Machine Learning for Document Classification

April, 2019

1 Document Classification

The document classification task can be framed as follows: given a set of documents $d_0, d_1, \ldots, d_n$ and a set of labels $c_0, c_1, \ldots, c_n$ assigned from some prespecified set C, create a classifier that can take a new document, d, and predict the appropriate class ($c \in C$) to which this document belongs.

For concreteness, let's consider the task where we're given a set of documents from three classes: Deeds of Trust (DT), Liens (L), and Deeds of Reconveyance (DR). That is:

$$C = \{DR, DT, L\}$$

From each class, we have a set of associated documents $D_c = \{d_{c_1}, d_{c_2}, \ldots, d_{c_n}\}$ where n here is defined on a per class basis. So, we might have three Liens $\{d_{L_1}, d_{L_2}, d_{L_3}\}$ and two Deeds of Trust $\{d_{DT_1}, d_{DT_2}\}$.

Each document is a potentially unique sequence of characters. Generally, however, for learning we don't want to use the document directly, but instead we want to create some more abstract representation upon which to base learning.

2 The Bag of Words Model

The "Bag of Words" model is commonly used to represent text documents of various sorts. The main advantage of the model is its simplicity. As the name implies, in the Bag of Words model, each document is modeled as a Bag (i.e., a set that allows duplicate entries). The model is often implemented as an associative array (map) between words from some dictionary ($w \in \mathcal{D}$) and numeric values. For example, we could create a map that contains each possible word in the english language as a key, and a numeric value for each key indicating how many times that word occurs in a given document. Since $\mathcal{D}$ may be much larger than any given document, it is more typical to include keys in the associative array/map only if they have non-zero values.

Often, the dictionary $\mathcal{D}$ is created simply by tokenizing all the documents in the training set. $\mathcal{D}$ then is simply the set of tokens discovered. Given the document:

```
amongst naive methods, Naive Bayes is good.
```

This approach would lead to a feature vector below:

```
{"amongst": 1, "naive":1, "methods,": 1,
 "Naive":1, "Bayes":1, "is":1, "good.":1}
```

Notice that this model distinguishes between `naive` and `Naive` and also includes punctuation on some words. We can increase generalization by collapsing some of these features. For example, we might choose to remove capitalization. However, we need to be a bit careful about reducing the feature space too much as we may loose information that is relevant to classification.

Some common ways to reduce the feature space (i.e., $\mathcal{D}$) is by:

- Changing text to lower case
- Removing punctuation
- Removing non-words
- Stemming (i.e., removing suffixes like "ing" or "ed")
- Removing stop words (words that have little or no information value because of their ubiquity like "and" or "the")
- Removing very infrequently occurring words (unique strings, typos, etc)

Many of these steps (the first four, or five above), can be done without any knowledge about the properties of the context (other than the parent language). That is to say, there is no need to analyze the properties of the training set before performing many of the operations above. In this sense, these can be viewed as potential preprocessing steps that can be applied even before training has begun[1].

In contrast, removing very infrequently occurring words probably makes most sense to do in a context sensitive way: that is, based on properties of the training data as opposed to properties of the english language itself. There are two arguments for this reasoning: (1) the training set may contain a vocabulary that is not representative of the parent language in general (e.g., the training data might only contain legal or scientific documents); (2) the training set may contain a variety of tokens (typos, unique identifiers, product numbers, etc) that cannot be anticipated by a general model of the parent language. We could make a similar argument for identifying stop words, and choose to select stop words based on features of the training set (e.g., words that occur in 99% of the the documents) instead of a general model of the parent language.

[1]You can obtain lists of stopwords or stemming algorithms easily from the internet.

3 From Bag of Words to Feature Vectors

Once we've decided on a set of words $\mathcal{D}$ with which to describe documents, it is easy to implement a Bag of Words model using a set or a map. In the simplest case, our model might simply tell us whether a given word $w \in \mathcal{D}$ is in document d. This model is known as a "boolean" bag of words and given the preprocessed text (with text transformed to lower case and punctuation removed):

```
amongst naive methods naive bayes is good
```

it would generate the following feature set:

```
{"amongst", "naive", "methods", "bayes", "is", "good"}
```

A more general Bag of Words model is constructed with a map between words and numeric values. Common numeric values are based on either the number of occurrences of a given word, or the relative frequency of a given word (number of occurrences divided by total words in the document). The former approach would lead to the following feature vector:

```
{"amongst":1, "naive":2, "methods":1, "bayes":1, "is":1, "good":1}
```

Of course, it is obviously possible to use this same representation for a boolean bag of words.

