Chapter 9

Introduction to Statistical Modeling and Supervised Machine Learning

In this chapter, we introduce the basic concepts and ideas behind machine learning. We will outline how machine learning relate to traditional statistical approaches that you already might now (and as you will see, there is a lot of overlap), present different types of models, and discuss how to validate them.

In a later chapter (Chapter 12), we will then specifically apply the knowledge you gained from this chapter to the analysis of textual data, arguably one of the most interesting tasks in the computational analysis of communication.

In this chapter, we mostly focus on *supervised* machine learning (SML) – a form of machine learning, where we aim to predict a variable that, for at least a part of our data, is known. SML is usually related to *classification* and *regression* problems. To illustrate the idea, imagine that you are interested in predicting the gender based on Twitter biographies. You determine the gender for some of the biographies yourself and hand these examples over to the computer. The computer "learns" this *classification* from your examples, and can then be used to predict the gender for other twitter biographies for which you do not know the gender.

In unsupervised machine learning (UML), in contrast, you do not have such examples, and is usually related to *clustering* and *associations* problems. We have discussed

some of such techniques in section Section 8.3, in particular cluster analysis and principal component analysis (PCA). Later, in chapter [ADD REFERENCE TO TOPIC MODELLING CHAPTER], we will discuss topic models, an unsupervised method to extract so-called topics from textual data.

Even though both approaches can be combined (for instance, one could first reduce the amount of data using PCA, and then predict some outcome), they can be seen as fundamentally different, also from a theoretical and conceptual point of view. Unsupervised machine learning is a bottom-up approach and corresponds to an inductive reasoning: you do not have a hypothesis of, for instance, which topics are present in a corpus of text; you rather let the topics emerge from the data. Supervised machine learning, in contrast, is a top-down approach and can be seen as more deductive: you define a priori which topics to predict.

Both SML and UML approaches are widely used in the computational analysis of communication, and you will meet them across the book. In this chapter, however, we will focus on supervised learning.

9.1. Statistical modeling and prediction

Machine learning, many people joke, is nothing else than a fancy name for statistics. And, in fact, there is some truth to this: If you say "logistic regression", this will sound familiar to both statisticians and machine learning practitioners. Hence, it does not make much sense to distinguish between statistics on the one hand and machine learning on the other hand.

Still, there are some differences between traditional statistical approaches that you may have learned about in your statistics classes and the machine learning approach, even if they use some of the same mathematical tools. One may say that that the focus is a different one, and the objective we want to achieve may differ.

Let us illustrate this with an example. `mediause.csv`[1] contains a few columns from survey data on how many days per week respondents turn to different media types (*radio*, *newspaper*, *tv* and *internet*) in order to follow the news[2]. It also contains their *age* (in years), their *gender* (coded as female = 0, male = 1), and their *education* (on a 5-point scale).

A straightforward question to ask is how in howfar the sociodemographic characteristics of the respondents explain their media use. Social scientists would typically approach this question by running a regression analysis. Such an analysis tell us how some independent variables $x_1, x_2 \dots x_n$ can explain y. In an ordinary least square regression (OLS), we would estimate $y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$.

In a typical social-science paper, we would then interpret the coefficients that we estimated, and say something like: When x_1 increases by one unit, y increases by β_1. We sometimes call this "the effect of x_1 on y (even though, of course, it depends on the study design weather the relationship can really be interpreted as a causal effect). Additionally, we might look at the explained variance R^2, to assess how good the model fits our data. In Example 9.1 we use this regression approach to model the relationship of *age* and *gender* over the number of days per week a person reads a *newspaper*. We fit the linear model using the *stats* function `lm` in R and the *statsmodels* (module formula.api) function `ols` in Python.

Most traditional social-scientific analyses stop after reporting and interpreting the coefficients of *age* ($\beta = 0.0676$) and *gender* ($\beta = -0.0896$), as well as the total explained variance (19%) of such a model over our dependent variable *newspaper*.

But we can go a step further. Given that we have already estimated our regression equation, why not use it to do some *prediction*?

We have just estimated that

$$\text{newspaperreading} = -0.0896 + 0.0676 \cdot \text{age} + 0.1767 \cdot \text{gender}$$

[1] You can download the file from http://cssbook.nl/d/mediause.csv

[2] For a detailed description of the dataset, see Trilling [2013].

Python Code

```python
import pandas as pd
import statsmodels.formula.api as smf
df = pd.read_csv('http://cssbook.net/d/mediause.csv')
model = smf.ols(formula = 'newspaper ~ age + gender', data =
        df).fit()
# model.summary() would give a lot more info, but we only
        care about the coefficients:
model.params
```

R Code

```r
df = read.csv('http://cssbook.net/d/mediause.csv')
model = lm(formula = 'newspaper ~ age + gender', data = df)
# summary(model) would give a lot more info, but we only care
        about the coefficients:
model
```

Output

```
Intercept  -0.089560
age         0.067620
gender      0.176665
dtype: float64
```

Example 9.1: Obtaining a model through estimating an OLS regression

By just filling in the values for a 20 year old man, or a 40 year old woman, we can easily calculate the expected number of days such a person reads the newspaper per week, *even if no such person exists in the original dataset.*

We learn that

$\hat{y}_{man20} = -0.0896 + 0.0676 \cdot 20 + 1 \cdot 0.1767 = 1.4391$

$\hat{y}_{woman40} = -0.0896 + 0.0676 \cdot 40 + 0 \cdot 0.1767 = 2.6144$

This was easy to do by hand, but of course, we could do this automatically for a large and essentially unlimited number of cases. This could be as simple as shown in Example 9.2.

Python Code

```python
newdata = pd.DataFrame([{'gender':1, 'age':20}, {'gender': 0,
          'age':40} ])
model.predict(newdata)
```

R Code

```r
gender = c(1,0)
age = c(20,40)
newdata = data.frame(age, gender)
predict(model, newdata)
```

Output

```
0    1.439508
1    2.615248
dtype: float64
```

Example 9.2: Using the OLS model we estimated before to predict the dependent variable for new data where the dependent variable is unknown.

In doing so, we shift our attention from the interpretation of coefficients to the prediction of the dependent variable for new, unknown cases. We do not care about the coefficents per se, we just need them for our prediction. In fact, in many machine learning models, we will have so many of them that we do not even bother to report them.

As you see, this implies that we proceed in two steps: First, we use some data to estimate our model. Second, we use that model to make predictions.

