Naive Bayes

Albert Mata

25/12/2018

Contents

Probabilistic Learning - Classification Using Naive Bayes

Bayesian methods have been used with success for:

- Text classification, such as junk e-mail (spam) filtering.
- Intrusion or anomaly detection in computer networks.
- Diagnosing medical conditions given a set of observed symptoms.

Bayesian classifiers work well when the information from numerous attributes should be considered simultaneously in order to estimate the overall probability of an outcome. Many machine learning algorithms ignore features that have weak effects, but Bayesian methods do not, because if large number of features have relatively minor effects, their combined impact can be quite large.

Bayesian classifiers rely on Bayesian probability theory, which is based on the idea that the estimated likelihood of an event, or a potential outcome, should be based on the evidence at hand across multiple trials, or opportunities for the event to occur.

The probability of an event is estimated from the observed data by dividing the number of trials in which the event occurred by the total number of trials. We denote this probability using notation $P(spam) = 0.4$. If a trial has two **mutually exclusive and exhaustive** outcomes, we can calculate the other just as $P(ham) = 1 - 0.4 = 0.6$. An event is always mutually exclusive and exhaustive with its complement (denoted A^C, A', or $\neg A$).

We may want to know the probability of the intersection of two events as in $P(spam \cap Viagra)$, that is the probability of the two events both occurring. **Venn diagrams** are useful to represent them. This probability depends on the **joint probability** of the two events or how the probability of one is related to the probability of the other. If the two events are totally unrelated, they are called **independent events**. When all events are independent, it's impossible to predict one event by observing another: **dependent events** are the basis of predictive modeling.

For independent events, $P(A \cap B) = P(A) \times P(B)$. So, if a text message containing the word "Hello" has $P(Hello) = 0.2$ and that message being spam has $P(spam) = 0.4$, and both events are not related (that is, independent), then probability of both events happening at the same time is $P(spam \cap Hello) = 0.4 \times 0.2 = 0.08$.

But this is different for $P(spam \cap Viagra)$, as we know these are highly dependent events. In cases like this, we use **Bayes' theorem**:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(B|A) \cdot P(A)}{P(B)}$$

$P(A|B)$ means the probability of event A given that event B occurred (**conditional probability**). In plain language, this tells us that if we know event B occurred, the probability of event A is higher the

more often that A and B occur together each time B is observed.

In our spam filtering example:

- Without any further knowledge, the best estimate of spam status would be $P(spam)$, so 40 percent. This estimate is known as the **prior probability**.
- The probability that the word Viagra was used in previous spam message, $P(Viagra|spam)$, is called the **likelihood**.
- The probability that Viagra appeared in any message at all, $P(Viagra)$, is known as the **marginal likelihood**.

We can calculate the **posterior probability** $P(spam|Viagra)$ using Bayes' theorem and **frequency and likelihood tables** (based on recorded numbers):

$$P(spam|Viagra) = \frac{P(Viagra|spam) \cdot P(spam)}{P(Viagra)}$$

The Naive Bayes algorithm

This algorithm is really named as such because it makes some "naive" assumptions about the data that are rarely true in most real-world applications, i.e. considering all the the features in the dataset equally important and independent. But the algorithm often works pretty well even when these assumptions are violated.

Strengths:

- Simple, fast and very effective.
- Does well with noisy and missing data.
- Requires relatively few examples for training, but also works well with very large number of examples.
- Easy to obtain the estimated probability for a prediction.

Weaknesses:

- Relies on an often-faulty assumption of equally important and independent features.
- Not ideal for datasets with many numeric features.
- Estimated probabilities are less reliable than the predicted classes.

Classification with Naive Bayes

We can train a Naive Bayes learner by constructing a likelihood table for the appearance of some words (i.e. W_1, W_2, W_3 and W_4) in a set of 100 messages.

