Introduction to Julia Machine Learning

Using ScikitLearn

Athul Sudheesh

Preface

Introduction to Julia Machine Learning with Scikit-Learn is an open access book aimed at undergraduate students in non-CS majors who are taking their first course in machine learning. The book assumes very little to no prior knowledge of programming. It is designed in such a way that the book helps the students to get to speed developing machine learning models in the shortest time without diluting the fundamental concepts in ML. Although the book provides some introduction on the basics of setting up a project in Julia, it is no definitive guide to either programming or Julia Language.

Motivations for writing this book:

1. At the time of writing this book, there exists a plethora of books on the ScikitLearn python library but none on the Julia port of ScikitLearn. While an experienced Julia user or someone who is new to Julia but has used python scikit-learn finds the documentation of ScikitLearn.jl complete and enough, it might be overwhelming for a complete beginner to both Julia and the ScikitLearn ecosystem. This book exists to cater to that audience.
2. Most introductory books I have come across try to include as many machine learning models as they can, and in the end, they become a survey of the models and their implementation. While these books still cover the foundational concepts in a beautiful manner, they are often lost in the haystack of model details. In this book, we adopt a concepts-first approach compared to the models-first approach adopted by the vast majority of introductory textbooks on applied machine learning. The goal of this book is not to replace these existing introductory books but rather to complement and act as a prequel to them.

Acknowledgement

I would like to thank my mentor, Dr. Richard M. Golden, for training me rigorously in Statistical Machine Learning and providing me with ample opportunities to fine-tune my statistical teaching skills.

Chapter 1

Hello Julia!

In this chapter you'll learn:

1. How to install & setup Julia & Visual Studio Code in your computer.
2. Understanding your Integrated Development Environment (IDE).
3. How to install packages to extend your Julia's capabilities.
4. What Project environments are and how to configure them.

1.1 Installation & Setup

i Note

This section is written for complete beginners to programming. If you know how to configure the language extensions in VS Code, please skip this section.

1.1.1 Installing Julia

To install the latest version of Julia, go to https://julialang.org/downloads/ and download the *Current stable release* corresponding to your operating system and architecture. For Windows machines, in most cases, you need to download the installer for 64-bit. For M1 Macs, it is recommended to download the Intel/Rosetta version than the M-series processor as the later version may be unstable. During the installation process, you might want to note down the installation directory for Julia (copy the path somewhere handy). This path might be required to configure your VS Code.

1.1.2 Installing Visual Studio Code

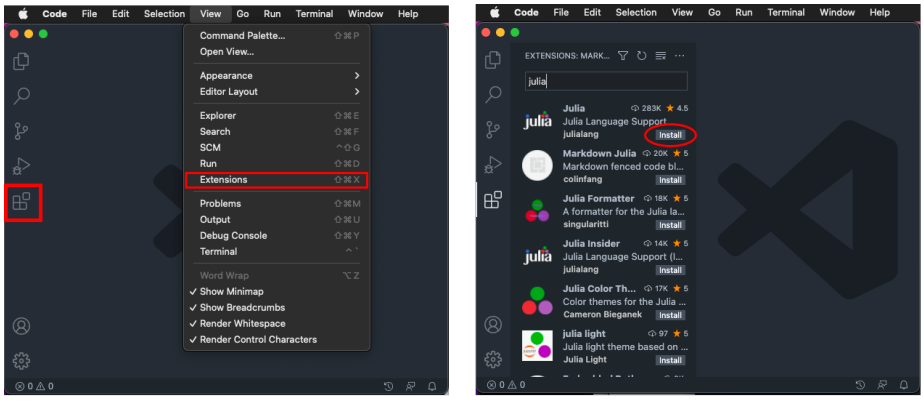
> **Note**
>
> Don't confuse Visual Studio Code (a.k.a VS Code) with Visual Studio; they are two different applications.

You can install the latest version of VS Code from their home page.

1.1.3 Installing Julia Extension for VS Code

Once your VS Code installation is complete, you can open the application either from the Desktop (Windows) or the applications launchpad (Mac). After launching the VS Code application, there are three ways you can access the VS Code Extensions panel:

- via hotkeys: Press `Ctrl + Shift +X` (for Windows) or `Cmd + Shift +X` (for Mac).
- via menu bar: From your VS Code top menu bar choose `View` –> `Extensions`
- via icons on the left of your VS Code application: Find and click on the icon with four squares, where one piece is detached from the rest (Figure 1.1. a).



(a) (b)

Figure 1.1: **Installing Julia extension**. Figure (a) illustrates the two ways you can access the extension panel. Figure (b) illustrates how to search and find the Julia extension.

This will open-up a panel towards the left side of your VS Code application; that's your extensions panel. Towards the top of your extension panel you will see a search box where you have to search for `Julia`. In the results, you will see

`Julia` listed as the top entry. Right next to it, you will also see the `Install` button. Click on it. (Figure 1.1. b).

1.1.4 Checking if the Julia Extension is configured properly

In most cases VS Code should be able to automatically configure the path to julia executable. To check if you have got everything right:

- Do `Ctrl + Shift + P` (Windows) or `Cmd + Shift + P` (Mac)
- Type *Start Julia REPL* and hit enter.

If a new panel pop-up at the bottom of your VS Code with `julia` prompt (as shown below), that means your VS Code Julia extension is properly configured

```
julia>
```

1.1.5 Fixing the Extension Configuration Manually

If you don't get the `julia` prompt that means VS Code has failed to automatically configure the path to julia executable. To manually configure, follow these steps (*also refer to Figure 1.2*):

- Open your VS Code extensions panel using any one of the methods mentioned above.
- Search for Julia. Now instead of seeing the install button next to `Julia` entry, you will see a gear. Click on it.
- Upon clicking on the gear button, a drop down menu will pop up. Choose `Extension Settings` from the list. This will open up the setting page.
- In the setting page, you should find a search box at the top. In the search box, type `julia executable`
- Now you will see an empty field with the title "Julia: Executable Path".
  - (Windows users): In this field, you need to enter/paste the path you copied during step 1 (Section 1.1.1). (Make sure the path ends with `\bin\julia.exe`).
  - (Mac users): Run julia from the applications launchpad (just the way you open any other application in Mac). Once the Julia application is open you will see a path within single quotes right before the Julia logo banner. Copy everything between those quotes (excluding the quotes) and paste them into the "Julia: Executable Path" field in VS Code Julia extension.

- Close VS Code completely and open it again for the changes to reflect.
- Now repeat the steps mentioned in Section 1.1.4 to check if your configuration is working.

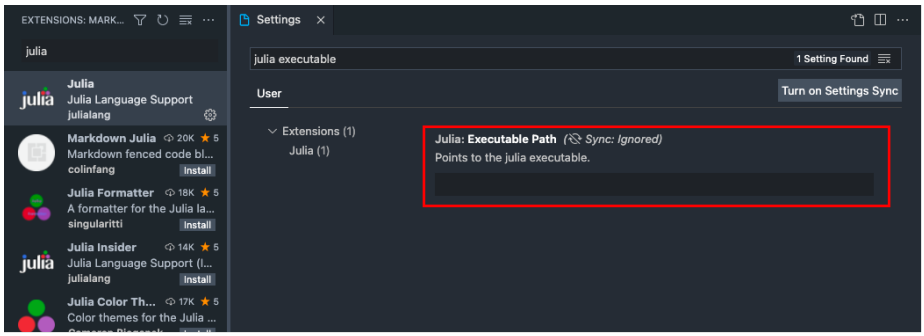


Figure 1.2: **Configuring Julia extension manually.** Paste the path you copied during step 1 (Section 1.1.1) in the textbox area highlighted.

1.1.6 Setting Julia as the default language in VS Code

- Follow the instructions described in Section 1.1.5 (until step three) to access the extension settings.
- Clear all existing text in the search box of extension setting and search for `default language mode`.
- In the textbox for *Files: Default Language* that appear, type `julia` (make sure all are lowercase).

1.2 Know Your IDE

IDE stands for Integrated Development Environment and are software that combine developer tools into a unified user interface. The primary goal of an IDE is to increase the productivity of the developer by automating as many redundant configuration steps as possible. Modern IDEs like VS Code provide functionalities like syntax highlighting (highlights different component of language in different fonts and color), code completion (similar to word completion in MS Word), debugging (tools to debug your code when it is not behaving the way you wanted it to behave), code search (a local search engine for your project), file explorer, language terminal (for quick prototyping), and many more (Figure 1.3).

