Chapter 2

More Text Classification

2.1 Introduction

If you were a naïve Bayes classifier, training would be like reading a bunch of texts belonging to different classes, and testing would be like reading a new text and asking yourself how plausible this text would be as an example of each of the classes. Now we're going to look at *discriminative* classifiers; if you were a discriminative classifier, training would be like reading a bunch of texts, guessing the classes of all of them, and then being told if you were right or wrong.

Above, we said that it would be harder to define a model for $P(k \mid d)$ – called a *discriminative* model as opposed to generative. It's harder, but by no means impossible. The advantage is that discriminative models are more closely matched to the task. Ultimately, we want to predict the best class given a document ($P(k \mid d)$). Generative models learn more than this; they learn $P(k, d)$, which means that they also try to learn which documents are more likely than others. In particular, they may get confused because they think that the features (words) of a document are independent, when in truth they are very interdependent. But discriminative models, because they don't care which documents are more likely than others, might do better. Ng and Jordan (2002) carry out a thorough theoretical and experimental comparison of a naïve Bayes classifier with its discriminative counterpart.

2.2 Perceptron

2.2.1 Model

The simplest (and oldest) such classifier is called the *perceptron*, which is an example of a *linear classifier*:

$$s(k \mid d) = \lambda(k) + \sum_{w \in d} \lambda(k, w) \tag{2.1}$$

Neither s nor the λ's are a probability distribution. The $\lambda(k, w)$ and $\lambda(k)$ are just weights – real-valued, positive or negative – which are added up into the total score $s(k \mid d)$. A higher score is better, so the predicted class is

$$k^* = \arg\max_k s(k \mid d). \tag{2.2}$$

The similarity with naïve Bayes (1.5) can be seen more clearly if we write naïve Bayes like this:

$$\log P(k,d) = \log p(k) + \sum_{w\in d} \log p(w \mid k). \tag{2.3}$$

The parameter $\lambda(k)$ corresponds to $\log p(k)$, and $\lambda(k,w)$ corresponds with $\log p(w \mid k)$. The difference is that there's no requirement for any of the λ's (or $\exp \lambda' s$) to sum to one.

Recall that we can generalize a bag of words to a bag of arbitrary features (like bigrams). Recall that if the features overlap (like word features and character features do), the model is still well defined. In practice, these models routinely have many overlapping features, and this is considered to be one of the main virtues of this approach.

For the rest of this chapter, assume that each document has a fake word (let's call it `<s>`) with count one, so that we can rewrite the model as

$$s(k \mid d) = \sum_{w\in d} \lambda(k,w), \tag{2.4}$$

where the sum includes $w =$ `<s>`. This makes the notation below much simpler.

2.2.2 Training

The training algorithm looks like this:

```
initialize parameters $\lambda(k,w)$ to zero
**repeat**
    **for** each $i$ **do**
        $k^* = \arg\max_k P(k \mid d_i)$
        **for** each $w$ **do**
            $\lambda(k_i, w) \leftarrow \lambda(k_i, w) + \eta \cdot c(w \in d_i)$
            $\lambda(k^*, w) \leftarrow \lambda(k^*, w) - \eta \cdot c(w \in d_i)$
        **end for**
    **end for**
**until** done
```

The constant η is called the *learning rate*; for now, it's fine to just set it to one. The algorithm is very intuitive: for each document, try to classify it. If it's correct ($k^* = k_i$), nothing happens. If it's incorrect, then make a change to the model that will encourage it to get this example right in the future.

2.2.3 Averaging

The perceptron works better if you use the *average* value of each parameter over all the iterations (Freund and Schapire, 1999). The intuition is that the perceptron might be fit too closely to the most recently seen examples; averaging helps to smooth the parameters out and generalize better to new examples.

