CHAPTER

19

A step-by-step tutorial on how to build a machine learning model

Sandra Vieira[1], *Rafael Garcia-Dias*[1], *Walter Hugo Lopez Pinaya*[1,2]

[1] **Department of Psychosis Studies, Institute of Psychiatry, Psychology & Neuroscience, King's College London, London, United Kingdom;** [2] **Centre of Mathematics, Computation, and Cognition, Universidade Federal do ABC, Santo André, São Paulo, Brazil**

19.1 Introduction

The growing popularity of machine learning has led to the development of several tools aimed at making the application of this approach accessible to the machine learning novice. These efforts have resulted in tools such as PRoNTo (Schrouff et al., 2013) and NeuroMiner (https://pronia.eu/neurominer/) that do not require any programming skills. However, while such tools can be very useful, their simplicity comes at the cost of transparency and flexibility. Learning how to program a machine learning pipeline (even if a simple one) is an excellent way of gaining insight into the strengths of this analytical approach as well as the possible "distortions" that may take place along the machine learning pipeline (Tandon & Tandon, 2018). In addition, it also allows for greater flexibility, such as using any machine learning algorithm or data modality of interest. Despite the clear benefits of learning how to program a machine learning pipeline, many researchers find it challenging to do so and do not know how to go about it.

In this chapter, we provide a step-by-step tutorial on how to implement a standard supervised machine learning pipeline using scikit-learn, a widely popular and easy to use machine learning library for the Python programming language. At each step of the pipeline, we provide a brief rationale and explanation of the code and, when relevant, we refer the reader to useful external resources that provide further examples or a more in-depth discussion. The sample code follows the same workflow described in Chapter 2. The reader is encouraged to refer to this chapter for a more in-depth theoretical explanation of the main elements of the machine learning pipeline.

This tutorial is aimed at the machine learning beginner. For this reason, it assumes minimal previous programming knowledge. However, the reader is encouraged to familiarize themselves with basic Python. Python is a widely used programming language for which there are several resources available. Intermediate-level users should also find this tutorial useful. The sample code is structured in a way that it is easy to remove, add, or replace certain techniques with others. This way, the reader can experiment with different approaches and/or build on the code to develop more sophisticated pipelines. The implementation follows a rigorous methodology to avoid common errors, such as double dipping, and obtain reliable results.

The sample code and toy dataset used in this tutorial are freely available and can be downloaded from https://mlmh-lab.github.io/mlbook/. This allows the reader to follow the explanation along the tutorial while also applying the code themselves. We first provide a brief instruction on how to install Python and all the necessary libraries, as well as how to access the toy dataset and online version of the code.

19.2 Installing Python and main libraries

In this tutorial, the source code is written using Python 3. As most programming languages, Python is organized in libraries. Each library includes a set of dedicated functions for a specific purpose. In this tutorial, we use the following libraries:

- Pandas
- Numpy
- Scipy
- Matplotlib and seaborn
- Scikit-learn

Pandas and numpy are widely used libraries to load, manipulate, and summarize data. While pandas is used to handle tabular data (i.e., data arranged in a table, with rows and columns), numpy is a more general-purpose library. We also use scipy, a fundamental library for scientific computing, to run some univariate statistics to explore the data and prepare them for the statistical analysis. Matplotlib and seaborn are libraries for data visualization. This can be very useful when exploring data or summarizing the results. Finally, scikit-learn, or more commonly known as sklearn, is arguably the most popular and accessible machine learning Python library. It is a high-level library, which means that many complex applications have been wrapped in much simpler, shorter, and easy-to-use functions.

To run the sample code in this tutorial, the reader will need to have Python 3, as well as all the above libraries, installed. The simplest way to do this is to download and install Anaconda. This is a free and open-source distribution of the Python programming language for scientific computing that aims to simplify package management and deployment. Anaconda can be downloaded and installed on Windows, MacOS, and Linux from the following website: https://anaconda.com/distribution/. Anaconda comes with the Conda package management that makes it easy to install additional libraries if necessary (https://conda.io/projects/conda/en/latest/index.html).

19.3 How to read this chapter

In this chapter, the reader will find different text styles that distinguish between different kinds of information. Blocks of code, called snippets, will appear in a box such as the one below. Snippets are numbered to make it easier to refer to different stages of the analysis. The output of each snippet is displayed below the code.

```
print('Hello world!')                                                  0
```

```
Hello world!
```

In some cases, a line of code may be too long and has to be split over two lines. This will be indicated with "\" (note that this is not in the online version of the code). The online version of the sample code is written in a notebook format. The '>>>' symbol indicates a command which output will be displayed within the notebook. For some snippets, the output may be too long, in which case only a portion of the output is shown. This will be flagged with " … ." References to the main libraries or specific tools from a particular library in the text are shown in courier new font.

19.4 Using brain morphometry to classify patients with schizophrenia and healthy controls

The example used in this tutorial involves the classification of patients with schizophrenia (SZ) and healthy controls (HC) using neuroanatomical data. The toy dataset simulating the output from FreeSurfer (Fischl, 2012) contains the gray matter volume and thickness of 101 brain regions, along with the ID, age, gender, and diagnosis for each participant. A summary of the final pipeline of this tutorial is illustrated in Fig. 19.1. For the purposes of this exercise, we assume that feature extraction, i.e., the process of extracting the volumes and thickness values from the raw structural magnetic resonance imaging (MRI) images, has already been done for us. Therefore, we start our tutorial by preparing the data for the machine learning analysis. In this step, we explore the data for missing data, confounding variables, and class imbalance and discuss how to address each one of these issues. Next, we define the cross-validation