VS Code, by default, is a simple code/text editor and it is the extensions (similar to the Julia extension you installed) that bring the complete IDE experience to VS Code users. Figure 1.3 provides a visual overview of the VS Code environment during a typical Julia workflow with an active project environment. To activate the Julia language environment inside VS Code:

- Do `Ctrl + Shift + P` (Windows) or `Cmd + Shift + P` (Mac). This opens the Command Palette with a search box.
- Type `Julia REPL` into the search box and hit enter.

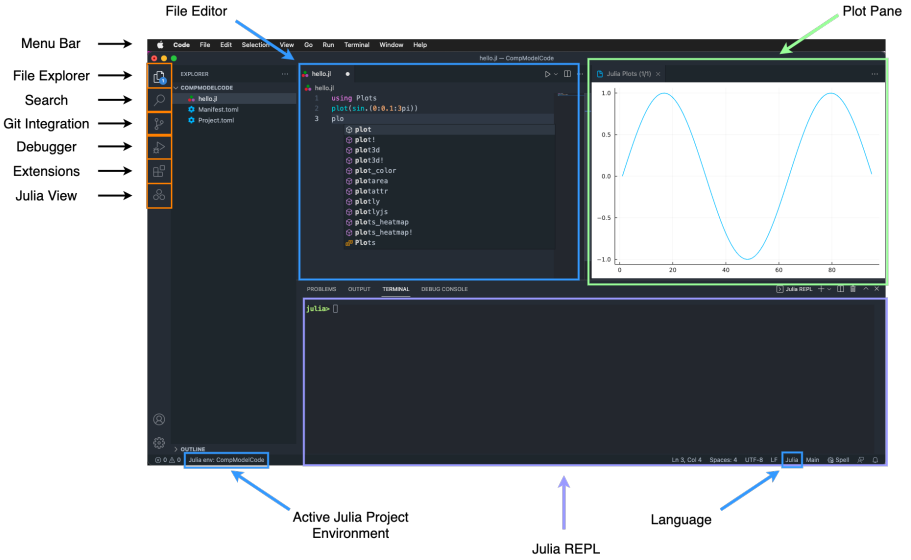


Figure 1.3: Julia VS Code IDE.

If the Julia extension is properly configured, the above commands will open a new terminal with `julia>` prompt (in most cases as the bottom panel in VS Code window). This terminal with `julia>` prompt is commonly known as the REPL.

1.2.1 Julia REPL

REPL stands for Read-Evaluate-Print-Loop and is a method for exploratory programming and debugging. Julia's REPL provides different prompt modes and the default one is the Julian (`julia>`) mode.

- **Julian Mode.** In julian mode, you can run any julia commands and the results will be displayed within the same terminal. It is very common among julia programmers to use the julian mode in the REPL to try out simple algorithms and ideas. After placing the cursor on the REPL, if you press the up arrow or down arrow, you can access your REPL history (i.e. commands you ran in the REPL).
- **Help Mode.** To access help mode, first place the cursor on the REPL and press `?` in your keyboard. You will see that the `julia>` prompt have changed to `help?>` prompt. This means you are in the help mode. Inside the help mode if you type a function name and hit enter, julia will attempt to print the documentation associated with that function/command in the same terminal.
- **Package Mode.** Package mode can be accessed from the julian mode

by pressing `]` key. This will turn the `julia>` prompt to `(@v1.7) pkg>` prompt. Package mode is needed for installing, updating, and removing julia packages from your projects and computer.

To return to the default julian mode from one of the other modes, press backspace.

1.2.2 Installing Packages

Once you are in the package manager mode you can install a package using the command `add`. For e.g., to install the `StatsBase` package you'll enter `add StatsBase` into your package manager mode and hit enter. Once the installation of the package is over, the prompt will return back to `(@v1.7) pkg`. To remove a package you use the `rm` command, and to update a package you use `update` command. Just like the `add` command, you also need to pass the name of the package in all these cases. You can use the `st` command to see the list of packages installed in your computer/project environment.

1.2.2.0.1 But What are Packages?

A Julia package or a library is a collection of functions and sub-modules surrounding an idea or concept bundled as a single unit. Each function can be considered as a collection of code whose objective is to perform a specific task. The standard Julia installation comes with only a handful of very important packages to get you started. To extend the capabilities of julia, you install packages with the help of julia's package manager (Section 1.2.2).

1.2.3 Julia Files

Although you can completely develop julia packages/programs/scripts within the julia REPL, an easier and faster workflow for developing code is by writing the commands in a file and running that file in the julia REPL. To open a new file, do `Cmd + N` (Mac) or `Ctrl +N` (Windows). You can also open a new file using the menu bar: Choose `File` –> `New File`. If you have configured the default language for VS Code as julia (Section 1.1.6), the newly opened file will be a julia file. It is important for VS Code to know the type of the file for syntax highlighting, auto code completion and for running the file in the appropriate language's compiler. Once you have a file open in your VS Code, you can start writing your code line by line in that file.

1.2.4 Running Your First Julia Script

Suppose you wanted to write a julia script to solve for hypotenuse using the pythagoras theorem. You know that as per the pythagoras theorem, $c = \sqrt{a^2 + b^2}$, where c is the hypotenuse, and a and b are the sides of a right triangle. You also know that $(3, 4, 5)$ is a pythagorean triple. So now let's implement this in code and see if it gives the right answer.

As a first step you enter the following lines of code into your newly opened julia file in VS Code:

```
a = 3
b = 4
c = sqrt(a^2 + b^2)
```

Now save this file either using `Cmd + S` command or `File` –> `Save`. You can give any names you want, but it is always recommended to use meaningful names (in this case, say `pythagoras.jl`) so it is easier to find these files later. While saving the file also make sure the file have a `.jl` extension. Once you have saved the file, you have three ways to run this script:

- Manually run the script line by line.
- Run the script as a whole.
- Run the file from REPL.

To manually run the script line by line, you can place the cursor on the first line of code and then do `Shift + Enter`. If the cursor hasn't moved automatically to the next line, you can use the down arrow key to move the cursor to the next line. Now you repeat `Shift+Enter` until the last line of code in your file. Once a line of code is executed, the output of that command is either shown right next to the command or is printed in the REPL.

To run the script as a whole, you click on the play button you see in the tab bar of VS Code. While running scripts using this method, only the output of the last line of code is displayed in the REPL. In our case, the output will be:

```
5.0
```

To run the file in REPL, you type `include("YourFileName.jl")` (in our case `include("pythagoras.jl")`) into your REPL and hit enter. The codes' behavior will be similar to the one when you run the script as a whole using the run button.

1.3 Project Environments

A good programming practice is to always have separate folders for each Julia project you are working on. However, just having separate folders isn't going to ensure either reproducibility or an isolated workspace. (Reproducibility is the property of your project/code to behave exactly in the same way in a computer other than one in which it was developed.). To have an isolated workspace, on top of having a separate folder for your projects, you should also be having what's called separate project environments (Figure 1.4). Project environments are like isolated pockets of spaces where your interaction with one pocket doesn't affect the state of another pocket.

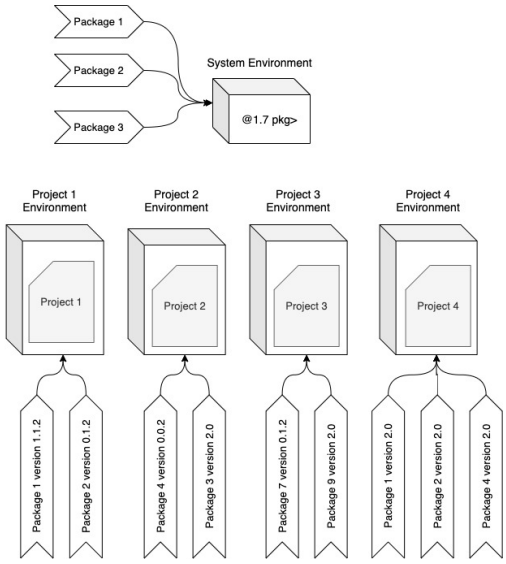


Figure 1.4: A schematic diagram to understand the concept of system environment and project environments.

