ECE 2400 Computer Systems Programming, Fall 2017
PA5: Handwriting Recognition Systems

School of Electrical and Computer Engineering
Cornell University

revision: 2017-11-26-09-04

1. Introduction

The fifth programming assignment is designed to combine all of the concepts you have learned throughout the semester and focus them by taking a step into the field of machine learning. In this assignment, you will design a handwriting recognition system that can classify handwritten numbers into ten classes, the digits from zero through nine, with high accuracy. To design your system, you will be working with specialized data structures and algorithms, and you will leverage your knowledge of C++ across multiple programming paradigms.

Machine learning encompasses the design of software programs that detect and learn features from inputs before generalizing what they learn and applying it to new contexts. A supervised machine learning model has two main phases. In the *training* phase, the model is provided with a large set of labeled data. For example, the dataset may contain millions of images from daily life, each with a corresponding label like "cat", "plant", or "boat". The model uses the training dataset to detect and learn the features that define a particular label class (e.g., cats will have ears and whiskers). In the *inference* phase, the model is provided with new inputs it has never seen before and then leverages what it has learned to predict their labels. For example, a model that is trained to recognize cats may be able to recognize new cats it has never seen before. Research in the field of machine learning today is driving innovation in many new fields, including object detection and decision making for autonomous vehicles, natural language processing, and even world-class board game playing.

In this assignment, we will focus on *handwriting recognition systems* based on the MNIST database of handwritten digits. The database is composed of 70,000 examples of 28×28 pixel images, organized into a training dataset of 60,000 images and a test dataset of 10,000 images for evaluation. The digits were handwritten by several hundred different writers ranging from average high school students to Census Bureau employees. This means that the legibility of the handwritten digits varies as well. The goal of the assignment is to design a system that can classify images of these handwritten digits with high accuracy into one of ten classes: the numbers zero through nine. Recall however that computers view images differently from the way we do as humans, and from the computer's point of view, these images are in fact each an array of 784 numbers ($28 * 28 = 784$). Can you imagine how to design a system that can correctly classify such images into digit classes?

You will implement various pattern recognition algorithms in this assignment including several variations of the *k-nearest neighbors* algorithm. You will also have room to be creative with your choice of algorithm as you improve the two primary metrics of a handwriting recognition system: accuracy and performance. While accuracy is important for correctness, performance is also critical to deliver a quick response. In the previous assignments, students were restricted from using standard libraries for common functions, but in this assignment we are allowing you to leverage any functionality available in the `std` namespace (e.g., `std::sort` and `std::vector`, etc.). Leveraging highly optimized code written by others will enormously increase your productivity! We will continue to leverage the Criterion framework for unit testing, TravisCI for continuous integration testing, and Codecov.io for code coverage analysis.

After your handwriting recognition systems are functional and verified, you will write a 2–4 page design report that describes your design for each implementation, discusses your testing strategy, and evaluates the accuracy and performance trade-offs between implementations across both the training and inference phases. **There is no incremental milestone for this assignment. The final code and report are all due at the end of the assignment. You should consult the programming assignment assessment rubric for more information about the expectations for all programming assignments and how they will be assessed.**

This handout assumes that you have read and understand the course tutorials. To get started, log in to an `ecelinux` machine, source the setup script, and clone your individual repository from GitHub:

```
% source setup-ece2400.sh
% mkdir -p ${HOME}/ece2400
% cd ${HOME}/ece2400
% git clone git@github.com:cornell-ece2400/<netid>
```

You should never fork your individual remote repository! If you need to work in isolation then use a branch within your individual remote repository. If you have already cloned your individual remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the assignment like this:

```
% cd ${HOME}/ece2400/<netid>
% git pull --rebase
% mkdir -p pa5-sys/build
% cd pa5-sys/build
% ../configure
% make check
```

For this assignment, you will work in the `pa5-sys` subproject, which includes the following files:

- `configure` – Configuration script to generate Makefile
- `Makefile.in` – Makefile template to run tests and eval
- `src/hrs-ones.h` – Header file for `HandwritingRecSysClassifyOnes`
- `src/hrs-ones.cc` – Source code for `HandwritingRecSysClassifyOnes`
- `src/hrs-ones-test.cc` – Test cases for `HandwritingRecSysClassifyOnes`
- `src/hrs-knn-brute.h` – Header file for `HandwritingRecSysKnnBruteForce`
- `src/hrs-knn-brute.cc` – Source code for `HandwritingRecSysKnnBruteForce`
- `src/hrs-knn-brute-test.cc` – Test cases for `HandwritingRecSysKnnBruteForce`
- `src/hrs-knn-kdtree.h` – Header file for `HandwritingRecSysKnnKdTree`
- `src/hrs-knn-kdtree.cc` – Source code for `HandwritingRecSysKnnKdTree`
- `src/hrs-knn-kdtree-test.cc` – Test cases for `HandwritingRecSysKnnKdTree`
- `src/hrs-alt.h` – Header file for `HandwritingRecSysAlt`
- `src/hrs-alt.cc` – Source code for `HandwritingRecSysAlt`
- `src/hrs-alt-test.cc` – Test cases for `HandwritingRecSysAlt`
- `src/kdtree.h` – Header file for `KdTree`
- `src/kdtree.inl` – Source code for `KdTree`
- `src/kdtree-test.cc` – Test cases for `KdTree`
- `src/kdtree-test-misc.h` – Supporting source code for `KdTree` tests
- `src/types.h` – Header file for image and label types

Figure 1: Four example MNIST images – Images are 28 × 28 pixels accompanied by a label.

- `src/types.inl` – Source code for image and label types
- `src/types.cc` – Source code for image and label types
- `src/types-test.cc` – Test cases for image and label types
- `src/types-adhoc.cc` – Adhoc program to familiarize with types
- `src/utils.h` – Header file for utility functions
- `src/utils.cc` – Source code for utility functions
- `src/utils-test.cc` – Test cases for utility functions
- `src/hrs.h` – Header file for `IHandwritingRecSys`
- `src/constants.h` – Header file with global constants
- `src/digits.dat` – Data file with selected test images
- `src/hrs-eval.cc` – Evaluation program for `IHandwritingRecSys`
- `scripts` – Scripts for build system and generating datasets

2. Design Description: Handwriting Recognition Systems

The MNIST database of handwritten digits is composed of 70,000 examples of 28 × 28 pixel images. Labels are provided for each image to tell us which digit is shown. Figure 1 shows several example images from the MNIST database. Notice how the edges of the handwritten digit are not well-defined. This is because each pixel is in grayscale and has an value in the range of [0, 255]. Higher numbers represent lighter shades (with 255 representing white), while lower numbers represent darker shades (with 0 representing black).

An image has 28 × 28 pixels and is therefore an array of 784 unsigned integers. A label can be: '0', '1', '2', etc..., up to '9'. Labels can be represented as `char` types. We encapsulate images into an `Image` class and labels into the `Label` class as shown in Figure 2. Notice that an `Image` encapsulates a `std::array` of type `unsigned int` (note that the `image_size` variable is defined to be 784 in `src/constants.h`), while a `label` just encapsulates a single `char`. Notice that a label of '?' marks a label as invalid. Finally, a `LabeledImage` is a class containing one `Image` and one `Label` as member fields. More detail can be found in the header file `src/types.h` and the implementation files.

To familiarize you more working with these types, we have provided a small ad-hoc program called `src/types-adhoc.cc`, and we have prepared a few assignment tasks that should take less than ten minutes to step through. You can run the ad-hoc program like this:

```
% cd ${HOME}/ece2400/<netid>
% mkdir -p pa5-sys/build
% cd pa5-sys/build
% ../configure
% make types-adhoc
% ./types-adhoc
```

```
class Image
{
 public:
  Image();
  Image( std::array<unsigned int, image_size>& data );

  void   print() const;
  size_t size()  const;

  ConstItr begin() const;
  ConstItr end()   const;

  unsigned int at        ( size_t x, size_t y ) const;
  unsigned int operator[]( size_t idx         ) const;

 private:
  std::array<unsigned int, image_size> m_data;
};
```

```
class Label
{
 public:
  static const char INVALID = '?';

  Label();
  Label( char label );

  void print() const;

  char get_label() const;
  void set_label( char label );

 private:
  char m_label;
};
```

Figure 2: Interface for `Image` and `Label` types – An `Image` encapsulates an array of unsigned integers for the 28×28 image, while a `Label` encapsulates a `char` to hold the label.