We used an OLS regression for our first example, because it is very straightforward to interpret, and most of our readers will be familiar with it. However, a model can take the form of *any* function, as long as it takes some characteristics (or "features") of the cases (in this case, people) as input and return a prediction.

Using such a simple OLS regression approach for prediction, as we did in our example, can come with a couple of problems, though. For instance, even though we know that the output should be something between 0 and 7 (as that is the number of days in a week), but our model will happily predict that once a man reaches the age of 105 (rare, but not impossible), he will read a newspaper on 7.185 out of 7 days; and a one year old girl will even have a negative amount of newspaper reading. Also, more in general, it is quite an assumption to make that the relationship between these variables are linear – we will therefore discuss multiple models that do not make such assumptions later in this chapter. And, finally, in many use cases, we are actually not interested in getting an accurate prediction of a continuous number (a *regression* task), but rather in predicting a category. We may want to predict whether a tweet goes viral or not, whether a user comment is likely to contain offensive language or not, whether an article is more likely to be about politics, sports, economy, or lifestyle. In machine learning terms, these tasks are known as *classification*.

In the next section, we will outline key terms and concepts in machine learning. After that, we will then discuss specific models that you can use for different use cases.

Table 9.1: Some common machine learning terms explained

machine learning lingo	statistics lingo
feature	independent variable
label	dependent variable
labeled dataset	dataset with both independent and dependent variables
to train a model	to estimate
classifier (classification)	model to predict nominal outcomes
to annotate	to (manually) code (content analysis)

9.2. Concepts and Principles

The goal of Supervised Machine Learning can be summarized in one sentence: Estimate a model based on some data, and then use the model to predict the expected outcome for some new cases, for which we do not know the outcome yet. Which is exactly what we have done in the introductory example in Section 9.1.

But when do we need it?

In short, in any scenario where the following two preconditions are fulfilled. First, we have a large dataset (say, $100,000$ headlines) for which we want to predict to which class they belong (say, whether they are clickbait or not). Second, for a random subset of the data (say, $2,000$ of the headlines), we already know the class. For example because we have manually coded ("annotated") them.

Before we start using SML, though, we first need to have a common terminology. At the risk of oversimplifying matters, Table 9.1 provides a rough guideline of how some typical machine learning terms translate to statistical terms that you may be familiar with.

Let us explain them more in detail by walking through a typical SML workflow.

Before we start, we need to get a *labeled dataset*. It may be given to us, or we need to create it ourselves. For instance, often we can draw a random sample of our data and use techniques of manual content analysis [e.g., Riffe et al., 2019] to *annotate* (i.e., to manually code) the data. You can download an example for this process

(annotating the topic of news articles) from http://dx.doi.org/10.6084/m9.figshare.7314896.v1 [Vermeer, 2018].

It is hard to give a rule of thumb of how many labelled data you need. It depends heavily on the type of data you have (if it is a *binary* or a *multi-class* classification problem), and on how evenly distributed (*class balance*) they are (after all, having 10,000 annotated headlines doesn't help you if 9,990 are no clickbait and only 10 are). These reservations notwithstanding, it is fair to say that typical sizes in our field are (very roughly) speaking often in the order of 1,000 to 10,000 when classifying longer texts [Burscher et al., 2014, see], even though researchers studying less rich data sometimes annotate larger datasets [e.g., 60,000 social media messages in Vermeer et al., 2019].

Once we have established that this labelled dataset is available and have ensured that it is of good quality, we randomly split it into two datasets: a *training dataset* and a *test dataset*.[3] We will use the first one to train our model, and the second to test how good our model performs. Common ratios range from 50:50 to 80:20; and especially if the size of your labelled dataset is rather limited, you may want to have a slightly larger training dataset at the expense of a slightly smaller test dataset.

In Example 9.3, we prepare the dataset we already used in Section 9.1 for classification by creating a dichotomous variable (the label) and splitting it into a training and a test dataset. We use `y_train` to denote the training labels and `X_train` to denote the feature matrix of the training dataset; `y_test` and `X_test` is the corresponding test dataset. We set a so-called random-state seed to make sure that the random splitting will be the same when re-running the code. We can easily split these datasets using the *rsample* function `initial_split` in R and the *sklearn* function `train_test_split` in Python.

We now can *train our classifier* (i.e., estimate our model using the training dataset

[3] In Section 9.4, we discuss more advanced approaches, such as splitting into training, validation, and test dataset, and cross-validation.

Python Code

```python
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('http://cssbook.net/d/mediause.csv')

df['uses-internet'] = df['internet']>0
df.dropna(inplace=True)
print("How many people used online news at all?")
print(df['uses-internet'].value_counts())

X_train, X_test, y_train, y_test = train_test_split(df[['age'
        , 'education', 'gender']], df['uses-internet'],
        test_size=0.2, random_state=42)

print(f'We have {len(X_train)} training and {len(X_test)}
        test cases.')
```

R Code

```r
library(tidyverse)
library(rsample)
library(glue)

df = read.csv('http://cssbook.net/d/mediause.csv')
df = na.omit(df %>% mutate(usesinternet=recode(internet, .
       default=TRUE, `0`=FALSE)))

set.seed(42)
df$usesinternet = as.factor(df$usesinternet)
print("How many people used online news at all?")
print(table(df$usesinternet))


split = initial_split(df, prop = .8)
traindata = training(split)
testdata = testing(split)

X_train = select(traindata, c('age', 'gender', 'education'))
y_train = traindata$usesinternet
X_test = select(testdata, c('age', 'gender', 'education'))
y_test = testdata$usesinternet

print(glue("We have {nrow(X_train)} training and {nrow(X_test
       )} test cases."))
```

Output

```
How many people used online news at all?
True    1262
False    803
Name: uses-internet, dtype: int64
We have 1652 training and 413 test cases.
```

Example 9.3: Preparing a dataset for supervised machine learning

contained in the objects `X_train` and `y_train`). This can be as straightforward as estimating a logistic regression equation (we will discuss different classifiers in Section 9.3). It may be that we first need to create new independent variables, so-called features, a step known as *feature engineering*, for example by transforming existing variables, combining them, or by converting text numerical word frequencies. Example 9.4 shows how easy it is to train a classifier using the Naïve Bayes algorithm with packages *caret*/*naivebayes* in R and *sklearn* in Python (this approach will be better explained in Subsection 9.3.1).