When you install a specific version of Julia, Julia creates an environment for that particular version. For example, when you installed Julia 1.7 in your computer, julia also created an environment with the name 1.7. This is sometimes referred as the global environment or the system environment. If you didn't activate a particular project's local environment before staring to work on your project, by default the system environment will be chosen and all your interaction with the package manager will be affecting the package state of your Julia's system environment. For every project environment (including system environment), Julia creates two files: Manifest.toml and Project.toml. The goal of these files is to captures the list of packages along with their version number that you are installing within that environment.

To activate a local environment for your project:

- Open your project folder using VS Code: `File` –> `Open Folder`.
- Now start the Julia REPL using `Ctrl + Shift + P` (or `Cmd + Shift + P`).
- Enter package manager mode. If you are seeing `(@v1.7) pkg`, that means you are in Julia's system environment.
- To create \ activate local environment for your project, type `activate .` and hit Enter.
- If your project environment was successfully activated, `(@v1.7) pkg` will turn into `(Your Folder Name) pkg`.

Chapter 2

Your First Model

In this chapter you'll learn:

1. What is machine learning and what are the different types of learning algorithms.
2. What do you mean by a model in machine learning.
3. How to implement a simple model using `ScikitLearn.jl`

2.1 What is Machine Learning?

Machine Learning is a sub-field of statistics and optimization where your goal is to design, develop, and analyze algorithms that can learn patterns in the data. Algorithms can be thought of as procedures you need to follow to achieve a goal. Some examples of instances in your life where you use an algorithm include recipes for food, instructions for the direction to a place, strategies for solving a math problem, etc. (Computer algorithms are definitely different from the above examples, but I hope you got the general gist of what an algorithm means) Let's take the example of food recipes to understand some concepts in machine learning further.

Everyone who has learned to cook by themselves knows that the meal isn't guaranteed to taste that well the first time they try a new recipe. But with multiple attempts, you learn to adjust the spiciness, sourness, sweetness, gravy level, etc., to the right proportion that you will be successful in preparing an outstanding meal. If they were to record each of their attempts in a table, it would have looked something like this:

Note: Values in the above table were randomly generated.

In most cases, a table like the above one is called the data and each of your

Table 2.1: Food ingredients and their proportions.

	Chilly_Powder	Sugar	Salt	Pepper	Broth_Oz	Serves	Tastes
	Float64	Int64	Float64	Int64	Float64	Int64	String
1	2.5	5	0.0	3	5.0	5	Edible
2	1.5	4	0.5	0	10.5	2	Average
3	2.5	4	0.0	2	10.5	5	Non-Edible
4	1.0	4	2.0	1	11.0	1	Best
5	2.0	1	3.0	2	5.0	5	Average
6	2.0	2	0.0	4	6.5	3	Best
7	2.0	0	3.0	4	12.0	4	Non-Edible
8	2.5	1	0.5	1	4.5	4	Average
9	3.0	0	1.0	4	3.0	2	Best
10	2.5	3	1.0	4	3.0	3	Edible

attempts (each row) is called an observation. With a data like this I can do 2 things:

1. Learn how values for each of `Chilly_Powder`, `Sugar`, `Salt`, `Pepper`, `Broth_Oz` and `Serves` influence the `Tastes` and use that information to come up with the best combination of values to ensure `Best` taste all the time. This is called ***inferential modeling***.
2. Given a set of values for `Chilly_Powder`, `Sugar`, `Salt`, `Pepper`, `Broth_Oz` and `Serves`, I can predict if the meal is going to be `Edible` or not. This is called ***predictive modeling***.

If we use the machine learning terminologies, the columns `Chilly_Powder`, `Sugar`, `Salt`, `Pepper`, `Broth_Oz` and `Serves` are called ***features*** and the column `Tastes` is called ***target***. The degree of effect each variable has on the `Tastes` are called parameters.

The mathematical representation of the above information in a functional form is called a ***model***. So, for the food recipe example, our model is:

Chances (Probability) of the meal being edible = $f($ $\theta_1 \times$ `Chilly_Powder` $+ \theta_2 \times$ `Sugar` $+ \theta_3 \times$ `Salt` $+ \theta_4 \times$ `Pepper` $+ \theta_5 \times$ `Broth_Oz` $+ \theta_6 \times$ `Serves` $)$

2.1.1 Supervised, Unsupervised, and Semi-supervised learning

The parameters, $\theta_1, \theta_2, ...\theta_6$, represent the patterns in the given dataset and the goal of a Machine Learning algorithm is to find values for $\theta_1, \theta_2, ...\theta_6$, so that I can reliably predict `Tastes` all the time. This type of machine learning problem, where I have information about the outcome of each attempt, is called ***supervised learning***.

Suppose in our food recipe example, we didn't have information about if the

meal was edible or not; finding patterns in the data is still possible. The type of machine learning problem, where I don't have information about the outcome of each attempt is called ***unsupervised learning***.

Sometimes we use both supervised and unsupervised learning strategy to solve a problem and those types of machine learning problems are called ***semi-supervised learning***.

Now let's learn how to implement a simple model for a supervised learning problem similar to the one we discussed above.

2.2 Implementing a simple model

In this section we'll learn how to implement a simple logistic regression model to predict if a woman is diabetic or not based on some of the medical information we have about that person. The dataset we are using in this section (refer Table 2.2) is structurally similar to the food recipe example we had in the last section. Before getting into the nitty gritty details of model implementation, let's learn more about Logistic Regression.

2.2.1 Logistic Regression

In Section 2.1, we learned that a model is nothing but a mathematical representation of the relationship between the features (aka predictors) and the target. In the diabetes dataset, our target is the variable that predicts if a person is diabetic or not, and all other variables are considered features. We can represent this information in a general form as:

Probability of being diabetic (i.e `Type` $== 1$) =

f(`NPreg`, `Glu`,`BP`, `Skin`, `BMI`, `Ped`, `Age`) =

$$f(\theta_1 \times \text{NPreg} + \theta_2 \times \text{Glu} + \theta_3 \times \text{BP} + \theta_4 \times \text{Skin}+$$

$$\theta_5 \times \text{BMI} + \theta_6 \times \text{Ped} + \theta_7 \times \text{Age}) \tag{1}$$

If we give a logistic parametric form to our function $f(.)$, then it's called the logistic regression model. A logistic function is defined as

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Using equation (2) on (1) we get,

Probability of being diabetic =

$$\frac{1}{1 + e^{-(\theta_1 \times \text{NPreg} + \theta_2 \times \text{Glu} + \theta_3 \times \text{BP} + \theta_4 \times \text{Skin} + \theta_5 \times \text{BMI} + \theta_6 \times \text{Ped} + \theta_7 \times \text{Age})}} \tag{3}$$

Applying different parametric forms to equation (1) yields you different machine learning models. For e.g., if we had used an identity function i.e. $f(x) = x$, the model we got is called the linear regression model (*Note:* Linear Regression models are not used for classification problems. The type of the problem you are trying to solve always restricts the type of models you can use.).

By using an activation function where the function will return `Yes` if the value we get using equation (3) is greater than or equal to 0.50 and return `No` otherwise, we can get prediction from our model that is comparable to the target in our data. The discrepancy between our model's prediction and target is called the ***prediction error***.

Once we have a model defined and the data available, the next step is to use an algorithm to learn optimal values for θ's so that I can predict values of the target consistently. The step where we use an algorithm to learn optimal values for θ's is called ***model training*** and the data we used for training is called the ***training dataset*** in machine learning.

2.2.2 How do Models learn?

Optimization algorithms are what make model training (learning) possible. In this section, let's learn how they work from a birds-eye-view, as explaining the technicalities of how optimization algorithms work is beyond the scope of this textbook.

An optimization algorithm learns pretty much the same way you learn things - through trail and error. With each trial, the goal of the optimization algorithm is to keep reducing the value of prediction error by manipulating the values for the model parameters (θs). After several trials, we get to a point where the prediction error is in an acceptable range and reducing prediction error further is impossible or futile. At that point, we save the values of θ that helped us to reach that particular prediction error value. These saved values for θ are called the ***coefficients*** of our learned model and corresponds to the patterns that were present in our data. Using the learned coefficients of our model, we will be able to make predictions on data the model has never seen. The data that the model hasn't seen is called the ***test dataset*** and the prediction error we get on the test data is called the ***test error*** and the prediction error we were getting during training phase is called ***training error***.