Take a few moments to complete these assignment tasks before moving on. After completing these assignment tasks, note in particular that the following member functions of the `Image`, `Label`, and `LabeledImage` classes are likely to be very useful:

- `operator[]` – Access the underlying array as a 1-D structure (high performance)
- `at` – Access the underlying array as a 2-D structure (lower performance with bounds checking)
- `print` – Print the `Image` or `LabeledImage` to the screen
- `get_label` – Get the label from a `Label` or `LabeledImage`
- `get_image` – Get a reference to the `Image` from a `LabeledImage`

In the following subsections, we will design handwriting recognition systems that manipulate `Image`, `Label`, and `LabeledImage` objects to classify images from the MNIST database.

Figure 3 shows the interface for the abstract base class `IHandwritingRecSys` (declared in `src/hrs.h`) which exposes the general interface for a handwriting recognition system. Notice that all handwriting recognition systems must define a `train` and a `classify` member function in order to adhere to the interface of an `IHandwritingRecSys`. The class diagram shown in Figure 4 illustrates the relationships between the `Image`, `Label`, and `LabeledImage` classes, and a separate class diagram illustrates the abstract base class called `IHandwritingRecSys` and the four classes of handwriting recognition systems that inherit from it.

In the following subsections, we will implement three handwriting recognition systems:

- `HandwritingRecSysClassifyOnes` – A lightweight algorithmic approach that classifies only '1's
- `HandwritingRecSysKnnBruteForce` – Uses brute-force k-nearest neighbors algorithm to classify
- `HandwritingRecSysKnnKdTree` – Approximate k-nearest neighbors using a k-d tree data structure

In your repo, you may also notice that there is a fourth system called `HandwritingRecSysAlt`. You are not required to implement this system. This placeholder system has been provided for you to

```
class IHandwritingRecSys
{
 public:
  virtual void  train    ( const std::vector<LabeledImage>& v ) = 0;
  virtual Label classify ( const Image& image                 ) = 0;
};
```

Figure 3: Interface for `IHandwritingRecSys` – An abstract handwriting recognition system (1) trains with a vector of `LabeledImage`'s and then (2) classifies new `Image`'s, returning the predicted `Label`.

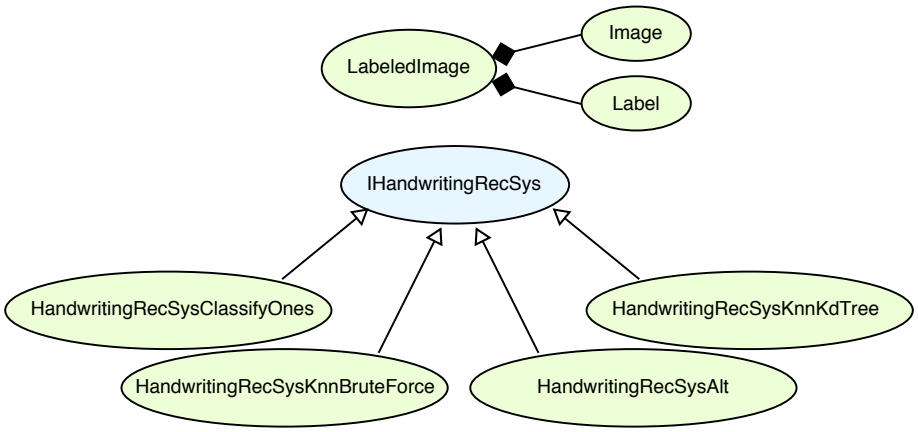

Figure 4: Class diagrams – A `LabeledImage` has-a `Label` and has-an `Image`. Each handwriting recognition system inherits the interface from the abstract base class `IHandwritingRecSys`.

implement a fourth system if you are very curious and want to try an open-ended approach for evaluation against the other systems. Extra credit will *not* be provided for a fourth system.

2.1. Classifying Ones – `HandwritingRecSysClassifyOnes`

Figure 5 shows a ones digit printed out as a two-dimensional array of unsigned integers in the range [0, 255]. You can print this out yourself by writing this code inside of `src/types-adhoc.cc`:

```
Image myimage( digit3_image ); // - digit3 in "digits.dat" is a '1'
myimage.print();               // - print the digit
```

Recall that you can access the data of any pixel of the `Image` by using the overloaded `operator[]` (one-dimensional array, high performance) or by using the `at` member function (more convenient as a two-dimensional matrix access, but slightly slower due to bounds checking).

