MA429 Algorithmic Techniques for Data Mining
Lecture 1: Fundamental concepts in data mining and statistical learning. The Naïve Bayes method

László Végh
L.Vegh@lse.ac.uk.

1 Data mining

Data mining is an interdisciplinary field that has emerged in the 1980s. It was necessitated by the previously unseen amounts of data aggregated in all domains of business, science, and public administration, as well as the emergence of the Internet. Data storage has become increasingly cheap and enabled automated data collection at large scale: the amount of data has been increasing exponentially over the past decades, and the trend is expected to continue. The main goal of data mining is the *automated extraction* of *implicit, previously unknown, and potentially useful* information from the data.

Data mining is at the intersection of a number of disciplines. It is closely related to *statistics*: data mining is often times called statistics at scale and speed. However, whereas classical statistics has a strong focus on hypothesis testing, in data mining there is more emphasis on *pattern discovery*. In data mining, we often do not have fixed hypotheses to start with, and it is part of the process to automatically identify potentially relevant ones. In statistics, the dataset is often times collected to test a specific hypothesis, for example, *"is there a significant difference between the number of men and women getting lung cancer"?*. In data mining, we may start with a dataset of hospital patients including all examination and measurement data. We would like to formulate hypotheses and predictions such as *"Is there any 'natural' typology of lung cancer patients? How efficiently can we predict if a person will get lung cancer, based on their various attributes?*

Also, whereas in statistics the number of samples is typically much larger than the number of predictors, in data mining it is often times the other way around. For example, we may have a dataset of digital pictures where every pixel can be seen as a predictor variable.

Data mining builds on different areas of *computer science*. A crucial area is *database methods*, that focuses on efficient storage, representation, and access of datasets. Data mining methods need to be implemented at enormous scale and speed (e.g. a search engine query): small improvements in the data structures might lead to massive savings. In this course, we will mainly focus on another key area: *machine learning*. Machine learning studies algorithmic techniques for the problem types specific for data mining. *Learning* in this context refers to using a set of examples, a *training dataset* to infer patterns that can be used to make predictions and inference on a larger unseen dataset where the training samples came from.

2 Basic concepts of learning

2.1 Training and test datasets

The usual input for a learning algorithm is a *training dataset* that comprises a set of *instances*, *samples*, or *data items* (these terms will be used interchangeably). The data items are described by *attributes*, *input variables*, or *features* (again, these terms can be interchanged). These variables will be mainly of two types: either *numeric (continuous)*, or *categorical (nominal/discrete)*. A numeric variable can have an arbitrary real value, or might be restricted to a certain interval (such as nonnegatives, or between 0 and 1). A categorical variable takes a value from a given fixed set. The simplest case is binary: yes/no or positive/negative. It can also take multiple values, e.g. the weather could be sunny/rainy/overcast.

Normally we think of the *training dataset* as an unbiased random sample from a larger dataset. For example, our hospital patient records are a sample from the total population. Our goal is to make predictions and inferences that are valid not only for the given training dataset but also for the entire dataset. We assume that besides the training dataset, we are also given another random sample, the *test dataset*. This dataset must not be used at all during the learning algorithm, but only to assess the reliability of predictions obtained using the training dataset.

In most cases it is (implicitly) assumed that the training and test datasets are independent and unbiased random samples. Note that this assumption is not at all obvious and in fact can be often times flawed: for example, people admitted to a hospital do not represent a uniform sample of the general population. We need to reflect on this assumption and make appropriate adjustments if necessary.

2.2 Supervised and unsupervised learning

We often times have an *output variable* or *target variable*. *Supervised learning* refers to the setting when the goal is to use a training dataset to build model to predict the value of the target variable. If the target variable is numerical, then the problem is a *regression problem*; if the output is categorical, then it is a *classification problem*. A fundamental example is linear regression. Here, our predictive model is a linear hyperplane that we construct such that it minimises the mean squared error of the prediction over the test set.

The fundamental model in supervised learning is the following. We let $X = (X_1, X_2, \ldots, X_p)$ denote the vector of p (numerical or categorical) predictor variables, and let Y denote the target variable. Then, the dependence is

$$Y = f(X) + \varepsilon,$$

where f is a (deterministic) function in p variables and ε is a random variable with mean 0, called the *noise term*. Thus, the predictor variables do not fully determine the value of the target, randomness is inherent. Our aim is to construct an *estimate function* $\hat{f}$. Given predictor values $x = (x_1, x_2, \ldots, x_p)$ for a data item, our prediction will be $\hat{y} = \hat{f}(x)$. The ideal case would identify $\hat{f} = f$, i.e. the estimate would identify the true dependence. This is typically impossible: the function f could be of an extremely complex form, and we have access to a limited data sample.

In contrast, in *unsupervised learning* there is no specific target variable. A basic example is *clustering*: we would like to identify distinct groups of the data items that are closer to each other (in a certain distance metric) than to members of the other groups. We will also study *association rule* mining: here, the dataset consists of groups of certain items, such as shopping basket data in a supermarket. The goal

is to identify items that occur together frequently. A further example is *anomaly detection*: we would like to identify irregular behaviour such as fraudulent transactions in a dataset of credit card transactions.

There is also the interm class of *semi-supervised* learning methods. In such problems, we have a prediction problem, but only a relatively small dataset L is labelled (i.e. the target variable is provided). We are also given another (typically larger) dataset U, where the input variables are the same but the target is unknown. We can make use of the unlabelled dataset U in various ways. For example, it might be possible to obtain labels for U but it is expensive (requires medical tests of patients or manual classification of images). Using L, we could identify the data items in U where querying the label has the most benefit.

2.3 Parametric and nonparametric methods

In linear regression, we always look for a linear estimate $\hat{f}$ in the form

$$\hat{f}(X) = \beta_0 + \sum_{i=1}^{p} \beta_i X_i.$$

Note that the "real" function f may be inherently nonlinear; still, we construct an estimate that gives the best fit on the training dataset achievable by a linear function (as measured by the mean squared error). This is called a *parametric method*, since the function is described by $p + 1$ parameters β_i; the learning algorithm aims to find the best combination of these parameters. The least square linear regression is the most fundamental choice, but we will also study further possibilities such as Ridge and Lasso regression.

In parametric methods we make a prior assumption that the estimate $\hat{f}$ belongs to a certain class of functions; every function in this class can be described by a fixed number of parameters. The number of parameters is the same irrespective to the number n, the number of points in the training dataset. A simple example is the class of linear functions, but parametric methods can be used to describe functions of very complex forms, and the number of parameters can be huge.

In *non-parametric methods*, the estimate $\hat{f}$ does not have a fixed form and can become increasingly complex as the number of training samples n grows. Simple non-parametric methods covered in the course will be the k-nearest neighbours method and decision trees.

2.4 Tradeoff between accuracy and interpretability

An important advantage of simple parametric methods is that they can give better insights. The goal is often time not only accurate prediction but possibly more complex inference, e.g. understanding the contribution of different input variables to the prediction. In linear regression, the coefficients β_i provide valuable such information. In contrast, more complex methods such as support vector machines or ensemble learning methods (that combine the output of several classifiers) may provide higher accuracy, but have a *black box* nature: we can hardly use them to understand structural aspects of the dataset.

Note that interpretability is not necessarily a feature of parametric methods. *Neural networks* can be seen as parametric methods (where the parameters are the weights associated with the different edges in the network); however, the number of parameters is very high and there is not much scope for interpretation. In contrast, the output of the non-parametric method of decision trees can be easy to interpret. Another example of a non-parametric method that are popular due to the easy interpretability are *rule based methods*. These methods can work well for classification task where all or most variables are categorical. We will later see the example of the *weather dataset*, where the goal is to predict whether a

certain game would be played given the weather conditions. A rule based method returns a list of rules such as

```
If outlook = rainy and windy = true then play = no
```

Depending on the application domain, one might prefer an output that has a clean interpretation but lower accuracy to a black-box method with higher accuracy. For example, trees or rules are more suitable to medical diagnosis or bank loan decision, where a causal justification is highly important.

3 The Naïve Bayes method

We now introduce a simple parametric classifier called the *Naïve Bayes* method. Let us first assume that all variables in the dataset are categorical; we will later extend the method for continuous variables as well.

Let us start by recalling *Bayes' theorem*. Let E and H be two events (we will think of E as 'evidence' and H as 'hypothesis'). The conditional probability $P(H|E)$ is the probability of the hypothesis assuming we know that E is true. Then,

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}.$$

Just as linear regression works under the assumption that the dependence between the variables is linear, the Naïve Bayes classifier is also based on a strong prior hypothesis on the dependence between the variables. We first introduce the version where all input variables are categorical. Let $X_1, X_2, \ldots, X_p$ be the categorical input and let Y be the categorical output variable. The possible values of Y are called *classes*; we denote them as $y_1, y_2, \ldots, y_k$.

For each class y_i, the input variables $X_1, X_2, \ldots, X_p$ are independent if conditioned on $Y = y_i$.

Let us now combine this assumption with Bayes' theorem. We are given a data instance with values $X_1 = x_1$, $X_2 = x_2$, …, $X_p = x_p$, and would like to compute the probability that the instance belongs to a certain class y_i. The hypothesis H is the event $Y = y_i$, whereas the evidence is the conjunction

$$E = (X_1 = x_1) \wedge (X_2 = x_2) \wedge \ldots \wedge (X_p = x_p).$$

Our goal is to compute $P(H|E)$ using Bayes' theorem. First, we can estimate the probability $P(H)$ as the proportion of instance in the class y_i among all instances in the training dataset. The independence assumption allows us to decompose $P(E|H)$ as

$$P(E|H) = P(X_1 = x_1|Y = y_i) \cdot P(X_2 = x_2|Y = y_i) \cdot \ldots \cdot P(X_p = x_p|Y = y_i).$$

Each term here is easy to estimate from the training dataset: we can estimate $P(X_j = x_j|Y = y_i)$ as the proportion of instances with $X_j = x_j$ among all instance in class y_i.

We can thus estimate $P(E|H) \cdot P(H)$ from the dataset. We do not need to compute $P(E)$, the denominator in Bayes' theorem. Instead, for each class y_i, we compute the *likelihood*

$$\ell_i = \hat{P}(E|Y = y_i) \cdot \hat{P}(Y = y_i).$$

Note that $P(Y = y_i|E) = \ell_i / P(E)$. Thus, the different probabilities $P(Y = y_i|E)$ are in the same ratios as the ℓ_i's. Therefore, we can estimate the probability

$$\hat{P}(Y = y_i|E) = \frac{\ell_i}{\sum_{i=1}^{p} \ell_i}.$$

The Naïve Bayes method is a parametric method where the parameters are the probability estimates $\hat{P}(Y = y_i)$ and $\hat{P}(X_j = x_j|Y = y_i)$ for every possible choice of j, X_j, x_j, and y_i. In the *learning algorithm*, we use the training dataset to estimate these values. In the test dataset, we can compute the probability estimates $\hat{P}(Y = y_i|E)$ for every instance and every class y_i using these parameters. The lecture slides demonstrate this calculation on the example of the *weather dataset*.

The output of the method will be a probability distribution on the classes $y_1, y_2, \ldots, y_k$. If we need to decide for one particular class, we simply return the one with the highest predicted probability. This is the default option, but in some applications we might decide differently. For example, assume we have a classification problem in medical diagnosis, where the two classes are positive and negative for a certain disease. Based on the outcome, we need to decide whether the patient needs to be sent for further tests or if treatment has to be started. In such a case, if the outcome probabilities are 0.4 for postive and 0.6 for negative diagnosis, it might be still justifiable to return positive as the answer (that is, decide for further tests or treatments). In certain cases, already a probability 0.1 for the positive outcome might be above the threshold. In the next lecture, we will study methods for finding the right threshold.

3.1 Missing values

It is very common for the dataset to include some missing entries. This can be a problematic issue for several learning method. For example, in linear regression we need to know the value of every input variable to substitute in the linear function. One option is to remove all instances with missing values, but this might be undesirable as it could mean forgoing a substantial part of the dataset. Another option is to substitute the missing values by certain estimates. Often the mean values of the dataset are used, however, this can lead to serious distortion in the data. Another option, discussed later on in more detail, is to compute missing values via another classification method.

A particularly nice feature of Naïve Bayes is that we can *simply ignore* all missing values. If the value of X_j is unknown for the instance, we skip the term $P(X_j = x_j|Y = y_i)$ for all classes y_i when computing the likelihoods ℓ_i.

3.2 Zero frequencies

Consider an input variable X_j and a class y_i. Let X_j have possible values $v_1, v_2, \ldots, v_m$. For each value h, we let r_h denote our estimate on the probability $P(X_j = v_h|Y = y_i)$. This is computed as follows. Let n_h denote the number of training instances with $X_j = v_h$ among all instances with $Y = y_i$, and let $N = \sum_{h=1}^{m} n_h$ denote the total number of training instances with $Y = y_i$. The simplest estimate is to use $r_h = n_h/N$, that is, the frequency of the value v_h in class y_i.

A problematic issue arises when $n_h = 0$ for some h. Using this estimate, the Naïve Bayes method will compute the likelihood $\ell_i = 0$ for every instance with $X_j = v_h$. This is undesirable: n_h might be due to the small size of the training sample; yet such a term would have 'veto' over all other input variables: we conclude 0 probability for class i even if the other input variables would suggest a high likelihood.

The *Laplace estimator* provides a simple fix. Instead of $r_h = n_h/N$, let us add +1 to each n_h value, and use the estimates

$$r_h = \frac{n_h + 1}{\sum_{h=1}^{m}(n_h + 1)} = \frac{n_h + 1}{N + m}.$$

3.3 Numerical variables

The Naïve Bayes method can be also extended to work with numerical variables. Here, we make the following assumption. *For each class $Y = y_i$, a numerical variable X_j comes from a normal distribution $N(\mu, \sigma)$.*

Let $v_1, v_2, \ldots, v_m$ be the values of X_j among the training instances in class y_i. Then, we use the estimates

$$\hat{\mu} = \frac{1}{m} \sum_{h=1}^{m} v_m,$$

$$\hat{\sigma} = \sqrt{\frac{1}{m-1} \sum_{h=1}^{m} (v_m - \hat{\mu})^2}.$$

Recall that the probability density function (p.d.f.) for the normal distribution $N(\mu, \sigma)$ is of the form

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Consider a test instance with $X_1 = x_1, \ldots, X_p = x_p$. When computing the likelihoods ℓ_i, for every continuous variable X_j we use the term $f_{ji}(x_j)$ instead of $P(X_j = x_j | Y = y_i)$ in the product, where f_{ji} is the p.d.f. of the normal distribution with parameters estimated from the values of X_j of the training instances in class y_i.

This can be justified as follows. Given X_j in class y_i with p.d.f. f_{ji}, and given a value x_j, the probability that X_j is in the interval $[x_j - \varepsilon/2, x_j + \varepsilon/2]$ for a small $\varepsilon > 0$ is

$$P(x_j - \varepsilon/2 \leq X_j \leq x_j + \varepsilon/2) \approx \varepsilon f_{ji}(x_j).$$

We can think of ε as a fixed accuracy of the numerical values. The same ε term appears for each class y_i, and hence we can remove it when computing the likelihoods.

3.4 Discretisation

Discretisation is a general technique that can be used to convert a numerical variable to a categorical one. For example, we can take a numerical variable 'temperature', and change it to a categorical variable with three possible values, 'cool', 'mild', and 'hot'.

Such a transformation necessarily looses information from the dataset: e.g. if the threshold between 'mild' and 'hot' was 25°C, then instances with 24°C and 26°C will receive different labels, yet the instance with 26°C will have the same label as another one with 39°C.

Still, discretisation can be advantageous for certain learning methods. For example, when using the Naïve Bayes method, we have a numerical variable where the normal distribution assumption is highly violated, the performance of the method can largely improve when discretising, as this replaces the normality assumption by a much weaker one. It can also be beneficial for other methods that work primarily with categorical variables.

There are various approaches for discretisation; these can be *unsupervised* or *supervised*. The key difference is that unsupervised methods do not take the class labels into account, only the values of the variable being considered. In contrast, supervised methods aim to construct 'pure' intervals, that is, where most instances in the same interval belong to the same class.

For *unsupervised* discretisation, we typically have to specify the number of intervals. Then, we can either split the value range into equal length intervals, or we can use intervals that all contain approximately the same number of instances.

In *supervised* discretisation, the extreme case would correspond to ordering all instances by the value, and starting a new interval whenever the class changes. (This might of course not be possible, as multiple instances may have the same value but can belong to different classes.) This would however fragment the values into too many intervals. It would also *overfit* the training dataset: the intervals would be the best possible for the training instances, but is likely to be a poor fit for test instances.

There are various supervised discretisation methods that compensate for these effects, and try to fit a small number of relatively homogenous intervals.

4 References

- General overview of data mining and machine learning: **ISLR** Sec. 2.1, and **Weka** Chapters 1-3 (skim through).
- Naïve Bayes: **Weka** Sec 4.2.

MA429 Algorithmic Techniques for Data Mining
Lecture 2: Evaluation methods. The k-nearest neighbours algorithm

László Végh
L.Vegh@lse.ac.uk.

