Course Project - MPCS 51040

November 21, 2016

1 General Instructions

1.1 Compiling

Your code must compile with `-std=c11 -Wall -Werror -pedantic`. There should be no warnings or errors when compiling with GCC (as installed on `linux.cs.uchicago.edu`). **Your code should be error and warning free when executed under valgrind.**

1.2 Handing in

To hand in your project, you need to commit all required files (including Makefile) to your personal git repository.

Make a subdirectory called 'project' and place your files under that directory. Don't forget to commit and push your files! You can check on `http://mit.cs.uchicago.edu` to make sure all files were committed to the repository correctly.

The deadline for this project is Friday December 9, 2016. To grade the project, the contents of your repository at exactly the deadline with be considered. Changes made after the deadline are not take into account.

1.3 Grading

Your code will be graded based on the following points (in order of descending importance):

- Correctness of the C code; there should be no compiler errors or warnings when compiling as described in 1.1. There should be no memory leaks or other problems (such as those detected by valgrind).
- Correctness of the solution. Your code should implement the required functionality, as specified in this document.
- Test the correctness of your implementations for all functions that have a prototype in your headers
- Code documentation. Properly documented code will help understand and grade your work.
- Code quality: your code should be easy to read and follow accepted good practices (**avoid code duplication, use functions to structure your programs, group logically related function into a single .c/.h file, ...**). Use of function pointers, as specified in later section of the document, will make the code modular and avoid unnecessary if/else blocks.
- Proper use of Makefile with (default, test and clean targets)
- Efficiency: your code should not use more resources (time or space) than needed

2 Project Description - Optical Character Recognition(OCR)

2.1 Primer on k - Nearest Neighbors

k - Nearest Neighbor[1] is a simple machine learning algorithm that achieves classification by comparing its sameness to k closest neighbors. In figure 1, we have $k = 3$ and 0s/1s are labeled training examples. X_1 and X_2 are the unseen test data points. As we can see X_1 looks at 3 of its closest neighbors. Since 2 of its 3 neighbors are labeled as 1, X_1 is labeled as class 1. Similarly, X_2 looks at 3 of its closest neighbors and all its neighbors are labeled as class 0 and hence X_2 is labeled as 0.

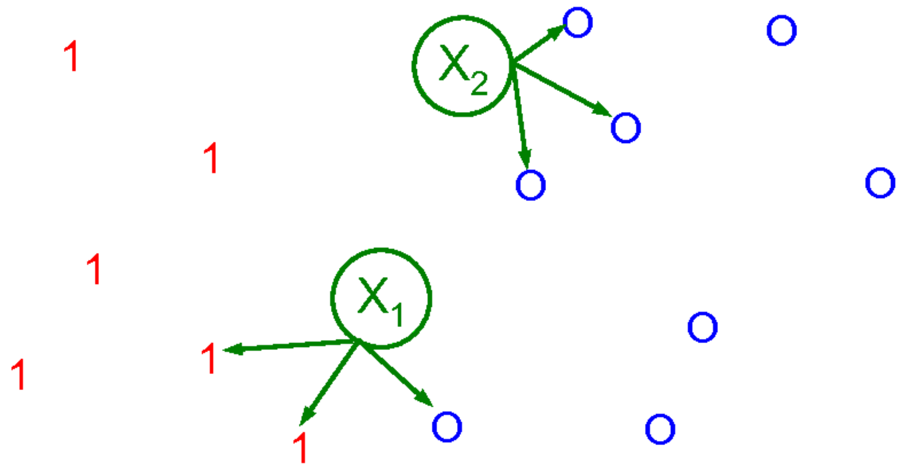

Figure 1: With K=3

As noted above, *spatial distance* here is only used as a convenient way of representing degree of sameness between the two points. Sameness between two images can be established based on several different schemes, by adjusting varying the distance function used. We list a few:

2.1.1 Euclidean Distance (euclid)

One way to compute distance is using the Euclidean distance formula. Given two points X_p and X_q of N dimensions, the Euclidean distance d is given by:

$$d = \sqrt{(X_{p1} - X_{q1})^2 + ... + (X_{pN} - X_{qN})^2}$$

For example, consider the 2-dimensional plane the figure 1. If we want to compute the distance between X_1,given by its 2-d coordinates (x_{x1}, y_{x1}) the blue circle closest to X_1. Let us call this X_o and is given by its 2-d coordinates (x_{xo}, y_{xo}). Then the distance between them can be computed as:

$$d_{x_1,x_o} = \sqrt{(x_{x1} - x_o)^2 + (y_{x1} - y_o)^2} \tag{1}$$

An each MNIST image with x pixels can be considered as a point in x dimensional space. For example, an image with 28×28 pixels, can be considered as point with 784 dimensions.

[1]For more details on nearest neighbor technique, we refer you to Introduction to Statistical Learning [1]. A PDF version of the book is freely available on the author's website.

2.1.2 Dimensionality Reduction [2] (reduced)

We can do a simple reduction of the image to a single integer value by adding the value(0 - 255) in each of the 784 pixels. The absolute difference between the reduced values of two images can be used to indicate degree of sameness. Pairs with least absolute value are considered similar (or closer together). For e.g. The number **8** will have more black pixels than **1** and hence should correspond to a higher reduced value. Going by this scheme, 8s should be close to each other and 1s should be close to each other.

2.1.3 Downsampling the images [2] (downsample)

Picking every other pixel, effectively reducing the number of pixels in the image by half, followed by computing the euclidian distance on the reduced set of pixels.

2.1.4 Cropping the image (crop)

If you consider that the image is centered and there is a certain amount of padding around the image (say a layer of 4-pixel wide empty border), then discarding that border will give you a cropped image, thereby reducing the number of dimensions for Euclidean distance computation.

2.1.5 Count pixels above a certain threshold (threshold)

Count the number of pixels that have a value between a certain threshold and 255. Let us say that we pick a fixed threshold of 127. The distance function can then count the number of pixels that contain a value $\geq$ than 127. Images that have a similar number of pixels satisfying the threshold are considered to be closer together (or more similar).

These are only a few of the many schemes that can be used to compute degree of sameness between images. You are required to implement two schemes for the project. The first scheme should be based on the Euclidean distance and the second one can be a scheme of your choice (not required to choose one from the above listed schemes).

2.2 Distance interface requirements

You are **required** to implement at least two distance schemes. One of them **must be** the euclidian distance scheme. It should be easy to add additional distance schemes to your code, without needing to change the code that uses the distance function.

The easiest way to obtain this functionality in C is using function pointers. So, you should define a type representing a pointer to a distance function (name this type `distance_t`). This is similar to homework 6, where using function pointers allowed the game engine to be oblivious of the exact agent type, and where adding a new agent type to the code does not affect the game engine.

For this to work correctly, you will also need a *factory function* which takes a string and returns the appropriate distance function to the caller.

Your factory/lookup function will have the following prototype:

```
distance_t create_distance_function(const char * schemename);
```

Place all the code related to the distance functions in `distance.c` and `distance.h`, only exposing what is needed in the header. **You also need to create a unit test (using cunit) which validates your factory function and your distance functions.** For example, you can calculate a few instances of each implemented distance function by hand and validate using your unit test that the result of calling the distance function results in the computed value.

2.3 k-NN Algorithm

For the k-NN algorithm, we will have:

- a set of training images, for which know the corresponding label.
- a set of test images, for which we are trying to determine the label and track the accuracy of the classification.

- a distance metric (represented by a `distance_t` variable).
- a parameter k

Note that while we do know the label for the test images, we will only use this knowledge to determine if the k-NN algorithm was able to correctly identify the image or not.

Your implementation of the k-NN algorithm should do the following, *for each image in the collection of test images*:

- Calculate the distance between the selected image of the test set and all the images of the training set.
- For the k images – for which we know the corresponding label – closest to the unknown image, find the label which occurred the most. If all labels occurred the same number of times, pick the label corresponding to the closest image.

