The Naive Bayesian Classifier

Anders Holst
Erik Ylipää

August 7, 2015

In this assignment you will implement a *naive Bayesian classifier*. The naive Bayesian classifier is one of the most simple machine learning algorithms, which nevertheless gives surprisingly good results for many classification tasks. Furthermore, it is suitable in a streaming, distributed, massive data context, since it is a compact model that does not grow with the data, it can be incrementally updated as new data arrives, and it can be implemented in a distributed way. Here we will not deal with the distributed implementation, but it will support incremental updating in case of streaming data.

1 Theoretical background of the naive Bayesian classifier

The naive Bayesian classifier is a statistical machine learning method. The approach is to try to calculate the probability that a certain feature vector $\mathbf{x}$ belongs to a class C_k, $k \in K$, where K is the set of classes. To minimize the number of wrong classifications, we should then select the class with the highest probability as our classification of the feature vector. Below we denote the dimensionality of $\mathbf{x}$, i.e. the number of features, with d.

Formally the probability of class C_k given x is denoted:

$$P(C_k|\mathbf{x}).$$

To calculate the class probability, we first rewrite the probability using Bayes theorem:

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k)P(C_k)}{P(\mathbf{x})}$$

Next we note that we only need to find the class with the highest probability, and since the denominator $P(\mathbf{x})$ in the expression is the same for all classes when classifying a specific

feature vector $\mathbf{x}$, it has no effect on the relative ordering of the class probabilities, and can be ignored. So we remove it and change the equal sign into the symbol $\propto$ which means proportional to:

$$P(C_k|\mathbf{x}) \propto P(\mathbf{x}|C_k)P(C_k)$$

Here $P(C_k)$ is the probability of observing class C_k, i.e. the proportion of all entities to classify that belong to class C_k. $P(\mathbf{x}|C_k)$ is the probability distribution of the features $\mathbf{x}$ given that the entity belongs to class C_k. Although this distribution is easier to handle than $P(C_k|\mathbf{x})$ directly, it is problematic when the number of features is high, since a high dimensional probability distribution requires much more training samples (exponentially in the number of dimensions) to estimate reliably.

To solve this we make a simplifying "naive" assumption: that all the features are independent of each other (within each class). Then we can express the joint probability distribution over all features in the vector as a product of probability distributions for each feature separately:

$$P(C_k|\mathbf{x}) \propto P(C_k)\prod_i P(x_i|C_k)$$

This means that for each class, instead of trying to estimate an d-dimensional probability distribution (combinatorial complexity), we simplify it into estimating d 1-dimensional probability distributions, one for each feature. This is the naive Bayesian classifier.

Note that the classifier given above uses a series of multiplication of probabilities. Multiplying many small values represented as floating point values in the computer can quickly lead to underflow, which means that the result is smaller than the floating point format can handle and will be truncated due to rounding. A common way to handle this is to use the logarithm of the probabilities instead. That will turn the product of probabilities into a sum of log probabilities instead, which is much more numerically stable to use:

$$\log(\text{constant} \cdot P(C_k|\mathbf{x})) = \log(P(C_k)) + \sum_i \log(P(x_i|C_k)) \tag{1.1}$$

(Note that since the logarithm is a monotonically increasing function, the class with the highest probability is still the class with the highest value of the right hand side above, so there is usually no need to actually calculate the probability on the left hand side.)

The machine learning task now consists of two steps: First, given a training set of feature vectors and their corresponding class labels, estimate the probability distributions $P(C_k)$ and $P(x_i|C_k)$, and then use these estimations in Equation (1.1) to find the most probable class label for an unlabeled feature vector.

2 Estimating the probabilities

The naive Bayesian classifier can handle both discrete and continuous valued features. When the features x_i are discrete, we can estimate the probability $P(x_i|C_k)$ as the ratio that each value x_i takes to all values of x_i for a given class C_k. This essentially amounts to keeping a count of the times we have seen a particular value of x_i for each class. The same method can

be used to estimate the class probabilities $P(C_k)$, i.e as the number of occurrences of each class divided by the total number of seen samples:

$$\hat{P}(C_k) = \frac{n_k}{n_{tot}} \tag{2.1}$$

If x_i instead is continuous, we need to use a continuous probability density function to estimate the distribution. This is often done by using a Gaussian distribution since it only depends on the mean and variance of the feature, which is easy to compute. This is what we will use in this task, since all features are continuous valued in the selected data sets.

The probability for a value x_i given a class C_k using a Gaussian distribution is:

$$P(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_{ik}^2}} \exp(-\frac{(x_i - \mu_{ik})^2}{2\sigma_{ik}^2})$$

Where μ_{ik} and σ_{ik}^2 is the mean and variance for feature x_i in the class C_k. To estimate μ_{ik} we use the mean of the observed values of feature i in klass k:

$$\hat{\mu}_{ik} = \frac{\sum_j x_{ji}}{n_k}$$

where here j goes over all feature vectors belonging to class k.

Similarly, to estimate σ_{ik}^2 we use the unbiased sample variance:

$$\hat{\sigma}_{ik}^2 = \frac{\sum_j (x_{ji} - \hat{\mu}_{ik})^2}{n_k - 1}$$

However, we will need to have incremental versions of these estimates here: To make the classifier work in a streaming data context, the estimates should get gradually updated based on their previous values and the new observation. Iterative versions of the estimates look like:

$$\hat{\sigma}_{ik}^2 \leftarrow \begin{cases} \frac{n_k - 1}{n_k}\hat{\sigma}_{ik}^2 + \frac{(x_i - \hat{\mu}_{ik})^2}{n_k + 1} & n_k > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

$$\hat{\mu}_{ik} \leftarrow \hat{\mu}_{ik} + \frac{(x_i - \hat{\mu}_{ik})}{n_k + 1} \tag{2.3}$$

$$n_k \leftarrow n_k + 1 \tag{2.4}$$

(Note that all the right hand sides in the assignments above refer to the *old* values of the variables, so make sure to update them in the right order to not confuse old and new values.)

