600.325/425 — Declarative Methods
Assignment 3: Machine Learning for Handwritten Digit Recognition[*]

Spring 2008
Prof. J. Eisner

Due date: Fri., April 11, 2 pm

In this open-ended assignment, you will build a series of classifiers to identify bitmap images of handwritten digits, and experiment with features to improve their quality.

Collaboration: *You may work in pairs on this open-ended assignment.* That is, if you choose, you may collaborate with one partner from the class, handing in a single homework with both your names on it. Of course, the two of you should observe academic integrity and not claim any work by third parties as your own.

How to hand in your work: Submission instructions will be announced before the due date (contact the TA with questions). Besides the comments you embed in your source code, include all other notes, as well as the answers to the questions in the assignment, into a **single** file named README. Include directions for building the executables, either in your README or in a Makefile.

Data: All the files you will need for this project are available on the ugrad machines in /usr/local/data/cs325/hw3/. You will find two files in the data/ subdirectory, train.gz and dev.gz.

As the filenames suggest, these files are compressed with gzip. You can uncompress your copies with gunzip, or examine them with zless or zcat.

Each of these files consists of a sequence of handwritten digits, each of which is encoded as a 16×16 grayscale bitmap. Each line corresponds to a single digit, consisting of 256 grayscale values between -1.0 (white) and 1.0 (black), followed by an integer between 0 and 9 which indicates the digit depicted. The first line is special: it contains a pair such as 6000 257, meaning that it will be followed by 6000 lines with 257 numbers on each line.

---

[*]Thanks to John Blatz for co-authoring this assignment.

The images have already been preprocessed to make classification easier. They were deslanted (rotated so that they are roughly upright), and scaled to a standard size.[1] Thus, encoding each image into this length-256 vector was already not trivial, and already allows machine learners to do surprisingly well. You will be trying to improve the encoding further.

To graphically view these deslanted, scaled, handwritten digits, you can type `view_digits train.gz` (thanks to Jim Skrenty). For a quick ASCII art rendering of the first 3 digits, try `print_digits train.gz 0 3`.

These programs are in `/usr/local/data/cs325/hw3/bin`, so you'll need to put that directory on your `PATH` first, or else give a full pathname to the program (e.g., `../bin/view_digits train.gz`).

There are 6000 examples in your training set `train.gz`, which you will use to train your classifiers, and 1650 more in your development set `dev.gz`, which you will use to evaluate them. There is another similar set `test.gz` which you will *not* be given and which the TA will use to evaluate your classifier. It is standard practice not to see the test set until the final evaluation. Otherwise, you might get unrealistically good performance on the test set because you have been "improving" your classifier to do well on that particular test set, and you may have inadvertently overfit to it. Instead, you should tune your classifier to do well on the development set, and hope that you still do well in the "real-world" condition of a totally new test set.

The digits 0 through 9 do not appear equally often in the data (can you guess why?). The class distribution for these files is as follows:

Digit	0	1	2	3	4	5	6	7	8	9	Total
train	980	840	600	536	534	470	542	500	449	549	6000
dev	277	226	175	127	150	111	146	167	145	126	1650
test	?	?	?	?	?	?	?	?	?	?	?

This dataset was made available by Yann Le Cun of AT&T Research Labs, and passed on to us by Laurent Younes from JHU's AMS department.

Binaries: In the directory `/usr/local/data/cs325/hw3/bin/` you will find three programs: `knn`, `mlp`, and `svm`. These programs will build and train classifiers from a data set, using the *k*-nearest neighbors algorithm, a neural network ("multi-layer perceptron"), or a support vector machine. They were easy to build using the Torch machine learning library (http://www.torch.ch).[2] Basic instructions on how to use them will be given in

---

[1]LeCun et. al. , "Handwritten Zip code recognition with multilayer networks", Proc. ICPR '90, 35-40, 1990.

