Chapter 8
Applied Statistical Learning in Python

Calvin J. Chiew

Abstract This chapter is based on a workshop I have conducted at several datathons introducing clinicians to popular statistical methods used in machine learning. It is primarily aimed at beginners who want a gentle, succinct guide to jumpstart their journey into practical machine learning and its applications in medicine. Thus, it is by no means a comprehensive guide on machine learning or Python. Rather, my hope is to present basic concepts in a simple, creative way, and demonstrate how they can be applied together.

Keywords Python · Crash course · Machine learning · Classification · Random forest · Support vector machine · Clinical prediction · Model fit · Cross-validation

Learning Objectives

- Readers will be able to run a simple program in Python
- Readers will be able to use a Jupyter Notebook
- Readers will understand basic concepts of supervised learning such as model fitting and cross-validation
- Readers will be able to differentiate between supervised learning methods for classification such as random forest and support vector machines

8.1 Introduction

A crash course on the basics of the Python language and Jupyter notebook environment will be presented in Sect. 8.2 to help those without prior programming experience get started quickly. You are welcome to skip this section if you are already familiar with Python. In Sects. 8.3, 8.4, 8.5, 8.6, I will introduce the random forest and support vector machine for classification, as well as general concepts of model fit and cross-validation. Finally, in a hands-on exercise in Sect. 8.7, you will be asked

C. J. Chiew (✉)
National University Health System, 1E Kent Ridge Rd, Singapore 119228, Singapore
e-mail: calvinjchiew@mail.harvard.edu

© The Author(s) 2020
L. A. Celi et al. (eds.), *Leveraging Data Science for Global Health*,
https://doi.org/10.1007/978-3-030-47994-7_8

to implement and evaluate these models on a clinical prediction problem. Suggested solutions are provided for your reference. Each section ends with a summary that reinforces key concepts from that section. The corresponding files for this chapter can be found at https://github.com/criticaldata/globalhealthdatabook.git. If after reading this chapter you are motivated to learn more, there are plenty of print and online resources available (see Suggested Readings and References lists at the end).

8.1.1 Requirements & Setup Instructions

There are accompanying demos and exercises to this chapter which you are encouraged to access for the best educational experience. To do that, you will need a computer installed with **Python** and **Jupyter notebook**, the environment in which we will write and run Python code. By far the most convenient and reliable installation method is through the **Anaconda** distribution. This also comes with all the commonly used libraries or packages (i.e. the ones we need) bundled in, saving you the hassle of downloading and installing them one by one.

First, download the installer for Anaconda (Python 3 version) on your respective OS (Windows, Mac or Linux) from https://www.anaconda.com/download/. Then, run the installer and use all default options when prompted. Finally, after installation is complete, make sure you can open **Anaconda Navigator** and launch Jupyter notebook. (If you need help troubleshooting or have any programming-related questions, Stack Overflow [https://stackoverflow.com/] is a great place to look for answers.)

8.2 Python Crash Course

8.2.1 Terminology

Python is a programming language that has become popular for data science and machine learning (Guttag 2013). A **Jupyter notebook**, which is denoted by the file format `.ipynb`, is a document in which you can write and run Python code. It consists of cells, which can contain either Markdown (text) or code. Each cell can be executed independently, and the results of any code executed are "saved" until the file is closed. Raw data files are often **comma-separated values (CSV)** files which store tabular data in plain text. Each record consists of values (can be numeric or text) separated by commas. To see an example, open the accompanying dataset `births.csv` in Notepad and examine its contents. You can also open it in Excel for a tabular view.

There are many useful **libraries** or **modules** in Python which can be **imported** and **called** to make our lives easier and more convenient. SciPy is an ecosystem of Python libraries for math and science. The core libraries include NumPy, Pandas and Matplotlib. **NumPy** (typically imported as `np`) allows you to work efficiently with

data in arrays. **Pandas** (typically imported as `pd`) can load csv data into **dataframes** which optimize storage and manipulation of data. Dataframes have useful methods such as `head`, `shape`, `merge` etc. The **pyplot** module (typically imported as `plt`) in **matplotlib** contains useful functions for generating simple plots e.g. `plot`, `scatter`, `hist` etc. You will encounter these libraries and their functions in the demo and hands-on exercise later.

