**More Program Development**

**What is To-down Refinement?**

Approaching design from the top down begins with a description of a solution at a very high level – this description is not written in any particular programming language. Once we have that high level description, we then refine that description into more detail, where we identify an appropriate piece of the solution description, and create functions to fulfill the role identified.

In the initial (top level) design, we worry about the “big picture” approach to solving the problem. We describe what data structures we need and what “big” processing steps we might take along the way.

To make this process concrete, we will use a complicated example that will require more development on your part.

**Example: The Breast cancer classifier**

Scientists make data sets available for use by other researchers in the hopes that the data will be helpful in solving important problems. Various repositories exist around the world that distribute such data sets, made freely available to everyone. One such repository is the UC-Irvine Machine Learning Repository (<http://archive.ics.uci.edu/ml/>)

One of the data sets describes tumors removed from the breast cancer patients. The data were provided by Dr. William H. Wolberg at the University of Wisconsin Hospitals in Madison. Each of the patients had a tumor examined by oncologists to describe various features of the tissue. Subsequent to that examination, they determined if the tumor was *benign* or *malignant*.

**The Problem**

The problem is to determine, based on the tumor attributes, whether the tumor is malignant or benign. We do so by examining the data provided by Dr. Wolberg. The data list has 699 patients, each with nine tumor attributes provided by the oncologist who examined the biopsy, as well as whether the patient was ultimately diagnosed as having a benign or malignant tumor. That is the “solution” (malignant or benign) is included in the data set. Therefore, every patient has a set of 11 values: a patient ID, the nine tumor attribute values, and the ultimate diagnosis. By examining these data, we hope to discover patterns that *predict*, based on the tumor attributes alone, whether the tumor is malignant or benign. That is, for a patient we have not yet seen (and do not yet know the diagnosis), we wish to predict whether the tumor is malignant or benign based on the tumor attributes.

**The Approach – Classification**

It turns out that there are a number of approaches we could take to solve this problem. In fact, there is an entire research area, “data mining” that works on ways to solve such problems. The approach we will use is to create a *classifier* – a program that takes in a new example (a patient, in our case) – and determines, based on previous examples it has observed, what “class” the new example belongs to.

For this problem, we consider patients, along with their associated tumor attributes, as our examples and separate each into one of two classes: benign or malignant

**Training and Testing the Classifier**

A classifier begins by *training* on examples with known solutions. In training, the classifier looks for patterns that indicate classifications (e.g., benign or malignant). After patterns have been identified, they are *tested* against “new” examples with known solutions. By testing with known solutions, we can determine the accuracy of the classifier.

In our case, we provide to the classifier patients’ tumor attributes that have a known result (benign or malignant). Each patient contributes toward building an *internal model* of what patterns are used to distinguish between our two classes. Once we have trained the classifier, we must test the classifier’s effectiveness. We do this by providing “new” patients, or patients who were not used as data for the training process, to see what class the classifier predicts each new patient belongs to.

We have to take our data and split them into two parts: data we will use to “train” the classifier and data we will use to “test” the classifier. In practice, we will create two separate files, with most of the data in the training file and the remainder in the testing file.

Now the issue is this: how do we write a program that can find patterns in the training data?

**Building the Classifier**

There are many interesting internal models that a classifier could use to predict the class label of a new example. We will use a simple one, but you will see that it can be quite effective.

Here is the approach. Look at a tumor attribute for each individual, and then combine the observations on that attribute into a decision value used to classify an individual for that particular attribute. For example, one attribute is tumor thickness, measured on a scale of 1 to 10. A good decision value for this attribute might be 7. For a value of 7 or greater (i.e., a thick tumor), our classifier will predict malignant. For a value of less than 7, our classifier will predict benign. We can use these values to predict a patient’s class.

How do we find these decision values? For each of the nine tumor attributes, let us develop two averages. The first average for each attribute will represent the average value over all the training data for women with *benign* tumors. The second average for that attribute will represent that average value over all the training data for women with *malignant* tumors. After training on the nine attributes, we should end up with 18 averages: 9 averages for benign tumors and 9 averages for malignant tumors.