Your task in this section is to think critically and design an algorithm that can detect ones and classify them correctly, while classifying all other digits as unknown by assigning `Label::INVALID` (see `src/types.h`). Write your implementation in `src/hrs-ones.cc`, in the `detect_ones` member function of the `HandwritingRecSysClassifyOnes` class. Tests for this handwriting recognition system have been provided to you in `src/hrs-ones-test.cc`. Try to pass each of the basic tests, and then slowly uncomment the larger tests to try more digits. Feel free to edit these test files to temporarily print out images by calling `print` on any image.

Here are the specifications for the functions declared in `src/hrs-ones.h`:

```
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   5  63 197   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  20 254 230  24   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254  48   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  20 254 255  48   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254  57   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  20 254 254 108   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0  16 239 254 143   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 143   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 143   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 162   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0 178 254 240   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0 113 254 240   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0  83 254 245  31   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0  79 254 246  38   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 214 254 150   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 241   8   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 240   2   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 144 254  82   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 230 247  40   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0 168 209  31   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figure 5: Image of a One – Can we design an algorithm that detects ones and classifies other digits as unknowns?

- `void train( const std::vector<LabeledImage>& v );`
  Do nothing, since this system uses an algorithmic approach that does not require training.
- `Label classify ( const Image& image );`
  Calls the helper function `detect_ones` to classify the given image.
- `Label detect_ones( const Image& image );`
  Detects whether the given image is a '1' and if so, returns a `Label` with a '1'. Otherwise, it returns a `Label` with the char `Label::INVALID`.

2.2. Brute-Force K-Nearest Neighbors – `HandwritingRecSysKnnBruteForce`

The k-nearest neighbors algorithm is a pattern recognition algorithm often used in machine learning.

Take a step back from handwritten digit recognition and consider classifying two-dimensional Point objects, which have dimensions x and y. Taking a step back from classifying digit labels, we will classify Points with one of two labels: either 'o' or 'x'. Figure 6 shows a number of Points spread out in the rectangular space bounded by (0, 0) on the lower left and (9, 9) on the upper right. Notice that the Points on the lower left are labeled 'o', and the Points on the upper right are labeled 'x'. A Point with an unknown label is at coordinate (2, 1). What do you think the label of this Point is? Why do you think your guess makes sense?

The k-nearest neighbors algorithm is based on the same simple intuition that you just exercised. The Points *closest* to the unknown Point are all labeled 'o'. If we consider one nearest neighbor ($k = 1$), we would only consider the label of the closest Point and classify the unknown Point as 'o'. If we consider three nearest neighbors ($k = 3$), we would consider the labels of the three closest Points and take the majority vote, still classifying the unknown Point as 'o'. Note that if we considered nine nearest neighbors ($k = 9$) and took a majority vote, we might classify this Point as 'x'.

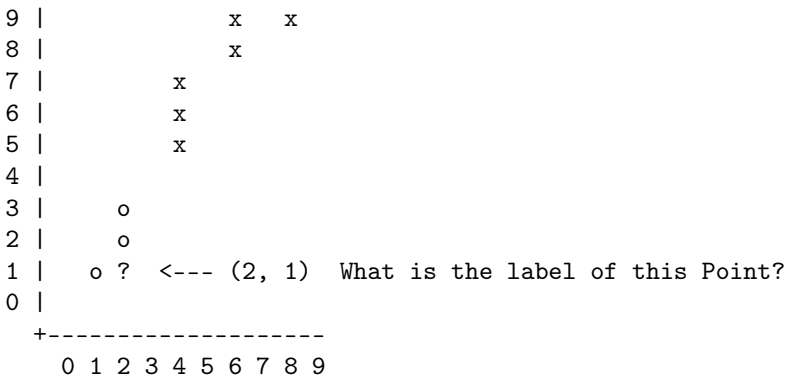

Figure 6: Example demonstrating the intuition behind the k-nearest neighbors algorithm – Each Point is labeled either as 'o' or as 'x'. The question mark is a Point with an unknown label. What do you think the label of this Point should be and why?

To make this example a little more concrete and closer to real life, consider the following:

- The x-axis indicates the number of wheels in a vehicle
- The y-axis indicates the number of passengers the vehicle can seat
- The 'o' label corresponds to "bicycle" (or "unicycle")
- The 'x' label corresponds to "car" (or "truck")

Unicycles have one wheel and seat one person. Bicycles have two wheels and can seat one or a few people. Cars have four wheels and can seat five or more passengers. Trucks have more wheels and can seat many more passengers. Our mystery Point is correctly classified as a bicycle because the closest neighbors are also all bicycles or unicycles.

More formally, we calculated the distance between Points in order to gauge the similarity between them. The distance metric typically used to compare two Points p and q is the *Euclidean distance*:

$$d(p,q) = d(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2}$$

We commonly use Euclidean distance metrics in two-dimensional and three-dimensional space.

The same formula can be applied to calculate the distance between MNIST images. An MNIST image can be viewed as a single point in 784-dimensional space. Look back to Figure 5 to see why. Consider each value from left to right, top to bottom. For each of the 784 values, consider that the i'th value is the distance along the i'th dimension. You can imagine that the same handwritten digits (e.g., a '2') drawn in slightly different ways will roughly exist as clusters in the 784-dimensional space. To classify any image of a handwritten digit with an unknown label, finding the nearest neighbor and taking its label can be a simple technique for classification with high accuracy.