When we then want to decide if a new message is spam or not, we need to calculate the posterior probability given the likelihoods of the words found in that new message. For instance if the new

message contains words W_1 and W_4 but not W_2 or W_3:

$$P(spam|W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) = \frac{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4|spam) \cdot P(spam)}{P(W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4)}$$

This is computationally difficult to solve (specially when we have more and more features). But it becomes much easier as Naive Bayes assumes class-conditional independence among events, that is, it assumes events are independent so long as they are conditioned on the same class value. So we can reduce previous calculation to:

$$P(spam|W_1 \cap \neg W_2 \cap \neg W_3 \cap W_4) \propto P(W_1|spam)P(\neg W_2|spam)P(\neg W_3|spam)P(W_4|spam)P(spam)$$

And so, using values from likelihood tables we can get the overall likelihood of spam (say 0.012) and, with a similar calculation, the likelihood of ham (say 0.002). Finally, to convert these numbers into probabilities we just do as follows:

$$P(spam) = \frac{0.012}{0.012 + 0.002} = 0.857$$
$$P(ham) = \frac{0.002}{0.012 + 0.002} = 0.143$$

As a summary, we begin by building a frequency table, use it to build a likelihood table, and multiply the conditional probabilities according to the Naive Bayes' rule. Finally, we divide by the total likelihood to transform each class likelihood into a probability.

The Laplace estimator

Because probabilities in the Naive Bayes formula are multiplied in a chain, if an event never occurs for one or more levels of a class, this 0 causes the posterior probability of that class to be zero no matter what.

We use the **Laplace estimator** to fix this. The idea is as simple as adding a small number (typically 1, but can be any number and does not need to be the same for every feature) to each of the counts in the frequency table to ensure that each feature has a nonzero probability of occurring with each class.

Using numeric features with Naive Bayes

Numeric features don't work well in the Naive Bayes algorithm, as it uses frequency tables to learn the data. To fix this, we need to **discretize** numeric features, putting numbers into categories known as **bins**. To do that, we can explore the data for natural categories or **cut points** or we can use **quantiles**.

We must keep in mind that discretizing a numeric feature results in a reduction of information. Too few bins can result in important trends being obscured. Too many bins can result in small counts in the frequency table and can increase sensitivity to noisy data.

Example - filtering mobile phone spam with the Naive Bayes algorithm

Step 1 - collecting data

We'll take the data from the SMS Spam Collection (Hidalgo, Almeida, and Yamakami 2012).

This dataset includes the text of SMS messages along with a lavel indication whether the message is spam or ham. Having a look at some messages we see some patterns:

- Spam messages often use the word "free", which is not so common in ham messages.
- Ham messages some times cite specific days of the week, which is not common in spam messages.

Our Naive Bayes classifier will compute the probability of spam and ham given the evidence provided by all the words in the message.

Step 2 - exploring and preparing the data

We need to transform our data into a representation known as **bag-of-words**.

```
sms_raw <- read.csv("sms_spam.csv", stringsAsFactors = FALSE)
str(sms_raw)
```

```
## 'data.frame':    5559 obs. of  2 variables:
##  $ type: chr  "ham" "ham" "ham" "spam" ...
##  $ text: chr  "Hope you are having a good week. Just checking in" "K..give back my thanks." "Am also doir
```

The first variable (`type`) has been coded as either `ham` or `spam`. The second variable (`text`) stores the full raw SMS text. As we already know, machine learning algorithms like the class to predict to be coded as a factor:

```
sms_raw$type <- factor(sms_raw$type)
table(sms_raw$type)
```

```
##
##  ham spam
## 4812  747
```

We can see about 13% of SMSs in our data are labeled as spam.

• Data preparation - cleaning and standardizing text data

We use the `tm` text mining package to get from messages with spaces, punctuation and uninteresting words (*and*, *but*, *or*…) to some form of data we can work with.