But before we can actually use this classifier to do some useful work, though, we need to test how capable it actually is to predict the correct labels, given a set of features. One might think that we could just feed it the same input data (i.e., the same features) again and see whether the predicted labels match the actual labels of the test dataset. In fact, we could do that. But this test would not be strict enough: After

Python Code

```python
from sklearn.naive_bayes import GaussianNB


myclassifier = GaussianNB()
myclassifier.fit(X_train, y_train)

y_pred = myclassifier.predict(X_test)
```

R Code

```r
library(caret)
library(naivebayes)

myclassifier = train(x = X_train, y = y_train, method = "
        naive_bayes")
y_pred = predict(myclassifier, newdata = X_test)
```

Example 9.4: A simple Naïve Bayes classifier

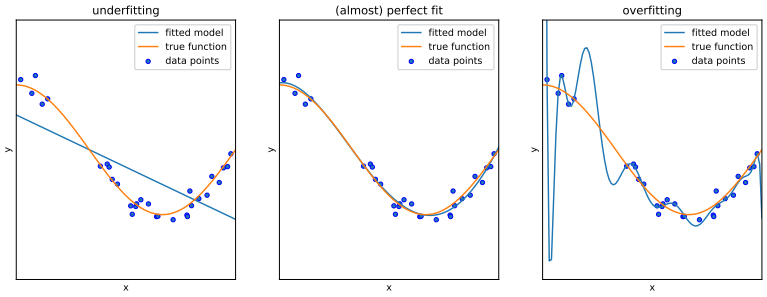

Figure 9.1: Underfitting and overfitting. Example adapted from https://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html

all, the classifier has been trained on exactly these data, and therefore one would expect it to perform pretty well. In particular, it may be that the classifier is very good in predicting its own training data, but fails at predicting other data, because it overgeneralizes some idiosyncrasy in the data, a phenomenon known as overfitting (see Figure 9.6).

Instead, we use the features of the *test dataset* (stored in the objects `X_test` and `y_test`) as input for our classifier, and evaluate in how far the predicted labels match the actual labels. Remember: the classifier has at no point in time seen the actual labels. Therefore, we can in fact calculate how often the prediction is right[4]

[4]We assume here that the manual annotation is always right; an assumption that one may, of course, challenge. However, in the absence of any better proxy for reality, we assume that this manual annotation is the so-called *gold standard* that reflects the *ground truth* as closely as possible, and that by definition cannot be outperformed. The manual annotation

Python Code

```python
from sklearn.metrics import confusion_matrix,
        classification_report

print('Confusion matrix:')
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

R Code

```r
print(confusionMatrix(y_pred, y_test))

print("Confusion matrix:")
confmat = table(testdata$usesinternet, y_pred)
print(confmat)

print('Precision for predicting True internet users and non-
        internet-users, respecitively:')
precision = diag(confmat) / rowSums(confmat)
print(precision)


print('Recall for predicting True internet users and non-
        internet-users, respecitively:')
recall = (diag(confmat) / colSums(confmat))
print(recall)
```

Output

```
Confusion matrix:
[[ 59 102]
 [ 42 210]]
              precision   recall f1-score  support

       False       0.58     0.37     0.45      161
        True       0.67     0.83     0.74      252