[2]R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical Report IDIAP-RR 02-46, IDIAP, 2002. http://www.idiap.ch/~bengio/publications/pdf/rr02-46.pdf.

the subsequent sections, but if you want full documentation of all the options, run each one with no arguments.

Using these tools and this data, your job is to construct the best classifier you can.

1. First, learn how to use the classifier training programs you have been given.

   (a) To build a k-nearest neighbors classifier, run the command

   ```
   knn --test -K <int> train.gz dev.gz 256 10
   ```

   where `<int>` is the value of k that you want to use, and 256 and 10 represent the number of features and the number of output classes, respectively. If you omit the `-K <int>` option, it will set $k = 1$. You can additionally also include the `-norm` option to scale each dimension of the input vectors so that the mean is 0 and the variance is 1.

   This will produce files called `training_error` and `test_error`, which contain the percentage of examples on the training and test sets, respectively, that were misclassified.

   Report the training and test errors for $k = 1, 3, 25$ in your `README`.

   (b) To build a neural net (also called a "multi-layer perceptron" or MLP) with an input layer of 256 nodes for the pixels, an intermediate ("hidden") layer consisting of 64 nodes, and an output layer of 10 nodes for the class, execute the command

   ```
   mlp train.gz model 256 10
   ```

   where `model` is a file that will be created which will store the parameters of the fully trained model. You can vary the number of nodes in the hidden layer by using the `-nhu <int>` option.

   This will produce the file `training_error` as before, except this time it will contain a list of training errors at each iteration of training.

   To test it, use

   ```
    mlp --test model dev.gz
   ```

   This produces a file `test_error` that reports the classification error on the test set.

   Report in your `README` the final training error and the test error for this method using the default options.

   (c) Build a support vector machine, execute the command

   ```
   svm train.gz model 10
   ```

and test it with

```
 svm --test model dev.gz
```

By default this uses a gaussian kernel; you can optionally specify a polynomial kernel instead. For example, a polynomial kernel of degree 3

Report in your README the training and test errors for this method using the default options.

2. Now, using these tools, construct the best classifier you can. Your classifier must be able to be invoked from a command line with the command

```
./classify <train_file> <test_file>
```

You can actually already run this command; the simple version that we provide just uses mlp with the 256 pixels as input features, as you did above. However, you should modify our version to get better accuracy.

To do this, you will probably (i.e. if you want a good grade) want to construct some of your own features. You are given grayscale values for each pixel in the image, and have so far been using them as the only features being input to the classifiers. Note that the classifier doesn't even know which of these pixels are physically next to each other in the image! You could replace or (probably better) augment these 256 features with other features, which might somehow quantify other properties of the image, such as some of these:

- the average darkness of the image,
- the average darkness in various large or small local regions of the image,
- the standard deviation of various quantities (e.g., is the width of the digit constant or does it vary across different heights?),
- the number or length of vertical lines, loops, or "T" or "X" crossings in the image,
- the direction the pen was apparently moving as it crossed each pixel,
- scaled versions of the pixels' grayscale values,
- products of the grayscale values of pixels that are near one another,
- some cleaned up or standardized version of the image (the images have already been deslanted and scaled, as discussed above; you might consider augmenting the feature vector with a version of the image that tries to straighten jagged lines, or sharpen thick gray lines into something more black and white, or perhaps blur the image so that two examples of "9" look more similar),

4

- the size, location, and shape of whitespace regions,

- measures of horizontal and vertical symmetry,

- the "similarity" of the image to "standard" drawings of "0," "1," "2," etc.

- how loudly your cat meows when you show it the bitmap,

- or anything else you can think of.

Intelligently chosen features will improve the discriminative capability of your classifier. It shouldn't be hard to play around with different feature computations. It might help (especially for k-nearest neighbors) to rescale your features or combine existing features into new ones.

You are also free to use whatever learning algorithm you want, using whatever options you find interesting, or even to write your own algorithm (though you shouldn't need to do this). Feel free to use or adapt open-source code from Torch[3] or Weka or other packages you find, as long as you acknowledge it.

