Machine Learning Fall 2023

Lecture notes for Fall 2023 at Arkansas Tech University

Xinli Xiao

2023-09-05

Table of contents

Preface

This is the lecture notes for STAT 4803/5803 Machine Learning Fall 2023 at ATU. If you have any comments/suggetions/concers about the notes please contact us at xxiao@atu.edu.

References

[1] Géron, A. (2019). *Hands-on machine learning with scikit-learn, keras, and TensorFlow concepts, tools, and techniques to build intelligent systems: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media.

[2] Chollet, F. (2021). *Deep learning with python, second edition*. MANNING PUBN.

[3] Harrington, P. (2012). *Machine learning in action*. Manning Publications.

[4] Klosterman, S. (2021). *Data science projects with python: A case study approach to gaining valuable insights from real data with machine learning*. Packt Publishing, Limited.

1 Introduction

In this Chapter we will discuss

- What is Machine Learning?
- What do typical Machine Learning problems look like?
- What is the basic structure of Machine Learning models?
- What is the basic work flow to use Machine Learning to solve problems?
- Some supplementary materials, such as Linear Algebra and Python.

1.1 What is Machine Learning?

Machine Learning is the science (and art) of programming computers so they can *learn from data* [1].

Here is a slightly more general definition:

> [Machine Learning is the] field of study that gives computers the ability to learn without being explicitly programmed.

`-- Arthur Samuel, 1959`

This "without being explicitly programmed to do so" is the essential difference between Machine Learning and usual computing tasks. The usual way to make a computer do useful work is to have a human programmer write down rules — a computer program — to be followed to turn input data into appropriate answers. Machine Learning turns this around: the machine looks at the input data and the expected task outcome, and figures out what the rules should be. A Machine Learning system is *trained* rather than explicitly programmed. It's presented with many examples relevant to a task, and it finds statistical structure in these examples that eventually allows the system to come up with rules for automating the task [2].

1.2 Types of Machine Learning Systems

There are many different types of Machine Learning systems that it is useful to classify them in braod categories, based on different criteria. These criteria are not exclusive, and you can combine them in any way you like.

The most popular criterion for Machine Learning classification is the amount and type of supervision they get during training. In this case there are four major types.

Supervised Learning The training set you feed to the algorithm includes the desired solutions. The machines learn from the data to alter the model to get the desired output. The main task for Supervised Learning is classification and regression.

Unsupervised Learning In Unsupervised Learning, the data provided doesn't have class information or desired solutions. We just want to dig some information directly from those data themselves. Usually Unsupervised Learning is used for clustering and dimension reduction.

Reinforcement Learning In Reinforcement Learning, there is a reward system to measure how well the machine performs the task, and the machine is learning to find the strategy to maximize the rewards. Typical examples here include gaming AI and walking robots.

Semisupervised Learning This is actually a combination of Supervised Learning and Unsupervised Learning, that it is usually used to deal with data that are half labelled.

1.2.1 Tasks for Supervised Learning

As mentioned above, for Supervised Learning, there are two typical types of tasks:

Classification It is the task of predicting a discrete class labels. A typical classification problem is to see an handwritten digit image and recognize it.

Regression It is the task of predicting a continuous quantity. A typical regression problem is to predict the house price based on various features of the house.

There are a lot of other tasks that are not directly covered by these two, but these two are the most classical Supervised Learning tasks.

> ℹ Note
>
> In this course we will mainly focus on **Supervised Classification problems**.

1.2.2 Classification based on complexity

Along with the popularity boost of deep neural network, there comes another classificaiton: shallow learning vs. deep learning. Basically all but deep neural network belongs to shallow learning. Although deep learning can do a lot of fancy stuffs, shallow learning is still very good in many cases. When the performance of a shallow learning model is good enough comparing to that of a deep learning model, people tend to use the shallow learning since it is usually faster, easier to understand and easier to modify.

1.3 Basic setting for Machine learning problems

> ℹ Note
>
> We by default assume that we are dealing with a **Supervised Classification** problem.

1.3.1 Input and output data structure

Since we are dealing with Supervised Classification problems, the desired solutions are given. These desired solutions in Classification problems are also called *labels*. The properties that the data are used to describe are called *features*. Both features and labels are usually organized as row vectors.

Example 1.1. The example is extracted from [3]. There are some sample data shown in the following table. We would like to use these information to classify bird species.

Table 1.1: Bird species classification based on four features

Weight (g)	Wingspan (cm)	Webbed feet?	Back color	Species
1000.100000	125.000000	No	Brown	Buteo jamaicensis
3000.700000	200.000000	No	Gray	Sagittarius serpentarius
3300.000000	220.300000	No	Gray	Sagittarius serpentarius
4100.000000	136.000000	Yes	Black	Gavia immer
3.000000	11.000000	No	Green	Calothorax lucifer
570.000000	75.000000	No	Black	Campephilus principalis

The first four columns are features, and the last column is the label. The first two features are numeric and can take on decimal values. The third feature is binary that can only be 1 (Yes) or 0 (No). The fourth feature is an enumeration over the color palette. You may either treat it as categorical data or numeric data, depending on how you want to build the model and what you want to get out of the data. In this example we will use it as categorical data that we only choose it from a list of colors (1 — Brown, 2 — Gray, 3 — Black, 4 — Green).

Then we are able to transform the above data into the following form:

Table 1.2: Vectorized Bird species data

Features	Labels
$\begin{bmatrix}1001.1 & 125.0 & 0 & 1\end{bmatrix}$	1
$\begin{bmatrix}3000.7 & 200.0 & 0 & 2\end{bmatrix}$	2

Features	Labels
$\begin{bmatrix}3300.0 & 220.3 & 0 & 2\end{bmatrix}$	2
$\begin{bmatrix}4100.0 & 136.0 & 1 & 3\end{bmatrix}$	3
$\begin{bmatrix}3.0 & 11.0 & 0 & 4\end{bmatrix}$	4
$\begin{bmatrix}570.0 & 75.0 & 0 & 3\end{bmatrix}$	5

Then the Supervised Learning problem is stated as follows: Given the features and the labels, we would like to find a model that can classify future data.

1.3.2 Parameters and hyperparameters

A model parameter is internal to the model and its value is learned from the data.

A model hyperparameter is external to the model and its value is set by people.

For example, assume that we would like to use Logistic regression to fit the data. We set the learning rate is `0.1` and the maximal iteration is `100`. After the computations are done, we get a the model

$$y = \sigma(0.8 + 0.7x).$$

The two cofficients 0.8 and 0.7 are the parameters of the model. The model `Logistic regression`, the learning rate `0.1` and the maximal iteration `100` are all hyperparametrs. If we change to a different set of hyperparameters, we may get a different model, with a different set of parameters.

The details of Logistic regression will be discussed later.

1.3.3 Evaluate a Machine Learning model

Once the model is built, how do we know that it is good or not? The naive idea is to test the model on some brand new data and check whether it is able to get the desired results. The usual way to achieve it is to split the input dataset into three pieces: *training set*, *validation set* and *test set*.

The model is initially fit on the training set, with some arbitrary selections of hyperparameters. Then hyperparameters will be changed, and new model is fitted over the training set. Which set of hyperparameters is better? We then test their performance over the validation set. We could run through a lot of different combinations of hyperparameters, and find the best performance over the validation set. After we get the best hyperparameters, the model is selcted, and we fit it over the training set to get our model to use.

To compare our model with our models, either our own model using other algorithms, or models built by others, we need some new data. We can no longer use the training set and the validation set since all data in them are used, either for training or for hyperparameters tuning. We need to use the test set to evaluate the "real performance" of our data.

To summarize:

- Training set: used to fit the model;
- Validation set: used to tune the hyperparameters;
- Test set: used to check the overall performance of the model.

The validation set is not always required. If we use cross-validation technique for hyperparameters tuning, like `sklearn.model_selection.GridSearchCV()`, we don't need a separated validation set. In this case, we will only need the training set and the test set, and run `GridSearchCV` over the training set. The cross-validation will be discussed in {numref}`Section %s<section-cross-validation>`.

The sizes and strategies for dataset division depends on the problem and data available. It is often recommanded that more training data should be used. The typical distribution of training, validation and test is $(6:3:1)$, $(7:2:1)$ or $(8:1:1)$. Sometimes validation set is discarded and only training set and test set are used. In this case the distribution of training and test set is usually $(7:3)$, $(8:2)$ or $(9:1)$.

1.3.4 Workflow in developing a machine learning application

The workflow described below is from [3].

1. Collect data.
2. Prepare the input data.
3. Analyze the input data.
4. Train the algorithm.
5. Test the algorithm.
6. Use it.

In this course, we will mainly focus on Step 4 as well Step 5. These two steps are where the "core" algorithms lie, depending on the algorithm. We will start from the next Chapter to talk about various Machine Learning algorithms and examples.

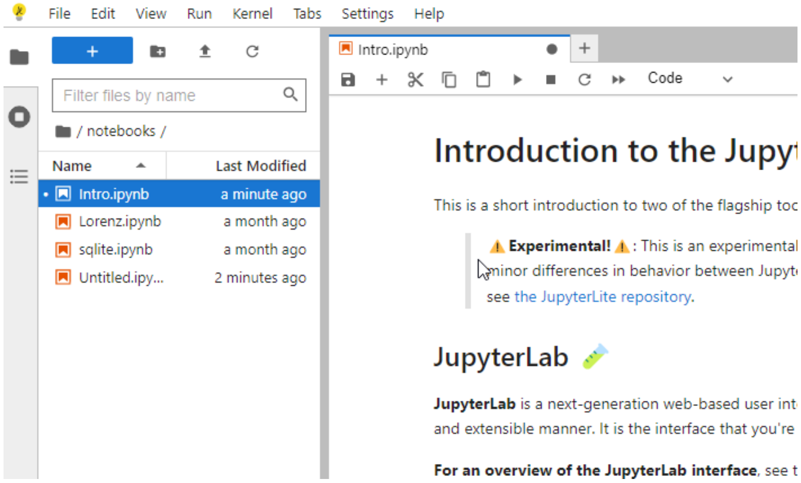
1.4 Python quick guide

1.4.1 Python Notebook

We mainly use Python Notebook (.ipynb) to write documents for this course. Currently all main stream Python IDE support Python Notebook. All of them are not entirely identical but the differences are not huge and you may choose any you like.

One of the easiest ways to use Python Notebook is through JupyterLab. The best part about it is that you don't need to worry about installation and configuration in the first place, and you can directly start to code.

Click the above link and choose JupyterLab. Then you will see the following page.



Name	Last Modified
Intro.ipynb	a minute ago
Lorenz.ipynb	a month ago
sqlite.ipynb	a month ago
Untitled.ipy...	2 minutes ago

Introduction to the Jupy

This is a short introduction to two of the flagship toc

> ⚠ **Experimental!** ⚠ : This is an experimental
> minor differences in behavior between Jupyt
> see the JupyterLite repository.

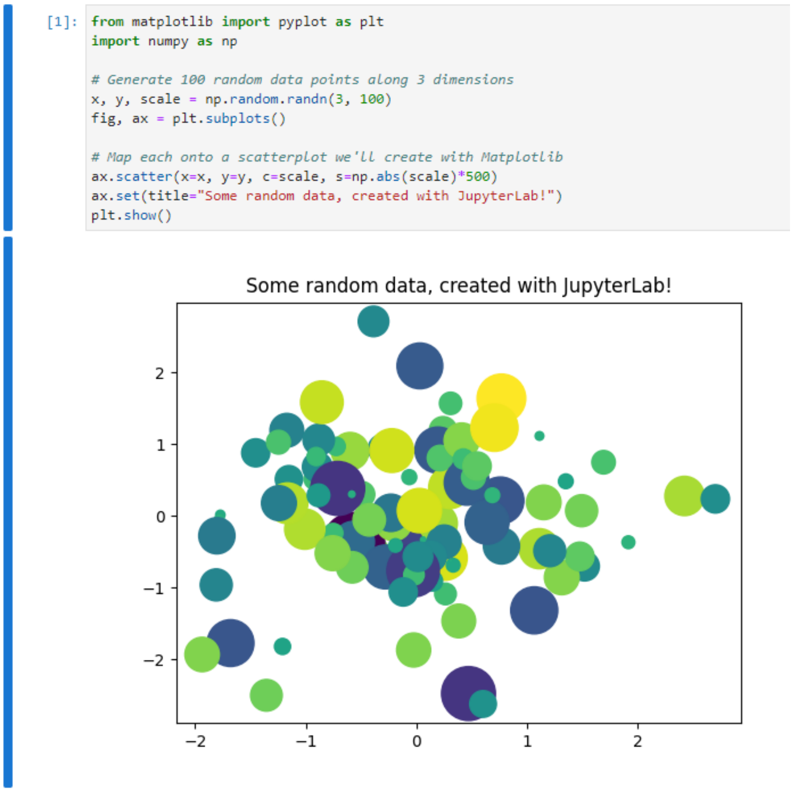
JupyterLab

JupyterLab is a next-generation web-based user int
and extensible manner. It is the interface that you're

For an overview of the JupyterLab interface, see t

The webapp you just started is called JupyterLite. This is a demo version. The full JupyterLab installation instruction can also be found from the link.

There is a small button `+` under the tab bar. This is the place where you click to start a new cell. You may type codes or markdown documents or raw texts in the cell according to your needs. The drag-down menu at the end of the row which is named `Code` or `Markdown` or `Raw` can help you make the switch. Markdown is a very simple light wighted language to write documents. In most cases it behaves very similar to plain texts. Codes are just regular Python codes (while some other languages are supported). You may either use the triangle button in the menu to execute the codes, or hit `shift + enter`.



```
from matplotlib import pyplot as plt
import numpy as np

# Generate 100 random data points along 3 dimensions
x, y, scale = np.random.randn(3, 100)
fig, ax = plt.subplots()

# Map each onto a scatterplot we'll create with Matplotlib
ax.scatter(x=x, y=y, c=scale, s=np.abs(scale)*500)
ax.set(title="Some random data, created with JupyterLab!")
plt.show()
```

JupyterLite contains a few popular packages. Therefore it is totally ok if you would like to play with some simple things. However since it is an online evironment, it has many limitations. Therefore it is still recommended to set up a local environment once you get familiar with Python Notebook. Please check the following links for some popular choices for notebooks and Python installations in general, either local and online.

- Jupyter Notebook / JupyterLab
- VS Code
- PyCharm
- Google Colab
- Anaconda

1.4.2 Python fundamentals

We will put some very basic Python commands here for you to warm up. More advanced Python knowledge will be covered during the rest of the semester. The main reference for this part is [3]. Another referenece is My notes.

1.4.2.1 Indentation

Python is using indentation to denote code blocks. It is not convienent to write in the first place, but it forces you to write clean, readable code.

By the way, the `if` and `for` block are actually straightforward.

```
if jj < 3:
    jj = jj
    print("It is smaller t
```

```
for i in range(3):
    i = i + 1
    print(i)
```

```
if jj < 3:
    jj = jj
print("It is smaller than 3.")
```

```
for i in range(3):
    i = i + 1
print(i)
```

Please tell the differences between the above codes.

1.4.2.2 `list` and `dict`

Here are some very basic usage of lists of dictionaries in Python.

```
newlist = list()
newlist.append(1)
newlist.append('hello')
newlist
```

```
[1, 'hello']
```

```
newlisttwo = [1, 'hello']
newlisttwo
```

```
[1, 'hello']
```

```
newdict = dict()
newdict['one'] = 'good'
newdict[1] = 'yes'
newdict
```

```
{'one': 'good', 1: 'yes'}
```

```
newdicttwo = {'one': 'good', 1: 'yes'}
newdicttwo
```

```
{'one': 'good', 1: 'yes'}
```

1.4.2.3 Loop through lists

When creating `for` loops we may let Python directly loop through lists. Here is an example. The code is almost self-explained.

```
alist = ['one', 2, 'three', 4]

for item in alist:
    print(item)
```

```
one
2
three
4
```

1.4.2.4 Reading files

There are a lot of functions that can read files. The basic one is to read any files as a big string. After we get the string, we may parse it based on the structure of the data.

The above process sounds complicated. That's why we have so many different functions reading files. Usually they focus on a certain types of files (e.g. spreadsheets, images, etc..), parse the data into a particular data structure for us to use later.

I will mention a few examples.

- `csv` files and `excel` files Both of them are spreadsheets format. Usually we use `pandas.read_csv` and `pandas.read_excel` both of which are from the package `pandas` to read these two types of files.
- images Images can be treated as matrices, that each entry represents one pixel. If the image is black/white, it is represented by one matrix where each entry represents the gray value. If the image is colored, it is represented by three matrices where each entry represents one color. To use which three colors depends on the color map. `rgb` is a popular choice.

  In this course when we need to read images, we usually use `matplotlib.pyplot.imread` from the package `matplotlib` or `cv.imread` from the package `opencv`.
- `.json` files `.json` is a file format to store dictionary type of data. To read a `json` file and parse it as a dictionary, we need `json.load` from the package `json`.

1.4.2.5 Writing files

- `pandas.DataFrame.to_csv`
- `pandas.DataFrame.to_excel`
- `matplotlib.pyplot.imsave`
- `cv.imwrite`
- `json.dump`

1.4.2.6 Relative paths

In this course, when reading and writing files, please keep all the files using relative paths. That is, only write the path starting from the working directory.

Example 1.2. Consider the following tasks:

1. Your working directory is `C:/Users/Xinli/projects/`.
2. Want to read a file `D:/Files/example.csv`.
3. Want to generate a file whose name is `result.csv` and put it in a subfoler named `foldername`.

To do the tasks, don't directly run the code `pd.read_csv('D:/Files/example.csv')`. Instead you should first copy the file to your working directory `C:/Users/Xinli/projects/`, and then run the following code.

```
import pandas as pd

df = pd.read_csv('example.csv')
df.to_csv('foldername/result.csv')
```

Please pay attention to how the paths are written.

1.4.2.7 .

- class and packages.
- Get access to attributes and methods
- Chaining dots.

1.4.3 Some additional topics

You may read about these parts from the appendices of My notes.

1.4.3.1 Package management and Virtual environment

- `conda`
  - `conda create`
    - `conda create --name myenv`
    - `conda create --name myenv python=3.9`
    - `conda create --name myenv --file spec-file.txt`
  - `conda install`
    - `conda install -c conda-forge numpy`
  - `conda activate myenv`

    - `conda list`
        - `conda list numpy`
        - `conda list --explicit > spec-file.txt`
    - `conda env list`
- `pip` / `venv`
    - `python -m venv newenv`
    - `newenv\Scripts\activate`
    - `pip install`
    - `pip freeze > requirements.txt`
    - `pip install -r /path/to/requirements.txt`
    - `deactivate`

1.4.3.2 Version Control

- `Git`
    - Install
    - `git config --list`
    - `git config --global user.name "My Name"`
    - `git config --global user.email "myemail@example.com"`
- `GitHub`

1.5 Exercises

These exercises are from [4], [1] and [3].

1.5.1 Python Notebook

Exercise 1.1.

Hello World!

Please set up a Python Notebook environment and type `print('Hello World!')`.

Exercise 1.2. Please set up a Python Notebook and start a new virtual environment and type `print('Hello World!')`.

1.5.2 Basic Python

Exercise 1.3.

Play with lists

Please complete the following tasks.

