Overview of the DAT200 Syllabus

Jorid Holmen

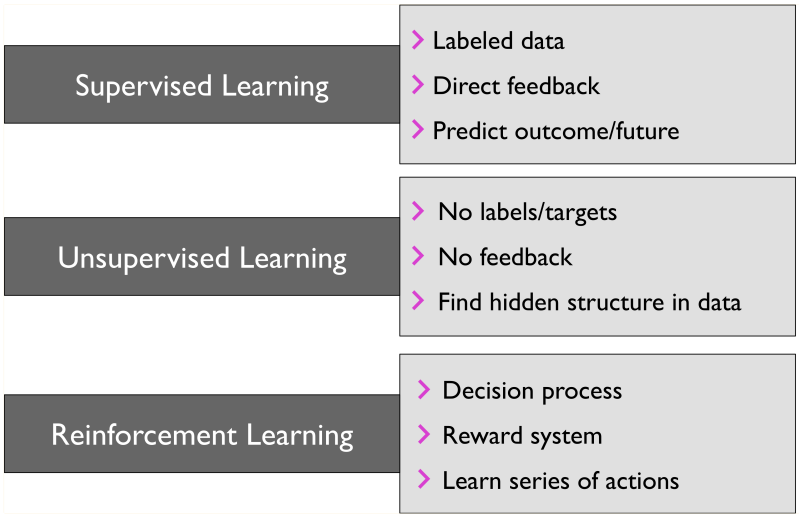
Spring 2023

Contents

Chapter 1

Giving Computers the Ability to Learn from data

1.1 The three different types of machine learning



1.1.1 Supervised learning

The main goal in supervised learning is to learn a model from labeled training data that allows us to make predictions about unseen or future data.

The labeled training data is passed to a machine learning algorithm for fitting a predictive model that can make predictions on new, unlabeled data inputs.

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels (for example spam and non-spam emails). **Regression** is another subcategory of supervised learning and is the predictions of continuous outcomes, and is also called regression analysis. We are given a number of predictor (explanatory) variables and a continuous response variable (outcome), and try to find a relationship between those variables that allows us to predict an outcome. The predictor variables are often called features and the response variables are often called target variables. An example of regression is finding a relationship between the time spent studying and SAT-scores.

1.1.2 Reinforcement learning

In reinforced learning the goal is to develop a system (agent) that improves its performance based on the interactions in the environment. Since the information of the current state also includes a so called reward signal, we can think of reinforced learning as related to supervised learning.

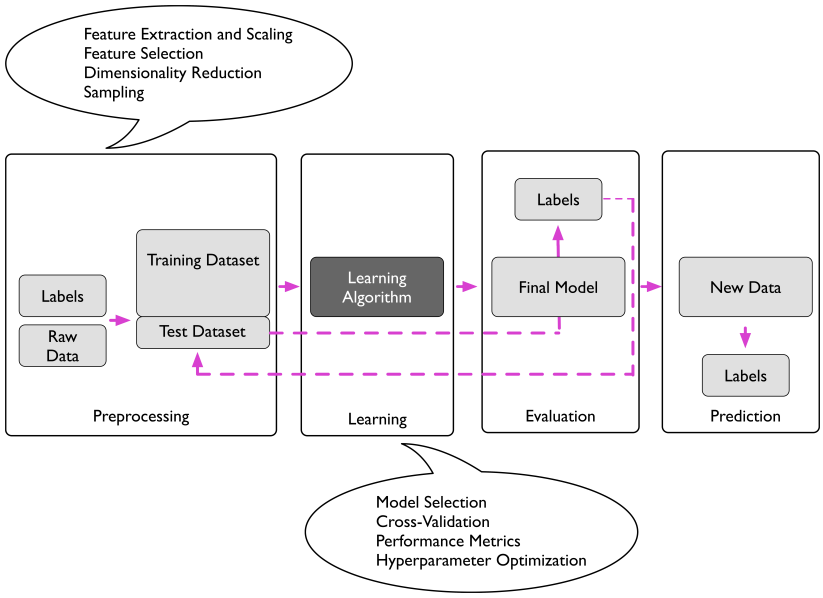
1.1.3 Unsupervised learning

In unsupervised learning we are dealing with unlabeled data or data of unknown structure. Using unsupervised learning techniques we are able to explore the structure of our data to extract meaningful information without the guidance of a known outcome or reward function.

Clustering is an exploratory analysis technique that allows us to organize a pile of information into meaningful subgroups (clusters) without having any prior knowledge of their group memberships. Objects within a cluster share a certain degree of similarity.

Dimensionality reduction is another subfield of unsupervised learning. High dimenstionalty means that each obseration comes with a high number of measurements, that can present a challenge for limited storage space or the computational performance of machine learning. Dimensionallity reduction is a common aproach in feature preprocessing to remove noise or compress data into smaller subspaces. It is also useful for visualisation.

1.2 A roadmap for building machine learning systems



1.2.1 Preprocessing

Raw data rarely comes in the shape and from that is necessary for the optimal performance of a learning algorithm. Thus, the preprocessing is one of the most crucial steps in any machine learning application.

Feature extraction is to extract meaningful data from the dataset. Many algorithms also require the selected features to be on the same **scale** for optimal performance. Some selected features might be highly correlated and therefore redundant (no longer useful) to a certain degree. In those cases **dimensionality reduction** techniques are useful for compressing the features onto a lower dimensional subspace. Then we **split** the dataset into training and test dataset.

1.2.2 Training

There are many different machine learning algorithms available, and each algorithm has is strengths and weaknesses. Therefore it is wise to measure the **performance** using for example accuracy, AUC or ROC. **Cross-validation** is used for estimation of generalisation performance, and is done by further dividing a dataset into training and validations subsets. Since we can't expect that the default parameters of the different learning algorithms provided by software libraries are optimal for our specific problem task, we also need to make frequent use of **hyperparameter optimization** techniques that help us fine tune the

performance of our model. We can think of hyperparameters as parameters that are not learned from the data but represent the knobs of a model that we can turn to improve it performance.

1.2.3 Evaluating

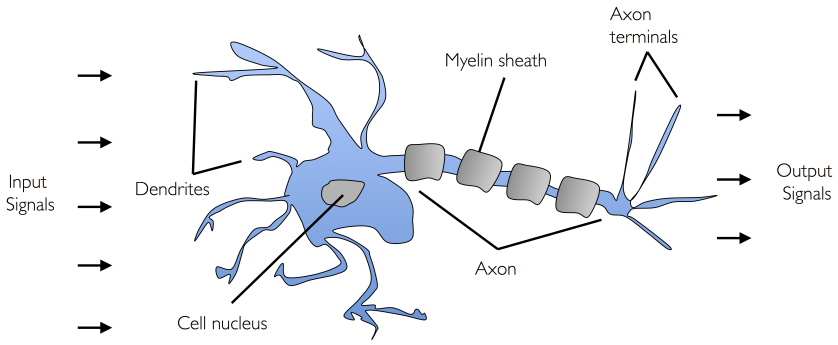
After we have selected a model that has been fitted on the training datset, we can use the test dataset to estiamte how well it performs on this unseen data to estimate the generalization error. All transformations of the training data are applied to the test data, using the parameters acquired from transformation of training data.

Chapter 2

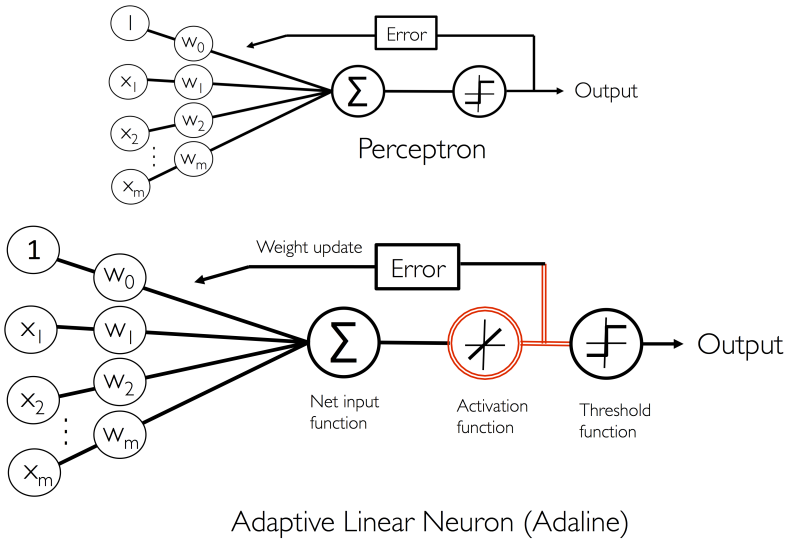
Training Simple Machine Learning Algorithms for Classification

2.1 Artificial Neurons

Neurons are interconnencted nerve cells in the brain, involved in processing and transmitting chemical and electrical signals. Neurons acts as a simple logic gate with binary output. Multiple signals arrive at the dendrites, they are then integrated into the cell body, and if the accumulated signal exceeds a certain threshold, an output signal is generated that will pass on by the axon.



The bias makes it possible to not go through origo when classifying.



2.2 Perceptron

The perceptron algorithm can be summarized as follows:

1. Initialize the weights to 0 or small random numbers.
2. For each training example $\mathbf{x}^{(i)}$:
   (a) Compute the output value, $\hat{y}$.
   (b) Update the weights. If the true class and the predicted output is the same, the weight will not be updated.

The update of each weight, w_j, can be formally written as:

$$w_j := w_j + \Delta w_j, \tag{2.1}$$

where

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}. \tag{2.2}$$

w_j: A single weight in the perceptron, and part of the weight vector, w.
Δw_j: The update to the weight as a result of the perceptron learning rule (A function of the learning rate and prediction error)
η: The learning rate (typically between 0 and 1) which decides the magnitude of updates to individual weights after each prediction
$y^{(i)}$: true class label/target
$\hat{y}^{(i)}$: predicted class label/target

$x_j^{(i)}$: value of feature j in sample vector i, the input value of the the training sample

We do not update the the predicted label $\hat{y}^{(i)}$ before alll the weights are updated via the respective update values, Δw_j.

when classifying a sample the perceptron sums up the input signals multiplied with a weight. The sum is apllied to the threshold to predict which class the sample belongs to. The net input function, z is th input multiplied with the weight:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m \tag{2.3}$$

The net input, z is then passed through the activation/threshold function. It transforms the net input into the final output of the perceptron. $x_[0]$ is the bias, and $w_[0]$ is the weight bolinging to the bias.

$$\Phi = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

Convergence of the perceptron algorithm is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If classes cannot be separated by a linear decision boundary, take at least one of the following two measures:

- Set maximum number of iterations (epochs) over training samples.
- Set threshold for maximum number of tolerated missclassifications.

If classes are not linearly separable AND none of the two above measures were taken, the Perceptron will never stop updating weights.

2.3 Adaline

Adaline stands for Adaptive Linear Neuron and illustrates the key concepts of defining and minimizing continuous cost functions. The main difference between adaline and pecreptron is that in adaline the weights are updated using a linear activation function:

$$\Phi(z) = w^T x. \tag{2.5}$$

While the linear activiation function is used for learning the weights, we still use a threshold function to make the final prediction. This means the weights can be updated according to how wrong or confident the model is.

2.4 Minimizing cost funcions with gradient descent

One of the key ingredients of supervised learning algorithms is a defined objective function that is to be optimized during the learning process. This objective function is often a **cost function** that we want to **minimize**.

The goal of using objective/loss/cost function is:

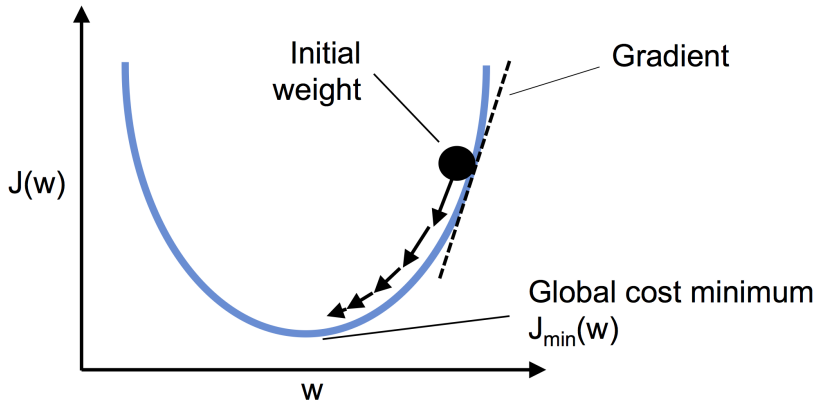
- Capture the performance of the model, how well the model is predicting classes (quality of the output).
- Doing so by computing error or distance score between the true class label and the output of the activation function.
- Compute weights such that cost function minimizes (find global cost minimum where error is as small as possible)

Adalines cost function, J, learns the weights as the **Sum of squared errors (SSE)** between the calculated outcome and the true class label:

$$J(w) = \frac{1}{2} \sum_i (y^{(i)} - \Phi(z^{(i)}))^2. \tag{2.6}$$

The $\frac{1}{2}$ is just added for out convenience and will make it easier to derive the gradient of the cost function with respect to the weight parameters.

The main advantage of this continuous linear activation function is that the cost function becomes differentiable. Another great thing is that it is convex, and thus we can use an optimization algorithm called **gradient descent** to find the weights that minimize our cost function.



Using gradient descent we can now update the weights by taking a step in the opposite direction of the gradient, $\nabla J(w)$, of our cost function $J(w)$:

$$w_: = w + \Delta w, \tag{2.7}$$

where

$$\Delta w_j = -\eta \nabla J(w). \tag{2.8}$$

To compute the gradient of the cost function, we need to compute the partial derivative of the cost function with respect to each weight, w_j:

$$\nabla J(w) = \frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \Phi(z^{(i)}))x_j^{(i)}, \tag{2.9}$$

which leads to

$$\Delta w_j = \eta \sum_i (y^{(i)} - \Phi(z^{(i)}))x_j^{(i)} \tag{2.10}$$

2.4.1 Improving gradient descent through feature scaling

Many machine learning algorithms require some sort of feature scaling for optimal performance (also called standardisation). Gradient descent is one of the many algorithms that benefit from feature scaling.

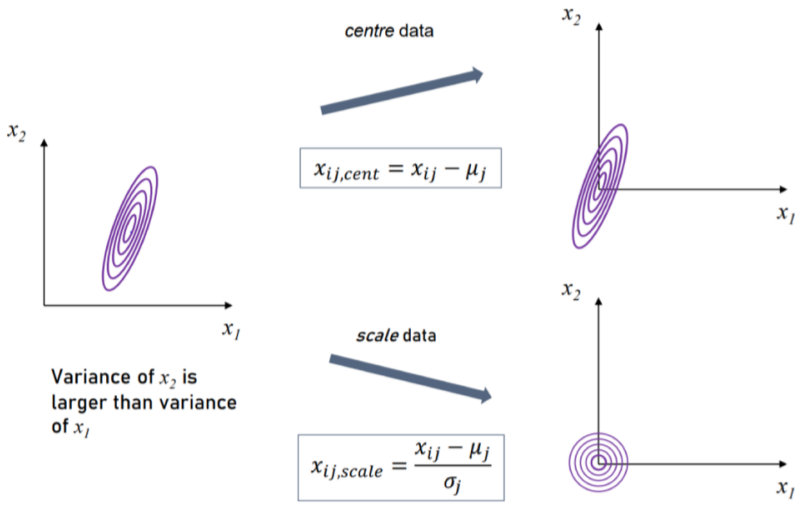
Standardisation/scaling gives the data some specific properties:

- Standard distribution, which helps the gradient descent learning to converge more quickly
- Shifts the mean of each feature so that it is centered at zero
- Each feature has a standard deviation of 1

The scaling helps our optimiser (gradient descent) to find good or optimal solutions in fewer steps.

$$x'_j = \frac{x_j - \mu_j}{\sigma_j} \tag{2.11}$$

x_j: feature vector (variable or column in data)
μ_j: mean offeature vector
σ_j: standard deviation of feature vector
x'_j: scaled feature vector



2.4.2 Stochastic gradient descent

The gradient descent referenced over is called **batch gradient descent**. With extremely large data, the batch gradient descent can be computationally quite costly. A popular alternative is stochastic gradient descent (SDG), which is also called iterative or online gradient descent.

Chapter 3

A Tour of Machine Learning Classifiers Using scikit-learn

3.1 Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice and experience; each algorithm has its own quirks based on certain assumptions.

No Free Lunch theorem: no single classifier works best across all possible scenarios

In practise it is recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem. The problem at hand may be influenced by a number of contextual settings:

- Number of features
- Number of samples
- Distribution of the data
- Amount of noise in the data
- Whether classes are linearly seperable or not
- Etc

The five main steps that are involved in training a supervised machine learning algorithm can be summarized as follows:

1. Selecting features and collecting labeled training examples.
2. Choosing a performance metric.
3. Choosing a classifier and optimization algorithm.

