Statistical NLP: course notes

Çağrı Çöltekin — SfS / University of Tübingen

2020-05-11

These notes are prepared for the class *Statistical Natural Language Processing* taught in Seminar für Sprachwissenschaft, University of Tübingen.

5 Classification

As we briefly discussed before, a supervised machine learning method is called *classification* if the outcome variable is a categorical variable. Figure 5.1 depicts a binary classification problem in a two-dimensional input space. Given the input variables, or features, the task is to predict the class label (represented as $+$ or $-$ in the figure). Problems that are suited for classification is more widespread in NLP in comparison to regression. The following are only a few problems that can be solved by classification methods.

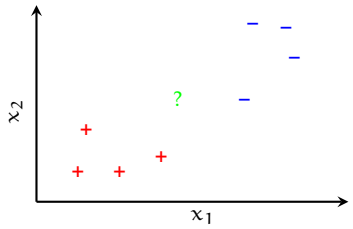

Figure 5.1: A demonstration of classification problem. Each data point is defined by two features (x_1 and x_2). The aim is to predict the binary label, $+$ or $-$, of an unknown data point based on a model learned from a labeled training set.

- Spam detection
- Author identification, author profiling (e.g., prediction gender of the author of a text)
- Sentiment analysis, e.g., given a product review is it positive or negative
- Topic classification, e.g., of news articles

Note that some these can be cast into regression problems. For example, sentiment analysis may be formulated as predicting a scale from negative to positive. Some other problems, such as POS tagging, require classifying a sequence of input items rather than predicting each target individually.[1] In this lecture, we will focus on simple classification problems, where the aim is to assign instances we want to classify to two or more class labels independently. Our discussion will mainly focus on binary classification, where there are only two outcomes. We will go through a few ways to generalize it to multi-class classification.

There are numerous classification methods with their strengths and weaknesses. Covering a large number of them, or an in-depth introduction of any of them is out of scope of this course, and the lecture. We introduce only three simple but important methods, namely, *perceptron*, *logistic regression* and *naive Bayes*, and focus on issues that arise during the use of classification methods in general. We leave the discussion of artificial neural networks to another lecture.

[1] A broad class of machine learning methods, as the ones we will study in this lecture, assume that the data to be classified is *independent and identically distributed* (i.i.d.). In contrast, some tasks have 'structured' outputs, such as a sequence. We will introduce common methods for predicting sequences in later lectures.

5.1 Perceptron

Perceptron has an important place in the history of machine learning. It is a basic binary classification method from 1950's. However,

it still finds its use in some modern/recent methods, it is (historically) related to modern artificial neural networks, and understanding the perceptron is also helpful for understanding some of the more recent and successful classification methods (e.g., support vector machines). Perceptron takes its inspiration from the biological neuron.[2] It receives a number of numeric inputs, multiplies each input with an associated weight, and outputs $+1$ (or 'fires') if the weighted sum is greater than 0, otherwise -1 it outputs. A schematic description of perceptron is presented in Figure 5.2.

[2] Like modern artificial neural networks, however, the relation to the biological neuron is rather weak. We take perceptron as a practical machine learning method, rather than a model of biological systems.

Formally, the output of the perceptron (y) is defined as

$$y = f\left(\sum_i^k w_i x_i\right)$$

where,

$$f(x) = \begin{cases} +1 & \text{if} \quad \sum_i^k w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}.$$

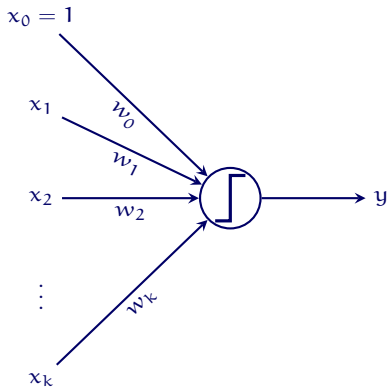

Figure 5.2: A schematic description of perceptron.

The weights $w_i \dots w_k$ are the parameters of the model that we want to estimate, or learn. Like in regression, an intercept term is often included, denoted as w_0 in Figure 5.2 and the equation above, in which case we also assume a constant input of $x_0 = 1$. Noticing that the weighted sum above is the dot product of the parameter and the input vectors helps us interpret the perceptron learning algorithm geometrically. The final function $f(x)$, which is called an *activation function* in some contexts, is simply a step function, mapping its input to two values.

So far, we defined how perceptron predicts the category. In simple words, if the sum of the weighted inputs is negative, the prediction is the negative class, if the sum is positive, the prediction is the positive class.[3] A perceptron model with a fixed set of weights is not interesting, nor is it practical to set the weights manually to solve any real-world problem. What makes perceptron interesting, like in any machine learning method, is that the parameters (the weight vector $\boldsymbol{w}$) are learned from the data, and there is a well-known algorithm for learning the weights.

[3] The way class labels assigned to -1 or $+1$ does not make any difference for the method. The assignment may sometimes be meaningful (e.g., in sentiment analysis it makes sense to assign positive values to the positive sentiments). In many problems, however, there are also no reasons to choose one of the classes as positive (e.g., in gender prediction).

The perceptron algorithm learns only from its mistakes, correct predictions do not contribute to learning. Similar to the gradient descent algorithm we discussed earlier, the perceptron algorithm is an iterative algorithm. The perceptron error is defined as

$$E(\boldsymbol{w}) = \sum_{i \in \text{misclassified}} -y_i \boldsymbol{w} \boldsymbol{x}_i \,. \tag{5.1}$$

The geometric interpretation of this formula is straightforward. Note that the last part in the sum is a dot product. For normalized (unit) vectors, it will tend to 1 if the direction of the weight vector and the input vector are similar, and it will tend to -1 if they point to opposite directions. Also noting that the class label y_i is either 1 or -1, the formula tells us that we are looking for a weight vector that is

'prototypical' for the positive examples. The error will be low if the input instance is similar to the weight vector, and the gold-standard label is positive, or input and the weight vectors are dissimilar and the gold-standard label is negative. Hence, the error promotes a solution where the weight vector is maximally similar to the positive examples, and minimally similar to the negative examples.