Our classifier will be constructed as follows: For each attribute we will find the midpoint between benign average and the malignant average. This midpoint of averages will be our decision value, which is better termed the *class separation value*. Our classifier will consist of nine separation values, one for each attribute. If a new sample comes along with a value less than the separation value of an attribute, we will predict that this patient is benign, at least on that attribute. If the sample is greater than the separation value, we predict that it is malignant.

To select which overall class we predict the patient belongs to, we compare each of the nine tumor attributes for that patient to attribute’s classifier separation value. We label that attribute based on whether it is larger or smaller than the separation value. Remember, in this case, smaller indicates benign and greater indicates malignant. For the overall patient, we let the majority rule. Whichever class label predominates over the nine attributes, we use that label for that patient overall. The following figure shows the overview of the process.



Here is our first cut of an algorithm in English:

1. Create a training set from a training file
2. Create a classifier by using the training set to determine separator values for each attribute
3. Create a test set from the test file
4. Use the classifier (separator values) to classify data in the test set while keeping score of the accuracy of those decisions.

Listing1.py is the Python version of the above algorithm. It shows the overall structure of the program, including both the names of the function and the arguments of those functions, even though it lacks most of the details. **This code will not run** as there are five undefined functions. However, it does show a couple of things we need to do:

* Define the four data structures: test\_set\_list, training\_set\_list, classifier\_list, result\_list
* Define five functions: make\_training\_set, train\_classifier, make\_test\_set, classify\_test\_set, report\_results

We can fix up our main program to run by defining skeleton versions of all the functions. Again, by skeleton, we mean that the function is defined, the argument numbers are correct, and the function returns a value. Otherwise, the function definitions are empty. By creating skeleton functions, we create a running program with all the requisite parts. It doesn’t do anything, but that is something we can work on. Listing2.py code listing has the skeleton functions. Remember that the functions *must* be defined before they are called. Most important though, it gives us an outline for the entire program.

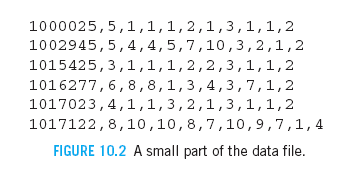
**Data Structures**

* training\_set\_list, test\_set\_list : The data structures training\_set\_list and test\_set\_list contain information about each individual patient. Because the data on each patient are never modified, each patient’s data could be stored as a tuple of values. A list of tuples would then hold all patient data. Therefore, a list of tuples will serve for the training and test data structures. The tuple format will be:
  + First position: Patient ID, string
  + Second position: patient diagnosis – String (single letter , ‘*m*’ or ‘*b*’)
  + Positions 3-12: tumor attributes 1-9 in order. Integer
* classifier\_list: The classifier is simply a sequence of nine values, the separation values. A tuple will suffice, as these values also do not change. It will consist of nine floating-point values: the average of each benign and malignant attribute average (the midpoint between each of the two averages)
* results\_list: For the results, we need a list of tuples again, but what to put in each tuple? The classifier provides the number of attributes for each patient that indicate malignant and benign. We also need the actual diagnosis for the patient. Therefore, we need to store:
  + The patient ID: String
  + The number of attributes that indicate malignant: integer
  + The number of attributes that indicate benign : integer
  + The actual diagnosis: string (either ‘*m*’ or ‘*b*’)

**File format**:

Here is the file format:

|  |  |  |
| --- | --- | --- |
| # | Attribute | Domain |
| 1 | Sample code number | Id number |
| 2 | Clump thickness | 1-10 |
| 3 | Uniformity of cell size | 1-10 |
| 4 | Uniformity of cell shape | 1-10 |
| 5 | Marginal adhesion | 1-10 |
| 6 | Single epithelial cell size | 1-10 |
| 7 | Bare nuclei | 1-10 |
| 8 | Bland chromatin | 1-10 |
| 9 | Normal nucleoli | 1-10 |
| 10 | Mitoses | 1-10 |
| 11 | Class | (2 for benign, 4 for malignant) |



**The make\_training\_set function**

The first function we are going to write is make\_training\_set function. This function takes a file name, opens the file, and reads each line. For each line it creates a tuple and returns a list of those tuples. Here is the list returned by this function:

def make\_training\_set(file\_name):

('1000025', 'b', 5, 1, 1, 1, 2, 1, 3, 1, 1)

('1002945', 'b', 5, 4, 4, 5, 7, 10, 3, 2, 1)

('1015425', 'b', 3, 1, 1, 1, 2, 2, 3, 1, 1)

('1016277', 'b', 6, 8, 8, 1, 3, 4, 3, 7, 1)

('1017023', 'b', 4, 1, 1, 3, 2, 1, 3, 1, 1)

('1017122', 'm', 8, 10, 10, 8, 7, 10, 9, 7, 1)

**The make\_test\_set function**

If you look closely, the make\_training\_set and make\_test\_set functions perform similar tasks. They take in a file name and return the same data structure format. They only real difference is that they read from two different files: the test data and the training data, both of which have the same file format.

