Overview

A. What is Machine Learning?

Machine learning is the branch of science that deals with algorithms and systems performing specific tasks using patterns and inference, rather than explicitly programmed instructions. There are a variety of different use cases for machine learning, from image recognition to text generation. Most machine learning tasks generalize to one of the following two learning types:

- **Supervised learning**: Using labeled data to train a model. The labels for the training dataset represent the class/category that each data observation belongs to. After training, the model should be able to predict labels for new data observations (from the same population distribution as the training data).
    - Example: Let's say you're training a machine learning model to predict whether a picture contains a lake or not. With supervised learning, you would train a model on a dataset of pictures where the label for each picture is "Yes" if it contains a lake or "No" if it doesn't. After training, the model will be able to take in a picture and determine whether or not it contains a lake.
- **Unsupervised Learning**: Using *unlabeled* data to allow a model to learn relationships between data observations and pick up on underlying patterns. Most data in the world is unlabeled, which makes unsupervised learning a very useful method of machine learning.
    - Example: Going back to the same picture dataset from above, but now assume the training dataset is unlabeled. Using unsupervised learning, a model will be able to pick up on the inherent differences between pictures with a lake and pictures without a lake, e.g. differences in pixel color or orientation. This allows the model to cluster the pictures into two separate groups.

If it is possible to get large enough labeled training datasets, supervised learning is the way to go. However, it is often difficult to get fully labeled datasets, which is why many tasks require unsupervised learning or semi-supervised learning (a mix of supervised and unsupervised learning). Deciding which type of learning method to use is only the first step towards creating a machine learning model. You also need to choose the proper model architecture for your task and, most importantly, be able to process data into a training pipeline and interpret/analyze model results.

B. ML vs. AI vs. Data Science

People often throw around the terms “machine learning”, “artificial intelligence”, and “data science” interchangeably. In reality, machine learning is a subset of artificial intelligence and overlaps heavily with data science. Artificial intelligence deals with any technique that allows machines to display “intelligence”, similar to humans. Machine learning is one of the main techniques used to create artificial intelligence, but other non-ML techniques (e.g. alpha-beta pruning, rule-based systems) are also widely used in AI.

On the other hand, data science deals with gathering insights from datasets. Traditionally, data scientists have used statistical methods for gathering these insights. However, as machine learning continues to grow, it has also penetrated into the field of data science.

In industry, any data scientist or AI researcher needs to have a good understanding of machine learning. Machine learning in industry has allowed us to create wonderful autonomous systems. These systems have matched, or sometimes even exceeded, the best human performance in their respective fields. A good example is AlphaGo, a machine-learning based system that has beaten the best human Go players in the world.

C. 7 Steps of the Machine Learning Process

1. Data Collection: The process of extracting raw datasets for the machine learning task. This data can come from a variety of places, ranging from open-source online resources to paid crowdsourcing. The first step of the machine learning process is arguably the most important. If the data you collect is poor quality or irrelevant, then the model you train will be poor

quality as well.

2. Data Processing and Preparation: Once you've gathered the relevant data, you need to process it and make sure that it is in a usable format for training a machine learning model. This includes handling missing data, dealing with outliers, etc.
3. Feature Engineering: Once you've collected and processed your dataset, you will likely need to transform some of the features (and sometimes even drop some features) in order to optimize how well a model can be trained on the data.
4. Model Selection: Based on the dataset, you will choose which model architecture to use. This is one of the main tasks of industry engineers. Rather than attempting to come up with a completely novel model architecture, most tasks can be thoroughly performed with an existing architecture (or combination of model architectures).
5. Model Training and Data Pipeline: After selecting the model architecture, you will create a data pipeline for training the model. This means creating a continuous stream of batched data observations to efficiently train the model. Since training can take a long time, you want your data pipeline to be as efficient as possible.
6. Model Validation: After training the model for a sufficient amount of time, you will need to validate the model's performance on a held-out portion of the overall dataset. This data needs to come from the same underlying distribution as the training dataset, but needs to be different data that the model has not seen before.
7. Model Persistence: Finally, after training and validating the model's performance, you need to be able to properly save the model weights and possibly push the model to production. This means setting up a process with which new users can easily use your pre-trained model to make predictions.