1 Measuring the error

In linear regression, we find a hyperplane that minimises the mean squared error (MSE) over the training dataset. This MSE value is however an overly optimistic performance measure. To assess the accuracy of the classifier, we need to strictly separate the training dataset Tr and the test dataset Te. Both the training and test dataset comprise pairs (x, y), where x is a vector of predictor variables and y is the target variable. We use Te to construct the estimate $\hat{f}$. The most common performance measure of a regression problem is the mean squared error on the test dataset:

$$\mathrm{MSE}_{\mathrm{Te}} = \frac{1}{|\mathrm{Te}|} \sum_{(x,y)\in \mathrm{Te}} (y - \hat{f}(x))^2.$$

1.1 Classification problems

In a classification problem, the target y is categorical with possible values $y_1, y_2, \ldots, y_k$. The estimate $\hat{f}(x)$ returns one of these values. An analogue of the MSE is the *error rate*, the average number of misclassifications:

$$\mathrm{err}_{\mathrm{Te}} = \frac{1}{|\mathrm{Te}|} |\{(x, y) \in \mathrm{Te} : y \neq \hat{f}(x)\}|.$$

More detailed information from the classification accuracy can be obtained from the *confusion matrix*. This is a $k \times k$ matrix M, where the entry M_{ij} represents the number of instances in Te that are in class y_i, and were predicted y_j. Clearly, the error rate corresponds to the off-diagonal entries:

$$\mathrm{err}_{\mathrm{Te}} = \frac{1}{|\mathrm{Te}|} \sum_{i \neq j} M_{ij}.$$

Let us now focus on the special case of binary classification, where the confusion matrix is a 2×2 matrix. We label the two classes *positive* (+) and *negative* (−).

		Predicted class	
		+	−
Actual class	+	True Positive (TP)	False Negative (FN)
	−	False Positive (FP)	True Negative (TN)

Using this notation, we define five important quantities:

- Error rate (as above)
$$\frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},$$
- Success rate or accuracy (= 1−error rate)
$$\frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}},$$
- Precision
$$\frac{\text{TP}}{\text{TP} + \text{FP}},$$
- Recall/ true positive rate
$$\frac{\text{TP}}{\text{TP} + \text{FN}},$$
- Selectivity/ true negative rate
$$\frac{\text{TN}}{\text{TN} + \text{FP}}.$$

Note that selectivity is the same as recall with the classes + and − swapped. Precision and recall can be important for different problem types. For example, for medical diagnosis a high recall is very important, i.e. we need to minimise false negatives, while we may care less about false positives. A high precision can be important e.g. for a loan decision, where false positives can be much worse than false negatives.

If there are more than two classes, one can define the above quantities by selecting one class as "positive", and all others as "negative".

Kappa statistics Another useful evaluation measure of the confusion matrix is the following. Assume we have a dataset of N instances and k classes $y_1, y_2, \ldots, y_k$ in proportion $(p_1, p_2, \ldots, p_k)$, that is, $y_i = Np_i$. Then, a random prediction made from this distribution would in expectation classify Np_ip_j instances of class y_i into class y_j. Such a random guess would correctly classify in expectation

$$\text{success(random)} = \sum_{i=1}^{k} p_i^2$$

fraction of all instances. We compare this to the success rate of the classifier,

$$\text{success(observed)} = \frac{1}{N} \sum_{i=1}^{k} M_{ii}.$$

The kappa-statistics is defined as

$$\kappa = \frac{\text{success(observed)} - \text{success(random)}}{1 - \text{success(random)}}.$$

Clearly, $\kappa \leq 1$. Note that the κ statistics could be negative, in which case our classifier does worse than a naïve random guess from the class distribution.

Probabilistic outputs Many classifier algorithms return a probability distribution over classes rather than a single class (recall the Naïve Bayes method). Then, instead of the error rate, we can use a loss function to measure accuracy. Consider a fixed test instance where the true class is y_h. Let $(p_1, p_2, \ldots, p_k)$ be the distribution returned by the classifier. The actual classification is given by a 0-1 vector $(a_1, a_2, \ldots, a_k)$, where $a_h = 1$ and $a_i = 0$ for $i \neq h$.

The most common measure is the *quadratic loss function*:

$$\sum_{i=1}^{k}(p_i - a_i)^2 = 1 - 2p_h + \sum_{i=1}^{j} p_i^2.$$

We compute the average quadratic loss over all test instances.

Another possible measure is the *informational loss function*, defined as

$$\log_2 \frac{1}{p_h}.$$

In contrast to the quadratic loss function, this does not depend on the probabilities p_i associated to classes $i \neq h$. However, in case $p_h = 0$, the loss is defined to be ∞. (This can be avoided using a solution similar to the Laplace estimator for Naïve Bayes.)

The two loss functions are on a different value scale: the quadratic loss will have values between $[0, 2]$, and the informational loss function in $[0, \infty)$.

For both loss functions, if the data instances come from a distribution with class probabilities $(p_1^*, p_2^*, \ldots, p_k^*)$, then both the expected quadratic loss as well as the informational loss are minimised for the distribution $p = p^*$.

The expected minimum value for quadratic loss is $\sum_i {p_i^*}^2$, and for informational loss it is $\sum_i p_i^* \log \frac{1}{p_i^*}$, the *entropy* of the distribution.

ROC-curves Consider a binary classification problem where we have a probabilistic classifier returning probabilities (p_0, p_1) for the negative and the positive class. By default, we can return the class with higher probability. However, we can set a different threshold: we return a positive answer if $p_1 \geq \alpha$ for some $\alpha \in [0, 1]$. We may get different confusion matrices for different values of α, with different selectivity and recall values that we denote by selectivity(α) and recall(α).

The *ROC-curve* (receiver operating characteristics) is defined as follows. We construct a graph where the x-axes corresponds to 1−selectivity (false positive rate) and the y-axes to recall (true positive rate). That is, the x-axes plots $\frac{\text{FP}}{\text{TN+FP}}$ and the y-axes plots $\frac{\text{TP}}{\text{TP+FN}}$.

We represent every $\alpha \in [0, 1]$ value by the point (1−selectivity(α),recall(α)). The best possible point would be the point $(0, 1)$, meaning that there are no false positives and no false negatives.

For $\alpha = 0$, both the false positive and false negative rates will be 1 since TN = FN = 0. For $\alpha = 1$, the false positive rate is close to 0, but the recall will also be close to 0. The ROC curve is the curve connecting these two points as α goes from 0 to 1.

In the best case scenario, the ROC curve passes close to the corner $(0, 1)$; in such case we can select a value of α that makes a very good prediction. Thus, a natural measure is the *area under the ROC curve (AUC)*. This is between 0 and 1. A random guess would correspond to a ROC curve near the diagonal with AUC = 0.5.

2 The bias-variance decomposition

Recall from the previous lecture the following model for supervised learning. Given a vector $X = (X_1, X_2, \ldots, X_p)$ of p predictor variables, and a target variable Y, the dependence is

$$Y = f(X) + \varepsilon,$$

ε is a random variable with mean 0 called the noise term. Let us now focus on regression problems, i.e. when Y is numerical.

The learning algorithm has access to a training dataset Tr, and can use this to construct an estimate $\hat{f}$; we add the index $\hat{f}_{\mathrm{Tr}}$ to emphasise the dependence on the training sample.

Consider now a fixed test instance with predictor values $x = (x_1, x_2, \ldots, x_p)$ and target value y. Our estimate from is $\hat{y} = \hat{f}_{\mathrm{Tr}}(x)$, and we would like to bound the variance of the error

$$E_{\mathrm{Tr},\varepsilon}(y - \hat{y})^2 = E_{\mathrm{Tr},\varepsilon}(y - \hat{f}_{\mathrm{Tr}}(x))^2.$$

Note that here the instance (x, y) is fixed, and the expectation is over the random choice of the training dataset Tr as well as over the random noise term ε. This variance can be decomposed into the sum of three terms:

$$\begin{aligned} E_{\mathrm{Tr},\varepsilon}(y - \hat{f}_{\mathrm{Tr}}(x))^2 &= \mathrm{Var}(\hat{f}_{\mathrm{Tr}}(x)) + \left[\mathrm{Bias}(\hat{f}_{\mathrm{Tr}}(x))\right]^2 + \mathrm{Var}(\varepsilon), \quad \text{where} \\ \mathrm{Var}(\hat{f}_{\mathrm{Tr}}(x)) &= E_{\mathrm{Tr}}\left[\hat{f}_{\mathrm{Tr}}(x) - E_{\mathrm{Tr}}[\hat{f}_{\mathrm{Tr}}(x)]\right]^2, \\ \mathrm{Bias}(\hat{f}_{\mathrm{Tr}}(x)) &= E_{\mathrm{Tr}}[\hat{f}_{\mathrm{Tr}}(x)] - f(x). \end{aligned} \tag{1}$$

The term $\mathrm{Var}(\varepsilon)$ is the variance of the noise term. The *variance* term measures the variance of the random variable $\hat{f}_{\mathrm{Tr}}(x)$ (again, this is over the random selection of the test dataset). The *bias* term measures the difference between the expected value of $\hat{f}_{\mathrm{Tr}}(x)$ and the value $f(x)$.

Note that all three terms in the decomposition are nonnegative, in particular, $E_{\mathrm{Tr},\varepsilon}(y - \hat{f}_{\mathrm{Tr}}(x))^2 \geq \mathrm{Var}(\varepsilon)$: this term represents an irreducible error due to the inherent random noise. In order to get close to this bound, we would need to simultaneously achieve a low variance and low bias.

There is a natural *tradeoff* between variance and bias. For example, linear regression (as well as other simple parametric methods) typically have low variance but high bias. Low variance means that the estimate $\hat{f}$ is robust against the choice of the training dataset: we get similar fits for different (but sufficiently large) subsamples of the dataset. On the other hand, if the dependence f is non-linear, then we can have a high bias: the estimate is expected to be far from the true value f in typical test cases.

More flexible methods that do not assume a simple parametric structure of $\hat{f}$ may have lower bias. On the other hand, $\hat{f}$ may highly depend on the choice of the training dataset, leading to high variance. We will study this phenomenon in the example of the k-nearest neighbours method.

3 The k-nearest neighbours method

The k-nearest neighbour is one of the simplest non-parametric methods. It is applicable to both regression and classification problems. In its simplest form, we do not do anything in the learning stage, only store the training dataset Tr. Consider a test instance $(x^0, y^0) \in \mathrm{Te}$, and let $N_k(x^0)$ denote the k nearest points

to x in Tr in a certain distance measure (to be discussed below). In a regression problem, we simply return the average value for the k nearest neigbours:

$$\hat{f}(x^0) = \frac{1}{k} \sum_{(x,y)\in N_k(x_0)} y$$

For a classification problem, we return the probability distribution of the classes in $N_k(x^0)$. If the possible class values are $(y_1, y_2, \ldots, y_k)$, then we let

$$\Pr(Y = y_j | X = x^0) = \frac{1}{k} \left| \{(x,y) \in N_k(x^0) : y = y_j\} \right|.$$

For $k = 1$, we simply consider the nearest neighbour to x^0 and return its target value or class. This well illustrates the point why we should *not* measure the success rate of a learning method on the training dataset: for 1-nearest neighbours this would show 100% accuracy!

The k-nearest neighbours method is called a "lazy" classifier, since the learning stage is void. However, the prediction stage requires a nontrivial computational effort: identifying the neighbour set $N_k(x^0)$. For a large training dataset Tr computing the pairwise distances between x^0 and any point in Tr would be very time consuming. There are various pre-processing methods to construct a data structure on the training dataset that enables faster identification of neighbours; we do not discuss them in this course.

3.1 The distance measure

An important detail is how to measure the distance of two data points. Assume we have two predictor vectors with p predictors: $x = (x_1, x_2, \ldots, x_p)$ and $z = (z_1, z_2, \ldots, z_p)$. If all predictors are numerical, then we can use the standard Euclidean distance

$$d(x,y) = \sqrt{\sum_{i=1}^{p} (x_i - z_i)^2},$$

or the ℓ_1-distance (also called Manhattan-distance)

$$d_1(x,y) = \sum_{i=1}^{p} |x_i - z_i|.$$

Normalisation Different predictors can be on vastly different scales. For example, in a dataset we can have the age and the annual income of a person. Simply adding up the corresponding distance terms means that a tiny difference in income counts more than a huge age difference.

Therefore, it is customary to *normalise* the predictors. Let us fix one of the predictors x_i for the discussion. We select lower and upper thresholds m and M in the value range, and apply a linear transformation that brings m to 0 and M to 1. Namely, we replace each value t by

$$t \to \frac{t - m}{M - m}.$$

A simple choice is to select m to the minimum and M to the maximum value in the training dataset. Then, all values in Tr will be transformed to the $[0, 1]$ interval (however, test instances may still have higher or lower values). A problematic issue with such normalisation is if the dataset contains *outliers*: instances with off-the-scale values could distort the normalisation.

Thus, we can use other possible choices, e.g. setting m to the first and M to the third quartile in Tr. We might also have some a priori domain knowledge that enables selecting some natural thresholds.

Alternatively, we can *standardise* the variables by setting their mean to 0 and standard deviation to 1. That is, we compute the estimates $\hat{\mu}$ and $\hat{\sigma}$ for mean and standard deviation, respectively. Then, we replace each value t by

$$t \to \frac{t - \hat{\mu}}{\hat{\sigma}}.$$

Categorical attributes Thus far, we assumed all predictors are continuous. The k-nearest neighbours method can also be extended if categorical attributes are present. If the j-th attribute is categorical, then the corresponding term the distance measure will be 0 if $x_j = z_j$, and 1 if $x_j \neq z_j$.

3.2 The choice of k and the bias-variance tradeoff

A key question is how to choose the value k. As the examples show (see slides), a small k results in ragged classification boundaries and a high sensitivity to the choice of Tr as well as to noise and error. For a large k we get more smooth and more robust boundaries. However, if k is *too* large, then it can "iron out" some genuine classification differences, and lead to some regions disappearing. (What happens if you set $k = |\mathrm{Tr}|$?)

The dependence on k can be understood through the lense of bias-variance decomposition. Let us consider a regression problem with dependence $Y = f(X) + \varepsilon$. For simplicity of discussion, let us assume that besides the test instance (x, y), the training data set Tr is also fixed while we compare different values of k. We measure expectation over the random noise terms ε both for the points in Tr as well as for the test instance. We order the points in Tr as $x^{(1)}, x^{(2)}, x^{(3)}, \ldots$ in increasing order of distance from x. These have target values $y^{(i)} = f(x^{(i)}) + \varepsilon^{(i)}$. Let

$$\hat{f}^{(k)}(x) = \frac{1}{k}\sum_{i=1}^{k} y^{(i)} = \frac{1}{k}\sum_{i=1}^{k} f(x^{(i)}) + \frac{1}{k}\sum_{i=1}^{k} \varepsilon^{(i)}$$

be the estimate obtained when using k neighbours. Let us consider the expected variance $E(y - \hat{f}^{(k)}(x))^2$ for each k (with expectation over the noise terms). The inequality (1) takes the form

$$\begin{aligned} E(y - \hat{f}^{(k)}(x))^2 &= \mathrm{Var}(\hat{f}^{(k)}(x)) + \left[\mathrm{Bias}(\hat{f}^{(k)}(x))\right]^2 + \mathrm{Var}(\varepsilon) = \\ &= \frac{\mathrm{Var}(\varepsilon)}{k} + \left[f(x) - \frac{1}{k}\sum_{i=1}^{k} f(x^{(i)})\right]^2 + \mathrm{Var}(\varepsilon) \end{aligned}$$

We see that as k increases, the variance term decreases, as it is the variance of k independent random variables from the same distribution. On the other hand, the bias term increases: we compare $f(x)$ to the average of an increasing number of points of larger and larger distance from x. We can find the best value of k where the bias-variance tradeoff is optimal.

3.3 The curse of dimensionality

The k-nearest neighbours method looks very intuitive in 2 or 3 dimensions. However, the number of predictors p can be much higher. Our low dimensional geometric intuitions are misleading for high dimensional spaces. In particular, the nearest neighbours will not be so near at all.

To illustrate this, assume that all p predictors are normalised between 0 and 1: the data points are elements of the p-dimensional hypercube $[0,1]^p$. Assume that our data set comes from a uniform distribution in the hypercube and we use Euclidean distance.

The points within distance ρ from a certain point z are in the p dimensional ball of radius ρ that has volume $C_p\rho^p$ (C_p is a constant dependent on the dimension, e.g. $C_2 = \pi$.)

Assume we have N data points in Tr. Then, the total volume of the set of points x that have at least one training instance $z \in$ Tr within $d_1(x,z) \leq \rho$ can be at most $NC_p\rho^p$. Thus, if we want to have a training dataset such that every point in $[0,1]^p$ has a point in Tr within distance ρ, we need

$$N \geq \frac{1}{C_p\rho^p}$$

For $p = 10$, with $\rho = 0.5$ we would need $N \geq 402$, and for $\rho = 0.2$, we need $N \geq 3,829,410$. For $p = 20$, $\rho = 0.5$ we would already need $N \geq 411,480$.

Note that in p dimensions, the largest possible distance between any two points (two opposite corners of the cube) is $\sqrt{p}$.

How can k-nearest neighbours still be used for high dimensional problems? One reason can be that the dataset is typically not uniformly distributed in the possible value range ($[0,1]^p$), but the attribute values are highly correlated and the points are concentrated in a smaller region close to a lower dimensional surface.

Still, if we consider k-nearest neighbours for high dimensional problems, it is often beneficial to reduce to a lower dimensional space, either by removing attributes or by transforming the attributes (see later lectures).

4 References

- Measuring the error and bias-variance decomposition: **ISLR** Sec. 2.2, **ESL** Sec 7.3, and **Weka** Chapter 5.
- k-nearest neighbours: **ISLR** Sec 2.2.3, **ESL** Sec 13.3

MA429 Algorithmic Techniques for Data Mining
Lecture 3: Cross validation and bootstrap. Model selection and regularisation

László Végh
L.Vegh@lse.ac.uk.

1 Cross-validation and bootstrap

As emphasised previously, to assess the accuracy of a learning method we must use separate training and test datasets. It is however very common that we only have a single dataset D provided with $|D| = n$. The larger dataset we use, the more accurate estimates $\hat{f}$ we can obtain since a larger dataset reduces the variance of the estimate. Hence, if computationally feasible, we will always use the entire available dataset to construct our estimate $\hat{f}$ that we will use for later applications.