Your task in this section is to implement the k-nearest neighbors algorithm with $k = 1$ (i.e., find the exact closest neighbor). Write your implementation in `src/hrs-knn-brute.cc`, in the `knn` member function of the `HandwritingRecSysKnnBruteForce` class. Notice that the `train` member function has already been written for you and essentially saves the training data inside of a vector for use by your knn algorithm. Tests for this handwriting recognition system have been provided to you in `src/hrs-knn-brute-test.cc`. Try to pass each of the basic tests, and then slowly uncomment the

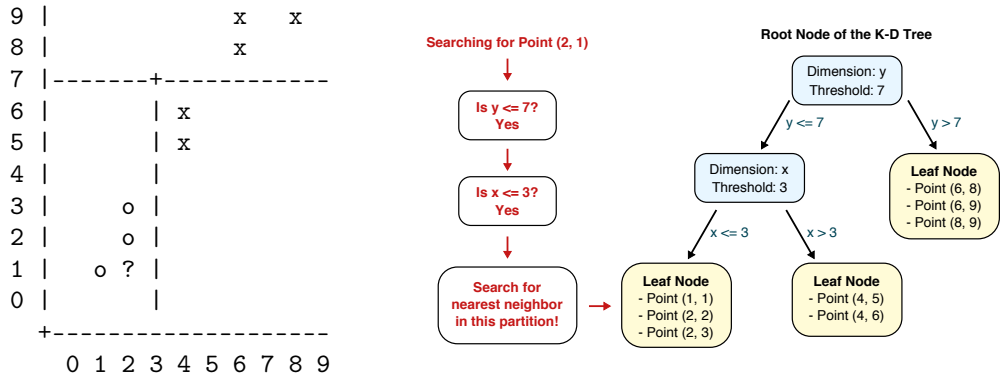

Figure 7: Example demonstrating partitions in a k-d tree – (*Left*) Two partitions are shown: one at $y = 7$ and one at $x = 3$. A nearest neighbor search for the point at coordinate (2, 1) using this k-d tree only needs to search the three points within the lower left partition for its nearest neighbor, rather than searching the entire space which may contain many points. (*Right*) The k-d tree is shown in tree form. To search for the point at (2, 1), we walk down the tree and end up inside the left-most leaf node, which we then search for the nearest neighbor.

larger tests to try more digits. Feel free to edit these test files to temporarily print out images by calling `print` on any image.

Here are the specifications for the functions declared in `src/hrs-knn-brute.h`:

- `void train( const std::vector<LabeledImage>& v );`
  Saves all training data in the given vector `v` into the member field vector `m_training_set`.
- `Label classify ( const Image& image );`
  Calls the helper function `knn` to classify the given image.
- `Label knn( const Image& image );`
  Uses the k-nearest neighbors algorithm to search all of the LabeledImages in the training dataset for the exact closest neighbor of the given image. Uses the Euclidean distance to calculate the distance between images. Returns the label of the closest neighbor. If no neighbors exist, it returns a `Label` with the char `Label::INVALID`.

2.3. K-Nearest Neighbors with a K-D Tree – `HandwritingRecSysKnnKdTree`

A k-d tree is a space-partitioning binary tree often used to speed up searches in k-dimensional space. The k-d tree slices the space along any of the k dimensions and divides elements into smaller *partitions*. This allows a nearest neighbor search algorithm, for example, to "zoom in" on the partition with the highest potential to contain the nearest neighboors.

Figure 7 shows how points in a two-dimensional space are partitioned, first along the y-axis at $y = 7$ and then along the x-axis at $x = 3$, until each partition has at most three points. A nearest neighbor search for the point at coordinate (2, 1) using this k-d tree only needs to search the three points within the lower left partition for its nearest neighbor, rather than searching the entire space, which may contain many points. Note that searching all points can slow down computation unnecessarily, especially when we know that most points are very far away and cannot be the nearest neighbors. The k-d tree is shown in tree form on the right. There are five nodes, three of which are leaf nodes with data. The leaf nodes correspond directly to the three partitions shown on the left plot. Searching

the k-d tree for the unknown point at (2, 1) involves walking down the tree and eventually arriving at the leaf node corresponding to the three points within the lower left partition.

Two decisions are central to the design of a k-d tree:

- **Partition strategy**. Choosing which dimension to cut and where to draw the partition lines makes a direct impact on where elements end up in the final partitions. The goal of the k-d tree is to enable a search algorithm to quickly zoom in on the most relevant partitions of the space. Therefore certain cuts may be more expressive, while others may group elements in less meaningful ways.
- **Leaf size**. The leaf size is the maximum number of points per partition and can be carefully chosen to tune the performance and accuracy of a k-d tree. Smaller leaf sizes increase performance but often hurt accuracy. For example, which nearest neighbor will be found if we classify an unknown point at coordinate (2, 6) in Figure 7? On the other hand, large leaf sizes trade slower performance for higher accuracy. For example, your brute-force search from the previous section is basically a search over one large leaf that contains all of the training points. It is very slow but will find the exact nearest neighbor.

A great deal of the k-d tree code has already been written for you! There are five mini-tasks that you will complete. Some of the mini-tasks involve very light modifications (e.g., change one line of code), but they are still required to pass one or more of the basic tests (which are all provided for you). The partition mini-task is a bit more involved, but we provide a working example of a simple partition to get you started.

The k-d tree is templated across the number of dimensions K. Static polymorphism will allow the same k-d tree implementation to be used in two dimensions or in 784 dimensions. **In this assignment, you will test the k-d tree using a vector of Points in two dimensions. Once you convince yourself that the k-d tree is fully functional, you will use the k-d tree to organize 784-dimensional MNIST images to accelerate the k-nearest neighbors search in the context of a new handwriting recognition system.** Take a look at the interface in `src/kdtree.h` and notice that the implementation is also templated over the type `T` to be contained, an iterator type `Itr` over elements of type T, and the type of T along any particular dimension which we call type `V`.