8.2.2 Basic Built-in Data Types

The basic built-in data types you should be familiar with in Python are **integer**, **float**, **Boolean**, **string** and **list**. Examples of each type are as follows:

Integer	`7`
Float	`7.0`
Boolean	`True, False`
String	`'Hi', "7.0"`
List	`[], ['Hello', 70, 2.1, True]`

Strings can be enclosed by either single or double quotation marks. Lists are collections of items, which can be of different types. They are indicated by square brackets, with items separated by commas. Unlike older programming languages like C, you do not need to declare the types of your variables in Python. The type is inferred from the value assigned to the variable.

8.2.3 Python Demo

You do not need to be a Python expert in order to use it for machine learning. The best way to learn Python is simply to practice using it on several datasets. In line with this philosophy, let us review the basics of Python by seeing it in action.

Open Anaconda Navigator and launch Jupyter Notebook. In the browser that pops up, navigate to the folder where you have saved the accompanying files to this chapter. Click on `demo.ipynb`. In this notebook, there are a series of cells containing small snippets of Python code. Clicking the "play" button (or hitting Shift + Enter) will execute the currently selected (highlighted) cell. Run through each cell in this demo one by one—see if you understand what the code means and whether the output matches what you expect. Can you identify the data type of each variable.

In cell 1, the `*` operator represents multiplication and in cell 2, the `==` operator represents equality. In cell 3, we create a list of 3 items and assign it to `lst` with the = operator. Note that when cell 3 is executed, there is no output, but the value of `lst` is saved in the kernel's memory. That is why when we index into the first item of `lst` in cell 4, the kernel already knows about `lst` and does not throw an error.

Indexing into a list or string is done using square brackets. Unlike some other programming languages, Python is **zero-indexed**, i.e. counting starts from zero, not one! Therefore, in cells 4 and 5, we use `[0]` and `[1:]` to indicate that we want the first item, and the second item onwards, respectively.

In cell 6, we ask for the length of `lst` with the built-in **function** `len()`. In cell 7, we create a **loop** with the `for`…`in`… construct, printing a line for each iteration of the loop with `print()`. Note that the number '5' is not printed even though we stated `range(5)`, demonstrating again that Python starts counting from zero, not one.

In cell 8, we define our own function `add()` with the `def` and `return` keywords. There is again no output here but the definition of `add()` is saved once we execute this cell. We then call our function `add()` in cell 9, giving it two inputs (arguments) 1 and 2, and obtaining an output of 3 as expected.

In cell 10, we define a more complicated function `rate()` which when given a letter grade (as a string), outputs a customized string. We create branches within this function with the `if`…`elif`…`else` construct. One important thing to note here is the use of **indentation** to indicate nesting of code. Proper indentation is non-negotiable in Python. Code blocks are not indicated by delimiters such as `{}`, only by indentation. If indentation is incorrect (for example if this block of code were written all flushed to the left), the kernel would throw an error. In cells 11 and 12, we call our function `rate()` and check that we obtain desired outputs as expected.

Taking a step back, notice how Python syntax is close to plain English. Code readability is important for us to maintain code (imagine coming back 6 months later and realizing you cannot make sense of your own code!) as well as for others to understand our work.

It is not possible (nor necessary) to cover everything about Python in this crash course. Below I have compiled a list of common operators and keywords into a "cheat sheet" for beginners.

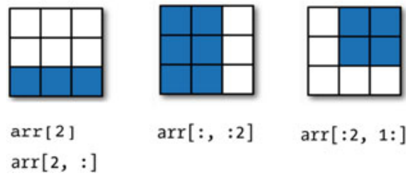
Arithmetic	`+, -, *, /, %, **, //`
Comparison	`== , != , > , < , >=, <=`
Boolean logic	`and, or, not`
Indexing lists/strings	`[n], [n:m], [n:], [:n]`
Selection	`if, elif, else`
Iteration/loop	`for, in, range`
Create function	`def, return`
Call function	`function(arg1, arg2, …)`
Call object's method or library's function	`object.method(arg1, arg2, …)` `library.function(arg1, arg2, …)`
Get length of list/string	`len(…)`
Import library	`import … as …`
Print	`print()`

8.2.4 Python Exercise

You are now ready to practice your Python skills. Open the notebook `python.ipynb` and give the exercise a shot. In this exercise, we will practice some simple data exploration, which is an important aspect of the data science process before model-building. Try to give your variables descriptive names (e.g. "age", "gender" are preferable to "a", "b"). If you are stuck, refer to `python_solutions.ipynb` for suggested solutions. Read on for more explanations.