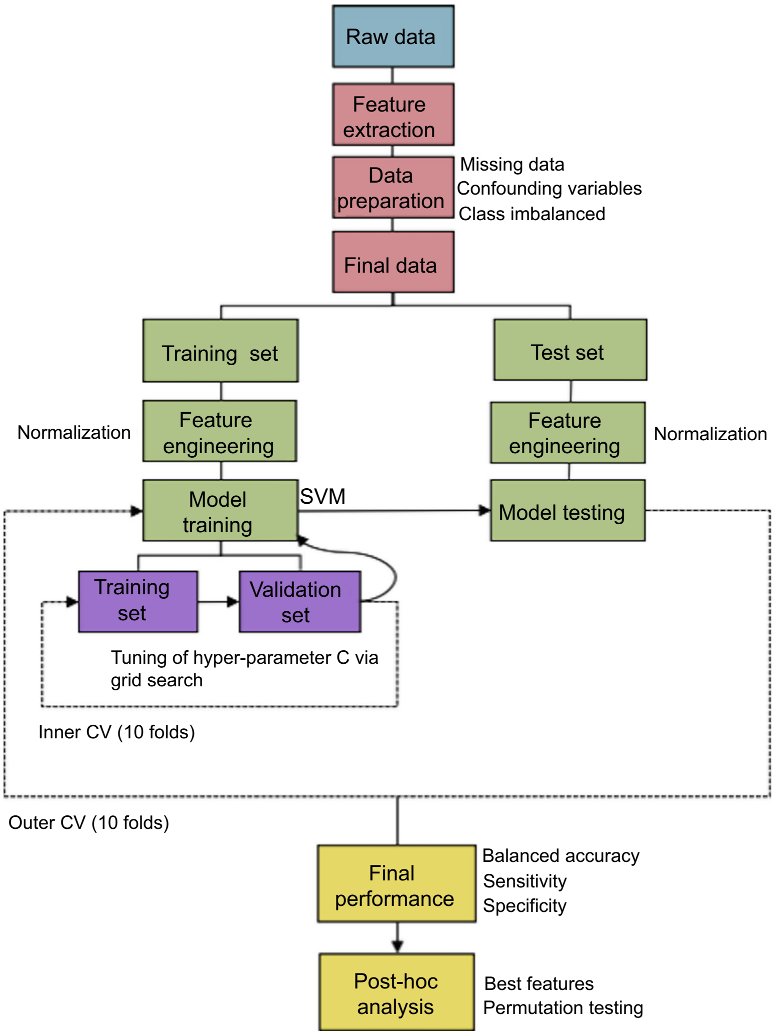


FIGURE 19.1 Summary of the machine learning pipeline implemented in this tutorial.

(CV) scheme with 10 iterations (outer CV). At each iteration, the training and test sets undergo data transformations separately to avoid "knowledge leakage." A Support Vector Machine (SVM) model is then fitted to the training set. As discussed in Chapter 6, SVM relies on the hyperparameter C. To decide which value of C to use, we create an inner CV with 10 folds. This means that, for every value of C we want to test, an SVM model is trained and tested 10 times; the final performance for a given value of C is then estimated by averaging all 10 performances. The best C parameter is then used to train an SVM model on the whole training set as defined by the outer fold. The performance of this trained model is measured in the test set. This whole process is repeated 10 times (i.e., 10 iterations for the outer CV). Once completed, the final performance of our SVM model is calculated by averaging the performance metrics from the 10 outer folds. Finally, we investigate which features were more important in driving the model's predictions and test the final performance of our model for statistical significance.

19.5 Sample code

The machine learning pipeline used in our example includes the following components: problem formulation, data preparation, feature engineering, model training, model evaluation, and post hoc analysis. Before we start, we first need to import all the necessary libraries, set random seed to a fixed value, and organize our workspace.

19.5.1 Importing libraries

The libraries needed to compute our machine learning analysis are not loaded by default. Therefore, it is good practice to begin the code file by importing all the libraries we are going to need. In addition to `pandas`, `numpy`, `scipy`, `seaborn`, `matplotlib`, and `sklearn`, we also use the `pathlib` module to organize our folders and the warnings module to suppress any unhelpful `warnings` during our analysis (both come with Python, hence we do not need to install them). If the reader decides to modify the code, we recommend reactivating the warnings by suppressing the last line in snippet 1. Understanding the warnings can help the reader avoid mistakes and debug the code. To make the code simpler to read, it is common to assign an alias when importing libraries that we use several times. For example, the pandas library is typically imported as pd. This way, we simply type pd every time we want to call this library.

```python
# Store and organize output files
from pathlib import Path

# Manipulate data
import numpy as np
import pandas as pd

# Plots
import seaborn as sns
import matplotlib.pyplot as plt

# Statistical tests
import scipy.stats as stats

# Machine learning
from sklearn.svm import LinearSVC
from sklearn.externals import joblib
from sklearn.metrics import balanced_accuracy_score, confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedKFold, GridSearchCV

# Ignore WARNING
import warnings

warnings.filterwarnings('ignore')
```

19.5.2 Set random seed

Some steps in our analysis will be subjected to randomness. For example, we might want to randomly remove some participants during data cleaning. Likewise, when defining the CV scheme, the train/test partition at each iteration is also done at random. In Python, this randomness can be controlled by setting a "seed value" to a fixed value. Not defining a specific seed value means that the variables that rely on this element of randomness will behave differently every time we run our code. For example, the train/test partition at each iteration will be different, which may lead to different model performance. Therefore, we set the seed value to a fixed number to guarantee that we get the same results every time we run the code. Some functions will need the random seed passed again as a parameter.

```python
random_seed = 1
np.random.seed(random_seed)
```

19.5.3 Organize the workspace

Before we start with our analysis, we should first create the structure of folders where we can store all the results. During this tutorial, the reader may wish to test different strategies along the machine learning pipeline, for example, different preprocessing strategies or machine learning algorithms. After a great deal of testing, it can be easy to lose track of which results refer to which strategy. It is good practice to give each experiment a name, create a folder in the results directory with the same name, and store the experiment outputs in this directory.

```
results_dir = Path('./results')
results_dir.mkdir(exist_ok=True)

experiment_name = 'linear_SVM_example'
experiment_dir = results_dir / experiment_name
experiment_dir.mkdir(exist_ok=True)                                    3
```

19.5.4 Problem formulation

Having a well-framed problem is essential when conducting any kind of project, especially in machine learning where there may be many possible ways of analyzing the same dataset. In this tutorial, our machine learning problem is as follows:

Classify patients with SZ and HC using structural MRI data.

From this formulation we can derive the main elements of our machine learning problem:

- **Features**: Structural MRI data
- **Task**: Binary classification
- **Target**: Patients with SZ and HC

19.5.5 Data preparation

The aim of this step is to perform a series of statistical analyses to get the data ready for the machine learning model. Here, different statistical analyses may be appropriate depending on the nature of the machine learning problem and the type of data.

19.5.5.1 Loading the data

In this tutorial, we use tabular data with columns as the features, targets, and demographic data and rows as participants. The data are stored as a comma-separated values (CSV) file. We use the function `read_csv( )` `from pandas` to load the csv file. This function loads the data into an object type called dataframe that we have named `dataset_df`.

```
dataset_file = Path('./Chapter_19_data.csv')
dataset_df = pd.read_csv(dataset_file, index_col='ID')                 4
```

In our dataset, diagnosis and gender are defined by words. Sometimes, people store their information using different names; for example, instead of using "sz" in the column diagnosis, we can use the word “schizophrenia” to define that the row in question belongs to a patient. To make it easier to adapt this code to different formats, we have defined our notation at the beginning of the code.

```
patient_str = 'sz'
healthy_str = 'hc'
male_str = 'M'
female_str = 'F'                                                        5
```

Let us start by focusing on the first six rows of the data. Selecting subsections of a dataframe using pandas is straightforward. There are different ways to do this. Here, we simply indicate the wanted indexes from the dataframe (note that the first row is indexed 0 and the last row is not included).

```
>>> dataset_df[0:6]                                                     6
```

```
      Diagnosis Gender   Age  ...  rh insula thickness
ID                            ...
c001         hc      M  22.0  ...             2.645844
c002         hc      F  24.0  ...             2.673699
c003         hc      F  22.0  ...             2.795989
c004         hc      F  30.0  ...             2.731654
c005         hc      M  31.0  ...             2.607771
c006         hc      F   NaN  ...             2.606643
```

From the output, we can see the column names at the top and the data of the first six participants. The columns include the diagnosis, gender, and age, as well as the gray matter volume and thickness for several brain regions. ID was set as the column index in snippet 4. We can see that there is at least one value missing (row `c006`). We will deal with this later.

