# Project 1

**Due**: Jun 20, before midnight.

**Important Reminder**: As per the course Academic Honesty Policy, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first provides the aims of this project. It then covers some necessary background for the project and lists the requirements as explicitly as possible. This is followed by hints about how these requirements can be met. Finally, it describes how it can be submitted.

## Aims

The aims of this project are as follows:

- To verify that you have GUI access to your VM.
- To get you started with programming in nodejs and JavaScript.
- To implement an application for optical character recognition OCR using the k-nearest neighbors algorithm.

## Background

In this project, we will do OCR for handwritten digits using the data from the MNIST database. This data consists of 28x28 grayscale `0-255` images of handwritten digits. Each image is labeled with a byte having value `0-9` depending on the digit the image represents.

The images are separated into training images and test images. The image data is split into pixel data and label data. These two splits result in 4 files:

**Training Data File and Training Labels File**
: The number of entries in both files must match (there are currently 60,000 images in the MNIST training database).

**Testing Data File and Testing Labels File**
: The number of entries in both files must match (there are currently 10,000 images in the MNIST test database).

This results in a total of 4 files.

A simple way of recognizing what character a test image represents is to do a pixel-by-pixel comparison of the test image with multiple training images and then choosing the label of the training image which is **closest** to the test image.

Specifically, we regard each 28 x 28 image as a point in a $28 \times 28 = 784$-dimensional space. For a test image, we assign it the most frequently occurring label within the $k$-nearest training images to the test image. To compute the distance between 2 images, we simply use the cartesian distance between the points represented by each image in the 784-dimensional space.

Hence our program will work as follows:

1. Read in the training images and test images from the MNIST database, along with their labels.
2. Determine the label of a test image as follows:
    a. Sort the training images by their distance from the test image.
    b. Choose the most commonly occuring label of the *k*-sized prefix of the sorted list as the label for the test image.

In this project you will be writing:

1. A JavaScript function to parse byte arrays in a parameterized MNIST format.
2. A function to determine the label and index for a training image which has the most commonly occcurring label among the k-nearest neighbors of the test image.

# Requirements

Add a **`submit/prj1-sol`** directory to your github **`i?44`** repository such that typing **`npm ci`** within that directory will install create a nodejs binary program which is executed with usage:

```
$ ./index.mjs
usage: index.mjs [OPTIONS] MNIST_DATA_DIR
    where OPTIONS are:
    --k=K
      hyper-parameter for KNN algorithm (default: 3)
    --index0=INDEX0
      index of first test image to be classified (default: 0)
    --index1=INDEX1
      index one beyond that of last test image to be classified
      (default: index0 + 200)
    --verbose
      report classification result for every test image
    --help
      print this help message
```

Note that the required **`MNIST_DATA_DIR`** argument should specify a directory containing uncompressed MNIST data using the standard MNIST filenames.

This command-line behavior is already implemented by a provided **`main()`** function in main.mjs. What you need to do is to implement the functions specified in parse-images.mjs and knn.mjs according to the specifications given in types.d.ts such that they pass all tests provided in the prj1-sol/test directory.