The above parameters, i.e. the estimated mean $\hat{\mu}_{ik}$ and variance $\hat{\sigma}_{ik}^2$ of each feature i for each class k, together with the estimate of the class probabilities $\hat{P}(C_k)$ of Equation (2.1), are the parameters to estimate from the training data.

3 THE ASSIGNMENT TASK

The assignment project is a Flink Scala project. The assignment files can be downloaded from the canvas web page. The project has a class called `StreamingClassifier` as its entrypoint. The class you have to modify for the assignment is `NaiveBayesModel` where you need to implement the following methods:

`update` This method is fed labeled samples. The update methods needs to update the mean, variance and counters the sample class, as shown in eqs. (2.2) to (2.4).

`predict` This method is fed unlabeled samples. The method should output the class label of the most likely class given the features. You need to use eqs. (1.1) and (2.1)

The places you need to add implementations are commented with the text:

```
// TODO: Write code here
```

3.1 TO SUBMIT

You should hand in the following for the assignment:

1. The file `NaiveBayesModel.scala` where you have implemented the `update` and `predict` functions.
2. The last output window from the file `prediction_output.txt` as produced by the command in 3.4. See figure 3.1 for an example of the output.

3.2 PREREQUISITES

You need to have flink and maven installed, if you are using the flink virtual machine, you should have the necessary prerequisites. To simplify the following commands, we set up some environmental variables below. If you are using the flink virtual machine ('eit-flink'), you can run the following:

```
$ export FLINK_PATH=/home/eit/flink/build-target
```

If you have your own flink installation, point the variable to the root which contains the flink installation (should have a bin directory).

3.3 BUILDING THE PROJECT

First you need to build the project. Navigate to the directory where you unzipped the project (the project directory is named `naive-bayes`) and set a environmental variable pointing to the current directory (which should be the project root):

```
$ export NB_PATH=$(pwd)
```

The project assumes you are using flink 0.10-SNAPSHOT. If you are using 0.9, you need to change the dependencies in the `pom.xml` file. Now compile the project and generate a `jar` using maven:

```
$ mvn clean package -Pbuild-jar
```

You will get a warning about POJO compliance, which can be ignored.

3.4 Running the project

Set an environmental variable for the data path. Make sure you are at the naive-bayes root directory. Then give the command:

```
$ export NB_DATA_PATH=$NB_PATH/data
```

Before running the project, start the flink runtime system:

```
$ $FLINK_PATH/bin/start-local.sh
```

The compiled package ended up in the target directory. Use flink to run it:

```
$ $FLINK_PATH/bin/flink run $NB_PATH/target/streamingbayes-0.1.jar \
    --training-data $NB_DATA_PATH/train.csv \
    --test-data $NB_DATA_PATH/test.csv \
    --output $NB_DATA_PATH/prediction_output.txt
```

This produces a text file with the performance of the classifier on the test data stream. The stream is consumed in a sliding window, and the model is applied to all samples in that window. After a window has been consumed, a summary is printed to the output file (`$NB_DATA_PATH/prediction_output.txt`). An example of such a window is given in figure 3.1.

The output shows how many samples the model has seen, as well as performance on the test data. The confusion matrix shows a count of what label the model assigns (along the columns), against the true label (the rows) of the matrix. A perfect predictor would give a matrix which is diagonal. Any off-diagonal counts are prediction errors.

Sometimes the model predicts data on the test stream before it has seen any training examples, in this case the parameters will be 0 or NaN.

Note: If the output file already exists, the execution will fail with a `java.io.IOException`. You have to either remove the existing file or give an output path which doesn't already exist.

3.5 The data

The data directory at the project root contains a number of comma separated value (CSV) files with data. The file `train.csv` and `test.csv` are used in the assignment. These are

```
================================================================================
Test window number: 4
Seen training examples: 391
Missclass ratio (incorrect predictions / total observations): 0,11500
Confusion matrix
              Predicted label
              _____0_____1
True label  0|   344    24
            1|    22    10
Class probabilities:
    0: 0,946
    1: 0,054
Class means:
    0: (49,514, 1,586, 39,992, 1,485, 0,955, 1,452, 10,328, 2,070, 1,237)
    1: (63,238, 3,619, 31,619, 4,169, 1,729, 3,074, 10,507, 3,281, 1,190)
Class variances:
    0: (583,400, 0,242, 29,486, 0,137, 0,195, 0,798, 13,313, 1,689, 2,418)
    1: (522,690, 90,123, 68,748, 91,936, 0,066, 0,487, 16,366, 4,600, 0,602)
================================================================================
```

Figure 3.1: Example of a single frame from the prediction output file on a dataset with 2 classes. In the confusion matrix we see that the classifier has problems with the positive class (label 1). It usually predicts that it belongs to the negative (label 0) class.

based on synthetic data generated from a gaussian mixture model with only 2 dimensions and 5 classes.

The python script `src/main/python/generate_data.py` can be used if you want to generate more test data. It depends on numpy.

The files `unknown_train.csv` and `unknown_test.csv` is a dataset based on real data, and is much harder to accurately classify.