It may also be useful to know the name of all the features available in the dataset. To do this, we simply ask for the names of the columns of our data.

```
>>> dataset_df.columns.tolist()                                         7
```

```
['Diagnosis',
 'Gender',
 'Age',
 'Left Lateral Ventricle',
 'Left Inf Lat Vent',
 'Left Cerebellum White Matter',
 'Left Cerebellum Cortex',
...
 'rh frontalpole thickness',
 'rh temporalpole thickness',
 'rh transversetemporal thickness',
 'rh insula thickness']
```

Next, let us check the size of the dataset.

```
print('Number of features = %d' % dataset_df.shape[1])
print('Number of participants = %d' % dataset_df.shape[0])                8
```

```
Number of features = 172
Number of participants = 740
```

To recreate some of the most common issues when building a machine learning pipeline, our data preparation stage will check the dataset for the following:

- Missing data
- Data imbalance with respect to the labels
- Confounding variables

19.5.5.2 Missing data

Most machine learning models do not support data with missing values. Therefore, it is important to check if there are any missing values in our `dataset_df`. Below we use the function `isnull()` from pandas to identify how many missing data are there for each feature and in total, as well as the IDs of the participants with missing data.

```
null_lin_bool = dataset_df.isnull().any(axis=1)
null_cols = dataset_df.columns[dataset_df.isnull().any(axis=0)]

n_null = dataset_df.isnull().sum().sum()
print('Number of missing data = %d' % n_null)
subj_null = dataset_df[null_lin_bool].index
print('IDs: %s' % (', ').join(subj_null.tolist()))

>>> pd.DataFrame(dataset_df[null_cols].isnull().sum(), columns=['N missing'])   9
```

```
Number of missing data = 43
IDs: c006, p149, p150, p156, p157, p175, p195, p196, p197, p210, p211, p
212, p227, p228, p229, p264, p265, p266, p267, p268, p269, p270, p271, p
281, p282, p283, p289, p302, p303, p307, p311, p312, p319, p321, p356, p
357, p358, p359, p360, p361, p362, p363, p364

     N missing
Age         43
```

We can see that there are 43 missing values for age. Not having this information does not allow for a thorough assessment of imbalanced demographic data, which could be problematic when interpreting the results. There are several options to go from here with different degrees of complexity (more information on these options can be found in Chapter 14). Because removing these participants would only result in losing 6% of the total data, we will simply remove them. We can do this by using the function dropna() from pandas.

```
dataset_df = dataset_df.dropna()
print('Number of participants = %d' % dataset_df.shape[0])                10
```

```
Number of participants = 697
```

As expected, the new dataframe has now 43 fewer participants than before.

19.5.5.3 *Class imbalance*

Next, let us check the number of total participants in each class.

```
>>> dataset_df['Diagnosis'].value_counts()                                    11
```

```
hc    367
sz    330
Name: Diagnosis, dtype: int64
```

There are a total of 367 controls and 330 patients in our dataset. There does not seem to be a large imbalance between classes. However, the two classes are not perfectly matched. As we mentioned in Chapter 2, this can cause problems when estimating the model's performance. One option would be to downsample the HC to match the SZ group. However, this would mean losing more data in addition to the 6% we have already discarded, which is undesirable. Because the imbalance is not too large, we will keep the same data and use balanced accuracy as our performance metric of choice as well as a stratified CV scheme to ensure the same proportion SZ/HC across the CV iterations.

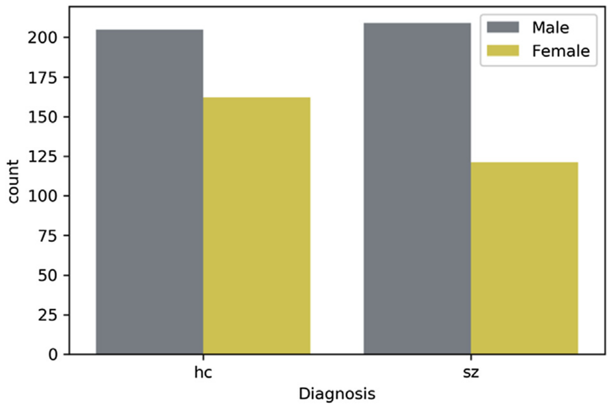
19.5.5.4 *Confounding variables*

There are many potential confounding variables one might want to inspect. Here, we will investigate two obvious ones: gender and age.

A simple way to investigate gender as a possible confounder is to verify the proportion of males and females in the patient and control groups. Let us start by visualizing the proportion of genders in each group using `seaborn`. Plotting data using this library is straightforward (for more information and examples of plots, see https://seaborn.pydata.org). Note that `seaborn` operates based on another library called `matplotlib`, the most widely used plotting library in Python. To edit some elements in our figures (e.g., change the legend box form "M" and "F" to "Male" and "Female"), we will also use `matplotlib` (more information about matplotlib in https://matplotlib.org).

```
sns.countplot(x='Diagnosis',
              hue='Gender',
              data=dataset_df,
              palette=['#839098', '#f7d842'])

plt.legend(['Male', 'Female'])
plt.show()                                                                    12
```



We can see that there is a fairly similar number of males in the two groups. However, the control group has more females than the patient group. In addition to visualizing the data, it is good practice to always perform an appropriate statistical test, even if there is no noticeable bias in the visual inspection. Because gender is a categorical variable, we will apply a chi-square test of homogeneity to check if this difference is statistically significant. In this case, we want to test the null hypothesis that the proportion of women in the HC group does not differ from the proportion of women in the patient group (equivalent to test if the proportion of men in the HC group does not differ from the proportion of men in the patient group).

$$\begin{aligned} \text{H0}: \text{Proportion men} / \text{women}_{\text{Healthy control}} \\ = \text{Proportion men/women}_{\text{Patients}} \end{aligned}$$

```
# Create the contingency table
contingency_table = pd.crosstab(dataset_df['Gender'], dataset_df['Diagnosis'])
print(contingency_table)

# Perform the homogeneity test
chi2, p_gender, _, _ = stats.chi2_contingency(contingency_table,
                                                correction=False)

print('Gender')
print('Chi-square test: chi2 stats = %.3f p-value = %.3f' % (chi2, p_gender)) 13
```

```
Diagnosis    hc   sz
Gender
F           162  121
M           205  209

Gender
Chi-square test: chi2 stats = 4.026 p-value = 0.045
```

The results above show that there is indeed a statistically significant difference between the two classes with respect to gender (p-value $<.05$).

This may be an issue, as the impact of gender on brain morphology is well-established. Therefore, the machine learning algorithm may use brain features that are associated with gender differences to distinguish between HC and SZ, as opposed to differences related to the disorder under investigation.

