Contents

Chapter 1

Training Simple Machine Learning Algorithms for Classification

1.1 Implementing a perceptron learning algorithm in Python

1.1.1 An object-oriented perceptron API

If all the weights are initialized to zero, the learning rate parameter, η, affects only the scale of the weight vector, not the direction. Consider a vector,$v_1 = [1\ 2\ 3]$, where the angle between v_1 and a vector, $v_2 = 0.5 \times v_1$, would be exactly zero.

1.2 Adaptive linear neurons and the convergence of learning

We will take a look at another type of single-layer **neural network (NN): ADAptive LInear NEuron (Adaline)**. The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing continuous loss functions.

The key difference between the Adaline rule (also known as the Widrow-Hoff rule) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function, $\sigma(z)$, is simply the identity function of the net input, so that $\sigma(z) = z$.

As Figure 1.1 indicates, the Adaline algorithm compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

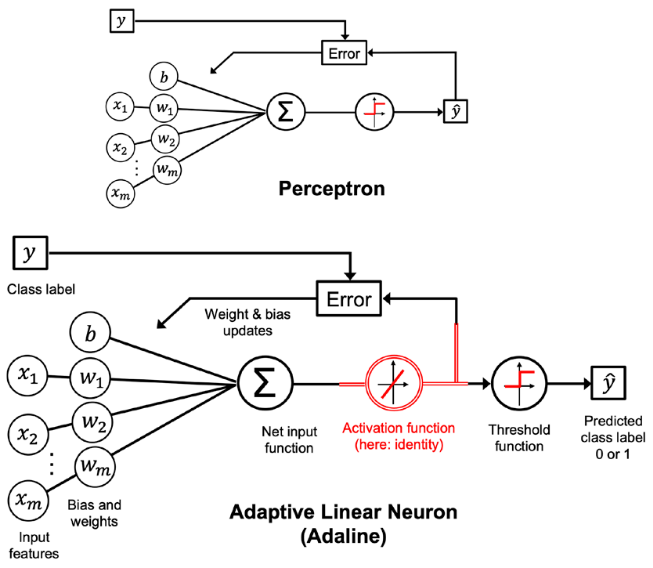

Figure 1.1: A comparison between a perceptron and the Adaline algorithm

Although the Adaline learning rule looks identical to the perceptron rule, we should note that $\sigma(z^{(i)})$ with $\sigma(z^{(i)}) = \mathbf{w}^T\mathbf{x}^{(i)} + b$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all examples in the training dataset (instead of updating the parameters incrementally after each training example), which is why this approach is also referred to as **batch gradient descent**. To be more explicit and avoid confusion when talking about related concepts later, we will refer to this process as **full batch gradient descent**.

1.2.1 Improving gradient descent through feature scaling

Gradient descent is one of the many algorithms that benefit from feature scaling. In this section, we will use a feature scaling method called standardization. This normalization procedure helps gradient descent learning to converge more quickly; however, it does not make the original dataset normally distributed.

One of the reasons why standardization helps with gradient descent learning is that it is easier to find a learning rate that works well for all weights (and the bias). If the features are on vastly different scales, a learning rate that works well for updating one weight might be too large or too small to update the other weight equally well. Overall, using standardized features can stabilize the training such that the optimizer has to go through fewer steps to find a good or optimal solution (the global loss minimum).

1.2.2 Large-scale machine learning and stochastic gradient descent

A popular alternative to the batch gradient descent algorithm is **stochastic gradient descent (SGD)**, which is sometimes also called iterative or online gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all training examples, $\mathbf{x}^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n} \sum_i \left(y^{(i)} - \sigma(z^{(i)}) \right) x_j^{(i)}$$

we update the parameters incrementally for each training example, for instance:

$$\Delta w_j = \eta \left(y^{(i)} - \sigma(z^{(i)}) \right) x_j^{(i)}, \ \Delta b = \eta \left(y^{(i)} - \sigma(z^{(i)}) \right)$$

Adjusting the learning rate during training

In SGD implementations, the fixed learning rate, η , is often replaced by an adaptive learning rate that decreases over time, for example:

$$\frac{c_1}{[number\ of\ iterations] + c_2}$$

where $c_1$and c_2 are constants. Note that SGD does not reach the global loss minimum but an area very close to it. And using an adaptive learning rate, we can achieve further annealing to the loss minimum.

Chapter 2

A Tour of Machine Learning Classifiers Using Scikit-Learn

2.1 Modeling class probabilities via logistic regression

2.1.1 Logistic regression and conditional probabilities

Under the logistic model, we assume that there is a linear relationship between the weighted inputs and the log-odds:

$$logit(p) = w_1x_1 + w_2x_2 + \cdots + w_mx_m + b = \sum_{i=j} w_j x_j = \mathbf{w}^T\mathbf{x} + b \tag{2.1}$$

2.1.2 Learning the model weights via the logistic loss function

To explain how we can derive the loss function for logistic regression, let's first define the likelihood, $\mathscr{L}$, that we want to maximize when we build a logistic regression model, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$\begin{aligned}\mathscr{L}(\mathbf{w}, b|\mathbf{x}) = p(y|\mathbf{x};\mathbf{w}, b) &= \prod_{i=1}^{m} p\left(y^{(i)}|\mathbf{x}^{(i)};\mathbf{w}, b\right) \\ &= \prod_{i=1}^{m}\left(\sigma(z^{(i)})\right)^{y^{(i)}}\left(1-\sigma(z^{(i)})\right)^{1-y^{(i)}}\end{aligned} \tag{2.2}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood** function:

$$\begin{aligned} l(\mathbf{w}, b|\mathbf{x}) &= \log \mathscr{L}(\mathbf{w}, b|\mathbf{x}) \\ &= \sum_{i=1}\left[y^{(i)}\log(\sigma(z^{(i)})) + (1-y^{(i)})\log(1-\sigma(z^{(i)}))\right]\end{aligned} \tag{2.3}$$

