

Discriminative classifiers

LING83800

1 Introduction

Today we will consider two types of *discriminative classifiers*:

- the *logistic regression* classifier, the discriminative analogue to naïve Bayes, and
- the *perceptron* classifier, a non-probabilistic discriminative classifier.

Bibliographic note

The notation for logistic regression has been adapted from Ng and Jordan 2002. The notation for perceptron classifiers is novel.

Software note

The scikit-learn (Pedregosa et al. 2011)¹ module `sklearn.linear_model` provides several types of discriminative classifiers. The NLTK (Bird et al. 2009)² module `nltk.classify.maxent` has an implementation of multinomial logistic regression. Perceptron classifiers can be implemented in just a few lines of Python, so many libraries contain a “one-off” implementation optimized for the problem at hand.

2 Logistic regression

Recall that *generative classifiers* directly estimate a joint distribution over features $\mathcal{D}$ and classes $\mathcal{Y}$. Classifiers which do not attempt to estimate this joint distribution are known as *discriminative classifiers*. Ng and Jordan (2002) observe that many forms of generative classifier correspond to a related discriminative classifier which instead estimates a conditional distribution over classes $\mathcal{Y}$ given features $\mathcal{D}$.³ One such generative-discriminative pair is the naïve Bayes classifier we saw last week and *logistic regression*.⁴ Logistic regression was originally developed for statistical inference, but much like naïve Bayes, it has proved **unreasonably effective**.

¹<http://scikit-learn.org/>

²<http://www.nltk.org/>

³Though as we will soon see, some types of discriminative classifiers are non-probabilistic.

⁴For historical reasons, logistic regression classifiers are sometimes referred to as *maximum entropy* (or *maxent*) models in computational linguistics.

2.1 Parameters

Recall that naïve Bayes classification assumes that the data is drawn i.i.d. from a joint distribution, then uses Bayes' rule and the chain rule to estimate the posterior probability of a given label $y' \in \mathcal{Y}$. In the multinomial case, and assuming boolean ("Bernoulli") features, this has the form

$$\hat{P}(y = y' \mid \mathcal{F}) = \hat{P}(y = y') \prod_{i=1}^n \hat{P}(\mathcal{F}_i = 1 \mid y = y') \quad (1)$$

where $\mathcal{F} \in \{0, 1\}^n$. In logistic regression, the parameters of the model are not themselves probabilities, but together they can be used to compute the posterior distribution $P(y = y' \mid \mathcal{F})$. We again make the "naïve" assumption of feature conditional independence

$$P(\mathcal{F}_i = 1 \mid \mathcal{F}_{j \neq i} = 1, \dots, y = y') = P(\mathcal{F}_i = 1 \mid y = y'). \quad (2)$$

under which we can represent the conditional probability distribution as the result of passing a real-valued "scoring function"—a sum of products—through a "squashing function".

Without loss of generality, let us first consider the binary classification problem. In this setting, the parameters of a logistic regression classifier are represented by a *bias* term $b \in \mathbb{R}$ and a vector $\beta \in \mathbb{R}^n$ where n is the number of features. Let us use θ to denote the set of parameters $\{b, \beta\}$. Then the scoring function $\sigma \subseteq \mathcal{F} \times \theta \times \mathbb{R}$ is given by

$$\sigma(\mathcal{F}; \theta) = b + \sum_{i=1}^n \beta_i \mathcal{F}_i. \quad (3)$$

Since the scoring function has a real-valued range, we need a way to "squash" the scores into probabilities. This is accomplished using the *logistic*, a function over $\mathbb{R} \times (0, 1)$ defined as

$$\text{logis}(x) = \frac{1}{1 + e^{-x}}. \quad (4)$$

The logistic function maps a real value x to some value between 0 and 1, and thus is suitable for converting real-valued σ scores to probabilities. Furthermore, as can be seen in figure 1, it is *monotonically increasing*: for any $x, x' \in \mathbb{R}$, if $x \leq x'$ then $\text{logis}(x) \leq \text{logis}(x')$.

Putting these two pieces together, we obtain the posterior probability by squashing the output of the scoring function into a probability like so:

$$\hat{P}(y = 1 \mid \mathcal{F}; \theta) = \text{logis}[\sigma(\mathcal{F}; \theta)]. \quad (5)$$

2.2 Decision rule

The decision rule for binary logistic regression is simple. As the logistic function is monotonically increasing (§2.1), a y' which maximizes the scoring function also maximizes the posterior probability (eq. 5). Furthermore, as can be seen in figure 1, a positive σ gives a posterior probability $\geq .5$, so by the law of total probability,

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(\mathcal{F}; \theta) \geq 0 \\ 0 & \text{otherwise} \end{cases}. \quad (6)$$

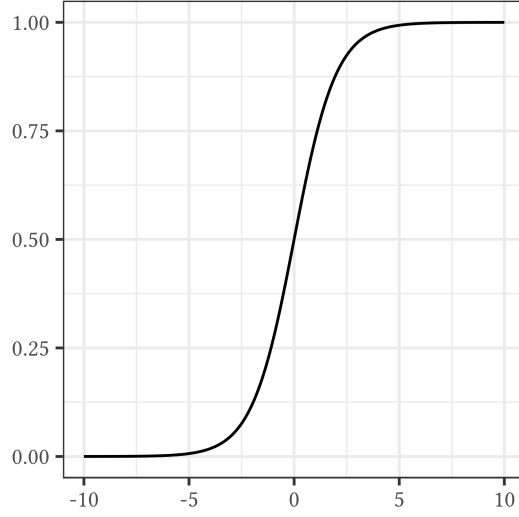


Figure 1: The logistic function.