To mitigate this potential source of bias, we will randomly remove one female in the HC class iteratively, until there is no longer a statistically significant difference in the proportion of male/female between the two classes.

```python
print('Removing participant to balance gender...')
while p_gender < 0.05:
    # Randomly select a woman from healthy controls
    hc_women = dataset_df[(dataset_df['Diagnosis'] == healthy_str) &
                          (dataset_df['Gender'] == female_str)]

    indexes_to_remove = hc_women.sample(n=1, random_state=1).index

    # Remove her from dataset
    print('Dropping %s' % str(indexes_to_remove.values[0]))
    dataset_df = dataset_df.drop(indexes_to_remove)

    contingency_table = pd.crosstab(dataset_df['Gender'],
                                     dataset_df['Diagnosis'])

    chi2, p_gender, _, _ = stats.chi2_contingency(contingency_table,
                                                  correction=False)

    print('new p-value = %.3f' % p_gender)

print('Gender')
print('Chi-square test: chi2 stats = %.3f p-value = %.3f' % (chi2, p_gender))

# Check new sample size
contingency_table = pd.crosstab(dataset_df['Gender'], dataset_df['Diagnosis'])
print(contingency_table)                                                   14
```

```
Removing participant to balance gender...
Dropping c082
new p-value = 0.049

Dropping c083
new p-value = 0.054

Gender
Chi-square test: chi2 stats = 3.698 p-value = 0.054

Diagnosis   hc   sz
Gender
F          160  121
M          205  209
```

From the output, we can see that the chi-square test is no longer statistically significant after removing two female controls.

Next, let us check for any imbalance with respect to age. The idea is to test the null hypothesis that the mean (or median) of age in the HC group does not differ from the mean (or median) of age in the SZ group. One way to do this is by using the parametric student's t-test for two samples. Importantly, this test assumes that age has a normal (Gaussian) distribution in both groups. To check for this assumption, we can plot the distribution for each group using `seaborn` combined with the Shapiro–Wilk test from the `stats` module of the `scipy` library.

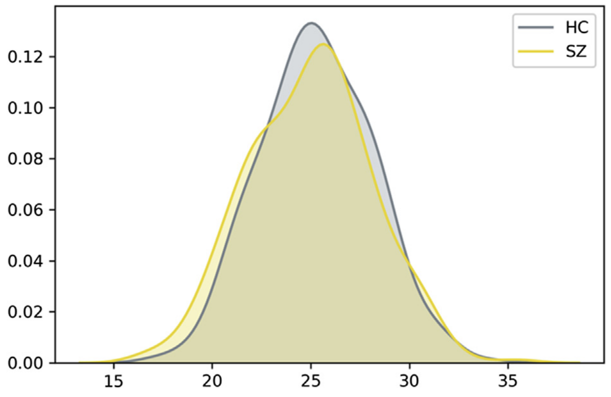
```
age_hc = dataset_df[dataset_df['Diagnosis'] == healthy_str]['Age']
age_sz = dataset_df[dataset_df['Diagnosis'] == patient_str]['Age']

# Plot normal curve
sns.kdeplot(age_hc,
            color='#839098',
            label='HC',
            shade=True)
sns.kdeplot(age_sz,
            color='#f7d842',
            label='SZ',
            shade=True)
plt.show()

# Shapiro test for normality
_, p_age_hc_normality = stats.shapiro(age_hc)
_, p_age_sz_normality = stats.shapiro(age_sz)

print('HC: Normality test: p-value = %.3f' % p_age_hc_normality)
print('SZ: Normality test: p-value = %.3f' % p_age_sz_normality)

# Descriptives
print('Age')
print('HC: Mean(SD) = %.2f(%.2f)' % (age_hc.mean(), age_hc.std()))
print('SZ: Mean(SD) = %.2f(%.2f)' % (age_sz.mean(), age_sz.std()))
```



```
HC: Normality test: p-value = 0.005
SZ: Normality test: p-value = 0.018

Age
HC: Mean(SD) = 25.31(2.84)
SZ: Mean(SD) = 24.98(3.12)
```

From the above, we can see that age is normally distributed for both groups. In addition, the distribution, mean, and standard deviation are fairly similar between the two groups. This means we can use student's t-test to check for any statistically significant difference in age between groups.

```
t_stats, p_age = stats.ttest_ind(age_sz, age_hc)
print('Age')
print('Student's t-test: t stats = %.3f, p-value = %.3f' % (t_stats, p_age))   16
```

```
Age
Student's t test: t-stats = -1.464, p-value = 0.144
```

It can be seen that there is no statistically significant difference in age between the two groups. Therefore, we will not consider age as a significant confounder in our analysis.

For more information on confounding variables in neuroimaging and machine learning, see Rao, Monteiro, and Mourao-Miranda (2017) and Chapter 14.

19.5.5.5 *Feature set and target*

Our next step is to retrieve the target and features from the dataset. For the target variable, we assign the column "diagnosis" in `dataset_df` to the variable `targets_df`. For the features, we select all rows from the fourth column onward (recall that dataframes are 0 indexed) and saved them in `features_df`.

```
# Target
targets_df = dataset_df['Diagnosis']

# Features
features_names = dataset_df.columns[3:]
features_df = dataset_df[features_names]

>>> targets_df
>>> features_df                                                                17
```

```
targets_df
ID
c001     hc
c002     hc
         ..
p370     sz
p371     sz
p372     sz
Name: Diagnosis, Length: 695, dtype: object

features_df
      Left Lateral Ventricle   ...  rh insula thickness
# ID                             ...
# c001             4226.907844  ...             2.645844
# c002             4954.912699  ...             2.673699
#                       ...   ...                  ...
# p370             3607.623866  ...             3.066604
# p371             8276.575805  ...             2.631420
# p372             5170.559424  ...             3.330186
# [695 rows x 169 columns]
```

The cleaned dataset comprises 695 subjects and 169 features. This number of subjects is well above the recommended sample size of 130 participants for a stable performance (Nieuwenhuis et al., 2012). However, in many studies of brain disorders, we might have a smaller sample size. In these cases, it is important to be cautious that less than optimal sample sizes can lead to unreliable results. The need for a minimum number of subjects highlights the importance of data sharing initiatives, such as Openneuro.org (https://openneuro.org/) and schizconnect (http://schizconnect.org)

Next, let us save the cleaned data to a CSV file in the directory created in snippet 3.

```
features_df.to_csv(experiment_dir / 'prepared_features.csv')
targets_df.to_csv(experiment_dir / 'prepared_targets.csv')
```

19.5.6 Feature engineering

In this step, we want to apply a series of transformations to our data that will help us build a good machine learning model. As described in Chapter 2, this series of transformations can involve different procedures depending on the nature of the data. Below we discuss each of these procedures following the same order as in Chapter 2.

19.5.6.1 Feature extraction

In our example, we want to use neuroanatomical data to classify SZ and HC. This requires the extraction of brain morphometric information from the raw MRI images. As mentioned earlier, in the present tutorial we assume that this feature extraction has already been done for us. The regional gray matter volumes and thickness that make up our `features_df` variable have been extracted with FreeSurfer.