In the very first cell, we import the libraries we need (e.g. pandas) and give them short names (e.g. `pd`) so that we can refer to them easily later. In Q1, we read in the dataset into a pandas dataframe `births` by calling the `read_csv()` function from `pd`. Note that the data file `births.csv` should be in the same folder as the notebook, otherwise you have to specify its location path. `births` is a dataframe **object** and we can call its **methods** `head` and `shape` (using the `object.method` notation) to print its first 5 rows and its dimensions. Note that the shape of dataframes is always given as (number of rows, number of columns). In this case, we have 400 rows and 3 columns.

It is worth spending some time at this juncture to clarify how we index into 2D arrays such as dataframes, since it is something we commonly need to do. The element at the n-th row and the m-th column is indexed as [n, m]. Just like lists, you can get multiple array values at a time. Look at the figures below and convince yourself that we can index into the blue elements of each 2D array by the following commands. Remember, Python is zero-indexed.



In Q2, we call the `mean` method to quickly obtain the mean value for each column in `births`. In Q3, we create 3 copies of the `births` dataframe—`group1`, `group2` and `group3`. For each group, we select (filter) the rows we want from `births` based on maternal age. Note the use of operators to specify the logic. We then apply `shape` and `mean` methods again to obtain the number of births and mean birth weight for each group and `print()` them out.

In Q4, we call `scatter()` from the `pyplot` module (which we have earlier imported as `plt`) to draw a scatterplot of data from `births`, specifying `birth_weight` as the x-axis, and `femur_length` as the y-axis. Note the use of `figure()` to start an empty figure, `xlabel()` and `ylabel()` to specify the axis labels, and `show()` to print the figure.

The code in Q5 is similar, except that we call `scatter()` 3 times, using data from `group1`, `group2` and `group3` instead of `births`, and specifying the different

colors we want for each group. We use `legend()` to also include a key explaining the colors and their labels in the figure. If we wanted to add a figure title, we could have done that with `title()`.

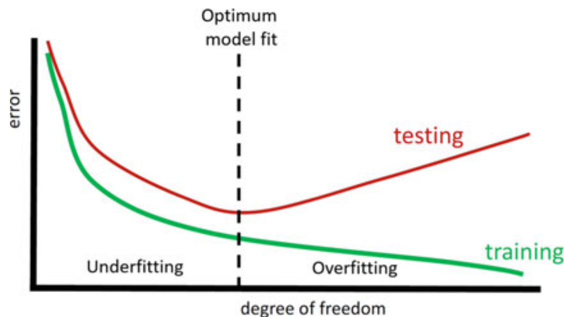
8.3 Model Fit

In machine learning, we are often interested in prediction. Given a set of **predictors** or **features** (X1, X2, X3…), we want to predict the **response** or **outcome** (Y). Mathematically speaking, we want to estimate f in $Y = f(X_1, X_2, X_3 \ldots) + \varepsilon$, where f is a function of our predictors and ε is some error. (James et al. 2013) If Y is a continuous variable, we call this task **regression**. If Y is categorical, we call it **classification**.

We choose an **error** or **loss function** that is appropriate for the prediction task. In regression, we commonly use mean squared error (MSE), which is the sum of residuals squared divided by sample size. In classification, the error can simply be the number of misclassifications.

Data is typically split into two distinct subsets—**training** and **testing**. The training set is used to create the model, i.e. estimate f. The testing set is used to evaluate the model, i.e. to see how good f is at predicting Y given a set of X. Therefore, the testing set acts as an independent, fair judge of our model's performance. The size of the train-test split is dependent on the size and specifics of the dataset, although it is common to use 60–80% of the data for training and the remainder for testing.

Both the training and testing error will decrease up to a point of optimum model fit (dotted line). Beyond that, **overfitting** occurs as the model becomes more specific to the training data, and less generalizable (flexible) to the testing data. Even though the training error continues to decline, the testing error starts to go up. Another way to think of overfitting is that an overfitted model picks up the "noise" of the function rather than focusing on the "signals". It is thus important for us to separate data into training and testing sets from the start, so that we can detect overfitting and avoid it.