The first step in processing text data involves creating a **corpus**, which is a collection of text documents. We can use either `VCorpus()` for a volatile -stored in memory- corpus or `PCorpus()` for a permanent -stored in a database- one.

```
sms_corpus <- VCorpus(VectorSource(sms_raw$text))
inspect(sms_corpus[1:2])
```

```
## <<VCorpus>>
## Metadata:  corpus specific: 0, document level (indexed): 0
## Content:  documents: 2
##
## [[1]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 49
##
## [[2]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 23
```

```
lapply(sms_corpus[1:2], as.character)
```

```
## $`1`
## [1] "Hope you are having a good week. Just checking in"
##
## $`2`
## [1] "K..give back my thanks."
```

Then we need to clean the text and divide these messages (documents) into individual words.

```
# Lowercase everything
sms_corpus_clean <- tm_map(sms_corpus, content_transformer(tolower))
as.character(sms_corpus[[1]])
```

```
## [1] "Hope you are having a good week. Just checking in"
```

```
as.character(sms_corpus_clean[[1]])
```

```
## [1] "hope you are having a good week. just checking in"
```

```
# Remove all numbers
# getTransformations() shows all available functions like removeNumbers()
sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers)
```

```
# Remove filler words (stop words)
# We can use stopwords(), or specify a language, or use a custom vector of words
sms_corpus_clean <- tm_map(sms_corpus_clean, removeWords, stopwords())
```

```
# Remove punctuation
sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation)
```

```
# Stemming (learned, learning, learns -> learn)
library(SnowballC)
sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument)
```

```
# Remove additional whitespace
sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace)
```

With all this process we get a final result as follows:

```
as.character(sms_corpus[[1]])
```

```
## [1] "Hope you are having a good week. Just checking in"
```

```
as.character(sms_corpus_clean[[1]])
```

```
## [1] "hope good week just check"
```

• Data preparation - splitting text documents into words

We now need to split the messages into individual components through a process called **tokenization** using `tm`'s `DocumentTermMatrix()` function, which creates a data structure where rows indicate documents (SMSs) and columns indicate terms (words). Each cell in this matrix stores a number indicating a count of the times the word represented by the column appears in the document represented by the row (so, the vast majority of the cells in the matrix are filled with zeros and that's why it's called a **sparse matrix**).

```
sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
```

We could create a similar matrix from raw data in just one step, but we will get slightly different results, as operations will be performed in a different order. So, in general, we need to know what we want to achieve in order to choose one approach or the other.

```
sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
  tolower = TRUE,
  removeNumbers = TRUE,
  stopwords = TRUE,
  removePunctuation = TRUE,
```

```
  stemming = TRUE
))
```

• Data preparation - creating training and test datasets

We need to split the data intro training and test datasets, but it is important that the split occurs only after the data have been cleaned and processed, as we need exactly the same preparation steps to occur on both datasets.

```
# Data is already in a random order
sms_dtm_train <- sms_dtm[1:4169, ]   # first 75%
sms_dtm_test <- sms_dtm[4170:5559, ] # remaining 25%

# Labels (from raw data as they are not stored in the DTM)
sms_train_labels <- sms_raw[1:4169, ]$type
sms_test_labels <- sms_raw[4170:5559, ]$type
```

Let's compare the proportion of spam in the training and test data frames:

```
prop.table(table(sms_train_labels))
```

```
## sms_train_labels
##       ham      spam
## 0.8647158 0.1352842
```