Now let's learn how to implement a logistic regression model and train them on the data we have

Step 1: Project environment activation and Package Installation

Note: We expect that you have created a separate folder for storing all the julia scripts you'll be developing as part of learning with this textbook. To open your

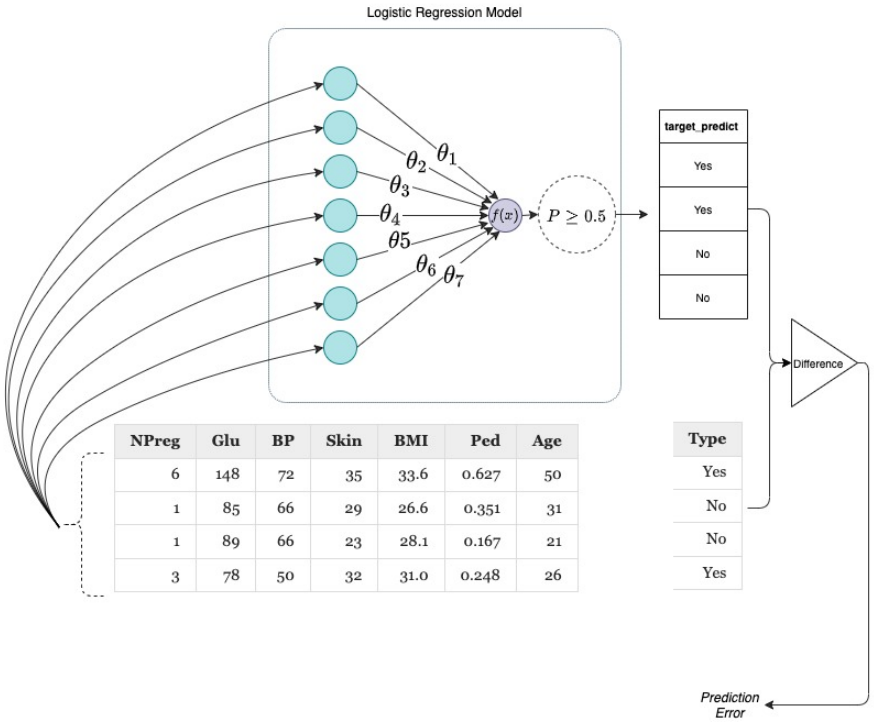


Figure 2.1: Schematic to understand the concept of model training.

project folder in VS Code, you can go to Menu –> File –> Open Folder. From the dialog box that pops up, you can choose the folder you created.

In order to make sure that you are always working in the correct project environment, have the following 2 lines of code towards beginning of every julia script you create:

```
using Pkg
Pkg.activate(".")
```

- Instructions on how to create a new julia script is described in Section 1.2.3 and Section 1.2.4

In this section we will require 3 packages (To learn how to install a package, refer Section 1.2.2):

- `RDatasets`: This package provides an easy access to a lot of toy datasets in machine learning.
- `ScikitLearn`: One of the industry standard packages for doing machine learning projects. Provides utilities for model definition, training, testing, tuning, and much more.
- `DataFrames`: A package for handling data in tabular form.

Step 2: Loading the packages

To load these packages, you can have the following line of code right below the code you wrote in step 1:

```
using ScikitLearn, RDatasets, DataFrames
```

- If you got an error while running the above line of code, most of the time it means one of the three things:
  1. You haven't installed the package that you are trying to load.
  2. You are in the wrong project environment. (This is why we highly recommend you to follow step 1 every time you create a new julia script.)
  3. You have typed a wrong package name. The name of all packages in Julia are case sensitive.

Step 3: Loading the dataset

In this example, we will use the *Diabetes in Pima Indian Women dataset* (available via `RDatasets`). (Instruction on how to load a dataset that is available to you are a `.CSV` file is provided in the Appendix (**?@sec-appendix**)). To load the dataset and show the first four observations, enter the following lines of code:

Table 2.2: Diabetes in Pima Indian Women dataset

	NPreg	Glu	BP	Skin	BMI	Ped	Age	Type
	Int32	Int32	Int32	Int32	Float64	Float64	Int32	Cat…
1	6	148	72	35	33.6	0.627	50	Yes
2	1	85	66	29	26.6	0.351	31	No
3	1	89	66	23	28.1	0.167	21	No
4	3	78	50	32	31.0	0.248	26	Yes

```
diabetes = dataset("MASS", "Pima.te");
first(diabetes,4)
```

- `dataset` is a function from `RDatasets` that provide a nice interface to load datasets in DataFrames format. The `dataset` function accepts two arguments: the data source, and the dataset name. In this case, the name of our dataset was `Pima.te` and the source was `MASS` package in R.
  - **Trivia:** If you see a word with `()` ending, then it is a function. A function is a collection of commands (several lines of codes) sharing a collective single objective. Anything that is passed inside `()` are called arguments. In our example, the objective of `dataset` function was to return the dataset (`Pima.te`) from the source (`MASS`) we mentioned.
- `diabetes` is the name we gave to the variable that stores the data that was returned from the `dataset` function. The variable name is arbitrary and you can give whatever name you like. However, it is always recommended to give meaningful names.

Step 4: Making sense of the dataset

The diabetes dataset that we are using in this section was collected by the US National Institute of Diabetes, Digestive, and Kidney Diseases from a population of women who were 21 years and older and were of Pima Indian heritage living near Phoenix, Arizona. The dataset contains the following information:

- `NPreg`: Number of pregnancies
- `Glu`: Plasma glucose concentration in an oral glucose tolerance test
- `BP`: Diastolic blood pressure (mm Hg)
- `Skin`: Triceps skin fold thickness (mm)
- `BMI`: Body Mass Index ($\frac{\text{weight (Kg)}}{\text{height (m)}^2}$)
- `Ped`: Diabetic pedigree function
- `Age`: age in years
- `Type`: Diabetic or not (according to WHO criteria)

Table 2.3: Accessing multiple columns (for rows from 5:10)

	BMI	Age
	Float64	Int32
1	30.5	53
2	25.8	51
3	45.8	31
4	43.3	33
5	39.3	27
6	29.0	29

Accessing elements in the data

Now let's take a small detour and learn how to access different cells and slice the data.

- To access the 10^{th} row in the data:

```
diabetes[10,:]
```

	NPreg	Glu	BP	Skin	BMI	Ped	Age	Type
	Int32	Int32	Int32	Int32	Float64	Float64	Int32	Cat…
10	9	119	80	35	29.0	0.263	29	Yes

  - The first position in `[]` indicated the row, and the second position indicated column. If you want to choose all columns then you put `:` in the second position.

- To access the column `BMI`:

```
diabetes[5:9,:BMI]
```

```
5-element Vector{Float64}:
 30.5
 25.8
 45.8
 43.3
 39.3
```

- If you want to choose all rows, then you put `!` in the first position instead of `5:9`.

- To select multiple columns:

```
diabetes[5:10,[:BMI,:Age]]
```

- To select all columns except `Type`:

Table 2.4: All columns except Type (for rows from 5:10)

	NPreg	Glu	BP	Skin	BMI	Ped	Age
	Int32	Int32	Int32	Int32	Float64	Float64	Int32
1	2	197	70	45	30.5	0.158	53
2	5	166	72	19	25.8	0.587	51
3	0	118	84	47	45.8	0.551	31
4	1	103	30	38	43.3	0.183	33
5	3	126	88	41	39.3	0.704	27
6	9	119	80	35	29.0	0.263	29

```
diabetes[5:10,Not(:Type)]
```

Step 5: Choosing the features and the target

Our goal in this chapter is to define a supervised machine learning model that can predict if a woman is diabetic or not given their pregnancy history, glucose level, blood pressure, skin fold thickness, BMI, diabetic pedigree function, and their age.

If the value we have to predict is a category, that's called a ***classification problem*** and if the value we had to predict was numeric, it's called a ***regression problem***. Both the examples (food recipe and diabetes) we discussed in this chapter are classification problems. In the food recipe example the categories of the target were: `Non-Edible`, `Edible`, `Average`, and `Best`. For the diabetes dataset, the categories of the target are `Yes` and `No` indicating if a woman is diabetic or not.