- Write a `for` loop to print values from 0 to 4.
- Combine two lists `['apple', 'orange']` and `['banana']` using `+`.
- Sort the list `['apple', 'orange', 'banana']` using `sorted()`.

Exercise 1.4.

Play with list, dict and pandas.

Please complete the following tasks.

- Create a new dictionary people with two keys name and age. The values are all empty list.
- Add Tony to the name list in people.
- Add Harry to the name list in people.
- Add number 100 to the age list in people.
- Add number 10 to the age list in people.
- Find all the keys of people and save them into a list namelist.
- Convert the dictionary people to a Pandas DataFrame df.

Exercise 1.5.

The dataset iris

```python
from sklearn.datasets import load_iris
iris = load_iris()
```

Please explore this dataset.

- Please get the features for `iris` and save it into `X` as an numpy array.
- What is the meaning of these features?
- Please get the labels for `iris` and save it into `y` as an numpy array.
- What is the meaning of labels?

Exercise 1.6.

Play with Pandas

Please download the Titanic data file from here. Then follow the instructions to perform the required tasks.

- Use `pandas.read_csv` to read the dataset and save it as a dataframe object `df`.
- Change the values of the `Sex` column that `male` is `0` and `female` is `1`.
- Pick the columns `Pclass`, `Sex`, `Age`, `Siblings/Spouses Aboard`, `Parents/Children Aboard` and `Fare` and transform them into a 2-dimensional `numpy.ndarray`, and save it as `X`.
- Pick the column `Survived` and transform it into a 1-dimensional `numpy.ndarray` and save it as `y`.

2 k-Nearest Neighbors algorithm (k-NN)

This algorithm is different from other algorithms covered in this course, that it doesn't really extract features from the data. However, since its idea is easy to understand, we use it as our first step towards machine learning world.

Similar to other algorithms, we will only cover the beginning part of the algorithm. All later upgrades of the algorithms are left for yourselves to learn.

References: [3].

2.1 k-Nearest Neighbors Algorithm (k-NN)

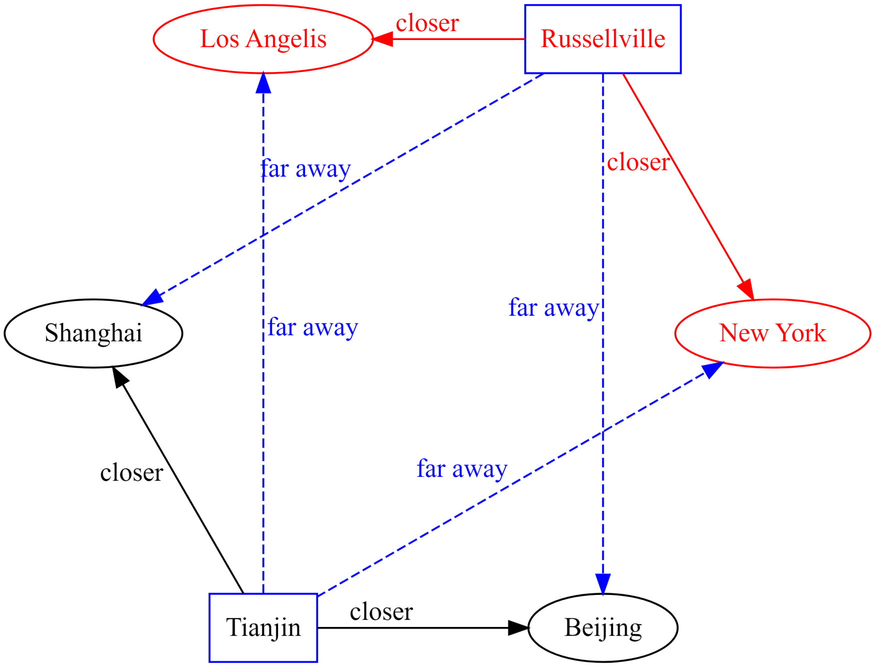
2.1.1 Ideas

Assume that we have a set of labeled data $\{(X_i, y_i)\}$ where y_i denotes the label. Given a new data X, how do we determine the label of it?

k-NN algorithm starts from a very straightforward idea. We use the distances from the new data point X to the known data points to identify the label. If X is closer to y_i points, then we will label X as y_i.

Let us take cities and countries as an example. New York and Los Angeles are U.S cities, and Beijing and Shanghai are Chinese cities. Now we would like to consider Tianjin and Russellville. Do they belong to China or U.S? We calculate the distances from Tianjin (resp. Russellville) to all four known cities. Since Tianjin is closer to Beijing and Shanghai comparing to New York and Los Angeles, we classify Tianjin as a Chinese city. Similarly, since Russellville is closer to New

York and Los Angeles comparing to Beijing and Shanghai, we classify it as a U.S. city.



This naive example explains the idea of k-NN. Here is a more detailed description of the algorithm.

2.1.2 The Algorithm

k-NN Classifier

Inputs: Given the training data set $\{(X_i, y_i)\}$ where $X_i = (x_i^1, x_i^2, \ldots, x_i^n)$ represents n features and y_i represents labels. Given a new data point $\tilde{X} = (\tilde{x}^1, \tilde{x}^2, \ldots, \tilde{x}^n)$.
Outputs: Want to find the best label for $\tilde{X}$.

1. Compute the distance from $\tilde{X}$ to each X_i.
2. Sort all these distances from the nearest to the furthest.
3. Find the nearest k data points.
4. Determine the labels for each of these k nearest

points, and compute the frenqucy of each labels.
5. The most frequent label is considered to be the label of $\tilde{X}$.

2.1.3 Details

- The distance between two data points are defined by the Euclidean distance:

$$dist\left((x_i^j)_{j=1}^n, (\tilde{x}^j)_{j=1}^n\right) = \sqrt{\sum_{j=1}^n (x_i^j - \tilde{x}^j)^2}.$$

- Using linear algebra notations:

$$dist(X_i, \tilde{X}) = \sqrt{(X_i - \tilde{X}) \cdot (X_i - \tilde{X})}.$$

- All the distances are stored in a 1-dim numpy array, and we will combine it together with another 1-dim array that store the labels of each point.

2.1.4 The codes

- `argsort`
- `get`
- `sorted`

```
def classify_kNN(inX, X, y, k):
    # create a new 2-d numpy array by copying inX for each row.
    Xmat = np.tile(np.array([inX]), (X.shape[0], 1))
    # compute the distance between each row of X and Xmat
    Dmat = np.sqrt(np.sum((Xmat - X)**2, axis=1))
    # sort by distance
    sortedlist = Dmat.argsort()
    # count the freq. of the first k items
    k = min(k, len(sortedlist))
    classCount = dict()
    for i in sortedlist[:k]:
        classCount[y[i]] = classCount.get(y[i], 0) + 1
    # find out the most freqent one
    sortedCount = sorted(classCount.items(), key=lambda x:x[1],
                         reverse=True)
    return sortedCount[0][0]
```

2.1.5 `sklearn` packages

You may also directly use the kNN function `KNeighborsClassifier` packaged in `sklearn.neighbors`. You may check the description of the function online from here.

There are many ways to modify the kNN algorithm. What we just mentioned is the simplest idea. It is correspondent to the argument `weights='uniform'`, `algorithm='brute` and `metric='euclidean'`. However due to the implementation details, the results we got from `sklearn` are still a little bit different from the results produced by our naive codes.

```
from sklearn.neighbors import KNeighborsClassifier
clf = KNeighborsClassifier(n_neighbors=10, weights='uniform',
                           algorithm='brute', metric='euclidean')
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

2.1.6 Normalization

Different features may have different scales. It might be unfair for those features that have small scales. Therefore usually it is better to rescale all the features to make them have similar scales. After examining all the data, we find the minimal value `minVal` and the range `ranges` for each column. The normalization formula is:

$$X_{norm} = \frac{X_{original} - minVal}{ranges}.$$

We could also convert the normalized number back to the original value by

$$X_{original} = X_{norm} \times ranges + minVal.$$

The sample codes are listed below.

```
def encodeNorm(X, parameters=None):
    # parameters contains minVals and ranges
    if parameters is None:
        minVals = np.min(X, axis=0)
        maxVals = np.max(X, axis=0)
        ranges = np.maximum(maxVals - minVals, np.ones(minVals.size))
        parameters = {'ranges': ranges, 'minVals': minVals}
    else:
        minVals = parameters['minVals']
        ranges = parameters['ranges']
    Nmat = np.tile(minVals, (X.shape[0], 1))
    Xnorm = (X - Nmat)/ranges
    return (Xnorm, parameters)


def decodeNorm(X, parameters):
    # parameters contains minVals and ranges
    ranges = parameters['ranges']
    minVals = parameters['minVals']
    Nmat = np.tile(minVals, (X.shape[0], 1))
    Xoriginal = X * ranges + Nmat
    return Xoriginal
```

If you use `sklearn` you could use `MinMaxScaler` from `sklearn.preprocessing` to achive the same goal. The related codes will be discussed later in projects. I keep our handwritten codes here for Python practicing.

2.2 k-NN Project 1: `iris` Classification

This data is from `sklearn.datasets`. This dataset consists of 3 different types of irises' petal / sepal length / width, stored in a 150×4 `numpy.ndarray`. We already explored the dataset briefly in the previous chapter. This time we will try to use the feature provided to predict the type of the irises. For the purpose of plotting, we will only use the first two features: sepal length and sepal width.

2.2.1 Explore the dataset

We first load the dataset.

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data[:, :2]
y = iris.target
```

Then we would like to split the dataset into trainning data and test data. Here we are going to use sklearn.model_selection.train_test_split function. Besides the dataset, we should also provide the propotion of the test set comparing to the whole dataset. We will choose test_size=0.1 here, which means that the size of the test set is 0.1 times the size of the whole dataset. stratify=y means that when split the dataset we want to split respects the distribution of labels in y.

The split will be randomly. You may set the argument random_state to be a certain number to control the random process. If you set a random_state, the result of the random process will stay the same. This is for reproducible output across multiple function calls.

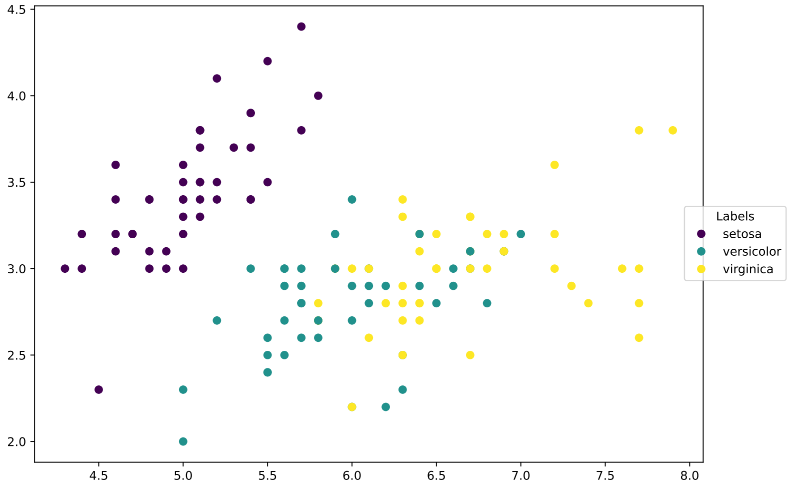
After we get the training set, we should also normalize it. All our normalization should be based on the training set. When we want to use our model on some new data points, we will use the same normalization parameters to normalize the data points in interests right before we apply the model. Here since we mainly care about the test set, we could normalize the test set at this stage.

Note that in the following code, the function encodeNorm defined in the previous section is used. I import it from assests.codes.knn. You need to modify this part based on your file structure. See Section 2.2.2 for more details.

```
from sklearn.model_selection import train_test_split
from assests.codes.knn import encodeNorm
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=1, strat

X_train_norm, parameters = encodeNorm(X_train)
X_test_norm, _ = encodeNorm(X_test, parameters=parameters)
```

Before we start to play with k-NN, let us look at the data first. Since we only choose two features, it is able to plot these data points on a 2D plane, with different colors representing different classes.

```
import matplotlib.pyplot as plt
import numpy as np

# Plot the scatter plot.
fig = plt.figure(figsize=(10,7))
ax = fig.add_subplot(111)
scatter = ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train)

# Generate legends.
labels = ['setosa', 'versicolor', 'virginica']
_ = fig.legend(handles=scatter.legend_elements()[0], labels=labels,
               loc="right", title="Labels")
```



2.2.2 Apply our k-NN model

Now let us apply k-NN to this dataset. We first use our codes. Here I use `from assests.codes.knn` to import our functions since I put all our functions in `./assests/codes/knn.py`. Then the poential code is

```
y_pred = classify_kNN(X_test, X_train, y_train, k=10)
```

Note that the above code is actually wrong. The issue ist that our function `classify_kNN` can only classify one row of data. To classify many rows, we need to use a `for` loop.

```
from assests.codes.knn import classify_kNN

n_neighbors = 10
y_pred = list()
for row in X_test_norm:
    row_pred = classify_kNN(row, X_train_norm, y_train, k=n_neighbors)
    y_pred.append(row_pred)
y_pred = np.array(y_pred)
```

We could use list comprehension to simply the above codes.

```
from assests.codes.knn import classify_kNN

n_neighbors = 10
y_pred = np.array([classify_kNN(row, X_train_norm, y_train, k=n_neighbors)
                   for row in X_test_norm])
```

This `y_pred` is the result we got for the test set. We may compare it with the real answer `y_test`, and calcuate the accuracy.

```
acc = np.mean(y_pred == y_test)
acc
```

```
0.7333333333333333
```

2.2.3 Apply k-NN model from `sklearn`

Now we would like to use `sklearn` to reproduce this result. Since our data is prepared, what we need to do is directly call the functions.

```
from sklearn.neighbors import KNeighborsClassifier
n_neighbors = 10
clf = KNeighborsClassifier(n_neighbors, weights="uniform", metric="euclidean",
                           algorithm='brute')
clf.fit(X_train_norm, y_train)
y_pred_sk = clf.predict(X_test_norm)

acc = np.mean(y_pred_sk == y_test)
acc
```

```
0.7333333333333333
```

2.2.4 Using data pipeline

We may organize the above process in a neater way. After we get a data, the usual process is to apply several transforms to the data before we really get to the model part. Using terminolgies from `sklearn`, the former are called *transforms*, and the latter is called an *estimator*. In this example, we have exactly one tranform which is the normalization. The estimator here we use is the k-NN classifier.

`sklearn` provides a standard way to write these codes, which is called `pipeline`. We may chain the transforms and estimators in a sequence and let the data go through the pipeline. In this example, the pipeline contains two steps: 1. The normalization transform `sklearn.preprocessing.MinMaxScaler`. When we directly apply it the parameters `ranges` and `minVals` and will be recorded automatically, and we don't need to worry about it when we want to use the same parameters to normalize other data. 2. The k-NN classifier `sklearn.neighbors.KNeighborsClassifier`. This is the same one as we use previously.

The code is as follows. It is a straightforward code. Note that the `()` after the class in each step of `steps` is very important. The codes cannot run if you miss it.

After we setup the pipeline, we may use it as other estimators since it is an estimator. Here we may also use the accuracy function provided by `sklearn` to perform the computation. It is essentially the same as our `acc` computation.

```python
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import accuracy_score

steps = [('scaler', MinMaxScaler()),
         ('knn', KNeighborsClassifier(n_neighbors, weights="uniform",
                                      metric="euclidean", algorithm='brute'))]
pipe = Pipeline(steps=steps)
pipe.fit(X_train, y_train)
y_pipe = pipe.predict(X_test)
accuracy_score(y_pipe, y_test)
```

```
0.7333333333333333
```

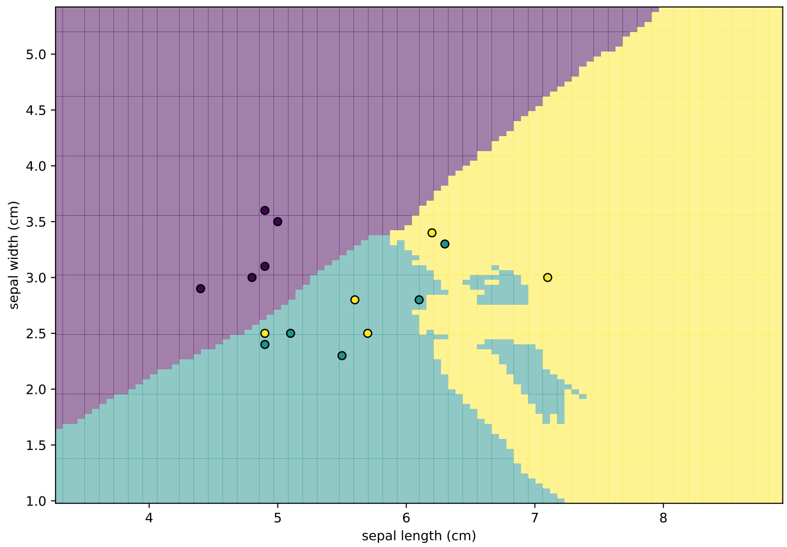
2.2.5 Visualize the Decision boundary [Optional]

Using the classifier we get above, we are able to classify every points on the plane. This enables us to draw the following plot, which is called the Decision boundary. It helps us to visualize the relations between features and the classes.

We use `DecisionBoundaryDisplay` from `sklearn.inspection` to plot the decision boundary. The function requires us to have a fitted classifier. We may use the classifier `pipe` we got above. Note that this classifier should have some build-in structures that our `classify_kNN` function doesn't have. We may rewrite our codes to make it work, but this goes out of the scope of this section. This is supposed to be Python programming exercise. We will talk about it in the future if we have enough time.

We first plot the dicision boundary using `DecisionBoundaryDisplay.from_estimator`. Then we plot the points from `X_test`. From the plot it is very clear which points are misclassified.

```python
from sklearn.inspection import DecisionBoundaryDisplay

disp = DecisionBoundaryDisplay.from_estimator(
            pipe,
            X_train,
            response_method="predict",
            plot_method="pcolormesh",
            xlabel=iris.feature_names[0],
            ylabel=iris.feature_names[1],
            alpha=0.5)
disp.ax_.scatter(X_test[:, 0], X_test[:, 1], c=y_test, edgecolor="k")
disp.figure_.set_size_inches((10,7))
```



2.2.6 k-Fold Cross-Validation

Previously we perform a random split and test our model in this case. What would happen if we fit our model on another split? We might get a different accuracy score. So in order to evaluate the performance of our model, it is natual to consider several different split and compute the accuracy socre for each case, and combine all these socres together to generate an index to indicate whehter our model is good or bad. This naive idea is called *k-Fold Cross-Validation*.

The algorithm is described as follows. We first randomly split the dataset into k groups. We use one of them as the test set, and the rest together forming the training set, and use this setting to get an accuracy score. We did this for each group to be chosen as the test set. Then the final score is the mean.

sklearn provides a function sklearn.model_selection.cross_val_score to perform the above computation. The usage is straightforward, as follows.

```
from sklearn.model_selection import cross_val_score
cv_scores = cross_val_score(pipe, X, y, cv=5)
cv_scores
```

```
array([0.66666667, 0.8       , 0.63333333, 0.8       , 0.7       ])
```

```
np.mean(cv_scores)
```

```
0.7200000000000001
```

2.2.7 Choosing a k value

In the previous example we choose k to be 10 as an example. To choose a k value we usually run some test by trying different k and choose the one with the best performance. In this case, best performance means the highest cross-validation score.

sklearn.model_selection.GridSearchCV provides a way to do this directly. We only need to setup the esitimator, the metric (which is the cross-validation score in this case), and the hyperparameters to be searched through, and GridSearchCV will run the search automatically.