There is an algorithm that provably converges to an optimum solution with 0 classification error under certain conditions. Before discussing how we actually do this, it is useful to introduce a useful alternation. The error function as defined Equation 5.1 calculates the error for all misclassified examples. In practice, often an *online* version of the algorithm is used, where at each iteration a single misclassified example is picked, and the weight values are updated before picking another misclassified example. This procedure is often more efficient and easy to grasp for most people, and we will continue our discussion based on the online version of the algorithm.[4]

[4] Another (probably more) common strategy in many iterative training algorithms is to process the data in *batches*, a fixed number of training instances at a time. The online algorithm is equivalent to a batch algorithm with a batch size of one.

We start the search procedure with a random weight assignment, and update the weights based on the gradient vector, such that

$$w \leftarrow w - \eta \nabla E(w)$$

where η is the learning rate as in gradient descent. The error function (Equation 5.1) for a single example is $-y_i \boldsymbol{w} \boldsymbol{x}_i$, whose gradient with respect to weights is simply $-y_i \boldsymbol{x}_i$. As a result, at every step of the online algorithm, we subtract the value $\eta \boldsymbol{x}_i y_i$ from the weight vector for a training instance i that algorithm misclassifies. For the online algorithm, the update rule is

$$w \leftarrow w + \eta \boldsymbol{x}_i y_i$$

where i refers to a single (random) training instance misclassified by the model at this step. Intuitively, if the gold-standard label of the training instance i, y_i, is $+1$, we add the feature vector $\boldsymbol{x}_i$ to the weight vector (after scaling with the learning rate), making the weight vector more similar to the misclassified positive example. If the gold-standard label, y_i is -1, then the update rule subtracts the feature vector $\boldsymbol{x}_i$ from the weight vector, the weight vector them less similar to the misclassified negative example.

The intuitions above rely on the fact that the objects we want to classify are represented by the feature vectors $\boldsymbol{x}$ in a way that the representations in each class are more similar to each other in comparison to the representations of the object in the other class. Furthermore, since we consider dot product as a measure of similarity, the scales of variables matter. In general, perceptron and related methods are sensitive to the scales of the feature vectors. Scaling (or normalizing) weight vectors may be important for successful application of the method.

Figure 5.3 presents a step-by-step demonstration of the perceptron algorithm. The top left panel shows the training set. In panel 1, the weight vector is initialized randomly. The discriminant line is perpendicular to the wight vector, as the points where $\boldsymbol{w}\boldsymbol{x} = 0$ is

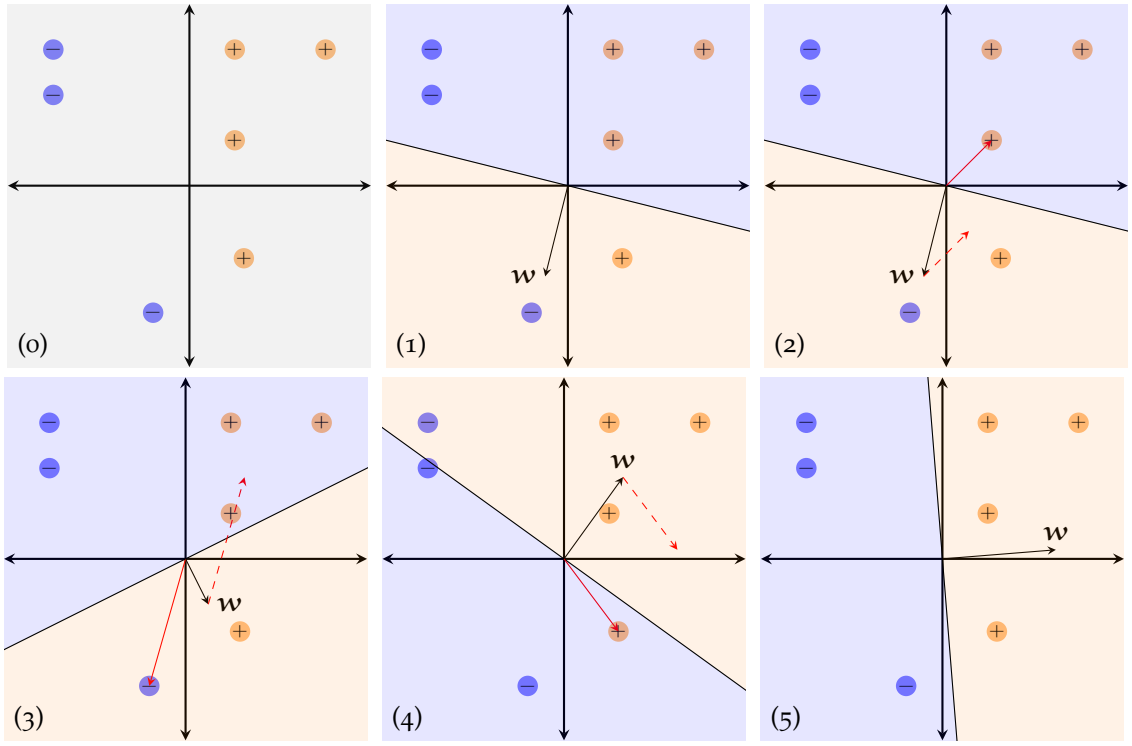

Figure 5.3: A step-by-step demonstration of perceptron algorithm.

satisfied when $\mathbf{x}$ and $\mathbf{w}$ are perpendicular. Hence, any input vector perpendicular to the weight vector lies on the decision boundary. The input vectors that lie on the same side as $\mathbf{w}$ are predicted as positive class (shaded in light orange in the figure), and the input vectors on the other side of the boundary (shaded in blue) are predicted as negative class. The initial configuration misclassifies three of the positive training instances on the upper right quadrant, and one of the negative instances (lower left quadrant). In the next step (panel 2), we pick one of the misclassified positive examples indicated by the red vector, and add it to the weight vector.[5] The resulting weight vector is shown in panel 3, along with the misclassified training instance picked for the next iteration. Since the misclassified instance picked is a negative one, we subtract it from the weight matrix, showing the results in panel 4, where we again pick another (positive) training instance that is misclassified. After adding it to the weight vector, the decision boundary shifts in a way that separates all the training instances correctly, which concludes the search process.