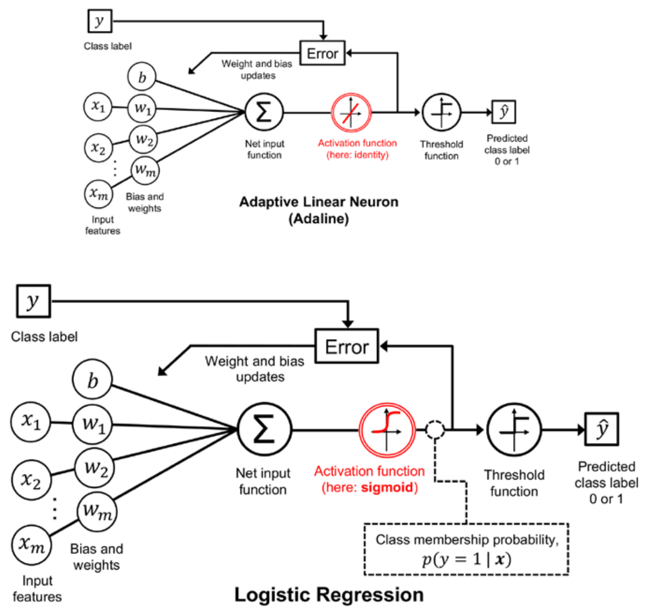
y
Class label
Error
Weight and bias updates
b
x1
w1
x2
w2
xm
wm
Net input function
Activation function (here: identity)
Threshold function
ŷ
Predicted class label 0 or 1
Bias and weights
Input features
Adaptive Linear Neuron (Adaline)
Activation function (here: sigmoid)
Class membership probability, p(y = 1 | x)
Logistic Regression

Figure 2.1: Logistic regression compared to Adaline

2.1.3 Tackling overfitting via regularization

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 a 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 useful method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting.

Regularization and feature normalization

Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

The loss function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:

$$L(\mathbf{w}, b|\mathbf{x}) = \sum_{i=1} \left[y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)})) \right] + \frac{\lambda}{2n} ||\mathbf{w}||^2 \tag{2.4}$$

The parameter, C, that is implemented for the LogisticRegression class in scikit-learn comes from a convention in support vector machines. The term C is inversely proportional to the regularization parameter, λ . Consequently, decreasing the value of the inverse regularization parameter, C, means that we are increasing the regularization strength.

The bias-variance tradeoff

Often, researchers use the terms "bias" and "variance" or "bias-variance trade-off" to describe the performance of a model—that is, you may stumble upon talks, books, or articles where speople say that a model has a "high variance" or "high bias." So, what does that mean? In general, we might say that "high variance" is proportional to overfitting and "high bias" is proportional to underfitting.
In the context of machine learning models, variance measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

2.2 Maximum margin classification with support vector machines

In SVMs, our optimization objective is to maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called support vectors.

2.2.1 Dealing with a nonlinearly separable case using slack variables

The motivation for introducing the slack variable was that the linear constraints in the SVM optimization objective need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate loss penalization.

The use of the slack variable, in turn, introduces the variable, which is commonly referred to as C in SVM contexts. We can consider C as a hyperparameter for controlling the penalty for misclassification. Large values of C correspond to large error

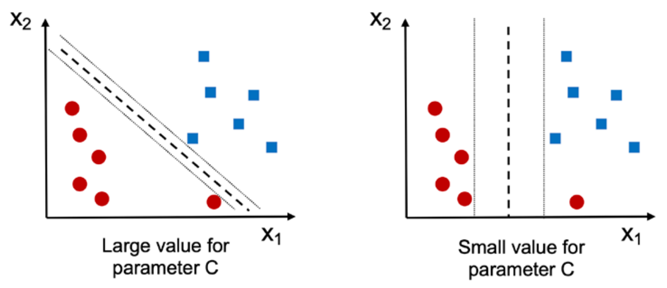

Figure 2.2: The impact of large and small values of the inverse regularization strength C on classification

penalties, whereas we are less strict about misclassification errors if we choose smaller values for C. We can then use the C parameter to control the width of the margin and therefore tune the bias-variance tradeoff, as illustrated in Figure 2.2:

Logistic regression versus SVMs

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs, which mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage of being a simpler model and can be implemented more easily, and is mathematically easier to explain. Furthermore, **logistic regression models can be easily updated, which is attractive when working with streaming data**.

2.2.2 Alternative implementations in scikit-learn

The advantage of using LIBLINEAR and LIBSVM over, for example, native Python implementations is that they allow the extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the SGDClassifier class, which also supports online learning via the `partial_fit` method.

2.3 Solving nonlinear problems using a kernel SVM

2.3.1 Kernel methods for linearly inseparable data

The basic idea behind kernel methods for dealing with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional space via a mapping function, ϕ , where the data becomes linearly separable. As shown in Figure 3.14, we can transform a two-dimensional dataset into a new

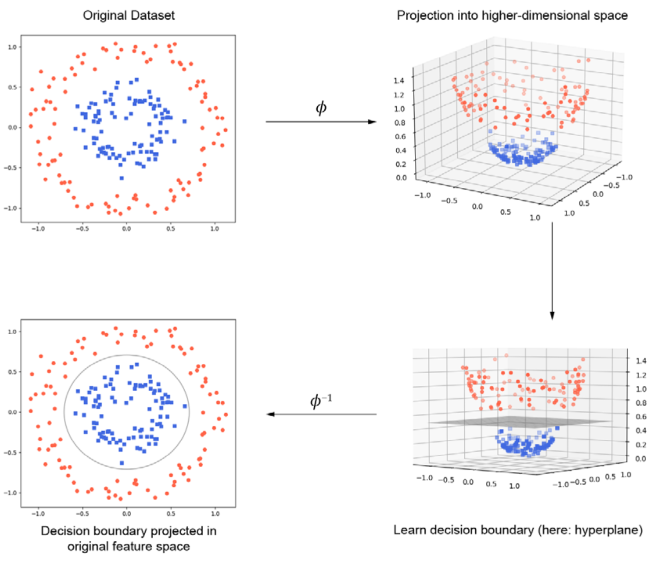

Figure 2.3: The process of classifying nonlinear data using kernel methods

three-dimensional feature space, where the classes become separable via the following projection:

$$\phi(x_1,x_2) = (z_1,z_2,z_3) = (x_1,x_2,x_1^2+x_2^2)$$

2.3.2 Using the kernel trick to find separating hyperplanes in a highdimensional space

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

In practice, we just need to replace the dot product $\mathbf{x}^{(i)T}\mathbf{x}^{(j)}$ by $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$. To save the expensive step of calculating this dot product between two points explicitly, we define a so-called kernel function:

$$k(\mathbf{x}^{(i)}\mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$$

Roughly speaking, the term “kernel” can be interpreted as a similarity function between a pair of examples. The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar examples) and 0 (for very dissimilar examples).