4. Evaluating the performance of the model.

5. Tuning the algorithm.

3.2 First steps with scikit-learn

Although many scikit-learn functions and class methods also work with class labels in string format, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention amongst most machine learning libraries.

3.2.1 Splitting into training and test data

To evaluate how well a trained model performs on unseen data, we will split the dataset into separate training and test datasets. Do this using the command `train_test_split` from scikit-learn's `model_selection`, which contains the following properties:

- Shuffle training set internally before splitting → we don't need to worry about ordering of the samples prior to splitting
- Provides a `random_state` parameter for a fixed random seed (default: `random_state=1`) → "Reproducible" shuffling of samples (handy when training multiple times)
- Built-in support for stratification with `stratify=y` → `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset

3.2.2 Feature scaling

Many machine learning and optimisation algorithms also require feature scaling for optimal performance. Standardise the features using the `StandardScaler` class from scikit-learns `preprocessing` module. Use `fit` method to estimate the sample mean, μ, and standard deviation σ for each feature dimension. Call `transform` to standardise training and test data using μ and σ acquired from training data.

```
from sklearn.preprocessing import StandardScaller
sc = StandardScaler()
sc.fit(X_train)
X_train_sc = sc.transfrom(X_train)
X_test_sc = sc.test(X_train)
```

Note that we use the same scaling parameters to standardize the test dataset so both the values in the training and test dataset are comparable to each other.

3.2.3 Multiclass support

Most algorithms in scikit-learn support multiclass classification through the OvA / OvR method. **One-vs-all (OvA)**, or One-vs-Rest (OvR) is a technique that allows us to extend a binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the samples from all other classes are considered negative classes. Most algorithms in scikit-learn support multiclass classification.

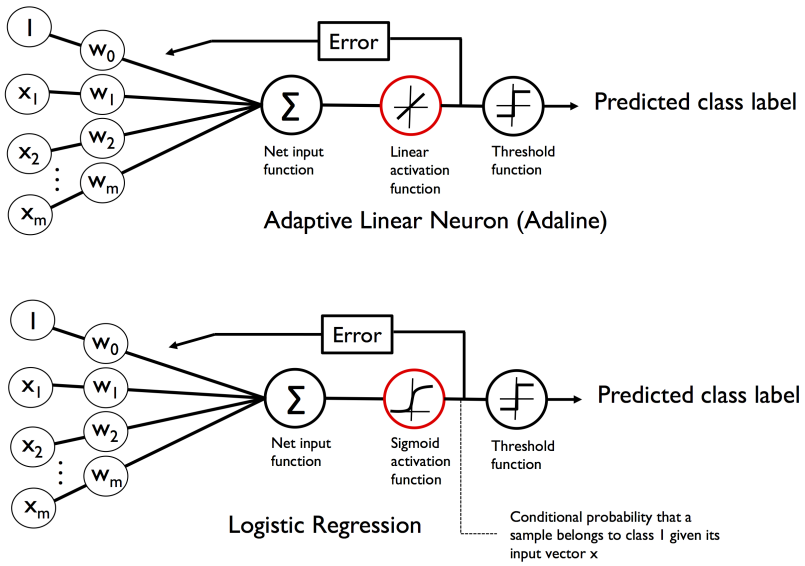
For a multiclass problem with 4 classes and 17 features, we will have 4 models, each with 17 weigths.

With Perceptron:

- For the new sample, compute net input value z for each of the n trained classifiers (one classifier per class)
- Choose the class that is associated with the largest net input value z among the n trained classifiers.

3.3 Logistic regression

Logistic regression is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry and is very similar to Perceptron and Adaline, and the only difference from Adaline is the use of a sigmoid activation function instead of a linear activation function. It is a binary classification and can be extended to multiclass classification, for example by OvR.



Logistic regression is interested in predicting the probability that a certain sample belongs to a particular class, using the sigmoid activation function:

$$\Phi(z) = \frac{1}{1 + e^{-z}}. \tag{3.1}$$

The sigmoid activation function activates the output in the area 0 to 1 in the case of binary classification. The output of $\Phi(z)$ is the output value of the net input after applying the sigmoid activation function. Large values of z lead to a $\Phi(z)$ value close to 1, and smaller values lead to values close to 0.

z is the net input, the linear combination of weights and sample features:

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{j=0}^{m} x_j w_j = w^T x. \tag{3.2}$$

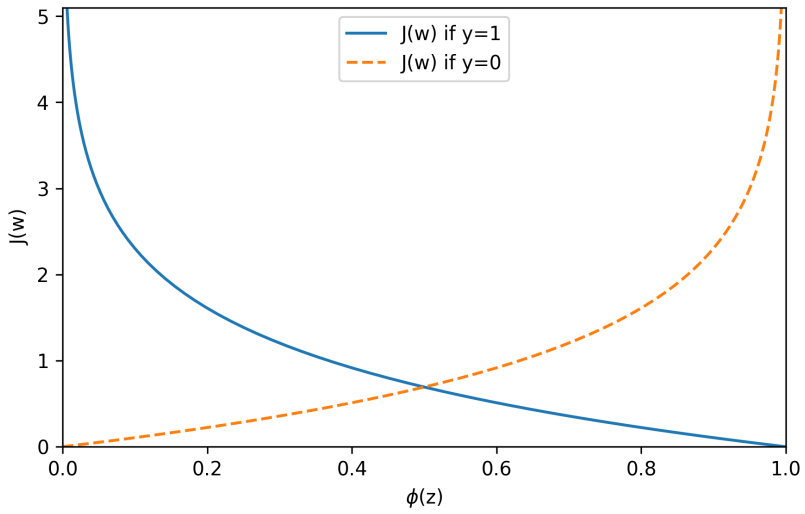
$\Phi(z)$ approaches 1 if z goes towards infinity, and 0 if z goes towards 0. A sigmoid function takes real numbers values as input and transforms them into values in the range [0,1] with an intercept at $\Phi(z) = 0.5$. The output is interpreted as the probabilty of a particular sample belonging to class 1 given its features x and parameterised by weights w. The predicted probability can then simply be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1, & \text{if } \Phi(z) \geq 0.5 \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

This is equivalent to

$$\hat{y} = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

3.3.1 Learning the weights of the logistic cost function



The cost function $J(w)$ for logistic regression based $\Phi(z)$ and the true class label y:

$$J(w) = \sum_{i=1}^{n}[-y^{(i)} \log(\Phi(z^{(i)})) - (1 - y^{(i)})log(1 - \Phi(z^{(i)}))]. \tag{3.5}$$

Looking at figure above one can see that a wrong predictions is penalised with increasingly larger cost. If a sample belongs to class 1 the cost function for a single sample is $log(\Phi(z^{(i)}))$, and $\log(1 - \Phi(z^{(i)}))$ if the class is 0.

Using gradient descent to update the weights by taking a step in the opposite direction of gradient $\nabla J(w)$ of our cost function $J(w)$

$$w_: = w + \Delta w, \tag{3.6}$$

where

$$\Delta w_j = -\eta \nabla J(w). \tag{3.7}$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = -\eta \sum_{i=1}^{n}[-y^{(i)} \log(\Phi(z^{(i)})) - (1 - y^{(i)}) \log(1 - \Phi(z^{(i)}))] \tag{3.8}$$

3.3.2 Overfitting

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from **overfitting**, we also say that the model has high variance,

which can be caused by having too many parameters, leading to a model that is too complex given the underlying data. Similarly our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very usefull method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting. The concept behind regularization is to introduce additional information (bias) to penalise extreme parameter (weight) values. To make regularization work properly, features need to be scaled/standardised.

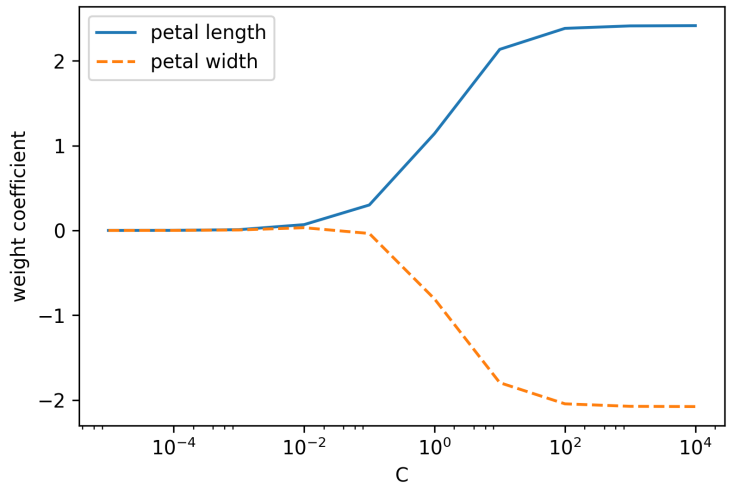
Most common form of regularisation is L2 regularisation which can be written as follows:

$$\frac{\lambda}{2}\|w\|^2 = \frac{\lambda}{2}\sum_{j=1}^{m} w_{j=1}^2 \tag{3.9}$$

λ is the so-called regularization parameter. The cost function for logistical regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:

$$J(w) = \sum_{i=1}^{n}[-y^{(i)}\log(\Phi(z^{(i)})) - (1-y^{(i)})\log(1-\Phi(z^{(i)}))] + \frac{\lambda}{2}\|w\|^2. \tag{3.10}$$

Via λ we control how well we fit the training data, while keeping the weights small. By increasing the value of λ we increase the regularization strength.



In scikit-learn implements `c` which is the inverce of λ. As wee can see in the plot, the weight coefficients shrink if we decrease parameter c, which means we increase the regularization strength.

3.4 Support vector machines (SVM)

Another powerful and widely used learning algorithm, is the **support vector machine (SVM)**, which can be considered an extension of the perceptron. Whereas perceptron minimises missclassification errors, SVM wants to **maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane. These training samples are the so called support vectors.**

Hard margin SVM attempts to find a hyperplane that perfectly separates the two classes, with no margin violations allowed. This means all the training examples must lie on the correct side of the decision boundary. **Soft margin** SVM allows for some margin violations in the training data, in exchange for a wider margin, and can better accommodate for noise and outliers.

3.4.1 Maximum margin intuition

The reasoning behind having decision boundaries with large margins is that they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting. The width of the margin is defined as $\frac{2}{||w||}$. This means that larger weights leads to wider margins. In summary, a hard margin SVM is a more strict approach that requires the data to be perfectly separable, and can therefor lead to overfitting, while a soft margin SVM allows some errors in exchange for a more robust decision boundary that can generalize better to new data.

Objective function of SVM (hard margin classification):

$$w_0 + w^t x^{(i)} \geq 1 \text{ if } y^{(i)} = 1 \tag{3.11}$$

$$w_0 + w^t x^{(i)} \leq -1 \text{ if } y^{(i)} = -1 \tag{3.12}$$

Dealing with nonlinearly separable case using slack variables

The slack variable, ξ, is used for **soft margin classification**, and controls the degree to which the SVM allows for margin violations. Larger ξ means softer margins. The reason for introducing the slack variable is to relax the linear constraints for nonlinearly separable classes, which allows for algorithm convergence in the presence of misclassifications, under appropriate cost penalisation. The positive-values slack variable is simply added to the the linear constraints:

$$w_0 + w^t x^{(i)} \geq 1 - \xi^{(i)} \text{ if } y^{(i)} = 1 \tag{3.13}$$

$$w_0 + w^t x^{(i)} \leq -1 + \xi^{(i)} \text{ if } y^{(i)} = -1 \tag{3.14}$$

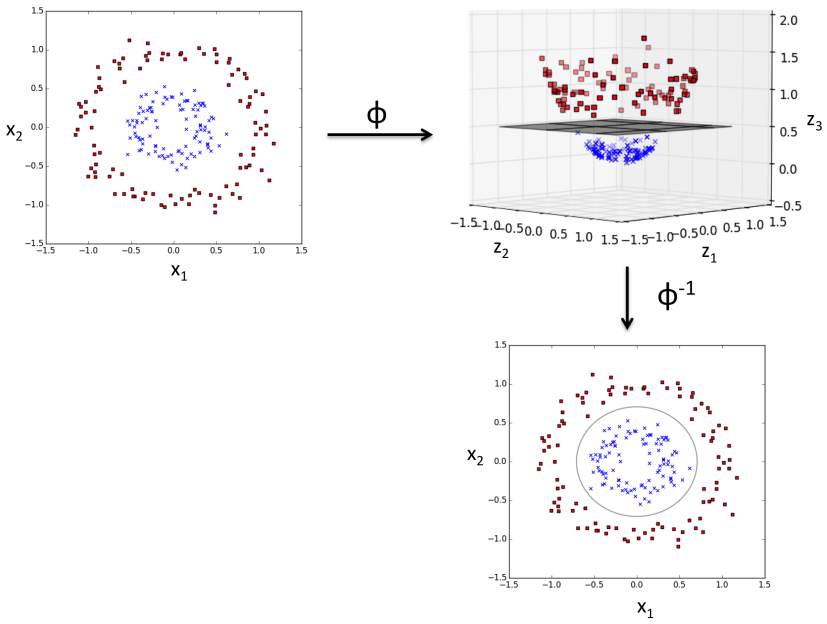
3.4.2 Logistic regression vs. linear support vector machines

- Often yield very similar results
- Logistic regression tries to maximize the conditional likelihoods for all of the training data, which makes it more prone to outliers than SVMs
- SVMs mostly care about the samples that are closest to the decision boundary (support vectors)
- On the other hand, logistic regression has the advantage that it is a simpler model and can be implemented more easily
- Logistic regression models can easily be updated, which is attractive when working with streaming data.

3.4.3 Solving non-linear problems using kernel SVM

Classifiers with a linear decision boundary cannot separate positive and negative classes very well, but the kernel method is able to deal with this. The idea behind kernel methods:

- Create nonlinear combinations of the original features
- Project original features onto a higher-dimensional space using a mapping function Φ
- in this higher-dimensional space the data becomes separable.



To solve a nonlinear problem using SVM we would transform onto a higher-dimensional feature space using a mapping function Φ, and train a linear SVM model to classify the data in this new higher-dimensional feature space. Then, we could use the same mapping function, Φ, to transform new unseen data to classify it using the linear SVM model.

A problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. Instead we use a computationally less expensive kernel function, **The kernel trick**. It computes the distance between samples for the higher-dimensional feature space, without ever actually computing the higher-dimensional feature space. To make a prediction for the new sample, the distance to each of the support vectors is measured, and a classification decision is made based on the distance to the support vectors and the importance of the support vectors that was learned during training. The distance between the two samples is measured by the kernel.

We have different kernels to compute distance between two sample:

- Linear kernel
- Polynomial kernel
- Radial Basis Function kernel (RBF, most widely used)
- Hyperbolic tangent kernel

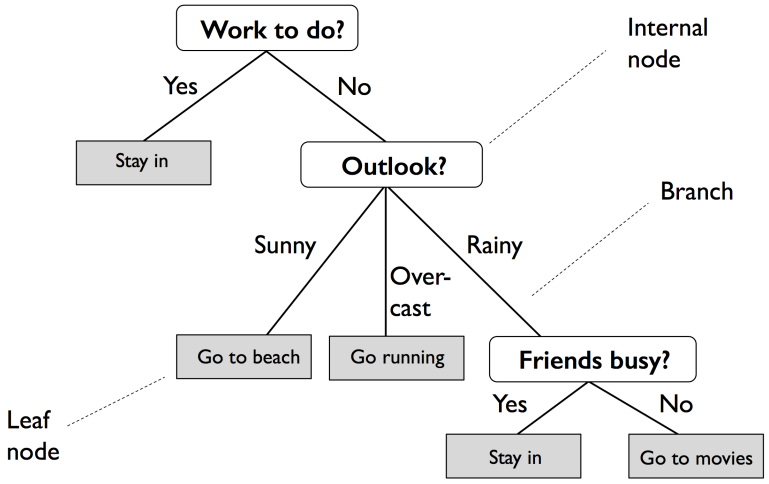
The parameter γ, which we can set to γ=0.1, can be understood as the as a cut-off parameter for the Gaussian sphere. For a relatively small value of γ the resulting decision boundary of the RBF kernel SVM model will be relatively

soft, which could lead to underfitting. If we increase γ, the influence or reach of the training examples will also increase, which leads to a tighter, bumpier decision boundary, which could lead to overfitting. This means that γ plays a role in controlling overfitting. Bigger gamma means more curves.

Another hyperparameter is C, which is a part of the L2 penalty. It controls the training error and reduce overfitting. A large value for C means smaller margin.

3.5 Decision Tree

Decision tree classifiers are attractive models if we care about interpretability. Decision tree models break down our data by making decisions based on asking a series of questions. Decision tree handles both categorical features and features with real numbers.