But this introduces an efficiency problem. Normally, at each iteration, we only have to update a few of the parameters, but if we want to maintain a running sum of the parameters, then at each iteration, we have to update every running sum. There is a trick (Daumé, 2006, p. 19) that gets around this problem. Let $\lambda^{(t)}$ be the vector of all parameters at time t. It is the sum of a sequence of updates,

$$\lambda^{(t)} = \sum_{t=1}^{T} g^{(t)}. \tag{2.5}$$

In addition, let's maintain an auxiliary value,

$$\mu^{(t)} = \sum_{t=1}^{T} (t-1) \cdot g^{(t)}. \tag{2.6}$$

Unlike the running sum, μ has sparse updates, just like λ does. Then the average value is

$$\begin{aligned} \bar{\lambda} &= \frac{1}{T} \sum_{t=1}^{T} \lambda^{(t)} && (2.7) \\ &= \lambda^{(T)} - \frac{1}{T} \mu^{(T)}. && (2.8) \end{aligned}$$

Question 5. Why does this work?

2.3 Logistic regression

Above, we highlighted the similarity between the naïve Bayes classifier and a linear classifier. Now we look at another model which has an even closer similarity. The discriminative version of the naïve Bayes classifier is called *logistic regression*:

$$P(k \mid d) \propto \exp\left(\lambda(k) + \sum_{w \in d} \lambda(k, w)\right) \tag{2.9}$$

which means

$$\begin{aligned} P(k \mid d) &= \frac{1}{Z(d)} \exp\left(\lambda(k) + \sum_{w \in d} \lambda(k, w)\right) && (2.10) \\ Z(d) &= \sum_{k} \exp\left(\lambda(k) + \sum_{w \in d} \lambda(k, w)\right). && (2.11) \end{aligned}$$

The $\lambda(k)$ and $\lambda(k, w)$ are, as before, the parameters of the model. As before, the similarity with naïve Bayes (1.5) can be seen more clearly if we write naïve Bayes like this:

$$P(k, d) = \exp\left(\log p(k) + \sum_{w \in d} \log p(w \mid k)\right). \tag{2.12}$$

The differences are:

- The whole distribution is conditional ($P(k \mid d)$) instead of joint ($P(k, d)$).
- The parameters ($\exp \lambda$) don't have to sum to one.
- There is a normalization factor (Z) that makes the whole distribution sum to one.

Sometimes, this type of model is also called *log-linear* (because if you take the log of it, it's linear) or *maximum-entropy* (because of another way of deriving the model which we don't cover here).

2.3.1 Classification

Finding the most probable class of a new document is just as easy as before, because we can ignore the normalization factor:

$$\begin{aligned} k^* &= \arg\max_k P(k \mid d) && (2.13)\\ &= \arg\max_k \frac{1}{Z(d)} \exp\left(\lambda(k) + \sum_{w\in d} \lambda(k,w)\right) && (2.14)\\ &= \arg\max_k \exp\left(\lambda(k) + \sum_{w\in d} \lambda(k,w)\right), && (2.15)\end{aligned}$$

or we can even drop the exp, giving the same decision rule as the perceptron:

$$= \arg\max_k \left(\lambda(k) + \sum_{w\in d} \lambda(k,w)\right). \qquad (2.16)$$

2.3.2 Training

However, training, or estimating the parameters $\lambda(k,w)$, is more difficult than before. The likelihood, which we wish to maximize, is

$$\begin{aligned} L &= \prod_i P(k_i, d_i) && (2.17)\\ &= \prod_i \frac{1}{Z(d_i)} \exp \sum_{w\in d_i} \lambda(k_i, w) && (2.18)\end{aligned}$$

but there isn't a closed-form solution; instead, we have to use numerical optimization.

There are lots of methods that we can use to do this. The one most frequently used these days is L-BFGS, which is related to Newton's method but doesn't require any second derivatives. We're going to do the easiest (but still very practical) method, *stochastic gradient ascent*. (It is more commonly known as *stochastic gradient descent (SGD)*, but here we're maximizing, not minimizing.)

Let λ be the vector of all the parameters of the model, and let $f(\lambda)$ be the function we're maximizing. Gradient ascent looks like this:

```
initialize parameters $\lambda$ to zero
**repeat**
    $\lambda \leftarrow \lambda + \eta\nabla f(\lambda)$
**until** done
```

This method looks for the direction, starting from λ, that goes uphill the steepest, and moves λ in that direction, by an amount controlled by $\eta > 0$, called the *learning rate*.

Question 6. What happens if η is too small? too big?

To guarantee convergence, η should decrease over time (for example, $\eta = 1/t$), but it's also common in practice to leave it fixed.

