Currency Prediction:

Machine Learning using Naïve Bayes

Group members

Masakazu Tanami <tanami@g.harvard.edu>
Ben Guild <bguild@fas.harvard.edu>
Ethan Brooks <ethanabrooks@gmail.com>
Kaleem Abdullah <kaleem.abdullah@gmail.com>

Overview

Our program applies the Naïve Bayes equation to the problem of predicting currencies. Our program performs the following commands (through the command line):

1. **Train**: trains the algorithm on training data.
2. **Predict**: using the Naïve Bayes equation, calculates probabilities of all possible classifications for a given input of currency data. Classifications are based on the general future direction of the data.
3. **Assess**: iterates over given currency data comparing the estimated classification with the actual classification and returning the percent of accurate classifications.

In our current implementation, we classify input data in three categories: "Up, Down, and Equal." The category Up, for example, would apply to the given currency data if some candle at a designated time (two minutes, for example) in the future trends upwards (closes higher than the given currency data). Our program calculates the probability that the given input data will fall into one of those three categories once that future time occurs.

Implementation

All three commands take the same four arguments on the command line: [c|d], which toggles between a discrete interpretation of attributes and a continuous interpretation (explained below), a currency type (e.g. USD), a currency to value it against (e.g. JPY) and a file containing currency values for the given currency pair.

The discrete and continuous interpretations treat attributes as discrete or continuous values respectively. To illustrate the difference, we consider the *volume* attribute. A discrete interpretation divides volume levels into three intervals: low, medium, and high. For any two records whose volume levels fall into the same interval, the *volume* attribute gets the same value. In contrast, a continuous interpretation the actual real-number volume levels of each data record. The mathematical treatment differs between interpretations and receives more attention in the Mathematical Foundations section.

Next we will describe the commands that the program takes in greater detail.

Train

Train takes the data from the input file and iteratively feeds it into an object called *nb_model* using a sliding window mechanism (overlapping data). *nb_model* has two implementations, one using discrete attributes and one using continuous attributes. As the discrete version of *nb_model* receives each "data record," it performs the following tasks:

1. classifies it and determines its attributes
2. tallies the number of records in each category
3. tallies the number of attribute/category intersections (e.g. data in the Up category with the "High Volume" attribute)
4. performs calculations on the data to prepare it for retrieval by the prediction module (details in the Mathematical Foundations section).

All values are stored in hash tables for quick insertion and lookup. The continuous version mostly performs the same tasks for steps 1 through 3, but for step 4, it calculates the mean and variation for each attribute within each category. (For example, for the attribute "third minute closing prices" and the category "up," it would calculate mean and variation for closing prices on the third minute of data records categorized as "Up.")

Once iteration is complete, the object writes itself to disk.

Predict

Instead of using the file as training data, *predict* scans the first data record and attempts to classify it. The first step in this process is to reload *nb_model* from the disk. Next, the program retrieves the data record to be classified and determines its attributes. For each possible category, and then for each of the determined attributes, it retrieves values from the hash tables in *nb_model* and calculates the probability of the data record belonging to that category (details in the Mathematical Foundations section). Finally, the program prints probabilities for each category to the console, sorted form least to greatest.

Assess

Assess simply iterates *predict* over all data records in the input file and calculates the percentage of accurate classifications. This can be helpful for testing different kinds of inputs to find which yield the strongest predictions.

Mathematical Foundations

As described in the Train section, *nb_model* tallies counts for each category/attribute intersection and for each category as a whole and then, in step 4, calculates certain values in preparation for prediction. The specific values that it calculates are $\log P(C)$ and $\log P(X_i|C)$ for all X_i and C where X_i is an attribute and C is a category.

In the discrete interpretation, $P(C)$ is simply the ratio of counts per category to total records. $P(X_i|C)$ is the ratio of counts in the intersection of attribute X_i and category C to counts in C as a whole, that is, $P(X_i \cap C)/P(C)$.