   micro avg       0.65     0.65     0.65      413
   macro avg       0.63     0.60     0.60      413
weighted avg       0.64     0.65     0.63      413
```

Example 9.5: Calculating precision and recall

As shown in Example 9.5, we can create a *confusion matrix* (generated with *caret* function `confusionMatrix` in R and *sklearn* function `confusion_matrix` in Python), and then estimate two measures: *precision* and *recall* (using base R calculations in R and *sklearn* function `classification_report` in Python). In a binary classification, the *confusion matrix* is a useful table in which each column usually represents the number of cases in a predicted class, and each row the number of cases in the real or actual class. With this matrix we can then estimate the number of *true positives* (TP) (correct prediction), *false positives* (FP) (incorrect prediction), *true negatives* (TN) (correct prediction) and *false negatives* (FN) (incorrect prediction).

For a better understanding of these concepts, imagine now that we build a sen-

might has some reliability measures to ensure its quality, such as the *intercoder reliability* which tests the degree of agreement between two or more annotators in order to check if our classes are well defined and the coders are doing their work correctly.

timent classifier, that predicts – based on the text of a movie review – whether it is a positive review or a negative review. Let us assume that the goal of training this classifier is to build an app that recommends the user only good movies. There are two things that we want to achieve: We want to find as many as possible positive films (recall), but we also want that the selection we found *only* contains positive films (precision).

Precision is calculated as $\frac{TP}{TP+FP}$, where TP are true positives and FP are false positives. For example, if our classifier retrieves 200 articles that it classifies as positive films, but only 150 of them indeed are positive films, then the precision is $\frac{150}{150+50} = \frac{150}{200} = 0.75$.

Recall is calculated as $\frac{TP}{TP+FN}$, where TP are true positives and FN are false negatives. If we know that the classifier from the previous paragraph missed 20 positive films, then the recall is $\frac{150}{150+20} = \frac{150}{170} = 0.88$.

In other words: Recall measures how many of the cases we wanted to find we actually found. Precision measures how much of what we have found actually is correct.

Often, we have to make a trade-off between precision and recall. For example, just retrieving *every* film would give us a recall of 1.0 (after all, we didn't miss a single positive film). But on the other hand, we retrieved all the negative films as well, so precision will be extremely low. It can depend on the task at hand whether precision or recall is more important. In Section 9.4, we discuss this tradeoff in detail, as well as other metrics such as *accuracy*, *f1-score* or the *area under the curve* (AUC).

9.3. From Naïve Bayes to Deep Neural Networks

To do supervised machine learning, we can use several models, all of which have different advantages and disadvantages, and are more useful for some use cases than

for others.

We limit ourselves to the most common ones in this chapter. The website of scikit-learn (http://www.scikit-learn.org) gives a good overview of more alternatives.

9.3.1. Naïve Bayes

The Naïve Bayes classifier is a very simple classifier that is often used as a "baseline". Before estimating more complicated and resource-intensive models, it is a good idea to estimate a simpler model first, to assess how much better the other model actually is. Sometimes, the simple model might even be just fine.

The Naïve Bayes classifier allows you to predict a binary outcome, such as: "Is this message spam or not?", "Is this article about politics or not?", "Will this go viral or not?". It, in fact, also allows you to do the same with more than one category, and both the Python and the R implementation will happily let you train a Naïve Bayes classifier on nominal data, such as whether an article is about politcs, sports, the economy, or something different.

For the sake of simplicity, we will discuss a binary example, though.

As its name suggests, a Naïve Bayes classifier is based on Bayes' theorem, and it is "naïve". It may sound a bit weird to call a model "naïve", but what it actually means is not so much that it is stupid, but that it makes very far-reaching assumptions about the data (hence, it is naïve). Specifically, it assumes that all features are independent from each other. Of course, that is hardly ever the case – for instance, in a survey data set, while age and gender indeed are fully independent from each other (unless, for instance, a war swept away a whole generation of males but not females), this is not the case for education, political interest, media use, and so on. And in textual data, whether a word W_1 is used is not independent from the use of word W_2 — after all, both are not randomly drawn from a dictionary, but depend on the topic of the text (and other things). Astonishingly, even though these assumptions are regularly violated, the Naïve Bayes classifier works reasonably well in practice.

The Bayes part of the Naïve Bayes classifier comes from the fact that it uses Bayes' formula,

$$P(A \mid B) = \frac{P(B \mid A) \cdot P(A)}{P(B)}$$

.

As a short refresher: The $P(A \mid B)$ can be read as: the probability of A, given B. Or: the probability of A if B is the case/present/true. Applied to our problem, this means that we are interested in estimating the probability of an item having a label, given a set of features:

$$P(label \mid features) = \frac{P(features \mid label) \cdot P(label)}{P(features)}$$

.

$P(label)$ can be easily calculated: It's just the fraction of all cases with the label we are interested in. Because we assume that our features are independent (remember, the "naïve" part), we can calculate $P(features)$ and $P(features \mid label)$ by just multiplying the probabilities of each individual feature. Let's assume we have three features, x_1, x_2, x_3. We now simply calculate the percentage of *all* cases that contain these features, $P(x_1), P(x_2)$ and $P(x_3)$. Then we do the same for the conditional probabilities and calculate the percentage of cases *with our label* that contain these features, $P(x_1 \mid label), P(x_2 \mid label)$ and $P(x_3 \mid label)$.

If we fill this in our formula, we get:

$$P(label \mid features) = \frac{P(x_1 \mid label) \cdot P(x_2 \mid label) \ \cdot P(x_3 \mid label) \cdot P(label)}{P(x_1) \cdot P(x_2) \cdot P(x_3)}$$

.

Remember that all we need to do to calculate this formula is: (1) counting how many cases we have in total; (2) counting how many cases have our label; (3) counting how many cases out of [1] have feature x; (4) counting how many cases out of [2] have

feature x. As you can imagine, doing this does not take much time to do, which makes the Naïve Bayes classifier such a fast and efficient choice. This may in particular be true if you have very many features (i.e., high-dimensional data).

Counting whether a feature is present or not, of course, is only possible for binary data. We could for example simply check whether a given word is present in a text or not. But what if our features are continous data, such as the number of times the word is present? We could dichotomize it, but that would throw away information. So, what we do instead, is that we estimate $P(x_i)$ using a distribution, for example a Gaussian, Bernoulli, or multinomial distribution. The core idea, though, stays the same.

Our examples in Section 9.2 illustrate how to do train a Naïve Bayes classifier. We first create the labels (whether someone uses online news at all or not), split our data into a training and a test dataset (here, we use 80% for training and 20% for testing) (Example 9.3), then fit (train) a classifier (Example 9.4), before we assess how well it predicts our training data (Example 9.5).

In section 9.4, we discuss in more detail how to evaluate different classifiers, but we will already look at some measures of how well our classifier performs.

First, the confusion matrix tells us how many non-users were indeed classified as non users (55), and how many (wrongly) as users (106).[5] That doesn't look very good; but on the other hand, 212 of the true users were correctly classified as such, and only 40 were not.

More formally, we can express this using precision and recall. When we are interested in finding true users, we get a precision of .67 ($\frac{212}{212+106}$) and a recall of .84 ($\frac{212}{212+40}$). However, if we want to know how good we are in identifying those who do *not* use online news, we do – as we saw in the confusion matrix – considerably worse: precision and recall are .58 and .34, respectively.

[5]These are the values from the Python example, the R example slightly differs, amongst other things due to different sampling.

9.3.2. Regression

Regression analysis does not make as strong an assumption about the independence of features as the Naïve Bayes classifier does. Sure, we have been warned about the dangers of multicollinearity in statistics classes, but correlation between features (for which is multicollinearity is a fancy term) affects the coefficients and their p values, but not the predictions of the model as a whole. To put it differently, in regression models, we do not estimate the probability of a label given a feature, independent of all the other features, but are able to "control for" their influence. In theory, this should make our models better, and also in practice, this regularly is the case. However, ultimately, it is an empirical question whether this is the case.

While we started this chapter with an example of an OLS regression to estimate a continous outcome (well, by approximation, as for "days per week" not all values make sense), we will now use a regression approach to predict nominal outcomes, just as in the Naïve Bayes example. The type of regression analysis to use for this is called *logistic regression*.

In a normal OLS regression, we estimate

$$y = \beta_o + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

But this gives us a continous outcome, which we do not want. In a logistic regression, we therefore use the sigmoid function to map this continous outcome to a value between 0 and 1. The sigmoid function is defined as $sigmoid(z) = \frac{1}{1+e^{-z}}$ and depicted in Figure 9.2.

Combining these formulas gives us:

$$P = \frac{1}{1 + e^{-(\beta_o + \beta_1 x_1 + \beta_2 x_2 = \ldots + \beta_n x_n)}}$$

Wait, you might say. Isn't P still continuous, even though it is now bounded between 0 and 1? Yes, it is. Therefore, after having estimated the model, we use a

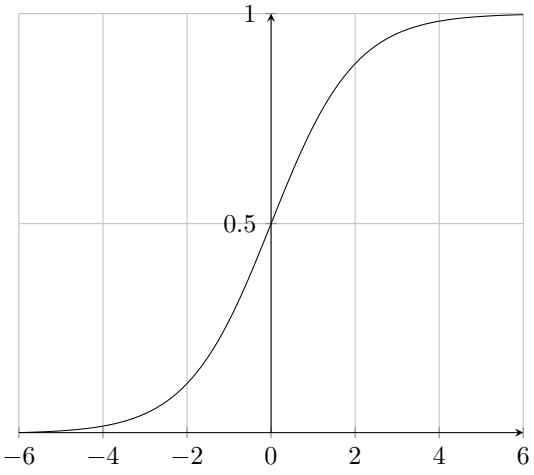

Figure 9.2: The sigmoid function

threshold value (typically, 0.5) to predict the label. If $P > 0.5$, we predict that the case is spam/about politics/will go viral, if not, we predict its not.

A nice side effect of this is that we still can use the probabilities in case we are interested in them, for example to figure out for which cases we are more sure in our prediction.

Just as with the Naïve Bayes classifier, also for logistic regression classifiers, Python and R will happily allow us to estimate models with multiple nominal outcomes instead of a binary outcome. In Example 9.6 w e fit the logistic regression using the *caret* method `logreg` in R and the *sklearn* (module linear_model) function `LogisticRegression` in Python.

And, of course, you actually can do OLS regression (or more advanced regression models) if you want to estimate a continuous outcome.

9.3.3. Support Vector Machines

Support Vector Machines (SVM) are another very popular and versatile approach to supervised machine learning. In fact, they are quite similar to logistic regression, but

Python Code