However, in order to assess the accuracy, we must devise methods for measuring test error. In the best case scenario, we can easily and cheaply obtain additional fresh data for testing. We now discuss the situation when this is not an option.

Holdout method We can randomly split D into Tr and Te, e.g. in half-half or $2/3 - 1/3$ proportions. This method has two drawbacks:

- Since Tr is significantly smaller than D, we will overestimate the test error rate.
- The error rate may be highly dependent on the particular split used: hence, the holdout error estimate has a high variance.

To reduce the variance of the error estimation, it is natural to repeat the experiment many times. The methods described next present more disciplined ways of doing this.

k-fold cross validation In k-fold cross validation, we randomly split D into k parts $D_1, D_2, \ldots, D_k$ of roughly equal size. We then perform k experiments, called *folds*. In the i-th experiment, we set $\mathrm{Tr} = D \setminus D_i$ and $\mathrm{Te} = D_i$. That is, we will have a training data set of size $(1 - \frac{1}{k})$ times the size of D.

We will now have k estimates for the test error rate (as well as for precision, recall, etc.) We return the mean of these k values as the overall estimate.

We can conveniently construct the confusion matrix in cross-validation: since each data item is used for testing exactly once, the predicted class will be the one returned in this fold.

What is the right choice of k as the number of folds? The extreme case is $k = n$, called *leave-one-out cross-validation*: every fold leaves out just a single item and uses that for testing. The advantages are that it uses the largest possible datasets for training, and that no randomness is involved in creating the

split. On the other hand, it is computationally very expensive, as we need to repeat the entire learning process n times.

Finding the right value of k has to balance between computational feasibility and having a good error estimation. The latter can be understood through the bias-variance decomposition. First, the larger training set, the smaller the bias; in terms of bias, $k = n$ would be the best choice.

On the other hand, as the value of k grows, the k models will become more and more correlated (since there is an increasing overlap between the training sets). Thus, the variance increases.

Typically, a value of k between 5 and 10 is a good choice that can be computationally feasible and also gives good estimates of the performance.

The bootstrap method Bootstrap is a general statistical approach to create new samples from a given dataset. From the dataset D, we sample n elements (recall $n = |D|$) *with replacement*. Let B be the sampled set that may contain multiple copies of the same item.

The number of elements that have been sampled at least once is

$$\sum_{v \in D} \Pr(v \in B) = n \cdot \left(1 - \left(1 - \frac{1}{n}\right)^n\right) \approx \left(1 - \frac{1}{e}\right) n \approx 0.632n$$

In the context of estimating error rates, we can construct k bootstrap samples. For each sample, we use $\mathrm{Tr} = B$, and $\mathrm{Te} = D \setminus B$, that is, the elements who have been completely left out from D. According to the above calculation, the size of Te is approximately $0.368n$. We estimate the error by averaging the test errors of these experiments; this is called the *out-of-bootstrap validation*.

There are further techniques for error estimation via bootstrap, see **ESL** Sec 7.11.

2 Subset selection methods

Typical datasets may include a vast number of predictor variables; we can often face situations when p, the number of predictors is much larger than n, the size of the dataset. Having more predictors is not always useful. Some of the predictors can amount to noise: randomly chosen values that are independent from the target variable. In k-nearest neighbours all predictors are weighted equally: thus, noise predictors will deteriorate the performance. For k-nearest neighbours, a low number of predictors is preferred also because of the curse of dimensionality.

For Naïve Bayes, noise by itself is not a major issue (why?). At the same time, having highly correlated predictors can be a problem as it violates the underlying independence assumption.

If $p > n$ then the least square fit is not unique and thus linear regression cannot be applied directly. Even if n is only slightly larger than p, there will be high variability in the estimates. Reducing the number of predictors is also desirable for interpretability of the coefficients, as we will have fewer non-zeros.

There are three main approaches to reduce the number of the predictors:

- *Subset selection:* Identify a subset of relevant predictors, and apply the learning method restricted to this subset.
- *Shrinkage/regularisation:* apply the learning method on the entire set of predictors, however, use a regulariser term to stipulate a solution with fewer and smaller coefficients.

- *Dimension reduction:* transform the space of predictors into a lower dimensional space, by creating new synthetic predictors. An important example will be principal component analysis (PCA), discussed later in the course.

We now start with subset selection methods. Let $P = \{1, 2, \ldots, p\}$ be the subset of all predictors. For each subset $S \subseteq P$ of the predictors, we assign a score $\omega(S)$ that measures how good this subset is (the bigger the score the better). We postpone the definition of the score. Ideally, we should select a subset that maximises this score. Given that the number of possible subsets is 2^p, for all but small values of p it is computationally infeasible to check all possible subsets.

Thus, subset selection methods try to obtain a *locally optimal* subset. The two simplest methods are forward selection and backward selection. In the *forward selection* method, we start with $S = \emptyset$, and in each iteration, we either increase the size of the subset by one or terminate. Namely, let S_k be our best subset with $|S_k| = k$. Then, we consider every predictor $i \in P \setminus S_k$ the candidate subset $S_k \cup \{i\}$. Among all these, we pick $i = i_{k+1}$ that maximises $\omega(S_k \cup \{i\})$. In case $\omega(S_k \cup \{i_{k+1}\}) > w(S_k)$, we set $S_{k+1} = S_k \cup \{i_{k+1}\}$ and proceed to the next stage. Otherwise, if adding a new element may only decrease the score, then we terminate by outputting the current subset S_k.

In *backward selection*, we start from the entire attribute set $S_p = P$, and every iteration proceeds by considering subsets of size one less. That is, if S_k is our best choice with $|S_k| = k$, then we pick $S_{k-1} = S_k \setminus \{i\}$ for some $i \in S_k$, such that the choice maximises $\omega(S_{k-1})$. If we cannot find a subset with a better ω value, then we terminate with the current subset.

There are further, more sophisticated methods, such as *bidirectional search*. In this variant, for the current subset S we check all neighbours that can be obtained by adding or removing one element, and selecting the modification that yields the highest gain in the score. We stop once we cannot increase the score any further by such changes. We could also look at larger neighbourhoods, i.e. all sets S' that can be obtained by adding and removing at most k attributes in total, for some fixed k.

Such methods can find better subsets, but the broader search we perform, the longer it might take. For the simple forward and backward selection methods, the subset is guaranteed to change $\leq p$ times; for bidirectional search, the number of iterations could be much larger.

2.1 Measuring the score

There are various approaches for evaluating the score $\omega(S)$. This should be an estimator of the error rate of the learning method if applied with the subset S of predictors. Note that the MSE (regression) or error rate (classification) are typically not appropriate choices, as they underestimate the true test error.

One can combine subset selection with *cross-validation*: when evaluating a subset S, we compute $\omega(S)$ as the error estimation from k-fold cross-validation for the subset S of predictors. This provides a direct estimate on the error rate and does not require any assumptions on the underlying model. The drawback is the computational cost: it increases the running time by a factor k. This used to be prohibitive in the past, but with the increased computational capabilities today, cross validation became a viable option.

The alternative is to use a corrected version of the error rate on the training set. The corrected error estimates the *in-sample prediction error*. Recall the model $Y = f(X) + \varepsilon$. If we take a point in the training sample with predictors X, and resample the error, we get a new value Y'. The in-sample error would compute the error rate with respect to these new target values for each point in the training sample.

The in-sample error can be estimated using various criteria using different statistical models: examples

are C_p statistics, AIC, and BIC. When using a subset with $|S| = d$ parameters, Mallow's C_p statistics is

$$C_p = \text{MSE} + \frac{2d}{n}\widehat{\text{Var}}(\varepsilon).$$

Here, $\widehat{\text{Var}}(\varepsilon)$ is an estimate on the variance of the noise term ε. For linear regression, we can estimate this by constructing a linear model on the full dataset with all predictors.

The AIC (Akaike information criterion) is a more general model. For the special case of linear regression with Gaussian noise assumption, it will be proportional to C_p.

More detail about various criteria and the underlying statistical assumptions can be found in **ESL** Section 7.5.

3 Shrinkage methods: Ridge and Lasso regression

We now discuss two simple extensions of linear regression, when a regularisation term is added to the SSE (sum of squared error) objective. Recall that for n data points and p predictors, linear regression minimises

$$\text{SSE}(\beta) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2.$$

3.1 Ridge regression

In the *ridge regression*, we instead minimise

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

for some parameter $\lambda \geq 0$. For $\lambda = 0$, this is the same as ordinary linear regression, whereas a very high value of λ would stipulate an estimate where all $\beta_i = 0$ for $i \geq 1$. Note that we do not penalise the intercept term β_0.

We can understand the effect of changing λ from the bias-variance decomposition. When the number of predictors p is high compared to the sample size n, linear regression can have a high variance; the variance decreases as λ increases; in turn, the bias will increase. Note that linear regression is not applicable for $p > n$, whereas ridge regression can still be used.

An equivalent way of writing ridge regression (via Lagrangian duality) is

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \\ \text{s.t.} \quad & \sum_{j=1}^{p} \beta_j^2 \leq t. \end{aligned}$$

In fact, for each value of λ there is a value of t such that the optimal y in the two formulations coincide, and vice versa. The intuition in this form is that we have a given budget for coefficients and want to find the best fit within this budget. Note that if t is sufficiently large, then the value β minimising the MSE will be feasible, and thus we get the same solution as for linear regression.

Scaling Since y_0 is not included in the penalty term, if we add the same constant value c to every y_i, the optimal β_0 becomes $\beta_0 + c$. In ordinary linear regression, a linear transformation replacing X_i by αX_i for any $\alpha \neq 0$ changes the optimal β_i to β_i/α (that is, the product $\beta_i X_i$ remains unchanged). For ridge regression, coefficients do not scale this way due to the penalty term. Therefore (similarly to e.g. k-nearest neighbours) it is customary to first scale all predictors to have mean 0 and standard deviation 1.

Selection of λ For each fixed value of λ, ridge regression has a closed form solution to find the best coefficient vector β_λ. We can use cross-validation to find the best value of λ: for a series of values we can use cross-validation to find the value minimising the test error.

3.2 Lasso regression

Formally, *lasso (least absolute shrinkage and selection operator)* regression is quite similar: it replaces the $\sum_i \beta_i^2$ term by $\sum_i |\beta_i|$ (i.e. use ℓ_1 norm instead of ℓ_2 norm.) That is, for a fixed λ, we are looking for the values of β minimising

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|,$$

or equivalently, for an appropriate value of λ, we wish to solve

$$\begin{aligned} \min \ & \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 \\ \text{s.t.} & \sum_{j=1}^{p} |\beta_j| \leq t. \end{aligned}$$

The main difference between ridge and lasso is that the ridge objective forces the β_i coefficients to be small; but there will be typically many nonzero values. In contrast, the ℓ_1-regularisation in lasso will return a *sparse solution*: it forces some of the coefficients to become exactly zero. Hence, lasso will have an effect similar to *subset selection*. But instead of searching the space of subsets, the optimal solution to the program will implicitly do the subset selection, by returning predictors with nonzero coefficients.

Scaling and parameter selection for lasso can be done similarly to ridge regression.

3.3 Ridge vs Lasso

The different behaviour between ridge and lasso is due to the difference between the geometry of the ℓ_2-ball $\{\beta_i : \sum_{j=1}^{p} \beta_j^2 \leq t\}$ and the ℓ_1-ball $\{\beta_i : \sum_{j=1}^{p} |\beta_j| \leq t\}$, see **ISLR**, Fig 6.7. The sharp corners of the ℓ_1-ball will yield sparse support solutions.

In terms of interpretability, lasso is clearly better as it explains the target as a function of a few predictors. Lasso is also efficient at filtering out noise: if some predictors are independent from the target, they are likely to get a coefficient zero. On the other hand, if there is a genuine dependence on many predictors, then lasso can return worse predictions by forcing coefficients to zero.

Another difference is that whereas ridge regression has a closed-form solution, lasso does not admit one. Finding an (approximately) optimal solution can be done via convex optimisation methods.

4 References

- Cross-validation and boostrap: **ISLR** Chapter 5, **ESL** Sec 7.10
- Subset selection: **ISLR** Sec. 6.1
- Ridge and lasso: **ISLR** Sec. 6.2.

MA429 Algorithmic Techniques for Data Mining
Lecture 4: Decision trees and ensemble learning methods

László Végh
L.Vegh@lse.ac.uk.

1 Decision trees

Decision trees are simple nonparametric methods that can be used both for classification and regression. The high-level goal is to partition the space of the predictor variables into disjoint regions $R_1, R_2, \ldots, R_J$. For each region, the prediction will be a single value: in case of classification, the majority class of the test instances in this region; for regression, the mean of their target values. The partitioning is represented by a tree, where $R_1, R_2, \ldots, R_J$ are the root nodes.

1.1 Basic regression tree construction

The partitioning is done via a recursive splitting procedure. We first describe it for the case when all predictors as well as all the target are continuous variables. By a *region* we mean the restriction of the parameter space where each predictor is restricted to a certain interval. For each region R we assign a value $\omega(R)$ that measures the "quality" of R. We prefer regions where the target values are concentrated around the mean. We let μ_R denote the mean of the target values of the training instances in R, and we define $\omega(R)$ as the sum of squared errors in R:

$$\omega(R) = \sum_{(x_i,y_i)\in R} (y_i - \mu_R)^2$$

Initially, we put the entire parameter space as a single region in the root node of the tree. At every subsequent iteration, we select a leaf node in the current tree, and find the best split into two regions based on a predictor value.

Namely, let R be the current region, X_j one of the predictors, and c a value in the range of X_j on R. Let $R_-(j,c) = \{X \in R : X_j \leq c\}$ and $R_+(j,c) = \{X \in R : X_j > c\}$ denote the sets of training instances in R where the value of X_j is smaller or larger than c, respectively. Then, we select the attribute X_j and cutoff point c that minimises

$$\omega(R_-(j,c)) + \omega(R_+(j,c)).$$

Then, we create two leaves under the current node with the regions $R_-(j,c)$ and $R_+(j,c)$. We continue until every leaf in the current tree contains either very few training instances, or has a very low $\omega(R)$ value. Note that for each j, there are only a limited number of relevant c values (the value range of X_j).

Assume now we still have a regression problem but some of the predictors are categorical. If X_j is a categorical predictor with t different values, then we can create t possible binary splits for X_j, selecting one of the values as R_- and all others as R_+. Again, we consider the split that minimises $\omega(R_-)+\omega(R_+)$. (Alternatively, we could allow a non-binary split with t descendants of the current node.)

Missing values The tree construction method can also be applied to datasets where predictors have some missing values. Assume we are splitting according to $X_j \leq c$, and a certain training instance X has missing X_j value. Let q be the proportion of training instances with known $X_j \leq c$ values. Then, we split X into two fractional instances with weight q for $R_-(j,c)$ and weight $1-q$ for $R_+(j,c)$. When computing $\omega(R)$, for R containing fractional instances, we compute μ_R as a weighted mean, and also take a weighted combination of the squared error terms $(y_j - \mu_R)^2$.

When using such a tree to evaluate an instance with missing predictor values, whenever we reach a branching corresponding to a missing value, we send fractional copies of the instance on the two branches, proportionally splitting the weights according to the number of training instances in the two branches. Thus, fractional copies of the instance may reach multiple leaves; the answer returned will be the corresponding probability distribution.

1.2 Classification trees

Consider now a classification problem with k possible class values $y_1, y_2, \ldots, y_k$. There are multiple possible measures for quality $\omega(R)$. Let p_i denote the proportion of training instances in the different classes that fall in class y_i ($\sum_{i=1}^k p_k = 1$). The *Gini index* is defined as

$$G(R) = \sum_{i=1}^{k} p_i(1-p_i).$$

This takes a small value when all p_i's are either 0 or 1. The other measure is the *entropy* of R,

$$H(R) = -\sum_{i=1}^{k} p_i \log_2 p_i.$$

Note that this is a positive value. Entropy is a basic concept in information theory. It can be illustrated via the 20 questions game: assume we pick one of k words from the probability distribution $(p_1, p_2, \ldots, p_k)$. The entropy corresponds to the expected number of questions needed to guess this word? For example, the distribution $(0.5, 0.5)$ has entropy 1.

We let $n(R)$ denote the number of instances in the regions R. We can use either $\omega(R) = n(R) \cdot G(R)$ or $\omega(R) = n(R) \cdot H(R)$, that is, the Gini index or entropy weighted by the number of instances in the region, and apply the above recursive binary splitting procedure. These two measures usually lead to similar outcomes. For the entropy measure, one can intuitively think of tree construction as finding the best scheme to play 20 questions to identify the class value.

1.3 Pruning

The tree construction procedure has a very high variance: the leaves are tailored to small subsets of the training instances. To create more robust trees, we could stop the tree construction earlier on, once the $\omega(R)$ value of every leaf drops below a relatively high threshold. However, in this case we might miss opportunities of splitting the dataset into more homogenous parts later on. Instead, the common strategy is first grow the full tree, and then cut it back appropriately. This is called the *pruning procedure*.

Let T_0 be the full tree. A *subtree* of T is a tree obtained by designating a subset L of the nodes as leaves, such that *(a)* no node in L is the ancestor of another one, and *(b)* every leaf of the original tree has an ancestor in L. Given these properties, we obtain a smaller tree if we remove all nodes and edges

below the nodes in L. We could use cross-validation to estimate the error rate of every subtree of T_0 and pick the best one. However, this is not computationally feasible as the number of subtrees can be exponentially large.