19.5.6.2 Cross-validation

Before we move on to apply any transformations to our features, we first need to split the data into training and test sets. Recall that this is a critical step to ensure independence between the training and test steps of the machine learning analysis. In this tutorial, we use a stratified 10-fold CV. See Chapter 2 for an overview of some of the most commonly used types of CV.

We first transform `targets_df` into a 1D `numpy` array, where 0 denotes HC and 1 denotes SZ; we call this new variable `targets`. We do the same for the features by transforming `features_df` into a 2D numpy array that we name `features`. Transforming the data from dataframe to arrays will make it easier later on, as some of the functions require the data to be in this format.

```
targets_df = targets_df.map({healthy_str: 0, patient_str: 1})
targets = targets_df.values.astype('int')

features = features_df.values.astype('float32')                    19
```

Next, we define the parameters of the stratified 10-fold CV. We do this by creating an object from the class `StratifiedKFold` from the sklearn library. We will call this object `skf`.

```
n_folds = 10
skf = StratifiedKFold(n_splits=n_folds,
                      shuffle=True,
                      random_state=random_seed)                      20
```

Notice the argument `random_state` in `skf`. This argument allows us to control the element of randomness intrinsic to splitting the total data into train and test sets. Our dataset comprises of 695 participants in total. In the code above, we have instructed the model to split the dataset into 10 groups (while maintaining the SZ/HC ratio similar throughout the CV iterations). Now, there are multiple possible solutions to this task, i.e., there is not just one way of dividing 695 individuals into 10 groups. In these situations, Python performs this division at random. Not setting random_state to a fixed value means that every time we run our code, the participants assigned to each group will differ. Consequently, our results will very likely differ as well. This is something we would like to avoid, at least while we build upon our model to improve it, as we want to be able to reproduce the same results for comparison between different models. For interesting discussions on the relation between sample size, CV, and performance, see Nieuwenhuis et al. (2012) and Varoquaux (2017).

Next, we prepare a set of empty objects that will be populated with the predictions, the performance metrics, and the coefficients of the machine learning model from each iteration of the CV. In the code below, we create (1) an empty dataframe predictions_df that will store the model's predictions; (2) three empty arrays for each performance metric: balanced accuracy (bac), sensitivity (sens), and specificity (spec); and (3) an empty array for the SVM's coefficients, where the weights (coefficient or "importance") from each feature will be stored. Finally, we also create an additional folder models_dir where all the above objects will be saved later on.

```
predictions_df = pd.DataFrame(targets_df)
predictions_df['predictions'] = np.nan

bac_cv = np.zeros((n_folds, 1))
sens_cv = np.zeros((n_folds, 1))
spec_cv = np.zeros((n_folds, 1))
coef_cv = np.zeros((n_folds, len(features_names)))

models_dir = experiment_dir / 'models'
models_dir.mkdir(exist_ok=True)                                    21
```

Now that the CV is defined, we can loop over each one of the 10 CV iterations. At each iteration, we perform any transformations to the training set (e.g., feature selection, normalization) and fit the machine learning algorithm to the same data; we then use the test set to test the algorithm after performing the same data transformations applied in the training set. This can be implemented using a "for loop" to iterate over the 10 i_folds. At each i_fold, we will have four new variables:

- features_train and targets_train: training set and corresponding labels
- features_test and targets_test: test set and corresponding labels

Let us now check how many participants are there in the train and test sets in each iteration of the CV. We can do this by simply asking for the length of targets_train and targets_test.

```
for i_fold, (train_idx, test_idx) in enumerate(skf.split(features, targets)):
    features_train, features_test = features[train_idx], features[test_idx]
    targets_train, targets_test = targets[train_idx], targets[test_idx]

    print('CV iteration: %d' % (i_fold))
    print('Training set size: %d' % len(targets_train))
    print('Test set size: %d' % len(targets_test))                        22
```

```
CV iteration: 0
Training set size: 625
Test set size: 70
```

Notice how the code inside the "for loop" is placed further to the right. This is called indentation and means that the instructions in the indented block of code will be performed for each iteration of the CV. The next snippets (22 through 31) will keep the same indentation to denote that they are still part of this for loop. Once the CV is finished, the indentation will be removed, i.e., the text will be placed again from the left end of the text box. Note that if running this code, all loop snippets will need to be ran together. Also note that the numbering of the folds starts at 0, not 1; this is simply because in Python, for loops are indexed 0.

19.5.6.3 Feature selection

As discussed in Chapter 2, feature selection can help remove redundancy in our feature set. However, given that neuroanatomical abnormalities in SZ tend to be subtle and widespread, it is reasonable to assume that most of the 169 features included in our feature set will make some contribution toward distinguishing patients from controls. In addition, we note that in our example the number of features is not too large compared to the total sample size. For these reasons, we will not remove any features from the dataset. If we had opted to use feature selection, there would have been plenty of strategies available through sklearn. More information about these strategies can be found at https://scikit-learn.org/stable/modules/feature_selection.html.

19.5.6.4 Feature scaling/normalization

Before inputting the data into the classifier, we want to ensure the fact that the measurements of different brain regions have different ranges will not affect the reliability of our model. If a feature's variance is orders of magnitude greater than the variance of other features, that particular feature might dominate other features in the dataset.

There are several possible solutions to avoid this issue. In this example, we will transform the data such that the distribution of each feature resembles a standard normal distribution (e.g., mean = 0 and variance = 1). Each newly normalized value z_{x_i} is calculated by taking each data point x_i, subtracting the mean $\overline{X}$ of the corresponding feature and then dividing by the standard deviation (SD) of the same feature:

$$z_{x_i} = \frac{\left(\overline{X}_{\text{Feature}_A} - x_i\right)}{\text{SD}_{\text{Feature }A}}$$

We can apply this formula automatically to each feature independently using the object StandardScaler from sklearn.

```
scaler = StandardScaler()

scaler.fit(features_train)

features_train_norm = scaler.transform(features_train)
features_test_norm = scaler.transform(features_test)
```

First, we create a scaler object. Then, we fit the scaler parameters (mean and standard deviation) using the training set. In other words, we calculate and store $\overline{X}$and SD from each feature in the training set in the object scaler. Then, we transform both the training and test sets using the formula above with the stored parameters. Sklearn also provides other scaling strategies; for instance, if the distribution is not normal or has several outliers, the function RobustScaler() would be a better fit.

19.5.7 Model training

19.5.7.1 Machine learning algorithm and hyperparameter optimization

In this tutorial, we use SVM as implemented by sklearn. As discussed in Chapter 6, SVM allows the use of different kernels. Here, we will use the linear kernel, as this will make it easier to extract the coefficients of the SVM model (feature importance) later on. More information about different kernels can be found at https://scikit-learn.org/stable/modules/svm.html.

```
clf = LinearSVC(loss='hinge')
```

Importantly, SVM relies on a hyperparameter C that regulates how much we want to avoid misclassifying each training example. The ideal approach for choosing the value of C is by letting the model try several values and then selecting the one with the best performance. This should be done via an additional CV inside the already defined CV, thus creating a nested CV, where different values of C are fitted to the training set and tested in the validation set; the value of C with the best performance is then used to fit the model to the training set as defined by the outer CV (see Chapter 2 for more details).