We can use the data slicing skills we learned in the previous section to extract the features and the target from the data:

```
features = Array(diabetes[!, Not(:Type)]);
target = Array(diabetes[!, :Type]);
```

- The first line selects all columns except the `Type` column and saves them as an Array in the variable `features`
- The second line selects just the `Type` column and saves them as an Array in the variable `target`

Step 6: Creating a Model Instance

Logistic regression model is one of the most simple, common, and baseline model we use for classification problems. To create a logistic regression model instance, we can import the Logistic Regression function from `linear_models` in `ScikitLearn` package.

```
@sk_import linear_model: LogisticRegression;
simplelogistic = LogisticRegression();
```

- The line `simplelogistic=LogisticRegression()` creates an empty logistic regression model object which can store information about the model, data, learning algorithm, and learned parameters. The fields that are stored in a model object varies depending on the model you are defining.
- the variable name `simplelogistic` is arbitrary and you can give whatever name you like.
- If you are going to create another model instance (say a neural network model), don't reuse the variable name. It's better to choose a different variable name each time you are defining a new model.

Step 7: Training your model

In Section 2.2.2, you learned how an optimization algorithm helps the model to learn patterns in the data. The `fit!` function from `ScikitLearn` implements that procedure.

```
fit!(simplelogistic, features, target);
```

- the `fit!` function takes three arguments: the model you want to train, the features, and the target.
  - Whenever you see an exclamation mark in functions, it means that the function is mutating (changing) the values of one or more arguments passed to that function. In this case `fit!` function is changing the values of θ, which is part of the model definition.

Now you have a logistic regression model (`simplelogistic`) that's trained on Pima diabetes dataset. To see the learned values for θ, you can run the following line of code:

```
simplelogistic.coef_
```

```
1×7 Matrix{Float64}:
 0.138633  0.0373939  -0.00897535  0.0134173  0.0783658  0.921752  0.0190434
```

You can plug in these values into equation (3) to reliably compute the probability of a woman being diabetic.

Code Summary for Chapter 2

```
# Activating the local project environment
using Pkg
Pkg.activate(".")

# Loading the packages
using ScikitLearn, RDatasets, DataFrames

# Loading the dataset
diabetes = dataset("MASS", "Pima.te");
first(diabetes,4)

# Choosing the features and target
features = Array(diabetes[!, Not(:Type)])
target = Array(diabetes[!, :Type])

# Creating a logistic regression model instance
@sk_import linear_model: LogisticRegression
simplelogistic = LogisticRegression()

# Training the model
fit!(simplelogistic, features, target)

# Viewing the learned parameters
simplelogistic.coef_
```

In the next chapter, we will learn how to check if our trained model is a good one or not.

Chapter 3

Evaluating Your Model's Performance

In this chapter you'll learn:

1. How to measure how good your model is.
2. Different metrics to measure goodness of fit and how to use the metrics functions available in `ScikitLearn`.
3. How to interpret the results from metrics functions.

3.1 Did our model learn anything?

In the last chapter, we learned how to train our simple model on the given dataset. But how do we know that our model learned the patterns in the data? Well, think about the human learning scenario. How do we come to the conclusion that somebody has learned something?

Through assessments that test their knowledge.

Similarly, we can put our model to test and see how well they perform on these tests. One of the most common and initial tests you do once you have a trained model is to check the number of times your model predicted the target value correctly, i.e., in our case, how many times our model predicted the women to be diabetic and in fact the women had diabetes as per our data records. This measure of the percentage number of times the model predicts the target value correctly is called the accuracy of the model.

$$\text{Accuracy} = \frac{\text{No. of correct predictions}}{\text{Total no. of predictions}}$$

To compute the accuracy of our model we first need to generate the predicted values for our target. This can be achieved using the predict function in `scikit-learn`.

```
logistic_target_predict = predict(simplelogistic,features);
```

```
4-element Vector{Any}:
 "Yes"
 "No"
 "No"
 "No"
```

Once we have the predicted values, we can pass the predicted values and the target values from our data to the `accuracy_score` function in the `metrics` module in `scikit-learn` to compute the accuracy of our model.

```
@sk_import metrics: accuracy_score
print(accuracy_score(target,logistic_target_predict))
```

```
0.7921686746987951
```

- The results show that our model has an accuracy of 79.22 %.

3.2 Is our model confused?

Although accuracy is a good measure to assess the quality of your model, most often, especially in classification type problems, they don't tell us the complete story.

When we do prediction in a classification task, there arise four situations. For example, in our case:

- the model predicted the woman to be diabetic and was in fact diabetic
- the model predicted the woman to be non-diabetic but was diabetic
- the model predicted the woman to be diabetic but was not actually diabetic
- the model predicted the woman to be non-diabetic and was not diabetic

The first and last cases where our predictions aligned with the actual values are called *true positives* and *true negatives* respectively. The case where the model predicted the woman to be diabetic while she did not have diabetes is called a *false positive* case, and the case where the model predicted the woman to be non-diabetic but she was in fact diabetic is called the *false-negative* case. This is illustrated in Figure 3.1.

When we just focus on a classification model's accuracy, all these information is hidden from us.

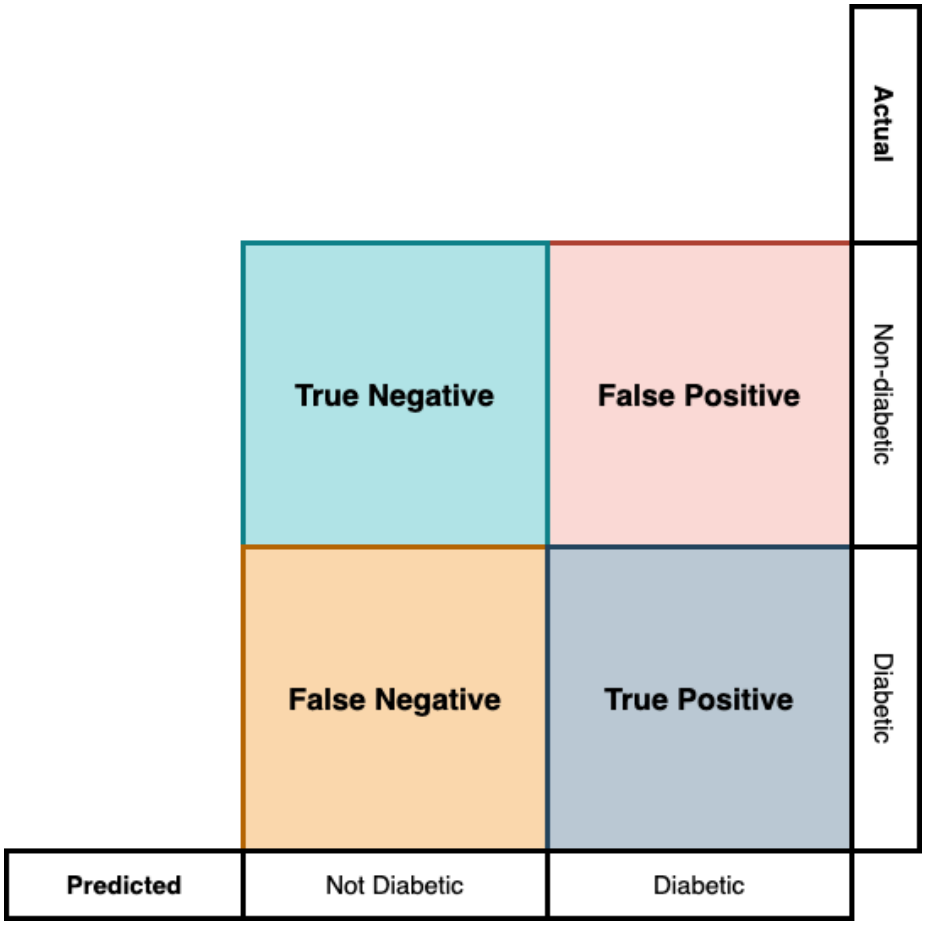


Figure 3.1: Four possible scenarios in the diabetes prediction task.