Instead, the following regularisation idea can be used. For a subtree $T \subseteq T_0$, we let $\mathcal{R}(T)$ denote the set of regions corresponding to the leaves. For a parameter $\alpha > 0$, we look for the subtree $T \subseteq T_0$ minimising

$$\sum_{R \in \mathcal{R}(T)} \omega(R) + \alpha |T|.$$

For $\alpha = 0$, the best choice is $T = T_0$, whereas for very large α the best choice will be the trivial tree (just a single root node). As we gradually increase α, we obtain a nested sequence of subtrees that can be computed (see **ESL** Sec 9.2.2 and references therein).

We use k-fold cross-validation to find the best value of α. Namely, we repeat the tree building process and construct the nested tree family for the different values of α for every fold. We evaluate each tree in the nested family on the held out test set. For each value of α, we take the average error rate in the k folds, and select the value α^* that minimises this error. Then, we build the tree on the entire training data, and prune it according to α^*. This will be our final output.

1.4 Pros and cons for trees

One appealing feature of trees is interpretability: decisions using trees can be very easily explained, and exhibit similarities with human decision making. They can naturally work with any combination of categorical and continuous variables. In the second part of the lecture, we will see how more complex models can be constructed from trees. These will have better accuracy, however, they will loose out on interpretability.

2 Ensemble learning methods

Ensemble learning is a collection of methods of aggregating the output of multiple learning models into a single one. One can intuitively understand it as making a decision by aggregating the views of different experts. Here, experts correspond to the different models (that could be a collection of decision trees); the various ensemble learning methods use different approaches to "invite" experts and aggregate their decisions to a single decision.

2.1 Bagging

Bagging (bootstrap aggregating) is based on bootstrap sampling. We consider a certain learning method; the example will be via decision trees, but the technique is equally applicable for any classification or regression method.

From the dataset D with size $|D| = n$ (this will be the entire dataset, no need to split it into Tr and Te; more about this later) we select T samples of size n with repetition (bootstrap sampling). We train a model $\hat{f}^t$ in the t-th sample D_t; in the example, a decision tree. Then, we output the model

$$\hat{f}(x) = \frac{1}{T} \sum_{t=1}^{T} \hat{f}^t(x)$$

for a regression problem, and the probability distribution of the estimates $\hat{f}^t(x)$ for a classification problem.

The purpose of bagging is *variance reduction*. Performing multiple experiments and averaging the output reduces the variance, assuming that the models $\hat{f}^t$ are not too highly correlated. At the same time, averaging does not increase the bias: hence, the overall expected prediction error will decrease.

In case of decision trees, bagging uses *unpruned* trees. Pruning is an alternative way of reducing variance; but after pruning, the trees will be more similar (and thus more correlated). Hence, unpruned trees work better for the purpose of variance reduction; at the same time, it is also computationally more efficient to work with them.

Estimating error in bagging Another appealing feature of bagging is that we can easily integrate error estimation with the model building without much increase in the running time. Consider a data instance $(x, y) \in D$. Recall that this instance is not included in each bootstrap sample with probability roughly 36.8%. Hence, among all sets D_i, approximately $0.368T$ will not contain our data instance. Let the total number of such sets by d, and let us now use the prediction

$$\hat{y} = \frac{1}{d} \sum_{t:(x,y)\notin D_t} \hat{f}^t(x)$$

for testing for regression problems, or the probability distribution of these for a classification problem. Then, we estimate the error for (x, y) by considering $(y - \hat{y})^2$ for regression problems and an appropriate measure for classification (e.g. quadratic or informational loss). The overall error estimate is given by averaging these for every data instance.

2.2 Random forests

Random forest is a very efficient variant of bagging for decision trees. We obtain a substantial variance reduction in bagging if the tree models $\hat{f}^t$ have low correlation. To further stipulate this, for each split in the tree we only use a *random subset of predictors*. Let $P = \{1, 2, \ldots, p\}$ be the subset of all predictors. As in bagging, the t-tree is built on a bootstrap instance subset D_t. During the tree building process, at each split we randomly sample a subset $P' \subseteq P$ with $|P'| = m$ for some fixed m, and only consider the predictors in P' eligible for the split. We sample a new P' at each split.[1] The typical choice is $m \approx \sqrt{p}$.

The motivation is as follows. Assume there is a predictor k that has higher correlation with the target than all others. Then, we can expect each tree model starting by splitting on predictor k, resulting in a higher correlation between trees. On the other hand, by selecting a random subset, many splits will not have access to k; but there could be some other predictors that even though are not as highly correlated with the target as k, but together could result in a better tree. Random trees can be particularly useful if there are several highly correlated predictors.

2.3 Boosting

In bagging, all models $\hat{f}^t$ were obtained independently from each other and play equal role (we can think of them as experts of the same authority). Computationally, we can parallelise the training of the T models. In contrast, boosting methods build the models sequentially, by taking the performance of the previous models into account. Intuitively, we can think of inviting new experts that have expertise complementary to the previous ones.

[1] The first version of the notes incorrectly suggested that P' is the same at each split of the t-th tree.

A standard variant of boosting for regression problems is described in **ISLR** Sec 8.2.3. Instead, we now described the *AdaBoost* algorithm, an efficient variant for classification problems.

Weighted datasets AdaBoost works with a dataset Tr where the training instances have weights: the i-th instance has weight w_i. A bootstrap sample can be seen as a weighted dataset where the weight of an instance i is the number of times it has been selected in the sample. However, the weights can also be fractional values: an example was given in the first part of this lecture for decision trees.

There are two approaches for learning with weighted datasets. The first one is to incorporate the weights directly into the method. In many cases this can be done straigthforwardly. For example, in ordinary linear regression, we can use the weighted SSE

$$\sum_{i=1}^{n} w_i \left(y_i - \beta_0 - \sum_{j=1}^{n} \beta_j x_{ij} \right)^2 .$$

Similarly, for decision trees we can calculate the measures $\omega(R)$ by weighting the instances with the w_i's.

If there is no obvious way of extending the method with weights, we can use the second approach. We sample n instances from the dataset Tr with replacement with the probability of sampling instance i being proportional to w_i. Note that for all $w_i = 1$, this is the same as bootstrap sampling.

The AdaBoost algorithm Assume we have a classification problem with k classes $y^1, y^2, \ldots, y^k$, and a training dataset Tr of n instances, $(x_1, y_1), \ldots, (x_n, y_n)$. The algorithm proceeds in iterations $t = 1, 2, \ldots, T$ for some value of T. We maintain a weighting w_i of the data instances, initialised as $w_i = 1$ for every instance. In every iteration, we will use the same classification method (say, decisions trees), but on the weighted dataset with the weights changing after every iteration. The high level idea is that we increase the weight (relative importance) of all misclassified instances after every iteration. This stipulates that the next round will "pay more attention" to these instances (we recruit new experts on the areas of weakness).

In the t-th iteration, we build the function $\hat{f}^t$ using the method with the current weights w_i. We compute the error err_t on the (weighted) training dataset, namely, the sum of the weights of all misclassified instances divided by sum of all weights. If $\text{err}_t = 0$ (i.e. the method is perfectly accurate on the training dataset) or if $\text{err}_t \geq 0.5$, then we terminate. Otherwise, for every correctly classified instance, we leave the weight as is, while for every incorrectly classified instance, we increase the weight by a factor $(1 - \text{err}_t)/\text{err}_t$.

At the end of the process, we return the combination of the classifiers where the output of the t-th classifier is weighted by $\log\left(\frac{1-\text{err}_t}{\text{err}_t}\right)$. By this, we mean that for a test instance, each f^t returns a probability distribution over the k classes; we combine these probabilities in proportions of the associated multipliers. You can find a more formal description of the algorithm below.

Note that the stopping criteria for $\text{err}_t = 0$ or $\text{err}_t > 0.5$ guarantees that *(a)* the weights of the misclassified instances are increasing, and *(b)* every $\hat{f}^t$ in the final output has a positive coefficient.

Boosting can dramatically increase the performance of any *weak classifier*, i.e. a classifier that does at least a little better than random guessing (has error rate say, strictly less than 0.49). These can be quite simplistic methods. In the context of decision trees, we can use *"decision stumps"*: trees with a single root and two leafs (branching only on a single predictor).

In bagging or random forests, increasing the number T of models can only improve the accuracy. However, in boosting, it may lead to overfitting: we could end up perfectly approximating the training

Algorithm 1 AdaBoost

1: Set $w_i \leftarrow 1$ for $i = 1, 2 \ldots, n$.
2: **for** $t = 1, 2, \ldots, M$ **do**
3: Apply learning method on weighted dataset to obtain model $\hat{f}^t$.
4: $\text{err}_t \leftarrow \frac{\sum_{i:\hat{f}^t(x_i) \neq y_i} w_i}{\sum_{i=1}^n w_i}$.
5: **if** $0 < \text{err}_t < 0.5$ **then**
6: $w_i \leftarrow \begin{cases} w_i & \text{if } \hat{f}^t(x_i) = y_i \\ w_i \cdot \frac{1-\text{err}_t}{\text{err}_t} & \text{if } \hat{f}^t(x_i) \neq y_i. \end{cases}$
7: **else terminate**.
8: **return** the function $\hat{f}(x) = \sum_{t=1}^T \log\left(\frac{1-\text{err}_t}{\text{err}_t}\right) \hat{f}^t$.

dataset. To avoid this, we can use cross-validation to select the right number.

3 References

- Decision trees: **ISLR** Chapter 8.1
- Bagging, random forest: **ISLR** Chapter 8.2
- AdaBoost: **ESL**

MA429 Algorithmic Techniques for Data Mining
Lecture 5: Support vector machines

László Végh
L.Vegh@lse.ac.uk.

1 Review: logistic regression

We start by briefly reviewing multiple logistic regression. We have a classification problem with p predictors $X_1, X_2, \ldots, X_p$, and a binary target Y that can take values $Y = 1$ or $Y = -1$. The output of the logistic classifier is a vector of $p + 1$ parameters $\beta = (\beta_0, \beta_1, \ldots, \beta_p)$. For a predictor vector X, we return $p(X) = \Pr(Y = 1|X)$ as the probability of $Y = 1$ (and $1 - p(X)$ for $Y = -1$). This is given as

$$p(X) = p_\beta(X) = \frac{e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p}}{1 + e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p}}.$$

For any choice of β and for any X, this will take a value between 0 and 1.

Choice of β Given a training dataset Tr, logistic regression finds β that best fits the training samples according to the *maximum likelihood estimate*. If Tr comprises instances (x^i, y^i) for $i = 1, 2, \ldots, n$, then the likelihood function is defined as

$$\ell(\beta) = \prod_{i:y^i=1} p_\beta(x^i) \cdot \prod_{i:y^i=-1} (1 - p_\beta(x^i)).$$

This is the probability predicted for the actual set of class labels in the training set from the predictors, using the parameter vector β.

We select the vector β that maximises $\ell(\beta)$. An approximately optimal vector can be found via an iterative algorithm. We do not discuss it here; you can find some details in **ESL** Sec 4.4.1.

Let us also mention that the principle of maximum likelihood estimation is a very general approach also for other methods; in particular, the least squares estimate for linear regression is a maximum likelihood estimate.

Decision boundary for linear regression In order to turn the probability $p(X)$ into a binary decision, we need to set a threshold τ and decide for $Y = 1$ if $p(X) \geq \tau$ and $Y = -1$ otherwise. By default we use the threshold $\tau = 0.5$. The inequality $p(X) \geq 0.5$ can be equivalently written as

$$e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p} \geq 0.5 + 0.5 e^{\beta_0+\beta_1 X_1+\ldots+\beta_p X_p},$$

that in turn is equivalent to

$$\beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p \geq 0. \tag{1}$$

This is a linear inequality corresponding to a *linear hyperplane*. Thus, the binary output of logistic regression corresponds to a linear separation of the feature space: we return 1 on one side and -1 on the other side.

2 Support vector classifiers

Support vector classifiers also find a linear separator: the output will be hyperplane of the form (1). In contrast to logistic regression, we only focus on the binary output and not on a probabilistic output. We first consider the special case where the training dataset is *linearly separable*: there exists a hyperplane such that all points in Tr with $y^i = 1$ fall on one side, and all points with $y^i = -1$ on the other.

2.1 Maximal margin hyperplane

Given a dataset Tr, the question whether there exists a perfect separation of the two classes in the form (1) corresponds to a *linear program (LP)*: the variables are $\beta_0, \beta_1, \ldots, \beta_p$, and every data point (x^i, y^i) provides a linear constraint: if $y^i = 1$ then the constraint is

$$\beta_0 + \beta_1 x_1^i + \ldots + \beta_p x_p^i > 0,$$

and if $y^i = -1$, then the constraint is

$$\beta_0 + \beta_1 x_1^i + \ldots + \beta_p x_p^i < 0.$$

More concisely, we can write

$$y^i(\beta_0 + \beta_1 x_1^i + \ldots + \beta_p x_p^i) > 0$$

for both cases. We impose strict inequalities to avoid ambiguity on the margin. This LP is feasible if and only if the dataset is perfectly separable. If it is feasible, it has infinitely many different solutions. The *maximal margin classifier* selects a separation where the margin (the distance from the hyperplane on both sides) is as large as possible. This can be obtained by the following nonlinear optimisation problem:

$$\begin{aligned} &\max \ M \\ &\text{subject to} \sum_{j=1}^{t} \beta_j^2 = 1, \\ &y^i \left(\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i \right) \geq M \quad \forall i = 1, 2, \ldots, n. \end{aligned} \tag{2}$$

The normalisation ensures that for each data point i, the expression on the left hand side measures the distance from the hyperplane. We now turn this into a convex quadratic optimisation problem; the transformation amounts to dividing the variables β_i by M. This second form is

$$\begin{aligned} &\min \ \sum_{j=1}^{t} \beta_j^2 \\ &y^i \left(\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i \right) \geq 1 \quad \forall i = 1, 2, \ldots, n. \end{aligned} \tag{3}$$

An optimal solution can be found using standard methods in convex optimisation; we do not discuss them here.

2.2 Support vector classifiers

In case the dataset is not linearly separable, the above programs are infeasible and thus the maximal margin classifier is not applicable. Even in the case a perfect separation exist, it can be very sensitive to noise: imagine a separation with a wide margin, and then adding a new data point close to the separating hyperplane (even if on the correct side). Then, the new optimal separation could be vastly different from the old one.

Support vector classifiers, or *soft margin classifiers* address these issues by incorporating a certain level of error tolerance. In the form (2), we allow an error term $\varepsilon_i \geq 0$ for every data instance i, and relax the right hand side to $M(1-\varepsilon_i)$. Namely, we change the program to

$$\begin{aligned} &\max \ M \\ &\text{subject to} \sum_{j=1}^{t} \beta_j^2 = 1, \\ y^i &\left(\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i \right) \geq M(1-\varepsilon_i) \quad \forall i = 1, 2, \ldots, n, \\ &\sum_{i=1}^{n} \varepsilon_i \leq C, \\ &\varepsilon \geq 0. \end{aligned} \tag{4}$$

Thus, we are looking for a hyperplane with ideally no data points within distance M on both sides. However, we allow violation of this constraint: we have a budget C for the total violation. If $0 < \varepsilon_i < 1$, then the data point (x^i, y^i) is still on the correct side of the separating hyperplane, but closer than the desired margin. If $\varepsilon_i > 1$, then the point is on the incorrect side.

The budget C is a parameter controls the bias-variance tradeoff. The choice $C = 0$ does not allow any errors: we obtain the strict maximal margin classifier. For a very high choice of C, we can afford arbitrary violations of the separation constraints and the optimisation problem becomes meaningless. Generally, a lower C corresponds to a strict margin and thus a separating hyperplane that highly fits the training data: therefore, the variance will be high. On the other hand, a higher C produces more robust decision boundaries, and thus we get a lower variance, but possibly higher bias. We can use cross-validation to tune the parameter C.

Support vectors A distinctive feature of support vector machines is that as long as a point is at distance more than M from the margin on the correct size, the distance does not matter. This is in contrast with logistic regression, where points further from the margin will be associated with higher probabilities. Given an optimal solution to the support vector classifier, we can arbitrarily add or remove points on the correct side at distance more than M: the optimal solution remains unchanged.

The only "critical" instances are those within the margin or on the wrong side: these are called the *support vectors*: these are the observations that will contribute to the separating hyperplane.

Equivalent forms Let us also give a second, equivalent form of (4). Using a similar trick as for maximal margin classifiers as well as Lagrangian duality, we can obtain the equivalent system for some $\lambda > 0$:

$$\begin{gathered} \min \sum_{i=1}^{n} \varepsilon_i + \lambda \sum_{j=1}^{t} \beta_j^2 \\ y^i \left(\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i \right) \geq 1 - \varepsilon_i \quad \forall i = 1, 2, \ldots, n, \\ \varepsilon_i \geq 0. \end{gathered} \tag{5}$$

To each budget C in (2) there will be a value λ giving the same optimal solution: a larger budget C corresponds to a larger λ.

Note that in any optimal solution, whenever $\varepsilon_i > 0$ then the inequality corresponding to (x^i, y^i) must be at equality. Thus, every optimal solution must satisfy

$$\varepsilon_i = \max \left\{ 0, 1 - y^i \left(\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i \right) \right\}.$$

We can now further transform (5) to an unconstrained form:

$$\min_{\beta} \sum_{i=1}^{n} \max \left\{ 0, 1 - y^i \left(\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i \right) \right\} + \lambda \sum_{j=1}^{t} \beta_j^2. \tag{6}$$

This form is analogous to the ridge regularisation: we have a penalty term for each training instance, and wish to minimise a tradeoff between the total penalty and a quadratic regulariser; the tradeoff is controlled by the parameter λ.

2.3 Classification with more than two classes

Both logistic regression and the support vector classifier were described for a binary classification problem. These methods (as well as all other binary classifiers) can also be used for more than two classes. Consider a classification problem with $k > 2$ classes. The following are two common extension schemes.