Fortunately, sklearn has a set of useful tools to implement this. Here, we will use grid search, a popular choice in the brain disorders literature. More information about grid search and other methods for hyperparameter optimization can be found at https://scikit-learn.org/stable/modules/grid_search.html.

To implement grid search, we first need to provide a range of possible values for C; this is our search space. Next, we specify the parameters for the GridSearchCV. We will use stratified k-fold with 10 iterations again, as with the outer CV previously defined. The reader can refer to Fig. 19.1 for a visual representation of the model design.

```
# Hyperparameter search space
param_grid = {'C': [2 ** -6, 2 ** -5, 2 ** -4, 2 ** -3, 2 ** -2, 2 ** -1,
                    2 ** 0, 2 ** 1]}

# Gridsearch
internal_cv = StratifiedKFold(n_splits=10)
grid_cv = GridSearchCV(estimator=clf,
                       param_grid=param_grid,
                       cv=internal_cv,
                       scoring='balanced_accuracy',
                       verbose=1)
```

We are now ready to fit our SVM model to the training data. We do this by applying the fit command to the features and labels from the training set.

```
grid_result = grid_cv.fit(features_train_normalized, targets_train)
```

```
Fitting 10 folds for each of 8 candidates, totalling 80 fits
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent
workers.
[Parallel(n_jobs=1)]: Done  80 out of  80 | elapsed:    8.3s finished
```

The code below shows how GridSearchCV works: we can see the performance for the different values of C in the validation set within the inner CV.

```
print('Best: %f using %s' % (grid_result.best_score_,
                             grid_result.best_params_))
means = grid_result.cv_results_['mean_test_score']
stds = grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']

for mean, stdev, param in zip(means, stds, params):
    print('%f (%f) with: %r' % (mean, stdev, param))                   27
```

```
Best: 0.675791 using {'C': 0.125}
0.673129 (0.076206) with: {'C': 0.015625}
0.674068 (0.091944) with: {'C': 0.03125}
0.668388 (0.089292) with: {'C': 0.0625}
0.675791 (0.077299) with: {'C': 0.125}
0.669378 (0.083826) with: {'C': 0.25}
0.662557 (0.057900) with: {'C': 0.5}
0.653957 (0.060696) with: {'C': 1}
0.657501 (0.067748) with: {'C': 2}
```

The value of C that yields the best performance in the validation set is shown at the top. The SVM model with this value of C is used again to train an SVM model in the whole training set (as defined by the outer CV) and is stored in an objected called best_estimator_, which we will use to define a second classifier best_clf. This is the model we will use later on for making predictions in the test set. We save the best model as well as the scaler object (mean and SD from the training set used to normalize our data) in the already created models_dir folder.

```
best_clf = grid_cv.best_estimator_

joblib.dump(best_clf, models_dir / ('classifier_%d.joblib' % i_fold))
joblib.dump(scaler, models_dir / ('scaler_%d.joblib' % i_fold))        28
```

19.5.7.2 Model's coefficients

In addition to model performance, we are also interested in knowing which features are driving the model's predictions. In the case of SVM with a linear kernel, this information is automatically stored in the property coef_ inside best_clf. We will store the coefficients of each feature in the empty list of coefficients coef_cv that we have previously created (snippet 21).

```
coef_cv[i_fold, :] = np.abs(best_clf.coef_)                          29
```

19.5.8 Model evaluation

Finally, we use the final trained model best_clf to make predictions in the test set. Predictions are stored in target_test_predicted. This variable is then used to populate the previously created empty dataframe predictions_df (created in snippet 21).

```python
target_test_predicted = best_clf.predict(features_test_norm)

for row, value in zip(test_idx, target_test_predicted):
    predictions_df.iloc[row,
                        predictions_df.columns.get_loc('predictions')] \
                         = value                                      30
```

Once we have the predicted labels, we can estimate the performance in the test set. First, we compute the confusion matrix. From here, we estimate bac, sens, and spec. There are plenty more metrics to choose from in sklearn. A comprehensive list can be found at https://scikit-learn.org/stable/modules/model_evaluation.html.

```python
print('Confusion matrix')
cm = confusion_matrix(targets_test, target_test_predicted)
print(cm)

tn, fp, fn, tp = cm.ravel()

bac_test = balanced_accuracy_score(targets_test, target_test_predicted)
sens_test = tp / (tp + fn)
spec_test = tn / (tn + fp)

print('Balanced accuracy: %.3f ' % bac_test)
print('Sensitivity: %.3f ' % sens_test)
print('Specificity: %.3f ' % spec_test)

bac_cv[i_fold, :] = bac_test
sens_cv[i_fold, :] = sens_test
spec_cv[i_fold, :] = spec_test                                        31
```

```
Confusion matrix
[[31  6]
 [10 23]]

Balanced accuracy: 0.767
Sensitivity: 0.697
Specificity: 0.837
```

The steps from snippet 22 through 31 are repeated for each one of the 10 iterations of the CV. This means that at the end of this process we will have 10 estimates of the model's performance, one for each iteration. The overall performance is then calculated by averaging the performance metrics across iterations.

```
print('CV results')
print('Bac: Mean(SD) = %.3f(%.3f)' % (bac_cv.mean(), bac_cv.std()))
print('Sens: Mean(SD) = %.3f(%.3f)' % (sens_cv.mean(), sens_cv.std()))
print('Spec: Mean(SD) = %.3f(%.3f)' % (spec_cv.mean(), spec_cv.std()))        32
```

```
CV results
Bac: Mean(SD) = 0.744(0.046)
Sens: Mean(SD) = 0.718(0.078)
Spec: Mean(SD) = 0.770(0.063)
```

We can see from the output above that our model was able to classify patients with SZ and HC with a bac of 74%. We must remember to save the main results, including the coefficients, predictions, and performance metrics in our dedicated directory.

```
# Saving coefficients
mean_coef = np.mean(coef_cv, axis=0).reshape(1, -1)

coef_df = pd.DataFrame(data=mean_coef, columns=features_names.values)
coef_df.to_csv(experiment_dir / 'feature_importance.csv', index=False)

# Saving predictions
predictions_df.to_csv(experiment_dir / 'predictions.csv', index=True)

# Saving metrics
metrics = np.concatenate((bac_cv, sens_cv, spec_cv), axis=1)
metrics_df = pd.DataFrame(data=metrics, columns=['bac', 'sens', 'spec'])
metrics_df.index.name = 'CV iteration'
metrics_df.to_csv(experiment_dir / 'metrics.csv', index=True)        33
```

This nested CV will take a couple of minutes to run. However, other analyses could take much longer, depending on the size and nature of the data as well as the machine learning pipeline.