The γ parameter can be understood as a cut-off parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training examples, which leads to a tighter and bumpier decision boundary.

2.4 Decision tree learning

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain (IG)**.

2.4.1 Maximizing IG – getting the most bang for your buck

To split the nodes at the most informative features, we need to define an objective function to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j) \tag{2.5}$$

Here, f is the feature to perform the split; D_p and D_j are the dataset of the parent and jth child node; I is our **impurity** measure; N_p is the total number of training examples at the parent node; and N_j is the number of examples in the jth child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain.

The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all non-empty classes ($p(i|t) \neq 0$):

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t) \tag{2.6}$$

The Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2 \tag{2.7}$$

Another impurity measure is the classification error:

$$I_E(t) = 1 - \max p(i|t) \tag{2.8}$$

This is a useful criterion for pruning, but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes.

2.4.2 Building a decision tree

Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms.

2.4.3

The idea behind a random forest is to average multiple (deep) decision trees that individually suffer from high variance to build a more robust model that has a better generalization performance and is less susceptible to overfitting. The random forest algorithm can be summarized in following:

Algorithm 1: The random forest algorithm

```
1 begin
2     for $k \leftarrow 1$ to $K$ do
3         Draw a random bootstrap sample of size $n$ (randomly choose $n$
            examples from the training dataset with replacement);
4         Grow a decision tree from the bootstrap sample; foreach node do
5             Randomly select $d$ features without replacement;
6             Split the node using the feature that provides the best split
                according to the objective function, for instance, maximizing the
                information gain;
7     Aggregate the prediction by each tree to assign the class label by majority
        vote;
```

2.5 K-nearest neighbors – a lazy learning algorithm

KNN is a typical example of a lazy learner. It is called "lazy" not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

The KNN algorithm itself is fairly straightforward and can be summarized by the following:

Algorithm 2: The KNN algorithm

```
1 begin
2     Choose the number of $k$ and a distance metric;
3     Find the $k$-nearest neighbors of the data record that we want to classify;
4     Assign the class label by majority vote;
```

In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the data record to be classified. If the neighbors have similar distances, the algorithm will choose the class label that comes first in the training dataset.

It is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. We can think of even the closest neighbors as being too far away in a high-dimensional space to give a good estimate.

Alternative machine learning implementations with GPU support

If you have a computer equipped with an NVIDIA GPU that is compatible with recent versions of NVIDIA's CUDA library, we recommend considering the RAPIDS ecosystem. For instance, RAPIDS' cuML library implements many of scikit-learn's machine learning algorithms with GPU support to accelerate the processing speeds. You can find an introduction to cuML.

Chapter 3

Building Good Training Datasets – Data Preprocessing

3.1 Dealing with missing data

3.1.1 Imputing missing values

One of the most common interpolation techniques is **mean imputation**, where we simply replace the missing value with the mean value of the entire feature column. A convenient way to achieve this is by using the SimpleImputer class from scikit-learn.

3.1.2 Understanding the scikit-learn estimator API

The SimpleImputer class is part of the so-called **transformer** API in scikit-learn, which is used for implementing Python classes related to data transformation. The two essential methods of those estimators are fit and transform. The fit method is used to learn the parameters from the training data, and the transform method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model.

The **classifiers** that we used belong to the so-called estimators in scikit-learn, with an API that is conceptually very similar to the scikit-learn transformer API. Estimators have a predict method but can also have a transform method.

3.2 Handling categorical data

When we are talking about categorical data, we have to further distinguish between ordinal and nominal features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, t-shirt size would be an ordinal feature, because we can define an order: $XL > L > M$. In contrast, nominal features don't imply any order.

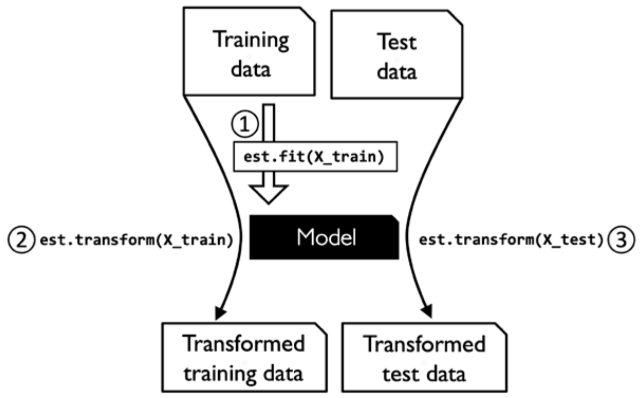

Figure 3.1: Using the scikit-learn API for data transformation

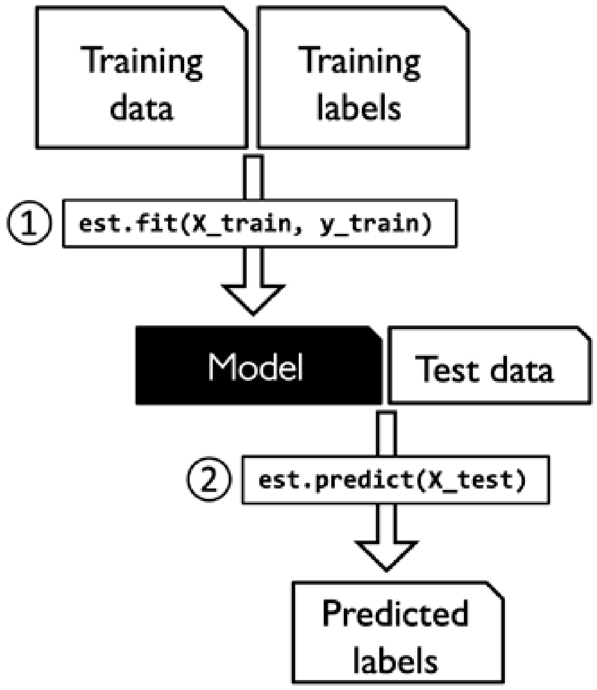

Figure 3.2: Using the scikit-learn API for predictive models such as classifiers

3.2.1 Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our size feature, so we have to define the mapping manually.