It is possible to generate a figure like Figure 3.1 in `scikit-learn`. For that, you need two functions, the `confusion_matrix` function to generate the confusion matrix and `ConfusionMatrixDisplay` function to generate the plot.

```
# Generating the confusion matrix
@sk_import metrics: confusion_matrix
cf = confusion_matrix(target, logistic_target_predict)

# Loading the plotting library & confusion matrix plotting function
using PyPlot
@sk_import metrics: ConfusionMatrixDisplay

figure() # Open a new canvas to plot

# Generating the plot
disp = ConfusionMatrixDisplay(confusion_matrix=cf,display_labels=simplelogistic.clas
disp.plot() # Transferring the plot to the canvas
gcf() # Freezing the canvas and printing it.
```

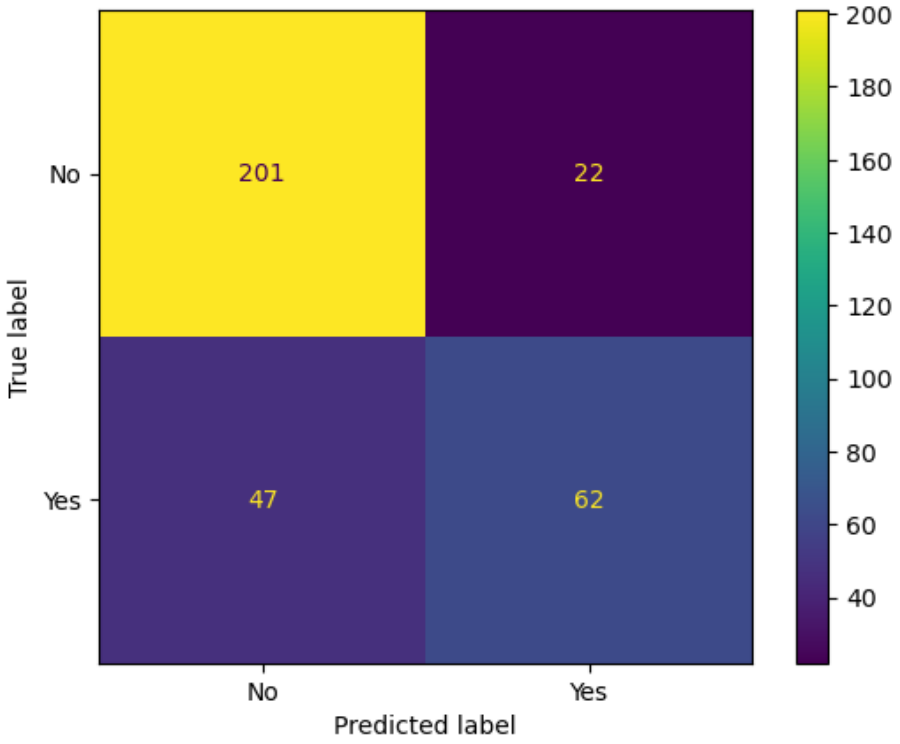


Figure 3.2: Confusion Matrix

3.3 Why our model's confusion pattern matters?

You might still be thinking why these false rates and true rates matter since we already have the accuracy scores. The importance of the confusion matrix comes into play when we consider the consequences of our prediction. For example, if it's a high consequence situation like predicting if somebody has early-stage cancer or not, miss-classifying a person as not having cancer, while they have cancer has a high cost. In such cases, the ML designer needs to look at the false negative rate more closely than the model's overall accuracy. Whereas, in a low consequence situation like credit card approval prediction, the false positive rate matters more than the false negative rate. Because, with higher false negative rate, you might be denying a credit card to a person with a good credit score and fewer chances of defaulting whereas if you have high false positive rate, you will be approving credit cards to people whose chances of defaulting are high.

The measure we are interested in the first case where the cost of missing a positive is high is called the recall or sensitivity of our model. Recall can be computed from your confusion matrix using the formulae:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

```
true_neg, false_neg, false_pos, true_pos = cf
recall = true_pos / (true_pos + false_neg)
```

```
0.5688073394495413
```

The measure we are interested in the second case, where the cost of false positive is higher is called the precision of our model. Precision can be computed using the formula:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

```
precision = true_pos / (true_pos + false_pos)
```

```
0.7380952380952381
```

The above results say that our model has high chances of missing a diabetes patient (due to low recall) and low chances of generating a false positive (high precision). In other words, if the model predicted you as having diabetes, there is a high chance you have diabetes, but if you were diagnosed non-diabetic by the model, you may or may not be diabetic.

If your model is used in a scenario where both false positives and false negatives have high consequences, a better metric to watch for is the F1 score. F1 score is also a better metric than accuracy score for evaluating model's performance when you have imbalances dataset (Imbalanced datasets are datasets that have unequal number of data points across its classes. For e.g., our diabetic dataset will be an imbalanced data if 90% of our data are that of non-diabetic people and only 10% is that of diabetic people.). F1 score is the weighted average of precision and recall and can be computed using the formula:

$$\text{F1 score} = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Precision} + \text{Recall}}$$

```
f1 = (2 * precision * recall) / (precision + recall)
```

```
0.6424870466321243
```

Although we had very good accuracy score, the F1 score for our model suggests that our model isn't that impressive.

To compute the recall, precision, f1 score, and accuracy with lesser lines of code, you can use the `classification_report` function from `metrics` module in `scikit-learn`.

```
@sk_import metrics: classification_report
print(classification_report(target, logistic_target_predict))
```

```
              precision    recall  f1-score   support

          No       0.81      0.90      0.85       223
         Yes       0.74      0.57      0.64       109

    accuracy                           0.79       332
   macro avg       0.77      0.74      0.75       332
weighted avg       0.79      0.79      0.78       332
```

- The first line list the precision, recall, and f1 score with respect to the target "No" and the second with respect to the target "Yes".
- In the above computations, we have calculated precision, recall, and f1 score with respect to the target "Yes". You can check the values we have got manually with the values printed in the classification report.

Code Summary for Chapter 3

```
# Generating the predictions from the trained model
logistic_target_predict = predict(simplelogistic,features);

# Calculating the accuracy score
@sk_import metrics: accuracy_score
print(accuracy_score(target,logistic_target_predict))

# Generating the confusion matrix
@sk_import metrics: confusion_matrix
cf = confusion_matrix(target, logistic_target_predict)

# Loading the plotting library & confusion matrix plotting function
using PyPlot
@sk_import metrics: ConfusionMatrixDisplay

figure() # Open a new canvas to plot

# Generating the plot
disp = ConfusionMatrixDisplay(confusion_matrix=cf,
            display_labels=simplelogistic.classes_)
disp.plot() # Transferring the plot to the canvas
gcf() # Freezing the canvas and printing it.

# Generating the classification report
@sk_import metrics: classification_report
print(classification_report(target, logistic_target_predict))
```

Chapter 4

Generalizability

In this chapter you'll learn:

1. What is generalizability.
2. How to ensure generaziability of your model.
3. What you mean by learning curve, under-fitting, over-fitting, bias and variance.
4. How to implement cross-validation techniques using `ScikitLearn`.

4.1 Did our model cheat?

In the last chapter, we learned how to check our model's performance using various metrics. But how do we know that our model really learned the patterns in the data and is not cheating by rote-memorizing the data? Think about how we would have assessed a human learner in this situation.

When we want to know if students have really learned what we have asked them to learn, we test students on material that is similar to the material that's familiar to them but not exactly the questions they have seen before. Similarly, to check if our models have really learned the patterns in the data, we can test our model against a similar but unseen data. The extent to which the model performance remains invariant with this new unseen data is called that model's ***generalizability***.

The portion of the data we use for training is called the ***training set*** and the portion of the data we use to test is called the ***test set***. We can use the `train_test_split` function from `model_selection` module in `scikit-learn` to create this partition.

```
@sk_import model_selection: train_test_split;
features_train, features_test,
    target_train, target_test = train_test_split(features,
        target, test_size=0.3, random_state=42);
```

- The `train_test_split` function has three important and mandatory arguments.
- The first two are the `features` and the `target`
- The third argument is `test_size` and specifies the proportion of the data that needs to kept aside for testing. In the above code, we have asked to keep aside 30% of the data as test set.

- `random_state` is an optional argument and set's the seed for randomness. Now, instead of training the model on the entire dataset, we'll train our model with training set.

```
@sk_import linear_model: LogisticRegression;
simplelogistic = LogisticRegression();

fit!(simplelogistic, features_train, target_train);
```

Now let's look at our model's performance with both training set and test set.