19.5.9 Post hoc analysis

Once we have our final model, we can run several additional analyses. Here, we will run the following:

- Test balanced accuracy, sensitivity, and specificity for statistical significance via permutation testing
- Identify the features that provided the greatest contribution to the task

We start by creating a separate folder inside the directory for our experiment to store the results from the permutation testing.

```
permutation_dir = experiment_dir / 'permutation'
permutation_dir.mkdir(exist_ok=True)        34
```

Next, we store our model's final bac, sens, and spec. This will come in handy later on.

```
bac_from_model = bac_cv.mean()
sens_from_model = sens_cv.mean()
spec_from_model = spec_cv.mean()                                          35
```

As explained in Chapter 2, permutation testing involves running the same model several times, each time with the labels shuffled at random. Therefore, we need to first define how many times we want to run our model (i.e., number of permutations). Typically, studies use 1000 to 10,000 permutations. This may take some time to run (for the purpose of completing this tutorial, the reader could try running a much smaller number of permutations; however, it should be noted that this would not be an acceptable number of permutations for a real study). To store the results from each permutation, we first create four empty objects which we will populate after each permutation.

```
n_permutations = 1000

bac_perm = np.zeros((n_permutations, 1))
sens_perm = np.zeros((n_permutations, 1))
spec_perm = np.zeros((n_permutations, 1))
coef_perm = np.zeros((n_permutations, len(features_names)))              36
```

Next, we set up a "for loop," which we use to iterate over each permutation. Because of the presence of the indentation after the "for loop," all commands inside this "for loop" will be repeated for each permutation (snippets 37–48). At each iteration, the diagnoses of the subjects will be randomly shuffled, using the function random.-permutation from numpy. This approach will remove any association between the features and targets. As we would like this shuffling to be different at every iteration, we set the random seed that numpy uses to a new fixed value.

```
for i_perm in range(n_permutations):
    print('Permutation: %d' % (i_perm))

    np.random.seed(i_perm)
    targets_permuted = np.random.permutation(targets)                     37
```

```
Permutation: 0
```

We then apply the exact same pipeline to the same dataset with the shuffled labels. Note how the code below follows the same instructions as in snippets 21 through 31.

```
    n_folds = 10
    skf = StratifiedKFold(n_splits=n_folds,
                          shuffle=True,
                          random_state=random_seed)

    bac_cv = np.zeros((n_folds, 1))
    sens_cv = np.zeros((n_folds, 1))
    spec_cv = np.zeros((n_folds, 1))
    coef_cv = np.zeros((n_folds, len(features_names)))

    for i_fold, (train_idx, test_idx) in enumerate(skf.split(features,
                                                              targets_permuted)):
        features_train = features[train_idx]
        features_test = features[test_idx]
        targets_train = targets_permuted[train_idx]
        targets_test = targets_permuted[test_idx]

        scaler = StandardScaler()
        features_train_norm = scaler.fit_transform(features_train)
        features_test_norm = scaler.transform(features_test)

        clf = LinearSVC(loss='hinge')

        param_grid = {'C': [2 ** -6, 2 ** -5, 2 ** -4, 2 ** -3, 2 ** -2,
                            2 ** -1, 2 ** 0, 2 ** 1]}

        internal_cv = StratifiedKFold(n_splits=10)
        grid_cv = GridSearchCV(estimator=clf,
                               param_grid=param_grid,
                               cv=internal_cv,
                               scoring='balanced_accuracy',
                               verbose=0)

        grid_result = grid_cv.fit(features_train_norm, targets_train)

        best_clf = grid_cv.best_estimator_

        coef_cv[i_fold, :] = np.abs(best_clf.coef_)

        target_test_predicted = best_clf.predict(features_test_norm)

        cm = confusion_matrix(targets_test, target_test_predicted)

        tn, fp, fn, tp = cm.ravel()

        bac_test = balanced_accuracy_score(targets_test, target_test_predicted)
        sens_test = tp / (tp + fn)
        spec_test = tn / (tn + fp)

        bac_cv[i_fold, :] = bac_test
        sens_cv[i_fold, :] = sens_test
        spec_cv[i_fold, :] = spec_test                                          38
```

With the code below, we estimate the mean bac, sens, spec, as well as the model coefficients across the 10 CV iterations, and save them in the directory permutation_dir.

```
    np.save(permutation_dir / ('perm_test_bac_%03d.npy' % i_perm),
            bac_cv.mean())
    np.save(permutation_dir / ('perm_test_sens_%03d.npy' % i_perm),
            sens_cv.mean())
    np.save(permutation_dir / ('perm_test_spec_%03d.npy' % i_perm),
            spec_cv.mean())
    np.save(permutation_dir / ('perm_coef_%03d.npy' % i_perm),
            coef_cv.mean(axis=0))

    bac_perm[i_perm, :] = bac_cv.mean()
    sens_perm[i_perm, :] = sens_cv.mean()
    spec_perm[i_perm, :] = spec_cv.mean()
    coef_perm[i_perm, :] = coef_cv.mean(axis=0)                                 39
```

Finally, we calculate the *p*-value for bac, sens, and spec by dividing the number of times that the performance of the permuted dataset is better than the performance of our model by the number of permutations. The resulting *p*-value will indicate if our model behaves similarly to a random classifier or if it is significantly better.

```
# Get p-values from metrics
bac_p_value = (np.sum(bac_perm >= bac_from_model) + 1) / (n_permutations + 1)
sens_p_value = (np.sum(sens_perm >= sens_from_model) + 1) / (n_permutations + 1)
spec_p_value = (np.sum(spec_perm >= spec_from_model) + 1) / (n_permutations + 1)

print('BAC: p-value = %.3f' % bac_p_value)
print('SENS: p-value = %.3f' % sens_p_value)
print('SPEC: p-value = %.3f' % spec_p_value)                          40
```

```
BAC: p-value = 0.001
SENS: p-value = 0.001
SPEC: p-value = 0.001
```

All *p*-values indicate that our model was able to classify HC and SZ with a performance above chance level (i.e., random classifier).

Let us also estimate the statistical significance of the model's coefficients. This will allow us to check which features made a statistically significant contribution to the task.

```
# Get p-values from coefficients
coef_p_values = np.zeros((1, len(features_names)))
for i_feature in range(len(features_names)):
    coef_value_from_perm = coef_perm[:, i_feature]
    coef_value_from_model = mean_coef[0, i_feature]

    n_perm_better_model = np.sum(coef_value_from_perm >= coef_value_from_model)

    coef_p_values[0, i_feature] = (n_perm_better_model + 1) / \
                                  (n_permutations + 1)                 41
```