Remember that your classifier will not be evaluated by its performance on dev.gz, but rather by its performance on another test set that you don't have access to. So although good results on dev.gz should be encouraging to you, be careful not to design a classifier so specialized that it does well on dev.gz but won't generalize well to other test sets.

As a practical matter, how do you build your classifier? It should be some modification of the shell script classify, which is provided. This script calls the program build_features to produce enriched versions of the training and development files. Then it trains a neural network on the improved training file, and tests it on the improved development file.

Obviously, you might want to change what build_features does. At present it runs BuildFeatures.java, which simply reads the 256 original features into a 16×16 grid and then writes them back out in a different order (useless, eh?). You can modify the provided BuildFeatures.java (in the src/ directory), using it as a starting point that already handles the annoying I/O. Or if you prefer, you could replace it entirely with something in a new language.

---

[3]As Billy Prin pointed out to the class list, the programs supplied with HW3 (knn, mlp, and svm) "are just examples of the types of classifiers that can be written with the Torch library. You can also write your own. To do so, download the Torch library source code from the website. You then must build the entire library before you build any individual program you write. The build instructions on the site are straightforward, just note that the names of the packages in the Makefile config are the names of their directories in Torch3/. To rebuild the example programs, you need nonparametrics and kernels, so make sure you put them in the config Makefile. To get started on writing your own classifier programs, you should read through the Torch tutorial. It provides an overview on how each aspect of the library is tied together and how you can use the library to fully customize your own machine learning program."

Obviously, you might also want to change `classify` so that it runs something other than a neural net, or trains with different options. Make sure that you invoke your modified version of `classify`, not the original version.

Hybrid classifiers are also an option. For example, we discussed two paradigms in class:

- **Local learning.** Given a test example x, get `knn` (or another implementation of k-NN) to actually identify the k nearest neighbors from training data. Train a little SVM or neural net or something on just those k training examples (perhaps weighted by their similarity to x). Use that "local classifier" only once, to classify x.

- **Neural nets for constructing new features.** A neural network's hidden units compute non-linear combinations of the input features (specifically, linear combinations that are then passed through a sigmoid function). These non-linear combinations were trained to be useful in whatever task the network was trained to carry out.

  So train a neural net on either the original task,

  - given a digit image x, identify which digit $y \in \{0, 1, \ldots 9\}$ it is (exactly as in question 1b)

  or else on some related task whose input is derived from x and for which the hidden units might construct new features that would *also* be useful in the original task:

  - given a blurry version of x, identify y
  - given only a portion of x, identify y
  - given half of x, identify whether it is the top or bottom half[4]
  - given a rotated version of x, identify the angle of rotation

  Now, use this trained neural net to get additional features of x that you can use when training and testing a new k-NN or SVM classifier. You will have to get `mlp` (or whatever other implementation of neural nets you're using) to tell you how strongly each of its hidden and output units is activated when it classifies a given input derived from x.

  You can obtain even more features by training additional neural nets, either on the same task each time (since different training runs may find different local minima) or on a different task each time.

---

[4]An advantage to this task, and the next, is that it does not require knowledge of y. Thus, they could be trained on *unlabeled* digit images, which might be available in large quantities.

Describe in your README (or a PDF file) what techniques you tried, *why you thought they would help the classifier*, and how well they performed (error rate on dev.gz). What conclusions do you draw from these experiments? Do they suggest anything about future directions for improvement?

Also submit your version of the classify script (or multiple versions), along with any other necessary code (e.g., build_features and BuildFeatures.java). Explain if necessary how to compile and run your code so that we can evaluate its performance on test.gz.

Grading will be based less on performance than on effort, thoughtfulness, and writeup. Good ideas for features will get a better grade than random tinkering with program options, even if the cool ideas turn out to help the error rate less. So implement something that you find interesting, and don't stress out too much over trying to improve the bottom line.

That said, there will be **special prizes** (probably made of chocolate) given to the couple of teams that design the best classifiers.