For the continuous interpretation, $P(C)$ is calculated the same way, but $P(X_i|C)$ is calculated as follows: as data records are added to *nb_model* during initial training, instead of simply tallying counts for each attribute/category intersection, *nb_model* calculates the mean μ and variation σ^2 of each.[1] For example, for the volume/up intersection, we take all the volume levels for records in the up category and calculate the aforementioned values, μ and σ^2. We then assume normal distribution and apply the following probability density function:

$$P(X_i|C) = P(y) = \frac{e^{-(y-\mu)^2/(2\sigma^2)}}{\sqrt{2\pi\sigma^2}}$$

Now that we have $P(C)$ and $P(X_i|C)$ for all possible categories C and attributes X_i, how do we calculate the desired value $P(C|X)$, the probability of a category given a set of attributes X? Let $X = X_1 \cap \ldots \cap X_n$ and let $\{C_1, \ldots, C_k\}$ describe the set of all possible categories (in our implementation, Up, Down, and Equal). Then the following is true:

$$P(C|X) = \frac{P(X \cap C)}{P(X)} = \frac{P(X \cap C)}{\sum_{i\in(1\ldots k)} P(X \cap C_i)}$$

Now assuming conditional independence of all attributes and applying Naïve Bayes, we get

$$P(X \cap C) = P(C) \prod_{j\in(1\ldots n)} P(X_j|C) = \exp\left(\log P(C) + \sum_{j\in(1\ldots n)} P(X_j|C)\right)$$

With this in mind, the values that are written to disk and that our program retrieves from *nb_model* while executing *predict* are $\log P(C)$ and $\{\, P(X_j|C) : j \in (1 \ldots n)\}$. They are then fed into the *Predict* module and into the previous equation, yielding the desired result.

Design

We had two primary goals with this project: first, maximize performance through application of machine learning principles and appropriate use of data structures; second, maximize abstraction in order to enable modularity and extensibility.

Performance

We designed our application in such as way that most of the calculations are done at the time of training the model using historical data, and storing the results of those calculations within the data model. This way various data values needed for the NB formula would be readily available at the time of prediction. To maximize performance even further, we made use of hash tables. Since all prediction-time

[1] The algorithm used for computing mean and variance in the Statistics module is derived from the following source:
Hoemmen, Mark. N.d. MS, CS 194-2: Parallel Programming. University of California, Berkeley. Computing the Standard Deviation Efficiently, Aug.-Sept. 2007. Web. 28 Apr. 2015.
<http://www.cs.berkeley.edu/~mhoemmen/cs194/Tutorials/variance.pdf>.

interactions with our data model involved look-ups, it was clear that a hash table was the best structure from a performance perspective because it performs look-ups in constant time.

To avoid repetition of training cycle for the same training data, we also added functionality in the model for it to save itself to the disk and rebuild itself from the store data.

Abstraction

By minimizing the “knowledge” each module had of the others, we gave ourselves freedom to modify features in each module without affecting the rest of the program. The *DataRecord* module, which handles all interaction with data pulled from input files, provides an illustration of this.

Because we did not know which scheme of assigning attributes would yield the best predictions, we wanted to maximize abstraction for this mechanism. We therefore designed a module type with a very simple interface consisting of a single value: a list of functions that take a data record and return a list of attribute ids. Each of these functions evaluates a data record for an attribute and returns an associated id with information about the attribute type and its value. These attribute types and values are then fed to the NB model, which stores them in its internal hash tables without any dependency on what the actual attribute type and values are. This data structure maximizes extensibility because if a new data attribute is identified to increase the predictability (e.g. an external attribute such as overall market direction, time of the year, gold price, etc) then that would required only the addition of a new function to determine the type and value of the new attribute, without making any change in the underlying NB model code. Moreover, the simplicity of the function interface (record -> id) allows for many different approaches to assigning attributes.

A second example of our approach to abstraction was our approach to selectively exposing information about the details of data records. On the one hand, the category and attribute modules needed complete access to that information, but on the other, all other parts of the program only needed very limited access. Limiting access was especially important since data records were a part of the code that we especially wanted to be extensible as described above. We therefore chose to place the Attribute and Category modules inside the Data module. This was possible because we had already designed the program so that the only module that interacted with these was the Data module.