4 Naive Bayes

Bayes' Theorem is generally stated as:

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

in the context of document classification, we are interested in knowing

$$P(c_j|d_i) = \frac{P(d_i|c_j)P(c_j)}{P(d_i)} \tag{1}$$

where c_j is a class to which the document d_i might belong. Since we are generally interested in which class is most likely, we can drop the denominator and simply find the class c^* with the highest value.

$$c^* = \arg\max_j P(d_i|c_j)P(c_j) \tag{2}$$

While it is clear that we can estimate $P(c_j)$ by counting the fraction of training documents from each set in C, it is not entirely obvious how to get from our feature vectors to a value for $P(d_i|c_j)$. Indeed, methods for calculating $P(d_i|c_j)$ vary. Below, we describe the two most common approaches.

4.1 Example Documents

For each of the approaches below, we'll examine data from the following one-line documents:

```
this deed is a deed of trust for a house
a deed of trust for the wallace house
a trust deed for wells fargo

a lien against a house is a lien
a lien placed on a house for a plumber
a lien on a deed removed
```

Assume that the first three documents belong to the class DT, and the second three documents belong to the class L. If we take $\mathcal{D}$ to be all the words encountered in the documents that are longer than 2 characters, we will create the following term-frequency feature vectors:

```
{this:1, deed:2, trust:1, for:1, house:1}
{deed:1, trust:1, for:1, the:1, wallace:1, house:1}
{trust:1, deed:1, for:1, wells:1, fargo:1}

{lien:2, against:1, house:1}
{lien:2, placed:1, house:1, for:1, plumber:1}
{lien:1, deed:1, removed:1}
```

4.2 Multi-variate Bernoulli Model

In the multi-variate Bernoulli model[2], a document is modeled as a set of independent choices in which each choice determines whether or not a particular feature is observed. Here, a document d_i is represented as a binary/boolean feature vector indicating whether or not a word w from our dictionary of terms $\mathcal{D}$ occurs in d_i. Under the naive assumption the occurrence of each term is independent. Thus:

$$P(d_i|c_j) = \prod_{k=1}^{|\mathcal{D}|} P(w_k|c_j) \tag{3}$$

That is, the probability of d_i given the class c_j is a product of the probability of observing each word w_k (given the class c_j) from our dictionary of terms $\mathcal{D}$. Note that w_k is either 0 or 1, and thus there are two distinct values of $P(w_k|c_j)$ (one indicating the term's presence, the other indicating the term's absence). More over, their sum must equal 1. In the multi-variate Bernoulli model, $P(w_k|c_j)$ is estimated by computing the fraction of documents in the

[2]This is the model required for an initial strategy in Project 2.

w	$P(w\|DT)$	$P(w\|L)$
deed	3/3	1/3
lien	0/3	3/3
this	1/3	0/3
trust	3/3	0/3
for	3/3	1/3
house	2/3	2/3
the	1/3	0/3
wallace	1/3	0/3
wells	1/3	0/3
fargo	1/3	0/3
plumber	0/3	1/3
removed	0/3	1/3
placed	0/3	1/3
against	0/3	1/3

Table 1: Term liklihoods using data from Section 4.1 after conversion to boolean feature vectors.

training set that belong to c_j and contain the word w_k. $P(\neg w_k|c_j)$ can then be obtained by calculating $1 - P(w_k|c_j)$ or by counting the documents in c_j that lack an occurrence of w_k.

Using the feature vectors from Section 4.1 (after converting to boolean values), we would compute probabilities in Table 1. Note that the table doesn't explicitly show values for $P(\neg w|DT)$ or $P(\neg w|L)$ because they can be computed directly as $1 - P(w|DT)$ and $1 - P(w|L)$ respectively.