The perceptron algorithm is guaranteed to converge if the training instances are linearly separable. That is, the negative and positive instances can be separated by a hyperplane (a line in two-dimensional space in the example). If the training instances are not linearly separable, the algorithm does not terminate. In theory, this is a major drawback, as we also do not know in advance whether the training set is linearly separable or not. However, in practice, the algorithm can be used with a stopping condition, such as a maximum number of iterations allowed, and/or stopping after the classification accuracy stops improving.

[5] Since we simply add the vectors, the learning rate this demonstration is 1.

Another, somewhat subtle issue, with the perceptron algorithm is that when it finds a solution, it will stop even if there may be a better solution that leaves a larger margin between the positive and negative classes. This issue is tackled by *large margin classifiers*, such as *support vector machines* (SVMs).[6]

The perceptron is a binary classifier. To use it in multi-class classification problems, we need to use one of the strategies of combing multiple binary classifiers for obtaining multi-class classifiers. We will discuss the general multi-class strategies soon in this lecture.

[6] We will not cover SVMs in this course. However, now that you know the perceptron, you will understand SVMs with less effort if you study it in other sources.

5.1.1 *A bit of history*

The perceptron was developed in late 1950's and early 1960's by Rosenblatt (1958). It caused excitement in many (then young) areas including computer science, artificial intelligence, cognitive science. The excitement (and funding) died away in early 1970's after the criticism by Minsky and Papert (1969) where main issue was the fact that the perceptron algorithm cannot handle problems that are not linearly separable.

Another interesting note about the perceptron is its similarity with other classifiers. Although it is technically more similar to the SVMs, historically it is related to the modern artificial neural networks (ANNs).

5.2 *Logistic regression*

A *logistic regression* model estimates the conditional probability of the outcome variable given the predictor(s), $P(y \mid x)$. It is one of the basic methods in statistics as used in experimental sciences, and also closely related to the more complex models, including artificial neural networks.

Despite 'regression' in the name, logistic regression is is a classification method. The name is related to the fact that it is an instance of *generalized linear models* (GLMs), which are generalizations over linear regression. In GLMs the outcome variable is transformed using a non-linear function,[7] and error distribution may be non-Gaussian. Below, we will start from ordinary least-squares regression and develop our GLM to a logistic regression through an example.

[7] Which is called a *link function* in the GLM literature.

Since we want to predict the probability of the positive class, we code positive class as 1 and negative class as 0.[8] Hence our training labels are either 0 or 1. During prediction we want a probability value in range $[0, 1]$. Typically, if the probability is larger than 0.5, then we predict the positive class, otherwise the negative class. Note that we expect the model to predict the probability of success in a Bernoulli trial conditioned on the predictor(s). The model's outcome $\hat{y}$ is simply the probability parameter of a Bernoulli (or binomial) distribution.

[8] In contrast to perceptron where we use 1 and -1.

Figure 5.4 presents an example training set for logistic regression with a single predictor, along with an ordinary least-squares model fit to the data. The dashed vertical line indicate the discrimination

point. In this example we have only a single predictor. As a result the model will predict positive or negative classes for values of x below or above a particular value. Since the regression slope is negative in our example, the points on the left of the discriminant will be assigned to the positive class, and the points on the right are predicted as negative.

The first problem with fitting a regression line to the data in Figure 5.4 is related to the fact that we want to predict probabilities. However, the regression model's predictions will be above 1, for example, for $x = -2$ and below 0 for $x = 2$. To solve this problem, we transform our outcome variable using the *logit* function. The logit function is defined as

$$\text{logit}(p) = \log \frac{p}{1-p}$$

where p is the probability of the positive class, for our purposes.

The term inside the logarithm is called the 'odds ratio', which is simply the probability that an event occurs divided by the probability that it does not. The odds ratio ranges between 0 to ∞. Taking its logarithm, we map our outcome variable to the range $-\infty$ to ∞. Now we can set our regression model as

$$\log \frac{p}{1-p} = w_0 + w_1 x$$

and during prediction invert the logit function to get the conditional probability we are interested in. The inverse of the logit function is the *logistic* function (hence the name of the method). Figure 5.5 plots the logistic function. The logistic function is a sigmoid-shaped function that squashes the real numbers into the range between 0 and 1. For our (almost logistic) regression example, the probability value returned from the model is, then, defined as

$$\hat{y} = \hat{p} = \frac{1}{1 + e^{-w_0 - w_1 x}} = \frac{e^{w_0 + w_1 x}}{1 + e^{w_0 + w_1 x}} \tag{5.2}$$

which is the model's prediction of the probability of success.

We simply take the regression model's prediction, and plug it into the logistic function to obtain the estimated probability of the positive class given a value of $\mathbf{x}$.

