

Project 1: Decision Trees and Nearest Neighbors

Introduction

In this project, we will explore two different learning algorithms (decision trees and k-nearest-neighbor) on two different data sets (hand-drawn digits and newsgroup postings).

As will be the case with almost all of the projects, there is both an *implementation* aspect and an *analysis* aspect to this project. You will be implementing a decision tree classifier and a k-nearest-neighbor classifier. You will then run both of these classifiers on two data sets: one of hand-drawn digits (“mnist”) and one for text categorization (“20ng”). These runs will produce a variety of results which you will need to interpret.

Grading: grading is done separately for the code and the analysis. Errors in code will result in a reduction of points. The ideal solution will be one with correct code and a correct analysis of the results. However, analysis can be tricky if your code is buggy. Therefore, if your code has bugs, but you *acknowledge* this in your analysis, you will recover some of the analysis points. That is, if your code is clearly buggy (i.e., your classifier gets < 50% accuracy on a binary classification problem), you should point this out in your analysis. In order to recover points, you will need to say specifically *what* you think the bug is and *why* you think it’s a bug. (Of course, it would be ideal if you could just fix it!)

You will need to hand in: (a) your analysis in a file called `writeup.pdf`, (b) your code for `DT.m` and `KNN.m`, (c) a file called `results.mat` that will be created for you.

I have provided quite a bit of skeleton code. More importantly, there is a `run.m` script. This is the central part of the project. Once you’re confident that your DT and KNN code is doing (more or less) the right thing, you’ll run `run.m`. This will run a battery of experiments and produce a fair amount of output. Roughly, it will load some data, run your implementations on the data with different parameters, and check (on development data) that the accuracies you get are close to what you should be getting. For every test like this that it runs, it will tell you if you passed or did not pass the test. Not passing these tests is bad: not only does it signal a bug, but it also will cost points! The results of all these tests are recorded in the `results.mat` file.

After running the tests, it will use your code to make predictions on some held-out test data. You don’t have access to the true labels for the test data, so you’ll just have to cross your fingers (you really need to *trust* the learning algorithm). The predictions of your model on the test data are stored in the `results.mat` file, which we will compare to the truth.

Important note: Every time you run `run.m` it will first load in `results.mat` so that it can resume where it left off (this way you don’t have to keep rerunning the same tests over and over). However, this also means that if you need to “back up” and run some old experiments, you will need to delete the `results.mat` file so that it can’t load it.

In terms of *implementation*, you will need to (1) implement an algorithm for decision tree construction based on information gain; and (2) implement a prediction algorithm for k-nearest-neighbor (training is really non-existent here).

Early warning: getting through each of the full experiment cycles takes about 16 minutes with my implementation (most of the time with the machine thinking). Yours will likely take longer, especially if you are new to Matlab. (And my same code running in Octave takes over an hour!) So, start early and don’t put off the “analysis” section until the end because your write-up will depend on the output that the `run` produces.

Remember, handins are at: <http://www.cs.utah.edu/~hal/handin.pl?course=cs5350>.

Finally, one of the datasets we’re using has 10 classes (labeled zero through nine), the other has two (labeled 0 and 1)... your implementations should be robust to this.

Task 1: Decision Trees

(35%)

Your first task is to implement a decision tree based on information gain. There is shell code for this in `DT.m`. Your task is to implement two subroutines in this file, called `DTconstruct` and `DTpredict`, whose names hopefully tell you exactly what they do. There are comments in both of these functions that explain the data structures you should use and what the different parameters are. I found it most convenient to implement these functions recursively, but if you prefer to do it another way, that is fine too.

To see how the DT code works, we can try building a model (the last parameter, “0” is the cutoff value: at how many examples should we automatically consider a base-case):

```
>> model = DT('train', [1 0 ; 1 1 ; 0 1], [1 1 2]', 0);
>> model.tree
ans =
    isLeaf: 0
    split: 1
    left: [1x1 struct]
    right: [1x1 struct]
```

Here, we can see that it built a tree. This tree is *not* a leaf, so it has to have a `split` node. In this case, the split is on feature 1. (If you look at the data, you see that this makes sense.) It then has left and right subtrees. We can look at the left one:

```
>> model.tree.left
ans =
    isLeaf: 1
    label: 2
```

This is a leaf and has label 2. This means that if in the parent, we had encountered an example with split feature 1 equal to zero, we would branch down the left and then produce a label 2.