We could then see if $P(d_i|DT)/P(d_i|L)$ was greater or less than 1 to predict the most likely classification. If we were given the document, d:

```
I have a house deed for that house
```

We would obtain the *boolean* feature vector: {`deed:1, for:1, house:1`}. Notice that since our dictionary is limited to words in the training set (as opposed to the english language) *have* and *that* don't occur in the feature vector. Using our model above, we would calculate:

This is wrong!

This is wrong... it needs to include the terms that haven't appeared in the example document as well.

$$\frac{P(DT|d)}{P(L|d)} = \frac{P(DT)P(deed|DT)P(for|DT)P(house|DT)}{P(L)P(deed|L)P(for|L)P(house|L)}$$

$$\frac{P(DT|d)}{P(L|d)} = \frac{(\frac{3}{6})(\frac{3}{3})(\frac{3}{3})(\frac{2}{3})}{(\frac{3}{6})(\frac{1}{3})(\frac{1}{3})(\frac{2}{3})} = 9$$

4.3 Multinomial Model

In the multinomial model, a document is modeled as a set of words pulled out of a hat (with replacement). Replacement here is used in the traditional sense of probability/statistics literature which is to say that if you can pull a word w from the hat with probability p at time t, then you can pull the word from the hat again later (at time $t+1$) with the same probability. In other words, the hat replenishes the items as they are pulled out.

In this model:

$$P(d_i|c_j) = P(|d_i|)|d_i|! \prod_{k=1}^{|\mathcal{D}|} P(w_k|c_j)^{n_{ik}} \tag{4}$$

That is, the probability of the document d_i given the class c_j is a function first, of the document length: $P(|d_i|)|d_i|!$. However, if we're mainly interested in the most likely class, then these terms will be identical regardless of the choice of class c_j, and so, we treat them as a part of the normalization constant. The other part of the expression looks similar to that of the multivariate Bernoulli model, except that here, $P(w_k|c_j)$ is raised to the power n_{ik} which is the number of times w_k is seen in document d_i.

Unlike the multivariate Bernoulli model where a document is modeled as a boolean feature vector, here a document is modeled as a term-frequency vector where each element in the feature vector indicates how many time that particular term appears in the specified document. $P(w_k|c_j)$ is now estimated by counting the number of times word w_k appears amongst all words in c_j. In other words, if you put all the documents from c_j together in a single "mega-document", $P(w_k|c_j)$ would simply indicate how many times w_k appears in the mega-document divided by the total number of words in that document.

Computing probabilities with the multinomial model can be difficult because of the sizes of numbers involved. $|d_i|!$ can easily become very large, while $P(w_k|c_j)^{n_{ik}}$ can quickly become extremely small as n_{ik} grows. For this reason, many implementations use logarithmic values for each of these terms.

This means that addition is used in place of multiplication (i.e., $\log(xy) = \log(x) + \log(y)$) and helps deal with issues of floating point underflow.

$$\begin{aligned}
P(L) &= 3/6 \\
P(DT) &= 3/6 \\
P(lien|L) &= 5/13 \\
P(lien|DT) &= 0/17 \\
P(deed|DT) &= 4/17 \\
P(deed|L) &= 1/13
\end{aligned}$$

Note that unlike the multivariate model, there is no entry corresponding to the *absence* of a word. We only concern ourselves here with the words that *do* appear. As before, we can see if $P(d_i|DT)/P(d_i|L)$ was greater or less than 1 to predict the most likely classification. If we were given the document, d:

```
I have a house deed for that house
```

We would obtain the term-frequency feature vector: {`deed:1, for:1, house:2`}.

$$\begin{aligned}
\frac{P(DT|d)}{P(L|d)} &= \frac{P(DT)P(deed|DT)^1 P(for|DT)^1 P(house|DT)^2}{P(L)P(deed|L)^1 P(for|L)^1 P(house|L)^2} \\
\frac{P(DT|d)}{P(L|d)} &= \frac{(\frac{3}{6})(\frac{4}{17})(\frac{3}{17})(\frac{2}{17})^2}{(\frac{3}{6})(\frac{1}{13})(\frac{1}{13})(\frac{2}{13})^2} > 1
\end{aligned}$$

4.4 Incremental Training of Naive Bayes

Both Naive Bayes models above have the useful property that they can be trained incrementally. That is, one can give the training procedure a single document d_i, and information from this document can be incorporated into the naive bayes model without reprocessing (or even retaining) any prior training documents. Below, we'll illustrate how this happens with the multi-variate Bernoulli model.