A second problem with using regression to estimate the logistic regression problem is the distribution of the errors. The distribution of residuals in Figure 5.4 are not normal. This means our estimation is not the maximum-likelihood estimation. More importantly, however, some errors are not useful for estimation, resulting in a non-optimal model. For example, the model classifies the right-most blue point ($x = 2.0$) in Figure 5.4 correctly and confidently. However, being away from the regression line, this data point will have a rather large residual, causing model to be penalized for a correct classification.

The solution to this problem is to realize that the model defines a binary distribution over the training samples (error is also binomially

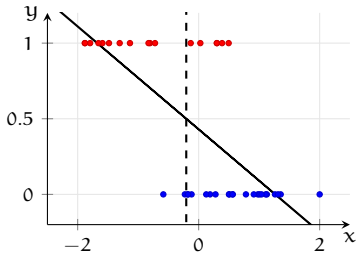

Figure 5.4: An example training set for logistic regression with a single predictor. Positive data points are indicated with red, and negative data points are indicated with blue dots. The thick line represents an ordinary least squares regression fit to the data. The dashed line represent the point in the x axis which is used for discriminating the positive and negative instances.

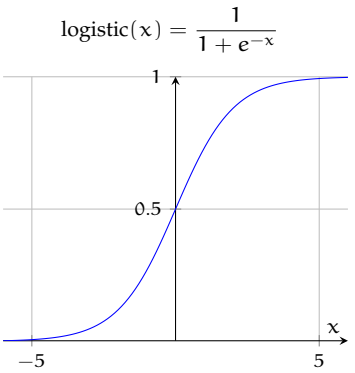

Figure 5.5: A plot of logistic function.

distributed). The model's prediction of the probability of a particular data point in the training data can be calculated using Equation 5.2. Once we have the predicted probabilities for the training set, we can write down the likelihood of the training set according to the model as

$$\mathcal{L}(\boldsymbol{w}) = \prod_i \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i}$$

where $\hat{y}_i$ is the model's prediction, and y_i is the gold-standard label for i^{th} training instance. Remember that $\hat{y}_i$ and y_i are both probability values. The equation simply uses the Bernoulli/binomial probability mass function we reviewed during our probability refresher. The likelihood of the whole data set is the multiplication of individual likelihoods, since we assume the training instances are independent of each other.

As in our earlier discussion of maximum likelihood estimation (of regression), instead of maximizing the likelihood, we minimize the 'minus log likelihood'.[9]

$$-\log \mathcal{L}(\boldsymbol{w}) = -\sum_i y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i) \tag{5.3}$$

Note the resemblance of this formula to the *cross entropy* we discussed during our discussion of information theory. In fact, by minimizing this function, we are minimizing the cross entropy of the gold-standard distribution defined by the training set with respect to the distribution of the model's predictions.

The error function above is differentiable. If we replace occurrences of $\hat{y}_i$ Equation 5.3 with its form defined in Equation 5.2, we can find/evaluate the gradient of this function with respect to the model parameters. However, there is no known analytic solution. The good news, on the other hand is that it is a convex function. As a result, we can use *gradient descent* to find the global minimum of the error.

Figure 5.6 shows the fitted logistic regression curve. Although the discriminant (indicated by the dashed vertical line) is similar to the one estimated by the ordinary regression (Figure 5.4), now the model's predictions are probability values in range $(0, 1)$, and the predicted probability is now a non-linear function of the predictor(s). To find the discriminant, we need to set the model expression to 0.5, the equi-probability value for both outcomes. Solving this equation is not very difficult. However, note that the predicted probability is $\frac{1}{2}$ when 'regression part of the model' is 0, i.e., $2.40 + 0.33x = 0$ in our example. This is a linear equation. Even though the probability assignments are made in a non-linear fashion, the discriminant function is linear. To show the linearity more clearly, Figure 5.7 plots a similar example with two predictors.

5.2.1 *Multi-class logistic regression*

We discussed logistic regression for the binary case, but it has a natural extension to multi-class case, which has been rather popular in

[9] To prevent overfitting, typically the objective function minimized includes a regularization term, e.g., L1 or L2 norm of the weight vectors.

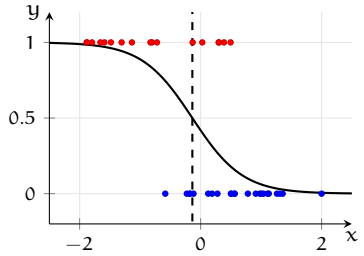

Figure 5.6: The logistic regression fitted to the data presented in Figure 5.4. The resulting logistic curve is the blue curve, which is $\hat{y} = \frac{1}{1+e^{0.33+2.40x}}$.

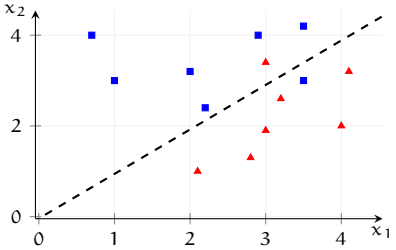

Figure 5.7: An example data with two predictors and the discriminator line estimated by logistic regression. The estimated logistic regression equation is

$$y = \frac{1}{1+e^{-0.1-2.53x_1+2.58x_2}}.$$

As a result, the equation for the discriminant line is defined by

$$-0.1 - 2.53x_1 + 2.58x_2 = 0.$$

the NLP literature. Recall that logistic regression estimates a conditional probability distribution, the probability distribution of the outcome variable conditioned on the predictors. In binary case the distribution estimated is a binomial distribution. For the multi-class case we need to estimate a multinomial distribution, which leads us to the formulation

$$P(C_k \mid x) = \frac{e^{w_k x}}{\sum_j e^{w_j x}}$$

where C_k is the k^{th} class label. This function is called the *softmax* function, and it is a generalization of the logistic sigmoid function that ensures that total probability (over all class labels) sum to 1. Similar to the binary case, we estimate the model parameters by minimizing the minus log likelihood assigned to the data by the model.[10] Similar to the binary case, the loss function is differentiable and convex, and the parameters can be estimated using a search procedure like gradient descent.