Based on the features in our training dataset, the decision tree model learns a series of questions to infer the class labels of the examples. We start at the tree root and split the data on the feature that results in the largest **information gain (IG)**. The information gain is the difference between the impurity of the parent node and the sum of the child node impurities. In an iterative process, we repeat this splitting procedure at each child note until the leaves are **pure**, which means the samples/training examples at each node all belong to the same class. This may result in a very deep tree with many nodes and can easily lead to overfitting. Therefore, we need to **prune** the tree by setting a limit for maximal depth of the tree.

3.5.1 Maximizing information gain (IG)

In order to split the nodes at the most informative features, we need to define an objective function that we want to optimise using the tree learning algorithm.

Here our objective function maximises IG at each split:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j) \tag{3.15}$$

IG: information gain
I: Impurity measure
f: feature that performs the split
D_p: Data set of the parent node
D_j: Data set of jth child node
N_p: Total number of samples at parent node
N_j: Number of samples at the jth child node

In order to reduce the combinatorial search space, most libraries (including scikit-learn) implement decision trees. Which means that each node is split into D_{left} and D_{right}.

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}), \tag{3.16}$$

where N_{left} and N_{right} is the number of samples at the left and right node, respectively.

Three commonly used **impurity measures** is

- Entropy I_H
- Gini impurity I_G
- Classification error I_E

Entropy I_H

The entropy criterion attempts to maximise the mutual information in the tree.

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t) \tag{3.17}$$

$p(i|t)$ is the proportion of the samples that belongs to class c for a particular node t. The entropy if 0 if all samples at a node belong to the same class. Entropy is maximal if we have uniform class distribution. The maximal entropy depends on number of classes.

Gini impurity I_G

The gini impurity is the criterion to minimise the probability of missclassification.

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2 \tag{3.18}$$

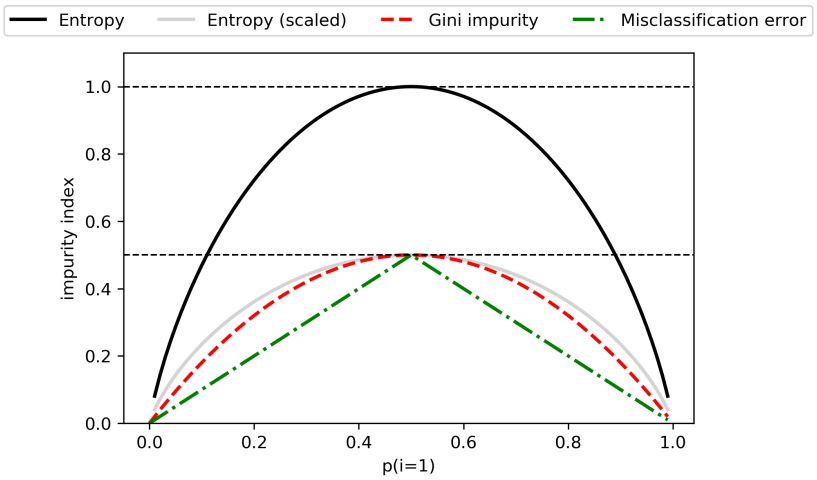
Entropy vs. Gini impurity

In practice both Gini and Entropy typically yield very similar results and they are both usually used for uniform distributions. It is often not worth spending time on evaluating trees using different impurity criteria, but rather experiment with different pruning cut-offs. Classification error is also useful for pruning.

Classification error I_E

$$I_E(t) = 1 - \max\{p(i|t)\} \tag{3.19}$$

Classification error is a useful criterion for pruning. However, it is not recommended for growing a decision tree, since it is less sensitive to changes in the class proportions of the nodes.



3.5.2 Build a decision tree

Decision trees can build complex (non-linear) decision boundaries by dividing the feature space into rectangles. However, we have to be careful since the deeper the decision tree, the more complex the decision boundaries becomes, which can easily result in overfitting.

Note: feature scaling is not a requirement for decision tree algorithms, and splits are easier to interpret with the original scale.

3.5.3 Random forest

Ensemble methods, like random forest, have gained huge popularity in machine learning applications during the last decade due to their good classification performance, robustnes towards overfitting, their scalability and their ease of use. Random forest belongs to the bagging methods of ensemble learning and can be considered an ensemble of decision trees.

The idea behind random forest is to average multiple (deep) decision trees that individually suffer from high variance. Thus building a more robust model that has better generalisation performance and is less susceptible to overfitting. Random forest can be summarised in four simple steps:

1. Draw a random bootstrap sample of size n (randomly choose n samples from training set with replacement)
2. Grow a decision tree from the bootstrap sample at each node
   (a) Randomly select d fatures without replacement
   (b) Split the node using the feature that provides the best split according to the objective function, for instance, maximising the information gain
3. Repeat the steps 1-2 k times (number of trees to be computed)
4. Aggregate the prediction by each tree to assign the class label by majority vote.

Random forest do not offer the same level of interpretability as decision trees. The big advantages of random forest:

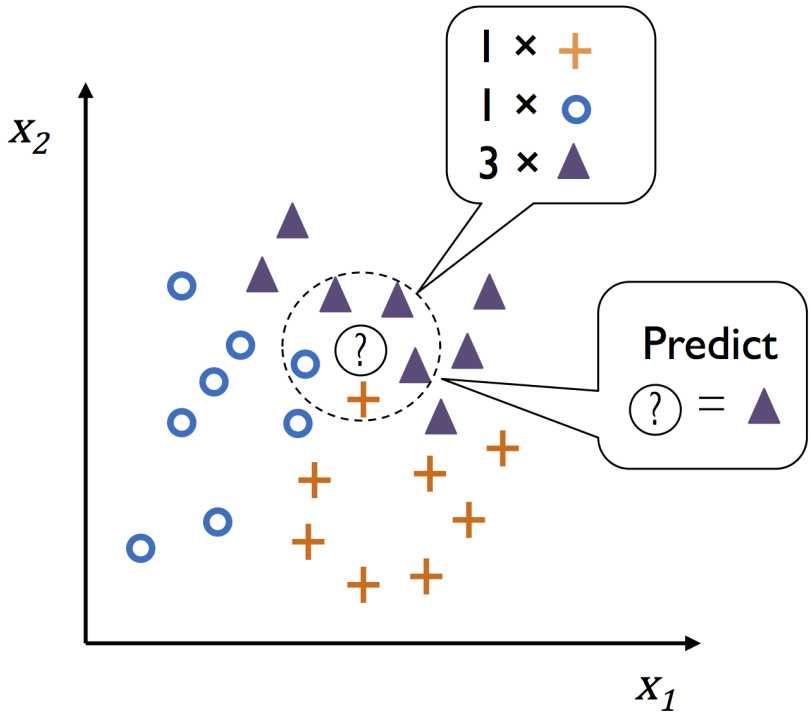
- No need to worry much about choosing good hyperparameters values
- Ensemble model is quite robust to noise from individual trees, which means no need to prune random forest.
- The only parameter that needs to be set: number of trees k
  - An increasing number of trees typically leads to increasing performance of the random classifier.
  - It also leeds to increasing expense in form of computational cost. To fix this use parameter `n_jobs` in `RandomForestClasifier` to set number of cores to be used. This parallelises model training using multiple cores of computer/processor. If *n_jobs*=`-1` then the number of jobs is set to the number of available cores.

You can also change the number of features d that is randomly chosen for each split, and also modify the size n of bootstrap sample. Decreasing n leads to diversity among individual trees, and increasing may increase the degree of overfitting.

3.6 K-nearest neighbour (KNN)

KNN is fundamentally different from the other supervised classifiers we have learned so far. KNN is a typical example of a **lazy learner** because it doesn't learn discriminative function from the training data, it rather memorizes the training dataset instead. KNN is fairly simple and can be summarized by the following steps:

1. Choose the number of k and a distance metric
2. Find k-nearest neighbours of the sample that is to be classified
3. Assign the class lable by majority vote



The main advantage to this classifier is that it immediately adapts as we collect new training data. The downsides to this is that the computational complexity grows with the number of samples in the training dataset, it cannot discard training samples since no training step is involved and the storage space can become a challenge if we are working with large datasets.

The right choice of k is crucial to find a good balance between overfitting and underfitting. The choosing of distance metric is also important and needs to be appropriate for the features in the dataset. Often Euclidean distance measure is used for real value samples, but this requires scaling of features. Minkowski distance is a generalisation of the Euclidean and Manhatten distance. Euclidean is plotted in a straight line, and Manhatten is plotted as steps.

Manhatten distance:

$$d(x^{(i)}, x^{(j)}) = \sum_k |x_k^{(i)} - x_k^{(j)}| \tag{3.20}$$

Eucledian distance:

$$d(x^{(i)}, x^{(j)}) = \sqrt{\sum_k |x_k^{(i)} - x_k^{(j)}|^2} \tag{3.21}$$

The Minkowski contains a parameter p. For $p = 1$ it becomes the Manhatten distance, and the Eucledian distance for $p = 2$:

$$d(x^{(i)}, x^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} - x_k^{(j)}|^p} \tag{3.22}$$

3.7 Parametric vs. non-parametric models

Parametric models estimates parameters from training data, which means it can learn a function that can classify new data points without requiring the original training dataset anymore. Perceptron, adaline, logistic regression and linear SVM are paramtric models.

Nonparametric models can't be characterized by a fixed set of parameters, and their number of parameters grows with the training data. Some examples are Decision Tree Classifier/Random Forest and kernel SVM. KNN belongs to subcategory of nonparametric models: instance-based learning.

Chapter 4

Building Good Training Datasets - Data Preprocessing

The quality of the data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore it is absolutely critical to ensure that we examine and preprocess a dataset before we feed it to a learning algorithm.

4.1 Dealing with missing data

It is not uncommon in real-world applications for our samples to be missing one or more values for various reasons. There can be an error in the data collection process, the measurements could not be applicable or the particular fields could have been left blank in a survey. The missing values are often represented by blank space, NaN-values or `NULL`. Most computational tools are not able to handle such missing values, and they should therefore not be ignored.

One of the easiest ways to deal with missing data is simply to remove the corresponding features (columns) or training examples (rows) from the dataset. Although it is convenient, there are certain disadvantages. If you remove to many samples it may be difficult to achieve a reliable analysis. If you remove to many feature columns you risk losing valuable information that classifiers need for discrimination between classes.

A common alternative is to use interpolation techniques for estimation of missing values from other traning samples in the dataset. One of the most common interpolation techinques is mean imputation, which replaces the missing values with the mean of the entire column. You can use `KNNImputer`, `SimpleImputer` or `IterativeImputer`, all from sccikit-learn. Which one to use may depend on the data, how much data that are missing and in which way the data are missing.

4.2 Handling categorical data

So far we have only been working with numerical values. However, it is not uncommon for real-world datasets to contain one or more categorical feature columns. We can distinguish between ordinal and nominal features. **Ordinal features** are categorical values that can be sorted or ordered, like XS $\rightarrow$ XL. **Nominal features** do not imply any order, like colours.

To change the categorical values to numeric values you can use:

- Python dictionaries
- `LabelEncoder` from scikit-learn
- `OneHotEncoder` from scikit-learn on nominal features
- one-hot encoding with `get_dummies` from pandas

4.3 Partitioning datasets into separate training and test datasets

When training you should randomly split X and y into seperate training and test datasets using `train_test_split`. Providing the class label array as an argument to stratify ensures that both training and test datasets have the same class proportions as the original dataset.

When diving a dataset you need to keep in mind not to withhold valuable information that the learning algorithm could benefit from and to be aware not to allocate to much information to the test data. However, the smaller the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset is all about balancing this trade-off, most commonly used splits are 60:40, 70:30, 80:20.

When done with model training and evaluation, it is common and recommended practice to retrain model on entire dataset, as it can improve the predictive performance of the model. Retraining could lead to worse generalization performance if the dataset is small and the test set contains outliers. Also, after refitting the model on the entire dataset, its performance can be evaluated only with new samples with known categories.

4.4 Bringing features onto same the scale

Feature scaling is a crucial step in our preprocessing pipeline that can easily be forgotten. Decision trees and random forest are two of the very few machine learning algorithms where do not need to worry about this.

There are two different approaches to bring features to same scale, normalisation and standardisation.

4.4.1 Normalisation

Normalisation is useful in situations where values in a bounded interval are needed, and is often referred to rescaling features to a range of [0,1] (a special case of min-max scaling),

$$x^{(i)}_{(norm)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

$x^{(i)}$: particular sample
$x^{(i)}_{(norm)}$: normalized particular sample
x_{min}: smallest value on feature column
x_{max}: largest value on feature column

4.4.2 Standardisation

Each feature takes mean value 0, with standard deviation 1.

$$x^{(i)}_{(std)} = \frac{x^{(i)} - \mu_x}{\sigma_x} \tag{4.2}$$

$x^{(i)}$: particular sample
$x^{(i)}_{(std)}$: normalized particular sample
mu_x: sample mean
σ_x: standard deviation

4.4.3 Min-Max scaling vs. Standardisation

Standardisation is often more practical for many machine learning algorithms, especially for optimization algorithms such as gradient descent. Many linear models, such as the logistic regression and SVM initialize the weights to 0 or small random values close to 0. Having feature columns with form of a normal distribution makes it easier for the algorithm to learn the weights. Standardisation maintains useful information about the outlier, but makes algorithms less sensitive to them in contrast to min-max scaling.

4.5 Selecting meaningful features

If a model performs much better on a training dataset than on the test dataset, this may be strong indication of overfitting. Overfitting means the model fits the parameters to closely with regard to the particular observations in the training dataset, if the model is to complex for the given training data, if the model does not generalise well to new data or the model has high variance. Some remedies for reducing generalisation error are:

- Collect more training data
- Introduce a penalty for complexity through regularisation
- Choose a simpler model with fewer parameters
- Reduce the dimensionality of the data through feature selection

4.5.1 L1 and L2 regularization as penalties against model complexity

L2 regularisation is one approach to reduce the complexity of a model by penalizing large individual weights. The definition norm of weight vector w is as follows:

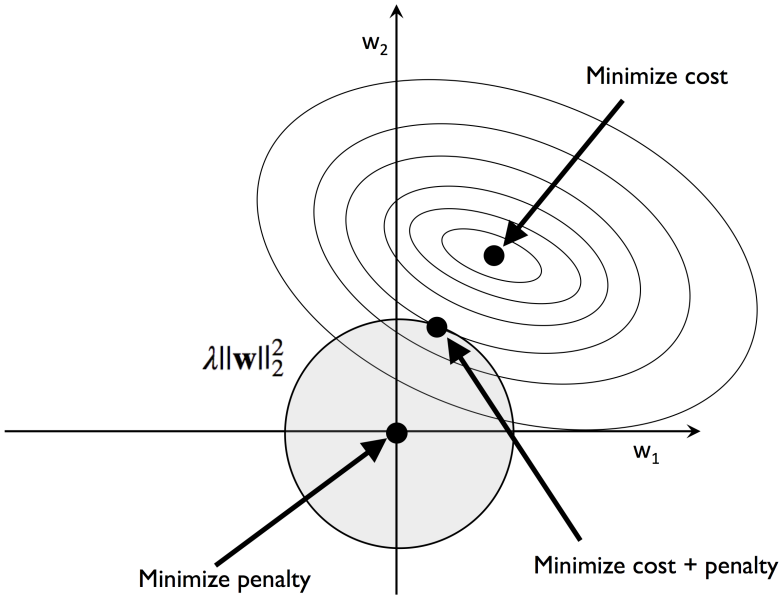
$$L2 : \lambda \|w\|_2^2 = \lambda \sum_{j=1}^{m} w_j^2 \tag{4.3}$$

Another approach to reduce the model complexity is the related **L1 regularisation**:

$$L1 : \lambda \|w\|_1 = \lambda \sum_{j=1}^{m} |w_j| \tag{4.4}$$

Geometric interpretation of L2 regularisation

L2 regularization adds a penalty term to the cost function that effectively results in less extreme weight values compared to a model with an unregularized cost function. We can think of regularization as adding a penalty term to the cost function to encourage smaller weights; in other words, regularization penalize large weights. Thus, by increasing the regularization strength using the regularization parameter λ, we shrink the weights toward zero and decrease the dependence of our model on the training data. Via λ we control how well we fit the training data, while keeping the weights small. **By increasing the value of λ we increase the regularization strength**.