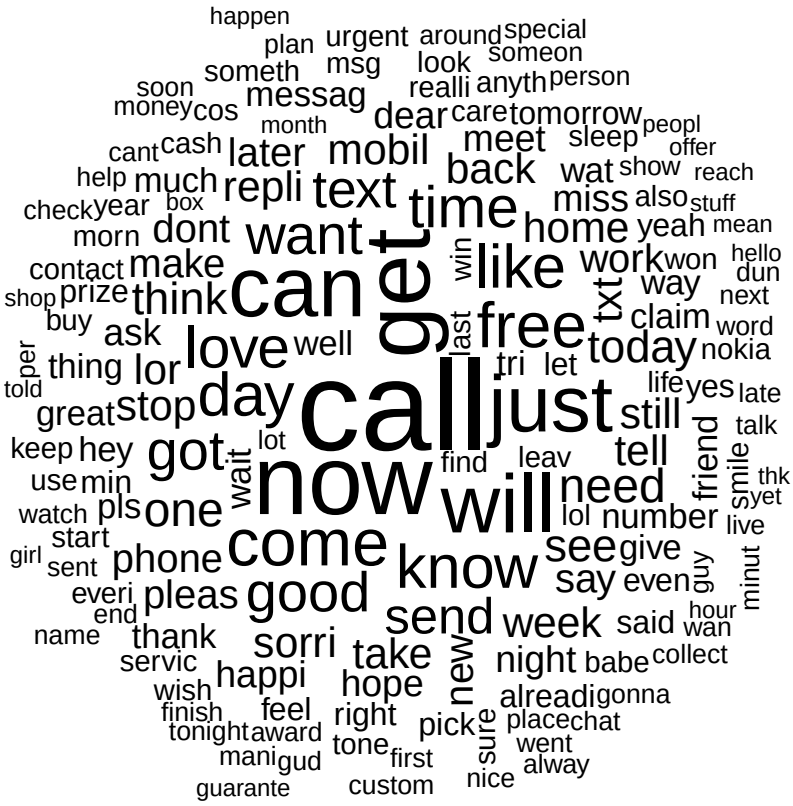
```
prop.table(table(sms_test_labels))
```

```
## sms_test_labels
##       ham      spam
## 0.8683453 0.1316547
```

• Visualizing text data - word clouds

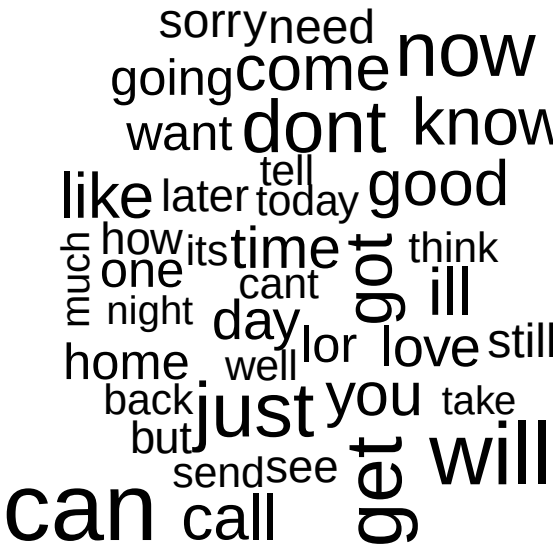
A **word cloud** is a way to visually depict the frequency at which words appear in text data. The `wordcloud` package has a function to create it.

```
library(wordcloud)
wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE)
```



We can create one word cloud for spam and another for ham and compare them to see if we can identify some patterns:

```
spam <- subset(sms_raw, type == "spam")
ham <- subset(sms_raw, type == "ham")
par(mar = c(1, 1, 1, 1))
par(mfrow = c(1, 2))
wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))
wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
```



The obvious differences in the most common words in each of the clouds suggest that our Naive Bayes model will have some strong key words to differentiate between the classes.

● Data preparation - creating indicator features for frequent words

The final step in data preparation is to transform the sparse matrix into a data structure that can be used to train a Naive Bayes classifier. Currently, the sparse matrix includes more than 6,500 features. It's unlikely that all of these are useful for classification. We will eliminate any word that appear in less than five messages (~0.1%):

```
sms_freq_words <- findFreqTerms(sms_dtm_train, 5)
str(sms_freq_words)
```

```
##  chr [1:1137] "£wk" "abiola" "abl" "abt" "accept" "access" "account" ...
```

This lets us reduce our training and test datasets from more than 6,500 features to the 1,137 appearing in at least five messages:

```
sms_dtm_freq_train <- sms_dtm_train[, sms_freq_words]
sms_dtm_freq_test <- sms_dtm_test[, sms_freq_words]
```