[10] What is the number of parameters in terms of number of classes and number of predictors?

The multinomial version of logistic regression is known as *maximum entropy model* (often shortened *max-ent*), or *log-linear* model. Furthermore, both binary and multi-class versions are important building blocks of (deep) neural networks.

5.3 *Naive Bayes classifier*

Another simple classification method that enjoyed quite some success and popularity, especially in spam detection, after it was introduced is the *naive Bayes classifier*. Similar to the logistic regression, our aim is to estimate the probability of the outcome variable given the predictor(s). Hence, the model assigns probabilities to each possible class given the set of predictors. During prediction, the class with the highest probability is selected as the predicted class. More formally, we can write this as

$$\hat{y} = \arg\max_y P(y \mid \mathbf{x})$$

which means we choose the y value that maximizes the $P(y \mid \mathbf{x})$ among all possible values of y.

Similar to logistic regression, we need to estimate the conditional probability distribution $P(y \mid \mathbf{x})$. Unlike logistic regression, however, naive Bayes does not estimate this probability distribution directly. Instead we use the Bayes' formula to rewrite it as[11]

[11] This is why we have 'Bayes' in the name.

$$\begin{aligned}\hat{w} &= \arg\max_w P(y \mid \mathbf{x}) \\ &= \arg\max_w \frac{P(\mathbf{x} \mid y)P(y)}{P(x)} \\ &= \arg\max_w P(\mathbf{x} \mid y)P(y)\,.\end{aligned}$$

Note that simplification on the third step is possible since $P(x)$, the (marginal) probability of the data is the same for all model instances

(defined by the values of $\boldsymbol{w}$). The model parameters to be estimated, $\boldsymbol{w}$, are the parameters of two probability distributions: the distribution of the output classes $P(y)$, and the conditional distribution of input features given the class, $P(\boldsymbol{x} \mid y)$. Estimating $P(y)$ is easy in most cases, all we need to do is count the number of times the each class occur in the training data, and divide it to total number of data points. The second term, $P(\boldsymbol{x} \mid y)$, is harder to estimate as we typically have a large number of predictors ($\boldsymbol{x}$), and we cannot expect to estimate the joint probability of all of the predictors by counting the number of occurrences of all combinations of feature values. For any realistic problem, many of the combinations will not be observed, even in very large data sets.[12] The solution to this problem is making a *conditional independence* assumption. We assume that given the class labels, the predictors are independent of each other. As a result, their joint probability is multiplication of probabilities of individual predictors given the class. The conditional assumption is wrong.[13] Words in a document are clearly not independent of each other. However, in practice this works reasonably well for a large number of problems.

To make the above discussion more concrete, we will go through a toy example. We assume that we are building a spam detection application. With classes of spam (S) and not spam (NS), we have a binary classification problem at hand. And, our predictors are the set of words presented in Figure 5.8. We take the occurrence as a binary variable, a word occurs in a document or not, and train a spam detector on the training set given in Figure 5.9.

To estimate the prior probabilities of the classes, we count how many times each class occurs in the training set, and divide it into the total number of training examples: $P(S) = 3/5$ and $P(NS) = 2/5$. For likelihood, we need to calculate probability of observing a particular word in training instances belonging to each class (spam and non spam documents). We can conveniently represent this in a table as in Table 5.1.

Although there are some variations, training naive Bayes generally amounts to counting. Once we have estimated the relevant probabilities (parameters), we can find the probability of an email being spam by multiplying $P(S)$ with $P(w \mid S)$ for each word in the document. For example, for a test document that contains words *book technology*, we calculate both as follows.

$$P(S)P(\text{book} \mid S)P(\text{technology} \mid S) = \frac{3}{5} \times \frac{1}{3} \times \frac{1}{3}$$

$$P(NS)P(\text{book} \mid NS)P(\text{technology} \mid NS) = \frac{2}{5} \times \frac{2}{2} \times \frac{1}{2}$$

Note that these are not probabilities. We need to divide both quantities to probability of observing this email, but since both expressions are about the same email, its probability is the same for both classes. As a result we can pick the largest value, in this case indicating that the email is not spam.

[12] Even though predictors would be categorical in the typical use of naive Bayes, some of the predictors may even be continuous, making it impossible to rely on counting and dividing approach.

[13] The reason for the 'naive' in the name.

```
medication
free
technology
advanced
book
now
lose
weight
good
```

Figure 5.8: The list of features for the naive Bayes example.

good book (NS)
now book free (S)
medication lose weight (S)
technology advanced book (NS)
now advanced technology (S)

Figure 5.9: Example training data for spam detection. Each line represents an email, followed by its label, spam (S) or not spam (NS) in parentheses.

Table 5.1: Conditional probabilities of words given the class labels for the example data in Figure 5.9.

w	$P(w \mid S)$	$P(w \mid NS)$
medication	1/3	0
free	1/3	0
technology	1/3	1/2
advanced	1/3	1/2
book	1/3	2/2
now	1/3	0
lose	1/3	0
weight	1/3	0
good	0	1/2

An interesting detail is what happens if the test document contains, for example, *good*. According to Table 5.1, $P(\text{good} \mid S) = 0$. This means any document that includes word *good* cannot be spam. To solve this problem, we use a *smoothed* estimate of the conditional probabilities where a part of the probability mass is reserved for unseen events. We will discuss the smoothing later in this course along with n-gram language models.

Naive Bayes is a simple algorithm, and admittedly, it does not generally perform better than other (more recent) classification mechanisms. However, understanding naive Bayes may help understand other, more complex, methods in the literature.