Additionally, your **`submit/prj1-sol`** **must** contain a **`vm.png`** image file to verify that you have set up your VM correctly. Specifically, the image must show an x2go client window running on your VM. The captured x2go window must show a terminal window containing the output of the following commands:

```
$ hostname; hostname -I
$ ls ~/projects
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

You will loose 10 points if you do not include the **`vm.png`**.

# Provided Files

The prj1-sol directory contains the following files:

**parse-images.mjs**
A skeleton file for the `parseImages()` function you are required to implement. You will need to edit this file.

**knn.mjs**
A skeleton file for the `knn()` function you are required to implement. You will need to edit this file.

**types.d.ts**
TypeScript documentation for the two functions you are required to implement. You should not change this file.

**main.mjs**
This file provides the `main()` function for your program. It handles the command-line arguments and loads the image data using the first of the two functions you are required to implement. It loops over the specified test images, calling the second of your two functions to label the image. It determines whether or not the label is correct, printing out information if not. Finally, it prints out the overall success rate.

**index.mjs**
A trivial driver for `main()`.

**README**
A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

Additionally, the extras directory contain files which you should not submit:

**fail-images.py**
A Python script into which you can pipe the output of your program in order to display the images which it fails to classify correctly.

**300-400.log**
A sample output of the program when run with `--index0=300` and `--index1=400` options.

Note that your output may differ from that given in this log since your implementation may break ties differently than the implementation which generated this log.

**mnist-download.sh**
A trivial shell script which will download the MNIST data files.

# Hints

This section is not prescriptive in that you may choose to ignore it as long as you meet all the project requirements.

The following points are worth noting:

- Both functions you are required to implement must return a Result which wraps either the success result or any encountered errors.
- Note that the specifications for `parseImages()` require you to handle byte arrays which are formatted using a generalization of the MNIST format. In particular, the magic numbers, number of images, number of labels and image dimensions are specified by the first `imageSpecs` argument.
- Since the KNN algorithm can be used for classifying general feature vectors, the specs for the `knn()` function do not refer to image classification but to finding a label for a feature vector given a set of

"training" labeled feature vectors.

- The `knn()` function is required to return not only the label of the closest labeled feature but also its index. This facilitates comparing the returned training image with test image.
- Since you have already been provided with tests, you will be practising Test-Driven Development (TDD).
    - If you would like to run only a particular test suite, change the **`describe(...)`** for that test suite to **`describe.only(...)`**.
    - If you would like to run only a particular test, change the **`it(...)`** for that test to **`it.only(...)`**.

You may proceed as follows:

1. Review the Practical Considerations slides covered in class.
2. If you have not already done so, set up your course VM as per the instructions specified in the VM Setup and Git Setup documents.
3. Install the python modules you will need to run the provided fail-images.py script:

    ```
    # will prompt for password
    $ sudo apt install -y python3-pip
    # all following will install in ~/.local
    $ pip3 install tensorflow
    $ pip3 install keras
    $ pip3 install keras-datasets #for MNIST data
    $ pip3 install matplotlib
    ```

    Once this is done, you should be able to run the script on the provided 300-400.log.

    ```
    $ cd ~/cs544/projects/prj1/extras/
    $ cat 300-400.log | ./fail-images.py
    ```

    Ignore the warnings about a missing CUDA library. You should get a popup displaying the incorrectly predicted training image and the test image. Type **`q`** into this window to display the next set of failing images.
4. Set up a shell variable to point to the project's **`extras`** directory:

    ```
    $ extras=$HOME/cs544/projects/prj1/extras
    ```

    This will make it possible for you to avoid typing in long paths.
5. Create a ~/i?44/data directory to hold the MNIST data which you will use for this and subsequent projects.

    ```
    $ mkdir -p ~/i?44/data
    $ cd ~/i?44/data
    $ $extras/mnist-download.sh
    $ ls -a
    ```

    You should see the four **`*-ubyte`** files you have downloaded plus a **`.gitignore`** file. The **`.gitignore`** file is set up to avoid uploading the **`*-ubyte`** files to your github repository since they are large and readily available.
6. Create a shell variable shortcut to point to your data directory:

    ```
    $ data=$HOME/i?44/data
    ```

7. Start your project by creating a new **prj1-sol** branch of your **i444** or **i544** directory corresponding to your github repository. Then copy over all the provided files:

```
$ cd ~/i?44
$ git checkout -b prj1-sol #create new branch
$ mkdir -p submit          #ensure you have a submit dir
$ cd submit                #enter project dir
$ cp -r ~/cs544/projects/prj1/prj1-sol .  #copy provided files
$ cd prj1-sol              #change over to new project dir
```

8. Commit into git:

```
$ git add .   #add contents of directory to git
$ git commit -m 'started prj1' #commit locally
$ git push -u origin prj1-sol  #push branch with changes
                               #to github
```

   [To avoid loosing work, you should get into the habit of periodically **commit**'ting your work and **push**'ing it to github.]

9. Replace the **XXX** entries in the **README** template and commit to github:

```
$ git commit -a -m 'updated README'
$ git push
```

10. Capture an image to validate that you have set up your course VM as instructed. Type in the following commands to a terminal within your **x2go** window:

```
$ hostname; hostname -I
$ ls ~/projects
$ ls ~/cs544
$ ls ~/i?44
$ crontab -l | tail -3
```

    Use an image capture program on your **workstation** to capture an image of your **x2go** window into a file **vm.png**. The captured image should show the terminal window containing the output of the above commands as part of the **x2go** window. Move the **vm.png** file from your workstation to your VM's **~/i?44/submit/prj1-sol** directory (you can use **scp**; if your workstation cannot connect directly to your VM, then do a 2-step copy using **remote.cs** as an intermediate).

    Add, commit and push the **vm.png** file.

11. Initialize your **prj1-sol** directory as an npm project:

```
$ cd ~/i?44/submit/prj1-sol
$ npm init -y
```

    Install the mocha and chai libraries required by the provided tests:

```
$ npm install --save-dev mocha chai
```

    The **--save-dev** option lets npm know that these libraries will only be used during development time (for testing) and will not be needed to actually run the project.

    Also install the **cs544-js-utils** library which will be needed both during development and when running your project:

```
$ npm install ../../../cs544/lib/cs544-js-utils
```

The above steps should create a `package.json` and `package-lock.json` file as well as a `node_modules` directory which will contain all the downloaded files. The `*.json` files should be committed to your `i?44` github repository, but the `node_modules` directory should not be committed (the provided .gitignore is set up to not commit it).

You should be able to run your project sufficiently to get an usage message:

```
$ ./index.mjs
usage: index.mjs [OPTIONS] MNIST_DATA_DIR
    where OPTIONS are:
    --k=K
      hyper-parameter for KNN algorithm (default: 3)
    --index0=INDEX0
      index of first test image to be classified (default: 0)
    --index1=INDEX1
      index one beyond that of last test image to be classified
      (default: index0 + 200)
    --verbose
      report classification result for every test image
    --help
      print this help message
```

You should also be able to run the tests using `npm test`, but they will all fail since you have to yet implement your code.

Add in your `*.json` files to git, commit and push.

12. Start work on your project by implementing the `parseImages()` function in your `src/parse-images.mjs` file.

    The two function arguments are parallel objects with both having keys `images` and `labels`. The values in the first argument provide the header specs and the values in the second argument provide `Uint8Array` byte arrays.

    a. Look at the MDN documentation on ArrayBuffer, Uint8Array and DataView.

    a. Write an auxiliary function to process the headers given an array of `HeaderSpec[]` and a `Uint8Array` byte array. You should have this function return a `Result<Headers, RestArray>` where `Headers` is an object mapping the header names to their integer values and `Uint8Array RestArray` is what is left over from the input bytes array after the headers.

       The implementation of this function will simply loop through the HeaderSpec[] array while tracking an offset in the bytes array. For each spec it should convert the 4 bytes at the offset into a local integer using DataView:getInt32(). If the spec provides a `value`, then your function should verify that the actual value matches, signalling a `BAD_VAL` error (using `err()`) if that is not the case.

    b. Have your `parseImages()` function call your auxiliary header-processing function for both `images` and `labels`. If the auxiliary function returns an error `Result`, then propagate that error `Result` back to its caller.

    c. Verify consistency between the returned `Headers`'s and `RestArray`'s, returning a `BAD_FMT` error `Result` if there is an inconsistency.

    d. Split the `images` and `labels` `RestArray`'s suitably, accumulating them as `LabeledFeatures` and return using `ok()`.

    Iterate your implementation until all the tests for `parseImages()` test suite pass.

13. Implement the required `knn()` function.

a. Define an auxiliary function to compute the cartesian distance between two feature vectors. Since we are merely comparing distances, it is not necessary to extract the square root. Whenever we use the term "distance" in the sequel, we are really referring to this distance-square.

   Note that the maximum of the square of the distances is$255^2 \times 28 \times 28$.

b. Start implementing your `knn()` function by looping through all the labeled features accumulating an array `distLabelIndexes` containing `{dist, label, index}` objects where `dist` is the distance of the labeled feature vector from the test feature vector and `label` and `index` are the `label` and `index` of the labeled feature vector.

c. Sort your `distLabelIndexes` by distance.

d. Extract the $k$-prefix of your sorted `distLabelIndexes` array.

e. Compute the most common label in the $k$-prefix as well as its index. This is best delegated to a second auxiliary function:

   i. Use a `counts` object to track label counts as well as `maxCount`, `maxLabel` and `maxIndex` variables.

   ii. Loop through the $k$-prefix updating `counts`. Whenever the `counts` for a particular label exceeds the current `maxCount`, update `maxCount` as well as `maxLabel` and `maxIndex`. `# Return`

   iii. Return `[ maxLabel, maxIndex ]`.

Iterate your implementation until all tests in the `knn()` test suite pass.

14. Run the provided `./index.mjs` on the MNIST data. It is best that you do not run more than around 200 test images at a time to avoid exceeding CPU time limits (which can trigger a reboot of your VM).

15. Iterate until you meet all requirements.

It is always a good idea to keep committing your project to github periodically to ensure that you do not accidentally lose work.

# Submission

Before submitting your project, update your README to specify the status of your project. Document any known issues.

Use the procedure provided in the git setup document to submit your project.

# References

This video was the inspiration for this project.