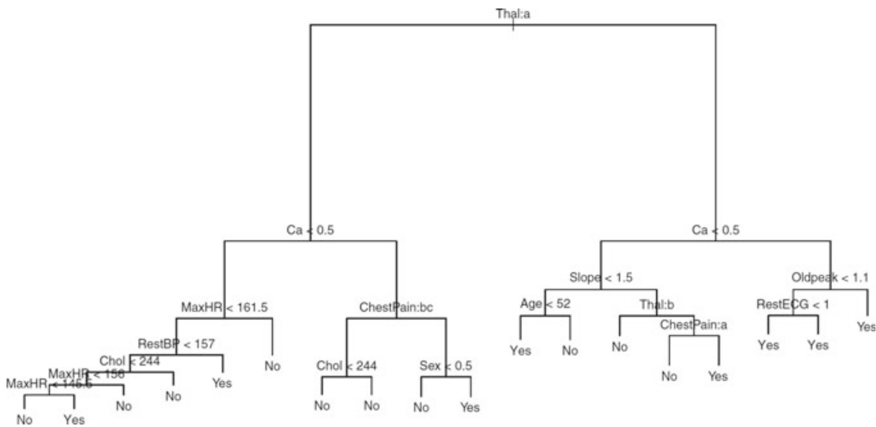
Summary

- In statistical modelling, we want to estimate f in Y = f(X) + e, where Y is the response (outcome), X is a set of features (predictors), and e is the error.
- To prevent overfitting, we split the data into training and testing sets. We develop the model on the training set, then evaluate its performance on the testing set.
- We choose an error (loss) function appropriate for the prediction task, e.g. mean squared error for regression (continuous Y), sum of misclassifications for classification (categorical Y).

8.4 Random Forest

8.4.1 Decision Tree

A **decision tree** is simply a series of splitting or **branching** rules. Take a look at this decision tree which predicts whether a patient walking into the Emergency Room with chest pain has Acute Myocardial Infarction (AMI), commonly known as "heart attack".



We start from the top and move our way down the tree. At each branching point or **node**, we either go left or right depending on whether the patient meets the criteria specified. For example, at the first branch, we go left if the patient is < 50 years old, or right if the patient is ≥ 50 years old. Each branch features one predictor, for example age, blood pressure, heart rate or cholesterol. The same predictor can appear multiple times in the tree and with different thresholds. Eventually, we reach the bottom, in one of the **terminal nodes** or **leaves**, where we obtain a prediction—either yes or no. A decision tree is thus very easy to visualize and understand.

To build the tree, an algorithm called **recursive binary splitting** is used. While the underlying mathematical theory of this algorithm is beyond the scope of this chapter,

we can think of it at a conceptual level. In the case of a classification tree here, the algorithm aims to increase **node purity**, indicated by a lower **Gini index**, with each successive split. This means we want observations that fall into each node to be predominantly from the same class. Intuitively, we understand why—if the majority (or all) of the observations in one node are "yes", then we are quite confident any future observation that follows the same branching pattern into that node will also be a "yes".

Since the branching can continue infinitely, we must specify a stopping criterion, for example until each terminal node has some minimum number of observations (minimum node size), or a certain maximum tree depth is reached. Note that it is possible to split a node into two leaves with the same predicted class, if doing so achieves higher node purity (creates more certainty).

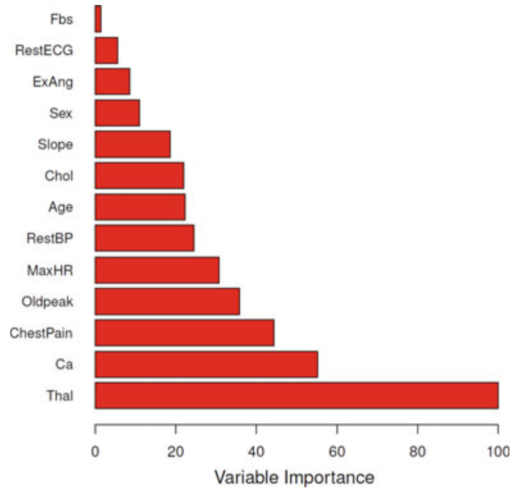
8.4.2 Random Forest

A **random forest**, as the name suggests, contains multiple decision trees. It is an example of the **ensemble** method, a commonly used machine learning technique of combining many models to achieve one optimal model. A disadvantage of decision trees is that they have high variance, that is if we change the training data by a little bit, we get a very different looking tree, so the result is not stable. To deal with this, we want to produce multiple trees and then take the **majority vote** of their predictions to reduce uncertainty.

We get that many trees form a forest, but why random? If we train all the trees the same way, they are all going to learn the same thing—all of them will choose the most important predictor as the top branch, and the next important predictor as the second branch, and so forth. We will end up with trees that are just clones of each other, defeating our original intent. What we really want are trees that can complement each other's weaknesses and errors. To harness the "power of crowds", we need diversity, not herd mentality.

Thus, at each branching point, only a **random subset** of all the predictors are considered as potential split candidates. Doing so enables us to get trees that are less similar to each other, obtaining a random forest.

In a random forest, feature importance can be visualized by calculating the total decrease in Gini index due to splits over each predictor, averaged over all trees. The following graph shows the relative importance of each feature in a random forest model predicting AMI in patients with chest pain from earlier.