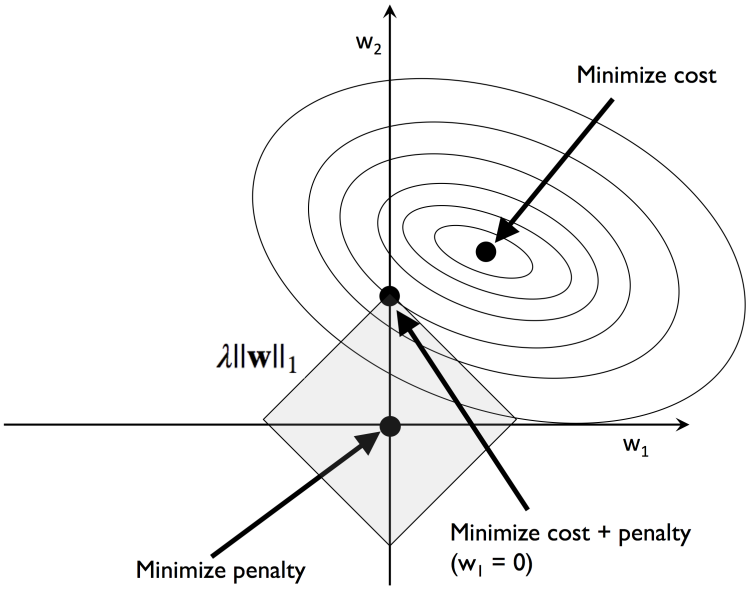
The cost need to be inside the grey circle, and it wants to be as near the "minimize cost"-point ass possible. Therefore it will try to by at the "minimize cost + penalty"-point.

Quadratic L2 regularization term is represented by the "shaded" ball. Our weight coefficients cannot exceed out regularization budget - the combination of the weight coefficients cannot fall outside the shaded area. On the other hand, we still need to minimize the cost function. Under the penalty constraint: choose the point where the L2 ball intersects with the contours of the unpenalized cost function. The larger the regularization parameter λ gets the faster the penalized cost grows, which leads to a narrower L2 ball. If we increase the regularization parameter towards infinity, the weight coefficients will become effectively zero, denoted the center of the ball.

Main message of the example, our goal is to minimize the sum of the unpenalized cost plus the penalty term, which can be understood as adding bias and preferring a simpler model to reduce the variance in the absence of sufficient training data to fit the model.

Sparse solution with L1 regularisation

The main concept behind L1 regularization is similar to that of L2. However, the L1 penalty is the sum of the absolute weight coefficients (remember that the L2 term is quadratic). L1 is represented as a diamond-shape budget.



The cost need to be inside the grey circle, and it wants to be as near the "minimize cost"-point ass possible. Therefore it will try to by at the "minimize cost + penalty"-point. This means that w_1 will often be 0.

The contours of the cost function touches the L1 diamond at w1 = 0. Since the contours of an L1 regularized system are sharp, it is more likely that the optimum - that is, the intersection between the ellipses of the cost function and the boundary of the L1 diamond - is located at the axes, which encourage sparsity (e.g. w_1 will often be 0).

L1 usually yield sparse feature vectors (solutions) and most feature vectors will be zero. This will act as a variable/feature selection.

4.5.2 Sequential feature selection algorithms

Dimensionality reduction using feature selection is an alternative to reduce the complexity of the model and avoid overfitting, and is especially useful for unregularised models. There are two main categories of dimensionality reduction techniques: feature selection and feature extraction. **Feature selection** selects a subset of the original features, whereas **feature extraction** derives information from the feature set to construct a new feature subspace.

Sequential feature selection algorithms are a family of greedy search algorithms and are used for reducing an initial d-dimensional feature space to a k-dimensional feature subspace where $k < d$. The motivation behind feature selection algorithms is to automatically select a subset of features that are most relevant to the problem, to improve computational efficiency, or to reduce the generalization error of the model by removing irrelevant features or noise, which

can be useful for algorithms that don't support regularization.

Greedy search algorithms make locally optimal choices at each stage of a combatorial search problem. **Exhaustive search algorithms** evaluate all possible combinations and are guaranteed to find the optimal solution. Greedy algorithms generally yield a suboptimal solution to the problem, in contrast to exhaustive search algorithms. However, in practice, an exhaustive search is often computationally not feasible. Greedy algorithms allow for a less complex, computationally more efficient solution.

Sequential Backward selection (SBS)

SBS aims to reduce the dimensionality of the initial feature subspace with a minimum decay in performance of the classifier to improve computational efficiency. In some cases, SBS can even improve the predictive power of the model if a model suffers from overfitting.

The idea behind SBS is quite simple:

- SBS sequentially removes features from the full feature subset d until the new feature subspace contains the desired number of k features ($k < d$)
- The criterion function j needs to be maximised, and it controls which feature is to removed at each stage.
- The criterion calculated by the criterion function can simply be the difference in performance on the classifier before and after the removal of a particular feature.
- Then, the feature to be removed at each stage can simply be defined as the feature that maximises this criterion (each stage we eliminate the feature that causes the least performance loss after removal).

The drop-out method

Using the dropout method, the importance of a feature is computed by:

- Removing feature f1 from the training and test set
- Train a new model using the same hyperparameters (as on the full model) on the training set
- Make predictions from the test set and compute the performance
- Compare that drop-out performance against the baseline performance (original test set where all the features were used). The importance is computed in the following way: importance of feature f1 == performance (baseline) - performance (drop-out f1). The higher the resulting value, the more important this feature is in the model and vice versa. This procedure needs to be repeated for each feature in the model.

4.6 Assessing feature importance with random forests

Random forest can be used to select relevant features from the dataset. Using random forest we can measure the feature importance as the averaged impurity decrease computed from all decision trees in the forest. It is done without making any assumptions about whether the data is linearly separable or not. `RandomForestClassifier` already collects the feature importance values, and they can be accessed through `feature_importance_`.

If two or more features are highly correlated one feature may be ranked very highly and the information of the other highly correlated features may not be fully captured.

Chapter 5

Compressing Data via Dimensionality Reduction

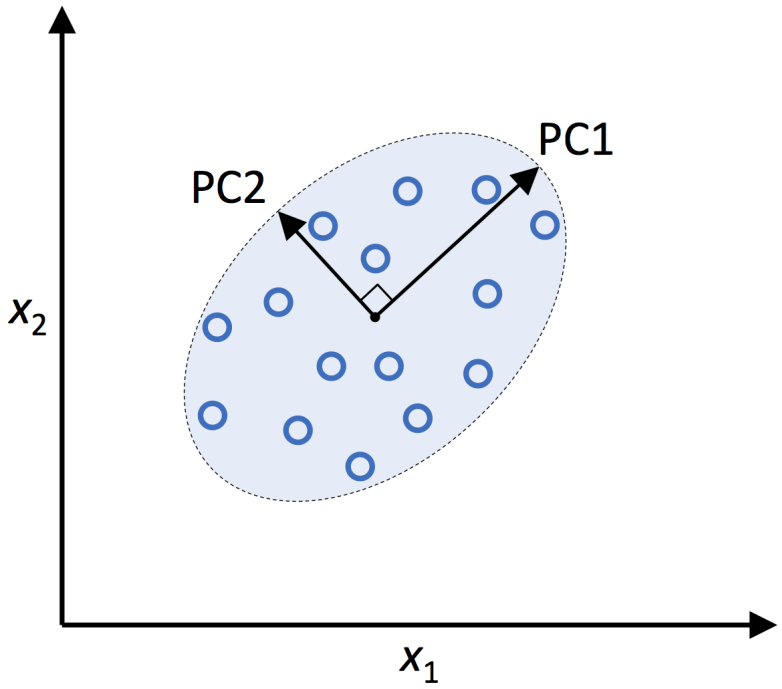
Chapter 4 was about different approaches for reducing the dimensionality of a dataset using different feature selection techniques. An alternative approach to feature selection for dimensionality reduction is **feature extraction**. This chapter is about three fundamental techniques that will help you to summarise the information content in the dataset by transforming it onto a new feature subspace of lower dimensionality than the original one.

5.1 Principal component analysis (PCA)

PCA is an unsupervised linear transformation technique that is widely used across different fields. It is used for **feature extracton and dimensionality reduction, exploratory data analysis and de-noising of signals** (and also faster training and visualization).

PCA helps us identify patterns in data based on the **covariance** (centered data) or **correlation** (standardised data) between features. It aims to find the directions of **maximum variance** in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one. The **orthogonal axes** (principal components) of the new subspace can be interpreted as the directions of maximum variance.

The hyperparameter for PCA controls the number of components/features/ dimensions to be extracted from the original data.



x_1 and x_2 are the original feature axes, and PC1 and PC2 are the principal components. Pc1 is in the direction of maximum variance, and PC2 is perpendicular (90°) to PC1. If we use PCA for dimensionality reduction, we construct a dxk-dimensional transformation matrix, W, that allows us to map the vector, x, the features of a training example, onto a new k-dimensional feature subspace that has fewer dimensions that the original d-dimensional feature space.

x: sample vector (1xd, where d is amount of features)
X: original dataset (nxd, where n is amount of samples)
Σ: covariance matrix (dxd)
W: dimensional transformation matrix (dxk, where k is amount of principal components)
X': XW (nxk)

5.1.1 Total and explained variance

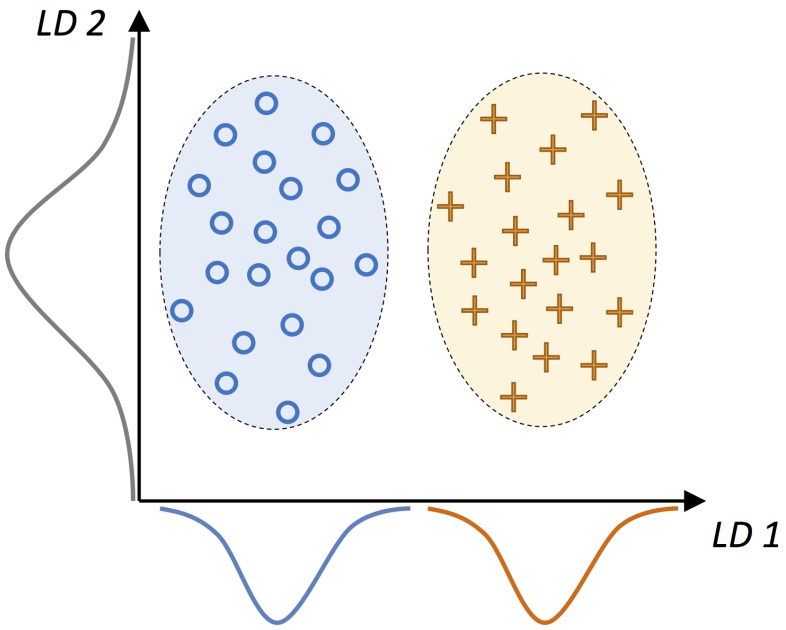
Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principial components) that contains most of the information (variance). The eigenvalues define the magnitude of the eigenvectors, so we sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of

their corresponding eigenvalues. PCA wil perform badly if the eigenvalues are roughly equal.

5.2 Linear discriminant analysis (LDA)

LDA can be used as a technique for feature extraction to increase the computational efficiency and reduce the degree of overfitting due to the curse of dimensionality in non-regularized models, in addition to vizualisation. The general concept behind behind LDA is very similar to PCA, but whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset, **the goal in LDA is to find the feature subspace that optimizes class separability**. In LDA we maximize between-class variance and minimize within-class variance.

LDA is **supervised**, while PCA is unsupervised. Both PCA and LDA are linear transformation techniques that can be used to reduce the number of dimensions in a dataset. Both decompose matrices into eigenvalues and eigenvectors, which will form the new lowerdimensionalal feature space.



A linear discriminant, as shown on the x-axis (LD 1), would separate the two normal distributed classes well. Although the exemplary linear discriminant shown on the y-axis (LD 2) captures a lot of variance, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.

One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the training examples are statistically independent of each other. However, even if one, or more of those assumptions is (slightly) violated, LDA for dimensionality reduction can still work reasonably well.

LDA and PCA is quite similarly implemented, but LDA takes class labels into account. The maximum number of features one can extract from the dataset is $c - 1$, where c is the number of classes in a dataset.

5.3 Kernelized PCA (KPCA)

KPCA relates to the concepts of kernel SVM and transforms data that is not linearly separable onto a new, lower dimensional subspace that is suitable for linear classifiers. In other words we perform nonlinear mapping via KPCA that transforms the data into higher-dimensional space. We then use standard PCA in this higher dimensional space to project the data back onto lower-dimensional space where the examples can be separated by linear classifier.

A downside to this is that it is computationally very expensive, and that is why we use the **kernel trick**(3.4.3). Using the kernel trick, we can compute the similarity between two high-dimensional feature vectors in the original feature space.

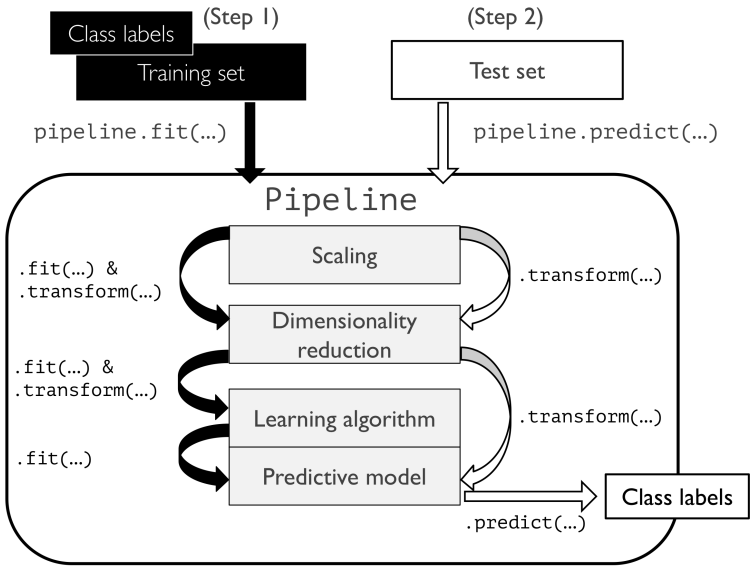
Chapter 6

Learning Best Practices for Model Evaluation and Hyperparamter Tuning

Now it is time to learn about the best practices of building good machine learning models by fine-tuning the algorithms and evaluating the performance of the models.

6.1 Pipelines

The `Pipeline` class in scikit-learn is an extremely handy tool that allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.



The `make_pipeline` function takes an unlimited number of scikit-learn transformers (objects that support the `fit` and `transform` methods as input), e.g. PCA, Standardscaler, logistical regression, followed by a scikit-learn estimator that implements the `fit` and `predict` methods, e.g. Perceptron.

6.2 Cross-validation

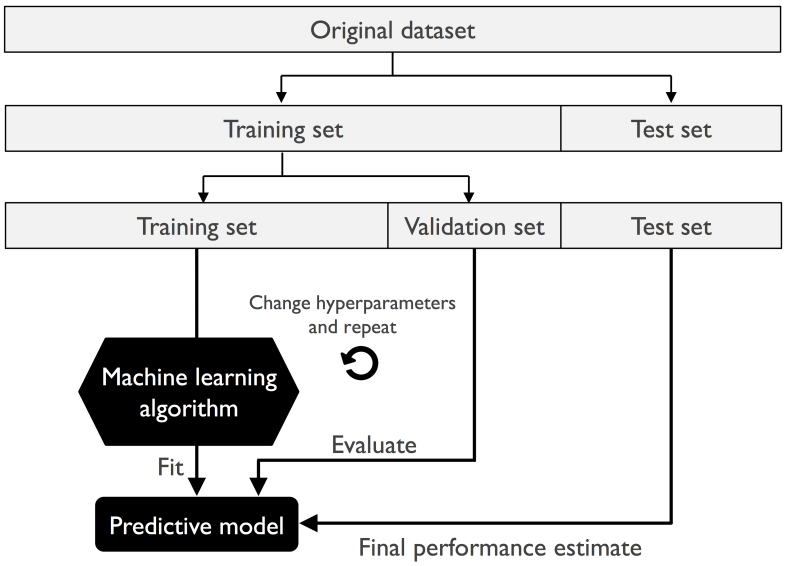
One of the key steps in building a machine learning model is to estimate its performance on data that the model has not seen before. To find an acceptable bias-variance trade-off, we need to evaluate our model carefully. Cross-validation can help us to obtain reliable estimates of the models generalization performance, that is, how well the model performs on unseen data.

6.2.1 The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is holdout cross-validation. Using the holdout method, we split our initial data set into separate training and test datasets, the former is used for model training, and the latter is used to estimate its generalization performance. However, in typical machine learning applications, we are also interested in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called model selection, with the name refering to a given classification problem for which we want to select the optimal values of tuning parameters (also called hyperparameters). However, if we reuse the same test data over and over again

during model selection, it will become part of our training data and thus the model will be more likely to overfit. A better way of using the holdout method for model selection is to separate the data into three parts: a training dataset, a validation dataset, and a test dataset. The training dataset is used to fit the different models, and the performance on the validation dataset is then used for the model selection.

A disadvantage of the holdout method is that the performance estimate may be very sensitive to how we partition the training dataset into the training and validation subsets; the estimate will vary for different examples of the data.