Optional: encoding ordinal features

3.3 Bringing features onto the same scale

Feature scaling is a crucial step in our preprocessing pipeline that can easily be forgotten. Decision trees and random forests are two of the very few machine learning algorithms where we don't need to worry about feature scaling.

There are two common approaches to bringing different features onto the same scale: **normalization** and **standardization**. Most often, normalization refers to the rescaling of the features to a range of $[0,1]$, which is a special case of **min-max scaling**. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value, $x^{(i)}_{norm}$, of an example, $x^{(i)}$, can be calculated as follows:

$$x^{(i)}_{norm} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}} \tag{3.1}$$

Although normalization via min-max scaling is a commonly used technique that is useful when we need values in a bounded interval, standardization can be more practical for many machine learning algorithms, especially for optimization algorithms such as gradient descent. The reason is that many linear models, such as the logistic regression and SVM, initialize the weights to 0 or small random values close to 0. Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns **have the same parameters as a standard normal distribution** (zero mean and unit variance), which makes it easier to learn the weights. However, we shall emphasize that standardization does not change the shape of the distribution, and it does not transform non-normally distributed data into normally distributed data. In addition to scaling data such that it has zero mean and unit variance, standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

The procedure for standardization can be expressed by the following equation:

$$x^{(i)}_{std} = \frac{x^{(i)} - \mu_x}{\sigma_x} \tag{3.2}$$

3.4 Selecting meaningful features

3.4.1 Sequential feature selection algorithms

Algorithm 3: Sequential Backward Selection (SBS) algorithm

```
1 begin
2     repeat
3         Determine the feature, $\mathbf{x}^-$, that maximizes the criterion:
            $\mathbf{x}^- = argmaxJ(\mathbf{X}_k - \mathbf{x})$, where $\mathbf{x}^- \in \mathbf{X}_k$;
4         Remove the feature, $\mathbf{x}^-$, from the feature set: $\mathbf{X}_{k-1} \leftarrow \mathbf{X}_k - \mathbf{x}$; $k \leftarrow k$–1;
5     until k equals the number of desired features;
```

Chapter 4

Compressing Data via Dimensionality Reduction

4.1 Unsupervised dimensionality reduction via principal component analysis

The difference between feature selection and feature extraction is that while we maintain the original features when we use feature selection algorithms, such as sequential backward selection, we use feature extraction to transform or project the data onto a new feature space.

In the context of dimensionality reduction, feature extraction can be understood as an approach to data compression with the goal of maintaining most of the relevant information. In practice, feature extraction is not only used to improve storage space or the computational efficiency of the learning algorithm but can also improve the predictive performance by reducing the curse of dimensionality—especially if we are working with non-regularized models.

4.1.1 The main steps in principal component analysis

In a nutshell, PCA aims to find the directions of maximum variance in high-dimensional data and projects the data 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 given the constraint that the new feature axes are orthogonal to each other, as illustrated in Figure 4.1:

If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix, $\mathbf{W}$, that allows us to map a vector of the features of the training example, $\mathbf{x}$, onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space. For instance, the process is as follows. Suppose we have a feature vector, $\mathbf{x}$:

$$\mathbf{x} = [x_1, x_2, \cdots, x_d], \mathbf{x} \in \mathbb{R}^d$$

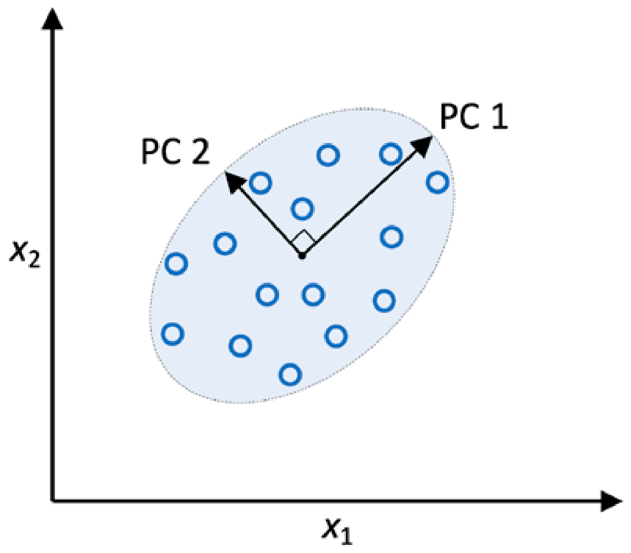

Figure 4.1: Using PCA to find the directions of maximum variance in a dataset

which is then transformed by a transformation matrix, $\mathbf{W} \in \mathbb{R}^{d \times k}$:

$$\mathbf{xW} = \mathbf{z}$$

resulting in the output vector:

$$\mathbf{z} = [z_1, z_2, \cdots, z_k], \mathbf{z} \in \mathbb{R}^k$$

Eigendecomposition: Decomposing a Matrix into Eigenvectors and Eigenvalues

Eigendecomposition, the factorization of a square matrix into so-called **eigenvalues** and **eigenvectors**, is at the core of the PCA procedure.
The covariance matrix is a special case of a square matrix: it's a symmetric matrix, which means that the matrix is equal to its transpose, $A = A^T$.
When we decompose such a symmetric matrix, the eigenvalues are real (rather than complex) numbers, and the eigenvectors are orthogonal (perpendicular) to each other. Furthermore, eigenvalues and eigenvectors come in pairs. If we decompose a covariance matrix into its eigenvectors and eigenvalues, the eigenvectors associated with the highest eigenvalue corresponds to the direction of maximum variance in the dataset. Here, this "direction" is a linear transformation of the dataset's feature columns.