We let k go from 1 to 100. The code is as follows.

Note that parameters is where we set the search space. It is a dictionary. The key is the name of the estimator plus double _ and then plus the name of the parameter.

```
from sklearn.model_selection import GridSearchCV
n_list = list(range(1, 101))
parameters = dict(knn__n_neighbors=n_list)
clf = GridSearchCV(pipe, parameters)
clf.fit(X, y)
clf.best_estimator_.get_params()["knn__n_neighbors"]
```

```
35
```

After we fit the data, the `best_estimator_.get_params()` can be printed. It tells us that it is best to use `31` neibhours for our model. We can directly use the best estimator by calling `clf.best_estimator_`.

```
cv_scores = cross_val_score(clf.best_estimator_, X, y, cv=5)
np.mean(cv_scores)
```

```
0.82
```

The cross-validation score using `k=31` is calculated. This serves as a benchmark score and we may come back to dataset using other methods and compare the scores.

2.3 k-NN Project 2: Dating Classification

The data can be downloaded from here.

2.3.1 Background

Helen dated several people and rated them using a three-point scale: 3 is best and 1 is worst. She also collected data from all her dates and recorded them in the file attached. These data contains 3 features:

- Number of frequent flyer miles earned per year
- Percentage of time spent playing video games
- Liters of ice cream consumed per week

We would like to predict her ratings of new dates when we are given the three features.

The data contains four columns, while the first column refers to `Mileage`, the second `Gamingtime`, the third `Icecream` and the fourth `Rating`.

2.3.2 Look at Data

We first load the data and store it into a DataFrame.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('./assests/datasets/datingTestSet2.txt', sep='\t', header=None)
df.head()
```

	0	1	2	3
0	40920	8.326976	0.953952	3
1	14488	7.153469	1.673904	2
2	26052	1.441871	0.805124	1
3	75136	13.147394	0.428964	1
4	38344	1.669788	0.134296	1

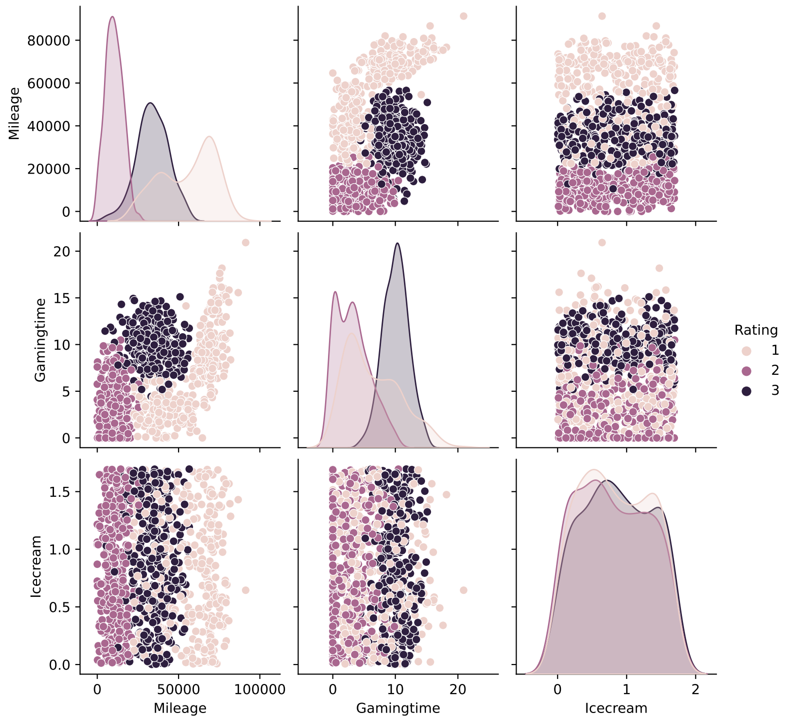
To make it easier to read, we would like to change the name of the columns.

```python
df = df.rename(columns={0: "Mileage", 1: "Gamingtime", 2: 'Icecream', 3: 'Rating'})
df.head()
```

	Mileage	Gamingtime	Icecream	Rating
0	40920	8.326976	0.953952	3
1	14488	7.153469	1.673904	2
2	26052	1.441871	0.805124	1
3	75136	13.147394	0.428964	1
4	38344	1.669788	0.134296	1

Since now we have more than 2 features, it is not suitable to directly draw scatter plots. We use `seaborn.pairplot` to look at the pairplot. From the below plots, before we apply any tricks, it seems that `Milegae` and `Gamingtime` are better than `Icecream` to classify the data points.

```python
import seaborn as sns
sns.pairplot(data=df, hue='Rating')
```



2.3.3 Applying kNN

Similar to the previous example, we will apply both methods for comparisons.

```python
from sklearn.model_selection import train_test_split
from assests.codes.knn import encodeNorm
X = np.array(df[['Mileage', 'Gamingtime', 'Icecream']])
y = np.array(df['Rating'])
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=40, stra

X_train_norm, parameters = encodeNorm(X_train)
X_test_norm, _ = encodeNorm(X_test, parameters=parameters)
```

- Using our codes.

```
# Using our codes.
from assests.codes.knn import classify_kNN

n_neighbors = 10
y_pred = np.array([classify_kNN(row, X_train_norm, y_train, k=n_neighbors)
                   for row in X_test_norm])

acc = np.mean(y_pred == y_test)
acc
```

```
0.93
```

- Using sklearn.

```
# Using sklearn.
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

steps = [('scaler', MinMaxScaler()),
         ('knn', KNeighborsClassifier(n_neighbors, weights="uniform",
                                      metric="euclidean", algorithm='brute'))]
pipe = Pipeline(steps=steps)
pipe.fit(X_train, y_train)
y_pipe = pipe.predict(X_test)
accuracy_score(y_pipe, y_test)
```

```
0.93
```

2.3.4 Choosing `k` Value

Similar to the previous section, we can run tests on `k` value to choose one to be used in our model using `GridSearchCV`.

```
from sklearn.model_selection import GridSearchCV, cross_val_score
n_list = list(range(1, 101))
parameters = dict(knn__n_neighbors=n_list)
clf = GridSearchCV(pipe, parameters)
clf.fit(X, y)
clf.best_estimator_.get_params()["knn__n_neighbors"]
```

```
4
```

From this result, in this case the best `k` is `4`. The corresponding cross-validation score is computed below.

```
cv_scores = cross_val_score(clf.best_estimator_, X, y, cv=5)
np.mean(cv_scores)
```

```
0.952
```

2.4 k-NN Project 3: Handwritten recognition

We would like to let the machine recognize handwritten digits. The dataset comes from the UCI dataset repository. Now we apply kNN algrotithm to it.

2.4.1 Dataset description

Every digit is stored as a 8×8 picture. This is a 8×8 matrix. Every entry represents a gray value of the corresponding pixel, whose value is from 0 to 16. The label of each matrix is the digit it represents. Note that the dataset provided is already splitted into a training set and a test set.

```
from sklearn import datasets
from sklearn.model_selection import train_test_split

X = datasets.load_digits().images
y = datasets.load_digits().target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.15)
```

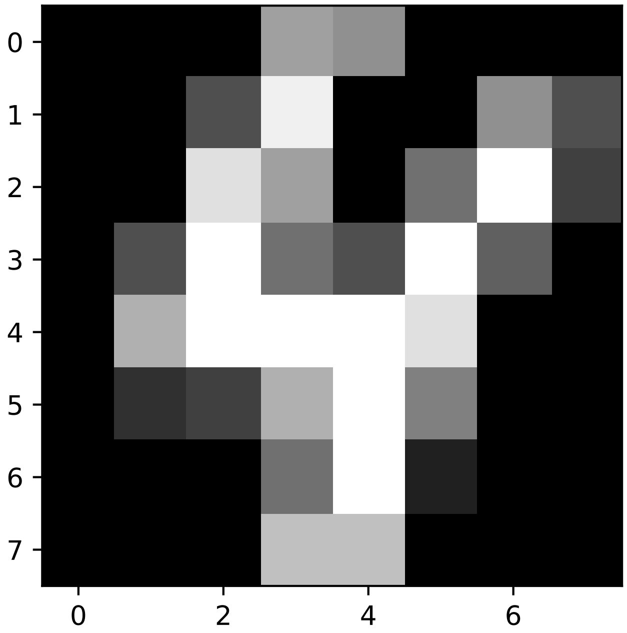
Let us play with these data first.

```
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
print(type(X_train))
```

```
(1527, 8, 8)
(1527,)
(270, 8, 8)
(270,)
<class 'numpy.ndarray'>
```

From these information we can see that the training set contains 1527 digits and the test set contains 270 digits. Each digit is represented by a 8×8 numpy array. Let us load one and display the digit by `matplotlib.pyplot.imshow`.

```
digit = X_train[0]

import matplotlib.pyplot as plt
plt.imshow(digit, cmap='gray')
```



This image represents a handwritten digit. Could you recognize it? We could check our guess by looking at the label. The following shows that it is a 4.

```
y_train[0]
```

```
4
```

Now we need to reshape these digits from 8 × 8 numpy arraies to 64 numpy arraies. Similar to previous examples, we will also normalize the dataset.

```
from assests.codes.knn import encodeNorm

X_train = X_train.reshape((X_train.shape[0], X_train.shape[1]*X_train.shape[2]))
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1]*X_test.shape[2]))

X_train_norm, parameters = encodeNorm(X_train)
X_test_norm, _ = encodeNorm(X_test, parameters=parameters)
```

2.4.2 Apply k-NN

Like the previous two examples, we now try to apply the k-NN algorithm to classify these handwritten digits.

```
from assests.codes.knn import classify_kNN
import numpy as np

n_neighbors = 10
X_test_sample = X_test_norm
y_test_sample = y_test
y_pred = np.array([classify_kNN(row, X_train_norm, y_train, k=n_neighbors)
                   for row in X_test_sample])

acc = np.mean(y_pred == y_test_sample)
acc
```

```
0.9888888888888889
```

Now let us try to apply `sklearn` package. Note that we could run the code over the whole test set (which contains 10000 digits) and the speed is much faster comparing to our codes. To save time we won't grid search `k` here. The code is the same anyway.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

steps = [('scaler', MinMaxScaler()),
         ('knn', KNeighborsClassifier(n_neighbors, weights="uniform",
                                      metric="euclidean", algorithm='brute'))]
pipe = Pipeline(steps=steps)
pipe.fit(X_train, y_train)
y_pipe = pipe.predict(X_test)
accuracy_score(y_pipe, y_test)
```

```
0.9888888888888889
```

2.5 Exercises and Projects

Exercise 2.1. Handwritten example :label: ex2handwritten Consider the 1-dimensional data set shown below.

Table 2.3

x	1.5	2.5	3.5	4.5	5.0	5.5	5.75	6.5	7.5	10.5
y	+	+	-	-	-	+	+	-	+	+

Please use the data to compute the class of $x = 5.5$ according to $k = 1$, 3, 6 and 9. Please compute everything by hand.

Exercise 2.2.

Titanic

Please download the titanic dataset from here. This is the same dataset from what you dealt with in Chapter 1 Exercises. Therefore you may use the same way to prepare the data.

Please analyze the dataset and build a k-NN model to predict whether someone is survived or not. Note that you have to pick `k` at the end.

3 Decision Trees

Given a dataset with labels, the decision tree algorithm firstly trys to split the whole dataset into two different groups, based on some speicific features. Choose which feature to use and set the threshold for the split are done.

3.1 Gini impurity

To split a dataset, we need a metric to tell whether the split is good or not. The two most popular metrics that are used are Gini impurity and Entropy. The two metrics don't have essential differences, that the results obtained by applying each metric are very similar to each other. Therefore we will only focus on Gini impurity since it is slightly easier to compute and slightly easier to explain.

3.1.1 Motivation and Definition

Assume that we have a dataset of totally n objects, and these objects are divided into k classes. The i-th class has n_i objects. Then if we randomly pick an object, the probability to get an object belonging to the i-th class is

$$p_i = \frac{n_i}{n}$$

If we then guess the class of the object purely based on the distribution of each class, the probability that our guess is incorrect is

$$1 - p_i = 1 - \frac{n_i}{n}.$$

Therefore, if we randomly pick an object that belongs to the i-th class and randomly guess its class purely based on the distribution but our guess is wrong, the probability that such a thing happens is

$$p_i(1-p_i).$$

Consider all classes. The probability at which any object of the dataset will be mislabelled when it is randomly labeled is the sum of the probability described above:

$$\sum_{i=1}^{k} p_i(1-p_i) = \sum_{i=1}^{k} p_i - \sum_{i=1}^{k} p_i^2 = 1 - \sum_{i=1}^{k} p_i^2.$$

This is the definition formula for the *Gini impurity.*

Definition 3.1. The **Gini impurity** is calculated using the following formula

$$Gini = \sum_{i=1}^{k} p_i(1-p_i) = \sum_{i=1}^{k} p_i - \sum_{i=1}^{k} p_i^2 = 1 - \sum_{i=1}^{k} p_i^2,$$

where p_i is the probability of class i.

The way to understand Gini impurity is to consider some extreme examples.

Example 3.1. Assume that we only have one class. Therefore $k = 1$, and $p_1 = 1$. Then the Gini impurity is

$$Gini = 1 - 1^2 = 0.$$

This is the minimum possible Gini impurity. It means that the dataset is **pure**: all the objects contained are of one unique class. In this case, we won't make any mistakes if we randomly guess the label.

Example 3.2. Assume that we have two classes. Therefore $k = 2$. Consider the distribution p_1 and p_2. We know that $p_1 + p_2 = 1$. Therefore $p_2 = 1 - p_1$. Then the Gini impurity is

$$Gini(p_1) = 1 - p_1^2 - p_2^2 = 1 - p_1^2 - (1 - p_1)^2 = 2p_1 - 2p_1^2.$$

When $0 \leq p_1 \leq 1$, this function $Gini(p_1)$ is between 0 and 0.5. - It gets 0 when $p_1 = 0$ or 1. In these two cases, the dataset is still a one-class set since the size of one class is 0. - It gets 0.5 when $p_1 = 0.5$. This means that the Gini impurity is maximized when the size of different classes are balanced.

3.1.2 Algorithm

ℹ Algorithm: Gini impurity

Inputs A dataset $S = \{data = [features, label]\}$ with labels.
Outputs The Gini impurity of the dataset.

1. Get the size n of the dataset.
2. Go through the label list, and find all unique labels: $uniqueLabelList$.
3. Go through each label l in $uniqueLabelList$ and count how many elements belonging to the label, and record them as n_l.
4. Use the formula to compute the Gini impurity:

$$Gini = 1 - \sum_{l \in uniqueLabelList} \left(\frac{n_l}{n}\right)^2.$$

The sample codes are listed below:

```
import pandas as pd
def gini(S):
    N = len(S)
    y = S[:, -1].reshape(N)
    gini = 1 - ((pd.Series(y).value_counts()/N)**2).sum()
    return gini
```

3.2 CART Algorithms

3.2.1 Ideas

Consider a labeled dataset S with totally m elements. We use a feature k and a threshold t_k to split it into two subsets: S_l with m_l elements and S_r with m_r elements. Then the cost function of this split is

$$J(k, t_k) = \frac{m_l}{m} Gini(S_l) + \frac{m_r}{m} Gini(S_r).$$

It is not hard to see that the more pure the two subsets are the lower the cost function is. Therefore our goal is find a split that can minimize the cost function.

Algorithm: Split the Dataset

Inputs Given a labeled dataset $S = \{[features, label]\}$.
Outputs A best split (k, t_k).

1. For each feature k:
    1. For each value t of the feature:
        1. Split the dataset S into two subsets, one with $k \leq t$ and one with $k > t$.
        2. Compute the cost function $J(k, t)$.
        3. Compare it with the current smallest cost. If this split has smaller cost, replace the current smallest cost and pair with (k, t).
2. Return the pair (k, t_k) that has the smallest cost function.

We then use this split algorithm recursively to get the decision tree.

Classification and Regression Tree, CART

Inputs Given a labeled dataset $S = \{[features, label]\}$ and a maximal depth `max_depth`.
Outputs A decision tree.

1. Starting from the original dataset S. Set the working dataset $G = S$.
2. Consider a dataset G. If $Gini(G) \neq 0$, split G into G_l and G_r to minimize the cost function. Record the split pair (k, t_k).
3. Now set the working dataset $G = G_l$ and $G = G_r$ respectively, and apply the above two steps to each of them.
4. Repeat the above steps, until `max_depth` is reached.

Here are the sample codes.

```
def split(G):
    m = G.shape[0]
    gmini = gini(G)
    pair = None
    if gini(G) != 0:
        numOffeatures = G.shape[1] - 1
        for k in range(numOffeatures):
            for t in range(m):
                Gl = G[G[:, k] <= G[t, k]]
                Gr = G[G[:, k] > G[t, k]]
                gl = gini(Gl)
                gr = gini(Gr)
                ml = Gl.shape[0]
                mr = Gr.shape[0]
                g = gl*ml/m + gr*mr/m
                if g < gmini:
                    gmini = g
                    pair = (k, G[t, k])
                    Glm = Gl
                    Grm = Gr
        res = {'split': True,
               'pair': pair,
               'sets': (Glm, Grm)}
    else:
        res = {'split': False,
               'pair': pair,
               'sets': G}
    return res
```

For the purpose of counting labels, we also write a code to do so.

```
import pandas as pd
def countlabels(S):
    y = S[:, -1].reshape(S.shape[0])
    labelCount = dict(pd.Series(y).value_counts())
    return labelCount
```

3.3 Decision Tree Project 1: the iris dataset

We are going to use the Decision Tree model to study the `iris` dataset. This dataset has already studied previously using k-NN. Again we will only use the first two features for visualization purpose.

3.3.1 Initial setup

Since the dataset will be splitted, we will put `X` and `y` together as a single variable `S`. In this case when we split the dataset by selecting rows, the features and the labels are still paired correctly.

We also print the labels and the feature names for our convenience.

```
from sklearn.datasets import load_iris
import numpy as np
from assests.codes.dt import gini, split, countlabels

iris = load_iris()
```

```
X = iris.data[:, 2:]
y = iris.target
y = y.reshape((y.shape[0],1))
S = np.concatenate([X,y], axis=1)

print(iris.target_names)
print(iris.feature_names)
```

```
['setosa' 'versicolor' 'virginica']
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

3.3.2 Apply CART manually

We apply `split` to the dataset `S`.

```
r = split(S)
if r['split'] is True:
    Gl, Gr = r['sets']
    print(r['pair'])
    print('The left subset\'s Gini impurity is {g:.2f},'.format(g=gini(Gl)),
          ' and its label counts is {d:}'.format(d=countlabels(Gl)))
    print('The right subset\'s Gini impurity is {g:.2f},'.format(g=gini(Gr)),
          ' and its label counts is {d}'.format(d=countlabels(Gr)))
```

```
(0, 1.9)
The left subset's Gini impurity is 0.00,  and its label counts is {0.0: 50}
The right subset's Gini impurity is 0.50,  and its label counts is {1.0: 50, 2.0: 50}
```

The results shows that `S` is splitted into two subsets based on the `0`-th feature and the split value is `1.9`.

The left subset is already pure since its Gini impurity is `0`. All elements in the left subset is label `0` (which is `setosa`). The right one is mixed since its Gini impurity is `0.5`. Therefore we need to apply `split` again to the right subset.