6.2.2 K-fold cross-validation

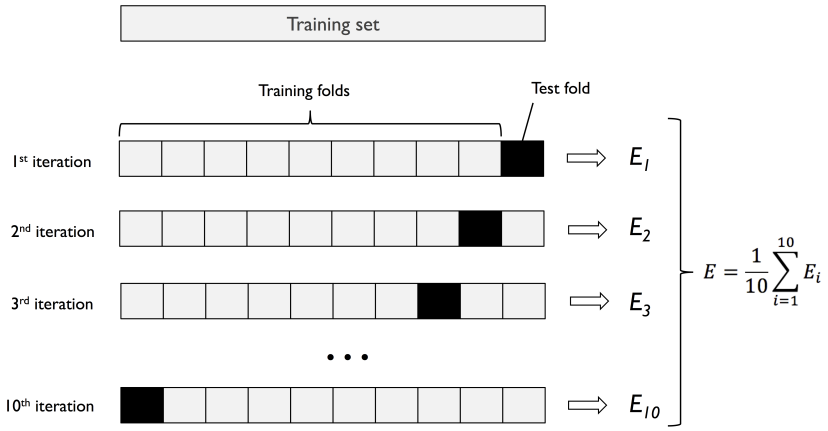
In k-fold cross-validation, we randomly split the training dataset into k folds without replacement, where $k-1$ folds are used for the model training, and one fold is used for performance evaluation. This procedure is repeated k times so that we obtain k models and performance estimates.

We then calculate the average performance of the models based on the different, independent test folds to obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method. Typically, we use k-fold cross-validation for model tuning, that is, finding the optimal hyperparameter values that yield a satisfying generalization performance, which is estimated from evaluating the model performance on the test folds.

Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training dataset and obtain a final performance estimate using the independent test dataset. The rationale behind fitting a model to the whole training dataset after k-fold cross-validation is that providing more

training examples to a learning algorithm usually results in a more accurate and robust model.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that each example will be used for training and validation (as part of a test fold) exactly once, which yields a lower-variance estimate of the model performance than the holdout method.



Ron Kohavi suggests that 10-fold cross-validation offers the best tradeoff between bias and variance. If we are working with small datasets, it might be better to increase the value of k, since more data will be used for training. This will however increase the runtime and yield estimates with higher variance.

Stratified k-fold cross-validation

Stratifies k-fold cross validation is a slight improvement over the standard k-fold cross-validation approach, which can yield better bias and variance estimates, especially in cases of unequal class proportions. In stratified cross-validation, the class label proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset.

6.3 Learning and validation curves

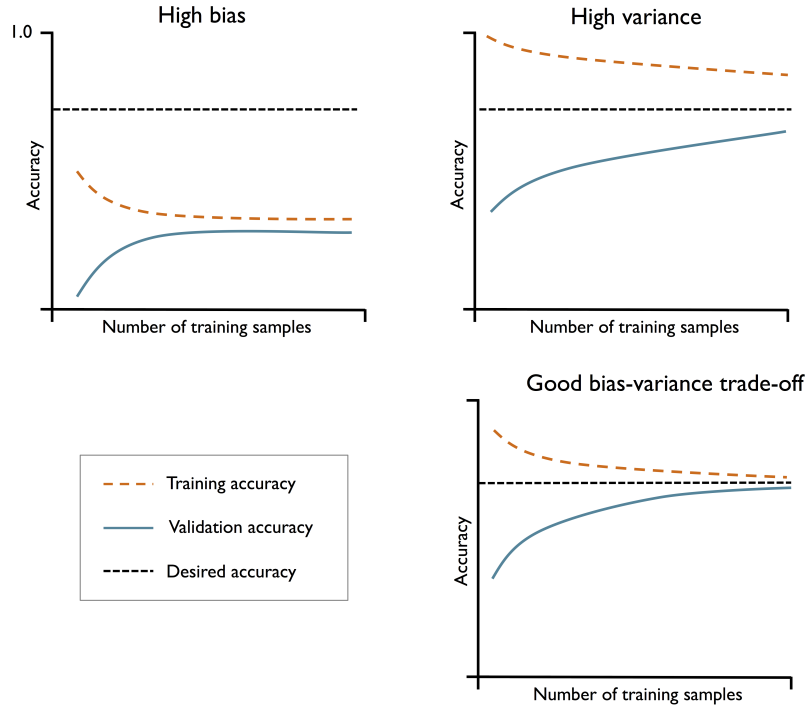
Learning curves and validation curves are powerful diagnostic tools that can help us to improve the performance of a learning algorithm.

6.3.1 Diagnosing bias and variance problems with learning curves

If a model is too complex for a given dataset (there are to many degrees of freedom or parameters) the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training

examples to reduce the degree of overfitting. However, in practice, it can often be very expensive or simply not feasible to collect more data.

By plotting the model training and validation accuracies as functions of the training dataset size, we can easily detect whether the model suffers from high varance or high bias, and whether the collection of more data could help to address this problem.

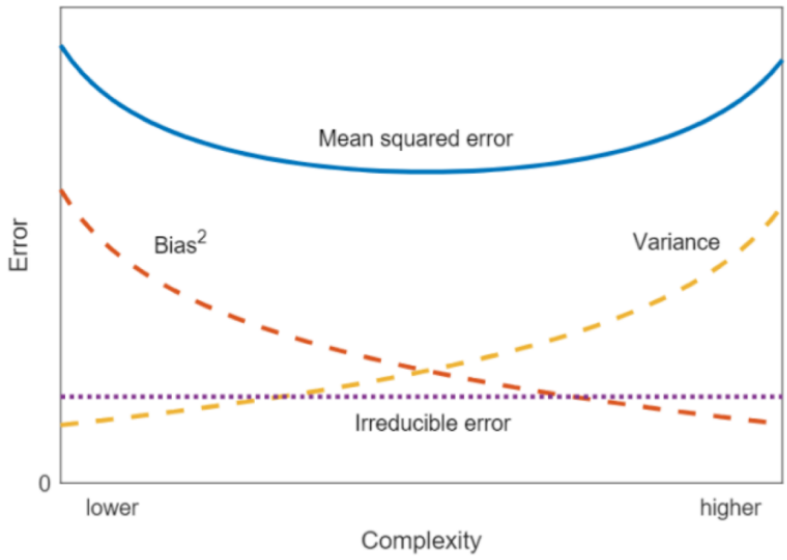


The graph in the upper-left shows a model with high bias. This model has both low training and cross-validation accuracy, which indicates that it underfits the training data. Common ways to address this issue are to increase the number of parameters of the model, for example, by collecting or constructing additional features, or by decreasing the degree of regularization, for example, in SVM or logistic regression classifiers.

The graph in the upper-right shows a model that suffers from high variance, which is indicated by the large gap between the training and cross-validation accuracy. To address this problem of overfitting, we can collect more training data, reduce the complexity of the model, or increase the regularization parameter. For unregularized models, it can also help to decrease the number of features via feature selection or feature extraction to decrease the degree of overfitting.

6.3.2 Addressing over- and underfitting with validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, we vary the values of the model parameters, for example, the inverse regularization parameter, C, in logistic regression.



6.4 Grid search

We have two types of parameters: those that are learned from the training data, like, the weights in logistic regression, and the parameter of a learning algorithm that are optimized separately. The latter are the tuning parameters (or hyperparameters) of a model, for example, the regularization parameter in logistic regression or the depth parameter of a decision tree. A popular hyperparameter optimization technique is grid search, which can further help to improve the performance of a model by finding the optimal combination of hyperparameter values.

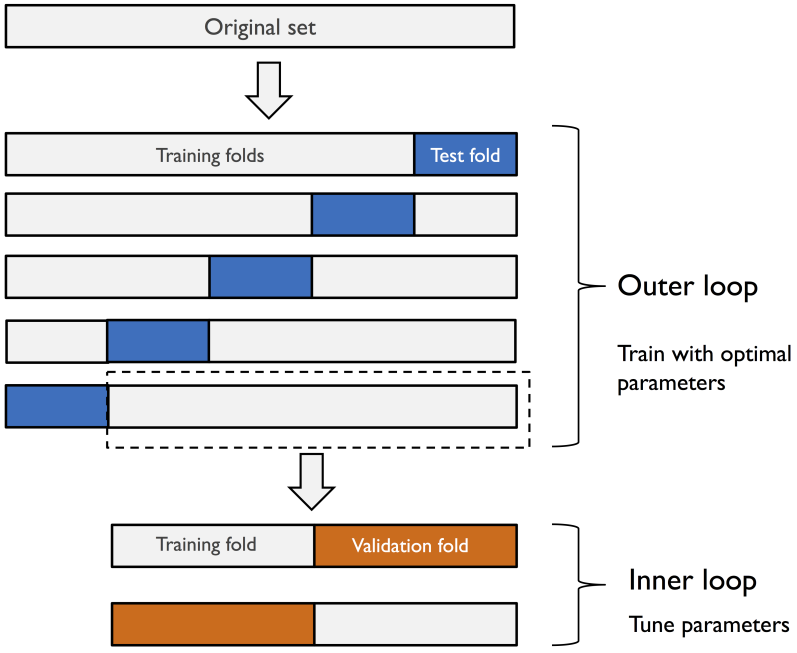
The grid search approach is quite simple: it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the computer evaluates the model performance for each combination to obtain the optimal combination of values from this set.

Grid search is computationally very expensive. An alternative approach for sampling different parameter combinations using scikit-learn is **randomized search**. Randomized search usually performs about as well as grid search but is much more cost- and time-effective.

6.4.1 Nested cross-validation

Using k-fold cross-validation in combination with grid search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameter values, as we saw in the previous subsection. If we want to select among different machine learning algorithms, another recommended approach is nested cross-validation.

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to train, evaluate and select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance over again. The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and use it on unseen data.

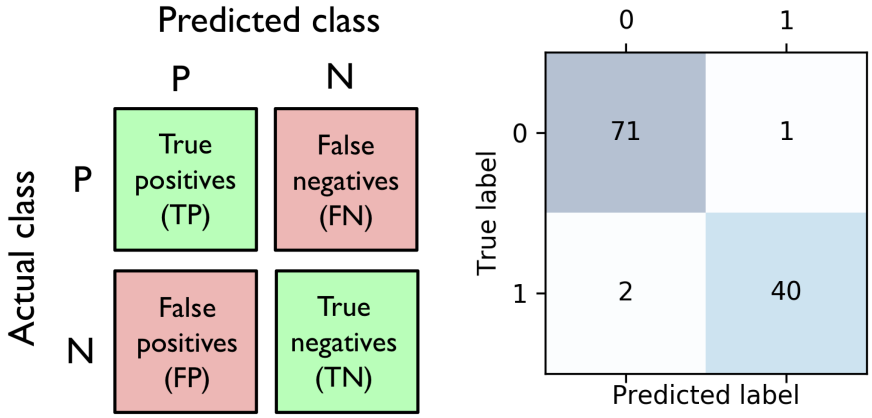


6.5 Performance evaluation metrics

In the previous sections and chapters, we evaluated different machine learning models using the prediction accuracy, which is a useful metric with which to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a models relevance, such as precision, recall and the F1 score.

6.5.1 Confusion matrix

The confusion matrix is a matrix that lays out the performance of a learning algorithm, by comparing the numbers of predictions for each class that are correct and those that are incorrect. It is simply a square matrix that reports the counts of the true positive (TP), true negative (TN), false positive (FP) and false negative (NP) predictions of a classifier, as shown in the following figure:



The accuracy is the simple ratio between the number of correctly classified points to the total number of points.

6.5.2 Optimizing the precision and recall of a classification model

Both the prediction error (ERR) and accuracy (ACC) provide generall information about how many examples are misclassified. The error can be understood as the sum of all false predicitons divided by the number of total predictions, and the accuracy is calculated as the sum of correct predicitons divided by the total number of predicitons:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN} \tag{6.1}$$

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR \tag{6.2}$$

The **true positive rate (TPR)** and **false positive rate (FPR)** are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} \tag{6.3}$$

$$TPR = \frac{TP}{N} = \frac{TP}{FN + TP} \tag{6.4}$$

The performance metrics **precision (PRE)** and **recall (REC)** are related to those TP and TN rates, and in fact, REC is synonymous with TPR:

$$PRE = \frac{TP}{TP + FP} \tag{6.5}$$

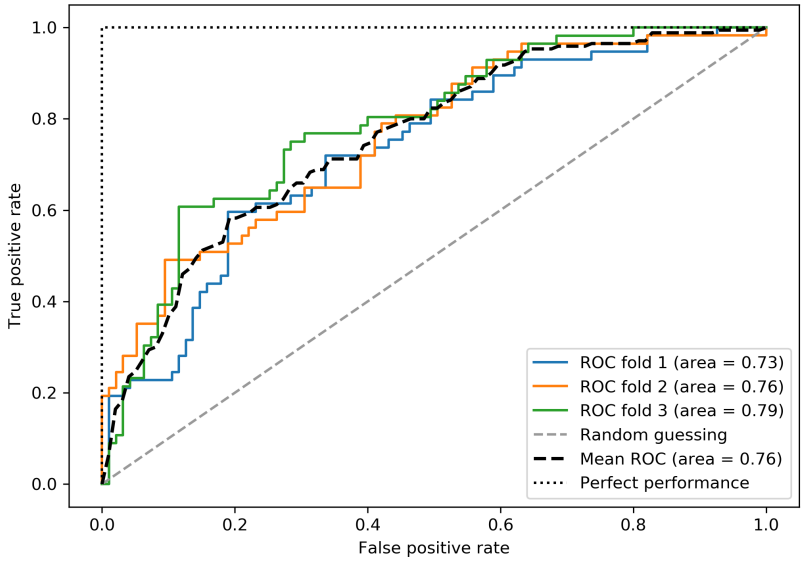
$$REC = TPR = \frac{TP}{N} = \frac{TP}{FN + TP} \tag{6.6}$$

To balance the up- and down-sides of optimizing PRE and REC, often a combinaton of PRE and REC is used, the so called F1 score:

$$F1 = 2\frac{PRE \text{ X } REC}{PRE + REC} \tag{6.7}$$

6.5.3 Receiever operating characteristic (ROC)

Receiver operating characteristic (ROC) graphs are useful tools to select models for classification based on their performance with respect to the FPR and TPR, which are computed by shifting the decision threshold of the classifier. The diagonal of the ROC graph can be interpreted as random guessing, and classification models that fall below the diagonal are considered worse than random guessing. A perfect classifier would fall into the top-left corner of the graph with a TPR of 1 and a FPR of 0. Based on the ROC curve, we can then compute the so-called ROC area under the curve (ROC AUC) to characterize the performance of a classification model.



6.5.4 Scoring metrics for multiclass classification

The scoring metrics that we have discussed so far are spesific to binary classification systems. Scikit-learn implements micro and macro averaging methods to extend those scoring methods to multiclass problems via the one-vs-all (OvA) classification.

F1 micro:

$$PRE = \frac{\sum TP}{\sum TP + \sum FP} \tag{6.8}$$

$$REC = TPR = \frac{TP}{N} = \frac{\sum TP}{\sum FN + \sum TP} \tag{6.9}$$

F1 macro:

$$PRE = \sum \frac{TP}{TP + FP} \tag{6.10}$$

$$REC = TPR = \sum \frac{TP}{N} = \sum \frac{TP}{FN + TP} \tag{6.11}$$

Micro-averaging is useful if we want to weight each instance or prediciton equally, whereas macro-averaging weights all classes equally to evaluate the overal performance of a classifier with regard to the most frequent class labels. Macro is best for unbalanced data.

6.5.5 Dealing with class imbalance

Class imbalance is a quite common problem when working with real-world data: examples from one class or multiple classes are over-represented in a dataset. If the positive class have 90% of the examples in the dataset we could just guess that all the samples are positive and we'll get an accuracy score equal to 90%.

Aside from evaluating machine learning models, class imbalance influences a learning algorithm during model fitting itself. Since machine learning algorithms typically optimize a reward or cost function that is computed as a sum over the training examples that it sees during fitting, the decision rule is likely going to be biased toward the majority class.

In other words, the algorithm implicitly learns a model that optimizes the predictions based on the most abundant class in the dataset, in order to minimize the cost or maximize the reward during training.

One way to deal with imbalanced class proportions during model fitting is to assign a larger penalty to wrong predictions on the minority class via scikit-learn, we set class weight parameter to `class_weight = 'balanced'`, which is implemented for most classifiers.

Other popular strategies for dealing with class imbalance include upsampling the minority class, downsampling the majority class, generate synthetic samples or to use macro settings.

Chapter 7

Combining Different Models for Ensemble Learning

In this chapter we will explore different methods for construction a set of classifiers that can often have a better predicitve performance than any of its individual members.