For each test image, determine if the guess produced by the k-NN algorithm matches the known label of the image. We define the accuracy as:

$$\text{Accuracy}_{knn} = \frac{\text{number of correctly predicted labels}}{\text{total number of labels}} \times 100$$

Your program should take, as command line arguments, the name of the training data set as well as the name as the test dataset. In addition, for the training set, it should be possible to specify set size(N) at the time of program invocation (using the command line). If N is less than the number of images in the dataset, your code is required to pick **a random sampling** of N instances from the specified training data set and pass it to the kNN function. (Reservoir sampling (`https://en.wikipedia.org/wiki/Reservoir_sampling`) provides an efficient way of doing so.) If N is larger than the number of messages, your program should report an error and exit.

Your code should work with any dataset and any size image – do not assume images are always 28x28!

3 Tasks & Deliverables

3.1 MNIST dataset/image library

- You must adapt your mnist library from homework 5.
- You must add a function that takes a number $N \leq$ total number of images and returns N randomly selected images from the dataset. This could either be at open time (for example `mnist_open_sample`, taking a dataset name and the number of images to sample from the dataset) or a function which, given a dataset handle and a number N, reduces the number of images in the dataset (in a random fashion) to N images. These functions do not need to preserve image order.

  The function must indicate an error if N is larger than the number of items in the dataset.
- You need to provide a (cunit) unit test for the new functionality (you can do this by adapting your existing mnist unit test). Make sure to check at least the following items:
  - that the reduced dataset indeed contains N items
  - that the reduced dataset is a random subset. One way to verify this is by generating 2 or more N element subsets and ensuring they contain (at least some) different images. (While it is possible that two random samplings pick the same images, the chances of doing so for $N \ll dataset\_size$ are vanishingly small.)

3.2 Distance library

Create a library (header and implementation) which makes it possible to:

- obtain a pointer to a distance function when given a string naming the distance function desired. **The string identifying the distance function must match the one from section 2.1 in this document, specified between '()' for each distance function.** For example, for euclidian distance, the string must be 'euclid'.
- obtain a (textual) description naming each of the distance functions supported by the library. This description needs to include at least the name identifying each implemented distance function (i.e. euclid).

Your library must support at least two distance metrics, and one of them must be euclidean distance. You will receive extra credit if you implement all distance metrics described in this document.

You must provide a **standalone** unit test (written using cunit) which must test at least:

- the public functions of your library (public functions are all functions that have prototypes in your header file).
- each implemented distance function. Your unit test must verify the resulting distance for a few known images, for which you calculated the distance by hand.

3.3 k - Nearest Neighbor logic

- A function which, given a training dataset, test image, k and distance function returns the best guess for the corresponding label.
- A standalone unit test (using cunit) for this logic. At the very least your test must ensure the k-Nearest Neighbor logic properly detects invalid inputs to the function (for example, but not limited to, $k = 0$, or an empty training dataset).

Note that constructing proper unit tests for the actual classification logic is not trivial. Do the best you can.

3.4 OCR application

The OCR application will be using your mnist library and k-NN code and make their functionality accessible via the command line.

As part of this task, **also include a `README` file** (named that way) which briefly describes the internal structure of your code and the design choices you made while solving this assignment.

Update the makefile to include a target called `ocr`, which builds the program (named `ocr`).

Make sure that your program complies with the following requirements:

- Program must not crash for k=0 or training size = 0
- Program must take command line arguments for the values of *train-dataset-name*, *training-size*, *test-dataset-name*, *k*, *distance-scheme* **in this order**.
- If command the above options are not passed to the command line (fewer or no options are passed) the program must exit with a message:

  ```
  Usage: ./ocr [train-name] [train-size] [test-name] [k] [distance-scheme]
  ```

  ```
  The following distance schemes are support: <LIST>
  ```

  Replace `<LIST>` with the actual list of distance schemes supported by your code. Note that you must **not hardcode** this list in your main program; the information must be obtained from the distance library, so that this information is not duplicated.