Thus, we can reuse our code from make\_training\_set to accomplish the same tasks in the make\_test\_set function. To reflect this, we will rename the function to make\_data\_set and use it in both operations in the main function. **Make this modification to your code.**

**The train\_classifier function**

The train-classifier function is the heart of the program. Here is a skeleton

def train\_classifier(training\_set\_list):

for patient\_tuple in training\_set\_list:

# if patient\_tuple is benign:

# add attributes from patient\_tuple to corresponding benign\_sums\_list

# attribute

# increase the benign\_count by 1

# else:

# add attributes from patient\_tuple to corresponding malignant\_sums\_list

# attribute

# increase the malignant\_count by 1

# create benign\_averages\_list by dividing each benign\_sums\_list by the

# benign\_count

# create malignant\_averages\_list by dividing each malignant\_sums\_list by the

# malignant\_count

# create classifier\_list by dividing the sum of each attribute from

# benign\_averages\_list and malignant\_averages\_list by 2

#

# return classifier\_list

Let us do little testing to make sure that our function is working. Write code for train\_classifier function. If your function works correctly, it should produce the following output.

Reading in training data...

('1000025', 'b', 5, 1, 1, 1, 2, 1, 3, 1, 1)

('1002945', 'b', 5, 4, 4, 5, 7, 10, 3, 2, 1)

('1015425', 'b', 3, 1, 1, 1, 2, 2, 3, 1, 1)

('1016277', 'b', 6, 8, 8, 1, 3, 4, 3, 7, 1)

('1017023', 'b', 4, 1, 1, 3, 2, 1, 3, 1, 1)

('1017122', 'm', 8, 10, 10, 8, 7, 10, 9, 7, 1)

Done reading training data.

Training classifier...

6.300

6.500

6.500

5.100

5.100

6.800

6.000

4.700

1.000

Done training classifier.

Reading in test data...

Done reading test data.

Classifying records...

Done classifying.

Reported the results

Program finished.

**Testing the classifier on new data**

Now we need to see whether the classifier we have created will properly predict the patient diagnosis based only on the patient’s tumor attributes. This testing is done with the classify\_test\_set function.

Let us remember what this function has to do. It takes a set of test data, consisting of patient tumor data along with the determined diagnosis. We compare each attribute in the patient with the corresponding classifier average. If the attribute is larger than the classifier average, that attribute is considered to be evidence of malignancy. If it is smaller, then that attribute indicates benignity. For each of those patients, we count the number of benign and malignant attributes, and the majority rules. That is, whichever type of attributes is in the majority, that is the classifier’s predicted diagnosis.

def classify\_test\_set\_list(test\_set\_list, classifier\_list):

'''Given test set and classifier, classisfy each patient in test set;

return list of result tuples: (id,benign\_count,malignant\_count,diagnosis)'''

result\_list = []

…

# return the list of result tuples

Complete the code for the above function and run your program. When you add the print statements in main to print each tuple in the result\_list, you should get the following output:

Reading in training data...

('1000025', 'b', 5, 1, 1, 1, 2, 1, 3, 1, 1)

('1002945', 'b', 5, 4, 4, 5, 7, 10, 3, 2, 1)

('1015425', 'b', 3, 1, 1, 1, 2, 2, 3, 1, 1)

('1016277', 'b', 6, 8, 8, 1, 3, 4, 3, 7, 1)

('1017023', 'b', 4, 1, 1, 3, 2, 1, 3, 1, 1)

('1017122', 'm', 8, 10, 10, 8, 7, 10, 9, 7, 1)

Done reading training data.

Training classifier...

6.300

6.500

6.500

5.100

5.100

6.800

6.000

4.700

1.000

Done training classifier.