```python
r = split(Gr)
if r['split'] is True:
    Grl, Grr = r['sets']
    print(r['pair'])
    print('The left subset\'s Gini impurity is {g:.2f},'.format(g=gini(Grl)),
          ' and its label counts is {d:}'.format(d=countlabels(Grl)))
    print('The right subset\'s Gini impurity is {g:.2f},'.format(g=gini(Grr)),
          ' and its label counts is {d}'.format(d=countlabels(Grr)))
```

```
(1, 1.7)
The left subset's Gini impurity is 0.17,  and its label counts is {1.0: 49, 2.0: 5}
The right subset's Gini impurity is 0.04,  and its label counts is {2.0: 45, 1.0: 1}
```

This time the subset is splitted into two more subsets based on the 1-st feature and the split value is 1.7. The total Gini impurity is minimized using this split.

The decision we created so far can be described as follows:

1. Check the first feature `sepal length (cm)` to see whether it is smaller or equal to 1.9.

    1. If it is, classify it as lable 0 which is `setosa`.
    2. If not, continue to the next stage.

2. Check the second feature `sepal width (cm)` to see whether it is smaller or equal to 1.7.

    1. If it is, classify it as label 1 which is `versicolor`.
    2. If not, classify it as label 2 which is `virginica`.

3.3.3 Use package `sklearn`

Now we would like to use the decision tree package provided by `sklearn`. The process is straightforward. The parameter `random_state=40` will be discussed {ref}`later<note-random_state>`, and it is not necessary in most cases.

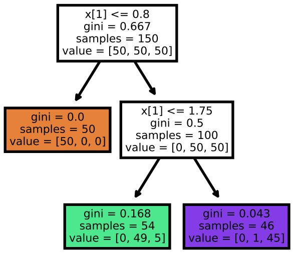
```python
from sklearn import tree
clf = tree.DecisionTreeClassifier(max_depth=2, random_state=40)
clf.fit(X, y)
```

```
DecisionTreeClassifier(max_depth=2, random_state=40)
```

`sklearn` provide a way to automatically generate the tree view of the decision tree. The code is as follows.

```python
import matplotlib.pyplot as plt
plt.figure(figsize=(2, 2), dpi=200)
tree.plot_tree(clf, filled=True, impurity=True)
```

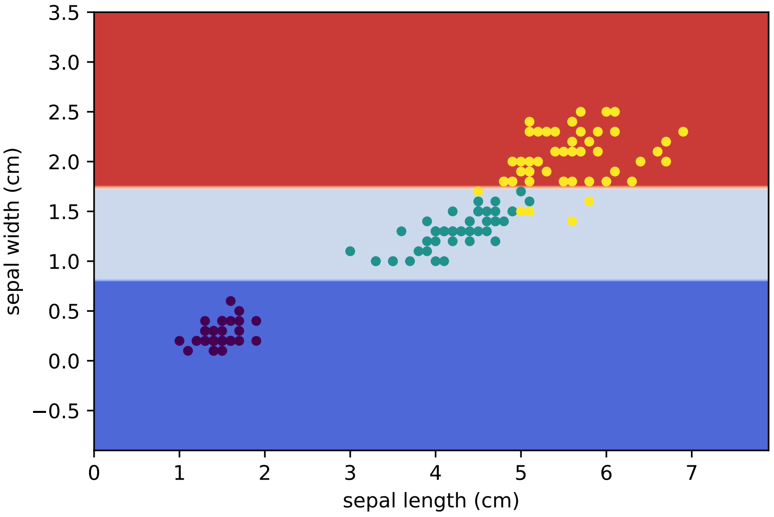
```
[Text(0.4, 0.8333333333333334, 'x[1] <= 0.8\ngini = 0.667\nsamples = 150\nvalue = [50, 50, 50]
 Text(0.2, 0.5, 'gini = 0.0\nsamples = 50\nvalue = [50, 0, 0]'),
 Text(0.6, 0.5, 'x[1] <= 1.75\ngini = 0.5\nsamples = 100\nvalue = [0, 50, 50]'),
 Text(0.4, 0.16666666666666666, 'gini = 0.168\nsamples = 54\nvalue = [0, 49, 5]'),
 Text(0.8, 0.16666666666666666, 'gini = 0.043\nsamples = 46\nvalue = [0, 1, 45]')]
```



Similar to k-NN, we may use `sklearn.inspection.DecisionBoundaryDisplay` to visualize the decision boundary of this decision tree.

```python
from sklearn.inspection import DecisionBoundaryDisplay
DecisionBoundaryDisplay.from_estimator(
    clf,
    X,
    cmap='coolwarm',
    response_method="predict",
    xlabel=iris.feature_names[0],
    ylabel=iris.feature_names[1],
)

# Plot the training points
plt.scatter(X[:, 0], X[:, 1], c=y, s=15)
```

```
<matplotlib.collections.PathCollection at 0x1a07451d1b0>
```



3.3.4 Analyze the differences between the two methods

The tree generated by `sklearn` and the tree we got manually is a little bit different. Let us explore the differences here.

To make it easier to split the set, we could convert the `numpy.ndarray` to `pandas.DataFrame`.

```python
import pandas as pd

df = pd.DataFrame(X)
df.head()
```

	0	1
0	1.4	0.2
1	1.4	0.2
2	1.3	0.2
3	1.5	0.2
4	1.4	0.2

Now based on our tree, we would like to get all data points that the first feature (which is marked as `0`) is smaller or equal to `1.9`. We save it as `df1`. Similarly based on the tree gotten from `sklearn`, we would like to get all data points taht the second feature (which is marked as `1`) is smaller or equal to `0.8` and save it to `df2`.

```
df1 = df[df[0]<=1.9]
df2 = df[df[1]<=0.8]
```

Then we would like to compare these two dataframes. What we want is to see whether they are the same regardless the order. One way to do this is to sort the two dataframes and then compare them directly.

To sort the dataframe we use the method `DataFrame.sort_values`. The details can be found here. Note that after `sort_values` we apply `reset_index` to reset the index just in case the index is massed by the sort operation.

Then we use `DataFrame.equals` to check whether they are the same.

```
df1sorted = df1.sort_values(by=df1.columns.tolist()).reset_index(drop=True)
df2sorted = df2.sort_values(by=df2.columns.tolist()).reset_index(drop=True)
print(df1sorted.equals(df2sorted))
```

```
True
```

So these two sets are really the same. The reason this happens can be seen from the following two graphs.

So you can see that either way can give us the same classification. This means that given one dataset the construction of the decision tree might be random at some points.

note-random_state

Since the split is random, when using `sklearn.DecisionTreeClassifier` to construct decision trees, sometimes we might get the same tree as

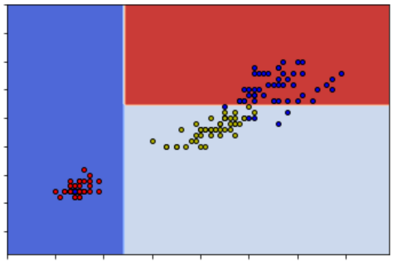


Figure 3.1: From our code

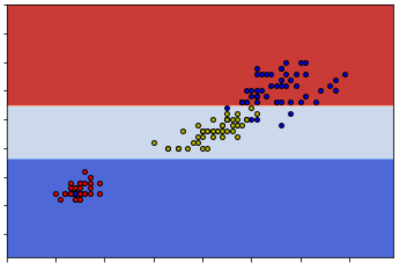


Figure 3.2: From sklearn

what we get from our naive codes.
To illustrate this phenomenaon I need to set the random state in case it will generate the same tree as ours when I need it to generate a different tree. The parameter random_state=40 mentioned before is for this purpose.

Another difference is the split value of the second branch. In our case it is 1.7 and in sklearn case it is 1.75. So after we get the right subset from the first split (which is called dfr), we would split it into two sets based on whether the second feature is above or below 1.7.

```
dfr = df[df[0]>1.9]
df2a = dfr[dfr[1]>1.7]
df2b = dfr[dfr[1]<=1.7]
print(df2b[1].max())
print(df2a[1].min())
```

```
1.7
1.8
```

Now you can see where the split number comes from. In our code, when we found a split, we will directly use that number as the cut. In this case it is 1.7.

In sklearn, when it finds a split, the algorithm will go for the middle of the gap as the cut. In this case it is (1.7+1.8)/2=1.75.

3.4 Decision Tree Project 2: make_moons dataset

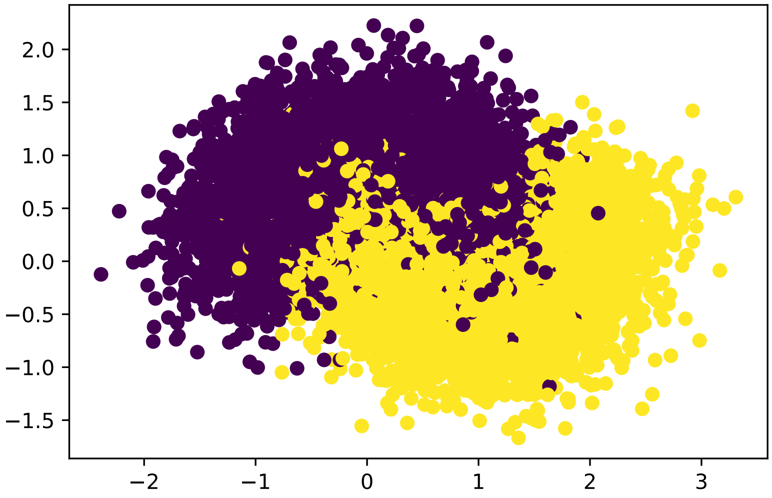
sklearn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity. We are going to use make_moons in this section. More details can be found here.

make_moons generate 2d binary classification datasets that are challenging to certain algorithms (e.g. centroid-based clustering or linear classification), including optional Gaussian noise. make_moons produces two interleaving half circles. It is useful for visualization.

Let us explorer the dataset first.

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt

X, y = make_moons(n_samples=10000, noise=0.4, random_state=42)
plt.scatter(x=X[:, 0], y=X[:, 1], c=y)
```

```
<matplotlib.collections.PathCollection at 0x1a0756559c0>
```



Now we are applying `sklearn.DecisionTreeClassifier` to construct the decision tree. The steps are as follows.

1. Split the dataset into training data and test data.
2. Construct the pipeline. Since we won't apply any transformers there for this problem, we may just use the classifier `sklearn.DecisionTreeClassifier` directly without really construct the pipeline object.
3. Consider the hyperparameter space for grid search. For this problme we choose `min_samples_split` and `max_leaf_nodes` as the hyperparameters we need. We will let `min_samples_split` run through 2 to 5, and `max_leaf_nodes` run through 2 to 50. We will use `grid_search_cv` to find the best hyperparameter for our model. For cross-validation, the number of split is set to be `3` which means that we will run trainning 3 times for each pair of hyperparameters.
4. Run `grid_search_cv`. Find the best hyperparameters and the best estimator. Test it on the test set to get the accuracy score.

```
# Step 1
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Step 3
from sklearn.model_selection import GridSearchCV
from sklearn.tree import DecisionTreeClassifier
import numpy as np

params = {'min_samples_split': list(range(2, 5)),
          'max_leaf_nodes': list(range(2, 50))}
grid_search_cv = GridSearchCV(DecisionTreeClassifier(random_state=42),
                              params, verbose=1, cv=3)
grid_search_cv.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 144 candidates, totalling 432 fits
```

```
GridSearchCV(cv=3, estimator=DecisionTreeClassifier(random_state=42),
             param_grid={'max_leaf_nodes': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                             13, 14, 15, 16, 17, 18, 19, 20, 21,
                                             22, 23, 24, 25, 26, 27, 28, 29, 30,
                                             31, ...],
                         'min_samples_split': [2, 3, 4]},
             verbose=1)
```

```
# Step 4
from sklearn.metrics import accuracy_score

clf = grid_search_cv.best_estimator_
print(grid_search_cv.best_params_)
y_pred = clf.predict(X_test)
accuracy_score(y_pred, y_test)
```

```
{'max_leaf_nodes': 17, 'min_samples_split': 2}
```

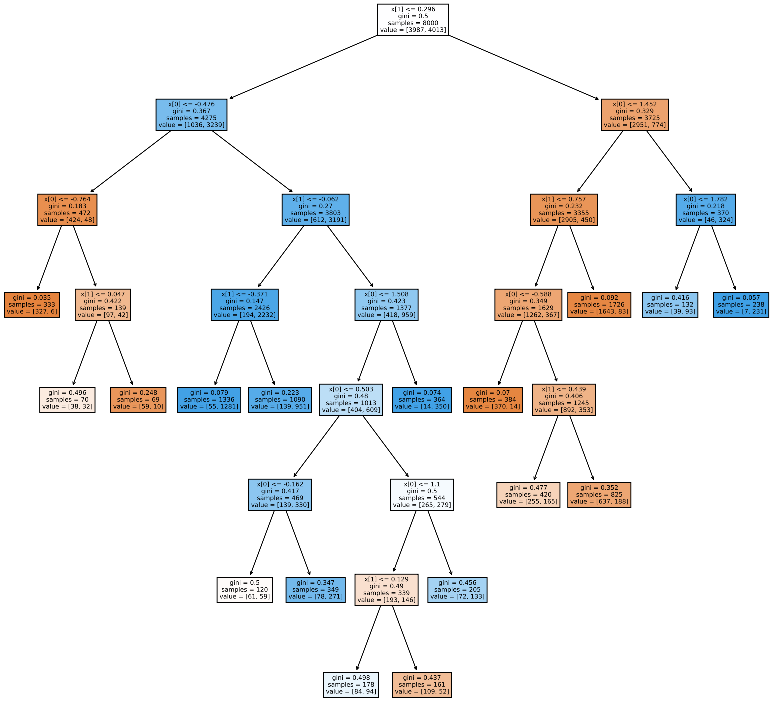
```
0.8695
```

Now you can see that for this `make_moons` dataset, the best decision tree should have at most 17 leaf nodes and the minimum number of samples required to be at a leaft node is 2. The fitted decision tree can get 86.95% accuracy on the test set.

Now we can plot the decision tree and the decision surface.

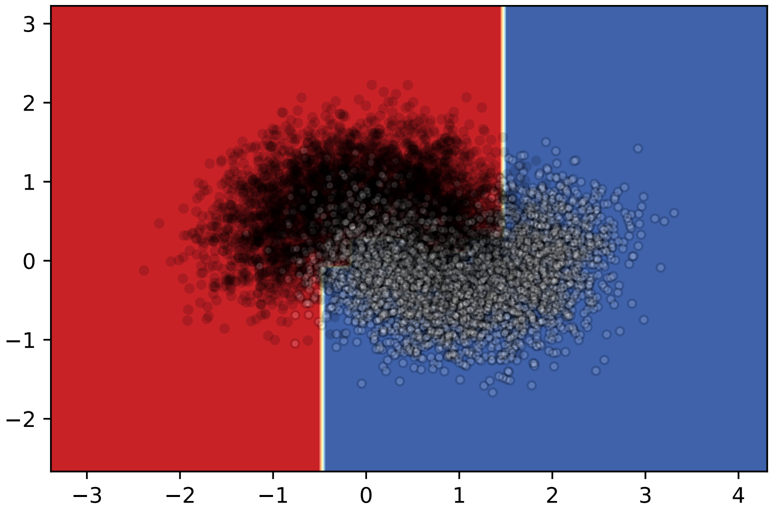
```
from sklearn import tree
plt.figure(figsize=(15, 15), dpi=300)
tree.plot_tree(clf, filled=True)
```

```
[Text(0.5340909090909091, 0.9375, 'x[1] <= 0.296\ngini = 0.5\nsamples = 8000\nvalue = [3987, 40
 Text(0.25, 0.8125, 'x[0] <= -0.476\ngini = 0.367\nsamples = 4275\nvalue = [1036, 3239]'),
 Text(0.09090909090909091, 0.6875, 'x[0] <= -0.764\ngini = 0.183\nsamples = 472\nvalue = [424,
 Text(0.045454545454545456, 0.5625, 'gini = 0.035\nsamples = 333\nvalue = [327, 6]'),
 Text(0.13636363636363635, 0.5625, 'x[1] <= 0.047\ngini = 0.422\nsamples = 139\nvalue = [97, 4
 Text(0.09090909090909091, 0.4375, 'gini = 0.496\nsamples = 70\nvalue = [38, 32]'),
 Text(0.18181818181818182, 0.4375, 'gini = 0.248\nsamples = 69\nvalue = [59, 10]'),
 Text(0.4090909090909091, 0.6875, 'x[1] <= -0.062\ngini = 0.27\nsamples = 3803\nvalue = [612, 3
 Text(0.3181818181818182, 0.5625, 'x[1] <= -0.371\ngini = 0.147\nsamples = 2426\nvalue = [194,
 Text(0.2727272727272727, 0.4375, 'gini = 0.079\nsamples = 1336\nvalue = [55, 1281]'),
 Text(0.36363636363636365, 0.4375, 'gini = 0.223\nsamples = 1090\nvalue = [139, 951]'),
 Text(0.5, 0.5625, 'x[0] <= 1.508\ngini = 0.423\nsamples = 1377\nvalue = [418, 959]'),
 Text(0.45454545454545453, 0.4375, 'x[0] <= 0.503\ngini = 0.48\nsamples = 1013\nvalue = [404, 6
 Text(0.36363636363636365, 0.3125, 'x[0] <= -0.162\ngini = 0.417\nsamples = 469\nvalue = [139,
 Text(0.3181818181818182, 0.1875, 'gini = 0.5\nsamples = 120\nvalue = [61, 59]'),
 Text(0.4090909090909091, 0.1875, 'gini = 0.347\nsamples = 349\nvalue = [78, 271]'),
 Text(0.5454545454545454, 0.3125, 'x[0] <= 1.1\ngini = 0.5\nsamples = 544\nvalue = [265, 279]')
 Text(0.5, 0.1875, 'x[1] <= 0.129\ngini = 0.49\nsamples = 339\nvalue = [193, 146]'),
 Text(0.45454545454545453, 0.0625, 'gini = 0.498\nsamples = 178\nvalue = [84, 94]'),
 Text(0.5454545454545454, 0.0625, 'gini = 0.437\nsamples = 161\nvalue = [109, 52]'),
 Text(0.5909090909090909, 0.1875, 'gini = 0.456\nsamples = 205\nvalue = [72, 133]'),
 Text(0.5454545454545454, 0.4375, 'gini = 0.074\nsamples = 364\nvalue = [14, 350]'),
 Text(0.8181818181818182, 0.8125, 'x[0] <= 1.452\ngini = 0.329\nsamples = 3725\nvalue = [2951,
 Text(0.7272727272727273, 0.6875, 'x[1] <= 0.757\ngini = 0.232\nsamples = 3355\nvalue = [2905,
 Text(0.6818181818181818, 0.5625, 'x[0] <= -0.588\ngini = 0.349\nsamples = 1629\nvalue = [1262
 Text(0.6363636363636364, 0.4375, 'gini = 0.07\nsamples = 384\nvalue = [370, 14]'),
 Text(0.7272727272727273, 0.4375, 'x[1] <= 0.439\ngini = 0.406\nsamples = 1245\nvalue = [892, 3
 Text(0.6818181818181818, 0.3125, 'gini = 0.477\nsamples = 420\nvalue = [255, 165]'),
 Text(0.7727272727272727, 0.3125, 'gini = 0.352\nsamples = 825\nvalue = [637, 188]'),
 Text(0.7727272727272727, 0.5625, 'gini = 0.092\nsamples = 1726\nvalue = [1643, 83]'),
 Text(0.9090909090909091, 0.6875, 'x[0] <= 1.782\ngini = 0.218\nsamples = 370\nvalue = [46, 324
 Text(0.8636363636363636, 0.5625, 'gini = 0.416\nsamples = 132\nvalue = [39, 93]'),
 Text(0.9545454545454546, 0.5625, 'gini = 0.057\nsamples = 238\nvalue = [7, 231]')]
```



```python
from sklearn.inspection import DecisionBoundaryDisplay

DecisionBoundaryDisplay.from_estimator(
    clf,
    X,
    cmap=plt.cm.RdYlBu,
    response_method="predict"
)
plt.scatter(
    X[:, 0],
    X[:, 1],
    c=y,
    cmap='gray',
    edgecolor="black",
    s=15,
    alpha=.15)
```