2.3 Learning

There are quite a few ways to estimate the parameters θ ; we put this aside for sake of time. In scikit-learn, for example, `sklearn.linear_model.SGDClassifier` uses *stochastic gradient descent* for parameter estimation, and `sklearn.linear_model.LogisticRegression` supports four other parameter learning techniques.

2.4 Multinomial classification

It is straightforward to adapt this model for cases where $|\mathcal{Y}| > 2$. First, the parameterization is extended so that θ consists of a bias vector $B \in \mathbb{R}^{|\mathcal{Y}|}$ and weights $\beta \in \mathbb{R}^{n \times |\mathcal{Y}|}$. Then the score is given by

$$\sigma(\mathcal{F}, y'; \theta) = B_{y'} + \sum_{i=1}^n \beta_{i,y'} \mathcal{F}_i \quad (7)$$

The maximum a posteriori decision rule is then

$$\hat{y} = \arg \max_{y' \in \mathcal{Y}} \hat{P}(y = y' \mid \mathcal{F}; \theta) \quad (8)$$

$$= \arg \max_{y' \in \mathcal{Y}} \sigma(\mathcal{F}, y'; \theta) \quad (9)$$

$$= \arg \max_{y' \in \mathcal{Y}} \left[B_{y'} + \sum_{i=1}^n \beta_{i,y'} \mathcal{F}_i \right] \quad (10)$$

3 The perceptron

The perceptron is a non-probabilistic discriminative classifier first proposed by Rosenblatt (1958) and is one of the first modern machine learning algorithms. For various technological and socio-logical reasons, it was not widely used in speech and language research until a series of influential papers by Michael Collins and colleagues (e.g., Collins 2002a,b, Collins and Duffy 2002, Collins and Roark 2004). Like logistic regression, the perceptron is a discriminative model with a decision rule based on a linear equation, but unlike prior models, it does not make use of probability.

3.1 Parameters

Much like logistic regression, the parameters for a binary perceptron classifier consist of a bias term $b \in \mathbb{R}$ and a weight vector $\beta \in \mathbb{R}^n$. Let $\theta = (b, \beta)$. We use the same scoring function as for logistic regression (repeated from above):

$$\sigma(\mathcal{F}; \theta) = b + \sum_{i=1}^n \beta_i \mathcal{F}_i \quad (11)$$

3.2 Decision rule

Similarly, the binary perceptron uses the same decision rule as logistic regression (repeated from above): D

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(\mathcal{F}; \theta) \geq 0 \\ 0 & \text{otherwise.} \end{cases} \quad (12)$$

3.3 Learning

The elegance of the perceptron derives from the simple, online method used for parameter learning, the *perceptron update rule*. To motivate this, let us define the *margin* of an example $(\mathcal{F}, y)$ as $\gamma = y \cdot \sigma(\mathcal{F}; \theta)$. This quantity is positive when $\mathcal{F}$ is correctly classified and negative otherwise.

When the margin is positive for all examples, all data is correctly classified. When the margin is positive for some training example, no changes are made to the parameters. When it is negative, the parameters are modified to increase the margin by a single unit (usually just 1). Let $\tau = y - \hat{y}$ where $\hat{y}$ is the predicted label. Thus

- if $y = 1$ and $\hat{y} = 0$, then $\tau = +1$;
- if $y = 0$ and $\hat{y} = 1$, then $\tau = -1$;
- if $y = \hat{y}$, then $\tau = 0$.

Then, we update θ as follows:

$$b \leftarrow b + \tau \quad (13)$$

$$\beta_i \leftarrow \beta_i + \mathcal{F}_i \tau \text{ for } i \text{ to } n \quad (14)$$

When $y = \hat{y}$, $\tau = 0$ and the update has no effect (and can in fact be ignored altogether). But when $y \neq \hat{y}$, the update increases y for that example. Thus, with all else held equal, the margin will be closer to positive the next time that example is processed.

To train the perceptron, we make repeated passes through the training data (usually in a random order), each called an *epoch*, and for each *step* (i.e., a single example) we apply the perceptron update rule. Perceptron training then proceeds until:

- a pre-determined number of steps is reached,
- a pre-determined number of epochs is reached, or
- convergence is detected.