```python
from sklearn.linear_model import LogisticRegression
myclassifier = LogisticRegression(solver='lbfgs')
myclassifier.fit(X_train, y_train)

y_pred = myclassifier.predict(X_test)
```

R Code

```r
library(tidyverse)
library(caret)

myclassifier = train(x = X_train, y = y_train, method = 'glm'
        ,family = "binomial")
y_pred = predict(myclassifier, newdata = X_test)
```

Example 9.6: A simple logistic regression classifier

try to optimize a different function. In technical terms, SVM minimizes *hinge loss* instead of logistic loss.

What does that mean to us? When estimating logistic regressions, we are interested in estimating probabilities, while when training a SVM, we are interested in finding a plane (more specifically, a hyperplane) that best separates the datapoints of the two classes (e.g., spam vs non-spam messages) that we want to distinguish. This also means that a SVM does not give you probabilities associated with your prediction, but just the label. But usually, that's all that you want anyway.

Without going into mathematical detail here (for that, a good source would be Kelleher et al. [2015], we can say that finding the widest separating margin that we can achieve constructing a plane in a graphical space (SVM) versus optimizing a log-likelihood function (logistic regression) results in a model that is less sensitive to outliers, and tends to be more balanced.

There are a lot of graphical visualizations available, for example in the notebooks supplementing [VanderPlas, 2016] (https://jakevdp.github.io/PythonDataScienceHandbook/05.07-support-vector-machines.html). For now, it may suffice to imagine the two-dimensional case: we construct a line that separates two groups of dots *with the broadest possible margin*. The dots that the margin of this line just touches are called the "support vectors", hence the name.

You could imagine that sometimes, we may want to be a bit lenient about the margins. If we have thousands of data points, then maybe it is okay if one or two of these datapoints are, in fact, within the margin of the separating line (or hyperpane). We can control this with a parameter called C: For very high values, this is not allowed,

but the lower the value, the "softer" the margin is. In Section 9.4.3, we will show an approach to find the optimal value.

A big advantage of SVMs is that they can be extended to non-linear separable classes. Using a so-called kernel function or *kernel trick*, we can transform our data so that the dataset becomes linearly separable. Choices include but are not limited to multinomial kernels, the radial basis function (RBF), or Gaussian kernels. If we, for example, have a two concentric rings of datapoints (like a donut), then we cannot find a straight line separating them. But a RBF kernel can transfer them into a linearly separable space. The aforementioned online visualizations can be very instructive here.

Example 9.7 shows how we implement standard SVM to our data using the *caret* method `svmLinear3` in R and the *sklearn* (module svm) function `SVC` in Python. You can notice in the code that feature data is standardized or normalized (with Mean = 0 and Standard Deviation = 1) before to model training in order to have all the features measured at the same scale, as required by SMV.

Python Code

```python
from sklearn.svm import SVC
from sklearn import preprocessing

# !!! We normalize our features to have M = 0 and SD = 1
# This is necessary as our features are not measured on the
        same scale, which SVM requires
# It may also be OK to rescale to a range of [0:1] or [-1:1]

scaler = preprocessing.StandardScaler().fit(X_train)

X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

myclassifier = SVC(gamma='scale')
myclassifier.fit(X_train_scaled, y_train)

y_pred = myclassifier.predict(X_test_scaled)
```

R Code

```r
library(tidyverse)
library(caret)
library(LiblineaR)

# !!! We normalize our features to have M = 0 and SD = 1,
        which we do with the preProcess argument
# This is necessary as our features are not measured on the
        same scale, which SVM requires
# It may also be OK to rescale to a range of [0:1] or [-1:1]