In the multi-variate Bernoulli model, we need to obtain the estimates $P(w_k|c_j)$ and $P(c_j)$ in order to calculate $P(d|c_j)$ and ultimately make a decision about the most likely class for an unseen document. Once we're given a dictionary of terms $\mathcal{D}$, we can begin to create boolean feature vectors to describe each document d_i. As we process each document d_i, we can simply keep a running total of the number of documents we've seen in total, and the number of documents we've seen in each particular class. Thus, at any point in the training process, we can easily estimate $P(c_j)$ and once training is complete, this value agrees with what we would have calculated if we had performed the analysis in batch.

Similarly, we can maintain a set of values for each word $w \in \mathcal{D}$ that indicates how many documents from each class contain w. This would allow us to calculate $P(w_k|c_j)$ by dividing by the number of documents from c_j that we've seen. Again, after processing all documents this incremental approach to the calculation will have the same result the batch method.

From a usability standpoint, incremental training is extremely useful. It allows the classifier to be updated easily over time and provides some assurance that the memory requirements for training are reasonable (since training data shouldn't need to be stored).

5 Perceptrons

A perception is a single node neural network with an arbitrary number of inputs and a single output. For the document classification task, we could create a neural network with one input for each feature in the feature vector. In essence, this network would take the same "inputs" as the naive bayes models described above.

A perceptron is characterized by: (1) the set of features it uses for input; (2) the relative weight associated with each input; and (3) a thresholding function used to compute the output activation given an input activation.

Assuming a feature vector with n features, and a *binary classification task* the training process works as follows: (1) initialize a set of random weights $\{w_0, w_1, \ldots, w_n, w_b\}$, one for each feature and a bias w_b; (2) for each document d_i consisting of features $\{f_0, f_1, \ldots, f_n\}$ in the training set; (2a) compute $in = \sum w_i f_i$ by summing over all inputs including the bias which has a fixed input value of 1; (2b) compute the output value $g(in)$ by applying some function g to the input; often this is a hard threshold that returns 1 if $g(in) > 0$ and 0 otherwise; (2c) determine the error in the output value by calculating the difference between the desired output, specified by the training data, and the actual output $g(in)$; (2d) finally, we modify the each weight w_j based on the error calculated in the previous step and the value of the corresponding f_j according to the following equation:

$$w_j = w_j + \alpha \cdot err \cdot f_j \tag{5}$$

α is referred to as the learning rate and is a parameter that governs how quickly the weights will change when an error in the classifier's output is detected. Generally, it makes sense to use values of α that decay slowly over time. Note that the update in step (2d) is applied to each weight in the perceptron including the *bias*.

The approach described above creates a perceptron that will produce a *binary* output. To classify objects that can be drawn from more than two classes, you can simply use a set of $|C|$ perceptrons where $|C|$ is the number of known classes. In this arrangement, the i^{th} classifier can be viewed as casting a vote as to whether a given documents belongs to the i^{th} class. If the vote is for the affirmative, the classifier will output 1, if it is the negative, the classifier will output 0. Thus, going back to our original example with documents from the classes $\{DT, DR, L\}$, we would begin by (1) initializing three separate sets of weights $\{w_{DT_0}, w_{DT_1}, \ldots, w_{DT_n}, w_{DT_b}\}$, $\{w_{DR_0}, w_{DR_1}, \ldots, w_{DR_n}, w_{DR_b}\}$, $\{w_{L_0}, w_{L_1}, \ldots, w_{L_n}, w_{L_b}\}$. Next, we would perform steps (2a) and (2b) for each of the three perceptrons. We would then calculate the error by comparing how each perceptron voted against the document's actual class. So, if the L classifier voted 0 for a document that was in fact a lien, the error would be 1. Likewise, if the DT classifier voted 1 for the same document, that classifier's error would be -1. The key thing to note here, is that the error term is computed on a "per classifier" basis. Finally, the updating step (2d) proceeds as normal once the per classifier error has been computed.

