/csc209h/winter/pub/datasets/a2_datasets`

Until you have some confidence in the correctness of your program, you should use smaller datasets for testing, especially if you are testing your work on teach.cs. This will avoid overloading the server, and you will be able to see results faster.

Analyzing the speedup from parallelization

An interesting parameter when running your program is the number of children the parent uses. In an ideal world, using two parallel processes instead of one sequential process will allow us to classify the same number of images in half the time. Or more generally, using n times the number of processing lets us finish classification n times faster. Run your program for n= 2, 3, 4, 5, .. 10 child processes and report the time classification took for each value of n. The starter code includes functions for timing how long the classification of all test images took. Do you observe a linear speed-up, where k child processes finish the job k times faster? If not, why do you think that is the case? (You do not need to submit your observations.)

Important Hints

Here are a few things to remember as you implement and test the functions:

1. When you look at the starter code you will see a few lines that look like the following. These lines are there to prevent compiler warnings about unused variables, and should be removed when you make use of the variables.

```
(void)K;
(void)dist_metric;
(void)num_procs;
```

```
```

2. Test thoroughly before submitting a final version. We are using automated testing tools, so your program must compile and run according to the specifications. In particular, your program must produce output in the specified format.
3. As always, build up functionality one small piece at a time.
4. Check for errors on every system or library call that is not a print. Send any potential error messages to stderr, use `perror` correctly, and exit when appropriate for the error.
5. Be very careful to initialize pointers to NULL and to make sure that you copy any strings that you use.
6. Close all pipes that you do not need to communicate between the parent process and its children.

Submission and Marking

Your program must compile on teach.cs, so remember to test it there before submission. We will be using the same compile flags in the `Makefile`.

Your submission minimally needs to include 3 files: `knn.c`, `classifier.c` and `Makefile`. Do not commit any datasets or executable files.

Your program should not produce any error or warning messages when compiled. As with previous assignments, programs that do not compile will receive a 0. Programs that produce warning messages will be penalized.

Submit your work by committing and pushing the files to the MarkUs git repo.