Reading in test data...

Done reading test data.

Classifying records...

('1000025', 9, 0, 'b')

('1002945', 7, 2, 'b')

('1015425', 9, 0, 'b')

('1016277', 6, 3, 'b')

('1017023', 9, 0, 'b')

('1017122', 1, 8, 'm')

Done classifying.

Reported the results

Program finished.

**The report\_results function**

Finally, we need to report the results. You might think that this would be pretty straightforward and not really require a function, but what should be reported? The accuracy would be nice, but one could imagine reporting how many 9/0 votes, how many close votes, and so on. Therefore, we write a function because we want to isolate potential future changes.

For now, we will just report the accuracy. Remember that the results data structure is a list with four elements, in order: id, benign\_count, malignant\_count, diagnosis. Also, remember that the diagnosis in the results is the *actual* patient’s diagnosis, not our classifier prediction.

What do we mean by accuracy? “Majority rules” means that if the benign\_count > malignant\_count, our prediction is benign (‘b’), and similarly for malignant. Does our predictor correctly predict the actual diagnosis?

Here is the pseudo code:

Initialize the total and inaccurate counts

if the benign\_count > malignant\_count and actual diagnosis is ‘m’ for malignant, there is an inaccuracy. Use a similar logic for malignant count. (Note that since the number of attributes is odd (nine), it is not possible for the counts to be equal)

Print the results.

After you write the code for report\_results function, and run your program, this is the output you should get:

Reading in training data...

('1000025', 'b', 5, 1, 1, 1, 2, 1, 3, 1, 1)

('1002945', 'b', 5, 4, 4, 5, 7, 10, 3, 2, 1)

('1015425', 'b', 3, 1, 1, 1, 2, 2, 3, 1, 1)

('1016277', 'b', 6, 8, 8, 1, 3, 4, 3, 7, 1)

('1017023', 'b', 4, 1, 1, 3, 2, 1, 3, 1, 1)

('1017122', 'm', 8, 10, 10, 8, 7, 10, 9, 7, 1)

Done reading training data.

Training classifier...

6.300

6.500

6.500

5.100

5.100

6.800

6.000

4.700

1.000

Done training classifier.

Reading in test data...

Done reading test data.

Classifying records...

('1000025', 9, 0, 'b')

('1002945', 7, 2, 'b')

('1015425', 9, 0, 'b')

('1016277', 6, 3, 'b')

('1017023', 9, 0, 'b')

('1017122', 1, 8, 'm')

Done classifying.

Of 6 patients, there were 0 inaccuracies

Program finished.

**Running the Classifier on full data**

Now comes the real test. We have assembled all the functions and have tested them in parts. Now we need to put the full code together and test it on the entire breast cancer data set.

We need to take the full 699 patients and divide them into two files: one file to train the classifier and one file for testing. There are many strategies for testing the accuracy of a classifier. The goal is to train the classifier on as many examples as possible while still testing many patient values. We will take the simple approach and just divide the original file in half: 349 for training, and 350 for testing.

Run your program by replacing test\_data.txt with training\_data.txt.

Did you get an error? Does the error make sense to you?

A look at the data file shows the problem. There are a number of patients out of the 699 that have an unknown value as part of their data, indicated by a ‘?’ instead of an integer. That is the source of your problem. We missed that in testing because we looked at only a small portion of the data.

We have a few choices. We can throw out the “bad” data (there are 16 patients that have a ‘?’ instead of an integer), or we can fix the problem in the make\_data\_set function. It may be a bad idea to throw away the 16 patients with a ‘?’ for an attribute. However, it may also be a bad idea to “make up” a substitute value for the missing data.

We choose to modify the make\_data-set function to ignore any patient with a ‘?’. We do that by modifying our code in make\_data\_set function by skipping the patient if there a ? in the patient data.

Now run the program – use the file training\_data.txt for making training\_set\_list and use the file fulltest\_data.txt for making test\_set\_list.

You should get the following output:

>>> main()

Reading in training data...

Done reading training data.

Training classifier...

Done training classifier.

Reading in test data...

Done reading test data.

Classifying records...

Done classifying.

Of 348 patients, there were 7 inaccuracies

Program finished.

>>>

**Deliverable:**

Submit the final working program – no need to submit any of the data files.

Program should be well documented.