Algorithm 4: PCA algorithm for dimensionality reduction

```
1 begin
2     Standardize the $d$-dimensional dataset;
3     Construct the covariance matrix;
4     Decompose the covariance matrix into its eigenvectors and eigenvalues;
5     Sort the eigenvalues by decreasing order to rank the corresponding
        eigenvectors; Select $k$ eigenvectors, which correspond to the $k$ largest
        eigenvalues, where $k$ is the dimensionality of the new feature subspace
        ($k \leq d$);
6     Construct a projection matrix, $\mathbf{W}$, from the "top" $k$ eigenvectors;
7     Transform the $d$-dimensional input dataset, X, using the projection matrix,
        $\mathbf{W}$, to obtain the new $k$-dimensional feature subspace;
```

4.1.2 Assessing feature contributions

Sometimes, we are interested to know about how much each original feature contributes to a given principal component. These contributions are often called **loadings**.

The factor loadings can be computed by scaling the eigenvectors by the square root of the eigenvalues. The resulting values can then be interpreted as the correlation between the original features and the principal component.

4.2 Supervised data compression via linear discriminant analysis

The general concept 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.

4.2.1 Principal component analysis versus linear discriminant analysis

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.

4.2.2 The inner workings of linear discriminant analysis

Chapter 5

Learning Best Practices for Model Evaluation and Hyperparameter Tuning

5.1 Fine-tuning machine learning models via grid search

5.1.1 More resource-efficient hyperparameter search with successive halving

Taking the idea of randomized search one step further, scikit-learn implements a successive halving variant, HalvingRandomSearchCV, that makes finding suitable hyperparameter configurations more efficient. Successive halving, given a large set of candidate configurations, successively throws out unpromising hyperparameter configurations until only one configuration remains. We can summarize the procedure via the following steps:

1. Draw a large set of candidate configurations via random sampling

2. Train the models with limited resources, for example, a small subset of the training data (as opposed to using the entire training set)

3. Discard the bottom 50 percent based on predictive performance

4. Go back to step 2 with an increased amount of available resources

5.1.2 Algorithm selection with nested cross-validation

If we want to select among different machine learning algorithms, though, another recommended approach is **nested cross-validation**.

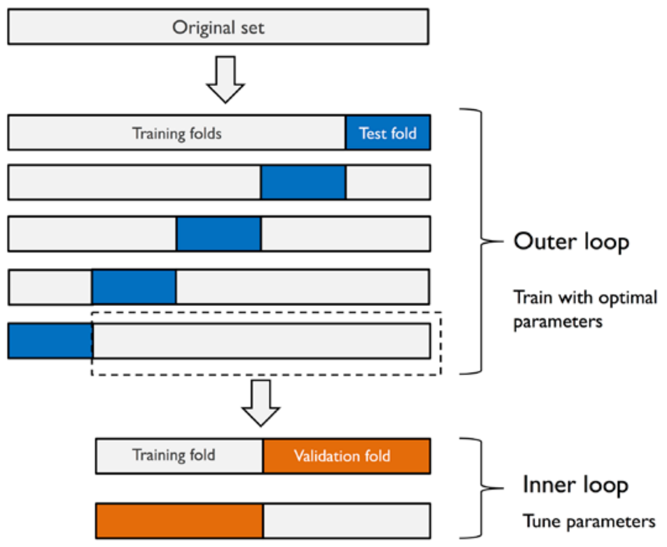

Figure 5.1: The concept of nested cross-validation

5.2 Looking at different performance evaluation metrics

There are several other performance metrics that can be used to measure a model's relevance, such as precision, recall, the **F1 score**, and **Matthews correlation coefficient (MCC)**.

5.2.1 Optimizing the precision and recall of a classification model

A measure that summarizes a confusion matrix is the MCC, which is especially popular in biological research contexts. The MCC is calculated as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \in [-1,1]$$

In contrast to PRE, REC, and the F1 score, the MCC ranges between –1 and 1, and it takes all elements of a confusion matrix into account—for instance, the F1 score does not involve the TN. While the MCC values are harder to interpret than the F1 score, it is regarded as a superior metric.

5.2.2 Scoring metrics for multiclass classification

scikit-learn also implements macro and micro averaging methods to extend those scoring metrics to multiclass problems via **one-vs.-all (OvA)** classification. The micro-average is calculated from the individual TPs, TNs, FPs, and FNs of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$RPE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{micro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label *by the number of true instances when calculating the average*. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

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 loss 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.

One way to deal with imbalanced class proportions during model fitting is to assign a larger penalty to wrong predictions on the minority class. Other popular strategies for dealing with class imbalance include upsampling the minority class, downsampling the majority class, and the generation of synthetic training examples.

Generating new training data to address class imbalance

Another technique for dealing with class imbalance is the generation of synthetic training examples. Probably the most widely used algorithm for synthetic training data generation is Synthetic Minority Over-sampling Technique (SMOTE)

Chapter 6

Combining Different Models for Ensemble Learning

6.1 Learning with ensembles

The goal of ensemble methods is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone.

6.2 Bagging – building an ensemble of classifiers from bootstrap samples

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 dataset, which is why bagging is also known as *bootstrap aggregating*.

The concept of bagging is summarized in Figure 6.1:

6.2.1 Bagging in a nutshell

In fact, random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.

In practice, more complex classification tasks and a dataset's high dimensionality can easily lead to overfitting in single decision trees, and this is where the bagging algorithm can really play to its strengths. Finally, we must note that the bagging algorithm can be an effective approach to reducing the variance of a model. However, bagging is ineffective in reducing model bias, that is, models that are too simple to capture the trends in the data well. This is why we want to perform bagging on an ensemble of classifiers with low bias, for example, unpruned decision trees.