Finally, the Naive Bayes classifier needs to know whether or not a word appears in a message, not how many times:

```
convert_counts <- function(x) {
  x <- ifelse(x > 0, "Yes", "No")
}

sms_train <- apply(sms_dtm_freq_train, MARGIN = 2, convert_counts)
sms_test  <- apply(sms_dtm_freq_test, MARGIN = 2, convert_counts)
```

Step 3 - training a model on the data

We will use the Naive Bayes implementation from the `e1071` package (but we could also use `NaiveBayes()` from `klaR`). First, we build our model on the `sms_train` matrix:

```
library(e1071)
sms_classifier <- naiveBayes(sms_train, sms_train_labels)
```

This `sms_classifier` object now contains a `naiveBayes` classifier object that can be used to make predictions.

Step 4 - evaluating model performance

We use our classifier to generate predictions and then compare them to the true values:

```
sms_test_pred <- predict(sms_classifier, sms_test)

library(gmodels)
CrossTable(sms_test_pred, sms_test_labels,
           prop.chisq = FALSE, prop.t = FALSE,
           dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1390
##
##
##              | actual
##    predicted |       ham |      spam | Row Total |
## -------------|-----------|-----------|-----------|
##          ham |      1201 |        30 |      1231 |
##              |     0.976 |     0.024 |     0.886 |
##              |     0.995 |     0.164 |           |
## -------------|-----------|-----------|-----------|
##         spam |         6 |       153 |       159 |
```

```
##              |     0.038 |     0.962 |     0.114 |
##              |     0.005 |     0.836 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      1207 |       183 |      1390 |
##              |     0.868 |     0.132 |           |
## -------------|-----------|-----------|-----------|
##
##
```

Only 36 of the 1,390 messages have been incorrectly classified (2.6%): 6 were misidentified as spam and 30 were incorrectly labeled as ham. This is an impressive performance that exemplifies why Naive Bayes is the standard for text classification. However, those 6 ham messages considered as spam could cause significant problems, so we should investigate further to see if we can achieve better performance.

Step 5 - improving model performance

We didn't set a value for the Laplace estimator while training our model. This allows words that appeared in zero spam or zero ham messages to have an indisputable say in the classification process. Let's see what happens if we set a Laplace estimator this time:

```
sms_classifier2 <- naiveBayes(sms_train, sms_train_labels, laplace = 1)
sms_test_pred2 <- predict(sms_classifier2, sms_test)
CrossTable(sms_test_pred2, sms_test_labels,
           prop.chisq = FALSE, prop.t = FALSE,
           dnn = c('predicted', 'actual'))
```

```
##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |-------------------------|
##
##
## Total Observations in Table:  1390
##
##
##              | actual
##    predicted |       ham |      spam | Row Total |
## -------------|-----------|-----------|-----------|
```

```
##          ham |      1202 |        28 |      1230 |
##              |     0.977 |     0.023 |     0.885 |
##              |     0.996 |     0.153 |           |
## -------------|-----------|-----------|-----------|
##         spam |         5 |       155 |       160 |
##              |     0.031 |     0.969 |     0.115 |
##              |     0.004 |     0.847 |           |
## -------------|-----------|-----------|-----------|
## Column Total |      1207 |       183 |      1390 |
##              |     0.868 |     0.132 |           |
## -------------|-----------|-----------|-----------|
##
##
```

We experience a small but significant improvement. We need to tweak the model carefully to maintain a balance between being overly aggressive and overly passive while filtering spam (and in general in any scenario when dealing with false positives and false negatives).

References

Hidalgo, Jose Maria Gomez, Tiago A Almeida, and Akebo Yamakami. 2012. "On the Validity of a New Sms Spam Collection." In *Machine Learning and Applications (Icmla), 2012 11th International Conference on*, 2:240–45. IEEE.

Lantz, Brett. 2015. *Machine Learning with R*. Packt Publishing Ltd.