Part 1

In this part of the assignment, you will use and slightly modify Weka to explore the use of priors in maximum
a posteriori (MAP) hypotheses.

Recall that in the Naive Bayes classifier each hypothesis $h$ consists of a set of probability estimates:

- For each possible value $y_i$ of the target attribute $y$, we need to estimate $P(y = y_i)$.
- For each possible value $y_i$ of $y$ and each possible value $k$ of each attribute $x_j$, we need to estimate
  $P(x_j = k|y = y_i)$.

The approach to estimating these probabilities that we introduced in class results in learning a maximum
likelihood (ML) hypothesis. To turn that ML hypothesis to a MAP hypothesis, we need to introduce prior
probabilities over hypotheses. An easy (but indirect) way of doing this is by changing the way in which
we estimate $P(y = y_i)$. Recall that in the ML hypothesis, $P(y = y_i)$ is estimated as (using the notation
introduced in class):

$$P(y = y_i) = \frac{|D[y = y_i]|}{|D|}$$

A simple way of introducing priors over hypotheses is by pretending that, for each possible value $y_i$ of
$y$, $D$ contains $m$ additional “phantom” examples in which $y = y_i$. This essentially amounts to changing the
estimate for $P(y = y_i)$ to

$$P(y = y_i) = \frac{|D[y = y_i]| + m}{|D| + m * n}$$

Above, $n$ is the number of possible values for $y$, i.e. if $y$ is Boolean, $n = 2$. This essentially results in
estimates for $P(y = y_i)$ that are closer to uniform, i.e. $P(y = y_1), P(y = y_2), ..., P(y = y_n)$ are closer to
being equal.

1. (5 points) Explain why the previous sentence is true.

Next, we will explore how estimating $P(y = y_i)$ in this new way has a smoothing effect. For this pur-
pose, you will need to use your Naive Bayes data generator from SA2 to generate the following three sets of
data.1 We will use artificially generated data rather than an existing data set so that we have perfect control

---

1 A python implementation of this generator is provided on the “Assignments” webpage.
over what is in the data.

2. Generate the data:
a) A test set (test.arff) using the command:

Generator.py 500 0.5 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3

This data is balanced in the sense that roughly half of the instances will have $y = 0$ and half will have $y = 1$.
b) A small balanced training set (balanced-train.arff) using the command:

Generator.py 100 0.5 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3

This data has exactly the same characteristics as the test set.
c) A small skewed training set (skewed-train.arff) using the command:

Generator.py 100 0.2 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3 0.7 0.3

Notice that in the skewed set, $y = 1$ has a much lower probability of occurring, but that all other parameters are the same as in the balanced data.

3. Weka implements Naive Bayes in the weka.classifiers.Bayes.NaiveBayesSimple class. Train and test two Naive Bayes classifiers on this data as follows, recording the accuracy on the test data in each case:
a) [Balanced classifier] Train on balanced-train.arff and test on test.arff
b) [Skewed classifier] Train on skewed-train.arff and test on test.arff

Hint: Read the instructions on p. 19 of the Weka manual on how to specify separate train and test files.

(10 points) Perform the experiment outlined in steps 2 and 3. Report the accuracy you obtained in each case. Which of the two classifiers (i.e. the balance or the skewed) was more accurate and what is this difference in accuracy due to? In addition to your answers, submit your generated data sets to the submit server.

4. Open the weka/classifiers/Bayes/NaiveBayesSimple.java class and inspect the buildClassifier method. This is the method that implements learning. This implementation can deal with both discrete and continuous attributes, but here we will focus on discrete ones, which in Weka’s terminology are called “Nominal,” so ignore any parts of the code that do not concern Nominal attributes.

The buildClassifier method implements Equation (2) from above using a small, hardcoded value for $m$ . The task here is to experiment with larger values for $m$.
a) Modify the code such that it uses a larger $m$, e.g. $m = 300$.

(25 points) Submit your modified NaiveBayesSimple.java on the submit server.

Hint 1: The code calls the $P(y = y_i)$ “priors” because they are class priors, not to be confused with hypothesis priors. Hint 2: You can accomplish this by changing no more than 2-3 lines of code.
b) Perform steps 2 and 3 above with this newly modified code.

(10 points) Answer the following questions: How did the new accuracies of the balanced and skewed classifiers compare to their previous accuracies? Explain why this is observed.
Part 2

(50 points + 10 extra credit) The task in this part is to implement the ID3 decision tree learning algorithm. Your method should take two file names as command line arguments. The first one is the name of a data file in .arff format; the second one is a file to which the output of learning will be written, as described later. You can assume that the target attribute is Boolean and that the other attributes are discrete.

Your method should work as follows:
1. Read in the data and randomly re-order the instances.
2. Split the instances into a train and test sets of equal size.
3. Learn a decision tree on the train set using ID3 with the following modifications:
   a) Before learning starts, 1/3 of the training set is set aside to be used as a Validation set; the remaining 2/3 is the Actual Training Set used for training of ID3.
   b) Your implementation should find the best attribute \( A^* \) to test for next as in the original ID3 algorithm (i.e. by computing gain on the Actual Training Set). However, it should then add this attribute only if adding it leads to an improvement in accuracy on the Validation set. If accuracy on the validation set increases, learning proceeds as in ID3; otherwise, learning stops and the currently learned decision tree becomes the final hypothesis.
4. (Extra credit 10 points) Output the decision tree to the output file (that was provided as a second argument) as a set of rules in the format described below.
5. Test the final hypothesis on the test set saved at step 2 and write the accuracy on the test set to the standard output using the format shown in the example below

Your implementation should be written in Java and well-documented, and should compile on the submit server.

Here is a sample trace of how your method should operate:

> java ID3 weather.arff output.txt
> Accuracy on test set: 40.3%

To get the extra credit, also as a result, the output.txt should contain the learned tree as a set of rules. The format of output.txt should be as follows. Suppose that the tree your algorithm learned was the one shown on Figure 3.1 (p. 53 of the textbook). Then, the output.txt file will be:

Outlook = sunny ˆ Humidity = high => No
Outlook = sunny ˆ Humidity = Normal => Yes
Outlook = Overcast => Yes
Outlook = Rain ˆ Wind = strong => No
Outlook = Rain ˆ Wind = weak => Yes

If you do not want the extra credit, just leave output.txt empty/nonexistent.

Submission Instructions

Submit written answers to the questions in Part 1 at the beginning of class on Oct. 7. Submit a) Your generated data sets b) Your modified NaiveBayesSimple.java c) Your implementation of ID3 to the submit server by 11:59pm on Oct. 7