```
template < unsigned int K, typename T, typename Itr, typename V >
class KdTree
```

Here are two instantiations of k-d trees that we will use in this assignment, along with concrete values for the template arguments to help you better understand what they represent:

- `KdTree< 2, Point, std::vector<Point>::iterator, int >`
  This k-d tree will be used for testing because it is easier for us to reason in two dimensions. This k-d tree organizes two-dimensional elements ($K = 2$), where each element is a Point ($T = Point$), where we iterate over Point objects using a standard vector iterator type, and where Point coordinates are integers ($V = int$).
- `KdTree< 784, LabeledImage, std::vector<LabeledImage>::iterator, unsigned int >`
  This k-d tree will be used for evaluation by organizing MNIST images in the context of the broader handwriting recognition system in order to accelerate the k-nearest neighbors algorithm. This k-d tree organizes 784-dimensional elements ($K = 784$), where each element is a LabeledImage ($T = LabeledImage$), where we iterate over LabeledImage objects using a standard vector iterator type, and where LabeledImage coordinates are unsigned integers ($V = unsignedint$). Recall that for a 784-dimensional MNIST image, a dimension is a grayscale pixel. Each coordinate is therefore in the range [0, 255]. This is why V is an unsigned int.

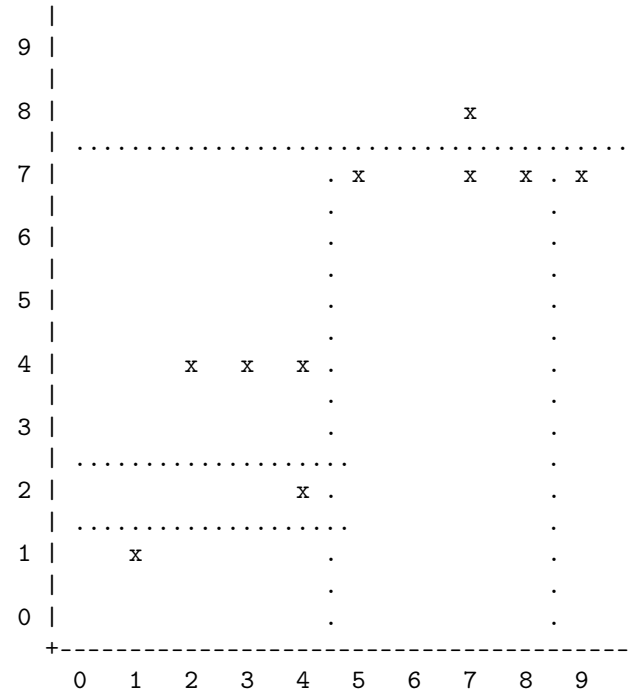

Figure 8: Generated printout of k-d tree partitions within a test case – You will be able to generate these printouts from your test cases to see how your k-d tree design decisions impact the partitioning.

Take a look at the interface for the k-d tree in `src/kdtree.h` and notice the member fields (e.g., leaf size parameter, root node) and member functions centered around either building the tree (e.g., choose dimension, partitioning) or searching an already-built tree. You should also take a look at the interface of the handwriting recognition system that uses the k-d tree in `src/hrs-knn-kdtree.h` and notice how a KdTree is declared as a member field. Open up `src/hrs-knn-kdtree.cc` and inspect how we build the k-d tree in the training phase from the training data, and how we search the k-d tree during inference to "zoom in" on a small subset of neighbors before running brute-force knn. See the `search` function in `src/kdtree.inl` for the short code snippet that walks down the k-d tree.

We have set up a highly visualizable test-driven development path for you to follow that will incrementally drive your design process to create a robust k-d tree implementation. Each test case has a `draw` function call that is commented out. If you uncomment it, then you will print out a visualization of the points and the partitions resulting from your k-d tree design decisions, as shown in Figure 8. At each step, you will implement the required functionality to pass a basic test. After passing each test, you will uncomment the next basic test (which will fail) and implement the next step in order to pass it. You are responsible for taking the following steps to complete the implementation:

- *Mini-task 1*: Fill in the code to create a leaf node in `src/kdtree.inl` in the `build_kernel` function. This is the base case of the function that recursively builds the k-d tree.
- *Mini-task 2*: Design a sorted median partition scheme in `src/kdtree.inl` in the `partition` function. A simple partition scheme has already been provided.
- *Mini-task 3*: Address a corner case in the partitioning scheme which can result in infinite recursion. Uncomment the corresponding basic test which hits the corner case, and modify your code to accommodate the corner case.

- *Mini-task 4*: Tweak the choice of dimension in `src/kdtree.inl` in the `choose_dimension` function. The provided implementation always chooses the 0th dimension (i.e., the x-axis) for partition. Modify this function to randomly pick across *K* dimensions.
- *Mini-task 5*: Lightly adapt your brute-force k-nearest neighbors implementation in `src/hrs-knn-kdtree.cc` so that it searches over a small vector instead of over all images in the training set.

Note that many of these steps may involve only a few lines of code, but this is only possible if you leverage the powerful functionality available within the C++ standard library (e.g., `std::sort` and `std::partition`)! We have provided additional comments in the code to help you learn to use the C++ standard library.

To get started, build and run `kdtree-test` in isolation like this:

```
% cd pa5-sys/build
% ../configure
% make kdtree-test
% ./kdtree-test
```

You should see the first basic test failing. You can uncomment the line that calls the `draw` function to see what the space looks like. After implementing *Mini-task 1*, uncomment the next basic test, uncomment the corresponding drawing function and see what the space looks like before beginning on *Mini-task 2*. We highly recommend using the drawing visualizations frequently to help you see how your design choices impact the partitioning in the k-d tree!

After passing all of the tests in `src/kdtree-test.cc`, you can then test the handwriting recognition system that uses the k-d tree by running the tests inside `src/hrs-knn-kdtree-test.cc`.

2.4. Final Notes

Remember that you may use any functionality available in the `std` namespace in this assignment! This means that you can use many powerful template containers (e.g., `std::vector` and `std::list`, etc.), as well as many useful algorithms as needed (e.g., `std::sort` and `std::swap`, etc.). If you are searching for some basic functionality, try posting on Piazza or doing a search on the internet to ask if such functionality already exists in a standard library.

Note that you are not allowed to import any third-party libraries, especially ones built for machine learning.

Also note that a few global constants including "image_size", "n_rows", and "n_cols" are defined in the `src/constants.h` header file. Take a brief look to familiarize yourself with them.

The `src/digits.dat` file is a convenient collection of single images taken from the MNIST dataset and stored in a `std::array` of 784 unsigned integers. No file reading or writing is required to use these images, and they are stored in the global namespace if you include "digits.dat". For each digit, there is a label as well. For example, "digit0_label" has label '5' and "digit0_image" has the corresponding data as a `std::array`. Take a look at this file and feel free to use these in your tests.

3. Testing Strategy

Unlike in previous programming assignments, a great deal of tests have already been provided for you. You can freely leverage the available tests to verify the functionality of your handwriting recognition systems. Remember however that your goal with respect to testing strategy is to convince

yourself and the staff that your code is functional, and then to argue why you think so in your report. If in order to convince yourself that your code is functional you realize further tests are needed (maybe just by copying and adjusting existing tests), then you should definitely write a few more.

Note that as you build new data structures (e.g., k-d trees), you *must* test your new code in isolation! This is good programming practice and prevents bugs from trickling into more complex code that uses your data structure in unexpected ways. Bugs become exponentially harder to debug as the complexity of the code grows, and you have no doubt experienced this in the previous assignments. Make sure to unit test your new code while the code complexity is still low!

In each of the basic tests for all handwriting recognition systems, we have left some commented `print` calls that print the image being classified or that print all images in the training dataset. In order to understand what each test is doing, feel free to temporarily uncomment these prints before commenting them again afterwards to avoid cluttering your test output.

Although many tests have been provided for you, you should still describe in your report what these tests do (they are well-commented) at a high level and then argue in your report that this convinces you that your code is fully functional.

4. Evaluation

Once you have verified the functionality of your handwriting recognition systems, you can evaluate their performance and accuracy against each other with breakdowns for both the training phase and the inference phase. You can build and run the evaluation program like this:

```
% cd ${HOME}/ece2400/<netid>
% mkdir -p pa5-sys/build-eval
% cd pa5-sys/build-eval
% ../configure --enable-eval
% make eval
```

We are working in a separate `build-eval` build directory, and we are using the `--enable-eval` command line option to the `configure` script. This tells the build system to create optimized executables without any extra debugging information.

The evaluation program runs all handwriting recognition systems on large subsets of the MNIST dataset. The overall procedure is to first declare an instance of each `IHandwritingRecSys` system. We train by calling the `train` member function and passing in the training dataset. We then iterate through each `LabeledImage` in the test dataset, we get the image from the `LabeledImage` using `LabeledImage::get_image()`, and then we call the `classify` member function and pass in the image. We compare the returned label to the actual label and count how many images were labeled correctly.

The evaluation program generates a table comparing the training time (i.e., the time inside the `train` member function call), the inference time (i.e., the time inside the `classify` member function calls), and the total accuracy across the test set (i.e., for how many images did the system predict the label correctly?). Multiple tables are printed out with different dataset sizes so you can see how the training and inference times scale with training dataset size and inference dataset size. You can also see the impact of the dataset sizes on accuracy.

When analyzing the performance results, think carefully about what work each algorithm does in each phase. Is training time increased in exchange for reduced inference time? What is the impact on accuracy by allowing approximate neighbors in systems using k-nearest neighbors algorithms?

More broadly, think about how the performance and accuracy apply to real-life use cases. If you hold up your phone and want to classify some image from your notebook, which system would be best to use and why?

Acknowledgments

This programming assignment was created by Christopher Batten and Christopher Torng as part of the ECE 2400 Computer Systems Programming course at Cornell University. We also thank the curators of the MNIST database of handwritten digits.