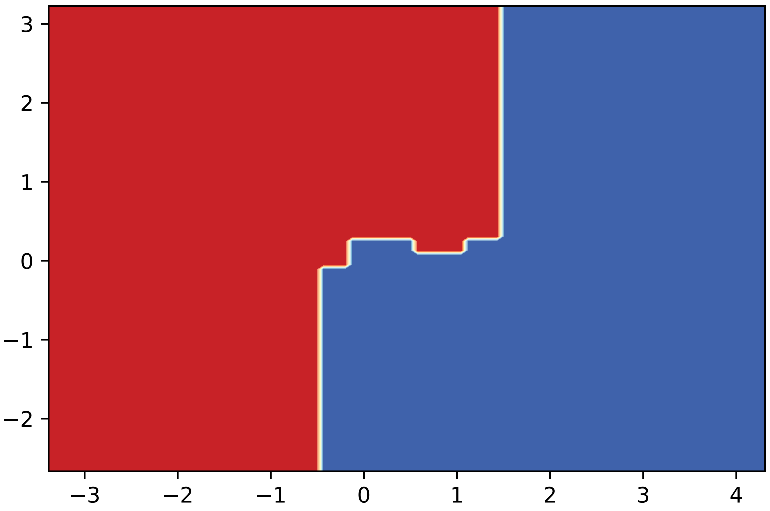
```
<matplotlib.collections.PathCollection at 0x1a0757b1750>
```



Since it is not very clear what the boundary looks like, I will draw the decision surface individually below.

```
DecisionBoundaryDisplay.from_estimator(
    clf,
    X,
    cmap=plt.cm.RdYlBu,
    response_method="predict"
)
```

```
<sklearn.inspection._plot.decision_boundary.DecisionBoundaryDisplay at 0x1a0757d07f0>
```



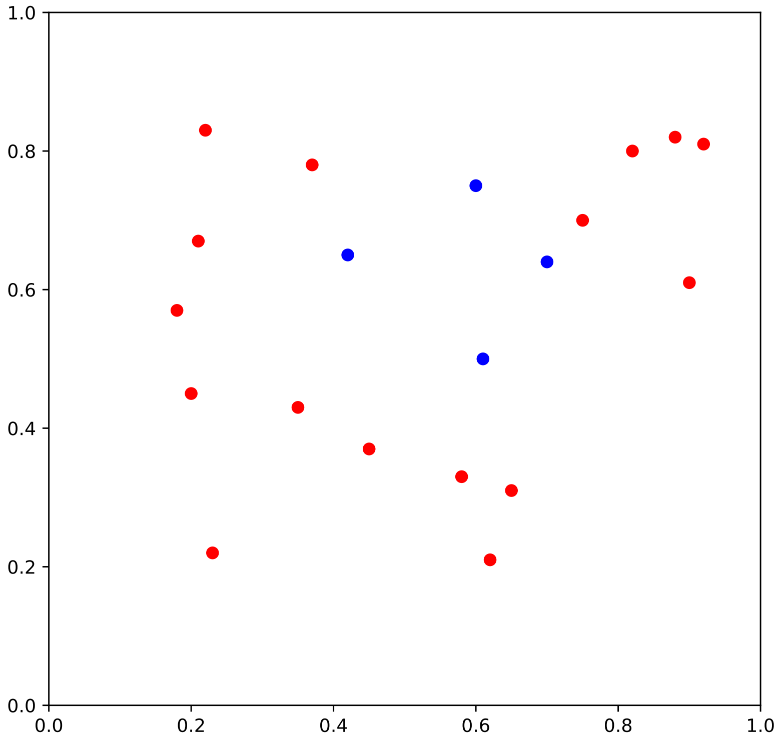
3.5 Exercises and Projects

Exercise 3.1. The dataset and its scattering plot is given below.

1. Please calculate the Gini impurity of the whole set by hand.
2. Please apply CART to create the decision tree by hand.
3. Please use the tree you created to classify the following points:

   - $(0.4, 1.0)$
   - $(0.6, 1.0)$
   - $(0.6, 0)$

The following code is for ploting. You may also get the precise data points by reading the code. You don't need to write codes to solve the problem.



Exercise 3.2. CHOOSE ONE: Please apply the Decision Tree to one of the following datasets.

- dating dataset (in Chpater 2).
- the `titanic` dataset.

Please answer the following questions.

1. Please use grid search to find the good `max_leaf_nodes` and `max_depth`.
2. Please record the accuracy (or cross-validation score) of your model and compare it with the models you learned before (kNN).
3. Please find the two most important features and explane your reason.
4. (Optional) Use the two most important features to draw the Decision Boundary if possible.

4 Ensemble methods

After we get some relatively simple classifiers (sometimes also called *weak classifiers*), we might put them together to form a more complicated classifier. This type of methods is called an *ensemble method*. The basic way to "ensemble'' classifiers together to through the voting machine.

There are mainly two ways to generate many classifiers.

- `bagging`: This is also called *bootstrap aggregating*. The idea is
    - First we randomly pick samples from the original dataset to form a bunch of new trainning datasets;
    - Then we apply the same learning methods to those trainning datasets to get a bunch of classifiers;
    - Finally apply all these classifiers to the data we are interested in and use the most frequent class as the result.
- `boosting`: There are a bunch of classifiers. We assign weights to each of the classifiers and change the weights adaptively according to the results of the current combination.

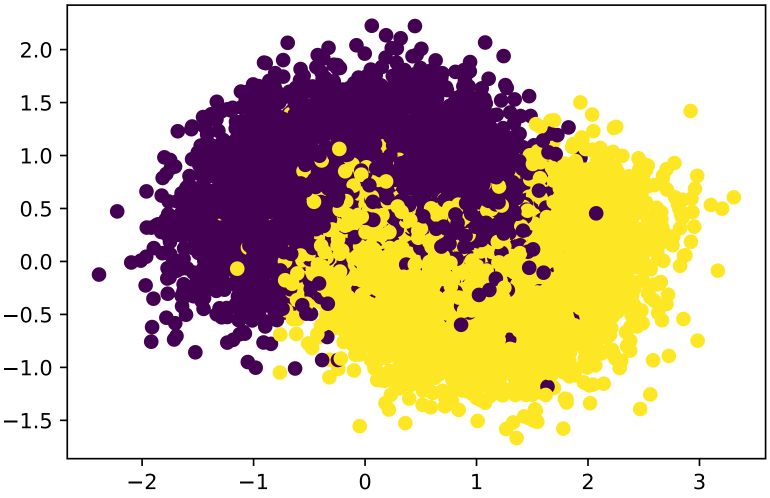
4.1 Bootstrap aggregating

4.1.1 Basic bagging

One approach to get many estimators is to use the same training algorithm for every predictor and train them on different random subsets of the training set. When sampling is performed with replacement, this method is called *bagging* (short for *bootstrap aggregating*). When sampling is performed without replacement, it is called *pasting*.

Consider the following example. The dataset is the one we used in Chpater 3: make_moon. We split the dataset into training and test sets.

```
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

X, y = make_moons(n_samples=10000, noise=0.4, random_state=42)
plt.scatter(x=X[:, 0], y=X[:, 1], c=y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```



We would like to sample from the dataset to get some smaller minisets. We will use sklearn.model_selection.ShuffleSplit to perform the action.

The output of ShuffleSplit is a generator. To get the index out of it we need a for loop. You may check out the following code.

Note that ShuffleSplit is originally used to shuffle data into training and test sets. We would only use the shuffle function out of it, so we will set test_size to be 1 and use _ later in the for loop since we won't use that part of the information.

What we finally get is a generator rs that produces indexes of subsets of X_train and y_train.

```
from sklearn.model_selection import ShuffleSplit
n_trees = 1000
n_instances = 100
rs = ShuffleSplit(n_splits=n_trees, test_size=1, train_size=n_instances).split(X_train)
```

Now we would like to generate a list of Decision Trees. We could use the hyperparameters we get from Chapter 3. We train each tree over a certain mini set, and then evaluate the trained model over the test set. The average accuracy is around 80%.

Note that rs is a generator. We put it in a for loop, and during each loop it will produce a list of indexes which gives a subset. We will directly train our model over the subset and use it to predict the test set. The result of each tree is put in the list y_pred_list and the accuracy is stored in the list acc_list. The mean of the accuracy is then computed by np.mean(acc_list).

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
import numpy as np

y_pred_list = list()
acc_list = list()
for mini_train_index, _ in rs:
    X_subset = X_train[mini_train_index]
    y_subset = y_train[mini_train_index]
    clf_ind = DecisionTreeClassifier(min_samples_split=2, max_leaf_nodes=17)
    clf_ind.fit(X_subset, y_subset)
    y_pred = clf_ind.predict(X_test)
    y_pred_list.append(y_pred)
    acc_list.append(accuracy_score(y_pred, y_test))

np.mean(acc_list)
```

```
0.8028373333333334
```

Now for each test data, we actually have `n_trees=1000` predicted results. We can treat it as the options from 1000 exports and would like to use the majority as our result. For this purpose we would like to use `mode()` which will find the most frequent entry.

```
from scipy.stats import mode
voting = np.array(y_pred_list)
y_pred_mode, _ = mode(voting, axis=0, keepdims=False)
```

Since the output of `mode` is a tuple where the first entry is a 2D array, we need to reshape `y_pred_mode`. This is the result using this voting system. Then we are able to compute the accuracy, and find that it is increased from the previous prediction.

```
accuracy_score(y_pred_mode, y_test)
```

```
0.878
```

4.1.2 Some rough analysis

The point of `Bagging` is to let every classifier study part of the data, and then gather the opinions from everyone. If the performance are almost the same between individual classifers and the Bagging classifiers, this means that the majority of the individual classifiers have the same opinions. One possible reason is that the randomized subsets already catch the main features of the dataset that every individual classifiers behave similar.

4.1.2.1 Case 1

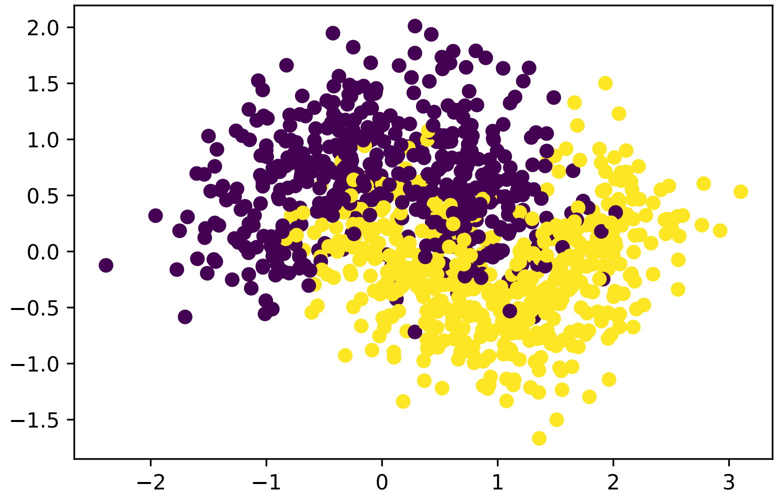
Let us continue with the previous dataset. We start from using Decision Tree with `max_depth=1`. In other words each tree only split once.

```
n_trees = 500
n_instances = 1000
rs = ShuffleSplit(n_splits=n_trees, test_size=1, train_size=n_instances).split(X_train)
y_pred_list = list()
acc_list = list()
for mini_train_index, _ in rs:
    X_subset = X_train[mini_train_index]
    y_subset = y_train[mini_train_index]
    clf_ind = DecisionTreeClassifier(max_depth=1)
    clf_ind.fit(X_subset, y_subset)
    y_pred = clf_ind.predict(X_test)
    y_pred_list.append(y_pred)
    acc_list.append(accuracy_score(y_pred, y_test))
print('The mean of individual accuracy: {}'.format(np.mean(acc_list)))

voting = np.array(y_pred_list)
y_pred_mode, _ = mode(voting, axis=0, keepdims=False)
print('The accuracy of the bagging classifier: {}'.format(accuracy_score(y_pred_mode, y_test)))
```

```
The mean of individual accuracy: 0.7691173333333333
The accuracy of the bagging classifier: 0.7773333333333333
```

The two accuracy has some differences, but not much. This is due to the fact that the sample size of the subset is too large: 1000 can already help the individual classifers to capture the major ideas of the datasets. Let us see the first 1000 data points. The scattering plot is very similar to that of the whole dataset shown above.

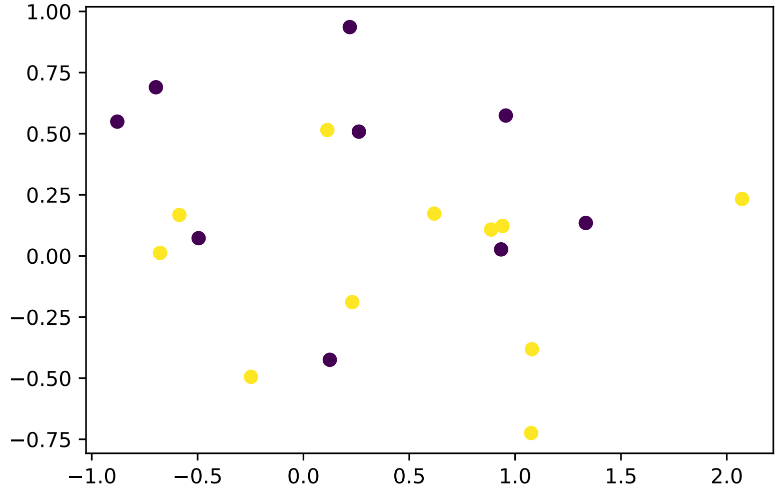
```
Npiece = 1000
plt.scatter(x=X[:Npiece, 0], y=X[:Npiece, 1], c=y[:Npiece])
```



4.1.2.2 Case 2

If we reduce the sample size to be very small, for example, 20, the sampled subset will lose a lot of information and it will be much harder to capture the idea of the original dataset. See the scattering plot of the first 20 data points.

```
Npiece = 20
plt.scatter(x=X[:Npiece, 0], y=X[:Npiece, 1], c=y[:Npiece])
```



In this case, let us see the performance comparison between multiple decision trees and the bagging classifier.

```
n_trees = 500
n_instances = 20
rs = ShuffleSplit(n_splits=n_trees, test_size=1, train_size=n_instances).split(X_train)
y_pred_list = list()
acc_list = list()
for mini_train_index, _ in rs:
    X_subset = X_train[mini_train_index]
    y_subset = y_train[mini_train_index]
    clf_ind = DecisionTreeClassifier(max_depth=1)
    clf_ind.fit(X_subset, y_subset)
    y_pred = clf_ind.predict(X_test)
    y_pred_list.append(y_pred)
    acc_list.append(accuracy_score(y_pred, y_test))
print('The mean of individual accuracy: {}'.format(np.mean(acc_list)))

voting = np.array(y_pred_list)
y_pred_mode, _ = mode(voting, axis=0, keepdims=False)
print('The accuracy of the bagging classifier: {}'.format(accuracy_score(y_pred_mode, y_test)))
```

```
The mean of individual accuracy: 0.7188066666666666
The accuracy of the bagging classifier: 0.822
```

This time you may see a significant increase in the performance.

4.1.3 Using sklearn

sklearn provides BaggingClassifier to directly perform bagging or pasting. The code is as follows.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(),
                            n_estimators=1000,
                            max_samples=100,
                            bootstrap=True)
```

In the above code, bag_clf is a bagging classifier, made of 500 DecisionTreeClassifers, and is trained over subsets of size 100. The option bootstrap=True means that it is bagging. If you would like to use pasting, the option is bootstrap=False.

This bag_clf also has .fit() and .predict() methods. It is used the same as our previous classifiers. Let us try the make_moon dataset.

```
bag_clf.fit(X_train, y_train)
y_pred_bag = bag_clf.predict(X_test)
accuracy_score(y_pred_bag, y_test)
```

```
0.8746666666666667
```

4.1.4 OOB score

When we use bagging, it is possible that some of the training data are not used. In this case, we could record which data are not used, and just use them as the test set, instead of providing extra data for test. The data that are not used is called *out-of-bag* instances, or *oob* for short. The accuracy over the oob data is called the oob score.

We could set oob_score=True to enable the function when creating a BaggingClassifier, and use .oob_score_ to get the oob score after training.

```
bag_clf_oob = BaggingClassifier(DecisionTreeClassifier(),
                                n_estimators=1000,
                                max_samples=100,
                                bootstrap=True,
                                oob_score=True)
bag_clf_oob.fit(X_train, y_train)
bag_clf_oob.oob_score_
```

```
0.863764705882353
```

4.1.5 Random Forests

When the classifiers used in a bagging classifier are all Decision Trees, the bagging classifier is called a `random forest`. `sklearn` provide `RandomForestClassifier` class. It is almost the same as `BaggingClassifier` + `DecisionTreeClassifer`.

```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=1000, max_leaf_nodes=17)
rnd_clf.fit(X_train, y_train)
y_pred_rnd = rnd_clf.predict(X_test)
accuracy_score(y_pred_rnd, y_test)
```

```
0.8686666666666667
```

When we use the Decision Tree as our base estimators, the class `RandomForestClassifier` provides more control over growing the random forest, with a certain optimizations. If you would like to use other estimators, then `BaggingClassifier` should be used.

4.1.6 Extra-trees

When growing a Decision Tree, our method is to search through all possible ways to find the best split point that get the lowest Gini impurity. Anohter method is to use a random split. Of course a random tree performs much worse, but if we use it to form a random forest, the voting system can help to increase the accuracy. On the other hand, random split is much faster than a regular Decision Tree.

This type of forest is called *Extremely Randomized Trees*, or *Extra-Trees* for short. We could modify the above random forest classifier code to implement the extra-tree algorithm. The key point is that we don't apply the Decision Tree algorithm to `X_subset`. Instead we perform a random split.