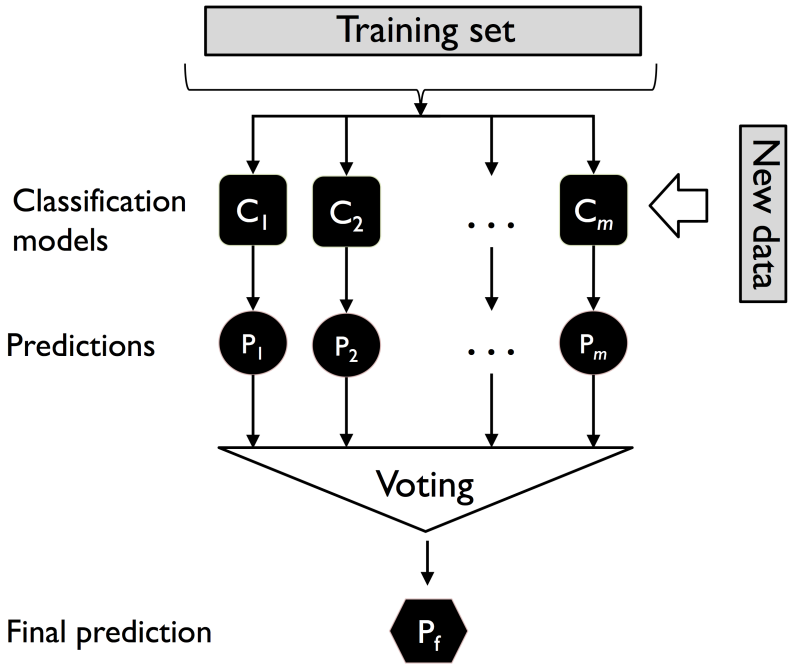
7.1 Ensembles

The goal of the ensemble methods is to combine classifiers into a metaclassifier that has better generalizatioon performance than each individual classifier alone. For example, assuming that we collected predicitons from 10 experts, ensemble methods would allow us to strategically combine those predictions to come up with a predictions that was more accurate and robust than the predictions by each individual expert. In short words: many models in combination can perform better than one.

7.1.1 Majority voting

Majority voting means that we select the class label that has been predicted by the majority of the classifiers. Majority voting refers to binary class, but it is easy to generalize it to multiclass setting using **plurality voting**. Here, we select the class label that recieved the most votes (the mode).

Using the training dataset, we start by training m different classifiers. Depending on the technique, the ensemble can be built from different classifications algorithms. Alternatively, we can also use the same basic classification algorithm, fitting different subsets from the training dataset, like in random forest which combines different decision tree classifiers.



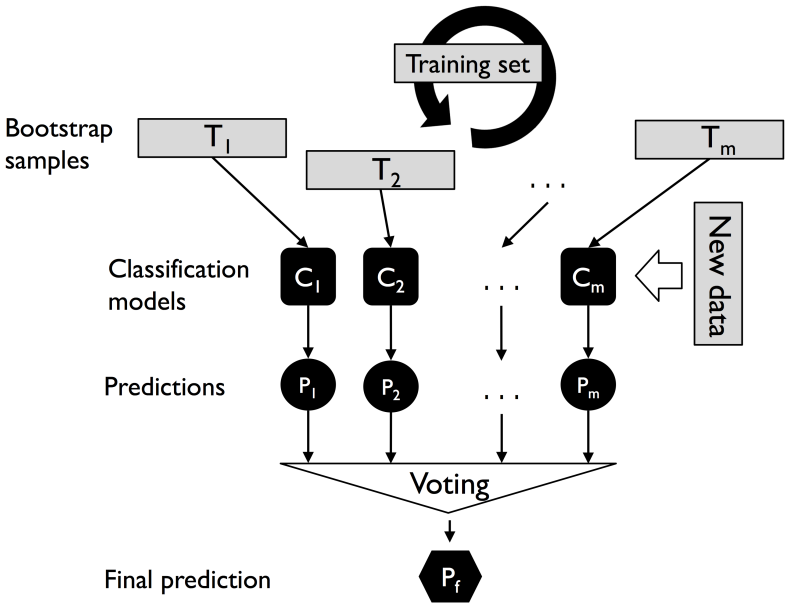
To predict a class label via simple majority or plurality voting we can combine the predicted class lables of each individual classifier C_j and select the class label $\hat{y}$ that receives the most votes:

$$\hat{y} = mode\{C_1(x), C_2(x), \ldots, C_m(x)\} \tag{7.1}$$

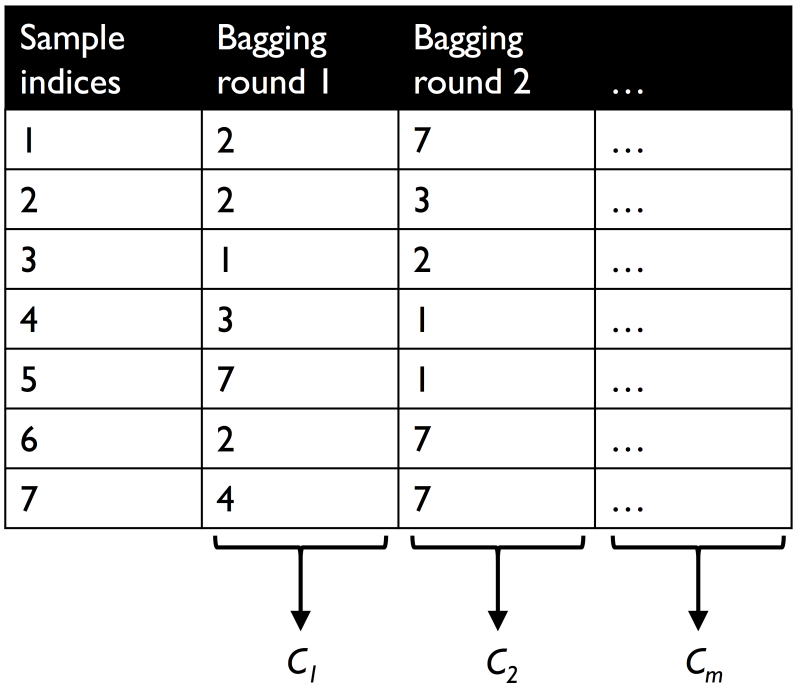
Majority voting is less prone to overfitting than individual models.

7.2 Bagging

Bagging is an ensemble learning technique that is closely related to the majority vote classifier, and can help **reduce variance**. However, instead of using the same training dataset to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training set, which is why bagging is also known as bootstrap aggregating.



To provide a more concrete example of how the bootstrap aggregating of a bagging classifier works, let's consider the example shown in the following figure. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier, C_j, which is most typically an unpruned decision tree.



As you can see from the previous illustration, each classifier receives a random subset of examples from the training dataset. We denote these random samples obtained via bagging as *Bagging round* 1, *Bagging round* 2, and so on. Each subset contains a certain portion of duplicates and some of the original examples do not appear in a resampled set at all due to sampling with replacement. Once the individual classifiers are fit to the bootstrap samples, the predictions are combined using **majority vote**.

7.3 Adaptive boosting (AdaBoost)

AdaBoost is the most common implementation of boosting. In boosting, the ensemble consists of very simple base classifiers, also often referred to as **weak learners**, which often only have a slight performance advantage over random guessing. A typical example of a weak learner is a decision tree stump. The key concept behind boosting is to **focus on training examples that are hard to classify**, that is, to let the weak learners subsequently learn from misclassified training examples to improve the performance ensemble.

In contrast to bagging, the initial forrmulation of the boosting algorithm uses random subsets of training examples drawn from the training dataset, without replacement. The original boosting algorithm can be summarized in the following four key steps:

1. Draw a random subset (sample) of training examples, d_1, without replace-

ment from the training set, D, to train the weak learner, C_1.

2. Draw a second random training subset, d_2, without replacement from the training dataset and add 50 percent of the examples that were previously misclassified to train a weak learner, C_2.

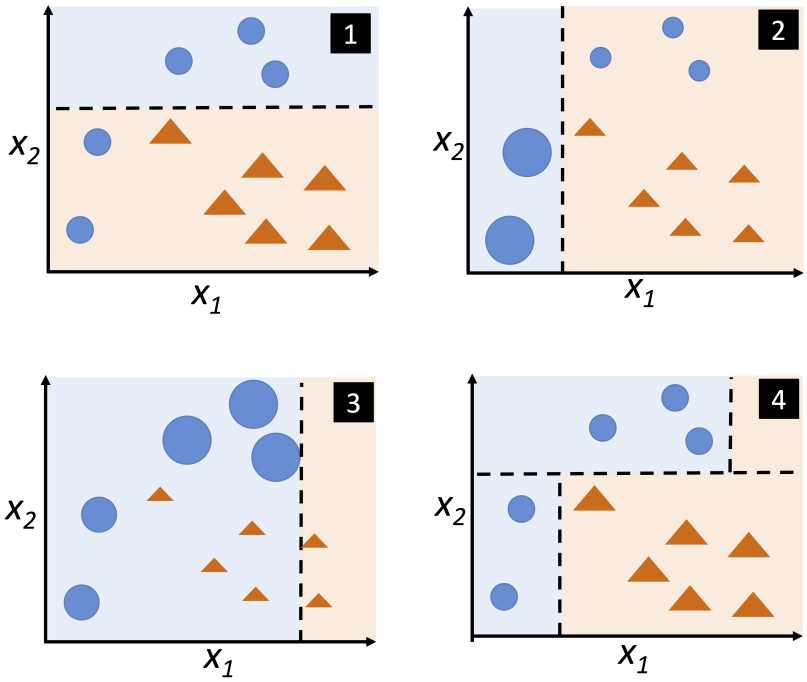
3. Find the training examples, d_3, in the training dataset, D, which C_1 and C_2 disagree upon, to train a third weak learner, C_3.

4. Combine the weak learners C_1, C_2 and C_3 via majority voting.

thee weight in each training example is assigned randomly to each example. The coefficints are assigned to the weak classifiers in the ensemble based on their accuracy in classifying the training examples.

Boosting can lead to a decrease in bias as well as variance compared to bagging models. In practice, however, bosting algorithms such as AdaBoost are also known for their high variance, that is, the tendency to overfit the training data.

In contrast to the original boosting procedure, AdaBoost uses the complete training dataset to train the weak learners, where the training examples are reweighted in each iteration to build a strong classifier that learns from the mistakes of the previous weak learners in the ensemble.



Subfigure **1** represents a training dataset for binary classification where all training examples are assigned equal weights. Based on this training dataset, we train a decision stump (shown as a dashed line) that tries to classify the examples

of the two classes (triangles and circles), as well as possibly minimizing the cost function (or the impurity score in the special case of decision tree ensembles).

For the next round (Subfigure **2**), we assign a larger weight to the two previously misclassified examples. Furthermore, we lower the weight of the correctly classified examples. The next decision stump will now be more focused on the training examples that have the largest weights: the training examples that are supposedly hard to classify. The weak learner shown in subfigure **2** misclassifies three different examples from the circle class, which are then assigned a larger weight, as shown in subfigure **3**.

Assuming that our AdaBoost ensemble only consist of three rounds of boosting, we then combine the three weak learners trained on different reweighted training subsets by a weighted majority vote as shown in subfigure **4**.

Index	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	No	0.167
8	8.0	1	0.1	-1	No	0.167
9	9.0	1	0.1	-1	No	0.167
10	10.0	-1	0.1	-1	Yes	0.072

The table illustrates a more concrete example using a training dataset consisting of 10 training examples.

Chapter 10

Predicting Continuous Target Variables with Regression Analysis

Regression models are used to predict target variables on a continuous scale, which makes them attractive for adressing many questions in science.

10.1 Linear regression

The goal of linear regression is to model the relationship between one or multiple features and a continuous target variable. In contrast to classification (another subcategory of supervised learning) regression analysis aims to predict outputs on a continuous scale rather than categorical class labels. The dependent vaiable is the variable being predicted, while the independent variable is a predictor value.

10.1.1 Simple linear regression

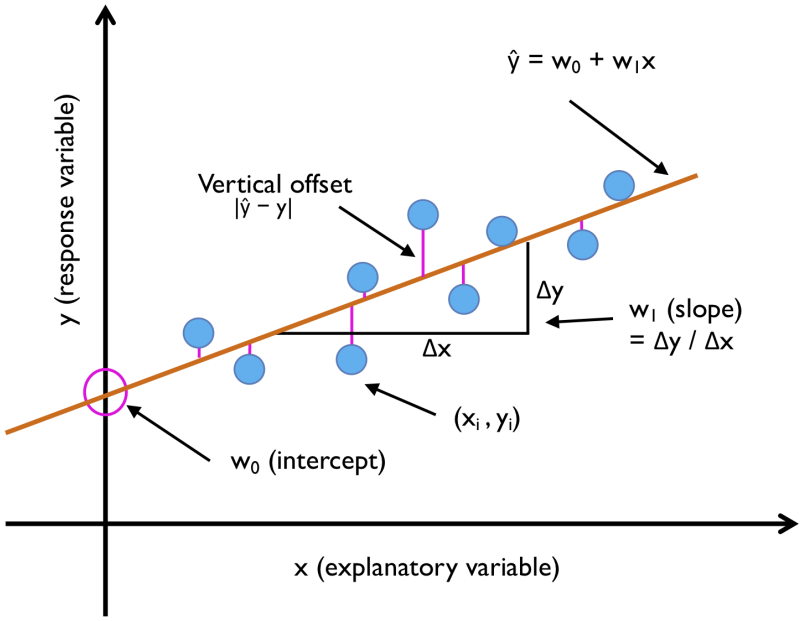
The goal of (univariate) linear regression is to model the relationship between a single feature (explanatory variable, x) and a continuous-valued target (response variable, y). The equation of a linear model with one explanatory variable is defined as follows:

$$y = w_0 + w_1 x \tag{10.1}$$

w_0: y-axis intercept
w_1: weigth coefficient of the explanatory variable
y: response variable
x: explanatory variable

The method is to learn the weights of the linear equation to describe the relationship between the explanatory variable and the target variable, which

then can be used to predict the responses of new explanantory variables that were not a part of the training dataset. Based on the equation, linear regression can be understood as finding the best-fitting straight line (**regression line**) through the training examples. The virtual line from regression line to the training examples are the **offsets** or **residuals** (the error of our prediction). Residuals is the difference between the predicted and actual values, error is the difference between the independent end dependent variables.



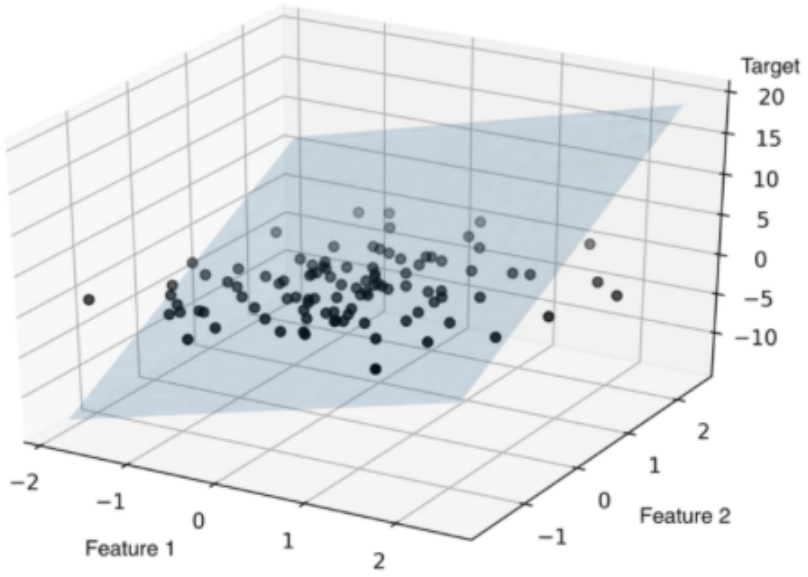
10.1.2 Multiple linear regression

We can generalize the linear regression model to multiple explanatory variables.

$$y = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \sum_{i=0}^{n} w_i x_i = w^T x \tag{10.2}$$

w_0: y-axis intercept
x_0: equals 0
w_m: weigth coefficient of the explanatory variable
y: response variable
x: explanatory variable

The following figure shows how the two-dimensional, fitted hyperplane of a multiple linear regression model with two features could look:



10.1.3 Exploratory data analysis (EDA)

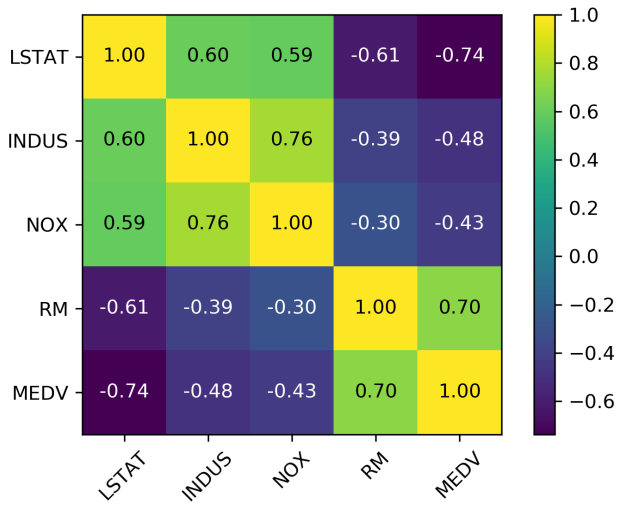
EDA is an important recommended first step prior to the training of a machine learning model. The graphical EDA toolbox may help to visually detect the presence of outliers, the distribution of the data, and the relationships between features.

