Naïve Bayes Classifiers

Introduction

In today’s fun-filled lab we’ll be using Bayes rule in the supervised learning setting to implement a naïve bayes classifier for nominal feature-based data. The goals of the lab include:

1. Experience writing “elegant” (haha, lol) code in Matlab. This will be achieved by using Matlab’s non-homogeneous data structures.

2. Experience building a probabilistic, supervised classifier.

Regarding non-homogeneous data usage, Belinda will give you some existing code to work with, in conjunction with a brief discussion of Matlab’s struct and cell constructs. One of your tasks is to figure out how to use these constructs when extending the stub code we have provided so that it performs naïve Bayesian learning and classification. Your other task is to simulate via pencil-and-paper this learning and classification task on a very simple data set. By first building the model in this way, the code you need to write will become much easier to understand. This plan of attack also provides you with some results with which to verify that your code works as you’ve intended.

Naïve Bayes

Recall that Naïve Bayes employs Thomas Bayes rule to calculate the class posterior:

$$P(y|\vec{x}) = \frac{P(\vec{x}|y)P(y)}{P(\vec{x})} = \frac{P(\vec{x}|y)P(y)}{\sum_y P(\vec{x}|y)P(y)} = \frac{P(y)\prod_{j=1}^{m} P(x_j|y)}{P(\vec{x})}$$

given some input $\vec{x} = \langle x_1, x_2, \cdots, x_j, \cdots, x_m \rangle$. The actual class that is assigned is chosen to be that value of $y$ who’s posterior probability is largest. The key assumption that naïve bayes makes is that feature values are class-conditionally independent; this allows $P(\vec{x}|y)$ to be replaced by the product over $P(x_j|y)$. Rarely is this assumption likely the case, and yet this method still tends to perform well in practice on certain domains.

Naïve Bayes doesn’t specify how $P(x_j|y)$ is actually learned. Here you will rely on the type of maximum likelihood estimate (MLE) that we saw in class for estimating $p$ from a series of coin flips. The only difference in this lab is that, some of the nominal features can take on 3 values, as opposed to 2; how can you naturally extend the MLE derivation from class to handle this situation?

Problem Statement

Lately, you’ve been stalking a very attractive tennis player (male or female, your pick), which is great fun but it takes about half an hour to arrive at the tennis courts, which can

---

1This problem, while taken from Tom Mitchell’s Machine Learning book, has been creepified out a bit for your edification.
be a bummer if you walk all that way only to discover they are not present. To save time,
you come up with plan. If you record what the weather was on the days that this person
played/didn’t play tennis, maybe you can predict when he/she will play in the future!

After patting yourself on the back for such a great idea, you proceed to collect weather
data over the next 14 days, conveniently store it in a file named tennis.dat. This data,
as well as its description (see tennis.info) are on the wiki. The data you collect includes
the following features: the overall feeling of the day (sunny, overcast, rainy), how warm it
is (cool, mild, hot), the humidity level (normal, high), and how hard the wind is blowing
(weak, strong). Out of 14 days, your little hottie played 9 of them.

On the 15th day it is sunny but cool, with high humidity and strong wind. After reflecting
on this rather bizarre SoCal weather, you realize that it’s almost 6pm and your tennis star
is about to take court (provided she/he is playing). So you ask your naïve bayes classifier
whether or not to run on down to the club to catch the game.

What’s the Answer?

Before continuing, build the corresponding naïve Bayes model by hand from tennis.dat.
Then use this model to derive what class conditional probabilities result given this bizarre
new day. Will your star grace the courts?

Implementation

Download the following four m-files: from the wiki:

1. preprocess.m is an auxiliary function that retuns some basic information concerning
the data set. learn uses this function to load the training data.

2. learn.m is a stub you need to add the code to for estimating the model’s parameters
from data.

3. classify.m is a stub that you need to add code to for using this model to estimate
the class of (some perhaps new) $\vec{x}$. Here you want to estimate both the posteriors and
the actual class (Why might the posteriors be useful? Do they require a little more
computation time to compute?)

4. test.m is where you will place the code that runs the “whole shib-ang”. After learning
commences, you will need to write code for evaluating your classifier’s performance.
See the Evaluation section for more.