In stochastic gradient ascent, we break up f into parts (typically, one for each document or minibatch of documents). Thus

$$f(\lambda) = \sum_i f_i(\lambda). \qquad (2.19)$$

Then the optimization looks like this:

```
initialize parameters $\lambda$ to zero
repeat
    for each $i$ do
        $\lambda \leftarrow \lambda + \eta \nabla f_i(\lambda)$
    end for
until done
```

This usually requires less memory and is easier to implement.

Now, let's let $f = \log L$ (which is simpler than L itself). Find the partial derivative with respect to each of the parameters.

$$\begin{aligned}
\frac{\partial}{\partial \lambda(k,w)} \log L &= \sum_i \frac{\partial}{\partial \lambda(k,w)} \left(\sum_{w \in d_i} \lambda(k_i, w) - \log Z(d_i) \right) && (2.20)\\
&= \sum_i \left(c(w \in d_i)\delta(k,k_i) - \frac{\partial}{\partial \lambda(k,w)} \log Z(d_i) \right) && (2.21)\\
&= \sum_i \left(c(w \in d_i)\delta(k,k_i) - \frac{1}{Z(d_i)} \frac{\partial}{\partial \lambda(k,w)} Z(d_i) \right) && (2.22)\\
&= \sum_i \left(c(w \in d_i)\delta(k,k_i) - \frac{1}{Z(d_i)} \frac{\partial}{\partial \lambda(k,w)} \sum_{k'} \exp \sum_{w' \in d_i} \lambda(k', w') \right) && (2.23)\\
&= \sum_i c(w \in d_i) \left(\delta(k,k_i) - \frac{\exp \sum_{w' \in d_i} \lambda(k,w')}{Z(d_i)} \right) && (2.24)\\
&= \sum_i c(w \in d_i) \left(\delta(k,k_i) - P(k \mid d_i) \right) && (2.25)\\
&= c(k,w) - E[c(k,w)]. && (2.26)
\end{aligned}$$

Some more calculus would demonstrate that the likeilhood is a concave function, so that the only local maximum is the global maximum, which is very good news for optimizing this function.

So the optimization algorithm looks like this:

```
initialize parameters $\lambda(k,w)$ to zero
repeat
    for each $i$ do
        for each $w$ do
            $\lambda(k_i, w) \leftarrow \lambda(k_i, w) + \eta \cdot c(w \in d_i)$
            for each $k$ do
                $\lambda(k, w) \leftarrow \lambda(k, w) - \eta \cdot P(k \mid d_i) \cdot c(w \in d_i)$
            end for
        end for
    end for
until done
```

Note the similarity between this algorithm and the perceptron: the only difference is that the perceptron used the feature values for the best class, k^*, but this algorithm uses the *expected* feature values (in other words, the mode versus the mean). We could think of the perceptron as an approximate version of this algorithm.

2.3.3 Regularization (smoothing)

Smoothing was important for the naïve Bayes classifier, and the most visible reason why was unknown words. Unknown words aren't as much of a problem for logistic regression, but smoothing is still important.

Question 7. How do naïve Bayes and logistic regression react to unknown words differently? Why is it not as much of a problem for logistic regression?

The general problem that both methods face is overfitting: learning the training data so well that it fails to generalize to new data. Suppose that a word w is seen only once in the training data; it's not a particularly spammy or hammy word, but it happens to occur in a spam document. So logistic regression will learn a very high weight for $\lambda(\text{spam}, w)$, but it really shouldn't.

The solution is to add a *regularization* term to the objective function that encourages weights to be small.

$$\begin{aligned} L &= \prod_i P(k_i, d_i) && (2.27) \\ &= \underbrace{\frac{C}{2}\sum_{k,w}\lambda(k,w)^2}_{\text{regularization}} + \prod_i \frac{1}{Z(d_i)} \exp \sum_{w\in d_i} \lambda(k_i, w) && (2.28) \end{aligned}$$

If we think of the weights λ as a vector, the regularization term is proportional to the square of the norm of the vector. More precisely, the Euclidean norm, also known as the ℓ_2 norm. So this kind of regularization is called ℓ_2 *regularization*.