5.4 A classification of classification models

The three classification methods we reviewed represent three different types of models in a broad categorisation of machine learning models. In case of perceptron, and similar models such as SVMs, the aim is to separate the classes from each other without making use of the probability theory.[14] These type of models are often called *discriminative* models in the literature. The aim of these models are to find a boundary that discriminate the data points that belong to different classes. The other two models we discussed are probabilistic. However, logistic regression predicts the *conditional probability* of class labels given the predictors, while the naive Bayes (under the hood) predicts the *joint probability* of the outcome and the predictors. Among these models, logistic regression is again a discriminative model. Although it assigns probabilities to each class for each data point, it forms a (soft) boundary for discriminating between classes. Naive Bayes, and similar models, are *generative*. They can assign probabilities to the data points, and since they model the complete data, one can generate data based on the model by sampling from the joint distribution.

[14] Although probability theory is not used for making predictions, some of the methods in this category make use of probability theory in the estimation, e.g., for minimizing the expected error on the test set.

In general, discriminative/conditional models tend to perform better in classification tasks mainly because they do model the problem directly, without putting additional attention to modeling the data. However, there are also cases where generative models are preferred. We will see some applications where the difference between generative and discriminative models is important.

5.5 Multi-class classification

Most of our discussion above focused on binary classification. Some classification methods, such as logistic regression and naive Bayes discussed above, has a natural way to handle multi-class classification problems. Some models, on the other hand, are designed to work with the binary case. There are two well-known strategies that turn any binary classifier to multi-class classifier. The main idea is training multiple binary classifiers, and making the final class assignment based on the predictions from all of the classifiers.

The first strategy we discuss is called *one vs. all*, or *one vs. rest*. To turn a binary classifier into a multi-class classifier that predicts one of k classes, we train k classifiers. Each time, we pick one of the classes as the positive class, and label the rest as the negative class. Figure 5.10 demonstrates the one-vs-rest strategy. The lines in the figure are the discriminators of individual classifiers, that are trained to discriminate one of the classes from the others.

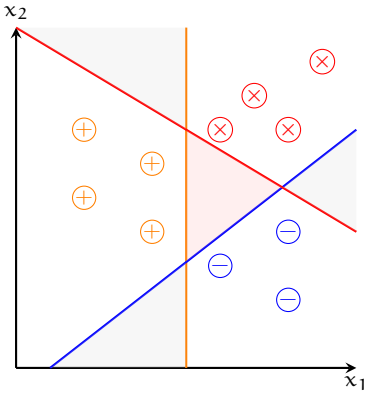

Figure 5.10: A demonstration of one-vs-rest multi-class strategy.

Note, however, that some regions of the input space will not be claimed by any of the classes, such as the shaded area in the middle. Furthermore, some regions will be claimed by more than one class. In these cases, if the base classifier returns a probability or confidence value, we pick the class with the highest score (or lowest negative score). Figure 5.11 presents the new multi-class decision boundaries based on the distances from the decision boundaries of the base classifiers.

The second strategy for forming multi-class classifiers from binary base classifiers is called *one vs. one*. One-vs-one strategy trains $\frac{k(k-1)}{2}$ classifiers where each classifier learns to separate only two of the classes from each other. The final prediction is typically made by majority voting. The class that is predicted most among all predictions is chosen as the predicted class. Ties, if they occur, may be broken based on either confidence values, or randomly. The one-vs-one strategy may be applied even when the base classifiers do not have any confidence or probability associated with their predictions. However, it also requires more computing power because of the number of classifiers it needs to train.

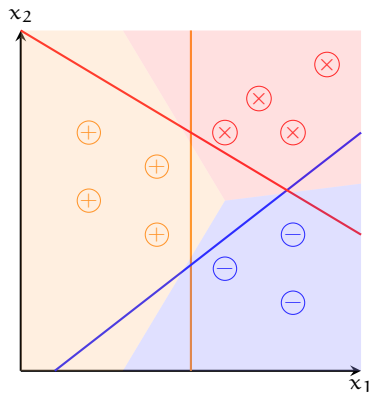

Figure 5.11: One-vs-rest classification with ambiguities resolved based on the distance from the decision boundary.

5.6 *Evaluation metrics for classification*

For regression, we use root mean squared error (RMSE) or the related measure R^2 to measure how badly or how well the model performs on a given data set with gold-standard outcome values. The RMSE is simply the square root of the average error we minimize while fitting the model. In case of classification, we care about the categorical match between the model's prediction and the actual values. The error function used by a particular classification algorithm is not necessarily a good measure of success that is easy to interpret.

A straightforward way to measure the success of a classifier is its *accuracy*. Accuracy is simply the number of correctly predicted labels divided by the total number of predictions.

Although accuracy is straightforward to calculate and interpret, it has a major flaw in some cases. We will illustrate this through an example. Let us assume that we are evaluating a search engine that labels documents in a large document collection as 'relevant' or 'not relevant' to a given query. Formulated like this, the search engine is a binary classifier. For most queries, there is only a small number of documents that are relevant. We assume that for a particular query, there are 1 000 relevant documents in a collection of one mil-

lion documents. Now, we will test a 'dummy' search engine which labels everything as 'not relevant'. If we calculate its accuracy in this setting,

$$\text{accuracy} = \frac{999\,000}{1\,000\,000} = 0.999.$$

Clearly, we do not want to credit this useless classifier with a success rate of 99.90 %. In general, accuracy is not a good measure if the class distribution is *imbalanced*, that is, if some of the classes are more frequent than the others.