In-sample performance

A model's performance with training set is also called it's in-sample performance.

```
logistic_target_predict_training =
    predict(simplelogistic,features_train);

@sk_import metrics: classification_report
print(classification_report(target_train,
                    logistic_target_predict_training))
```

```
              precision    recall  f1-score   support

          No       0.84      0.91      0.87       166
         Yes       0.71      0.56      0.63        66

    accuracy                           0.81       232
   macro avg       0.78      0.74      0.75       232
weighted avg       0.80      0.81      0.80       232
```

Out-of-sample performance

A model's performance with test set is also called it's out-of-sample performance.

```
logistic_target_predict_test =
    predict(simplelogistic,features_test);

@sk_import metrics: classification_report
print(classification_report(target_test,
                    logistic_target_predict_test))
```

```
              precision    recall  f1-score   support

          No       0.72      0.98      0.83        57
         Yes       0.95      0.49      0.65        43

    accuracy                           0.77       100
   macro avg       0.84      0.74      0.74       100
weighted avg       0.82      0.77      0.75       100
```

By comparing our model's performance with both the training set and test set, we can see that the overall accuracy of our model slightly dropped for the test case. We can also see that our model had better precision but a little worse recall and f1-score with the test performance compared to training performance. So, we can conclude that our model has an ok-ish generalizability.

4.2 Cross-validation: A robust measure of generalizability

We can extend the concepts of in-sample and out-of sample performance to create a more robust measure of generalizability. There are two motivations for creating this new robust measure of generalizability:

1. When the sample size is small (less data), our training and test performance scores can get flaky and unreliable.
2. There are some bells and whistles (which are called hyperparameters) that we can tweak in our models to improve our models' learning process (This is explained in detail in the coming chapters). If we tweak these hyperparameters with respect to our training data, we might be overfitting our data to the training set (***Overfitting*** happens when our model have high performance on training set but very poor performance on test set.). But instead, if we tweak these hyperparameters with respect to our test data, information in the test data leaks to the model and the data is no more unseen data for the model.

ML developers came up with a solution to this problem by partitioning the

training data into different data blocks, holding out one block as test set, and training on the remaining blocks. Then model's performance on the hold-out test set is saved. This process is repeated until all blocks had its chance of beginning the hold-out test set. Model performance from all these iterations is then averaged to get the robust generalization performance. This process of deriving model performance is called the ***cross-validation*** technique and is illustrated in Figure 4.1.

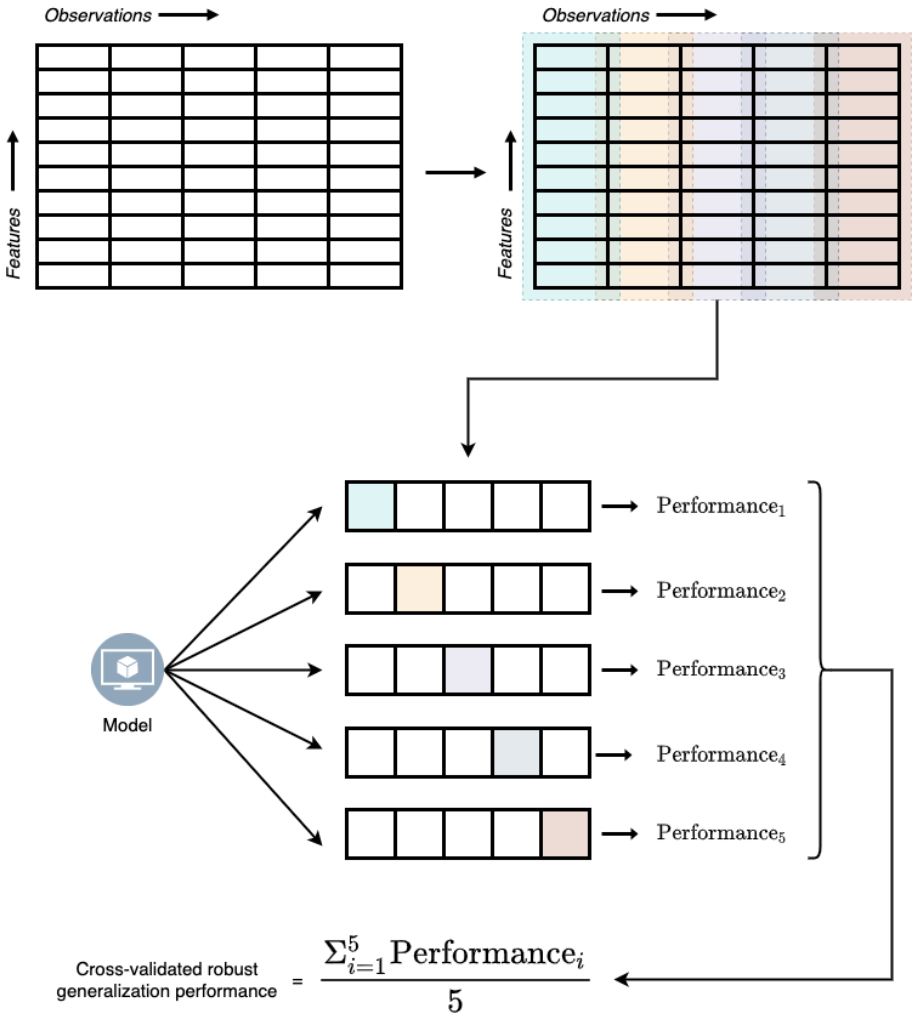


Figure 4.1: Five-fold cross validation

To implement the K-fold cross validation technique we can use the `KFold` function and `cross_validate` function `model_selection` module in `scikit-learn`.

```
@sk_import model_selection: KFold
@sk_import model_selection: cross_validate

cv_results = cross_validate(simplelogistic,
        features_train, target_train,
            cv=KFold(5),
            return_estimator=true,
            return_train_score=true,
            scoring=["accuracy",
                 "recall_weighted", "precision_weighted"]);
```

To print the results from cross validation in a more human readable table form, we can use the following lines of code:

```
cv_df = DataFrame(cv_results)[!,
        Not([:estimator, :fit_time, :score_time])]

rename!(cv_df, ["Test Accuracy",
                "Test Precision",
                "Test Recall",
                "Train Accuracy",
                "Train Precision",
                "Train Recall"])
```

	Test Accuracy	Test Precision	Test Recall	Train Accuracy	Train Precision	Train Recall
	Float64	Float64	Float64	Float64	Float64	Float64
1	0.829787	0.818237	0.829787	0.837838	0.833854	0.837838
2	0.808511	0.801009	0.808511	0.8	0.792456	0.8
3	0.76087	0.742992	0.76087	0.811828	0.804097	0.811828
4	0.847826	0.851999	0.847826	0.811828	0.805028	0.811828
5	0.782609	0.782609	0.782609	0.817204	0.808555	0.817204

To compute the cross validated average model performance measures, we can print the mean of each column in the above table.

```
describe(cv_df)[!,[:variable, :mean]]
```

4.3 Learning curves: Can more data improve model training?

Now we know how to measure a model's generalization performance. But what if our models perform poorly in both training and test sets? The scenario where models fit poorly with training and test sets equally is called ***underfitting***. Underfitting usually happens due to one of the two reasons or both: a) our

Table 4.1: Cross validated average model performance measures

	variable	mean
	Symbol	Float64
1	Test Accuracy	0.80592
2	Test Precision	0.799369
3	Test Recall	0.80592
4	Train Accuracy	0.81574
5	Train Precision	0.808798
6	Train Recall	0.81574

model is too simple for the task at hand, b) we don't have enough data to learn from.

Solving the first problem is relatively simple; we can train a more complex model on the given dataset. However, solving the second problem can get complicated. Gathering more data may not always be feasible and can be expensive. In such cases, before deciding on acquiring more data, we need to make sure more data will solve the problem of underfitting.

The way we figure out if more data can help with training is by training our model with data of different sizes (e.g., using 10%, 50%, and 100% of our data) and check how the model performance is varying as a function of the sample size. The plot that illustrates this relationship is called ***learning curves***.

We can get our model's performance at different sample sizes using the `learning_curve` function from `model_selection` module in `scikit-learn`.

```
@sk_import model_selection: learning_curve;
lc_results = learning_curve(simplelogistic,
        features_train,target_train,
            train_sizes = [0.06, 0.1, 0.25, 0.5, 0.75, 1.0]);
```

- The values we pass to `train_sizes` argument specify the different sample sizes we want to try. In this case we are looking at 10%, 25%, 50%, 75% and 100% of the data.

```
train_sizes, train_scores, test_scores, =
    lc_results[1], lc_results[2], lc_results[3]

using Statistics
# Calculating the error bars
y_ax = vec(mean(test_scores, dims=2))
y_err = vec(std(test_scores, dims=2))

using PyPlot
begin
    figure();
    plot(train_sizes,
        vec(mean(test_scores, dims=2)), label="Cross Val");

    fill_between(train_sizes,
        y_ax - y_err, y_ax + y_err,alpha=0.2);

    plot(train_sizes,
        vec(mean(train_scores, dims=2)), label="Training");
    xlabel("Training Size");
    ylabel("Score");
    title("Learning Curve");
    legend(loc=4);
    gcf()
end;
```

From the above learning curve, we see that our model's accuracy isn't getting that much influenced by increasing the sample size. So, it will be futile to collect more data for our diabetes detection ML system. However, if the blue line in Figure 4.2 had a steeper upward trend, it would have made sense to collect more data to improve our model's performance.