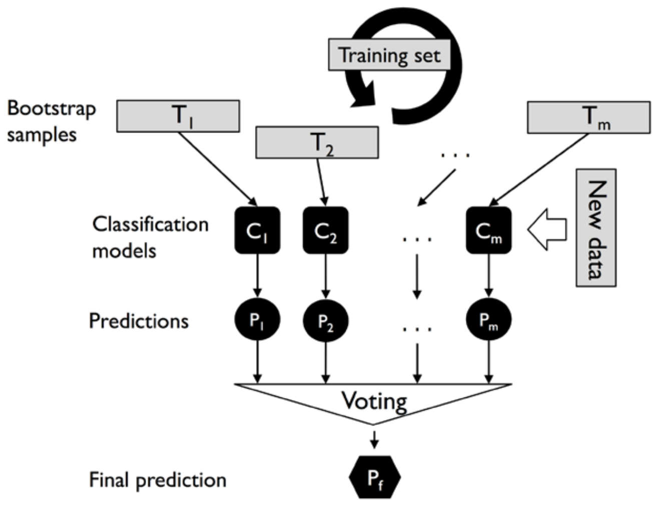

Figure 6.1: The concept of bagging

6.3 Leveraging weak learners via adaptive boosting

We will discuss **boosting**, with a special focus on its most common implementation: **Adaptive Boosting** (AdaBoost).

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 of the ensemble.

6.3.1 How adaptive boosting works

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

1. Draw a random subset (sample) of training examples, d_1, without replacement from the training dataset, D, to train a 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.

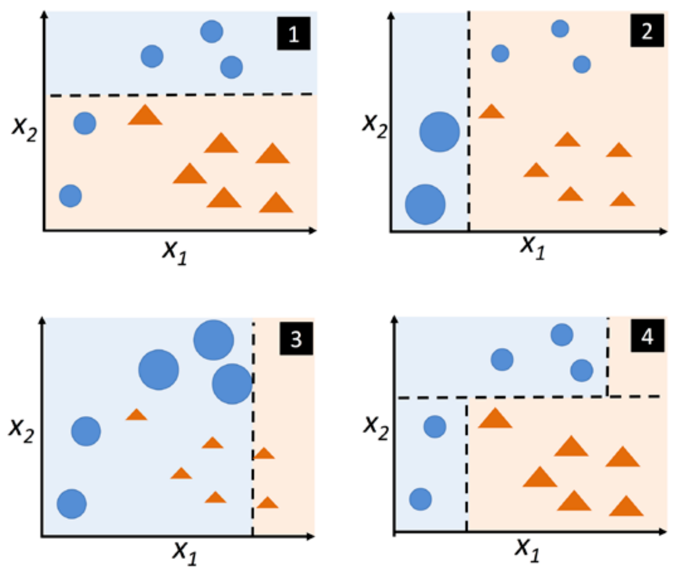

Figure 6.2: The concept of AdaBoost to improve weak learners

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

6.4 Gradient boosting – training an ensemble based on loss gradients

6.4.1 Outlining the general gradient boosting algorithm

Algorithm 5: AdaBoost algorithm

```
1  begin
2      initialization, Set the weight vector, $\mathbf{w}$, to uniform weights, where
         $\sum_i w_i = 1$;
3      for $j \rightarrow 1$ to $m$ do
4          Train a weighted weak learner: $C_j = train(\mathbf{X}, \mathbf{y}, \mathbf{w})$;
5          Predict class labels: $\hat{y} = predict(C_j, \mathbf{X})$;
6          Compute the weighted error rate: $\varepsilon = \mathbf{W} \cdot (\hat{\mathbf{y}} \neq \mathbf{y})$;
7          Compute the coefficient:$\alpha_j = 0.5 \log \frac{1-\varepsilon}{\varepsilon}$;
8          Update the weights: $\mathbf{w} \leftarrow \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y})$;
9          Normalize the weights to sum to 1:$\mathbf{w} \leftarrow \mathbf{w} / \sum_i w_i$;
10     Compute the final prediction: $\hat{\mathbf{y}} = \left( \sum_{j=1}^{m} (\alpha_j \times predict(C_j, \mathbf{X})) \right)$;
```

Algorithm 6: The general outline of the gradient boosting algorithm.

```
1 begin
2     Initialize a model to return a constant prediction value. For this, we use a
        decision tree root node; that is, a decision tree with a single leaf node. We
        denote the value returned by the tree as $\hat{y}$ , and we find this value by
        minimizing a differentiable loss function $L$ that we will define later:

        $$F_0(x) = \underset{\hat{y}}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, \hat{y})$$

        Here, $n$ refers to the $n$ training examples in our dataset;
3     foreach $m$=1,..., $M$ do
4         Compute the difference between a predicted value $F(x_i) = \hat{y}_i$and the
            class label $y_i$. This value is sometimes called the pseudo-response or
            pseudo-residual. More formally, we can write this pseudo-residual as
            the negative gradient of the loss function with respect to the predicted
            values:

            $$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, \ i = 1, \ldots, n$$

            Note that in the notation above $F(x)$ is the prediction of the previous
            tree, $F_{m-1}(x)$;
5         Fit a tree to the pseudo-residuals rim. We use the notation $R_{jm}$to
            denote the $j = 1...Jm$ leaf nodes of the resulting tree in iteration $m$.
            For each leaf node Rjm, we compute the following output value:

            $$\gamma_{jm} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma)$$

            At this point, we can already note that leaf nodes $R_{jm}$ may contain
            more than one training example, hence the summation;
6         Update the model by adding the output values $\gamma_m$ to the previous tree:

            $$F_m(x) = F_{m-1}(x) + \eta \gamma_m$$

            However, instead of adding the full predicted values of the current
            tree $\gamma_m$ to the previous tree $F_{m-1}$ , we scale $\gamma_m$ by a learning rate $\eta$ ,
            which is typically a small value between 0.01 and 1. In other words,
            we update the model incrementally by taking small steps, which
            helps avoid overfitting.
```

Chapter 7

Applying Machine Learning to Sentiment Analysis

7.1 Introducing the bag-of-words model

The idea behind bag-of-words is quite simple and can be summarized as follows:

1. We create a vocabulary of unique tokens—for example, words—from the entire set of documents.
2. We construct a feature vector from each document that contains the counts of how often each word occurs in the particular document.

7.1.1 Assessing word relevancy via term frequency-inverse document frequency

you will learn about a useful technique called the term frequency-inverse document frequency (tf-idf), which can be used to downweight these frequently occurring words in the feature vectors. The tf-idf can be defined as the product of the term frequency and the inverse document frequency:

$$tf-idf(t,d) = tf(t,d) \times idf(t,d)$$

Here, $tf(t,d)$ is the term frequency, and $idf(t,d)$ is the inverse document frequency, which can be calculated as follows:

$$idf(t,d) = \log \frac{n_d}{1+df(d,t)} \tag{7.1}$$

Here, n_d is the total number of documents, and $df(d,t)$ is the number of documents, d, that contain the term t.