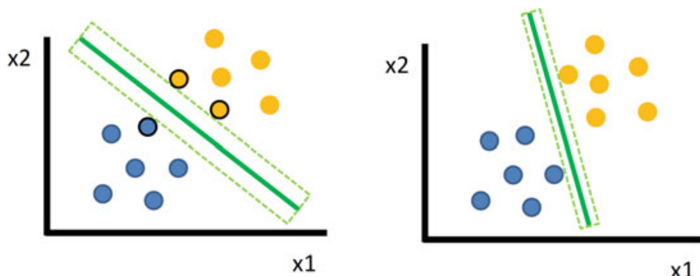
Summary

- Random forest is an ensemble method combining multiple decision trees to improve prediction accuracy.
- A decision tree is essentially a series of branching rules based on the predictors.
- To build a classification tree, we use recursive binary splitting, and aim to increase node purity with each split. A stopping criterion is specified, e.g. minimum node size, maximum tree depth.
- At each branching point, only a random subset of all predictors are considered as potential split candidates. This is done to decorrelate the trees.

8.5 Support Vector Machine

8.5.1 Maximal Margin Classifier

Imagine we have only two predictors, x1 and x2, and we plot our training observations on a graph of x2 against x1 as follows. Now if asked to draw a line that separates the two classes (yellow and blue), where would you draw it?

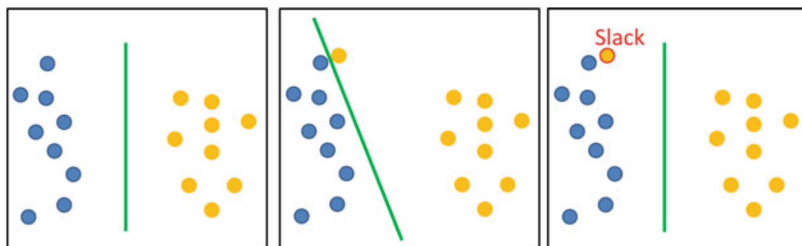


There are in fact infinitely many possible lines that could be drawn to separate the yellow and blue observations in this case. However, we naturally tend to draw a line with the largest **margin**—the one furthest away from the training observations (i.e. we prefer the line on the left to the one on the right). Intuitively, we understand why—the margin reflects our confidence in the ability of the line to separate the two classes. Therefore, we want this margin to be as big as possible.

Once we have determined the separating line, we can easily predict the class of a test observation, by plugging its values of x1 and x2 into the equation of the line, and see if we obtain a positive or negative result. The observations that lie on the margin (dashed box), closest to the separating line, are known as **support vectors** (points with black outline). Note that the position of the line depends solely on the support vectors. If we erase all the other data points, we will still end up drawing the same line. In this way, the other data points are redundant to obtaining the solution.

We can extend this basic premise to situations where there are more than two predictors. When there are 3 predictors, the data points are now in a 3-dimensional space, and the separating line becomes a separating plane. When there are p predictors, the data points are in a p-dimensional space, and so we now have a (p-1)-dimensional **separating hyperplane**.

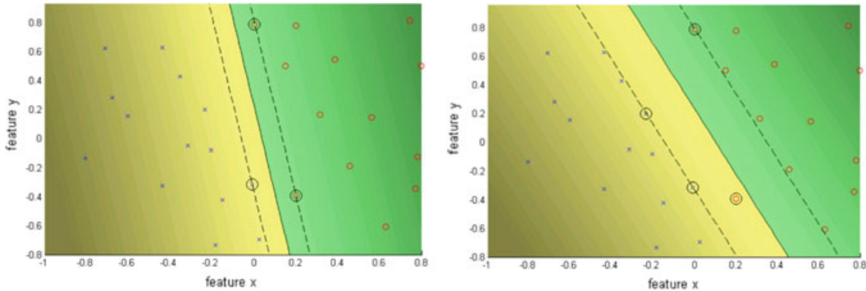
8.5.2 Support Vector Classifier



Now imagine we have an outlier in the yellow group, which causes the position of the separating line to shift dramatically (second box). We are uncomfortable with