One-versus-all classification We run the binary classification algorithm k times for the test observation; in the i-th run, we use class i as $+1$ and all other jointly as -1. In the best case scenario, we obtain only a single $+1$ classification answer, and return the corresponding class. We may end up with more than one $+1$ or none. In these cases, we return the class with the highest associated probability in case of a probabilistic classifier. For the support vector classifier, we do not obtain probabilities. Instead, we select the class with the highest $\beta_0 + \sum_{j=1}^{t} \beta_j x_j^i$ value as this corresponds to the largest margin.

One-versus-one classification Alternatively, we can run a "tournament" of classifications. That is, for each pair of classes i and j, we run the binary classifier by restricting the training dataset only to instances in classes i and j: this gives altogether $\binom{k}{2}$ classifiers. For a given test instance, and for each class i, we count how many of the $k-1$ pairwise classifiers involving class i decided for class i. We then select the class that has received the highest number of points. In case of a draw, we also look at the classification probabilities.

Note that the number of binary classifiers in this scheme is much larger, $\binom{k}{2}$ instead of k. At the same time, each classifier uses a smaller restricted dataset, whereas in the one-versus-all scheme, we always use the entire dataset. Therefore, the running time of the two methods could be comparable.

3 Nonlinear separation

Whereas the soft margin classifier is applicable for non-linearly separable problems, it still looks for a linear boundary between the two classes. We now consider two general approaches to extend linear classifiers to genuinely nonlinear settings.

3.1 Higher dimensional embedding

A universally applicable approach would be to extend the variable space by including higher dimensional polynomial terms. For example, if we have two predictors (X_1, X_2), then we can embed the feature space into the 5 dimensional space $(X_1, X_2, X_1X_2, X_1^2, X_2^2)$. A nonlinear separation such as $X_1^2 - X_2^2 \leq 9$ can be interpreted as a linear separation in this higher dimensional space. We could add all monomials up to degree d formed by the variables, for some $d \geq 2$. E.g., for two variables the above was the example for $d = 2$; for $d = 3$ this would further include $X_1^3, X_2^3, X_1^2X_2$, and $X_1X_2^2$. In general, let $h(X)$ denote a higher dimensional embedding of the features.

In theory, it can be shown that for any decision boundary and any desired accuracy, one can select a high enough d, for which we can approximate the boundary to the desired accuracy. In practice, this is not really feasible: we might need to introduce an astronomic number of new variables. Besides the computational cost, this might increase the number of predictors p way above n, leading to overfitting and high variance.

3.2 Support vector machines

Support vector machines extend support vector classifiers to the nonlinear setting, using *kernels*. This approach mimics a higher dimensional embedding, taking advantage of its flexibility but without much of the drawbacks.

The idea of kernels is to replace the standard *scalar product* in the support vector classifier by a kernel function. Let us first discuss the role of the scalar product. In the linear setting, we use the standard inner product:

$$\langle x, y \rangle = x^\top y = \sum_{i=1}^{p} x_i y_i$$

Using Lagrangian duality, one can show that the optimal separating hyperplane $f(x) = \beta_0 + \sum_{j=1}^{t} \beta_j x_j$ will be always obtained in the following form for some parameters α:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, x^i \rangle. \tag{7}$$

Moreover, we will have nonzero α_i's only for the support vectors, and a coefficient 0 for all others.

More generally, we can replace the scalar product $\langle x, y \rangle$ by a kernel function $K(x, y)$ that corresponds to a scalar product in a higher dimensional embedding. That is, for an embedding h, we will have

$$K(x, y) = \langle h(x), h(y) \rangle$$

However, we do not have to compute the embedding explicitly $h(x)$, but work in the original feature space. As an example, take the polynomial kernel of degree d:

$$K(x, y) = \left(1 + \sum_{i=1}^{p} x_i y_i\right)^d.$$

This will correspond to the scalar product for an embedding h with monomials up to degree d, but we do not have to compute all these terms in order to compute the function value $K(x, y)$.

It turns out that both in the optimisation problems as well as in the Lagrangian dual describing the form of the optimal solutions (7), the feature vectors are only involved via the inner products. In *support vector machines*, we will find a separator in the form

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(x, x^i) \tag{8}$$

for a kernel function K, giving a concise representation of the higher dimensional embedding. We do not give the precise details of the optimisation problem here.

Besides the polinomial kernel function, another popular choice is the *radial kernel*, defined as

$$K(x, y) = \exp\left(-\gamma \sum_{i=1}^{p} (x_i - y_i)^2\right).$$

Thus, the kernel function depends on the Euclidean distance of x and y, and decays exponentially with the distance. That is, we get a *local* behaviour: distant data points will have little impact on the classification. Using this kernel function *without* regularisation (without allowing error terms, that is, $C = 0$ or $\lambda = \infty$), we get a decision boundary similar to the 1-nearest neighbours method. However, the results will change after adding regularisation.

4 References

- Logistic regression: ST447 notes, **ISLR** Chapter 4.3, **ESL** Chapter 4.4
- Support vector machines: **ISLR** Chapter 9

MA429 Algorithmic Techniques for Data Mining
Lecture 6: Missing values and clustering

László Végh
L.Vegh@lse.ac.uk.

1 Missing values

Having missing values is a very common issue in data mining. Some methods (including Naïve Bayes or decision trees) are able to handle missing entries, but other methods (such as linear regression, k-nearest neighbours, or SVM) require values for all features of all points. There are several different approaches to deal with missing data. The first, most important step is to understand the reason why data might be missing.

1.1 Reasons why data is missing

- Values of a feature are *missing completely at random (MCAR)* if the probability that a particular value is missing is independent of the value of the feature as well as from the values of all other features. For example, some entries in the dataset have been removed by an error in the data input process data or by some database operation.

- Values of a feature can be *missing at random (MAR)* if the probability that a particular value is missing is independent of the value of the feature, but may depend on other features. For example, in a survey men might be more willing to tell their weight as women. As an other survey example, those who answered "no" to currently studying in higher education, will not answer questions relating to their degree of study.

- Values of a feature are *missing not at random (MNAR)* if the probability that a particular value of a feature is missing depends on the value. For example, on questions of sensitive personal information, members of minorities can be more likely to refuse the answer than members of more common groups.

It might not be obvious which type is exhibited in our dataset (and it can be different for the different features).

- To determine whether the missing entries in a feature X_i are due to MCAR, we can use statistical tests to decide whether there is a significant difference between the populations with and without the missing value. Namely, we create a new binary feature that has value 1 if X_i is missing and 0 if not; then we can run t-tests and χ^2-tests to see if this feature is related to the other variables.

- There are no automated methods to decide between MAR and MNAR, and usually requires domain expertise.

The simplest ways of dealing with missing data are:

- *Deleting data* items with one or more missing entries. This is the simplest solution and can be the best option if it only removes a small fraction of the data. However, it only gives an unbiased estimation in case of MCAR; for MAR or MNAR we introduce new biases. Instead of deleting data items, we can also consider removing some of the features that have excessive number of missing entries.
- For categorical data, we can treat missing values as a *separate category*. It can be by itself informative that a certain entry is missing. For example, if features correspond to medical test results, a missing value can mean that this test has not been carried out.

1.2 Data imputation

In case the above simple options are not available, we can replace the missing entries by estimations. This is called *data imputation*. The two main types are single imputation and multiple imputation.

- In *single imputation*, we fill in the missing data entries with some of the approaches discussed below, and then build our prediction model.
- *Multiple imputation* is applicable when the missing values are sampled according to some distribution. We create multiple samples for the missing values, apply the learning method on each of these samples and return the mean. Whenever computationally feasible, this should be the preferred method as it reduces the variance arising from the imputation.

We now describe some common methods for (single or multiple) imputation. Some of these (such as imputing mean) are deterministic and therefore make sense for single imputation only.

- Replacing missing values by the *mean* or *median* of the non-missing values.
- *Univariate sampling:* fit a distribution to the non-missing values, and sample according to this distribution. This can be a known distribution such as normal distribution, or a non-parametric estimation method such as kernel density estimation (not discussed in this course). Imputing mean, median, and univariate sampling are easy to implement but ignore correlations between the features.
- *Nearest neighbours estimation:* Predict the value of the missing entry from the values of the k nearest neighbours of the data point (where we only consider the non-missing entries to compute distances).
- *Multivariate sampling:* Can use more complex sampling methods by taking all features into account. We do not discuss this in more detail here.

2 Clustering

Clustering is a basic problem in *unsupervised learning*. We have a dataset D of n items with p feature vectors $(X_1, X_2, \ldots, X_p)$, and would like to identify "natural" groups of the data points. Data items groups together should be *similar* to each other but *differ* from those in other clusters. There is no target variable to be predicted; the purpose of clustering is to obtain structural information from the dataset. For example, the data items may represent customers of a company and clustering creates groups of customers with similar profiles.

2.1 K-means clustering

We start by K-means, a simple classical method. Here as well as for the other methods, we will assume that the features are all continuous. The number K of the different groups is part of the input of the algorithm. Of course, this may not be known in advance; we will need to find the best value of K.

Our goal is to find a partition of the items into K disjoint groups $C_1, C_2, \ldots, C_K$. That is, each item will belong to exactly one of the groups. The algorithm starts with an initial partition, and repeatedly reassigns items to groups.

For a group C_i, we let

$$m_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$$

denote the *cluster centroid*: that is, for each feature X_j, we take the mean value of all the items in C_i. The point m_i corresponds to these mean feature values (and is itself typically not one of the data points). We associate a *variance measure* $\omega(C_i)$ defined as

$$\omega(C_i) = \sum_{x \in C_i} \|x - m_i\|^2,$$

that is, the sum of the Euclidean distances between m_i and each of the data points in C_i. The smaller this quantity, the more concentrated the points are.

The purpose of K-means clustering is to find a partiton $C_1, C_2, \ldots, C_K$ that with a low value

$$\sum_{i=1}^{K} \omega(C_i)$$

Finding the partition where the exact minimum is taken is a computationally difficult task (NP-complete). The K-means algorithm is a heuristic approach to find a *local minimiser* of the objective. Let us now describe the algorithm.

Algorithm 1 K-MEANS

1: Pick K random centre points $m_1, m_2, \ldots, m_K$ in the feature space.
2: **repeat**
3: For each data point x, select the nearest m_j, and reassign x to C_j.
4: **for** $i = 1, \ldots, K$ **do** recompute m_j as the centroid of C_j.
5: **until** No set C_j changes
6: **return** groups $C_1, C_2, \ldots, C_K$.

We now show that this algorithm always terminates in a finite number of iterations. Progress can be measured in

$$\sum_{i=1}^{K} \sum_{x \in C_i} \|x - m_i\|^2, \tag{1}$$

- The quantity (1) strictly decreases whenever a data point is reassigned to a different cluster. Indeed, we move x from cluster C_i to cluster C_j if $\|x - m_i\|^2 > \|x - m_j\|^2$; these are the contributions of x to (1) before and after the reassignment, respectively.

- When we recompute the centroids m_j, the quantity (1) may not increase. This is because for every C_i, it is easy to see that $v = m_i$ is the minimiser of $\sum_{x \in C_i} \|x - v\|^2$.

Even though the method is finite, one cannot give a good (i.e. polynomial) guarantee on the running time. Still, in practice the method typically terminates in a few iterations. However, the output hugely depends on the initial choice of the centre points.

2.1.1 Preprocessing

Just as for k-nearest neighbours, using the "right" scaling of the features is off essence. It is customary to standardise the features so that all of them are on the same scale. We can also enhance the performance by removing redundant features. We will see next week that more radical transformations of the feature space can lead to further improvement.

Having *outliers* in the dataset (i.e. points with feature values very far from the mean) can highly distort the clusters; detecting and removing outliers is also advisable before clustering.

2.1.2 K-means++

Instead of choosing the initial centres ("seeds") independently at random, the variant K-means++ selects them to be well-spread. Namely, we select the first centre m_1 uniformly at random among the data points. Afterwards, for each data point x we compute $d(x)$ as the minimum distance of a centre already selected, and sample x as the next centre with probability proportional to $d^2(x)$.

We then run the standard K-means algorithm using this initial selection. This variant is more efficient both in theory as well as in practice.

2.1.3 Application: vector quantization

A perhaps surprising application of K-means clustering appears in image and signal compression. Please see **ESL** Sec 14.3.9 and the slides for details.

2.2 Hierarchical clustering

A main limitation of K-means clustering is that the number K of clusters must be fixed in advance. In *hierarchical clustering* we simultaneously compute a clustering for each value of K and we can decide for the best value afterwards. We obtain a tree representation of all instances, called a *dendrogram*.

A dendrogram is a rooted tree where the instances are assigned to the leaves and branching nodes are placed at different heights. These heights will correspond to the similarity between the groups split at this node.

We consider two main variants: *divisive (top-down)* and *agglomerative (bottom-up)* hierarchical clustering.

2.2.1 Divisive clustering: bisecting K-means

For $K = 2$ the 2-means algorithm is more stable than for higher values of K. The idea of the bisecting K-means algorithm is to repeatedly apply 2-means.

1. We start with all data points being in the same cluster, and divide it into 2 clusters by 2-means.
2. At each subsequent step, we select the current cluster C_i that has the largest $\omega(C_i)$ value, and split it into 2-clusters again by 2-means.

3. We continue until all $\omega(C_i)$ values fall below a given threshold, or if we reach the desired number of clusters K (if there is one).

2.2.2 Agglomerative clustering

In agglomerative clustering, every data point starts with its own separate cluster of size 1. In every iteration, we merge two clusters, and continue until all points have been merged into a single cluster. Thus, for a dataset with n points, there will be $n-1$ mergers altogether, creating a complete dendogram.

The difference between the various agglomerative methods is how we pick the next two clusters to be merged. We use a distance measure $d(x, y)$ between the data points. This can be the standard Euclidean distance, i.e. $d(x, y) = \|x - y\|^2$, but could also be different. Based on this distance, we will define a distance $D(C_i, C_j)$ between clusters, and we will merge C_i and C_j that minimise the distance D among the current clusters. We will always have $D(C_i, C_j) = d(x, y)$ for singleton clusters $C_i = \{x\}$ and $C_j = \{y\}$; but there are different possible choices for larger sets. The four main variants of agglomerative clustering correspond to such choices.

- **Complete linkage** We pick the smallest distance according to

$$D_{\max}(C_i, C_j) = \max_{x \in C_i, y \in C_j} d(x, y),$$

  i.e. we take the largest pairwise distance between points in C_i and C_j

- **Single linkage** Here, we look for the smallest pairwise distance, by using

$$D_{\min}(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y).$$

- **Average linkage**

$$D_{avg}(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{x \in C_i, y \in C_j} d(x, y).$$

- **Centroid distance** We compute the centroids m_i and m_j of C_i and C_j, respectively, and set

$$D_{centr}(C_i, C_j) = d(m_i, m_j).$$

Using different variants may lead to different cluster shapes, e.g. complete linkage stipulates more "globular" clusters, whereas single linkage will have a "chaining effect", creating long and thin clusters.

An advantage of hierarchical clustering is that we obtain the full taxonomy of the data points in the form of a dendogram; we also have flexibility by picking the distance measure $d(x, y)$. Euclidean may not always be the right choice, see **ISLR** Sec 10.3.2.

Among the disadvantages, we note that it can be very time consuming as we need to perform $n - 1$ merging steps, and distance measures have to be recomputed after the creation of a new cluster. Further, the clusters cannot be undone: once we decided for a merger, the instances in them remain together for all subsequent iterations.

2.3 Density based clustering

Density based clustering uses a different approach: it locates clusters of high density points separated from each other by regions of low density points. Such methods are capable of identifying highly irregular cluster shapes. We will discuss the *DBSCAN (Density-based spatial clustering of applications with noise)* algorithm.

The algorithm does not need to know the number of clusters K; instead, it requires two parameters: the neighbourhood radius ε, and the threshold T. We also use a distance metric $d(x, y)$ on the data set D. We say that two data points are *neighbours* if $d(x, y) \leq \varepsilon$. For a data point $x \in D$, we denote the number of its neighbours by

$$N(x) = |\{y \in D : d(x, y) \leq \varepsilon\}|.$$

(We have $x \in N(x)$, i.e. every point counts as its own neighbour.)

- **Core points**: $x \in D$ is a core point if $\text{Density}(x) \geq T$.
- **Border points**: $x \in D$ is a border point if it is not a core point, but there exists a core point y with $d(x, y) \leq \varepsilon$.
- **Noise points**: Every point that is neither core nor border point.

The DBSCAN algorithm proceeds as follows:

Algorithm 2 DBSCAN

```
Remove all noise points
repeat
    Pick an uncoloured core point x, and assign it a new colour c.
    while there exists a core point y with colour c having uncoloured neighbours do
        Colour every uncoloured neighbour of y with colour c.
until Every core point is coloured.
return colour groups as clusters.
```

The algorithm has multiple advantages compared to K-means.

- Robustness against noise. The algorithm starts by removing all noise points; these will not be included in any cluster. In K-means, every point must be added to a cluster and outliers may have a distorting effect.
- The number of clusters needs not be fixed.
- K-means aims for globular-shaped clusters, whereas DBSCAN is able to identify arbitrary irregular shapes.
- DBSCAN does not depend on the initial choice of centres.

However, it is only suitable for certain data types and may not be always applicable.

- DBSCAN may not be able to identify clusters of different density. We can search over various combinations of the parameters ε and T, but there might not be a choice that can successfully identify all clusters simultaneously.

We also note that the *curse of dimensionality* is a problem for all clustering methods discussed so far.