Equation 7.1 for the inverse document frequency implemented in scikit-learn is computed as follows:

$$idf(t,d) = \log \frac{1+n_d}{1+df(d,t)} \tag{7.2}$$

Similarly, the tf-idf computed in scikit-learn deviates slightly from the default equation we defined earlier:

$$tf-idf(t,d) = tf(t,d) \times (idf(t,d)+1)$$

Note that the "+1" in the idf equation is due to setting `smooth_idf=True`, which is helpful for assigning zero weight (that is, $idf(t,d) = log(1) = 0$) to terms that occur in all documents.

7.1.2 Processing documents into tokens

After successfully preparing the movie review dataset, we now need to think about how to split the text corpora into individual elements. One way to tokenize documents is to split them into individual words by splitting the cleaned documents at their whitespace characters.

In the context of tokenization, another useful technique is word stemming, which is the process of transforming a word into its root form.

7.2 Working with bigger data – online algorithms and out-of-core learning

Since not everyone has access to supercomputer facilities, we will now apply a technique called out-of-core learning, which allows us to work with such large datasets by fitting the classifier incrementally on smaller batches of a dataset.

The word2vec model

A more modern alternative to the bag-of-words model is word2vec, an algorithm that Google released in 2013 (*Efficient Estimation of Word Representations in Vector Space by T.Mikolov, K. Chen, G. Corrado, and J. Dean*,).
The word2vec algorithm is an unsupervised learning algorithm based on neural networks that attempts to automatically learn the relationship between words. The idea behind word2vec is to put words that have similar meanings into similar clusters, and via clever vector spacing, the model can reproduce certain words using simple vector math, for example, *king–man* + *woman* = *queen*.

Chapter 8

Predicting Continuous Target Variables with Regression Analysis

8.1 Fitting a robust regression model using RANSAC

Algorithm 7: RANdom SAmple Consensus (RANSAC) algorithm

```
1 begin
2     repeat
3         Select a random number of examples to be inliers and fit the model;
4         Test all other data points against the fitted model and add those points
            that fall within a user-given tolerance to the inliers;
5         Refit the model using all inliers;
6         Estimate the error of the fitted model versus the inliers;
7     until the performance meets a certain user-defined threshold or if a fixed
        number of iterations was reached;
```

8.2 Evaluating the performance of linear regression models

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

Another useful quantitative measure of a model's performance is the mean squared error (MSE) that is often used to simplify the loss derivative in gradient descent:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \tag{8.1}$$

Note that it can be more intuitive to show the error on the original unit scale, which is why we may choose to compute the square root of the MSE, called root mean squared error, or the mean absolute error (MAE), which emphasizes incorrect prediction slightly less:

$$MAE = \frac{1}{n}\sum_{i=1}^{m}|y^{(i)} - \hat{y}^{(i)}| \tag{8.2}$$

When we use the MAE or MSE for comparing models, we need to be aware that these are unbounded in contrast to the classification accuracy, for example. In other words, the interpretations of the MAE and MSE depend on the dataset and feature scaling.

Thus, it may sometimes be more useful to report the coefficient of determination (R^2), which can be understood as a standardized version of the MSE, for better interpretability of the model's performance. Or, in other words, R^2 is the fraction of response variance that is captured by the model. The R^2value is defined as:

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} \\ SSE &= \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 \\ SST &= \sum_{i=1}^{n}(y^{(i)} - \mu_y)^2 \end{aligned} \tag{8.3}$$

Now, let's briefly show that R^2 is indeed just a rescaled version of the MSE:

$$\begin{aligned} R^2 &= 1 - \frac{SSE}{SST} \\ &= 1 - \frac{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \mu_y)^2} \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned} \tag{8.4}$$

For the training dataset, R^2 is bounded between 0 and 1, **but it can become negative for the test dataset**. A negative R^2 means that the regression model fits the data worse than a horizontal line representing the sample mean. (In practice, this often happens in the case of extreme overfitting, or if we forget to scale the test set in the same manner we scaled the training set.) If $R^2 = 1$, the model fits the data perfectly with a corresponding $MSE = 0$.

8.3 Using regularized methods for regression

The most popular approaches to regularized linear regression are the so-called ridge regression, least absolute shrinkage and selection operator (LASSO), and elastic net.

Ridge regression is an L2 penalized model where we simply add the squared sum of the weights to the MSE loss function:

$$\begin{aligned} L(\mathbf{w})_{Ridge} &= \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda ||\mathbf{w}||_2^2 \\ ||\mathbf{w}||_2^2 &= \sum_{j=1}^{m} w_j^2 \end{aligned} \tag{8.5}$$

An alternative approach that can lead to sparse models is LASSO. Depending on the regularization strength, certain weights can become zero, which also makes LASSO useful as a supervised feature selection technique:

$$\begin{aligned} L(\mathbf{w})_{Lasso} &= \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda ||\mathbf{w}||_1 \\ ||\mathbf{w}||_1 &= \sum_{j=1}^{m} |w_j| \end{aligned} \tag{8.6}$$

A compromise between ridge regression and LASSO is elastic net, which has an L1 penalty to generate sparsity and an L2 penalty such that it can be used for selecting more than n features if $m > n$:

$$L(\mathbf{w})_{Elastic\ Net} = \sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 + \lambda_2 ||\mathbf{w}||_2^2 + \lambda_1 ||\mathbf{w}||_1 \tag{8.7}$$

Chapter 9

Working with Unlabeled Data – Clustering Analysis

9.1 Grouping objects by similarity using k-means

9.1.1 k-means clustering using scikit-learn

Algorithm 8: The k-means algorithm

```
1 begin
2     Randomly pick $k$ centroids from the examples as initial cluster centers;
3     repeat
4         Assign each example to the nearest centroid, $\mu^{(i)}, j \in \{1, \ldots, k\}$;
5         Move the centroids to the center of the examples that were assigned to it;
6     until the cluster assignments do not change or a user-defined tolerance or maximum number of iterations is reached;
```

A problem with k-means is that one or more clusters can be empty.