There is provided code for printing a full tree recursively:

```
>> DTdraw(model)
split 1
  Y=2
  Y=1
```

This prints the first split, then recursively prints left- and right-subtrees.

We can try predicting with this model:

```
>> DT('predict', model, [0 0])
ans =
    2
```

Does this seem like a sensible prediction?

We can run this on some randomly-created data just to make sure that nothing wacky happens (hint: if you run the following a few times and you get recursion errors, you probably need to make sure you’re not trying to split on features you’ve already split on!). This uses random data with 100 examples and 4 features.

```
>> DTdraw(DT('train', rand(100,4)>0.5, rand(100,1)>0.5,0))
```

```

split 1
  split 3
    split 4
      Y=1
        split 2
          Y=0
          Y=0
        split 4
          split 2
            Y=1
            Y=0
          split 2
            Y=1
            Y=1
    split 4
      split 3
        Y=1
          split 2
            Y=0
            Y=0
        split 3
          split 2
            Y=0
            Y=0
          Y=1

```

We can also test cutoffs. With a higher cutoff, we should get a shorter tree:

```

>> DTdraw(DT('train', rand(100,4)>0.5, rand(100,1)>0.5,40))
split 3
  split 4
    Y=0
    Y=0
  split 1
    Y=1
    Y=0

```

Yay! My implementation seems to work!

Task 2: K-Nearest-Neighbors

(25%)

Your second task is to implement the prediction phase of KNN. There is shell code for this in `KNN.m`. Your job is to implement the subroutine called `KNNpredict`. This takes as input the training data points (with labels), a value for K and a single test point. Its output should be the most frequent label among the K closest points in the training data to the test point. If there's a tie among the K , you can break it however you want.

We can now run a few tests on the KNN model. Let's first create some simple data that lies on a unit square, with $(0,0)$ having class 0 and all other points having class 1. We'll just do one nearest neighbor. (The last argument is the number k of nearest neighbors.)

```

>> model = KNN('train', [0 0; 0 1 ; 1 0 ; 1 1], [0 1 1 1]', 1)
model =

```

```

trainX: [4x2 double]
trainY: [4x1 double]
      K: 1

>> KNN('predict', model, [0 0])
ans =
     0
>> KNN('predict', model, [0 1])
ans =
     1
>> KNN('predict', model, [0 0.75])
ans =
     1
>> KNN('predict', model, [0 0.25])
ans =
     0

```

These predictions seem fairly reasonable.

Let's see what happens if we use three nearest neighbors:

```

>> model = KNN('train', [0 0; 0 1 ; 1 0 ; 1 1], [0 1 1 1]', 3)
model =
      trainX: [4x2 double]
      trainY: [4x1 double]
           K: 3

>> KNN('predict', model, [0 0])
ans =
     1
>> KNN('predict', model, [0 1])
ans =
     1
>> KNN('predict', model, [0 0.75])
ans =
     1
>> KNN('predict', model, [0 0.25])
ans =
     1

```

Since no matter how it chooses three points, at least two of them are going to be class one, it will always predict class one on this data.

Task 3: Analysis

(40%)

If all the above tests passed okay, it's probably not time to try the `run` script. This will produce a *large* amount of output.

When you execute `run`, it will begin with some tests on the digits data and then move on to the 20-newsgroups data. It runs roughly the same experiments on each data set. It also tells you for the initial baseline experiments whether your implementation seems to be getting roughly the right levels of performance.

Whenever the script produces a figure, it also *saves* the figure according to figure number (eg., `fig1.png`). If you prefer a format other than `.png`, edit the first line of the `run` script. (*Octave note:*) *I can't figure out how to get Octave to do this—so it won't. Bonus points if you can figure it out!*

The script runs through the following experiments:

- MNIST data [about 2 minutes]:
 - Test of performance of DT with cutoffs of 20 and 50, and test of KNN with $k = 1$ and $k = 3$. [10 seconds]
 - DT performance as a function of cutoff. [30 seconds].
 - KNN performance as a function of k . [30 seconds].
 - Predictions of dev and test data using DT. [9 seconds].
 - Predictions of dev and test data using KNN. [4 seconds].
 - Plots of digits erroneously classified. [17 seconds].
- 20 newsgroups data [about 16 minutes]:
 - Test of performance of DT with cutoffs of 20 and 50, and test of KNN with $k = 1$ and $k = 3$. [50 seconds]
 - DT performance as a function of cutoff. [2 minutes].
 - KNN performance as a function of k . [8 minutes].
 - Predictions of dev and test data using DT. [30 seconds].
 - Predictions of dev and test data using KNN. [4 minutes].
 - Plots of digits erroneously classified. [0 seconds].