3 References

- Missing values: Gelman, Hill: Data Analysis Using Regression and Multilevel/Hierarchical Model, Chapter 25. `http://www.stat.columbia.edu/~gelman/arm/missing.pdf`
- K-means and hierarchical clustering: **ISLR** Chapter 10.3
- DBSCAN: Tan, Steinbach, Kumar: Introduction to Data Mining, Chapter 8. `http://www-users.cs.umn.edu/~kumar/dmbook/index.php`

MA429 Algorithmic Techniques for Data Mining
Lecture 7: Principal component analysis and spectral clustering

László Végh
L.Vegh@lse.ac.uk.

1 Principal component analysis

In *Lecture 3* we have considered subset selection and shrinkage methods to reduce the number of predictors in the dataset. The methods identify a subset of the original predictors. In *dimension reduction methods*, we create new, synthetic predictors from the original one. *Principal component analysis (PCA)* is an important such method. Given a dataset D with n data items and features $(X_1, X_2, \ldots, X_p)$, all continuous, we create new features $(Z_1, Z_2, \ldots, Z_q)$ for some $q < p$ in the form:

$$Z_i = w_{i1}X_1 + w_{i2}X_2 + \ldots + w_{ip}X_p, \quad i = 1, 2, \ldots, q.$$

That is, the Z_i's are linear combinations of the original X_i's. Thus, we represent the original dataset in a different coordinate system, and just keep $q < p$ coordinates. We aim to find the "natural" coordinate system for the dataset. Namely, we try to capture as much of the variance as possible in the first coordinate Z_1, then as much of the remaining variance as possible on Z_2, etc. We stop with Z_q once the representation $(Z_1, Z_2, \ldots, Z_q)$ already captures a substantial part of the total variance.

For PCA, an initial standardisation of the features is of essence: we assume that all X_i's have mean 0 and standard deviation 1. Let $X \in \mathrm{R}^{n\times p}$ denote the data matrix, i.e. the j-th column corresponds to X_j, and X_{ij} is the j-th feature value of the i-th data instance. Due to the to the mean 0 assumption, the empirical covariance of the j-th and k-th feature is

$$\mathrm{cov}(X_j, X_k) = \frac{1}{N}\sum_{i=1}^{n} X_{ij}X_{ik} = \frac{1}{N}X_j^\top X_k.$$

Due to the standard deviation 1 assumption, we have $\mathrm{cov}(X_j, X_j) = \frac{1}{N}\|X_j\|^2 = 1$, i.e. the squared norm of each column is N. We let $C = \frac{1}{N}X^\top X \in \mathbb{R}^{p\times p}$ denote the *covariance matrix*, where

$$C_{jk} = \frac{1}{N}X_j^\top X_k = \mathrm{cov}(X_j, X_k).$$

Recall the basic properties of this matrix: it is *symmetric* (i.e. $C_{jk} = C_{kj}$ for all $1 \le j \le k$), and *positive semidefinite* (i.e. $y^\top Cy \ge 0$ for any $y \in \mathbb{R}^p$).

Consider now a weight vector $w \in \mathbb{R}^p$ that gives a combination $Z = \sum_{j=1}^{p} w_j X_j = Xw$ of the columns. The empirical variance of this synthetic feature is

$$\mathrm{Var}(Z) = \frac{1}{N}\|Z\|^2 = \frac{1}{N}w^\top X^\top Xw = w^\top Cw.$$

The first *principal component* Z_1 will be obtained by maximising the variance $\mathrm{Var}(Z)$ under the normalisation $\|w\| = 1$. That is, we ask for

$$\max_{\|w\|=1} w^\top C w.$$

This is equivalent to projecting the data to a single direction that maximises the variance. Since the matrix C is symmetric, the maximiser corresponds to the largest eigenvalue λ_1, and the corresponding eigenvector $w_{(1)}$ with $\|w_{(1)}\| = 1$, i.e. $Cw_{(1)} = \lambda_1 w_{(1)}$, and $w_{(1)}^\top C w_{(1)} = \lambda_1$. We let w_{1j} denote the j-the coordinate of $w_{(1)}$. Hence, the first principal component will be

$$Z_1 = X w_{(1)} = \sum_{i=1}^{p} w_{1j} X_j,$$

with $\mathrm{Var}(Z_1) = \lambda_1$. To obtain the further principal components, we need to focus on the variance that is not captured by the first principal component. It can be shown that the best choice is to select $w_{(1)}, w_{(2)}, \ldots, w_{(p)}$ as pairwise orthogonal unit eigenvectors of C corresponding to the decreasing sequence of eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p$. Note that we take the eigenvalues with multiplicities and therefore there is exactly p of them. Hence, the t-th principal component will be $Z_t = X w_{(t)}$.

Due to the orthogonal choice of the eigenvectors, the principal components Z_j and Z_k will be *uncorrelated* if $j \neq k$, since

$$\mathrm{cov}(Z_j, Z_k) = \frac{1}{N} Z_j^\top Z_k = \frac{1}{N} w_{(j)}^\top X^\top X w_{(k)} = \frac{1}{N} w_{(j)}^\top C w_{(k)} = \frac{1}{N} \lambda_{(k)} w_{(j)}^\top w_{(k)} = 0.$$

Here, we first used that $w_{(k)}$ is an eigenvector of C, and then the orthogonality of $w_{(j)}$ and $w_{(k)}$. The total variance is

$$\sum_{j=1}^{p} \mathrm{Var}(Z_j) = \sum_{j=1}^{p} \lambda_j = \sum_{j=1}^{p} \mathrm{Var}(X_j).$$

The last equation follows by the well-known fact that the sum of the eigenvalues of C equals its trace, i.e. the sum of variances of the original variables.

Instead of the full list of p eigenvalues, we will only return the top $q < p$ ones that capture a given fraction (say, 85%) of the total variance. This may happen for a small number of eigenvalues already.

We emphasise that PCA is an *unsupervised* method: the principal components are selected from the features without taking the target variable (if any) into account. PCA plays an important role in data visualisation, and can also be used in conjunction with many supervised learning methods.

1.1 Data compression

PCA can lead to very efficient data compression. We can represent every data item v in the *eigenvector basis*, i.e.

$$v = \sum_{j=1}^{p} \mu_j Z_j$$

for some coefficients μ_j. We now truncate to the top $q < p$ eigenvectors, and approximate v by

$$v' = \sum_{j=1}^{q} \mu_j Z_j.$$

Instead of storing the original dataset, we just store the eigenvectors $Z_1, \ldots, Z_q$, and for each data item the q coefficients $(\mu_1, \ldots, \mu_q)$. See the example on *eigenfaces* on the slides.

1.2 Principal components regression

A simple and important application for supervised learning is *principal component regression (PCR)*: we use PCA to create q principal components, and build an ordinary linear regression model on them. If we set $q = p$, i.e., use all principal components, then we get an identical model as for ordinary least squares (see class exercise). As we increase q from 1 to p, then the bias decreases, but the variance increases; we can find the value of q representing the best tradeoff by cross-validation.

PCR can work particularly well in case of multicollinearity, namely, if the predictors are highly correlated. This makes least squares unstable and makes it difficult to interpret the coefficients. In PCR, we create synthetic predictors that are uncorrelated.

2 Spectral clustering

Spectral clustering has a high-level analogy to PCA: it uses a simple clustering method such as K-means, but on a transformed feature space instead of the original one. However, the transformation will be different from PCA. Instead of transforming the feature space, we view clustering as a graph optimisation problem. We consider the undirected graph where the nodes are the n data points in D, and there is an edge of weight s_{ij} representing the similarity between nodes i and j: the higher the better. The goal is to find a partition of the nodes into K sets such that the similarities between nodes in the same set are high, and similarities between different sets are high.

A common similarity measure is given by the Gaussian kernel. Namely, the similarity between data points x^i and x^j is defined as

$$s_{ij} = e^{-\frac{\|x_i - x_j\|}{c}}$$

for some $c > 0$. Note that $s_{ii} = 1$ for every i. The similarity exponentially decays as the distance grows. Sometimes, this is further modified by rounding the small values to 0.

For the i-th data point, we define

$$T_i = \sum_{j \neq i} s_{ij}$$

as the total similarity of the other data points to i. We define the *normalised Laplacian matrix* $L \in \mathbb{R}^{n \times n}$ as follows. We let

$$L_{ij} = \begin{cases} 1 & \text{if } i = j, \\ -s_{ij}/T_i & \text{if } i \neq j. \end{cases}$$

By the choice of T_i, we see that all rows sum up to exactly 0, that is, $L\mathbf{1}_n = 0$, where $\mathbf{1}_n$ is the n-dimensional all ones vector. In other words, 0 is an eigenvalue of L and $\mathbf{1}_n$ is a corresponding eigenvector.

It can be shown that 0 is in fact the smallest eigenvalue of L. We let

$$0 = \lambda_0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_{n-1}$$

denote the eigenvalues in non-decreasing order, and $v_0 = \mathbf{1}_n, v_1, \ldots, v_{n-1} \in \mathbb{R}^m$ the corresponding eigenvectors, that is,

$$Lv_i = \lambda_i v_i \quad \forall i = 0, 1, 2, \ldots, n-1.$$

We note that here we listed the eigenvalues with multiplicity: if for example 0 is an eigenvalue with multiplicity 3, then $\lambda_0 = \lambda_1 = \lambda_2 = 0$. The eigenvectors belonging to the same eigenvalue can be chosen

to be pairwise orthogonal. The eigenvectors should be normalised to have length 1. (We omitted this for v_0; the normalised version would be $\left(\frac{1}{\sqrt{n}}, \ldots, \frac{1}{\sqrt{n}}\right)$.)

For some $1 \leq r \leq n-1$, we construct the following representation of the dataset. Let v_{ij} denote the j-th component of the i-th eigenvector v_i. Then, we let the vector

$$z_j = (v_{1j}, v_{2j}, \ldots, v_{rj}) \quad \forall i = 1, 2, \ldots, n.$$

That is, the original p-dimensional feature space will be transformed to an r-dimensional one defined by the eigenvectors of the normalised Laplacian matrix.

Now we apply a simple clustering method, for example, K-means, to create K clusters from the feature vectors z_j; distance is now measured as the Euclidean distance between the z_j's.

In contrast to using K-means in the original dataset, spectral clustering is able to identify highly irregular cluster shapes; see examples on slides.

To understand why spectral clustering works, we first assume that the graph has m disconnected components. That is, the items can be partitioned into m sets $C_1, C_2, \ldots, C_m$ such that $s_{ij} = 0$ between the different sets. Then the eigenvalue 0 has multiplicity m. Besides $\mathbf{1}_n$, we also have an eigenvector for each $i = 1, 2, \ldots, m$ with

$$v_k^{(i)} = \begin{cases} 1 & \text{if } k \in C_i, \\ 0 & \text{if } k \notin C_i. \end{cases}$$

It is easy to see that all the $v^{(i)}$'s satisfy $Lv^{(i)} = 0$, that is, they are eigenvectors corresponding to 0. If we now use spectral clustering for $r = m - 1$, it will select $\lambda_1 = \lambda_2 = \ldots = \lambda_{m-1} = 0$ and $m - 1$ of the eigenvectors $v^{(i)}$ (or an equivalent linear representation). Assume e.g. it selected $v^{(1)}, v^{(2)}, \ldots, v^{(m-1)}$. Then, we will have $z_{ji} = 1$ if $j \in C_i$ and 0 otherwise. In particular, we will have $z_j = z_k$ whenever $j, k \in C_i$, and $z_j \neq z_k$ if j and k are in different groups.

The above illustration is only for the ideal case of perfect clusters, and the real instances always have some noise (i.e. links between clusters with positive similarity). Still, as long as the first r eigenvalues are small, we can think of them as corresponding to $r + 1$ well-separated clusters. This explains why using the eigenvector representation is natural for clustering. To find the best value of r, we can look for a gap between the eigenvalues so that all of them before the gap are close to 0 and all after the gap are significantly larger. However, this might not always be clear how to determine.

Random walk interpretation An other interpretation of spectral clustering comes from analysing the following random process. Assume we have a Markov chain where the states are the data points, and s_{ij}/T_i is the transition probability from state i to state j. We let M denote the transition matrix; note that $L = I_n - M$ where I_n is the n-dimensional identity matrix. Consider a random walk according to these transition probabilities. Then, spectral clustering identifies groups of states (data points) where the random walk gets "stuck": it enters or leaves the group with small probability.

3 Evaluating clusters

Evaluating clusters in notoriously more difficult than evaluating supervised learning methods. There are no universal criteria, and we need to find the right assessment approach on a case-by-case basis. We briefly review some basic approaches.

- *Supervised cluster evaluation*: In certain cases we also have a target variable that is not used for clustering. For example, we have class labels, and would like to see to what extent the groups identified by the clustering correspond to the classes. Evaluating clusters is somewhat similar to evaluating the quality of a region R in decision tree construction (*see lecture 4*). We consider the regions obtained by clustering, and would like them to have low Gini index or entropy measure. We do not discuss supervised clustering in more detail here.

- *Unsupervised cluster evaluation*: If no outside information is available, we need to assess the quality of the clusters using internal criteria; we review some approaches next.

It is already a nontrivial question to decide whether there is any clustering tendency in the dataset. For example, we could start with randomly generated data. The clustering methods will output some decomposition into clusters, and, by accident, these might capture some more concentrated sets of data points.

Visualising the similarity matrix A simple qualitative approach is the following. Let us reorder the dataset so that data points in the same C_ℓ belong to a consecutive interval. In the similarity matrix S where the entry s_{ij} represents the similarity between i and j, we expect a block-diagonal structure. For a good clustering, within each block corresponding to the same C_ℓ, we have high s_{ij} values, whereas outside these blocks we can see low values. We can represent the values by a heat-map (red for high and blue for low), and inspect whether there is an apparent block structure. See slides for illustration.

Cohesion and separation measures We have seen that K-means aims to minimise

$$\sum_{i=1}^{K} \omega(C_i) = \sum_{i=1}^{K} \sum_{x \in C_i} \|x - m_i\|^2.$$

This is a *cohesion measure*: it is small if items in the same cluster are close to each other, but it does not account for distances between different clusters. If we are free to change the value of K, then we get the optimal solution for $K = n$, i.e. if every data item forms a separate cluster.

Thus, cohesion measures can be complemented by *separation measures* that penalise different clusters being too close. We use a similarity measure $D(C_i, C_j)$ between clusters; this could be one of $D_{\max}, D_{\min}, D_{\text{avg}}, D_{\text{centr}}$ used in hierarchical clustering. Then, we can look at

$$\sum_{i=1}^{K} D(C_i, C_j),$$

and consider a clustering into K clusters better if it has a higher total separation. There are various possible combinations of separation and cohesion measures used to evaluate clusters.

Silhouette coefficent We can associate a value to each data item that measures both aspects of separation and cohesion. Given a clustering $C_1, C_2, \ldots, C_K$, and a data point $x_i \in C_\ell$, we let

$$a_i = \frac{1}{|C_\ell|} \sum_{x \in C_\ell} d(x_i, x)$$

denote the average distance of the points in the same cluster, and

$$b_i = \min_{j \neq \ell} \frac{1}{|C_j|} \sum_{x \in C_j} d(x_i, x)$$

denote the minimum average distance from points in another cluster. Then the *silhouette coefficient* is

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}.$$

This can take values between -1 and 1. A negative value indicates that it might be better to move the point to a different cluster. To obtain a single index, we can compute the mean or median of the silhouette coefficients.

4 References

- Principal component analysis: **ESL** Chapter 14.5.1.
- Spectral clustering: **ESL** Chapter 14.5.3.
- Cluster evaluation: Tan, Steinbach, Kumar: Introduction to Data Mining, Chapter 8. `http://www-users.cs.umn.edu/~kumar/dmbook/index.php`

MA429 Algorithmic Techniques for Data Mining
Lecture 8: Association rule mining

László Végh
L.Vegh@lse.ac.uk.

1 Frequent item sets and association rules

Association rule mining is a type of unsupervised learning. It is applicable to datasets where instances comprise a collection of items, for example, the set of items bought together by customers in a supermarket. An association rule would establish connections between different item sets: for two item sets A and B, $A \to B$ means that customers who bought all items in the set A have frequently purchased also all items in B.

Association rule mining has been used by supermarkets for using the information on designing store layouts and promotional offers. A very common, more recent application is online recommendation systems.

A *market basket dataset* for subsets of M items can be interpreted as data instances with M features, where X_i is a binary feature with value 1 if the i-th item is present in the collection and 0 otherwise. For N data instances, directly storing an $N \times M$ matrix is not a good way to represent such data. Typically, M can be very high and the data matrix is sparse, with few nonzero entries. (E.g. there could be $M = 5000$ possible items in the supermarket, with buyers on average purchasing just around 20.)

Thus, we prefer a *concise* representation, listing only the items contained in the basket in an appropriate data structure. By a *basket* we mean such a list of items corresponding to a data instance; the ordering of the items does not matter.

The first and most important step in association rule mining is to identify *frequent item sets*, i.e. sets of items that occur together often times. By the *support* of an item set A we mean the number of occurrences in the dataset, and denote this by $\mathrm{supp}(A)$. This will be an integer between 0 and N (where N is the total number of data items). An item set will be called *frequent*, if its support is above a specified *minimum support threshold*, denoted by minsupp; that is, $\mathrm{supp}(A) \geq \mathrm{minsupp}$. Sometimes, the support and the threshold are given as fractions of N, the number of baskets; it should be clear from the context whether it is an absolute number or a fraction.

An *association rule* is of the form $A \to B$ for two disjoint item sets A and B. The *confidence* of this rule is the proportion of the baskets containing all items in B among those that contain all items in A. That is,

$$\mathrm{conf}(A \to B) = \frac{\mathrm{supp}(A \cup B)}{\mathrm{supp}(A)}.$$

By the support of the rule $A \to B$ we mean $\mathrm{supp}(A \cup B)$.