```
n_trees = 500
n_instances = 20
rs = ShuffleSplit(n_splits=n_trees, test_size=1, train_size=n_instances).split(X_train)
y_pred_list = list()
acc_list = list()
for mini_train_index, _ in rs:
    X_subset = X_train[mini_train_index]
    y_subset = y_train[mini_train_index]
    clf_ind = DecisionTreeClassifier(max_depth=1)
# random split
    i = np.random.randint(0, X_subset.shape[0])
    j = np.random.randint(0, X_subset.shape[1])
    split_threshold = X_subset[i, j]
    lsetindex = np.where(X_subset[:, j]<split_threshold)[0]

    if len(lsetindex) == 0:
        rsetindex = np.where(X_subset[:, j]>=split_threshold)
        rmode, _ = mode(y_subset[rsetindex], keepdims=True)
        rmode = rmode[0]
        lmode = 1 - rmode
    else:
        lmode, _ = mode(y_subset[lsetindex], keepdims=True)
        lmode = lmode[0]
        rmode = 1 - lmode
    y_pred = np.where(X_test[:, j] < split_threshold, lmode, rmode).reshape(-1)
# The above code is used to use the random split to classify the data points
    y_pred_list.append(y_pred)
    acc_list.append(accuracy_score(y_pred, y_test))
print('The mean of individual accuracy: {}'.format(np.mean(acc_list)))

voting = np.array(y_pred_list)
y_pred_mode, _ = mode(voting, axis=0, keepdims=False)
print('The accuracy of the bagging classifier: {}'.format(accuracy_score(y_pred_mode, y_test))
```

```
The mean of individual accuracy: 0.6236053333333333
The accuracy of the bagging classifier: 0.8153333333333334
```

From the above example, you may find a significant increase in the performace from the mean individual accuracy to the Extra-tree classifier accuracy. The accuracy of the Extra-tree classifier is also very close to what we get from the original data points, although its base classifier is much simpler.

In `sklearn` there is an `ExtraTreesClassifier` to create such a classifier. It is hard to say which random forest is better beforehand. What we can do is to test and calculate the cross-validation scores (with grid search for hyperparameters tuning).

```
from sklearn.ensemble import ExtraTreesClassifier

ext_clf = ExtraTreesClassifier(n_estimators=1000, max_leaf_nodes=17)
ext_clf.fit(X_train, y_train)
y_pred_rnd = ext_clf.predict(X_test)
accuracy_score(y_pred_rnd, y_test)
```

```
0.8706666666666667
```

In the above example, `RandomForestClassifier` and `ExtraTreesClassifier` get similar accuracy. However from the code below, you will see that in this example `ExtraTreesClassifier` is much faster than `RandomForestClassifier`.

```
from time import time
t0 = time()
rnd_clf = RandomForestClassifier(n_estimators=1000, max_leaf_nodes=17)
rnd_clf.fit(X_train, y_train)
t1 = time()
print('Random Frorest: {}'.format(t1 - t0))

t0 = time()
ext_clf = ExtraTreesClassifier(n_estimators=1000, max_leaf_nodes=17)
ext_clf.fit(X_train, y_train)
t1 = time()
print('Extremely Randomized Trees: {}'.format(t1 - t0))
```

```
Random Frorest: 9.22689414024353
Extremely Randomized Trees: 2.5578181743621826
```

4.1.7 Gini importance

After training a Decision Tree, we could look at each node. Each split is against a feature, which decrease the Gini impurity the most. In other words, we could say that the feature is the most important during the split.

Using the average Gini impurity decreased as a metric, we could measure the importance of each feature. This is called *Gini importance*. If the feature is useful, it tends to split mixed labeled nodes into pure single class nodes.

In the case of random forest, since there are many trees, we might compute the weighted average of the Gini importance across all trees. The weight depends on how many times the feature is used in a specific node.

Using `RandomForestClassifier`, we can directly get access to the Gini importance of each feature by `.feature_importance_`. Please see the following example.

```
rnd_clf.fit(X_train, y_train)
rnd_clf.feature_importances_
```

```
array([0.44173708, 0.55826292])
```

In this example, you may see that the two features are relavely equally important, where the second feature is slightly more important since on average it decrease the Gini impurity a little bit more.

4.2 Voting machine

4.2.1 Voting classifier

Assume that we have several trained classifiers. The easiest way to make a better classifer out of what we already have is to build a voting system. That is, each classifier give its own prediction, and it will be considered as a vote, and finally the highest vote will be the prediction of the system.

In `sklearn`, you may use `VotingClassifier`. It works as follows.

```
from sklearn.ensemble import VotingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

clfs = [('knn', KNeighborsClassifier(n_neighbors=5)),
        ('dt', DecisionTreeClassifier(max_depth=2))]
voting_clf = VotingClassifier(estimators=clfs, voting='hard')
```

All classifiers are stored in the list `clfs`, whose elements are tuples. The syntax is very similar to `Pipeline`. What the classifier does is to train all listed classifiers and use the majority vote to predict the class of given test data. If each classifier has one vote, the voting method is `hard`. There is also a `soft` voting method. In this case, every classifiers not only can predict the classes of the given data, but also estimiate the probability of the given data that belongs to certain classes. On coding level, each classifier should have the `predict_proba()` method. In this case, the weight of each vote is determined by the probability computed. In our course we mainly use `hard` voting.

Let us use `make_moon` as an example. We first load the dataset.

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

X, y = make_moons(n_samples=10000, noise=0.4, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

We would like to apply kNN model. As before, we build a data pipeline `pipe` to first apply `MinMaxScaler` and then `KNeighborsClassifier`.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
```

```
pipe = Pipeline(steps=[('scalar', MinMaxScaler()),
                       ('knn', KNeighborsClassifier())])
parameters = {'knn__n_neighbors': list(range(1, 51))}
gs_knn = GridSearchCV(pipe, param_grid=parameters)
gs_knn.fit(X_train, y_train)
clf_knn = gs_knn.best_estimator_
clf_knn.score(X_test, y_test)
```

```
0.864
```

The resulted accuracy is shown above.

We then try it with the Decision Tree.

```
from sklearn.tree import DecisionTreeClassifier

gs_dt = GridSearchCV(DecisionTreeClassifier(), param_grid={'max_depth': list(range(1, 11)), 'ma
gs_dt.fit(X_train, y_train)
clf_dt = gs_dt.best_estimator_
clf_dt.score(X_test, y_test)
```

```
0.8653333333333333
```

We would also want to try Logistic regression method. This will be covered in the next Chapter. At current stage we just use the default setting without changing any hyperparameters.

```
from sklearn.linear_model import LogisticRegression
clf_lr = LogisticRegression()
clf_lr.fit(X_train, y_train)
clf_lr.score(X_test, y_test)
```

```
0.8346666666666667
```

Now we use a voting classifier to combine the results.

```
from sklearn.ensemble import VotingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
clfs = [('knn', KNeighborsClassifier()),
        ('dt', DecisionTreeClassifier()),
        ('lr', LogisticRegression())]
voting_clf = VotingClassifier(estimators=clfs, voting='hard')
voting_clf.fit(X_train, y_train)
voting_clf.score(X_test, y_test)
```

```
0.8473333333333334
```

You may compare the results of all these four classifiers. The voting classifier is not guaranteed to be better. It is just a way to form a model.

4.3 AdaBoost

This is the first algorithm that successfully implements the boosting idea. AdaBoost is short for *Adaptive Boosting*.

4.3.1 Weighted dataset

We firstly talk about training a Decision Tree on a weighted dataset. The idea is very simple. When building a Decision Tree, we use some method to determine the split. In this course the Gini impurity is used. There are at least two other methods: cross-entropy and misclassified rate. For all three, the count of the elemnts in some classes is the essnetial part. To train the model over the weighted dataset, we just need to upgrade the count of the elements by the weighted count.

Example 4.1. Consider the following data:

Table 4.1

x0	x1	y	Weight
1.0	2.1	+	0.5
1.0	1.1	+	0.125
1.3	1.0	-	0.125
1.0	1.0	-	0.125
2.0	1.0	+	0.125

The weighted Gini impurity is

$$\text{WeightedGini} = 1-(0.5+0.125+0.125)^2-(0.125+0.125)^2 = 0.375.$$

You may see that the original Gini impurity is just the weighted Gini impurity with equal weights. Therefore the first tree we get from `AdaBoost` (see below) is the same tree we get from the Decision Tree model in Chpater 3.

4.3.2 General process

Here is the rough description of `AdaBoost`.

1. Assign weights to each data point. At the begining we could assign weights equally.
2. Train a classifier based on the weighted dataset, and use it to predict on the training set. Find out all wrong answers.
3. Adjust the weights, by inceasing the weights of data points that are done wrongly in the previous generation.
4. Train a new classifier using the new weighted dataset. Predict on the training set and record the wrong answers.
5. Repeat the above process to get many classifiers. The training stops either by hitting 0 error rate, or after a specific number of rounds.
6. The final results is based on the weighted total votes from all classifiers we trained.

Now let us talk about the details. Assume there are N data points. Then the inital weights are set to be $\frac{1}{N}$. There are 2 sets of weights. Let $w^{(i)}$ be weights of the ith data points. Let α_j be the weights of the jth classifier. After training the jth classifier, the error rate is denoted by e_j. Then we have

$$e_j = \frac{\text{the total weights of data points that are misclassified by the } j\text{th classifier}}{\text{the total weights of data points}}$$

$$\alpha_j = \eta \ln\left(\frac{1-e_j}{e_j}\right).$$

$$w_{\text{new}}^{(i)} \leftarrow \text{normalization} \leftarrow w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if the } i\text{th data is correctly classified,} \\ w^{(i)}\exp(\alpha_j) & \text{if the } i\text{th data is misclassified.} \end{cases}$$

> Note
>
> The first tree is the same tree we get from the regular Decision Tree model. In the rest of the training process, more weights are put on the data that we are wrong in the previous iteration. Therefore the process is the mimic of "learning from mistakes".

> Note
>
> The η in computing α_j is called the *learning rate*. It is a hyperparameter that will be specified mannually. It does exactly what it appears to do: alter the weights of each classifier. The default is `1.0`. When the number is very small (which is recommended although it can be any positive number), more iterations will be expected.

4.3.3 Example 1: the `iris` dataset

Similar to all previous models, `sklearn` provides `AdaBoostClassifier`. The way to use it is similar to previous models. Note that although we are able to use any classifiers for `AdaBoost`, the most popular choice is Decision Tree with `max_depth=1`. This type of Decision Trees are also called *Decision Stumps*.

In the following examples, we initialize an `AdaBoostClassifier` with 500 Decision Stumps and `learning_rate=0.5`.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=1000,
                             learning_rate=.5)
```

We will use the `iris` dataset for illustration. The cross_val_score is calculated as follows.

```
from sklearn.model_selection import cross_val_score
from sklearn.datasets import load_iris

X, y = load_iris(return_X_y=True)
scores = cross_val_score(ada_clf, X, y, cv=5)
scores.mean()
```

```
0.9533333333333334
```

4.3.4 Example 2: the Horse Colic dataset

This dataset is from UCI Machine Learning Repository. The data is about whether horses survive if they get a disease called Colic. The dataset is preprocessed as follows. Note that there are a few missing values inside, and we replace them with `0`.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/horse-colic/horse-colic.data'
df = pd.read_csv(url, delim_whitespace=True, header=None)
df = df.replace("?", np.NaN)
df = df.fillna(0)
X = df.iloc[:, 1:].to_numpy().astype(float)
y = df[0].to_numpy().astype(int)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)

clf = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1), n_estimators=50, learning_rate=0
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

```
0.5777777777777777
```

4.4 Exercises

Exercise 4.1. CHOOSE ONE: Please apply the random forest to one of the following datasets.

- the `iris` dataset.
- the dating dataset.
- the `titanic` dataset.

Please answer the following questions.

1. Please use grid search to find the good `max_leaf_nodes` and `max_depth`.
2. Please record the cross-validation score and the OOB score of your model and compare it with the models you learned before (kNN, Decision Trees).
3. Please find some typical features (using the Gini importance) and draw the Decision Boundary against the features you choose.

Exercise 4.2. Please use the following code to get the `mgq` dataset.

```
from sklearn.datasets import make_gaussian_quantiles

X1, y1 = make_gaussian_quantiles(cov=2.0, n_samples=200, n_features=2,
                                 n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5, n_samples=300,
                                          n_features=2, n_classes=2, random_state=1)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, -y2 + 1))
```

Please build an `AdaBoost` model.

Exercise 4.3. Please use `RandomForestClassifier`, `ExtraTreesClassifier` and `KNeighbourClassifier` to form a voting classifier, and apply to the `MNIST` dataset.

5 Logistic regression

Consider a set of training data $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots$, where $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_n^{(i)})$ is a n-dim vector, and $y^{(i)}$ is a real number. We would like to use Linear regression to find the relation between x and y.

In this case, we assume that y is a linear function of x:

$$y = \theta_0 + \sum_{j=1}^{n} \theta_j x_j.$$

The purpose of Linear regression is to used the given training data to find out the best $\Theta = (\theta_0, \theta_1, \theta_2, \ldots, \theta_n)$.

If we set $\hat{x} = (1, x_1, \ldots, x_n)$, then the above formula can be reformulated by matrix multiplication.

$$y = \Theta \hat{x}^T.$$

When we want to deal with classification problem, we may still use this regression idea, but we have to do some modification.

5.1 Basic idea

The Logsitic regression is used to predict the probability of a data point belonging to a specific class. It is based on linear regression. The major difference is that logistic regreesion will have an activation function σ at the final stage to change the predicted values of the linear regression to the values that indicate classes. In the case of binary classification, the outcome of σ will be between 0 and 1, which is related to the two classes respectively. In this case, the number is interepted as the probability of the data to be in one of the specific class.

The model for Logistic regression is as follows:

$$p = \sigma(L(x)) = \sigma\left(\theta_0 + \sum_{j=1}^{n} \theta_j x_j\right) = \sigma\left(\Theta \hat{x}^T\right).$$

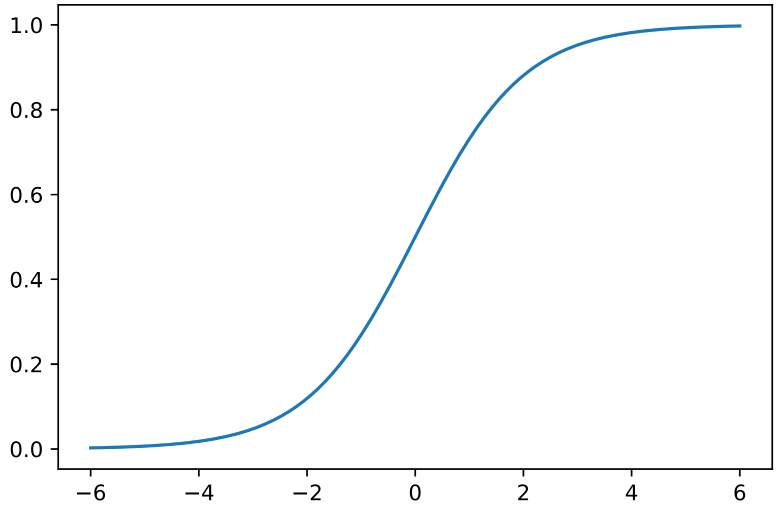
In most cases, this activation function is chosen to be the Sigmoid funciton.

5.1.1 Sigmoid function

The *Sigmoid* function is defined as follows:

$$\sigma(z) = \frac{1}{1 + \mathrm{e}^{-z}}.$$

The graph of the function is shown below.



The main properties of σ are listed below as a Lemma.

Lemma 5.1. *The Sigmoid function $\sigma(z)$ satisfies the following properties.*

1. $\sigma(z) \to \infty$ *when* $z \mapsto \infty$.
2. $\sigma(z) \to -\infty$ *when* $z \mapsto -\infty$.
3. $\sigma(0) = 0.5$.

4. $\sigma(z)$ is always increasing.
5. $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

Solution. We will only look at the last one.

$$\begin{aligned}\sigma'(z) &= -\frac{(1+\mathrm{e}^{-z})'}{(1+\mathrm{e}^{-z})^2} = \frac{\mathrm{e}^{-z}}{(1+\mathrm{e}^{-z})^2} = \frac{1}{1+\mathrm{e}^{-z}}\frac{\mathrm{e}^{-z}}{1+\mathrm{e}^{-z}} \\ &= \sigma(z)\left(\frac{1+\mathrm{e}^{-z}}{1+\mathrm{e}^{-z}} - \frac{1}{1+\mathrm{e}^{-z}}\right) = \sigma(z)(1-\sigma(z)).\end{aligned}$$

5.1.2 Gradient descent

Assume that we would like to minimize a function $J(\Theta)$, where this Θ is an N-dim vector. Geometricly, we could treat J as a height function, and it tells us the height of the mountain. Then to minimize J is the same thing as to find the lowest point. One idea is to move towards the lowest point step by step. During each step we only need to lower our current height. After several steps we will be around the lowest point.

The geometric meaning of ∇J is the direction that J increase the most. Therefore the opposite direction is the one we want to move in. The formula to update x is

$$\Theta_{\text{new}} = \Theta_{\text{old}} - \alpha \nabla J(\Theta_{\text{old}}),$$

where α is called the *learning rate* which controls how fast you want to learn. Usually if α is small, the learning tends to be slow and stble, and when α is big, the learning tends to be fast and unstable.

In machine learning, in most cases we would like to formulate the problem in terms of finding the lowest point of a *cost function* $J(\Theta)$. Then we could start to use Logistic regression to solve it. For binary classification problem, the cost function is defined to be

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log(p^{(i)}) + (1-y^{(i)})\log(1-p^{(i)})\right].$$

Here m is the number of data points, $y^{(i)}$ is the labelled result (which is either 0 or 1), $p^{(i)}$ is the predicted value (which is between 0 and 1).

ℹ Note

The algorithm gets its name since we are using the gradient to find a direction to lower our height.

5.1.3 The Formulas

Theorem 5.1. *The gradient of J is computed by*

$$\nabla J = \frac{1}{m}(\boldsymbol{p} - \boldsymbol{y})^T \hat{\boldsymbol{X}}. \tag{5.1}$$

Click for details.

Proof. The formula is an application of the chain rule for the multivariable functions.

$$\begin{aligned}
\frac{\partial p}{\partial \theta_k} &= \frac{\partial}{\partial \theta_k}\sigma\left(\theta_0 + \sum_{j=1}^{n}\theta_j x_j\right) = \frac{\partial}{\partial \theta_k}\sigma(L(\Theta)) \\
&= \sigma(L)(1-\sigma(L))\frac{\partial}{\partial \theta_k}\left(\theta_0 + \sum_{j=1}^{n}\theta_j x_j\right) \\
&= \begin{cases} p(1-p) & \text{if } k = 0, \\ p(1-p)x_k & \text{otherwise.} \end{cases}
\end{aligned}$$

Then

$$\nabla p = \left(\frac{\partial p}{\partial \theta_0}, \ldots, \frac{\partial p}{\partial \theta_n}\right) = p(1-p)\hat{x}.$$

Then

$$\nabla \log(p) = \frac{\nabla p}{p} = \frac{p(1-p)\hat{x}}{p} = (1-p)\hat{x}.$$

$$\nabla \log(1-p) = \frac{-\nabla p}{1-p} = -\frac{p(1-p)\hat{x}}{1-p} = -p\hat{x}.$$

Then

$$\begin{aligned}
\nabla J &= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\nabla \log(p^{(i)}) + (1-y^{(i)})\nabla \log(1-p^{(i)})\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}(1-p^{(i)})\hat{x}^{(i)} + (1-y^{(i)})(-p^{(i)}\hat{x}^{(i)})\right] \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left[(y^{(i)}-p^{(i)})\hat{x}^{(i)}\right].
\end{aligned}$$

We write $\hat{x}^{(i)}$ as row vectors, and stack all these row vectors vertically. What we get is a matrix $\hat{\mathbf{X}}$ of the size $m \times (1+n)$. We stack all $y^{(i)}$ (resp. $p^{(i)}$) vectically to get the m-dim column vector $\mathbf{y}$ (resp. $\mathbf{p}$).

Using this notation, the previous formula becomes

$$\nabla J = \frac{1}{m}(\mathbf{p}-\mathbf{y})^T\hat{\mathbf{X}}.$$

After the gradient can be computed, we can start to use the gradient descent method. Note that, although Θ are not explicitly presented in the formula of ∇J, this is used to modify Θ:

$$\Theta_{s+1} = \Theta_s - \alpha\nabla J.$$

□

5.1.4 Codes

We will only talk about using packages. `sklearn` provides two methods to implement the Logistic regression. The API interface is very similar to other models.

Note that Logistic regression is very sensitive to the scale of features. Therefore we need to normalize the features before throwing them into the model.

Let's still take `iris` as an example.