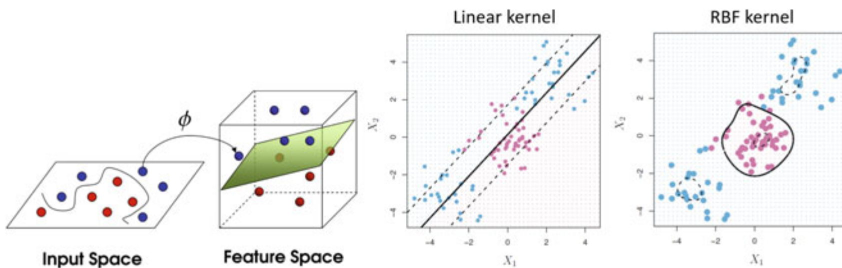
this new line because it has been unduly influenced by a single data point and is probably not generalizable to the testing data. Ideally, we want the line to remain in its original position, ignoring the outlier (third box). To achieve this, we allow some "**slack**" for data points to be on the "wrong" side of the hyperplane in exchange for a more robust hyperplane against outliers.

The tuning parameter 'C' controls the amount of slack—when C is small, more slack is allowed (more tolerant of wrongly classified points), resulting in a softer (but wider) margin. The value of 'C' is usually chosen by cross-validation (see Sect. 8.6).



Hard margin (large C) (left); soft margin (small C) (right)

Given a set of data points that are not linearly separable on the input space, we can use a **kernel function** Φ to project them onto a higher-dimensional feature space and draw the linear separating hyperplane in that space. When projected back onto the input space, the decision boundary is non-linear. The kernel function can also be chosen by cross-validation (see Sect. 8.6), or commonly the radial basis function (RBF) kernel is used.



Summary

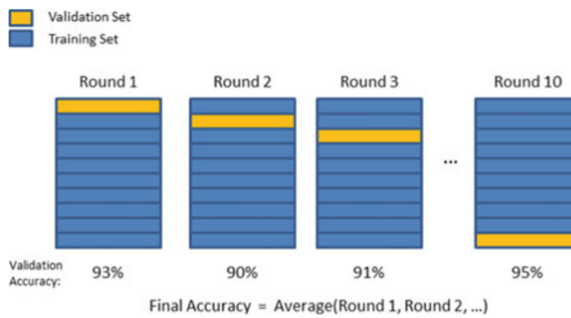
- In SVM, we want to draw a (p-1)-dimensional separating hyperplane between the classes, where p is the number of predictors.
- If multiple hyperplanes are possible, we choose the one with the largest margin.
- To make the separating hyperplane more robust to outliers, we tolerate some observations on the wrong side of the hyperplane. The tuning parameter C controls the amount of slack given. A smaller C results in a softer margin.

- Given a set of data points that are not linearly separable, we can use a non-linear kernel function (e.g. radial basis function, RBF) to project them onto a higher-dimensional space and draw the separating hyperplane in that space.

8.6 Miscellaneous Topics

In this section, we will cover 3 more concepts that are important for the hands-on exercise later.

8.6.1 Cross-Validation



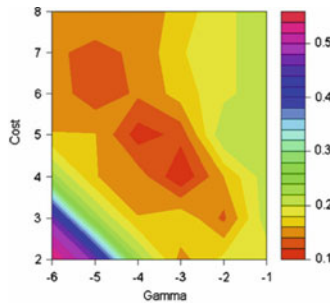
Cross-validation (CV) is a method of resampling often used to choose (tune) parameters of a model. We should not arbitrarily choose model parameters ourselves if we cannot justify or defend these choices that may impact model performance. CV helps us to make the best choices that maximize model performance based on the available data.

In k-fold CV, we split the *training* data into k folds, take one fold to validate and remaining k-1 folds to train. We then calculate a chosen performance metric (e.g. accuracy or error rate), repeat k times and take the average result. Note that we do not touch the independent set of testing data until the model is complete for evaluation.

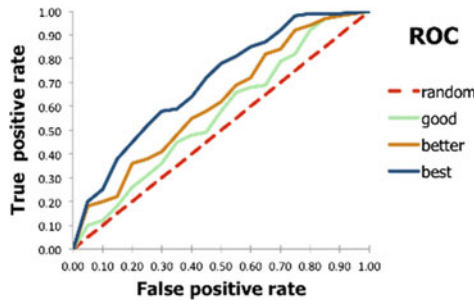
Examples of parameters that could be tuned for a random forest model are the number of trees, the number of predictors considered at each split and the maximum tree depth or minimum node size. Examples of parameters that could be tuned for a SVM model are the amount of slack tolerated (C), the kernel and kernel coefficient. Before building any model, check the library's documentation to see what tuning parameters are available.

When there are two or more parameters we wish to tune concurrently (e.g. number of trees *and* maximum tree depth for a random forest), we can turn to **Grid Search CV**. We first define the range of candidate values for each parameter through which

the algorithm should search. The algorithm then performs CV on all possible combinations of parameters to find the best set of parameters for our chosen evaluation metric.