In the standard association rule mining setup, we are given two thresholds: minsupp between 0 and N, and minconf between 0 and 1. The goal is to list all association rules $A \to B$ that satisfy

$$\mathrm{conf}(A \to B) \geq \mathrm{minconf}, \text{ and } \mathrm{supp}(A \cup B) \geq \mathrm{minsupp}.$$

Listing all such rules is a hard problem in general. Already the number of rules meeting the criteria can be extremely large: we need to set appropriate thresholds to get a manageable output.

We now focus on the task of listing these rules. A naïve approach would be to list all disjoint pairs of item sets A and B and check the support and confidence of these rules. Such a method would necessarily investigate an astronomical number of cases.

2 The Apriori Algorithm

The Apriori Algorithm is a more efficient way of enumerating and finding all association rules. We have the thresholds minsupp and minconf given in the input. The algorithm proceeds in two phases.

1. Identify all frequent item sets, i.e. all item sets S such that $\text{supp}(S) \geq \text{minsupp}$.
2. For every frequent item set S, find all partitions $S = A \cup B$ such that $\text{conf}(A \to B) \geq \text{minconf}$.

The more important and more time-consuming phase is the first one. It is based on the following **monotonicity property**:

Claim 2.1. *If an item set is frequent, then all its subsets are also frequent.*

Proof. Let A be a frequent item set. If $B \subseteq A$, then all baskets containing A will also contain B. Hence, $\text{supp}(B) \geq \text{supp}(A) \geq \text{minconf}$. □

We will monotonicity it in the following logically equivalent form: *if an item set is* **not** *frequent, then all item sets containing it are* **not** *frequent either.*

This property enables an important pruning step. The first phase of the Apriori algorithm proceeds finding frequent item sets by increasing order of the cardinality. For each $k = 1, 2, \ldots$, we let C_k be the set of *candidate item sets* of cardinality k, and $F_k \subseteq C_k$ the set of frequent item sets. Using the above rule, we form the set of candidates C_{k+1} for the next level from the sets that have all their k element subsets contained in F_k. See the slides for an example.

Algorithm 1 Apriori Algorithm Phase 1

$k \leftarrow 1$, $C_1 \leftarrow$ all singleton item sets.
repeat
    Scan over all transactions, and count how many times every set in C_k occurs, that is, determine $\text{supp}(S)$ for every $S \in C_k$.
    $F_k \leftarrow \{S \in C_k : \text{supp}(S) \geq \text{minsupp}\}$.
    $C_{k+1} \leftarrow \{S : |S| = k+1, \text{and } S \setminus \{v\} \subseteq F_k \ \forall v \in S\}$.
    $k \leftarrow k + 1$.
until $C_k = \emptyset$.
return $\cup_{i=1}^{k-1} F_k$.

2.1 Phase 2 of Apriori Algorithm

Once we have identified all frequent itemsets S with $\text{supp}(S) \geq \text{minsupp}$, for each set S we need to find all partitions $S = A \cup B$ such that $\text{conf}(A \to B) \geq \text{minconf}$. Since these sets S are typically small, here

we could even afford exhaustive search, i.e. checking all $2^{|S|} - 2$ possible splits. We can however improve on this by using an appropriate version of monotoncity.

Claim 2.2. *Let $A \cup B = A' \cup B'$ with $A \subset A'$, and consequently, $B \supset B'$. Then,* $\text{conf}(A \to B) \leq \text{conf}(A' \to B')$.

Proof. Let $S = A \cup B = A' \cup B'$. Recall that $\text{conf}(A \to B) = \text{supp}(S)/\text{supp}(A)$ and $\text{conf}(A' \to B') = \text{supp}(S)/\text{supp}(A')$. The numerators are the same, and for the denominators we have $\text{supp}(A) \geq \text{supp}(A')$ by monotonicity, implying the claim. □

Using this property, in Phase 2 we consider the candidates $A \to B$ in increasing order of the cardinality of B. We start with all candidates $S \setminus \{v\} \to \{v\}$ for $v \in S$. In the k-th round, we consider the pairs $A \to B$ with $|B| = k$, and only if for every $v \in B$, $A \cup \{v\} \to B \setminus \{v\}$ has met the minconf threshold.

Note that in Phase 2 we do not have to access the original dataset anymore: we only use the $\text{supp}(A)$ values computed in the first phase.

2.2 The FP-Growth algorithm

A limitation of the Apriori algorithm is that it must scan the entire dataset k times if k is the cardinality of the largest frequent itemset. This can be very time-consuming for a large dataset.

In contrast, the *FP-Growth (Frequent-Pattern Growth)* algorithm only scans the dataset once. It creates a data structure called *FP-tree* that represents the data set concisely. For each item set, we can extract its support from the tree. In practice FP-Growth can be much more efficient than the Apriori algorithm. We do not present the details here.

2.3 Apriori with flexible minsupp threshold

Choosing the right minsupp threshold can be difficult or in some cases impossible. There might be some interesting association rules among less frequent items. To detect this, we need to use a low minsupp; but then we will identify a vast number of less interesting frequent itemsets containing common items. Different support thresholds might be appropriate for different rules. For example, the rule {food processor} → {cooking pan} might be a relevant one but with support only 0.5%; in contrast, the less interesting {bread, cheese, milk} → {butter} could have support 50%.

This can be addressed by setting an individual threshold for each item: $\sigma(i)$ for item i. We will call an item set A frequent, if its support is above the smallest $\sigma(i)$ value among the items $i \in A$:

$$\text{supp}(A) \geq \min\{\sigma(i) : i \in A\}.$$

We have to be careful as this definition can violate the monotonicity property used in Apriori (and also in FP-Growth). E.g. assume $\sigma(1) = \sigma(2) = 5$, $\sigma(3) = 1$. If $\text{supp}(\{1, 2\}) = 4$, then the set $\{1, 2\}$ is not considered frequent. However, the larger set $\{1, 2, 3\}$ may be still frequent as long as $\text{supp}(\{1, 2, 3\}) \geq 1$.

Still, one can modify the Apriori algorithm to work with such thresholds. Even though monotonicity does not hold, it "almost" does. In Apriori, we included a set S in the candidate set C_{k+1} if all its k-element subsets were in F_k, i.e. frequent. We will now require that all except possibly one k-element subsets of S are frequent.

In more detail, consider a candidate set $S = \{s_1, s_2, \ldots, s_{k+1}\}$, and let us order its elements such that $\sigma(s_1) \leq \sigma(s_2) \leq \ldots \leq \sigma(s_{k+1})$.

- If $\sigma(s_1) = \sigma(s_2)$, then we see that

$$\min\{\sigma(i) : i \in S \setminus \{v\}\} = \min\{\sigma(i) : i \in S\} \quad \forall v \in S,$$

that is, S has the same threshold as all of its k-element subsets. Therefore, we require that all k-element subsets of S are in F_k in order to add S to the candidate set C_{k+1}.

- If $\sigma(s_1) < \sigma(s_2)$, then the threshold for $S \setminus \{s_1\}$ is larger than for S; thus, we do not require $S \setminus \{s_1\} \in F_k$. However, we still have

$$\min\{\sigma(i) : i \in S \setminus \{v\}\} = \min\{\sigma(i) : i \in S\} \quad \forall v = s_i, i > 1,$$

and therefore $S \setminus \{s_i\} \in F_k$ will be required for all $i > 1$.

3 Alternative measures for association rules

Confidence is a natural measure for rules, but it can be sometimes misleading. To see this, consider the following example: we have 100 transactions, out of which 90 include tea, 25 include coffee, 20 include both tea & coffee, and 5 include neither. Then the rule $\{\text{coffee}\} \to \{\text{tea}\}$ has confidence 0.8, thus, one might consider it as a strong rule. In fact, the occurrence of tea and coffee are *negatively correlated* (check!).

Thus, some alternative measures have also been considered. Let us use

$$P(A) = \frac{\text{supp}(A)}{N}$$

denote the fraction of baskets containing the item set A, that we can interpret as the probability of the occurrence of A. With this notation, the confidence can be written as

$$\text{conf}(A \to B) = \frac{P(A \cup B)}{P(A)} = P(B|A).$$

The *lift* is defined as

$$\text{lift}(A \to B) = \frac{P(A \cup B)}{P(A) \cdot P(B)},$$

and the *leverage* is

$$\text{lev}(A \to B) = P(A \cup B) - P(A) \cdot P(B).$$

Both these measure correlation between A and B. The range of lift is $[0, \infty)$, with independence corresponding to 1. The range of leverage is $[-0.25, 0.25]$, with independence corresponding to 0. On the negative side, both these measures are *symmetric*: $\text{lift}(A \to B) = \text{lift}(B \to A)$ and $\text{lev}(A \to B) = \text{lev}(B \to A)$. These are therefore not appropriate measures of implication.

The *conviction* measure combines aspects of confidence and correlation. It is defined as

$$\text{conv}(A \to B) = \frac{1 - P(B)}{1 - \text{conf}(A \to B)}.$$

This is an asymmetric measure. The denominator is the probability that we observe A but do not observe B, whereas the numerator is the same probability assuming A and B were independent. Hence, conviction measures how much more likely it would be that our rule failed if A and B were independent. The range is again $[0, \infty)$, with independence corresponding to 1.

If we want to identify rules with high lift, leverage, or conviction, the first phase of Apriori on finding frequent item sets remains unchanged. In the second stage we can use the chosen measure or a combination of multiple measures, in the simplest case, with exhaustive search.

Interpretation Once the relevant rules have been identified, they require careful interpretation using domain knowledge. In particular, one should *not* translate a high confidence, conviction, etc. to a causal relationship. The implication that in most cases when event A happens, the event B also happens, does not mean that B happens because of A; there could be e.g. some common explanation for the two events that are otherwise causally independent of each other.

4 References

- Apriori: **Weka** Sections 4.5, 6.3, **ESL** 14.2
- Association measures and FP-Growth: Tan, Steinbach, Kumar: Introduction to Data Mining, Chapter 5. `http://www-users.cs.umn.edu/~kumar/dmbook/index.php`

MA429 Algorithmic Techniques for Data Mining
Lecture 9: Neural networks

László Végh
L.Vegh@lse.ac.uk.

1 Basic concepts

Neural networks are a classical nonlinear learning technique mainly developed in the 1970-80s. They became again prominent over the last 10 years, in what is often times called the *deep learning revolution*. They are currently considered to be the most powerful machine learning technique, with very impressive successes across a range of domains. The aim of this lecture is to review the basic concepts and ideas of neural networks.

The Perceptron Algorithm Neural networks were inspired by the idea to build models of the brain in the 1950s. The basic building blocks of neural networks are *artificial neurons*. The simplest neural network, the *perceptron*, comprises as single neuron. This is a parametric classification model for data with p continuous features $(X_1, X_2, \ldots, X_p)$, and a binary target Y that takes values $+1$ and -1. The parameters are $p+1$ weights $(w_0, w_1, \ldots, w_p)$, that are the *inputs* of the neuron, and the output is

$$\hat{f}(X) = \begin{cases} 1 & \text{if } w_0 + \sum_{i=1}^p w_i X_i > 0, \\ -1 & \text{if } w_0 + \sum_{i=1}^p w_i X_i < 0 \end{cases} \tag{1}$$

Hence, perceptron is a *linear classifier*. The term w_0 corresponds to the bias. To simplify the formalism, we introduce an additional feature X_0 that always has value 1. Then, we can use $w^\top X = \sum_{i=0}^p w_i X_i$ to denote the expression ont he left hand side.

Rosenblatt's *Perceptron Algorithm* is one of the simplest examples of *reinforcement learning*:

- We start with the weights $w_i = 0$, $i = 0, 1, \ldots, p$. Since the classifier requires strict inequalities, this will give an incorrect prediction for all data points.
- We consider the data instances one by one, repeatedly.
- If an instance is correctly classified, we move on to the next one.
- If an instance (x, y) with $x = (x_0, x_1, \ldots, x_p)$ is incorrectly classified (that is, $\hat{f}(x) \neq y$), then we change the weight vector according to

$$w \leftarrow w + y \cdot x.$$

The intuition is that we tilt the separating hyperplane to be more "in favour of" (x, y). Consider for example the case when $y = 1$ for the incorrectly classified (x, y), that is, $w^\top x < 0$. For the modified weight vector $w' = w + y \cdot x$, we get

$$w'^\top x = \sum_{i=0}^{p} w'_i x_i = \sum_{i=0}^{p} (w_i + x_i) x_i = \sum_{i=0}^{p} w_i x_i + \sum_{i=0}^{p} x_i^2 = w^\top x + \|x\|^2 > w^\top x.$$

Note that, even after the update $w'^\top x$ could be negative. Further, the update could turn previously correct classifications of other instances into incorrect ones. Therefore, we might need to make several passes over the dataset.

Still, it can be shown that if the dataset is linearly separable, then *the perceptron algorithm will find a separating hyperplane in a finite number of steps.* Otherwise, it would keep finding violated instances ad infinitum; the algorithm has to be terminated after some number of iterations.

We recall the basic version of the support vector classifier (with no error tolerance). That method finds the maximum margin separating hyperplane for linearly separable classes and is infeasible otherwise. The separating hyperplane found by perceptron is *not* the maximum margin one. In fact, we cannot give a guarantee on the quality of the hyperplane found and will depend on the order in which the data instances were considered. On the other hand, SVM requires to optimally solve a convex program that is computationally more expensive than the simple iterative method described above.

From the perceptron algorithm, two basic concepts will be relevant for more complex neural networks: *neurons* and *reinforcement learning.*

Multilayer perceptron The simplest example of a linearly non-separable dataset corresponds to the XOR function from logic. We now let 1 correspond to *yes* and 0 to *no.* The truth table is

0	0	0
1	0	1
0	1	1
1	1	0

It is easy to see that no linear separation is possible (see also Ex 5.2). Hence, perceptron cannot learn this function. We have already seen a standard approach to non-linear separation in the context of SVM: using kernels. Neural networks use a different approach: they combine multiple neurons into a network. In particular, three neurons in two layers can represent the XOR function *(see lecture slides).* The inputs to the neurons on the first layer are the original features (and bias); whereas the input to the neuron on the 2nd layer is the outputs of the neurons on the first layer (and bias).

Multilayer neural networks have a dramatically better expressive power. In fact, every binary function $f : \{0, 1\}^p \to \{0, 1\}$ can be expressed by a neural network with *only two layers.* Of course, the number of neurons needed might be exponentially large. Also, this result on expressiveness does not indicate how such a representation could be found using a training dataset.

Feedforward neural networks and activation functions We now describe a fundamental class of neural networks called *feedforward networks.* These are composed of artificial neurons with an input layer, one or more hidden layers, and a final output layer. In the simplest case the output layer comprises just a single neuron; but in general there could be more than one outputs. For classification with more than two classes, there is typically an output neuron for each of the classes.

Neurons on the input layer represent the feature vector. Neurons on every subsequent layer take their inputs from the previous layer, as well as a bias term. The neuron is equipped with a weight vector associated with the input vector, and uses an *activation function* to produce a single output. This output will serve as one of the inputs for the next layer.

Let $z = (z_0, z_1, \ldots, z_t)$ denote the input vector of a neuron, with $z_0 = 1$ representing the bias. For the associated weight vector $w = (w_0, z_1, \ldots, z_t)$, we compute the scalar product $w^\top z$. The activation function $h : \mathbb{R} \to \mathbb{R}$ is applied to this scalar product to obtain the output. The perceptron method uses the *binary step function* as in (1). This is a simple natural choice, but has some limitations: most importantly, it is not continuous (and therefore not differentiable). There are several other possible choices of the activation function $h(x)$, shown in the table below:

Sigmoid	$\frac{e^x}{e^x+1}$
Rectified linear unit (ReLU)	$\max(0, x)$
Softplus	$\log(1 + e^x)$
Tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$

The most commonly used one is the sigmoid function. This is the same function as in logistic regression. In fact, we can see logistic regression as a special neural network with no hidden layers and a single output neuron using the sigmoid activation function.

We note that there are also other models of neural networks. *Recurrent neural networks* may include cycles, allowing for feedback loops.

Forward propagation Neural networks (with a single binary output) can be seen as a parametric method representing a function $\hat{f}(x) : \mathbb{R}^p \to \mathbb{R}$, where the parameters are the weights, associated with each connection between the neurons. (Normally, every neuron on a certain layer is connected to every neuron in the next layer.)

Given a fixed set of weights, we can evaluate the function value $\hat{f}(x)$ in a natural way in a feedforward network. We substitute the feature values to the input layer. Then, we move layer-by-layer in *forward* direction: after we have computed the outputs on the i-th layer, we can use them as the inputs to the $(i + 1)$-st layer, until we reach the output layer. This substitution process is called *forward propagation*.

2 Training neural networks

2.1 Loss functions

In what follows, we describe how a neural network can be trained for a classification problem with p continuous inputs $(X_1, X_2, \ldots, X_p)$ and a binary output $Y \in \{0, +1\}$, using a training dataset Tr.

We assume the network structure is given: we have a certain number of neurons arranged in layers, and the inputs of every neuron on the hidden and output layers are all neurons on the previous layer as well as the bias term. Our goal is to calibrate the weights associated with the inputs of the neurons so that the function $\hat{f}(x)$ gives a good estimation on y on Tr.

The weights w_{ij} will be arranged in a weight matrix W. We let $\hat{f}_W(x) : \mathbb{R}^p \to [0, 1]$ denote the function encoded by the network using weights w. The output $p \in [0, 1]$ gives a probability distribution over the two values 0 and +1: we predict outcome +1 with probability p and 0 with probability $1 - p$.