- If train-size is 0, it must use all images from the set specified on the commandline (for training).
- Your program must print this exact format (including whitespace) to the screen with no extra characters or output:

  ```
  K = k
  [euclid] 123/60000 (  0.21%) 120/123 ( 97.56%)
  [euclid] 210/60000 (  0.35%) 200/210 ( 95.24%)
  ...
  [euclid] 60000/60000 (100.00%) 59123/60000 ( 98.54%)
  ```

  These are not actual numbers from an experiment. Your results will differ.

  Each line consists out of the name of the distance function, between '[]', followed by the number of images processed relative to the total number of images (percent, **with 2 digits after the decimal point** – see the `printf` man page), followed by the number of correctly recognized images relative to the total number of processed images (followed by the percentage, with 2 digits after the decimal point).

  Your program must output the current status once per second, as well as a final status output when all images have been processed.

  See the manpage for the `time.h` header for inspiration on how to do this. Alternatively, you could consult the C11 standard. Make sure your program uses *actual time*, not *processor time*, and that the functions you use are described in the C11 standard.

- To make analyzing the performance of each distance scheme easier, and to determine how the training set size and/or k parameter affect the accuracy, your program must also support the following:
  - If the word 'all' (without quotes) is specified for `train-size`, your program must act as if it was invoked with `train-size` set to 25, 50, 75 and 100% of the number of images in the training dataset.
  - If the word 'all' (without quotes) is specified for `k`, your program must act as if it was invoked with `k` set to 1, 5, 10, 15 and 20.
  - If the word 'all' (without quotes) is specified for `distance scheme`, your program must act as if it was invoked with every implemented distance function.

  Any combination of the above options must be supported. For example, if all is specified for distance scheme, and all is specified for k, your program must explore the different settings for k for each distance function, using the specified number of items (on the command line) for the number of images in the training data set.

  If any of the above is active, your program must (after running all the experiments) output the following **exactly** (in addition to the normal output):

  ```
  # distance k 15000 30000
  euclid 1 50.34 60.33
  euclid 5 53.34 79.22
  ...
  crop 1 10.12 9.15
  crop 5 12.91 29.56
  ...
  ```

  (your values will be different)

  In other words, for each distance function tested, for each value of k tested, there will be an output line. That line will contain the accuracy for each training set size tested (without trailing '

  If a subset of the training images is selected, that subset must be the same for each test.

3.5 Makefile

You are required to provide a single makefile capable of building all the code and tests. Please commit all required files to build your project to your repository (so that a single `make` command issued in your project directory will work).

Ensure that your Makefile is usable on linux.cs.uchicago.edu and **properly takes header dependencies into account**. It might be a good idea to checkout your repository in a new directory and try to build in that directory to ensure that all required files were properly committed.

At least the following targets must be supported:

clean Remove all object files and binaries built by the other targets.

all Build the `ocr` program and all unit tests. This must be the default target.

test Build **and run** the unit tests.

ocr The ocr program.

4 Methodology

It might be helpful to follow the approach below:

1. MNIST module
   (a) Copy `mnist.h/mnist.c` from HW5 and adapt it to project requirements
   (b) Copy all relevant unittests
   (c) Add the extra mnist library functionalities
   (d) Add relevant unit tests
2. Distance metric module
   (a) Declare the distance function prototypes in .h file and implement them in their corresponding .c file
   (b) Create distance.h, declare function pointer typedef
   (c) Add the factory function
   (d) Write unit tests for all distance functions. Use factory function to create distance functions and test correctness.
3. k-NN module
   (a) Complete knn.h and knn.c
   (b) Test with unittests
4. Main module
   (a) Read input argument and run prediction as per the arguments
   (b) Format the output according to the previous section
   (c) Check corner cases, crashes, memory leaks etc.

References

[1] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. [Online]. Available: http://www-bcf.usc.edu/~gareth/ISL

[2] https://www.classes.cs.uchicago.edu/archive/2013/spring/12300-1/pa/pa1/.