10.1.4 Correlation matrix

A correlation matrix quantify and summarize linear relationships between variables. A correlation matrix is a rescaled version of the covariance matrix.

The correlation matrix is a square matrix that contains the Pearson product-moment correlation coefficient (often abbreviated as Pearsons's r), which measures the linear dependence between pairs of features. The correlation coefficients are in the range -1 to 1. Two features have a perfect positive correlation if $r = 1$, no correlation if $r = 0$, and a perfect negative correlation if $r = -1$.

To fit a linear regression model, we are interested in those features that have a high correlation with our target variable.



10.2 Ordinary least square linear regression (OLS)

OLS is a method for determining the best-fit line by minimizing the sum of the squared residuals. Differently put: it is used to estimate the parameters of the linear regression to minimize the sum of the squared vertical distances (residuals or errors) from the traning examples. The cost function $J(w)$ for OLS is the same as for adaline.

$$J(w) = \frac{1}{2}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2. \tag{10.3}$$

Essentially OLS can be understood as adaline without the unit step function so that we obtain continuous target variables instead of class labels.

10.3 RANSAC

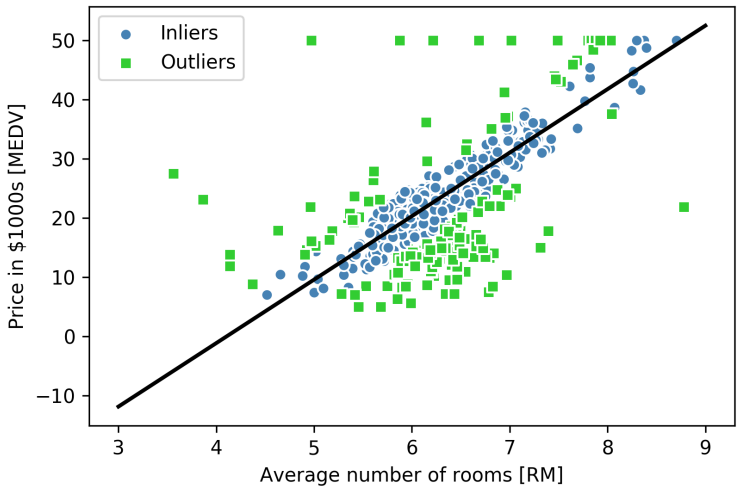
Linear regression models can be heavily impacted by the precense of outliers. An alternative to throwing them out, is a robust method of regression using the random sample consensus (RANSAC) algorithm, which fits a regression model to a subset of the data, the so-called inliers. The iterative RANSAC can be summarized as follows:

1. Select a random number of examples to be inliers and fit the model.
2. Test all other data points against the fitted model and add those points that fall within a user given-tolerance to the inliers.

3. Refit the model using all inliers.

4. Estimate the error of the fitted modell versus the inliers

5. Terminate the algorithm if the performance meets a certain user-defined threshold or if a fixed number of itrations were reached; go back to step 1 otherwise.



The linear regression model was fitted on the detected set of inliers. Using RANSAC, we reduce the potential effect of the outliers in this dataset, but we do not know whether this approach will have a positive effect on the predictive performance for unseen data or not.

10.4 Evaluating the performance of linear regression

Residual plots are a commonly used graphical tool for diagnosing regression models. They can help detect nonlinearity and outliers, and check whether the errors are randomly distributed.

In the case of perfect, prediction the residuals would be exactly zero, which we will never encounter in realistic and practial applications. For a good regression model we would expect the errors to be randomly distributed and the residuals to be randomly scattered around the centerline. If we see patterns in the residual plot, it means our model is unable to capture some explanatory information. Points with large deviation from the centerline are outliers.

Another useful measure of a models performance is the **mean squared error (MSE)**, which is the average of SSE cost that we minimize to fit the

linear regression model. It is useful for comparing different regression models or for tuning parameters via grid search and cross validation:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2 \tag{10.4}$$

If the MSE is much larger on the test set than it was on the training set, it is an indicator that the model is overfitting the training data.

The MSE is unbounded, which means the interpretation of the MSE depends on the dataset and feature scaling. Therefore, it may be useful to report the **coefficient of determination (R^2)**, which is the standardized version of the MSE. R^2 is used to measure the proportion of the variation in the dependent variable that is explained by the independent variable. **High R^2 means the modell explains a lot of variance.**.

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{MSE}{Var(y)} \tag{10.5}$$

SSE: the sum of squared errors
SST: total sum of squares (the variance of the response)
y: the response

However, R^2 does not describe how the models adapt locally, how outliers are handled or if the possible model assumptions are met. It is also vulnerable because of its basis in SST.

10.5 Using regularized method for regression

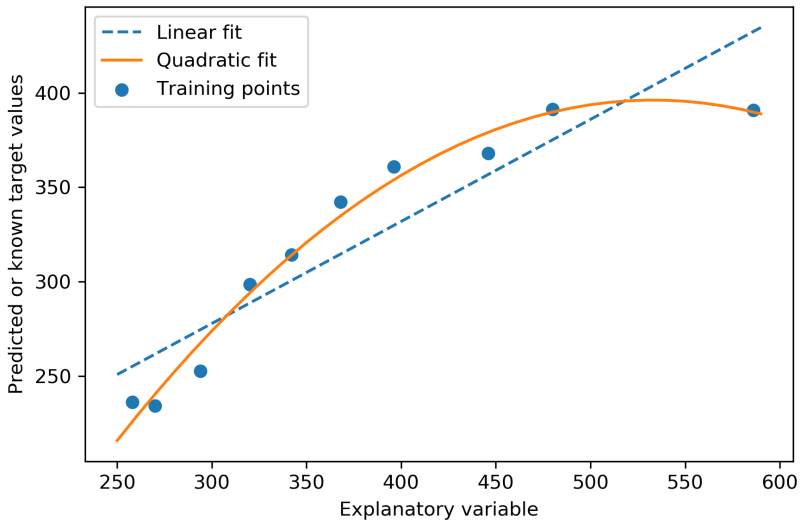
As said earlier, regularization is one approach of tackling the problem of overfitting by adding additional information, and thereby shrinking the parameter values of the model to induce a penalty against complexity. The most popular approaches to regularized linear regression are the so-called ridge regression (L2), least absolute shrinkage and selection operator (LASSO, L1) and elastic net. OLS is also used. Elastic net is a combination of L1 and L2.

10.6 Polynomial regression

In the previous sections we assumed a linear relationship between explanatory and response variables. One way to account for the violation of linearity assumption is to use a polynomial regression model by adding polynomial terms:

$$y = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d \tag{10.6}$$

Here, d denotes the degree of the polynomial. Although we can use polynomial regression to model a nonlinear relationship, it is still considered a multiple linear regression model because of the linear regression coefficients, w.



The polynomial fit (quadratic fit/second degree) captures the relationship between the response and explanatory variables much better than the linear fit. However, be aware that adding more and more polynomial features increase the complexity of a model and therefore increases the chance of overfitting. Thus, in practice it is always recommended to evaluate the performance of the model on a separate test dataset to estimate the generalization performance.

10.7 Dealing with nonlinear relationships using random forest

A random forest, which is an ensemble of multiple decision trees, can be understood as the sum of piecewise linear functions, in contrast to the global linear and polynomial regression models that we discussed previously. In other words, via the decision tree algorithm, we subdivide the input space into smaller regions that become more manageable.

10.7.1 Decision tree regression

An advantage of the decision tree algorithm is that it does not require any transformation of the features if we are dealing with nonlinear data, because decision trees analyze one feature at a time, rather than taking weighted combinations into account (likewise, normalizing or standardizing features is not required for decision trees). The leaf node predicts the target variable for a given observation.

When we used decision trees for classification, we defined entropy as a measure of impurity to determine which feature split maximizes the **information gain (IG)**, which can be defined as follows for a binary split:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}) \tag{10.7}$$

x_i: the feature to perform the split
N_p: number of training examples in the parent node
I: impurity function
D_p: the subset of training examples at the parent node
D_{right}: the subset of training examples at the right child node after the split
D_{left}: the subset of training examples at the left child node after the split

We want to find the feature split that reduces the impurities in the child nodes the most. We cannot use Gini or entropy as measures of impurity for regression, we need an impurity metric that is suitable for continuous variables, so we define the impurity measure of a node t, as the MSE instead:

$$I(t) = MSE(t) \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{t}_t)^2 \tag{10.8}$$

N_t: the number of training examples at node t
D_t: the training subset at node t

Tree based regression has a limited number of possible predictions. Modeling is done by sequential splitting of the target into a discrete number of levels along when the features are split. Any prediction will be done on one of these discrete levels.

10.7.2 Random forest regression

A random forest usually has a better generalization performance than an individual decision tree due to randomness, which helps to decrease the model's variance. Other advantages of random forests are that they are less sensitive to outliers in the dataset and don't require much parameter tuning. The only parameter in random forests that we typically need to experiment with is the number of trees in the ensemble.

The basic random forest algorithm for regression is almost identical to the random forest algorithm for classification. The only difference is that we use the MSE criterion to grow the individual decision trees, and the predict target variable is calculated as the average prediction over all decision trees. Random forest regression can use bagging to reduce overfitting and increase model generalization.

Chapter 11

Working with Unlabeled Data - Clustering Analysis

Cluster analysis is an unsupervised learning technique. The goal of clustering is to find a natural grouping in data so that items in the same cluster are more similar to each other than those from different clusters.

In clustering, the primary objective of minimizing the objective function is to minimize the distance between points within a cluster.

11.1 K-means clustering

K-mean is one of the most popular clustering algorithmsm and the objective is to minimize the within-cluster distance. It is very easy to implement and it is also very computationally efficient compared to other clustering algorithms. K-means belongs to the category of **prototype-based clustering**. **Hierarchical** and **density-based** clustering are other categories.

Prototype-based clustering means that each cluster is represented by a prototype, which is usually either the **centroid** (average) of similar points with continuous features, or the metoid (the most representative or the point that minimizes the distance to all other points that belong to a particular cluster) in the case of categorical features. While k-means is very good at identifying clusters with a spherical shape, one of the drawbacks of this clustering algorithm is that we have to specify the number of clusters, k. The wrong choice for k can result in poor clustering performance. Elbow methods and silhouette plots are useful techniques to evaluate quality of a clustering to help us determine the optimal number of clusters, k.

In real world applications of clustering we do no not have any ground truth category information. If we were given class labels, this task would fall into the category of supervised learning. Thus, our goal is to group the examples based on their feature similarities, which can be achieved using the k-means algorithm, as summarized by the following four steps:

1. Randomly pick k centroids from the examples as initial cluster centers.
2. Assign each example to the nearest centroid, $\mu^{(j)}$, $j \in 1, \ldots, k$.
3. Move the centroids to the center of the examples that were assigned to it.
4. Repeat steps 2 and 3 until the cluster assignment do not change or a user-defined tolerance or maximum number of iteration is reached.

The similarity between objects can be defined as opposite of distance, and a commonly used distance for clustering examples with continuous features is the **squared euclidian distance** between two points, x, and y, in m-dimensional space:

$$d(x, y)^2 = \sum_{j=1}^{m} (x_j - y_j)^2 = ||x - y||_2^2 \tag{11.1}$$

The cluster index j refers to the jth dimension (feature column) of the example inputs, x, and y.

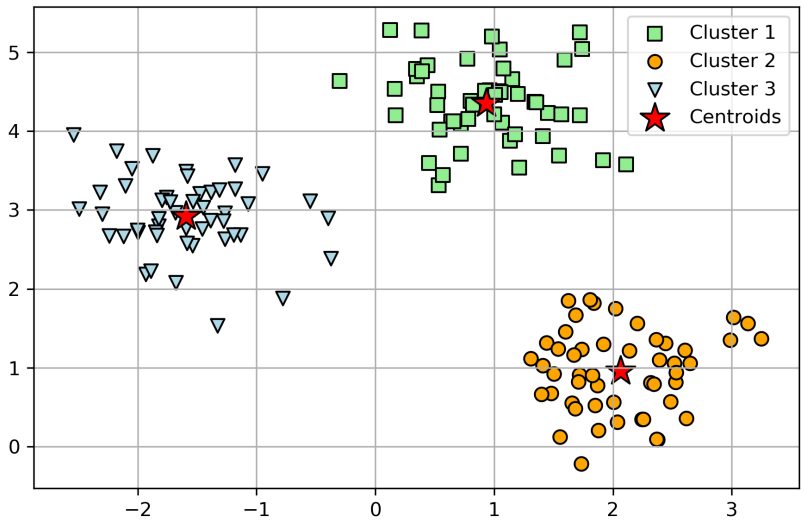
Based on this euclidian distance metric, we can describe the k-means algorithm as a simple optimization problem, an iterative approach for minimizing the within cluster sum of squared errors (SSE), which is sometimes called cluster inertia:

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} ||x^{(i)} - \mu^{(j)}||_2^2 \tag{11.2}$$

i: refers to the index of the example (data record)
$\mu^{(j)}$: the represantive point (centroid) for cluster j

$$w^{(i,j)} = \begin{cases} 1, & \text{if } x^{(i)} \in j \\ 0, & \text{otherwise} \end{cases} \tag{11.3}$$

A problem with k-means is that one or more cluster can be empty. However, this problem is accounted for in the current k-means implementation in sckikit-learn. If a cluster is empty, the algorithm will search for the example that is farthest away from the centroid of the empty cluster. Then it will reassign the centroid to be this farthest point.



In the figure you see that k-means placed the three centroids at the centre of each sphere, which looks like a reasonable grouping given this dataset.

Another drawback of k-means is that we have to specify the number of cluster, k before starting. The number might not be obvious in real-world applications. Other properties of k-means is that clusters do not overlap and are not hierarchical, and we also assume there is at least one item in each cluster.

11.1.1 K-means ++

K-means++ does not adress the problems in k-means, but will greatly improve the clustering results through more clever seeding of the initial cluster centers. Normal clustering uses a random seed to place the initial centroids, which can sometime result in bad clustering or slow convergence if the initial centroids are chosen poorly. A way to address this is to run the k-means algorithm several times and choose the best performing model in terms of SSE. However, this is not k-means++.

The initalization in k-means++ can be summarized as follows:

1. Initialze an empty set, M, to store the k centroids being selected.
2. Randomly choose the first centroid, $\mu^{(j)}$, from the input examples and assign it to M.
3. For each example, $x^{(i)}$, that is not in M, find the minimum squared distance, $d(x^{(i)}, M)^2$, to any of the centroids in M.
4. To randomly select the next centroid, $\mu^{(p)}$, use a weighted probability distribution equal to $\frac{d(\mu^{(p)},M)^2}{\sum_i d(x^{(i)},M)^2}$
5. Repeat steps 2 and 3 until k centroids are chosen.

6. Proceed with the classic k-means algorithm.

11.1.2 Hard versus soft clustering

Hard clustering describes a family of algorithms where each example in a dataset is assigned to exactly one cluster, as in the k-means and k-means++. In contrast, algorithms for soft clustering (sometimes called fuzzy clustering) assign an example to one or more clusters. A popular example of soft clustering is fuzzy c-means (FCM) algorithm (also called fuzzy k-means or soft k-means).

Fuzzy clustering assign probabilities of cluster members, to minimize the sum of the squared distances between data points and their assigned clusters. It is very similar to k-means, but replace the hard cluster assignment with probabilities for each point belonging to each cluster. It can also handle noisy data better than K-means, but PCM is more sensitive to the initial choice of centroids.