The stub code is fairly well commented. Take some time to make sure you understand this
code before you start modifying it. The code you add should be as clean and well commented
as the code you found in there already (:-). As usual, start with the simplest approach (e.g.
for loops) before worrying about optimizing.
Evaluation

So, given your fancy naïve bayes code, does this help you get on with your stalking endeavor? Or, did you just waste an hour of your life?

Confusion Matrix

One way of evaluating a classifier algorithm is to create a *confusion matrix*—also known as a *contingency table*. This table displays the following information in a tabular format:

<table>
<thead>
<tr>
<th></th>
<th>True Positives (TP)</th>
<th>False Positives (FP)</th>
<th>False Negatives (FN)</th>
<th>True Negatives (TN)</th>
</tr>
</thead>
</table>

Any instance that is correctly classified as “Yes” is a *True Positive*. Similarly, any “Yes” instance that is misclassified as “No” is a *False Negative*. Any instance that is correctly classified as “No” is a *True Negative*; misclassifying it as a “Yes” is a *False Positive*. (Recall from Stats 101 that false positive and false negative correspond to “Type I” and “Type II” errors respectively.)

Contingency tables can only be used when data is supervised (why?). Also note that confusion matrices can be extended to include more rows and columns when more than just two classes exist. You will learn in future class lectures some of the measures of a classifier that can be derived from a confusion matrix.

Evaluating Naïve Bayes

Usually when conducting supervised learning experiments, you separate data into a *training set* and a *test set*. The *training set* is then the data that you use to learn the model, and the *test set* is an *independent* set of data used to evaluate how well this model does. (Why is the model not independent of the training data?)

In the pedagogical tennis example, there really isn’t enough data to warrant splitting into disjoint sets. Instead, you’ll create a contingency matrix based on the training data alone. (What can you conclude if this matrix shows few examples are properly classified? What if most of them are properly classified?)

Cool and Easy Extensions

The *preprocess* functionality makes this code very easy to extend. Thus, you should have very little trouble training this classifier on a more complicated nominal data set. For instance, try running it on the *mushroom data* from the *UCI repository* (you’ll first need to change the format of the mushroom data from letters to numeric values so that Matlab’s *load* or *csvread* function doesn’t choke). This data set is really cool because there is enough of it that you can take independent testing seriously (try running 10-fold stratified cross validation).
If you feel gutsy, you could extend your implementation to use either a KDE or a Gaussian distribution, which would enable your classifier to handle continuous values. The SpamBase database, which is also attainable from the UCI repository, might be a nice dataset to try.

Consider too that all features are not made equal. Some features will have more predictive power than others (we will explore this issue later in a future lecture). For instance, if it were windy but sunny, I’d likely play tennis, but if it were rainy with little wind, I probably wouldn’t play tennis. You might begin this analysis using ideas from information theory, principle components analysis, etc.

A key issue arises when there is no data available in the training set which could be used to answer a new question, i.e. a situation happens which has never happened before and we must classify it. For instance, suppose our data set didn’t have any data in which the humidity was high and the person played? Is it the case that the person would never play tennis in high humidity? Issues relating to how best to “generalize” when no “relevant” data has yet been seen is known as the zero-frequency problem. One way to combat against this issue is to “smooth out” these zero-probability areas by sampling new data from an underlying uniform distribution and spreading them across all possible outcomes. In more general Bayesian frameworks, such “hacks” can even be derived using additional assumptions.

Research

Naïve Bayes classifiers have been used in many different settings by many people. They were essentially the first mainstream probabilistic classification scheme and thus, have already received a great deal of fanfare (one might argue they are have become passe). However, there still are a couple areas where these classifiers are hot topics:

- Determining the structure of a bayes network using naïve bayesian reasoning has been researched at CSAIL (MIT AI Lab) in last couple years.

- Whenever a new dataset or style of data (DNA microarrays in the last couple years) becomes available, researchers rush to old favorites. If nothing else, they are almost guaranteed a paper :-).

- Combining other methods with naïve bayes is fairly common. Recently papers combining boosting (AdaBoost), n-gram language models, finite mixtures, and creative tree data structures have been published in academic and industry research journals. While it may not be monumentally original, some of the results obtained from these experiments seemed very solid.