myclassifier = train(x = X_train, y = y_train, preProcess = c
        ("center", "scale"), method = "svmLinear3")
y_pred = predict(myclassifier, newdata = X_test)
```

Example 9.7: A simple Support Vector Machine classifier

9.3.4. Decision Trees and Random Forests

In the models we discussed so far, we essentially were modeling linear relationships. If the value of a feature is twice as high, its influence on the outcome will be twice as

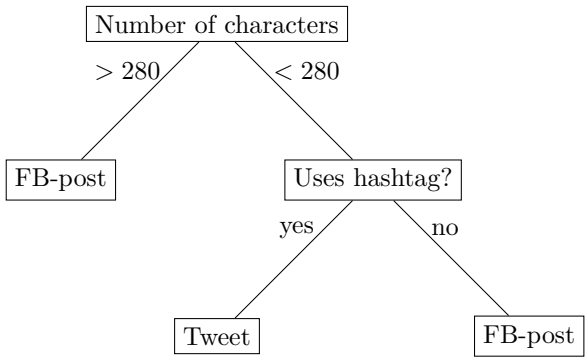

Figure 9.3: A simple decision tree

high as well. Sure, we can (and do, as in the case of the sigmoid function) apply some transformations, but we have not really considered yet how we can model situations in which, for instance, we care about whether the value of a feature is above (or below) a specific threshold. For instance, if we have a set of social media messages and want to model the medium where they most likely come from, then its length is very important information. If it is longer than 280 characters (or, historically, 140), then we can be *very* sure it is not from Twitter, even though the reverse is not necessarily true. But it does not matter at all whether it is 290 or 10000 characters long.

Entering this variable into a logistic regression, thus, would not be a smart idea. We could, of course, dichotomize it, but that would only partly solve the problem. In this example, we *know* how to dichotomize it based on our prior knowledge about the number of characters in a tweet, but this does not necessarily need to be the case; it might be something we need to estimate.

A step-wise decision, in which we first check one feature (the length), before checking another feature, can be modeles as a decision tree. Figure 9.3 depicts a (hypothetical) decision tree with three *leaves*.

Faced with the challenge to predict whether a social media message is a tweet or a Facebook post, we could predict 'Facebook post' if its length is greater than 280 characters. If not, we check whether it includes hashtags, and if so, we predict 'tweet',

otherwise, 'Facebook post'.

Of course, this simplistic model will be wrong at some times, because not all tweets have hashtags, and some Facebook posts actually do include hashtags.

While we constructed this hypothetical decision tree by hand, usually, we are more interested in learning such non-linear relationships from the data. This means that we do not have to determine the cutoff point ourselves, but also that we do not determine the order in which we check multiple variables by hand.

Decision trees have two nice properties. First, they are very easy to explain. In fact, a figure like Figure 9.3 is understandable for non-experts, which can be important in scenarios where for accountability reasons, the decision of a classifier must be as transparent as possible. Second, they allow us to approximate almost all non-linear relationships (be it not necessarily very accurately).

However, this comes at large costs. Formulating a model as a series of yes/no questions, as you can imagine, inherently uses a lot of nuance. More importantly, in such a tree, you cannot "move up" again. In other words, if you make a wrong decision early on in the tree (i.e., close to its root node), you cannot correct it any more. This rigidity makes decision trees also prone to overfitting: they may fit the training data very well, but may not generalize well enough to slightly different (test) data.

Because of these drawbacks, decision trees are seldom used in real-life classification tasks. Instead, one uses the ensemble model so-called random forests. Drawing random samples from the data, we estimate multiple decision trees – hence, a forest. To arrive at a final prediction, we then can let the trees "vote" on which label we should predict. This procedure is called "majority voting", but there are also other methods available. For example, *scikit-learn* in Python by default uses a method called `probabilistic prediction`, which takes into account probability values instead of simple votes.

In Example 9.8 we create a random forest classifier with 100 trees using the *caret* method `rf` in R and the *sklearn* (module ensamble) function `RandomForestClassifier` in Python.

Because random forests alleviate the problems of decision trees, but keep the ad-

Python Code

```python
from sklearn.ensemble import RandomForestClassifier
myclassifier = RandomForestClassifier(n_estimators=100)
myclassifier.fit(X_train, y_train)

y_pred = myclassifier.predict(X_test)
```

R Code

```r
library(tidyverse)
library(caret)
library(randomForest)