```python
from sklearn import datasets
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

The first method is `sklearn.linear_model.LogisticRegression`.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

steps = [('normalize', MinMaxScaler()),
         ('log', LogisticRegression())]

log_reg = Pipeline(steps=steps)
log_reg.fit(X_train, y_train)
log_reg.score(X_test, y_test)
```

```
1.0
```

Note that this method has an option `solver` that will set the way to solve the Logistic regression problem, and there is no "stochastic gradient descent" provided. The default solver for this `LogsiticRegression` is `lbfgs` which will NOT be discussed in lectures.

The second method is `sklearn.linear_model.SGDClassifier`.

```
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

steps = [('normalize', MinMaxScaler()),
         ('log', SGDClassifier(loss='log_loss', max_iter=100))]

sgd_clf = Pipeline(steps=steps)
sgd_clf.fit(X_train, y_train)
sgd_clf.score(X_test, y_test)
```

```
1.0
```

This method is the one we discussed in lectures. The `log_loss` loss function is the binary entropy function we mentioned in lectures. If you change to other loss functions it will become other models.

From the above example, you may notice that `SGDClassifier` doesn't perform as well as `LogisticRegression`. This is due to the algorithm. To make `SGDClassifier` better you need to tune the hyperparameters, like `max_iter`, `learning_rate`/`alpha`, `penalty`, etc..

> ℹ Note
>
> The argument `warm_start` is used to set whether you want to use your previous model. When set to `True`, it will reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. The default is `False`.
> Repeatedly calling `fit` when `warm_start` is `True` can result in a different solution than when calling `fit` a single time because of the way the data is shuffled.

> ℹ Note
>
> Note that for both methods, regularization (which will be discussed later) is applied by default.

5.1.5 Several important side topics

5.1.5.1 Epochs

We use epoch to describe feeding data into the model. One *Epoch* is when an entire dataset is passed through the model once. When using gradient descent, we tend to run several epochs. The number of maximal epochs is one important hyperparameter of this model.

The general idea is that more epochs are better for the score of the model, but it will definitely be slower. In addition, sometimes due to many other factors, after a few epochs, the model becomes stall. To train for more epochs cannot improve the model. In this case you have to turn to other methods.

5.1.5.2 Batch Gradient Descent vs SGD vs Minibatch

Recall the Formula Equation 5.1:

$$\nabla J = \frac{1}{m}(\mathbf{p}-\mathbf{y})^T\hat{\mathbf{X}}.$$

We could rewrite this formula:

$$\nabla J = \frac{1}{m}(\mathbf{p}-\mathbf{y})^T\hat{\mathbf{X}} = \frac{1}{m}\sum_{i=1}^{m}\left[(p^{(i)}-y^{(i)})\hat{x}^{(i)}\right].$$

This new formula can be understood in the following way: For every data point, we could get one gradient direction. Then ∇J is the average of all gradient directions. So this algorithm can be expressed as that compute the gradient for every data points and then take the average, and finally update the parameters once. This algorithm is called *batch gradient descent*.

Following the idea, there is another way to update the model. For every data point, we could compute one gradient direction, and we could use the gradient direction to update the parameters of the model. This algorithm is called *stochastic gradient descent*.

Then there is an algrothm living in the middle, called *mini-batch gradient descent*. In this case, we will group the data set into a collection of subsets of a fiexed number of training examples. Each subset is called a *mini-batch*, and the fixed number of elements of each mini-batch is called the *batch size*. Using this method, we will just go through mini-batches one at a time, compute the average of the gradient for these data, and then update the parameters of the model after we finish one mini-batch. Assume that the total number of the dataset is `N`, the mini-batch size is `m`. Then there are `N/m` mini-batches, and during one epoch we will update the model `N/m` times.

Mini-batch size is one important hyperparameters of this model. Usually the larger the batch size is, the less variance the model has. Then it tends to behave more smoothly, but it will also be slower, and might be stuck to a local minimal. The smaller batch size is more chaotic. It might go faster, but it tends not to converge.

5.2 Regularization

5.2.1 Three types of errors

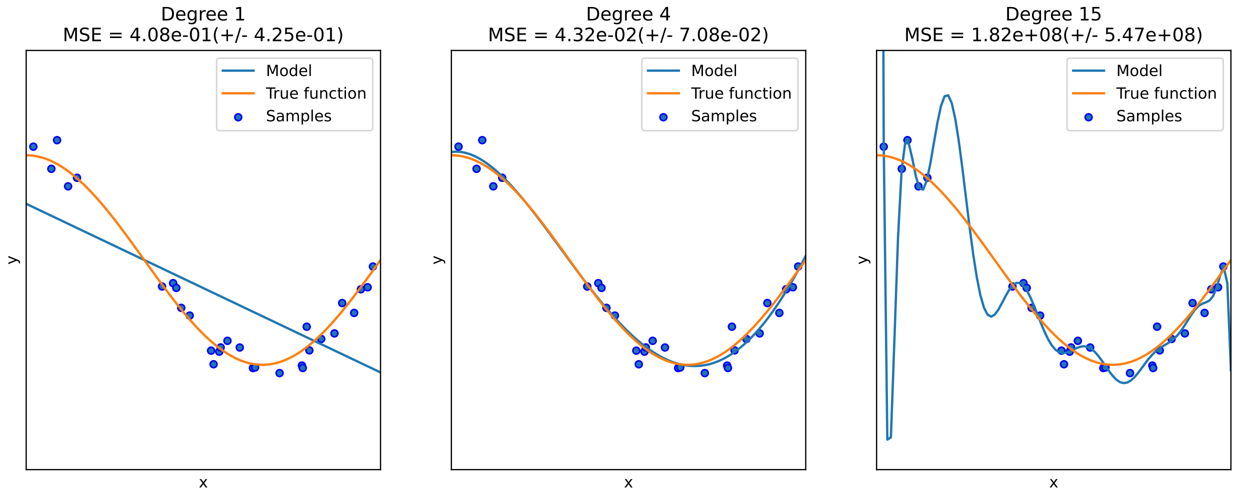
Every estimator has its advantages and drawbacks. Its generalization error can be decomposed in terms of bias, variance and noise. The **bias** of an estimator is its average error for different training sets. The **variance** of an estimator indicates how sensitive it is to varying training sets. Noise is a property of the data.

5.2.2 Underfit vs Overfit

When fit a model to data, it is highly possible that the model is underfit or overfit.

Roughly speaking, **underfit** means the model is not sufficient to fit the training samples, and **overfit** means that the models learns too many noise from the data. In many cases, high bias is related to underfit, and high variance is related to overfit.

The following example is from the `sklearn` guide. Although it is a polynomial regression example, it grasps the key idea of underfit and overfit.



5.2.3 Learning curves (accuracy vs training size)

A learning curve shows the validation and training score of an estimator for varying a key hyperparameter. In most cases the key hyperparameter is the training size or the number of epochs. It is a tool to find out how much we benefit from altering the hyperparameter by training more data or training for more epochs, and whether the estimator suffers more from a variance error or a bias error.

`sklearn` provides `sklearn.model_selection.learning_curve()` to generate the values that are required to plot such a learning curve. However this function is just related to the sample size. If we would like to talk about epochs, we need other packages.

Let us first look at the learning curve about sample size. The official document page is here. The function takes input `estimator`, dataset `X`, `y`, and an arry-like argument `train_sizes`. The dataset `(X, y)` will be split into pieces using the cross-validation technique. The number of pieces is set by the argument cv. The default value is cv=5. For details about cross-validation please see Section 2.2.6.

Then the model is trained over a random sample of the training set, and evaluate the score over the test set. The size of the sample of the training set is set by the argument train_sizes. This argument is array-like. Therefore the process will be repeated several times, and we can see the impact of increasing the training size.

The output contains three pieces. The first is train_sizes_abs which is the number of elements in each training set. This output is mainly for reference. The difference between the output and the input train_sizes is that the input can be float which represents the percentagy. The output is always the exact number of elements.

The second output is train_scores and the third is test_scores, both of which are the scores we get from the training and testing process. Note that both are 2D numpy arrays, of the size (number of different sizes, cv). Each row is a 1D numpy array representing the cross-validation scores, which is corresponding to a train size. If we want the mean as the cross-validation score, we could use train_scores.mean(axis=1).

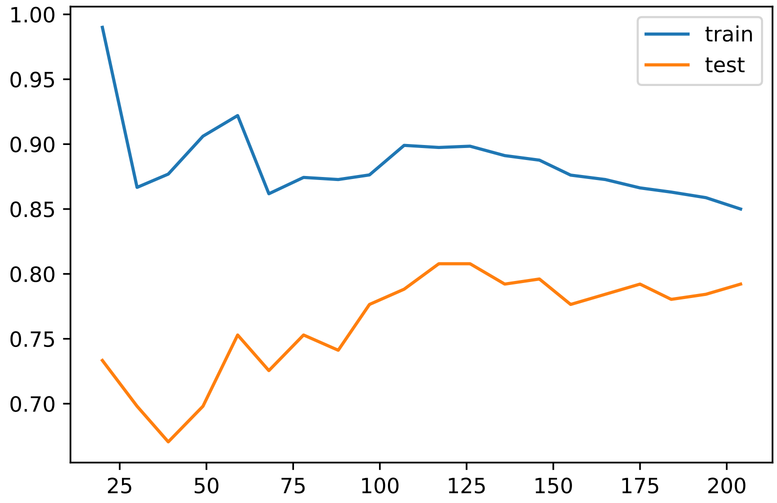
After understanding the input and output, we could plot the learning curve. We still use the horse colic as the example. The details about the dataset can be found in Section A.3.

```
import pandas as pd
import numpy as np

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/horse-colic/horse-colic.data'
df = pd.read_csv(url, delim_whitespace=True, header=None)
df = df.replace("?", np.NaN)

df.fillna(0, inplace=True)
df.drop(columns=[2, 24, 25, 26, 27], inplace=True)
df[23].replace({1: 1, 2: 0}, inplace=True)
X = df.iloc[:, :-1].to_numpy().astype(float)
y = df[23].to_numpy().astype(int)
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)
```

We use the model `LogisticRegression`. The following code plot the learning curve for this model.

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import Pipeline

clf = LogisticRegression(max_iter=1000)
steps = [('scalar', MinMaxScaler()),
         ('log', clf)]
pipe = Pipeline(steps=steps)
from sklearn.model_selection import learning_curve
import numpy as np
train_sizes, train_scores, test_scores = learning_curve(pipe, X_train, y_train,
                                                        train_sizes=np.linspace(0.1, 1, 20))

import matplotlib.pyplot as plt
plt.plot(train_sizes, train_scores.mean(axis=1), label='train')
plt.plot(train_sizes, test_scores.mean(axis=1), label='test')
plt.legend()
```



The learning curve is a primary tool for us to study the bias and variance. Usually

- If the two training curve and the testing curve are very close to each other, this means that the variance is low. Otherwise the variance is high, and this means that the model probabily suffer from overfitting.
- If the absolute training curve score is high, this means that the bias is low. Otherwise the bias is high, and this means that the model probabily suffer from underfitting.

In the above example, although regularization is applied by default, you may still notice some overfitting there.

5.2.4 Regularization

Regularization is a technique to deal with overfitting. Here we only talk about the simplest method: ridge regression, also known as Tikhonov regularizaiton. Because of the formula given below, it is also called L_2 *regularization*. The idea is to add an additional term $\frac{\alpha}{2m}\sum_{i=1}^{m}\theta_i^2$ to the original cost function. When training with the new cost function, this additional term will force the parameters in the original term to be as small as possible. After finishing training, the additional term will be dropped, and we use the original cost function for validation and testing. Note that in the additional term θ_0 is not presented.

The hyperparameter α is the *regularization strength*. If $\alpha = 0$, the new cost function becomes the original one; If α is very large, the additional term dominates, and it will force all parameters to be almost 0. In different context, the regularization strength is also given by $C = \frac{1}{2\alpha}$, called *inverse of regularization strength*.

5.2.4.1 The math of regularization

Theorem 5.2. *The gradient of the ridge regression cost function is*

$$\nabla J = \frac{1}{m}(\boldsymbol{p}-\boldsymbol{y})^T\hat{\boldsymbol{X}}+\frac{\alpha}{m}\Theta.$$

Note that Θ doesn't contain θ_0, or you may treat $\theta_0=0$.

The computation is straightforward.

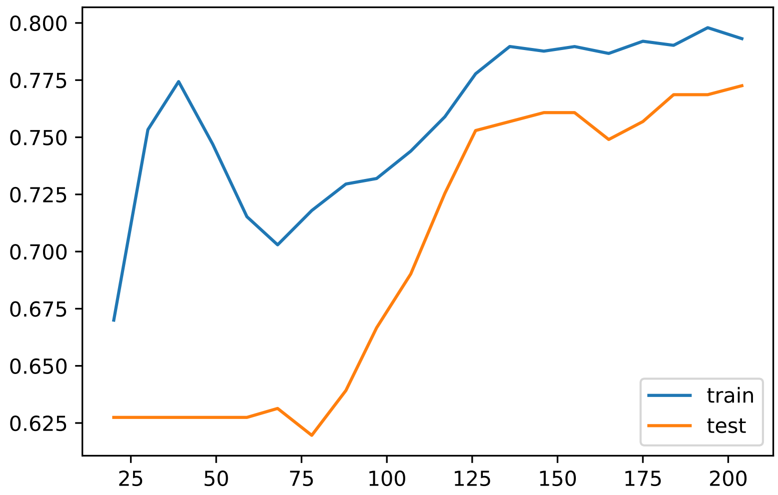
5.2.4.2 The code

Regularization is directly provided by the logistic regression functions.

- In `LogisticRegression`, the regularization is given by the argument `penalty` and `C`. `penalty` specifies the regularizaiton method. It is `l2` by default, which is the method above. `C` is the inverse of regularization strength, whose default value is `1`.
- In `SGDClassifier`, the regularization is given by the argument `penalty` and `alpha`. `penalty` is the same as that in `LogisticRegression`, and `alpha` is the regularization strength, whose default value is `0.0001`.

Let us see the above example.

```
clf = LogisticRegression(max_iter=1000, C=0.1)
steps = [('scalar', MinMaxScaler()),
         ('log', clf)]
pipe = Pipeline(steps=steps)
from sklearn.model_selection import learning_curve
import numpy as np
train_sizes, train_scores, test_scores = learning_curve(pipe, X_train, y_train,
                                                        train_sizes=np.linspace(0.1, 1, 20))

import matplotlib.pyplot as plt
plt.plot(train_sizes, train_scores.mean(axis=1), label='train')
plt.plot(train_sizes, test_scores.mean(axis=1), label='test')
plt.legend()
```



After we reduce `C` from `1` to `0.1`, the regularization strength is increased. Then you may find that the gap between the two curves are reduced. However the overall performace is also reduced, from 85%~90% in `C=1` case to around 80% in `C=0.1` case. This means that the model doesn't fit the data well as the previous one. Therefore this is a trade-off: decrease the variance but increase the bias.

5.3 Neural network implement of Logistic regression

In the previous sections, we use gradient descent to run the Logistic regression model. We mentioned some important concepts, like epochs, mini-batch, etc.. But we didn't implement them. In fact `sklearn` doesn't provide a very good tool to do all these computations. Hence we turn to another package for this model. We are going to use `keras` to build a Logistic regression model, and plot the "loss vs epochs" learning curves.

`keras` is high level Neural network library built upon `tensorflow`. We will talk about Neural network next Chapter. The package is installed along with `tensorflow`. You may follow the instructions to install it.

To use keras to implement logistic regression, we need the following modules: a Sequential model, a Dense layer. The model is organized as follows.

We still use the horse colic dataset as an example.

```
import pandas as pd
import numpy as np

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/horse-colic/horse-colic.data'
df = pd.read_csv(url, delim_whitespace=True, header=None)
df = df.replace("?", np.NaN)

df.fillna(0, inplace=True)
df.drop(columns=[2, 24, 25, 26, 27], inplace=True)
df[23].replace({1: 1, 2: 0}, inplace=True)
X = df.iloc[:, :-1].to_numpy().astype(float)
y = df[23].to_numpy().astype(int)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)
```

Note that we need to perform normalization before throwing the data into the model. Here we use the MinMaxScaler() from sklearn package. The normalization layer in keras is a little bit more complicated and doesn't fit into situation.

```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
mms.fit(X_train)
X_train = mms.transform(X_train)
X_test = mms.transform(X_test)
```

In the following code, we first set up the model, and then add one Dense layer. This Dense layer means that we would perform a linear transformation to the input, by the formula $\theta_0+\theta_1x_1+\theta_2x_2+...+\theta_nx_n$. Then there are three arguments:

- 1: means that there is only output.

- `activation='sigmoid'`: means that we will apply the `sigmoid` function after the linear transformation.
- `input_dim`: means the dimension of the input. Note that this dimension is the dimension of one individual data point. You don't take the size of the training set into consideration.

After building the basic architectal of the model, we need to speicify a few more arguments. In the `model.compile()` step, we have to input the `optimizer`, the loss function (which is the `binary_crossentropy` in our case) and the metrics to test the performance of the model (which is accuracy in our case).

The `optimizer` is how the parameters are updated. The best choice in general is `adam`. The default setting is `RMSprop` and the optimizer discussed in our lecture is `sgd`. We will use `adam` here, since the learning curve it produces looks better (for illustration).

Finally we could train the model. The argument is straightforward.