The objective function of FCM, J_m, looks very similar to the within-cluster SSE that we minimize in k-means:

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)^m} ||x^{(i)} - \mu^{(j)}||_2^2 \tag{11.4}$$

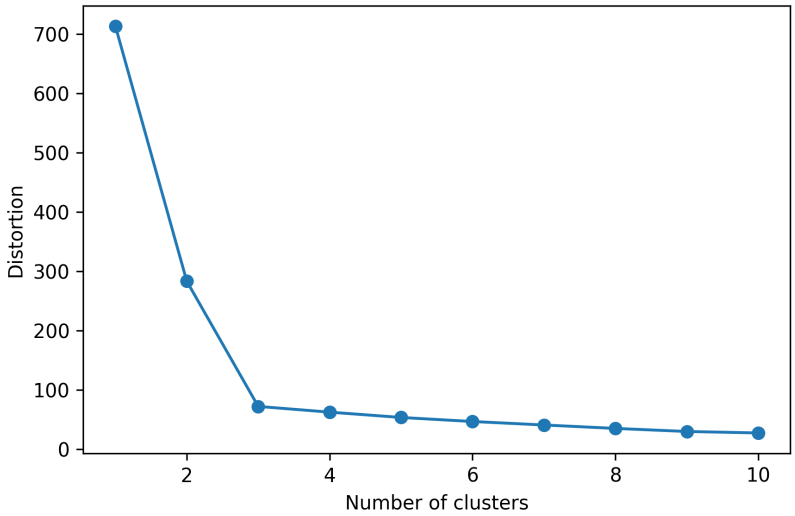
m controls the degree of fuzziness, and a large m means very fussy, which means small cluster membership (more diffuse membership values).

A disadvantage to Fuzzy C-means clustering is that it may struggle to find meaningful clusters in high-dimensional spaces.

11.1.3 The elbow method

One of the main challenges in unsupervised learning is that we do not know the definitive answer. Thus, to quantify the quality of clustering, we need to use intrinsic metrics (such as the within cluster SSE (distortion)) to compare the performance of different k-means clusterings.

Conveniently, we do not need to compute the within-cluster SSE when we are using scikit-learn, as it is already accesible via `inertia_` attribute after fitting a `KMeans` model. Based on the within-cluster SSE we can use a graphical tool, the elbow method, to estimate the optimal numer of clusters, k, for a given task. We can say that if k increases, the distortion will decrease. This is because the examples will be closer to the centroids they are assigned to. The idea behind the elbow method is to identify the value of k where the distortion begins to increase most rapidly, which becomes clear if we plot the distortion.



The *elbow* is located at $k = 3$.

11.1.4 Silhouette plots

Another intrisic metric to evaluate the quality of a clustering, by measuring the separation between clusters, is silhouette analysis, which can also be applied to clustering algorithms other than k-means. Silhouette analysis can be used as a graphical tool to plot a measure of how tightly grouped the examples in the clusters are. To calculate the silhouette coeficient of a single example in our dataset, we can apply the following three steps:

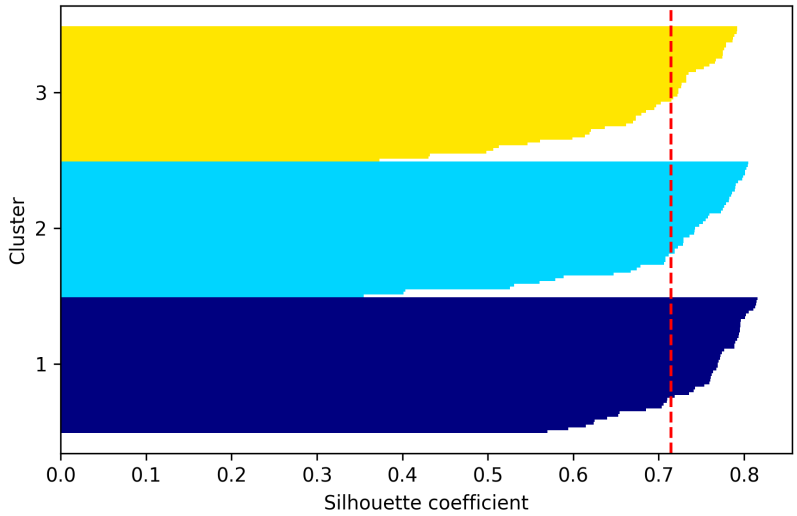
1. Calculate the cluster cohesion, $a^{(i)}$, as the average distance between an example, $x^{(i)}$, and all other points in the same cluster.

2. Calculate the cluster separation, $b^{(i)}$, from the next closest cluster as the average distance between the example, $x^{(i)}$, and all examples in the nearest cluster.

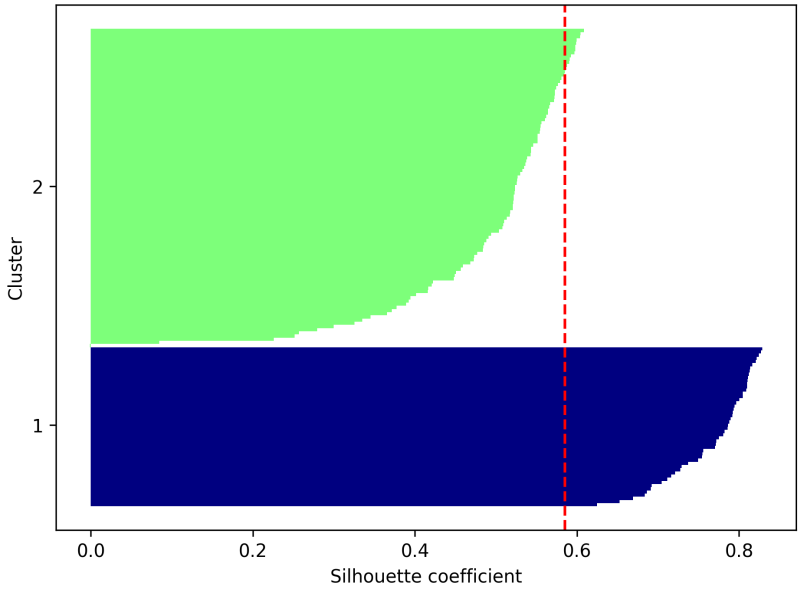
3. Calculate the silhouette, $s^{(i)}$, as the difference between cluster cohesion and seperation divided by the greater of the two, as shown here:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max b^{(i)}, a^{(i)}} \tag{11.5}$$

The silhouette coefficient is bounded in the range -1 to 1.



As you can see inthis plot the silhouetes is not even close to 0, which is an indicator of a good clustering.



In this plot the silhouettes now have visibly different lengths and widths, which is evidence of a relativily bad or at least suboptimal clustering.

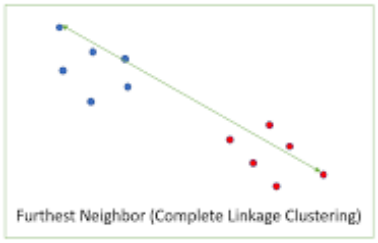
11.2 Organzing clusters as a hierarchical tree

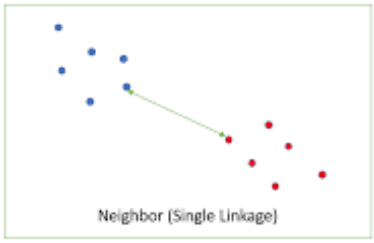
One advantage of the hierarchical clustering algorithm is that it allows us to plot **dendrograms** (visualization that shows the hierarchy of nested clusters), which can help with the interpretation of the results by creating meaningful taxonomies. Another advantage of this hierarchical approach is that we do not need to specify the number of clusters upfront.

The two main approaches to hierarchical clustering are **agglomerative** and **divisive** hierarchical clustering. In divisive hierarchical clustering, we start with one cluster the encompasses the complete dataset, and we iteratively split the cluster into smaller clusters until each cluster only contains one example. Agglomerativ clustering takes the opposite approach. We start with each example as an individual cluster and merge the closest pairs of clusters until one cluster remains.

11.2.1 Agglomerative hierarchical clustering

The two standard algorithms for agglomerative hierarchical clustering are **single linkage** and **complete linkage**. Using single linkage, we compute the distance between the most similar members for each pair of clusters and merge the two cluster for which the distance between the most similar members is the smallest. The complete linkage approach is similar to single linkage but, instead of comparing the most similar members in each pair of clusters, we compare the most dissimilar members to perform the merge.





This section focuses on agglomerative clustering using complete linkage. It is an iterative procedure that can be summarized by the following steps:

1. Compute the distance matrix of all examples.
2. Represent each data point as a singleton cluster.
3. Merge the two closetst clusters on the distance between the most dissimilar (distant) members.
4. Update the similiarity matrix.
5. Repeat steps 2-4 until a single cluster remains.

In practical applications, hierarchical clustering, **dendrograms** are often used in combination with a heat map, which allows us to represent the individual

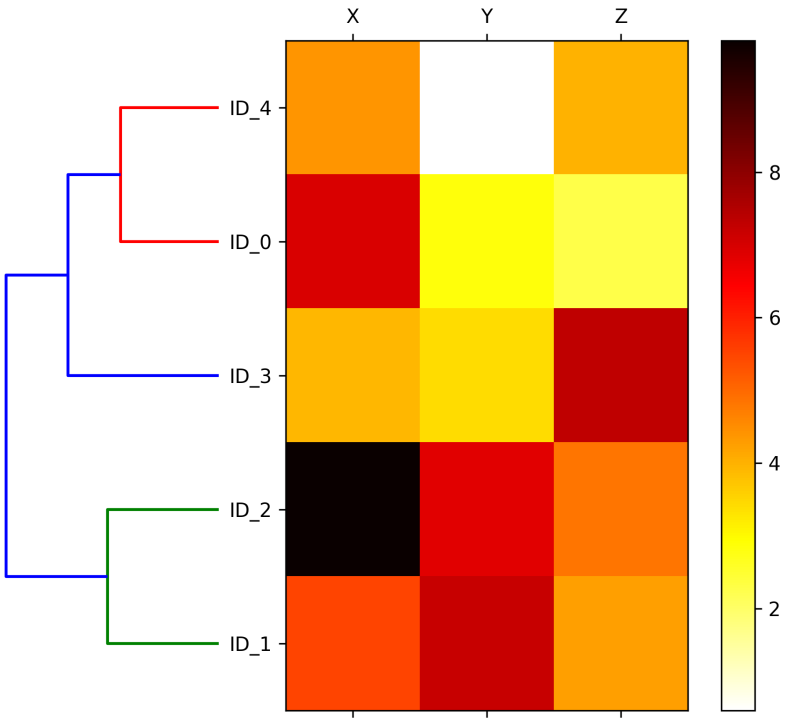
values in the data array or matrix containing our training examples with color code. However, attaching a dendrogram to a heat map can be a little bit tricky:

1. We create a new figure object and define the x axis position, y axis position, width and height of the dendrogram via the add axes attribute. Furthermore, we rotate the dendrogram 90 degrees counter-clockwise.

2. Next, we reorder the data in our initial DataFrame according to the clustering labels that can be accessed from the dendrogram object, which is essentially a Python dictionary, via the leaves key.

3. Now, we construct the heat map from the reordered DataFrame and position it next to the dendrogram.

4. Finally, we modify the aesthetics if the dendrogram by removing the axis ticks and hiding the axis spines. Also, we add a color bar and assign the feature and data records names to the x and y axis tick labels, respectively.



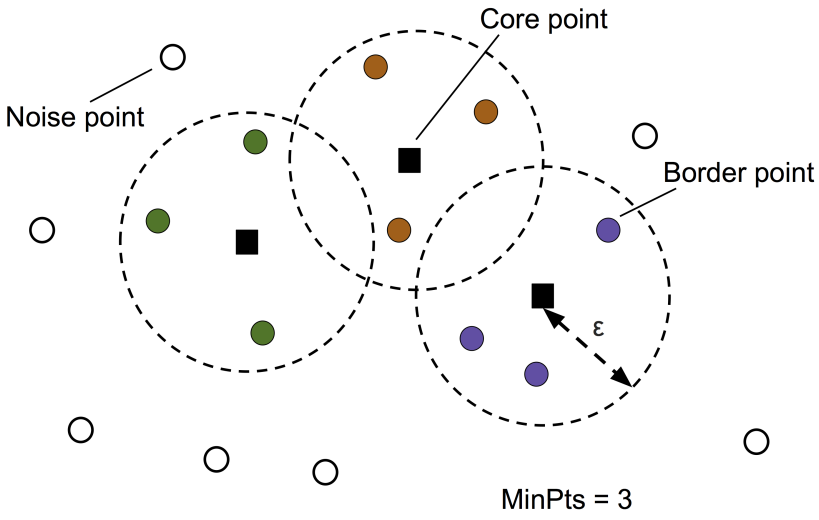
The order of rows in the heat map reflects the clustering of the examples in the dendrogram. In addition to a simple dendrogram, the color-coded values of each example and feature in the heat map provide us with a nice summary of the test dataset.

11.3 Locating regions of high density via DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN), does not make assumptions about sphericall clusters, like k-means, nor does it partition the dataset into hierarchies that require a manual cut-off point. Density-based clustering assigns cluster labels based on dense regions of points. The notion of density is defined as the number of points within a specidied radius, ϵ.

According to the DBSCAN algorithm, a special label is assigned to each example (data point) using the following criteria:

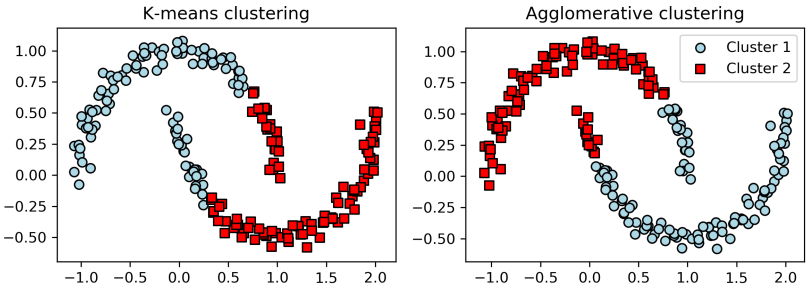
- A point is considered a core point if at least a specified number (MinPts) of neighbouring points fall within the specified radius, ϵ.
- A border point is a point that that has fewer neighbours than MinPts within ϵ, but lies within the ϵ radius of a core point.
- All other points that are neither core nor border points are considered noise points.



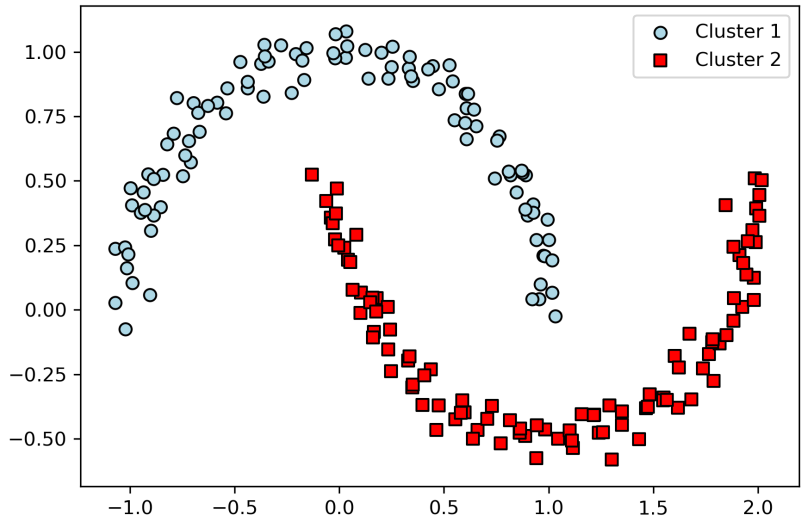
After labeling the points as core, border, or noise, the DDBSCAN algorithm can be summarizes in two simple steps:

1. From a seperate cluster for each core point or connected group of core points. (Core points are connected if they are no farther away that ϵ.)
2. Assign each border point to the cluster of its corresponding core point.

One of the main advantages of using DBSCAN is that it does not assume that the clusters have a spherical shape, like in k-means. Furthermore, DBSCAN is different from k-means and hierarchical clustering in that it does not necessarily assign each point to a cluster but is capable of removing noise points.



Based on the visualized clustering results, we can see that the k-means algorithm was unable to separate the two clusters, and also, the hierarchical clustering algorithm.



DBSCAN can successfully detect the half-moon shapes, which highlights one of the strengths of DBSCAN - clustering data of arbitrary shapes.

However, we should also note some of the disadvantages of DBSCAN. With an increasing number of features in our dataset - assuming a fixed number of training examples - the negative effect of the curse of dimensionality increases. This is especially a problem if we are using the Euclidean distance metric. However, the problem of the curse of dimensionality is not unique to DBSCAN: it also affects other clustering algorithms that use the Euclidean distance metrics, for example, k-means and hierarchical clustering algorithms. In addition, we have two hyperparameters in DBSCAN (MinPts and ϵ) that need to be optimized to yield good clustering results. Finding a good combination of MinPts

and ϵ can be problematic if the density differences in the dataset are relatively large.

In the context of the curse of dimensionality, it is common practice to apply dimensionality reduction techniques prior to performing clustering. Such dimensionality reduction for unsupervised datasets include PCA and RBF. Also it is particularly common to compress datasets down to two-dimensional subspaces, which allows us to visualize the clusters and assigned labels using two-dimensional scatterplots, which are particularly helpful for evaluating the results.