myclassifier = train(x = X_train, y = y_train, method = "rf")
y_pred = predict(myclassifier, newdata = X_test)
```

Example 9.8: A simple Random Forest classifier

vantage of being able to model non-linear relationships, they are frequently used when we expect such relationships (or have no idea about how the relationship looks like). Also, random forests may be a good choice if you have very different types of features (some nominal, some continuous, etc.) in your model. The same holds true if you have a lot (really a lot) features: Methods like SVM would require constructing large matrices in memory, which random forests do not. But if the relationships between your features and your labels are actually (approximately) linear, then you are probably better off with one of the other models we discussed.

9.3.5. A glimpse into Deep Learning

An extensive treatment of Deep Learning is out of the scope of this book (we recommend Géron [2019], instead), but we will give you a brief introduction, so that you can decide whether it is worth diving deeper into it.

In all of the models we discussed so far (which are also referred to as ”classical” or ”shallow” machine learning), we assume a direct relationship between the features and the labels. Grapically, we could depict this as direct arrows from all features to the output we want to predict (Figure 9.4).

In tasks such as image recognition or text classification, though, this often is too simplistic. For instance, when we want to predict whether an image shows a person or not, we do not really believe that the color of the pixels in the image (the “input layer”) can directly predict whether it is politician (the “output layer”). A more reasonable assumption is that these pixels can predict some more abstract concepts (a “hidden

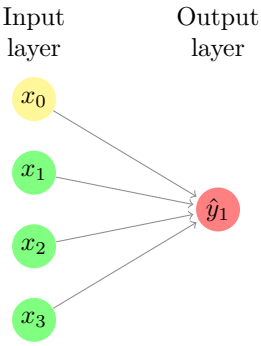

Figure 9.4: Schematic representation of a typical classical machine learning model

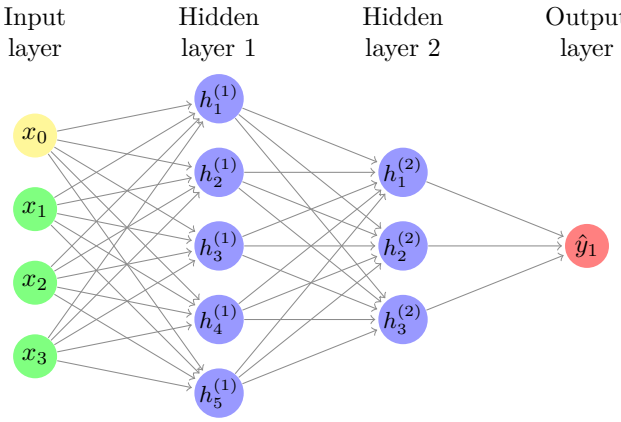

Figure 9.5: A neural network

layer") which then can predict another hidden layer or the output layer (Figure 9.5).

These layers are often not interpretable for humans (some even call this approach a "black box"), which can be problematic in situations where a classification needs to be transparent (for instance, to prevent discrimination). On the other hand can these layers often clearly enhance the accuracy of our predictions, which may be more important than explainability for many applications we are interested in.

It is up to the researcher to specify the number of layers, the number of neurons per layer, the mathematical functions to connect the neurons, and the general architecture of the network. Often, neural networks with many layers (e.g., ten) are referred to

as “deep networks” (*deep learning*). We will cover this approach with more detail in section Section 12.2 with *recurrent neural networks* for text classification and in [ADD REFERENCE TO CH 15 Introduction to Image and Video Data] with *convolutional neural networks* for image classification.

9.4. Validation and best practices

9.4.1. Finding a balance between precision and recall

In the previous sections, we have learned how to fit different models: Naïve Bayes, logistic regressions, support vector machines, and random forests. We have also had a first look at confusion matrices, precision, and recall.

But how do we find the best model? “Best”, here, should be read as “best for our purposes” – some models may be bad, and some may be good, but which one is really the best may depend on what matters most for us: Do we care more about precision or about recall? Are all classes equally important to us? And of course, other factors, such as explainability or computational costs may factor into our decision.

But in any event, we need to decide on which metrics to focus. We can then either manually inspect them and look, for instance, which model has the highest *accuracy*, or the best balance of precision and recall, or a recall higher than some threshold you are willing to accept.

If we build a classifier to distinguish spam messages from legitimate messages, we could ask the following questions:

precision Which percentage of what our classifier predicts to be spam really is spam?

recall What percentage of all spam messages has our classifier found?

accuracy In which percentage of all cases was our classifier right?

We furthermore have:

f1-score The harmonic mean of precision and recall: $F_1 = 2 \cdot \frac{precison \cdot recall}{precison + recall}$

AUC The AUC (Area under Curve) is the area under the curve that one gets when plotting the True Positive Pate (TPR) against the False Positive Rate (FPR) at various threshold settings. A perfect model will receive a value of 1.0, while random guessing between two equally probable classes will result in a value of 0.5

micro- and macroaverage Especially when we have more than two classes, we can calculate the average of measures such as precision, recall, or f1-score. We can do so based on the sperately calculated measures (macro), or based on the underlying values (TP, FP, etc.) (micro), which has different implications in the interpretation – especially if the classes have very different sizes.

So, which one to choose? If we really do not want to be annoyed by any spam in our inbox, we need a high recall (we want to find all spam messages). If, instead, we do not want to be sure that we do not accidentally throw away legitimate messages, we need a high precision (we want to be sure that all spam really is spam).

Maybe you say: well, I want both! You could look at the accuracy, a very straightforward to interpret measure. However, if you get many more legitimate messages than spam (or the other way round), this measure can be misleading: after all, even if your classifier finds almost none of the spam messages (it has a recall close to zero), you still get a very high accuracy, simply because there are so many legitimate messages. In other words, the accuracy is not a good measure when working with highly unbalanced classes.

Often, it is therefore a better idea to look at the harmonic mean of precision and recall, the f1-score, if you want to find a model that gives you a good compromise between precision and recall.

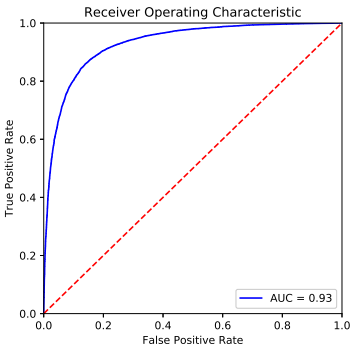

Figure 9.6: A ROC curve.

In fact, we can even fine-tune our models in such a way that they are geared towards either a better precision or a better recall.

As an example, let us take a logistic regression model. It predicts a class label (such as "spam" versus "legitimate"), but it can also return the assigned probabilities. For a specific message, we can thus say that we estimate its probability of being spam as, say, .65. Unless we specify otherwise, everything above .5 will then be judged to be spam, everything below as legitimate. But we could specify a different cutoff point: we could, for instance, decide to classify only everything above .7 as spam. This would give us a more conservative spam filter, with probably a higher precision at the expense of a lower recall.

We can visualize this with a so-called ROC (reveicer operator chraracteristic), a plot in which (Figure 9.6) we plot true positives against false positives at different thresholds. A good model extends until close to the upper left corner, and hence has a large area under the curve (AUC). If we choose a threshold at the left end of the curve, we get little false positives (good!), but also little true positives (bad!), if we go too far to the right, we get the other extreme. So, how can we find the best spot?

One approach would be to print a table with three columns: the false positive rate, the true positive rate, and the threshold value. You then decide which FPR-TPR

combination is most appealing to you, and use the corresponding threshold value.

The second approach (also knwon as Yoden's J) is to find the threshold value with the maximum distance between TPR and FPR, and use that one.

integrate code from https://github.com/damian0604/bdaca/blob/master/rm-course-2/week10/Determining%20the%20cutoff-point%20in%20logistic%20regression.ipynb. But first look up how much space we actually have...

9.4.2. Train, test, validate

By now, we have established which measures we can use to decide which model to use. For all of them, we have assumed that we split our labeled dataset into two: a training dataset and a test dataset. The logic behind it was simple: If we would calculate precision and recall on the training data itself, our assessment would be too optimistic – after all, our models have been trained on exactly these data, so predicting the label isn't too hard. Assessing the models on a different dataset, the test dataset, instead, gives us an assessment of how precision and recall look like if haven't seen the labels earlier – which is exactly what we want to know.

Unfortunately, if we calculate precision and recall (or any other metric) for multiple models on the same test dataset, and use these results to determine which metric to use, we can run into a problem: We may avoid overfitting of our model on the training data, we may now overfit it on the test data! After all, we could tweak our models as long until they fit our test data perfectly, even if this makes the predictions for other cases worse.

One way to avoid this is to split the original data into three datasets instead of two: a training dataset, a validation dataset, and a test dataset. We train multiple model configurations on the training dataset and calculate the metrics of interest for all of them on the validation dataset. Once we have decided on a final model, we calculate its performance (once) on the test dataset, to get an unbiased estimate of

its performance.

9.4.3. Cross-validation and grid search

In an ideal world, we would have a huge labelled dataset and do not need to worry about the decreasing size of our training dataset as we set aside our validation and test datasets.

Unfortunately, our labelled datasets in the real world have a limited size, and setting aside too many cases can be problematic. Especially if you are already on a tight budget, setting aside not only a test dataset, but also a validation dataset of meaningful size may lead to critically small training datasets. While we have addressed the problem of overfitting, this could lead to underfitting: We may have removed the only examples of some specific feature combination, for instance.

A common approach to address this issue is k-fold cross-validation. To do so, we split our data into k partitions, so-called folds. We then estimate our model k times, and each time leave *one* of the folds aside for validation. Hence, every fold is exactly one time the validation dataset, and exactly $k - 1$ times part of the training data. We then simply average the results of our k values for the evaluation metric we are interested in.

If our classifier generalizes well, we would expect that our metric of interest (e.g., the accuracy, or the f1-score, ...) is very similar in all folds. Example 9.9 performs a cross-validation based on the logistic regression classifier we build above. We see that the standard deviation is really low, indicating that there are almost no changes between the runs, which is great.

Running the same cross-validation on our random forest, instead, would produce not only worse (lower) means, but also worse (higher) standard deviations, even though also here, there are no dramatic changes between the runs.

Very often, cross-validation is used when we want to compare many different model specifications, for example to find optimal hyperparameters.

Python Code