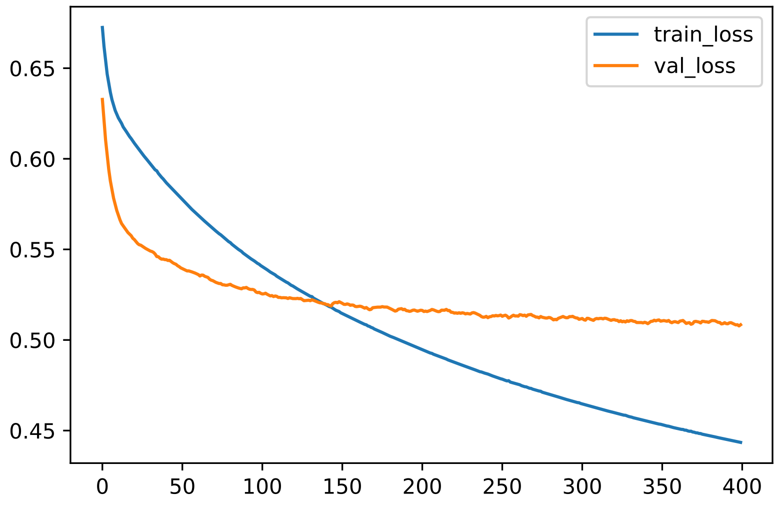
```
import keras_core as keras
from keras import models, layers
model = models.Sequential()

model.add(layers.Dense(1, activation='sigmoid', input_dim=X_train.shape[1]))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(X_train, y_train, epochs=400, batch_size=30, validation_data=(X_test, y_test))
```

Note that we assign the output of `model.fit()` to a variable `hist`. The infomation about this training process is recorded inside. We will extract those information.

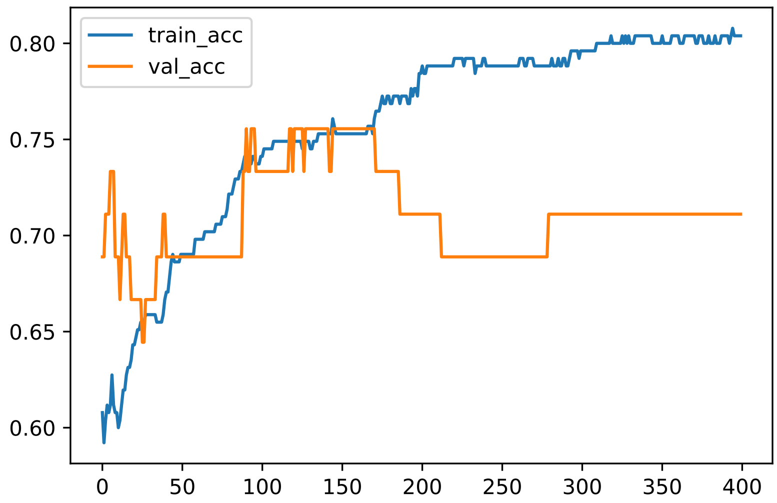
```
loss_train = hist.history['loss']
loss_val = hist.history['val_loss']

acc_train = hist.history['accuracy']
acc_val = hist.history['val_accuracy']
```

We now could plot the learning curve (loss vs epochs) and the learning curve (accuracy vs epochs).

```
import matplotlib.pyplot as plt
plt.plot(loss_train, label='train_loss')
plt.plot(loss_val, label='val_loss')
plt.legend()
```



```
plt.plot(acc_train, label='train_acc')
plt.plot(acc_val, label='val_acc')
plt.legend()
```



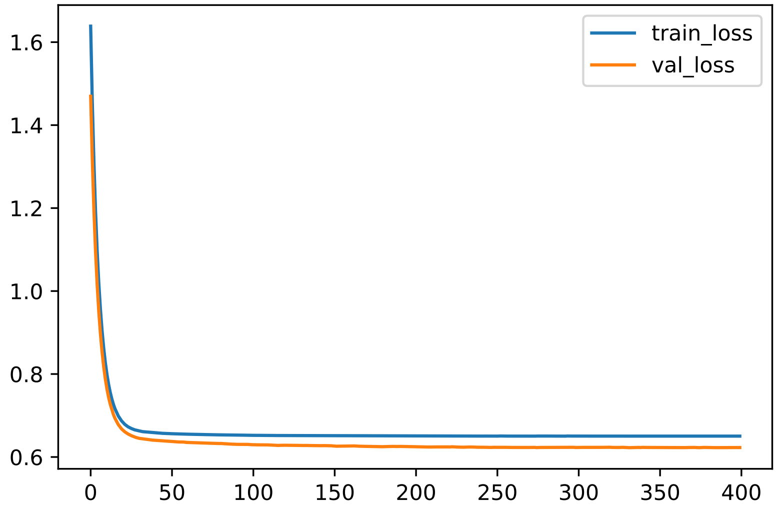
5.3.1 Regularization

To apply regularization, we just need to modify the layer we added to the model. The argument is `kernel_regularizer`. We would like to set it to be `keras.regularizers.L2(alpha)`, where `alpha` is the regularization strength.

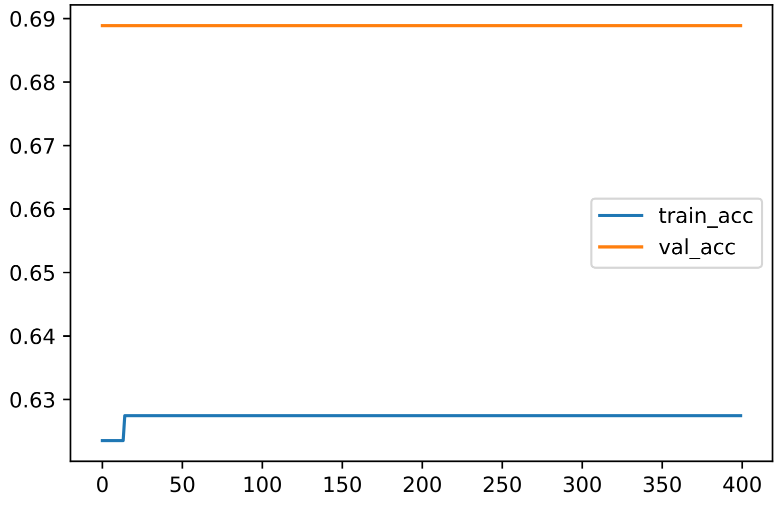
```
from keras import models, layers, regularizers

model = models.Sequential()

model.add(layers.Dense(1, activation='sigmoid', input_dim=X_train.shape[1],
                       kernel_regularizer=regularizers.L2(0.5)))

model.compile(optimizer='sgd', loss='binary_crossentropy', metrics=['accuracy'])
hist = model.fit(X_train, y_train, epochs=400, batch_size=30,
                 validation_data=(X_test, y_test))
```

```
loss_train = hist.history['loss']
loss_val = hist.history['val_loss']

acc_train = hist.history['accuracy']
acc_val = hist.history['val_accuracy']
```

```
plt.plot(loss_train, label='train_loss')
plt.plot(loss_val, label='val_loss')
plt.legend()
```



```
plt.plot(acc_train, label='train_acc')
plt.plot(acc_val, label='val_acc')
plt.legend()
```



You may compare what we get here with the codes we get before.

5.4 Multi class case

5.4.1 Naive idea (one-vs-all)

Assume that there are N classes. The naive idea is to decompose this N-class classification problem into N binary classification problems. The model will contains N classifiers. The ith classifer is used to classify whehter the given features has the label `i` or not.

For example, asusme we are dealing with the `iris` dataset. There are 3 classes. By the naive idea, we will modify the labels into `Setosa` and `not Setosa`, and use it to train the first classifier. Similarly we can have two more classifiers to tell `Versicolour/not Versicolour` and `Virginica/not Virginica`. Then we combine all three classifiers to get a final classifier to put the data into one of the three classes.

5.4.2 `Softmax` function

A better method is mimic the sigmoid function. Recall that in binary classification problem, after

$$z = L(x) = \theta_0 + \sum_{i=1}^{n} \theta_i x_i,$$

the sigmoid function is applied $p = \sigma(z)$. This p is interepreted as the probability for the data belonging to class 1. For N-class problem, we could generalize the sigmoid function to `softmax` function, whose value is a N-dimensional vector $p = [p_k]_{i=1}^{N}$. Here p_k represents the probability for the data belonging to class k. Then after we get the vector p, we then find the highest probability and that indicates the class of the data point.

The `softmax` function is defined in the following way:

$$p_k = \sigma(z) = \frac{\exp(z_k)}{\sum_{i=1}^{N}\exp(z_i)}, \qquad \text{for } z = [z_1, z_2, \ldots, z_N].$$

In the model, each $z_i = L_i(x) = \theta_0^{(i)} + \sum_{i=1}^{n} \theta_i^{(i)} x_i$, has its own weights.

The related cost function is also updated:

$$J(\Theta) = -\sum_{i=1}^{N} y_k \ln(p_i).$$

Therefore the same gradient descent algorithm can be applied.

> ℹ Note
>
> Note that `sigmoid` function and the `binary crossentropy` cost functions are the special case of `softmax` function.

> ℹ Note
>
> The labels are not recorded as labels, but as vectors. This is called dummy variables, or one-hot encodings.

5.4.3 Codes

- Both `LogisticRegression()` and `SGDClassifier()` by default uses the one-vs-all naive idea.
- Using `kears`, `softmax` can be implemented. The key configuration is the loss function `loss='categorical_crossentropy'` and the activation function `softmax`. Note that in this case the labels should be translated into one-hot vectors.

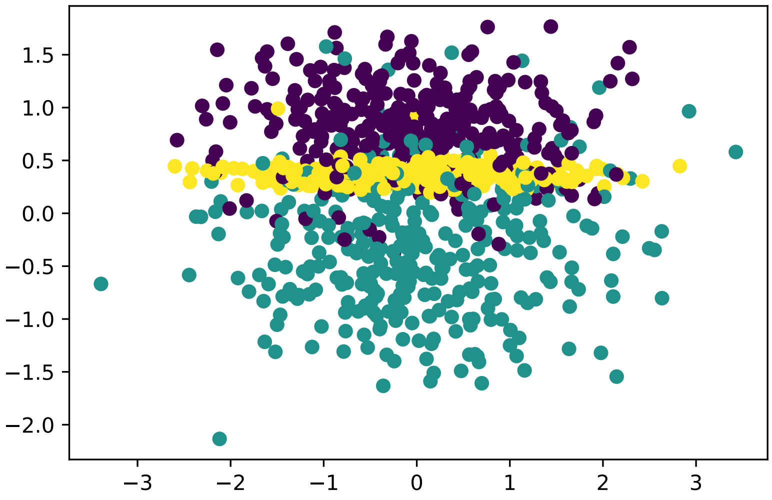
We use `make_classification` as an example. To save time we won't carefully tune the hyperparameters here.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
```

```
X, y = make_classification(n_samples=1000, n_features=10, n_informative=2, n_redundant=2, n_rep
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

Although there are totally 10 features, the dataset can be visualized using the informative features. By description, the informative features are the first two.

```
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], c=y)
```



```
from sklearn.linear_model import LogisticRegression

clf = LogisticRegression()
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

```
0.8466666666666667
```

```
from sklearn.linear_model import SGDClassifier

clf = SGDClassifier()
clf.fit(X_train, y_train)
clf.score(X_test, y_test)
```

```
0.8266666666666667
```

To apply `keras` package, we should first change `y` into one-hot vectors. Here we use the function provided by `keras`.

```python
import keras_core as keras
from keras.utils import to_categorical
from keras import models, layers

vy_train = to_categorical(y_train, num_classes=3)
vy_test = to_categorical(y_test, num_classes=3)

model = models.Sequential()
model.add(layers.Dense(3, activation='softmax', input_dim=10))
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

model.fit(X_train, vy_train, epochs=50, batch_size=50, verbose=0)
_ = model.evaluate(X_test, vy_test)
```

5.5 Exercises and Projects

Exercise 5.1. Please hand write a report about the details of the math formulas for Logistic regression.

Exercise 5.2. CHOOSE ONE: Please use `sklearn` to apply the LogisticRegression to one of the following datasets. You may either use `LogisticRegression` or `SGDClassifier`. - the `iris` dataset. - the dating dataset. - the `titanic` dataset.

Please in addition answer the following questions. 1. What is your accuracy score? 2. How many epochs do you use? 3. Plot the learning curve (accuracy vs training sizes).

Exercise 5.3. CHOOSE ONE: Please use `keras` to apply the LogisticRegression to one of the following datasets. - the `iris` dataset. - the dating dataset. - the `titanic` dataset.

Please in addition answer the following questions. 1. What is your accuracy score? 2. How many epochs do you use? 3. What is the batch size do you use? 4. Plot the learning curve (loss vs epochs, accuracy vs epochs). 5. Analyze the bias / variance status.

A Datasets

A.1 the iris dataset

```python
from sklearn import datasets
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

A.2 The breast_cancer dataset

```python
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()
X = cancer.data
y = cancer.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

A.3 the Horse Colic Dataset

The data is from the UCI database. The data is loaded as follows. ? represents missing data.

```
import pandas as pd
import numpy as np

url = 'http://archive.ics.uci.edu/ml/machine-learning-databases/horse-colic/horse-colic.data'
df = pd.read_csv(url, delim_whitespace=True, header=None)
df = df.replace("?", np.NaN)
```

The description of the data is listed here. We will preprocess the data according to the descrption.

1. The data tries to predict Column 24. Since Python index starts from 0, in our case we are interested in Column 23.
2. Column 25-27 (in our case is Column 24-26) use a special code to represent the type of lesion. For simplicity we remove these three columns.
3. Column 28 (in our case Column 27) is of no significance so we will remove it too.
4. Column 3 (in our case Column 2) is the IDs of Hospitals which should have very little impact so we will remove it too.
5. We will fill the missing values with `0`.
6. We also would like to change the label from `1` and `2` to `0` and `1` for the purpose of Logistic regression.

This part should be modified if you want to improve the performance of your model.

```
df.fillna(0, inplace=True)
df.drop(columns=[2, 24, 25, 26, 27], inplace=True)
df[23].replace({1: 1, 2: 0}, inplace=True)
X = df.iloc[:, :-1].to_numpy().astype(float)
y = df[23].to_numpy().astype(int)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

A.4 the Dating dataset

The data file can be downloaded from here.

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

df = pd.read_csv('datingTestSet2.txt', sep='\t', header=None)
X = np.array(df[[0, 1, 2]])
y = np.array(df[3])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

A.5 The dataset randomly generated

- `make_moon` dataset

This is used to generate two interleaving half circles.

```
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split

X, y = make_moons(n_samples=10000, noise=0.4, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

- `make_gaussian_quantiles` dataset

This is a generated isotropic Gaussian and label samples by quantile.

The following code are from this page. It is used to generate a relative complex dataset by combining two datesets together.

```
from sklearn.datasets import make_gaussian_quantiles
from sklearn.model_selection import train_test_split

X1, y1 = make_gaussian_quantiles(cov=2.0, n_samples=200, n_features=2,
                                 n_classes=2, random_state=1)
X2, y2 = make_gaussian_quantiles(mean=(3, 3), cov=1.5, n_samples=300,
                                 n_features=2, n_classes=2, random_state=1)
X = np.concatenate((X1, X2))
y = np.concatenate((y1, -y2 + 1))
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

It can also be used to generate multiclass dataset.

```
from sklearn.datasets import make_gaussian_quantiles
from sklearn.model_selection import train_test_split

X, y = make_gaussian_quantiles(cov=2.0, n_samples=200, n_features=2,
                               n_classes=4, random_state=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

- `make_classification`

This will create a multiclass dataset. Without shuffling, `X` horizontally stacks features in the following order: the primary `n_informative` features, followed by `n_redundant` linear combinations of the informative features, followed by `n_repeated` duplicates, drawn randomly with replacement from the informative and redundant features.

For more details please see the official document.

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

X, y = make_classification(n_samples=1000, n_features=10, n_informative=2, n_redundant=2, n_rep
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15)
```

A.6 `MNIST` dataset

There are several versions of the dataset.

- `tensorflow` provides the data with the original split.

```
import keras_core as keras
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
```

```
Using TensorFlow backend
```

A.7 titanic dataset

This is the famuous Kaggle101 dataset. The original data can be download from the Kaggle page. You may also download the {Download}training data<./assests/datasets/titanic/train.csv> and the {Download}test data<./assests/datasets/titanic/test.csv> by click the link.

```
import pandas as pd
dftrain = pd.read_csv('train.csv')
dftest = pd.read_csv('test.csv')
```

The original is a little bit messy with missing values and mix of numeric data and string data. The above code reads the data into a DataFrame. The following code does some basic of preprocess. This part should be modified if you want to improve the performance of your model.

1. Only select columns: Pclass, Sex, Age, SibSp, Parch, Fare. That is to say, Name, Cabin and Embarked are dropped.
2. Fill the missing values in column Age and Fare by 0.
3. Replace the column Sex by the following map: {'male': 0, 'female': 1}.

```
import pandas as pd
import numpy as np

def getnp(df):
    df['mapSex'] = df['Sex'].map(lambda x: {'male': 0, 'female': 1}[x])
    dfx = df[['Pclass', 'mapSex', 'Age', 'SibSp', 'Parch', 'Fare']].copy()
    dfx['Fare'].fillna(0, inplace=True)
    dfx['Age'].fillna(0, inplace=True)
    if 'Survived' in df.columns:
        y = df['Survived'].to_numpy()
    else:
        y = None
    X = dfx.to_numpy()
    return (X, y)
```

```
X_train, y_train = getnp(dftrain)
X_test, _ = getnp(dftest)
```

For the purpose of submitting to Kaggle, after getting `y_pred`, we could use the following file to prepare for the submission file.

```
def getdf(df, y):
    df['Survived'] = y
    return df[['PassengerId', 'Survived']]

getdf(dftest, y_pred).to_csv('result.csv')
```

A.8 Plant survival data with salt and microbe treatments

This dataset is supported by DART SEED grant. It is provided by Dr. Suresh Subedi from ATU. The dataset is about the outcomes of certain treatments applied to plants. We would like to predict whether the plants survive based on the status of the plants and the treatments. The datafile can be downloaded from here.

We could use the following code to read the data.

```
import pandas as pd

df = pd.read_excel('assests/datasets/plants.xlsx', engine='openpyxl', sheet_name='data')
```

There are a few missing values. The missing values in `Outcome_after 12 months` are all `dead`. These are not recorded as `dead` because the cause of the death is more complicated and needs to be studied separatedly. In our case we could simply fill it with `dead`.

There are two more missing values in `Stem diameter`. For simplicity we drop them directly.

```
df['Outcome_after 12 months'].fillna('dead', inplace=True)
df = df.dropna()
```

Then we would like to transform the data. Here are the rules.

- `Endophyte`: `I+`->`1`, `I-`->`-1`
- `Treatment`: `Salt`->`1`, `Fresh`->`0`
- `Tree_Replicate`: `T1`->`1`, `T2`->`2`, `T3`->`3`
- `Outcome_after 12 months`: `survived`->1, `dead`->0

Column `SN` will be dropped.

Finally we put these together to get the features `X` and the label `y`.

```
df['Endophyte '] = df['Endophyte '].map({'I+': 1, 'I-': -1})
df['Treatment'] = df['Treatment'].map({'Fresh': 0, 'Salt': 1})
df['Tree_Replicate'] = df['Tree_Replicate'].str[1].astype(int)
df['Outcome_after 12 months'] = df['Outcome_after 12 months'].map({'survived': 1, 'dead': 0})

X = df.iloc[:, 1: -1].to_numpy()
y = df['Outcome_after 12 months'].to_numpy()
```