Feature scaling

When we are applying k-means to real-world data using a Euclidean distance metric, we want to make sure that the features are measured on the same scale and apply z-score standardization or min-max scaling if necessary.

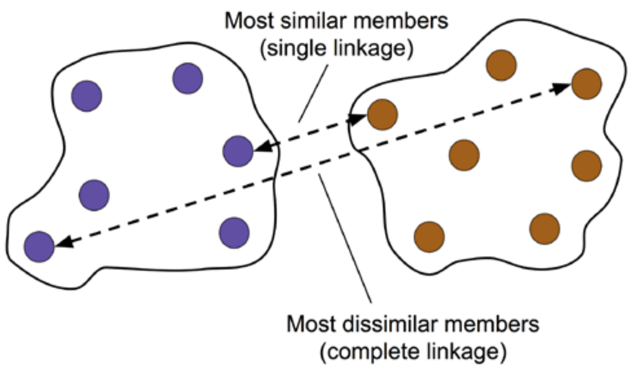

Figure 9.1: The complete linkage approach

9.2 Organizing clusters as a hierarchical tree

9.2.1 Grouping clusters in a bottom-up fashion

The two standard algorithms for agglomerative hierarchical clustering are single linkage and complete linkage. Using single linkage, we compute the distances between the most similar members for each pair of clusters and merge the two clusters 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 is shown in Figure 9.1:

Alternative types of linkages

Other commonly used algorithms for agglomerative hierarchical clustering include average linkage and Ward's linkage. In average linkage, we merge the cluster pairs based on the minimum average distances between all group members in the two clusters. In Ward's linkage, the two clusters that lead to the minimum increase of the total within-cluster SSE are merged.

9.3 Locating regions of high density via DBSCAN

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 neighboring points fall within the specified radius, ε

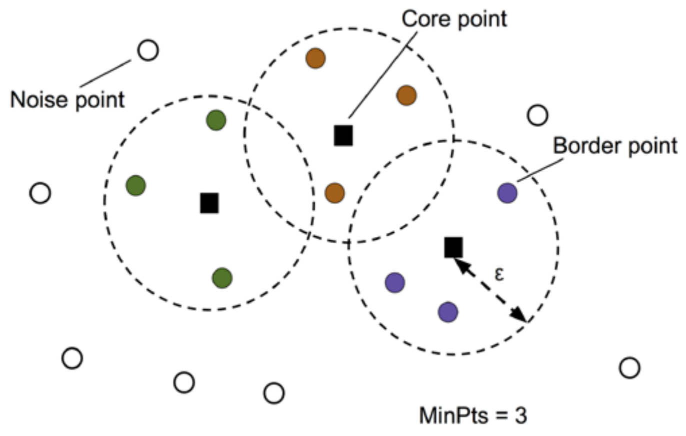

Figure 9.2: Core, noise, and border points for DBSCAN

- A **border point** is a point that has fewer neighbors 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**

Chapter 10

Implementing a Multilayer Artificial Neural Network from Scratch

10.1 Modeling complex functions with artificial neural networks

10.1.1 Introducing the multilayer neural network architecture

Adding additional hidden layers

We can add any number of hidden layers to the MLP to create deeper network architectures. Practically, we can think of the number of layers and units in an NN as additional hyperparameters that we want to optimize for a given problem task using the cross-validation technique,

10.1.2 Activating a neural network via forward propagation

Furthermore, we can generalize this computation to all n examples in the training dataset:

$$\begin{aligned} \boldsymbol{Z}^{(h)} &= \boldsymbol{X}^{(in)}\boldsymbol{W}^{(h)T} + \boldsymbol{b}^{(h)} \\ \boldsymbol{A}^{(h)} &= \sigma(\boldsymbol{Z}^{(h)}) \end{aligned} \tag{10.1}$$

Here, $\boldsymbol{X}^{(in)}$ is now an $n \times m$ matrix, and the matrix multiplication will result in an $n \times d$ dimensional net input matrix, $\boldsymbol{Z}^{(h)}$. Finally, we apply the activation function $\sigma(\cdot)$ to each value in the net input matrix to get the $n \times d$ activation matrix in the next layer.

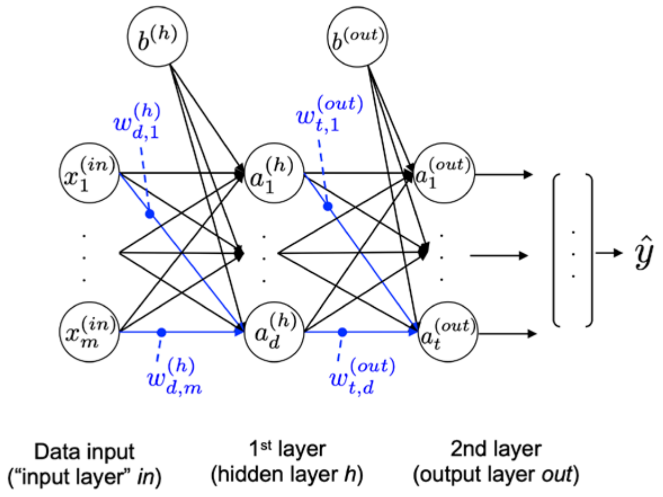

Figure 10.1: A two-layer MLP. We will use the *in* superscript for the input features, the *h* superscript for the hidden layer, and the *out* superscript for the output layer. For instance, $x_i^{(in)}$ refers to the *i*th input feature value, $a_i^{(h)}$ refers to the *i*th unit in the hidden layer, and $a_i^{(out)}$ refers to the *i*th unit in the output layer. Note that the *b*'s denote the bias units. In fact, $b^{(h)}$ and $b^{(out)}$ are vectors with the number of elements being equal to the number of nodes in the layer they correspond to. For example, $b^{(h)}$ stores d bias units, where *d* is the number of nodes in the hidden layer. The connection between the *k*th unit in layer *l* to the *j*th unit in layer $l+1$ will be written as $w_{j,k}^{(l)}$.