To (partly) avoid this shortcoming of accuracy, we use two measures that originate from information retrieval.[15] The measures are called *precision* and *recall*. Before defining and discussing the measures, we need to define a few more concepts about the errors of a binary classifier makes. Table 5.2 shows possible outcomes of a binary classifier compared to the true (gold-standard) labels. *True positives* (TP) are the positive instances the model predicts correctly, *false positives* (FP) are the instances the model mistakenly predicts as positive, *false negatives* (FN) are the instances the model mistakenly classifies as the negative class, and finally, *true negatives* (TN) are the instances the model correctly predicts as the negative class.[16] Based on these counts, precision and recall are defined as

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

In words, precision is the ratio of the number of correct positive predictions to the number of instances predicted as positive by the classifier. Recall is the ratio of the number of correctly predicted positive instances to the number of all positive instances in the gold-standard data. If we return to the 'dummy' search engine example, since the number of true positives is zero, its precision and recall are both 0.

Typically, high precision comes with the cost of low recall, and high recall leads to low precision. Figure 5.12 presents the change in precision and recall of a logistic regression classifier, where the probability threshold for deciding for the positive class is varied. That is, each data point corresponds to a probability threshold. In some problems, we may want to trade one for the other. For example, for spam detection, probably a high-precision classifier is more important, since classifying a legitimate email as spam is worse than a few spam messages appearing in the inbox time to time. In tasks where the result is further refined (manually), one may prefer high recall not to miss many of the positive instances in the data.

Although having both measures often tell more about the performance of the classifier, sometimes we want a single number summary. The standard single-measure summary in this case is called F_1 *score* (or F-measure, or F-score) defined as

$$\text{F}_1 \text{ score} = \frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Table 5.2: A comparison of predictions of a binary classifier with gold-standard (true) labels.

		true label	
		positive	negative
predicted	positive	TP	FP
	negative	FN	TN

[15] Where, as evident from our example above, class imbalance is common.

[16] False positives and false negatives are called *Type I* and *Type II* errors respectively in the context of scientific tests.

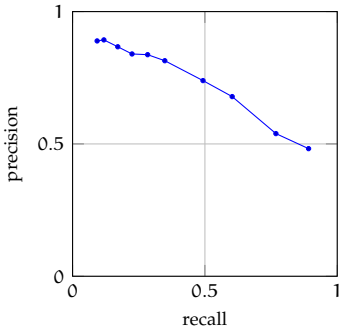

Figure 5.12: Precision-recall curve for a logistic regression classifier. Each data point corresponds to the classifier with the same parameters, but a different probability threshold (instead of 0.5 over which the model decides for the positive class. The graph does not show the threshold values. However, as expected, higher threshold values lead to high precision, low recall, and lower threshold values lead to low precision high recall. These graphs are also useful for comparing alternative models based on the *area under curve* (AUC). Other factors being equal, the models with larger AUC is preferable.

F_1 score is the harmonic mean of precision and recall. It is similar to the arithmetic mean if precision and recall are similar, but the harmonic mean is lower than the arithmetic mean if the difference between the precision and recall is high. A more general measure, F_β score, is defined as

$$F_\beta = \frac{(1+\beta^2) \times \text{precision} \times \text{recall}}{(\beta^2 \times \text{precision}) + \text{recall}}.$$

In this formulation, β values lower than 1 weighs precision higher than recall, and β values higher than 1 weighs recall higher than precision. Sometimes $F_{0.5}$ and F_2 scores are reported in the literature when there is a reason to prefer precision or recall. However, the uses of β other than 1 is rather rare, and if the subscript is dropped, F-score, refers to F_1 score.

Precision, recall and F-score as defined above works only for binary classification problems where there is a natural positive class.[17] For multi-class classification problems, or binary classification problems with no natural positive class, averaged versions of precision recall and F-score are used. There are two common methods of obtaining average performance scores. In *macro averaging*, we calculate the score for each class separately, and divide the result to the number of classes. In *micro averaging*, we average over all data points regardless of their class labels. More formally,

$$\text{precision}_M = \frac{\sum_i^C \frac{TP_i}{TP_i+FP_i}}{C} \qquad \text{recall}_M = \frac{\sum_i^C \frac{TP_i}{TP_i+FN_i}}{C}$$

$$\text{precision}_\mu = \frac{\sum_i^C TP_i}{\sum_i^C TP_i + FP_i} \qquad \text{recall}_\mu = \frac{\sum_i^C TP_i}{\sum_i^C TP_i + FN_i}$$

where C is the number of classes, the subscript i ranges over classes, e.g., TP_i is the true positives for class i, and M indicate macro, and μ indicate micro averaged scores.

Macro averaging favors models that perform equally well on all classes, regardless of number of instances in each class. Micro averaging yields higher scores if the model performs well on the classes with a large number of instances. In fact, the micro averaged F-score is equal to accuracy.

Although single-number performance indicators are useful for assessing the model's performance in general, there are a number of other well-known ways to get further insights about model's behavior. A very useful diagnostic for a model's behavior is the *confusion matrix*. The confusion matrix is a square matrix whose rows and columns correspond to predicted and true class labels. The cells of the matrix count the corresponding predicted and true labels.

Table 5.3 show an example (hypothetical) confusion matrix for a three-class sentiment classification. We can see from the confusion matrix that the model does well in predicting the negative class, predicting only 2 of the true negatives as neutral, but having quite a

[17] These measures do not care how the model performs on non-positive class(es). Note that true negatives (TN) is not even used in any of the definitions.

Table 5.3: An example confusion matrix.

		true class		
		neg.	neu.	pos.
predicted	negative	10	3	4
	neutral	2	12	8
	positive	0	7	7

few 'false positives' with respect to the negative class, predicting 3 of the neutral and 4 of the positive instances as negative. On the other hand, it is not very good at predicting the positive class, confusing it more often with neutral than predicting correctly. In general inspecting the confusion matrix may tell more about the model, and may also point to potential ways to improve the classification models.

We will return to issues of evaluation again. For now we close this part with repeating the most important rule of evaluation: since we want our models to be useful outside the training data, we have to evaluate them on a separate validation/test set.