8.6.2 Receiver Operating Characteristic (ROC) Curve

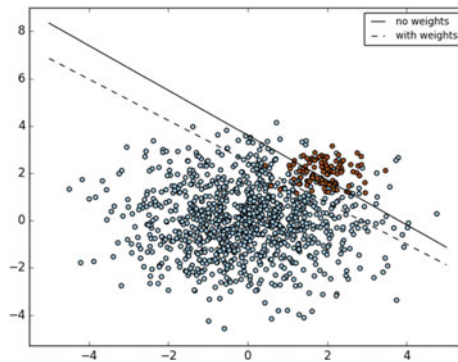


Receiver Operating Characteristic (ROC) curves are often used to evaluate and compare the performance of various models. It is a plot of true positive rate (sensitivity) against false positive rate (1-specificity), and illustrates the trade-off between sensitivity and specificity. Sensitivity refers to the proportion of positives that are correctly classified as positive (true positive rate), while specificity refers to the proportion of negatives that are correctly classified as negative (true negative rate). The overall performance of a classifier is given by the **area under the curve (AUC)**. An ideal ROC curve will hug the top left corner of the graph, maximizing the AUC. Random guessing is equivalent to AUC of 0.5.

8.6.3 *Imbalanced Data*

It is quite common to encounter **imbalanced datasets** in medicine, where most of the samples belong to one class, with very few samples from the other class. Usually, the number of negatives (non-events) significantly outweighs the number of positives (events). This makes training of the models difficult, as there is sparse data to learn how to detect the minority class, which tends to get "overwhelmed" by the majority class.

Possible solutions include under- or over-sampling to create balanced datasets, or re-weighting the sample points. For example, in this SVM model, if **class weights** are applied (dotted line), we penalize the misclassification of the minority class (red) more than the majority class (blue), i.e. we sacrifice the majority class to prioritize the correct classification of the minority class. In doing so, we obtain a better separating hyperplane than if class weights were not applied (solid line).



Summary

- Cross-validation is a resampling method that can be used to tune parameters of a model.
- In k-fold CV, we split the training data into k folds, take one fold to validate and remaining k-1 folds to train. Then calculate the chosen performance metric, repeat k times and average the result.
- A Receiver Operating Characteristic (ROC) curve is a plot of true positive rate (sensitivity) against false positive rate. An ideal classifier will produce a curve that hugs the top left-hand corner, maximizing the area under the curve (AUC). Random guessing is equivalent to AUC of 0.5.
- When dealing with imbalanced data, we can under- or over-sample to create balanced datasets, or apply class weights.

8.7 Hands-on Exercise

8.7.1 Sample Code Review

Let us now review some sample code for a simple machine learning project together. Open the notebook `sample.ipynb`. The premise for this project is described at the top.

We begin by importing the libraries we need, the most important of which is `sklearn`, a library for machine learning containing functions for creating various statistical models and other useful functions (Geron 2017). The code in this sample is interspersed with **comments**, indicated by #, explaining what each code block does.

In the Data Preparation section, we load the dataset into `data` with `read_csv()` and check its `head` and `shape` to make sure they match what we expect (see Sect. 8.2.4 if this is unfamiliar to you). We then split `data` into the predictor variables (named `x`) and response variable (named `y`) using its `values` method and appropriate indexing (see Sect. 8.2.4 for more help). Again, we perform a sanity check on the shapes of `x` and `y`. Next, we feed `x` and `y` into the `train_test_split()` function from `sklearn` to split our data into training and testing sets. The argument `test_size = 0.3` indicates that we want to use 30% of the observations for testing, with the remaining 70% for training. The `random_state = 123` argument indicates the seed for the random number generator. Fixing the seed (any random number is okay) ensures that we obtain the same train-test split every time for reproducibility. If this argument was not specified, we would obtain different train-test splits each time this code is executed. Lastly, we perform sanity checks again—we have 773 samples in the training set and 332 samples in the testing set. In both sets, more patients have benign tumour than malignant cancer, so we have some imbalanced data.