This is very easy to implement: just inside the **for** i loop, before anything else, add

```
for each $k, w$ do
    $\lambda(k,w) \leftarrow \lambda(k,w) \cdot (1 - \eta \cdot C)$
end for
```

2.3.4 Experiment

We trained a logistic regression classifier on the same gender-classification task as before. We used 30 iterations, a learning rate of $\eta = 0.01$, and ℓ_2 regularization with $C = 0.1$. We tried just the standard bag of words (unigram) and we also added bigram features.

method	unigram	bigram
naïve Bayes	65.1	66.5
logistic regression	66.7	69.3

So logistic regression wins, and we can see that it is more able to take advantage of the bigram features.

2.4 Optional: Max-margin methods

There is another family of methods called *max-margin* or *large-margin* methods, the most well-known of which are *support vector machines*. These methods learn a linear model, like the perceptron; for our example of document classification, the model is:

$$s(k \mid d) = \sum_{w\in d} \lambda(k, w). \tag{2.29}$$

The basic intuition is that we want to learn a model that not only rates the correct labels the highest for all the training examples, but also rates the incorrect labels lower by some safe *margin*. This way, we can be more confident that the model will generalize better to new examples.

To do this, we *minimize* an objective function like:

$$L = \frac{C}{2}\|\lambda\|^2 + \sum_i \max_k(\ell(k,k_i) - \underbrace{(s(k_i \mid d_i) - s(k \mid d_i))}_{\text{margin}}) \tag{2.30}$$

where $\ell(k,k_i)$ is a *loss* function that measures how bad it would be to guess label k instead of the correct label k_i. For now, assume that it is 0 if k is correct and 1 is k is wrong. The *margin* measures how much the model prefers the correct label k_i over some other label k. So the second term of our objective function says: for each example i, we want the margin for every wrong label to be bigger than 1.

What's the purpose of the first term ($\frac{C}{2}\|\lambda\|^2$), called the *regularizer*? Notice that if we learn a model that can classify all the training examples correctly, then we can also make all the margins as big as we want simply by multiplying all the weights λ by some constant. The regularizer prevents us from doing this, by penalizing large weights.

There are many ways of optimizing this objective function, but we will continue to use stochastic gradient descent. Let

$$k^- = \arg\max_k (\ell(k,k_i) - (s(k_i \mid d_i) - s(k \mid d_i))) \tag{2.31}$$

$$= \arg\max_k (\ell(k,k_i) + s(k \mid d_i)). \tag{2.32}$$

Then

$$\frac{\partial L}{\partial \lambda(k)} = C\lambda(k) - \sum_i (\delta(k,k_i) - \delta(k,k^-)) \tag{2.33}$$

$$\frac{\partial L}{\partial \lambda(k,w)} = C\lambda(k,w) - \sum_i (\delta(k,k_i) - \delta(k,k^-)) \cdot c(w \in d_i). \tag{2.34}$$

This looks kind of complicated, but leads to an algorithm extremely similar to the perceptron (Shalev-Shwartz et al., 2011):

```
initialize parameters $\lambda(k,w)$ to zero
repeat
    for each $i$ do
        $k^- = \arg\max_k (\ell(k,k_i) + s(k \mid d_i))$
        for each $w$ do
            $\lambda(k_i,w) \leftarrow \lambda(k_i,w) + \eta \cdot c(w \in d_i)$
            $\lambda(k^-,w) \leftarrow \lambda(k^-,w) - \eta \cdot c(w \in d_i)$
        end for
    end for
until done
```

References

Daumé III, Hal (2006). "Practical Structured Learning Techniques for Natural Language Processing". PhD thesis. University of Southern California.

Freund, Yoav and Robert E. Schapire (1999). "Large Margin Classification Using the Perceptron Algorithm". In: *Machine Learning* 37.3, pp. 277–296.

Ng, Andrew Y. and Michael I. Jordan (2002). "On Discriminative vs. Generative classifiers: A comparison of logistic regression and naive Bayes". In: *Advances in Neural Information Processing Systems*. Vol. 14, pp. 841–848.

Shalev-Shwartz, Shai et al. (2011). "Pegasos: Primal Estimated sub-GrAdient SOlver for SVM". In: *Mathematical Programming, Series B* 127.1, pp. 3–30.