```python
from sklearn.model_selection import cross_val_score
myclassifier = LogisticRegression(solver='lbfgs')
accuracy = cross_val_score(estimator=myclassifier, X=X_train,
        y=y_train, scoring='accuracy', cv=5)
print(accuracy)
print(f"M = {accuracy.mean():.2f}, SD = {accuracy.std():.3f}"
      )
```

R Code

```r
library(tidyverse)
library(caret)

myclassifier = train(x = X_train, y = y_train, method = 'glm'
       , family='binomial', metric='Accuracy',
                  trControl = trainControl(method = "cv",
                         number = 5, returnResamp ='all',
                         savePredictions = TRUE),)
print(myclassifier$resample)
print(myclassifier$results)
```

Python Output

```
[0.64652568 0.64048338 0.62727273 0.64242424 0.63636364]
M = 0.64, SD = 0.007
```

R Output

```
   Accuracy    Kappa parameter Resample
1 0.6303030 0.1525638    none    Fold1
2 0.6223565 0.1161266    none    Fold2
3 0.6484848 0.1942410    none    Fold3
4 0.6223565 0.1247647    none    Fold4
5 0.6858006 0.2914248    none    Fold5
  parameter Accuracy    Kappa AccuracySD  KappaSD
1      none 0.6418603 0.1758242 0.02678153 0.07143985
```

Example 9.9: Crossvalidation

Hyperparameters are parameters of the model that are not estimated from the data. These depend on the model, but could for example be the estimation method to use, the number of times a bootstrap should be repeated, etc. A very good example are the hyperparameters of support vector machines (see above): It is hard to know how soft our margins should be (the C), and we may also be unsure about the right kernel (Example 9.11), or in the case of a polinomial kernel, how many degrees we want to consider).

Using the help function (e.g., `RandomForestClassifier` in Python or the documentation of the modules you are using, you can look up which hyperparameters you can specify. For a random forest classifier, for instance, this includes the number of estimators in the model, the criterion, and whether or not to use bootstrapping. Example 9.10, Example 9.11, and Example 9.12 illustrates how you can automatically assess which values you should choose.

Supervised machine learning is one of the areas where you really see differences between Python and R. While in Python, virtually all you need is

Python Code

```
from sklearn.model_selection import GridSearchCV

myclassifier = RandomForestClassifier()

grid = {
    'n_estimators' : [10, 50, 100, 200],
    'criterion': ['gini', 'entropy'],
    'bootstrap': [True, False]
}
search = GridSearchCV(estimator=myclassifier,
                      param_grid=grid,
                      scoring='f1',
                      cv=5)
search.fit(X_train, y_train)
print(f'Using these hyperparameters {search.best_params_}, we get the best performance:')
print(classification_report(y_test, search.predict(X_test)))
```

Output

```
Using these hyperparameters {'bootstrap': True, 'criterion': 'gini', 'n_estimators': 50}, we get the best performance:
              precision    recall  f1-score   support

       False       0.42      0.35      0.38       161
        True       0.62      0.68      0.65       252

   micro avg       0.55      0.55      0.55       413
   macro avg       0.52      0.52      0.52       413
weighted avg       0.54      0.55      0.55       413
```

Example 9.10: A simple gridsearch in Python

available via *scikit-learn*, in R, we often need to combine *caret* with various libraries providing the actual models. In contrast, all components we need for machine learning in Python are developed within one package, which leads to less friction. This is what you see in the gridsearch examples in this section. In scikit-learn, *any* hyperparameter can be part of the grid, but no hyperparameter has to be. Note that in R, in contrast, you cannot (at least, not easily) put any parameter of the model in the grid. Instead, you can look up the "tunable parameters" which *must* be present part of the grid in the caret documentation. This means that an exact replication of the grid searches in Example 9.10 and Example 9.11 is not natively supported using *caret* and requires either manual testing or writing a so-called caret extension.

Python Code

```
from sklearn.model_selection import GridSearchCV

myclassifier = SVC(gamma='scale')

grid = {
    'C' : [100, 1e4],
    'kernel': ['linear','rbf', 'poly'],
    'degree': [3,4]
}

search = GridSearchCV(estimator=myclassifier,
                      param_grid=grid,
                      scoring='f1',
                      cv=5,
                      n_jobs=-1, # use all cpus
                      verbose=10)
search.fit(X_train_scaled, y_train)
print(f'Using these hyperparameters {search.best_params_}, we get the best performance:')
print(classification_report(y_test, search.predict(X_test_scaled)))
```

Output

```
Fitting 5 folds for each of 12 candidates, totalling 60 fits
Using these hyperparameters {'C': 100, 'degree': 3, 'kernel': 'poly'}, we get the best performance:
              precision   recall f1-score  support

       False      0.58     0.04     0.08      161
        True      0.62     0.98     0.76      252

   micro avg      0.62     0.62     0.62      413
   macro avg      0.60     0.51     0.42      413
weighted avg      0.60     0.62     0.49      413
```

Example 9.11: A gridsearch in Python using multiple CPUs

While in the end, you can find a supervised machine learning solution for all your use cases in R as well, if supervised machine learning is at the core of your project, it may save you a lot of cursing to do this in Python.

R Code

```r
# Create the grid of parameters
grid <- expand.grid(Loss=c('L1','L2'),
                    cost=c(100,1000))

# Train the model using our previously defined parameters
gridsearch = train(x = X_train, y = y_train, preProcess = c("center", "scale"),
                   method = "svmLinear3",
                   trControl = trainControl(method = "cv", number = 5),
                   tuneGrid = grid)
gridsearch
```

Output

```
L2 Regularized Support Vector Machine (dual) with Linear Kernel

1653 samples
   3 predictor
   2 classes: 'FALSE', 'TRUE'

Pre-processing: centered (3), scaled (3)
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 1322, 1322, 1323, 1322, 1323
Resampling results across tuning parameters:

  Loss  cost  Accuracy  Kappa
  L1     100  0.6243248  0.04526380
  L1    1000  0.4930074 -0.05516973
  L2     100  0.6436986  0.17939011
  L2    1000  0.6436986  0.17939011

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were cost = 100 and Loss = L2.
```

Example 9.12: A gridsearch in R. Note that in R, not all parameters are "tunable" using standard *caret*. Therefore, an exact replication of the grid searches in Example 9.10 and Example 9.11 would requires either manual comparisons or writing a so-called caret extension.