When you provide your analysis, be sure to embed the figures so that we’re looking at what you’re looking at!

The questions you should answer in your analysis are:

1. In Figure 1, we see a handful of examples of different digits with their corresponding class labels. Do you see any that would be difficult for you to classify manually?
2. In Figure 2, we see how the DT cutoff affects performance. The x-axis is the cutoff, the y-axis is the accuracy. The blue x-ed line is accuracy on the training data; the black o-ed line is accuracy on test (or, more specifically, development) data. Do these curves appear to have the right form? Based on these results, what cutoff value would you choose?
3. In Figure 3, we compare training/test accuracy for KNN with different values of k . Do these curves have a reasonable form? Is the blue curve close to (1,1)? Should it be? What value of K would you choose?
4. Comparing Figures 2 and 3, which, of DT and KNN, did better? Why do you think this is?
5. In Figure 4, we see example digits (from the development data) that *both* the KNN *and* the DT misclassified. They are shown with their true labels. Would you, as a human, have had trouble classifying these? Which ones?
6. In Figure 5, we see example digits (development) that the DT misclassified but the KNN got right. They are shown with the *predicted* label. How many of these do you think you would misclassify? Is there any pattern to the errors, or do they seem “random”?
7. In Figure 6, we see the same thing, but for examples that KNN got wrong (and DT got right). Answer the same questions.
8. Once the 20 newsgroups experiments start, it will print six “documents”, three from each of the two classes. These documents are just “bags of words” and some of the words have been stripped of morphology. Can you guess what classes 0 and 1 are based on these examples?

9. Figure 7 shows DT performance as a function of cutoff for the 20 newsgroups data. What cutoff would you choose? How does this compare to your answer for question (2)? How does the optimal *training error* compare to that of MNIST? Why do you think this is the case?
10. Figure 8 shows KNN performance as a function of K . Do these curves seem reasonable? How do they compare to the KNN curves for the MNIST data? If they are different (they should be!), why do you suspect that they are?
11. We then see some documents that were misclassified by both systems. Are there any words in these documents that (based on the six training examples you saw and whatever you believe about what the underlying classes are) you think should have pointed the models in the right way?
12. Next, we see documents erroneously classified only by the DT. Are these errors explainable, or can you not tell the difference either? What features seem most indicative?
13. Finally, we see documents erroneously classified only by KNN. Answer the same question.
14. Based on all your responses, which algorithm seems to work better for digits? Which seems to work better for document classification? Based on what you know about these algorithms, why do you expect this to be the case?

Bonus Task: Improving the Results

(up to 15%)

If you are up to it at this point, I welcome you to try to improve on one of the models in some way. You may work with either data set and with any of the two learning algorithms. You may do *whatever* you want. If you accept this challenge, then you need to include the following in your handin: a description of what you've done (in the writeup), the code implementing what you've done (as a .m file with a note on how to run it) and the predictions your method makes on the `testX` data in either `mnist` or `20ng`, saved as a .mat file. You will get bonus points for: significantly improving (a few percentage points) the quality of the predictions on the test data; a particularly clever idea that may not have been as successful but at least didn't hurt; both.

If you choose to do the bonus, you should package all the extra material into a `tgz` file and include it as "supplemental" material in the handin.

Appendix: Helper Files

There are a few matlab commands that do not exist in Octave that you may find useful for this project. They are `mode` and `unique`, provided to you as `mymode` and `myunique`. They work as follows. `mymode(A)` returns the *most common element* in the list `A`. `myunique(A)` produces a list of all the *unique* values in `A`. For example:

```
>> mymode([0 1 1 1 2 0 1 2 3 3 3])
ans = 1

>> mymode([0 1 1 1 2 0 1 2 3 3 3 3 3])
ans = 3

>> myunique([0 1 1 1 2 0 1 2 3 3 3])
ans = [0 1 2 3]

>> myunique([0 1 1 1 2 0 1 2 3 3 3 3 3])
ans = [0 1 2 3]
```