Next, we create a dataframe to store the coefficient values and respective *p*-values.

```
coef_df = pd.DataFrame(index=['coefficients', 'p value'],
                       data=np.concatenate((mean_coef, coef_p_values)),
                       columns=features_names)

>>> coef_df.sort_values('coefficients', axis=1, ascending=False).T       42
```

```
                                         coefficients   p value
lh middletemporal thickness                  0.874659  0.000999
Right Amygdala                               0.669204  0.000999
3rd Ventricle                                0.666117  0.000999
lh parahippocampal thickness                 0.633003  0.000999
lh middletemporal volume                     0.446395  0.005994
rh parahippocampal thickness                 0.399505  0.002997
Left Hippocampus                             0.385345  0.038961
Left Amygdala                                0.341837  0.033966
lh medialorbitofrontal thickness             0.302409  0.043956
lh rostralanteriorcingulate thickness        0.296009  0.066933
rh superiorfrontal volume                    0.293653  0.144855
Left Lateral Ventricle                       0.290743  0.130869
...                                               ...       ...
rh paracentral volume                        0.028124  0.998002
rh lateralorbitofrontal volume               0.027997  1.000000

[169 rows x 2 columns]
```

In our final step, we save the overall performance metrics and corresponding *p*-values, as well as the coefficients in two separate CSV files.

```
# Saving
perm_metrics_df = pd.DataFrame(data={'metric': ['bac', 'sens', 'spec'],
                                     'value': [bac_from_model,
                                               sens_from_model,
                                               spec_from_model],
                                     'p_value': [bac_p_value,
                                                 sens_p_value,
                                                 spec_p_value]})

perm_metrics_df.to_csv(experiment_dir / 'metrics_permutation_pvalue.csv',
                       index=False)

coef_df.to_csv(experiment_dir / 'coef_permutation_pvalue.csv', index=True)     43
```

19.5.10 Main results from this tutorial

The results of our example show that we were able to classify SZ and HC with 75% balanced accuracy, 72% sensitivity, and 77% specificity. The main features driving the model's predictions included the third ventricle, the thickness of the bilateral parahippocampal, left middle temporal, medial orbitofrontal, and rostral anterior cingulate gyri, as well as the volume of the bilateral amygdala, left hippocampus, and middle temporal gyrus.

In the directory, the reader will find the following files and folders:

- prepared_features.csv—cleaned features
- prepared_targets.csv—file containing cleaned targets
- models—folder containing the trained models and scalers for each iteration of the CV
- predictions.csv—predictions made by the classifier
- metrics.csv—bac, sens, and spec for each iteration of the CV
- feature_importance.csv—final coefficients for each feature
- permutation—folder with the bac, sens, spec, and coefficients for each permutation
- metrics_permutation_pvalue.csv—final bac, sens, and spec with corresponding *p*-values
- coef_permutation_pvalue.csv—final coefficients and corresponding *p*-values for each feature

19.6 Conclusion

In this tutorial, we were able to distinguish between SZ and HC at the individual level with 75% accuracy. However, more sophisticated machine learning approaches should lead to higher performances (for a metaanalysis of machine learning studies in psychosis, see Kambeitz et al. (2015)). There are several strategies that could be used to improve our model. Perhaps one of the most obvious and straightforward strategies would be to try different classifiers. Indeed, several of the most commonly used classifiers can be implemented without having to make too many changes to the code above. A comprehensive list of all the classifiers in sklearn can be found at https://scikit-learn.org/stable/supervised_learning.html. Another possible strategy would be to add a feature selection step to remove less relevant features. However, the element of the pipeline with the greatest impact on performance is probably the type of features used as input. Therefore, a further strategy would be to try other types of features, such as voxel-level data (as opposed to region level) that might convey more useful information for the classification task. We could also try using a combination of different types of features within the same model, based on the notion that different features might capture different aspects of disease mechanism. This would, however, raise other issues such as high dimensionality that would need to be considered and addressed. Adding a dimensionality reduction step to our model, such as principal component analysis covered in Chapter 12, would help with dealing with this issue. Overall, it is considered good practice to experiment with different approaches. However, it is important to avoid building bespoke pipelines that fit a specific dataset well but are unlikely to perform well if tested in another dataset. This is especially important in brain disorders, where datasets are typically small resulting in high risk of overfitting.

In this tutorial we used tabular data. This type of data is straightforward to use, visualize, and manipulate. However, many studies of brain disorders have used more complex data types. For example, neuroimaging data in other formats such as 3D images would have to be handled differently using dedicated libraries such as nibabel (https://nipy.org/nibabel/), nilearn (https://nilearn.github.io), or DLTK (https://dltk.github.io). The analysis of speech or text data would also require specific tools such as the NLTK library (https://www.nltk.org). With regard to the machine learning analysis, sklearn is by far the most commonly used Python library. However, it does not cover all machine learning approaches. For example, deep learning is often implemented using Keras (https://keras.io), a high-level and intuitive tool.

In light of the increasing interest in the application of machine learning methods to investigate brain disorders, it is important for researchers and clinicians to develop a greater awareness of its potential as well as its limitations. Learning how to apply machine learning, or at least understanding its practical implementation, is an excellent way of gaining insight into its strengths and shortcomings. For example, in this tutorial, we have learned how flexible a machine learning analysis can be, with several possible options to be chosen from at each stage. On the other hand, we have also discussed several issues that need to be addressed to minimize the risk of bias, such as missing data, data imbalance, confounding variables, and appropriate use of nested CV for hyperparameter tuning. We hope that this tutorial will encourage the machine learning novice take the first steps toward designing, building, and testing their own machine learning models.

References

Fischl, B. (2012). FreeSurfer. *NeuroImage, 62*(2), 774–781. https://doi.org/10.1016/j.neuroimage.2012.01.021.

Kambeitz, J., Kambeitz-Ilankovic, L., Leucht, S., Wood, S., Davatzikos, C., Malchow, B., et al. (2015). Detecting neuroimaging biomarkers for schizophrenia: A meta-analysis of multivariate pattern recognition studies. *Neuropsychopharmacology, 40*(7), 1742–1751. https://doi.org/10.1038/npp.2015.22.

Nieuwenhuis, M., van Haren, N. E. M., Hulshoff Pol, H. E., Cahn, W., Kahn, R. S., & Schnack, H. G. (2012). Classification of schizophrenia patients and healthy controls from structural MRI scans in two large independent samples. *NeuroImage, 61*(3), 606–612. https://doi.org/10.1016/J.NEUROIMAGE.2012.03.079.

Rao, A., Monteiro, J. M., & Mourao-Miranda, J. (2017). Predictive modelling using neuroimaging data in the presence of confounds. *NeuroImage, 150*, 23–49. https://doi.org/10.1016/J.NEUROIMAGE.2017.01.066.

Schrouff, J., Rosa, M. J., Rondina, J. M., Marquand, A. F., Chu, C., Ashburner, J., et al. (2013). PRoNTo: Pattern recognition for neuroimaging toolbox. *Neuroinformatics, 11*(3), 319–337. https://doi.org/10.1007/s12021-013-9178-1.

Tandon, N., & Tandon, R. (2018). Will machine learning enable us to finally cut the gordian knot of schizophrenia. *Schizophrenia Bulletin, 44*(5), 939–941. https://doi.org/10.1093/schbul/sby101.

Varoquaux, G. (2017). Cross-validation failure: Small sample sizes lead to large error bars. *NeuroImage, 180*, 68–77. https://doi.org/10.1016/j.neuroimage.2017.06.061.