6 Discriminative Multinomial Naive Bayes

The Discriminative Multinomial Naive Bayes (DMNB) classifier is a naive bayes model that shares some similarities with a neural network. Before delving into the details of DMNB, let's consider the relationship that already exists between the multinomial naive bayes (MNB) model and the perceptron.

6.1 Perceptrons and Multinomial NB

Recall that the perceptron works by finding the weighted sum of its inputs using the formula below where we've separated out the bias term w_b from the weighted inputs $f_i w_i$:

$$in = w_b + \sum_{i=0}^{n} w_i f_i \tag{6}$$

For a multinomial naive bayes classifier distinguishing between two classes, we can determine the most likely class by looking at whether or not the ratio of $P(c_0|d_i)/P(c_1|d_i)$ is less than or greater than 1. That is we can look at the following value:

$$\frac{P(c_0|d_i)}{P(c_1|d_i)} = \frac{P(c_0)}{P(c_1)} \prod_{k=1}^{|\mathcal{D}|} \frac{P(w_k|c_0)}{P(w_k|c_1)}^{n_{ik}} \tag{7}$$

Note that if we take the log of both sides we obtain:

$$\log \frac{P(c_0|d_i)}{P(c_1|d_i)} = \log \frac{P(c_0)}{P(c_1)} + \sum_{k=1}^{|\mathcal{D}|} n_{ik} \log \frac{P(w_k|c_0)}{P(w_k|c_1)} \tag{8}$$

This looks very similar to the perceptron, and we might imagine creating a "perceptron" to represent this function by: (1) using a term-frequency vector to represent the input values f_i; (2) setting each weight $w_i = \log(P(w_k|c_0)/P(w_k|c_1))$; (3) setting the weight on the bias to $\log(P(c_0)/P(c_1))$. Indeed we could "train" these values using the following procedure: initialize counts for each class $n[c_0] = 0$ and $n[c_1] = 0$; initialize total word counts on a per-class $nw[c_0] = 0$ and $nw[c_1] = 0$ and per-class-per-word basis $nwc[w_k][c_0] = 0$, $nwc[w_k][c_1] = 0$; then for each document in the training set, update the counts so that, at the end of training you can calculate:

$$
\begin{aligned}
P(c_0) &= \frac{n[c_0]}{n[c_0] + n[c_1]} \\
P(c_1) &= \frac{n[c_1]}{n[c_0] + n[c_1]} \\
P(w_k|c_i) &= \frac{nwc[w_k][c_i]}{nw[c_i]}
\end{aligned}
$$

6.2 Training DMNB

The DMNB classifier leverages this relationship between the perceptron, where the weights are incrementally modified to improve classification, and the MNB classifier, where the "weights" are calculated by examining attributes in the training data as described above.

Here is how it works. During training, we iterate over each item in the training set performing: (1) a prediction about the document's classification; and (2) an adjustment to the classifiers parameters when an error occurs. This basic loop mimics what happens with a perceptron. Unlike a perceptron, however, the adjustment will happen in such a way that the "weights" for each input correlate much more closely to the values $\log(P(c_k|c_0)/P(w_k|c_1))$.

Just as with multinomial naive bayes, we'll keep track of the same counts as we process the training documents, however, since we want to use them for classification immediately, we need to initialize them to some reasonable guesses as opposed to 0. For this we use the Laplace estimators: initialize counts for each class $n[c_0] = 1.0$ and $n[c_1] = 1.0$; initialize total word counts on a per-class $nw[c_0] = |\mathcal{D}|$ and $nw[c_1] = |\mathcal{D}|$ and per-class-per-word basis $nwc[w_k][c_0] = 1.0$, $nwc[w_k][c_1] = 1.0$. Now, we process each training document and: (1) produce a feature vector for the document d_i just as with multinomial naive bayes; (2) compute the relative likelihood of class c_0 and c_1 using equation 8; (3) given a correct classification c^* for the training document d_i, calculate $P(c^*|d_i)$ using equation 8 and the knowledge that $P(c_0|d_i) + P(c_1|d_i) = 1$; (4) compute the classifier's error as $err = 1 - P(c^*|d_i)$. Finally, update the "counts" as per the multinomial naive bayes. However, instead of adding 1 to $n[c^*]$, $|d_i|$ to $nw[c^*]$ and n_{ik} to each per-class-per-word count $nwc[w_k][c^*]$, add $1 \cdot err$, $|d_i| \cdot err$, and $n_{ik} \cdot err$ respectively. That is, weight based on the classifier's error. This approach has the effect of modifying the conditional probability tables *as needed* to model the data and as such it helps to eliminate some of the issues with naive bayes (broken, at least for this domain) assumption that the features will be independent.