The objective is to minimise a function $F_{\mathrm{Tr}}(W)$ that measures the performance. The function will be of the form

$$F_{\mathrm{Tr}}(W) = \frac{1}{|\mathrm{Tr}|} \sum_{(x,y)\in \mathrm{Tr}} L(\hat{f}_W(x), y) + \lambda R(W).$$

Here, the first term is the average *loss function* on the training set, $R(W)$ is a *regulariser*, and $\lambda > 0$ a parameter calibrating the tradeoff. The loss function compares the predicted class $\hat{f}_W(x)$ to the actual class y. We can use the quadratic of informational loss functions for probabilistic outputs from Lecture 2; we will now use informational loss (sometimes called softmax loss):

$$L(p, y) = -y \log p - (1 - y) \log(1 - p)$$

Note that this value is nonnegative and equals 0 only for perfect prediction ($y = 1$ and $p = 1$ or $y = 0$ and $p = 0$). The regulariser plays a similar role as in ridge or lasso regression; a common choice is $R(W) = \sum_{i,j} w_{ij}^2$.

Finding the weights exactly minimising $F_{\mathrm{Tr}}(W)$ is a very hard problem. The training process will use *gradient descent* to find an *approximate local minimum* of this function.

2.2 The gradient descent method

We now briefly review the *gradient descent method*, a simple first order optimisation technique for finding the local minimum of a function. Let $F : \mathbb{R}^n \to \mathbb{R}$ be a multivariate differentiable function with domain $\mathbf{dom}\ F \subseteq \mathbb{R}^n$. The goal is to solve the unconstrained optimisation problem

$$\min_{x\in\mathbb{R}^n} F(x).$$

Since F is assumed to be differentiable, its gradient exists at every point of the domain; this is the vector of partial derivatives:

$$\nabla F(x) = \left(\frac{\partial F}{\partial x_1}(x), \frac{\partial F}{\partial x_2}(x), \ldots, \frac{\partial F}{\partial x_n}(x) \right).$$

The value $x \in \mathbf{dom}\ F$ is a *local minimiser* of F if there is some $\varepsilon > 0$ such that $F(x) \leq F(y)$ for any $y \in \mathbb{R}^n$ with $\|x - y\| \leq \varepsilon$. If x is a local minimiser, then $\nabla F(x) = \mathbf{0}$ must hold.

The gradient vector gives a local linear approximation of the function F around x: for a short enough vector $v \in \mathbb{R}^n$, we have

$$F(x + v) \approx F(x) + v^\top \nabla F(x).$$

From this viewpoint, the direction of *steepest decrease* of this approximation is given by the negative gradient direction $-\nabla F(x)$.

The gradient descent method starts from a point $x^{(0)} \in \mathbf{dom}\ F$, and constructs a sequence of points $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$ using the update rule

$$x^{(i+1)} = x^{(i)} - \gamma_i \nabla F(x^{(i)}), \quad i = 0, 1, 2, \ldots$$

Here, $\gamma_i > 0$ is the *step size*, that can be determined in every iteration, or can be a fixed constant throughout. For small enough choice of $\gamma_i > 0$, we get that $F(x^{(i+1)}) < F(x^{(i)})$. Hence, we construct a sequence of points with decreasing objective values.

Under certain assumptions on the function F, one can give guarantees on the speed of convergence to a local optimum. Gradient methods are particularly well-suited for *convex minimisation problems*. If

the function F is convex, then every local optimum is a global optimum. Hence, in this case, gradient descent can be used to approximate the global minimiser of the function.

The function $F(W)$ optimised in neural networks is typically *non-convex*. However, there is an important special case when this yields a convex function, discussed next.

2.3 Logistic regression

Let us now consider the example of logistic regression: recall that this can be seen as a neural network with no hidden layers and a single output neuron using sigmoid activation function. We let $w = (w_0, w_1, \ldots, w_p)$ denote the weight vector that is now associated with the bias term and the p features.

For simplicity of notation, we again let $x = (x_0, x_1, \ldots, x_p)$ denote the feature vector including the bias term $x_0 = 1$. Then, the output value will be

$$\hat{f}_w(x) = \frac{e^{w^\top x}}{1 + e^{w^\top x}}$$

Then,

$$\begin{aligned} L(\hat{f}_w(x), y) &= -y \log \frac{e^{w^\top x}}{1 + e^{w^\top x}} - (1 - y) \frac{1}{1 + e^{w^\top x}} \\ &= -y \cdot w^\top x + \log(1 + e^{w^\top x}). \end{aligned}$$

If we do not use a regulariser term, then the goal is to minimise

$$F(w) = \frac{1}{|\mathrm{Tr}|} \sum_{(x,y) \in \mathrm{Tr}} \left(-y \cdot w^\top x + \log(1 + e^{w^\top x}) \right)$$

We note that this is the same as the negative logarithm of the likelihood estimate (see Lecture 5). This function is *convex*: every local minimiser is also a global minimiser. We now demonstrate how gradient descent and its variant can be applied to this loss function.

First, let us compute the gradient $\nabla F(w)$. We first compute separately each term $\nabla L(\hat{f}_w(x), y)$:

$$\frac{\partial L(\hat{f}_w(x), y)}{\partial w_i}(w) = x_i \left(\frac{e^{w^\top x}}{1 + e^{w^\top x}} - y \right) = x_i (\hat{f}_w(x) - y). \tag{2}$$

We can obtain the gradient $\nabla F(w)$ by averaging these over the test set, namely,

$$\frac{\partial F}{\partial w_i}(w) = \frac{1}{|\mathrm{Tr}|} \sum_{(x,y) \in \mathrm{Tr}} x_i (\hat{f}_w(x) - y).$$

The standard gradient descent, in this context is called *batch gradient descent*, computes the full gradient and updates

$$w \leftarrow w - \gamma \nabla F(w).$$

Note that this involves evaluating $\hat{f}_w(x)$ for every point in the training set, and can be computationally expensive for a large dataset. An efficient alternative is *stochastic gradient descent*: instead of the entire dataset, we only use single, randomly chosen sample $(x, y) \in \mathrm{Tr}$, and use only the term $\nabla L(\hat{f}_w(x), y)$ as an approximation of the full gradient ∇F. That is, the update becomes

$$w_i \leftarrow w_i - \gamma x_i (\hat{f}_w(x) - y).$$

In practice stochastic gradient descent is nearly as efficient as batch gradient descent, at much reduced computational cost. Note that the update rule is very similar to the one used in the perceptron method. Let $\alpha = -\gamma(\hat{f}_w(x) - y)$, that is, $w \leftarrow w + \alpha \cdot x$. Since $0 < \hat{f}_w(x) < 1$, if $y = 1$ then $\alpha > 0$, and if $y = 0$ then $\alpha < 0$.

An option between batch and stochastic gradient descent is to use a *minibatch*: select a fixed number (say 20) data items, and estimate the gradient by averaging the gradients of the loss functions of these items.

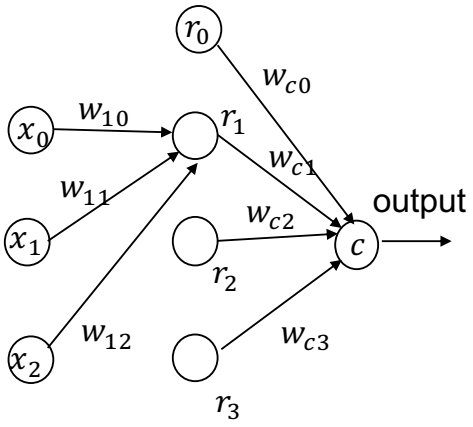
2.4 Backward propagation

The weight updates in multilayer neural networks will also be carried out via stochastic gradient descent or some other variant. Let $(x, y) \in \text{Tr}$; we need to compute the gradient of $L(\hat{f}_W(x), y)$. We can compute the partial derivatives $\frac{\partial L(\hat{f}_W(x),y)}{\partial w_{ij}}$ for the weights w_{ij} in a *backward order*. Namely, we first compute the partial derivatives for the edges entering the output nodes. For the sigmoid activation function this will be just the same as what we have seen for logistic regression. Then, we use these partial derivatives to compute those corresponding to weights on preceding layers.

While the notation and technicalities of these calculations may seem complicated, it all boils down to a simple application of the *chain rule of partial derivatives*. Namely, if $f = g(h(x))$ for multivariate functions $f : \mathbb{R}^n \to \mathbb{R}^k$ and $h : \mathbb{R}^n \to \mathbb{R}^m$, $g : \mathbb{R}^m \to \mathbb{R}^k$, then

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial h} \cdot \frac{\partial h}{\partial x}.$$

Here, the left hand side is a $k \times n$ dimensional matrix, and the right hand side is the matrix product of an $k \times m$ and an $m \times n$ dimensional matrix.



We demonstrate backpropagation on the example of a neural network with a single hidden layer containing k neurons, and a single output neuron. Every neuron uses the sigmoid activation function $\sigma(z) = \frac{e^z}{1+e^z}$. The i-th neuron on the hidden layer associates weight w_{ij} the the j-th input, $j = 0, 1, 2, \ldots, p$. The output of this neuron is

$$r_i = \sigma\left(\sum_{i=0}^{p} w_{ij} x_j\right).$$

The neuron on the output layer uses the weights $w_{c0}, w_{c1}, \ldots, w_{ck}$, where w_{c0} is associated with the bias

$r_0 = 1$ and w_{cj}, $j = 1, \ldots, k$ to r_j. Thus, the final output will be

$$\hat{f}_W(x) = \sigma\left(\sum_{j=0}^{k} w_{cj} r_j\right).$$

Note that $\hat{f}_W(x)$ is a function of W; the values of x are fixed according to the training instance. We can see the above expression as a composite function, where the r_j's are function of the weights from the previous layer. We start by computing the partial derivatives of $L = L(\hat{f}_W(x), y)$ with respect to the weights w_{cj}, as well as with respect to the r_j's. The derivation w.r.t. the w_{cj}'s is just the same as for logistic regression in (2):

$$\frac{\partial L}{\partial w_{cj}} = r_j(f_W(x) - y), \quad \forall j = 0, 1, \ldots, k.$$

Since the role of the r_j's and w_{cj}'s is symmetric, the same derivation shows:

$$\frac{\partial L}{\partial r_j} = w_{cj}(f_W(x) - y), \quad \forall j = 0, 1, \ldots, k. \tag{3}$$

We will now use the chain rule to compute the partial derivatives $\frac{\partial L}{\partial w_{ij}}$ for $i = 1, 2, \ldots, k$, $j = 0, 1, \ldots, p$. We can see L as a composite function where the outermost function uses the variables $(r, w_c) = (r_0, r_1, r_2, \ldots, r_k, w_{c0}, w_{c1}, \ldots$ (again, with the convention that $r_0 = 1$). Then,

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial (r, w_c)} \cdot \frac{\partial (r, w_c)}{\partial w_{ij}} = \sum_{t=0}^{k} \frac{\partial L}{\partial r_t} \cdot \frac{\partial r_t}{\partial w_{ij}} + \sum_{t=0}^{k} \frac{\partial L}{\partial w_{ct}} \cdot \frac{\partial w_{ct}}{\partial w_{ij}}.$$

Observe that if $i \neq t$, then the weight w_{ij} does not impact the value of r_t, and therefore $\frac{\partial r_t}{\partial w_{ij}} = 0$ whenever $i \neq t$. Also, $\frac{\partial w_{ct}}{\partial w_{ij}} = 0$ for all t. Therefore, the above formula simplifies to:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial r_i} \cdot \frac{\partial r_i}{\partial w_{ij}}. \tag{4}$$

Let us now compute $\frac{\partial r_i}{\partial w_{ij}}$. One can verify that the derivative of the sigmoid function is

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)).$$

Recall that $r_i = \sigma(\sum_{t=0}^{p} w_{it} x_t)$. Therefore,

$$\frac{\partial r_i}{\partial w_{ij}} = x_j r_i (1 - r_i).$$

Consequently, from (3) and (4) we obtain

$$\frac{\partial L}{\partial w_{ij}} = w_{ci}(f_W(x) - y) x_j r_i (1 - r_i).$$

Thus, we have obtained the entire gradient of $L = L(\hat{f}_W(x) - y)$. In case there are more than two layers, the same process can be continued backwards.

Initialisation To start the gradient descent algorithm, we need to initialise the weights. These are typically chosen at random. Note that starting with the same weights (say, $w_{ij} = 1$ for all values) is not a good choice. In this case, the backpropagation would make the same changes to all the weights between any two layers, thus all neurons on the same layer would behave the same. It would have the same effect as if we had just a single neuron on each layer.

2.5 Early stopping and hyperparameter selection

Backpropagation optimises the network for the training set, and there is a vast number of parameters (the edge weights). Due to the large flexibility, it is prone to overfitting. The purpose of the regulariser term is to mitigate this effect: bounding the size of the weights will result in more robust networks.

Early stopping is also used to prevent overfitting. We split the dataset into training and validation set (say, in proportion 2:1). We use only the training set to update the weights, and after every T update iterations (for some value of T), we compute the error rate on the training set. We continue training until this error rate decreases, and stop once it starts to increase.

Note that this is different from cross-validation: we use the validation set as part of the training process. Cross-validation is typically not feasible for neural networks as training networks can be very time consuming. We usually do split test: we keep a separate test data (different from the validation data), and use it to test the network performance once training is finished.

Neural networks involve lots of *hyperparameters*: we need to specify the number of layers, the number of neurons on each layer, the activation functions, the loss function, the learning rate, etc. Again, we can us a validation set to tune these parameters. Given the high number of parameters, we cannot afford a grid search to test all combinations, but rather tune them separately.

3 References

- **Weka** 4th edition, Sec. 7.2, Chapter 10 (not in 3rd edition)
- **ESL** Chapter 11 (2nd ed.)
- There are several blog posts and online resources explaining the basic ideas:
  e.g. `http://neuralnetworksanddeeplearning.com/`, `https://ujjwalkarn.me/`
- A nice visual demonstration of neural network training: `http://playground.tensorflow.org`.

MA429 Algorithmic Techniques for Data Mining
Lecture 10: Neural networks II: Deep learning

László Végh
L.Vegh@lse.ac.uk.

1 Two or more layers

Most research in the 1960-80s focussed on neural networks with two layers only: a single hidden layer and the output layer. This choice was based on multiple theoretical and practical considerations. As mentioned on the previous lecture, a neural network with a single hidden layer can be used to express any binary function, as well as can approximate continuous functions arbitrarily. Hence, using more than two layers does not bring any increase in the expressivity power.

Another important consideration was that training neural networks with multiple hidden layers becomes challenging via gradient descent, and it was seen impractical. The most important issue is the *vanishing gradient problem*: the partial derivatives corresponding to deeper layers can be exponentially small: thus, the training speed becomes very slow. Note that backpropagation computes gradients via the chain rule as products of several terms. Many traditional activation functions, such as sigmoid or hyperbolic tangent, have derivatives strictly between 0 and 1. Taking the products of many such terms can lead to exponential decrease in the gradient terms as we move to layers further back. Using different activation functions with larger gradients may escape this phenomenon, but can lead to the opposite: *exploding gradients*. That is, the gradients could become very large and unstable: small changes in the input may result in very different updates.

There have been various approaches to tackle these problems. A common feature is to deviate from the standard feedforward network structure, where every neuron on the same layer is symmetric: they receive outputs of every neuron on the previous layer and pass on to everyone on the next layer. *Recurrent neural networks* use feedback loops, and comprise structured elements simulating logic circuits of a computer architecture. One particular example that deals with the vanishing gradient problem is *Long short-term memory (LSTM)* networks: these include units that are able to retain important information for a longer period of time.

Besides using more advanced, structured forms of networks, the success of deep networks can be heavily attributed to hardware development and enhanced computational capacities. An important factor is the use of *Graphical Processing Units (GPUs)*. These are ideal for the large scale parallel matrix computations in gradient descent. A third main factor is the increase in the amount and availability of training data.

Deep learning vs representation learning Classical machine learning methods work with the original set of features, or with a subset of those. In representation learning, we first transform the set of features to a more suitable form: a prime example would be principal component analysis. Deep learnings can be understood as taking this idea to a next level: they can build up increasingly complex represen-

tations layer-by-layer: every subsequent layer corresponds to a higher level of abstraction. This will be illustrated by the example of convolutional neural networks.

2 Convolutional neural networks

Convolutional neural networks (CNNs) are a class of neural networks particularly suitable for image recognition and classification in computer vision. Here, the input layer consists of pixels of an image; in the simplest case, the grayscale colour represented by 1 byte (values 0-255). Instead of looking at an 1000×1000 picture as a 1,000,000 dimensional vector (as done e.g. in the eigenfaces example), convolutional networks aim to recognise local patterns of the picture. The neurons in the first few layers of the network correspond to *filters*. The network between these layers is very *sparse*: every neuron is connected to only a small number of inputs from the previous layer. A convolution filter is represented by a $k \times k$ matrix. This is placed over a $k \times k$ region of the image matrix, and we compute the pointwise product of the two matrices; this will be the input of the neuron. The output is typically obtained using the ReLU activation function $h(x) = \max(0, x)$. *See lecture slides for illustrations.*

Different convolution matrices can extract different abstract features and patterns. Using each of the filters we can obtain a *feature map*: a transformed version of the picture that highlights the occurrences of the pattern. The filters are not fixed, but will be learned using the training process: the architecture only fixes the number of feature maps, and the sizes of the filters. Then, the size of the filter maps is reduced using a *pooling* step.

Convolutional filters can be applied in multiple iterations; this can be thought of extracting increasingly more abstract features. The output of the convolutional steps is fed into a traditional, fully connected feedforward network, the will provide the outputs.

3 References

- **Weka** 4th edition, Chapter 10 (not in 3rd edition)
- Convolutional networks: `/https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/`
- Societal impacts of data mining: Cathy O'Neil: Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy, Crown Books, 2016.