In the Model Building section, we see that it is in fact very simple to create the models using `sklearn`. We instantiate two objects `rf` and `svm` by calling `RandomForestClassifier()` and `SVC()` from `sklearn` respectively. Then, we `fit()` them with the training data. The `class_weight='balanced'` argument indicates that we want to apply class weights to address class imbalance. The `n_estimators = 30` argument for the random forest (RF) model indicates the number of trees in the forest. The `kernel='linear'` argument for the support vector machine (SVM) model indicates a linear kernel function (as opposed to RBF for example). We have defined a custom `score()` function, which when given a model and testing data, uses the model's innate `score` method to calculate its overall test accuracy, specificity and sensitivity. Lastly, we present all the scores neatly in a dataframe. Both models have similar test accuracies (RF 83%, SVM 85%). The RF model has higher specificity (90% vs. 87%) while the SVM model has higher sensitivity (82% vs. 73%).

In the Parameter Tuning section, we use grid search cross-validation to find the best maximum depth of trees for the RF model and best C parameter for the SVM model. We define the parameters and range of candidate values to search in `parameters`. (Increasing the range and granularity of our search would be more thorough but at the expense of computation time.) We then input the model and `parameters` to the function `GridSearchCV()`. The `cv = 5` argument indicates that we want to use 5 folds for cross-validation. `GridSearchCV()` returns the tuned model which we `fit()` and `score()` again. We see that after tuning, both models perform slightly better (overall test accuracies RF 85%, SVM 86%). The best `max_depth` was determined to be 6 and the best `C` was 0.1.

In the Model Evaluation section, we use the tuned models to generate predicted probabilities on the testing data, and input them with the true outcome (y) labels into `roc_curve()` to obtain a series of true positive rates (`tpr`) and corresponding false positive rates (`fpr`). We then graph these `tpr` and `fpr` using the `plot()` function from `pyplot`, forming ROC curves. `auc()` is used to calculate the area under the curve for each model. We see that the ROC curves and AUC for both models are similar (RF 0.91, SVM 0.92).

In addition, we visualize the top 5 most predictive features and their relative importance in the RF model. We do this by calling the `feature_importances_` method from the `rf` model, which returns the importance of each feature based on the total decrease in Gini index method described in Sect. 8.4.2. We sort them in order and obtain the indices of the last five (with highest importances). We then match them to column names from `data` based on their indices. Finally, we graph the information on a horizontal bar plot using `barh()` from `pyplot`.

8.7.2 Hands-on Exercise

You are now ready to apply all that you have learnt! Complete the questions in `exercise.ipynb`. You may adapt code from `sample.ipynb` as a template, but you will need to make necessary changes as appropriate. Copying-and-pasting without understanding will most certainly lead to errors! When you are done, check your answers against the suggested solutions in `solutions.ipynb`.

I hope this chapter has been a useful introduction to machine learning and to programming in Python for those who are new. We have barely just scratched the surface of this vast, exciting field. Indeed, there are many more modelling techniques beyond random forest and support vector machine which we have discussed here. The table below lists some of the popular algorithms currently. You should have sufficient foundation now to explore on your own. Many of these methods are implemented in `sklearn` and you can Google the documentation for them. The best way to make all of this come alive is to design and implement your own machine learning project that is of interest and value to you or your organization.

Supervised Learning	*Unsupervised Learning*
• K-nearest neighbours • Regression (linear, logistic, polynomial, spline etc.) ± regularization • Linear/quadratic discriminant analysis • Tree-based approaches: decision tree, bagging, random forest, boosting • Support vector machine • Neural network	• Principal components analysis • Clustering • Neural network

References

Géron, A. (2017). Hands-on machine learning with scikit-learn and tensorflow: concepts, tools, and techniques to build intelligent systems. O'Reilly Media.

Guttag, J. (2013). Introduction to computation and programming using Python. The MIT Press.

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). An introduction to statistical learning with applications in R. Springer.

Suggested Readings

Codementor. Introduction to Machine Learning with Python's Scikit-learn. https://www.codementor.io/garethdwyer/introduction-to-machine-learning-with-python-s-scikit-learn-czha398p1.

DataCamp. Introduction to Python. https://www.datacamp.com/courses/intro-to-python-for-data-science.

DataCamp. Kaggle Python Tutorial on Machine Learning. https://www.datacamp.com/community/open-courses/kaggle-python-tutorial-on-machine-learning.

Google for Education. Google's Python Class. https://developers.google.com/edu/python/.

Kaggle Learn. Introduction to Python. https://www.kaggle.com/learn/python.

Kaggle Learn. Pandas. https://www.kaggle.com/learn/pandas.

Towards Data Science. Logistic Regression using Python (scikit-learn). https://towardsdatascience.com/logistic-regression-using-python-sklearn-numpy-mnist-handwriting-recognition-matplotlib-a6b31e2b166a.

Udacity. Introduction to Machine Learning. https://eu.udacity.com/course/intro-to-machine-learning-ud120.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.