Note that the DMNB classifier is a boolean classifier, just like a perceptron. If you want to distinguish between three or more classes, you need one DMNB classifier *per* class, just as with the perceptron.

6.3 A Worked Example

We'll use the feature vectors from Section 4.1 to work through the training. We'll initialize our counts as below:

$$n[DT] = n[L] = 1.0, nw[DT] = nw[L] = 14, nwc[w_k][DT] = nwc[w_k][L] = 1.0$$

Now, we need to calculate $P(DT|d_0)$ for the first training document since equation 8 doesn't tell us how to normalize, we need to do a bit of work here. We begin with $w : n_w = \{$`this:1, deed:2, trust:1, for:1, house:1`$\}$

$$\begin{aligned}
P(DT|d_0) &= \alpha P(DT) \prod_w P(w|DT)^{n_w} \\
&= \left(\frac{1}{2}\right)\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^2\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1 \\
P(L|d_0) &= \alpha P(L) \prod_w P(w|L)^{n_w} \\
&= \left(\frac{1}{2}\right)\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^2\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1
\end{aligned}$$

$$P(DT|d_0) = 1 - P(L|d_0)$$

Note that initially $P(DT|d_0) = P(L|d_0)$, so it's a draw: $P(DT|d_0) = .5$. We then set $err = 1 - .5$ and update:

$$\begin{aligned}
n[DT] &\mathrel{+}= .5 \\
nw[DT] &\mathrel{+}= 6 \cdot .5 \\
nwc[this][DT] &\mathrel{+}= .5 \cdot 1 \\
nwc[deed][DT] &\mathrel{+}= .5 \cdot 2 \\
nwc[trust][DT] &\mathrel{+}= .5 \cdot 1 \\
nwc[for][DT] &\mathrel{+}= .5 \cdot 1 \\
nwc[house][DT] &\mathrel{+}= .5 \cdot 1
\end{aligned}$$

Now, we make a prediction about document d_1 containing the words: $\{$`deed:1, trust:1, for:1, the:1, wallace:1, house:1`$\}$:

$$P(DT|d_0) = \alpha \left(\frac{1.5}{2.5}\right)\left(\frac{2}{17}\right)^1\left(\frac{1.5}{17}\right)^1\left(\frac{1.5}{17}\right)^1\left(\frac{1}{17}\right)^1\left(\frac{1}{17}\right)^1\left(\frac{1.5}{17}\right)^1$$
$$P(L|d_0) = \alpha \left(\frac{1}{2.5}\right)\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1\left(\frac{1}{14}\right)^1$$
$$P(DT|d_0) = 1 - P(L|d_0)$$

The prediction is DT with $err = .24$. Be careful here, you'll probably need to use logs already to avoid underflow. Now update:

$$\begin{aligned} n[DT] &+= .24 \\ nw[DT] &+= 6 \cdot .24 \\ nwc[deed][DT] &+= .24 \cdot 1 \\ nwc[trust][DT] &+= .24 \cdot 1 \\ nwc[for][DT] &+= .24 \cdot 1 \\ nwc[the][DT] &+= .24 \cdot 1 \\ nwc[wallace][DT] &+= .24 \cdot 1 \\ nwc[house][DT] &+= .24 \cdot 1 \end{aligned}$$

We continue in this fashion, until one pass through the training set is complete. Often, as with a perceptron, it makes sense to run through training data more than one time to help the values converge. For DMNB, unlike a perceptron, often a few iterations (on the order of 5) is enough.