3.4 Weight averaging

Because the perceptron update rule can result in a change to the weights for every example presented, the *vanilla* perceptron—as the above model is sometimes called—is unstable: a single mislabeled example at the end of the training set can cause serious harm. Freund and Schapire (1999) propose a mechanism—*weight averaging*—to prevent this. The resulting model, the *averaged* (or *voted*) *perceptron*, is quite common in speech and language technology.

A naïve implementation of weight averaging is performed by maintaining two sets of parameters. Let w_c denote some parameter in θ at step c , and $\sum_{i=1}^c w_i$ the running sum of that parameter. At every step, we simply add w_c (for each parameter in θ) to the associated running sum. Then, at the end of training, we divide the running sum by the number of steps, using the resulting parameters for future inference. However, this is inefficient—particularly when dealing with sparse features—since most parameters will not be changed at every step. Daumé (2006:9f.) suggests the following solution. Rather than recomputing the running sum $\sum_{i=1}^c w_i$ at every step c , instead we only compute it when w is updated, and at the end of training. To accomplish this, we store three values for each parameter w :

- the current weight w_c ,
- a timestamp $t \leq c$ which denotes the step on which it was last “freshened”, and
- $\sum_{i=1}^t w_i$, the running sum of w up to step t .

To freshen a summed weight w at step c , we simply set

$$\sum_{i=1}^t w_i \leftarrow [\sum_{i=1}^t w_i] + (c - t)w_c \quad (15)$$

$$t \leftarrow c \quad (16)$$

In Python, this might look something like the following class.

```
class LazyWeight:

    def __init__(self):
```

```

    self.w_c = 0
    self.t = 0
    self.sum_w = 0

    def freshen(self, c: int) -> None:
        self.sum_w += (c - self.t) * self.w_c
        self.t = c

    def update(self, c: int, tau: int) -> None:
        self.freshen(c)
        self.w_c += tau

    def average(self, c: int) -> float:
        self.freshen(c)
        return self.sum_w / c

```

3.5 Multinomial extension

It is also straightforward to adapt this model for multinomial classification. The parameterization, scoring function, and decision functions for a multinomial perceptron classifier are the same as those used for multinomial logistic regression (§2.4). The one complexity is the definition of the update rule. Let $y, \hat{y} \in \mathcal{Y}$ denote the gold label and the predicted label at the current step, respectively. When $y = \hat{y}$, we make no updates; when $y \neq \hat{y}$, we “reward” all weights associated with y and “penalize” all weights associated with $\hat{y}$. Thus, for the bias vector B we set

$$B_y \leftarrow B_y + 1 \tag{17}$$

$$B_{\hat{y}} \leftarrow B_{\hat{y}} - 1 \tag{18}$$

and for β we set

$$\beta_{i,y} \leftarrow \beta_{i,y} + \mathcal{F}_i \text{ for } i \text{ to } n \tag{19}$$

$$\beta_{i,\hat{y}} \leftarrow \beta_{i,\hat{y}} - \mathcal{F}_i \text{ for } i \text{ to } n \tag{20}$$

If we define the margin in the multinomial case as the difference in scores between y and $\hat{y}$

$$\gamma = \sigma(\mathcal{F}, y; \theta) - \sigma(\mathcal{F}, \hat{y}; \theta) \tag{21}$$

then this update rule attempts to enforce a positive margin for all examples.

3.6 Extensions

The perceptron update rule attempts to enforce a positive margin across the training data. Several related linear classifiers, including *support vector machines* (Cortes and Vapnik 1995) and *passive-aggressive classifiers* (Crammer et al. 2006), attempt to enforce a larger margin, providing either stronger theoretical guarantees and/or faster convergence.

References

- Bird, Steven, Ewan Klein, and Edward Loper. 2009. *Natural Language Processing with Python*. O'Reilly Media.
- Collins, Michael. 2002a. Discriminative training methods for hidden Markov models: theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, 1–8.
- Collins, Michael. 2002b. Ranking algorithms for named entity extraction: boosting and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 489–496.
- Collins, Michael, and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, 263–270.
- Collins, Michael, and Brian Roark. 2004. Incremental parsing with the perceptron algorithm. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics*, 111–118.
- Cortes, Corinna, and Vladimir Vapnik. 1995. Support-vector networks. *Machine Learning* 20:273–297.
- Crammer, Koby, Ofer Dekel, Joseph Keshet, Shai Shalev-Shvartz, and Yoram Singer. 2006. Online passive-aggressive algorithms. *Journal of Machine Learning Research* 7:551–585.
- Daumé, Hal, III. 2006. Practical structured learning techniques for natural language processing. Doctoral dissertation, University of Southern California.
- Freund, Yoav, and Robert E. Schapire. 1999. Large margin classification using the perceptron algorithm. *Machine Learning* 37:277–296.
- Ng, Andrew Y., and Michael I. Jordan. 2002. On discriminative vs. generative classifiers: a comparison of logistic regression and naive Bayes. In *NeurIPS*, 841–848.
- Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, ..., and Édouard Duchesnay. 2011. Scikit-learn: machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.
- Rosenblatt, Frank. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review* 65:386–408.