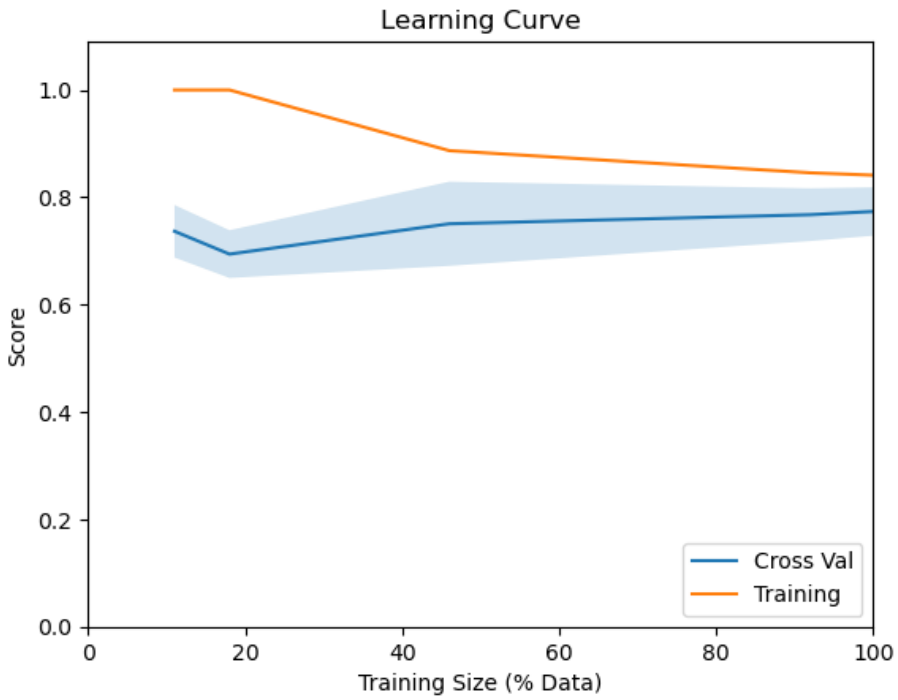


Figure 4.2: Learning Curve for simple logistic regression model on diabetes dataset

Code Summary for Chapter 4

```
# Creating the train-test split
@sk_import model_selection: train_test_split;
features_train, features_test,
    target_train, target_test = train_test_split(features,
        target, test_size=0.3, random_state=42);

# Creating a logistic regression model instance
@sk_import linear_model: LogisticRegression;
simplelogistic = LogisticRegression();

# Fitting the model on training data
fit!(simplelogistic, features_train, target_train);

# Generating the predictions for train data
logistic_target_predict_training =
    predict(simplelogistic,features_train);

# Checking the in-sample performance
@sk_import metrics: classification_report
print(classification_report(target_train,
                    logistic_target_predict_training))

# Generating the predictions for train data
logistic_target_predict_test =
    predict(simplelogistic,features_test);

# Checking the out-of-sample performance
@sk_import metrics: classification_report
print(classification_report(target_test,

# K-Fold Cross validation
@sk_import model_selection: KFold
@sk_import model_selection: cross_validate

cv_results = cross_validate(simplelogistic,
        features_train, target_train,
            cv=KFold(5),
            return_estimator=true,
            return_train_score=true,
            scoring=["accuracy",
                 "recall_weighted", "precision_weighted"]);

# Printing cross validated results in table form
cv_df = DataFrame(cv_results)[!,
        Not([:estimator, :fit_time, :score_time])]

rename!(cv_df, ["Test Accuracy",
                "Test Precision",
                "Test Recall",
                "Train Accuracy",
                "Train Precision",
                "Train Recall"])
# Cross validated means
describe(cv_df)[!,[:variable, :mean]]
```

Chapter 5

Tuning your model

In this chapter you'll learn:

1. What you mean by hyper-parameters and how to set them in your `scikit-learn` model.
2. How to search the model hyperparameter space to find the best model.
3. How to compare models with different hyperparameters.

In the last chapter, we mentioned that models have some bells and whistles that you can tune to improve a model's learning process. These tunable parts of the model are called ***hyperparameters*** in machine learning literature. These are different from the parameters of the model in the sense that model parameters are learned from the data and represent patterns in the data, whereas hyperparameters are parameters set by the machine learning developer and control the model's architecture and learning process. Since the developer sets the hyperparameters, the optimal values that maximize the model performance are often found by trial and error. However, running each model several times with different combinations of hyperparameters and tracking the results manually can get tedious and error-prone. So we use some of the semi-automated methods of finding the hyperparameter.

5.1 Grid Search

One of the simplest and most often used semi-automated method for finding the optimal values for hyperparameters is the Grid Search method. In a grid search, we pass the list of values that we think are good candidates for the hyperparameters. The grid search algorithm then runs our model with all combinations of given hyperparameters and store all the results. The algorithm also stores the

values that gave the best result separately as the best model (estimator).

Finding available hyperparameters

The list of hyperparameters for your model can be found in the Scikit-learn's documentation page. A simple google search of "your model name + scikit-learn" will take you to the correct documentation page. For our simple logistic model, you can search logistic classifier scikit-learn and it will take you to this page. In the model documentation page, all arguments/variables listed under the parameters section are hyperparameters of the model that you can play around with.

Once we identify the hyperparameters we are interested in and the values we want to check, we call the `GridSearchCV` function from the `GridSearch` module in `scikit-learn`.

```
using ScikitLearn.GridSearch: GridSearchCV
@sk_import linear_model: LogisticRegression;
gridsearch = GridSearchCV(LogisticRegression(),
            Dict(:solver => ["newton-cg", "lbfgs", "liblinear"],
            :C => [0.01, 0.1, 0.5, 0.9]));
```

- The hyperparameters and their values are passed as a dictionary.
- `:solver` corresponds to the different learning algorithms and `:C` hyperparameter is a regularization constant.

Once we have initialized the grid search model object with the values we are interested in, we can call the `fit!` function to start the training process.

```
fit!(gridsearch, features_train, target_train);
```

The results of the grid search are stored in the `grid_scores_` field in `gridsearch` model object.

```
search_results = DataFrame(gridsearch.grid_scores_)
hcat(DataFrame(search_results.parameters),
            search_results)[!,Not(:parameters)]
```

	solver	C	mean_validation_score	cv_validation_scores
	String	Float64	Float64	Array…
1	newton-cg	0.01	0.806034	[0.782051, 0.844156, 0.792208]
2	lbfgs	0.01	0.806034	[0.782051, 0.844156, 0.792208]
3	liblinear	0.01	0.75	[0.730769, 0.766234, 0.753247]
4	newton-cg	0.1	0.814655	[0.807692, 0.831169, 0.805195]
5	lbfgs	0.1	0.814655	[0.807692, 0.831169, 0.805195]
6	liblinear	0.1	0.767241	[0.717949, 0.779221, 0.805195]
7	newton-cg	0.5	0.814655	[0.807692, 0.831169, 0.805195]
8	lbfgs	0.5	0.814655	[0.807692, 0.831169, 0.805195]
9	liblinear	0.5	0.775862	[0.730769, 0.805195, 0.792208]
10	newton-cg	0.9	0.810345	[0.807692, 0.818182, 0.805195]
11	lbfgs	0.9	0.810345	[0.807692, 0.818182, 0.805195]
12	liblinear	0.9	0.784483	[0.75641, 0.805195, 0.792208]

The best model is stored in the `best_estimator_` field in `gridsearch` model object.

```
best_model = gridsearch.best_estimator_
```

```
PyObject LogisticRegression(C=0.1, solver='newton-cg')
```

We can now use the `best_model` object the way we used `simplelogisitc` for predictions and other stuffs.

```
best_model_predictions = predict(best_model, features_train);
first(best_model_predictions,4)
```

```
4-element Vector{Any}:
 "No"
 "Yes"
 "No"
 "No"
```