5.7 *What we did not cover*

There are many classification methods, that we would not be able to cover in this lecture. Although it is not possible to cover all of them, a few of them deserve at least a short mention here.

Decision trees are interesting especially in cases where it is important to interpret the model's decision. An example decision tree is depicted in Figure 5.13. Decision trees are non-linear classifiers that are typically used with categorical features (but the example in the figure uses continuous features), and rely on information theoretic measures for learning. A related method *random forests*, which are a collection of decision trees trained on a subset of the data and/or features, generally perform better.

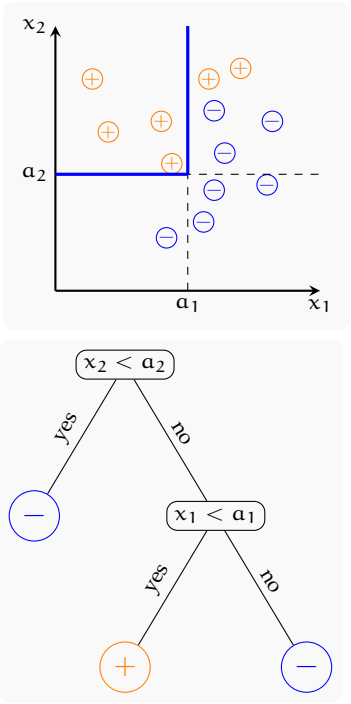

Figure 5.13: A decision tree (bottom) and the decision boundary it defines (top).

Another classification method we did not cover is *memory based learning*, also called *instance based learning* or *lazy learning*. The idea in memory based learning is to store all the training instances without processing. Classification decisions are made based on the nearest neighbors of the new instance with unknown label during prediction time. Figure 5.14 demonstrates the memory based learning. In the example given in the figure, the we use three nearest neighbors to predict the class of a test instance indicated with question mark. In this case, the negative class wins assuming we are using a simple majority voting decision.

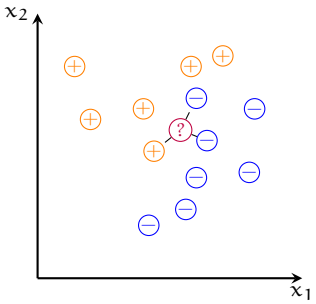

Figure 5.14: A demonstration of the memory-based learning.

Yet another interesting interesting method we already mentioned a few times is the *support vector machines* (SVMs). SVMs are quite similar to the perceptron. However, they pick the linear discriminator that maximizes the margin between the classes. Figure 5.15 demonstrates the SVM solution for the same problem demonstrated in Figure 5.3. Unlike perceptron algorithm, the linear discriminator found by SVM is equally distant to the nearest members of each class, which are called support vectors. The SVM also comes with a built-in regularization scheme that is motivated by minimizing the expected test error. SVMs are theoretically sound, successful linear classifiers that still produce the best results in quite a few classification tasks.

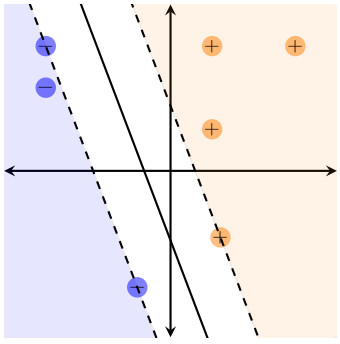

Figure 5.15: A demonstration of support vector machines.

Besides the various other interesting classification methods, we have not discussed how to handle non-linearity either. For most linear classifiers, the answer is similar to the non-linear regression. Use

of non-linear basis functions help linear classifiers learn non-linear solutions too. We will discuss non-linearity alongside artificial neural networks in a separate lecture.

Our discussion of the classification methods have been limited to binary or multi-class classification where label are mutually exclusive. In some classification problems, the objects we want to classify may belong to more than one class. For example, in topic classification, an article can be both on 'politics' and 'economy'. This type of classification problems are called *multi-label classification*. Similarly, the predictions we want to make in some problems fall into a natural hierarchy. Such classification problems, *hierarchical classification*, is yet another interesting variation of the problem that was not discussed in this lecture.

Finally, another related topic beyond the scope of this introduction is the *classifier ensembles*. It is known that combining predictions of different classifiers, under certain conditions perform better than a single classifier.[18] The general idea with ensembles is to combine prediction of a large number of *diverse* classifiers. The diversity can be achieved by different ways, for example by using different methods or alternations in the training procedure.

[18] The random forest classifier we briefly mentioned above is simply an ensemble of decision trees.

Summary

This lecture covers some of the main concepts in classification. We introduced three three simple (but important) classification methods, presented common general ways of extending a binary classifier to multi-class problems, defined and discussed common evaluation methods for classification, and provided a few related topics that we did not cover.

Classification is a highly applied and well-studied topic. Besides for our usual textbook references (Hastie, Tibshirani, and Friedman 2009; MacKay 2003; Bishop 2006), there is an immense amount of information (online or offline) on classification. It is difficult to suggest a good text that works for every purpose. The interested readers are recommended to find a source that works best for them, possibly starting with the references above.

We will cover classification by neural network in a separate lecture, and also cover sequence classification, both with traditional methods and neural networks, later in this course.

Bibliography

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning*. Springer. ISBN: 978-0387-31073-2.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Second. Springer series in statistics. Springer-Verlag New York. ISBN: 9780387848587. URL: http://web.stanford.edu/~hastie/ElemStatLearn/.

MacKay, David J. C. (2003). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press. ISBN: 978-05-2164-298-9. URL: http://www.inference.phy.cam.ac.uk/itprnn/book.html.

Minsky, Marvin and Seymour Papert (1969). *Perceptrons: An introduction to computational geometry*. MIT Press.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, pp. 386–408.