D. What this course will provide

After taking this course, you'll be able to take process and clean a raw dataset, train a machine learning model on the data, and validate the model's performance. Specifically, you will be able to:

- Take a raw dataset and process it for a given task. This means dealing with missing data and outliers, normalizing and transforming features

with missing data and outliers, normalizing and transforming features,

figuring out which features are the most relevant to the task, and picking out the best combination of features to use.

- Picking the correct model architecture to use based on the data. Many people will always default to using a large neural network for any machine learning task, but many times this is unnecessary and can even hurt the model's final performance if the dataset is not large enough.
- Code a machine learning model and train it on processed data. Validate the model's performance on held-out data and understand techniques to improve a model's performance.

Introduction

An overview of data processing and the NumPy library.

In the **Data Manipulation** section, you will learn how to perform data manipulation using NumPy.

A. Data Processing

When asked about Google's model for success, Peter Norvig, the director of research at Google, famously stated,

> *"We don't have better algorithms than anyone else; we just have more data."*

Though probably an understatement (given the amount of talent employed at Google), the quote does provide a sense of just how vital data is to having successful outcomes.

People normally discuss the importance of data in the context of machine learning. No matter how sophisticated a machine learning model is, it will not perform well unless it has a reasonable amount of data to train on. On the other hand, given a large and diverse set of training data, a good deep learning model will significantly outperform non-deep learning algorithms.

However, data is not just limited to machine learning. Companies use data to identify customer trends, political parties use data to determine which demographics they should target, sports teams use data to analyze players, etc.

Player	PA	AVG	OBP	SLG	OPS	K%	BB/K	ISO	Spd	BABIP	wRC	wRC+	wOBA
Joe Mauer	576	0.261	0.363	0.389	0.752	16.1%	0.85	0.128	3.6	0.301	72	102	0.327
Carlos Santana	688	0.259	0.366	0.498	0.865	14.4%	1.00	0.239	3.5	0.258	110	132	0.37
John Jaso	432	0.268	0.353	0.413	0.766	17.1%	0.61	0.145	3.0	0.314	57	111	0.335
Mike Napoli	645	0.239	0.335	0.465	0.800	30.1%	0.40	0.226	3.7	0.296	89	113	0.343
Albert Pujols	650	0.268	0.323	0.457	0.780	11.5%	0.65	0.189	2.5	0.260	83	111	0.331
Miguel Cabrera	679	0.316	0.393	0.563	0.956	17.1%	0.65	0.247	1.3	0.336	125	152	0.399
Mark Teixeira	438	0.204	0.292	0.362	0.654	24.0%	0.45	0.158	2.4	0.238	40	76	0.287
Joey Votto	677	0.326	0.434	0.550	0.985	17.7%	0.90	0.225	3.9	0.366	133	158	0.413
Billy Butler	274	0.284	0.336	0.416	0.752	15.3%	0.50	0.132	1.1	0.320	34	105	0.324
Ryan Howard	362	0.196	0.257	0.453	0.710	31.5%	0.24	0.257	0.8	0.205	37	83	0.298
James Loney	366	0.265	0.307	0.397	0.703	10.1%	0.43	0.131	1.4	0.275	38	89	0.302
Prince Fielder	370	0.212	0.292	0.334	0.626	17.0%	0.51	0.123	0.9	0.235	31	65	0.276
Adrian Gonzalez	633	0.285	0.349	0.435	0.784	18.5%	0.47	0.150	1.4	0.328	83	112	0.334

Example baseball data used in sabermetrics. The concept was popularized by the 2011 film, *Moneyball*.

The universal usage of data makes *data processing*, the act of converting raw data into a meaningful form, an essential skill to have.

B. NumPy

Many scenarios involve mostly numeric datasets. For example, medical data contains many numeric metrics, such as height, weight, and blood pressure. Furthermore, the majority of neural networks use input data that is either numeric or has been converted to a numeric form.

When we deal with numeric data, the best Python library to use is NumPy. The NumPy library allows us to perform many operations on numeric data, and convert the data to more usable forms.

```
import numpy as np  # import the NumPy library

# Initializing a NumPy array
arr = np.array([-1, 2, 5], dtype=np.float32)

# Print the representation of the array
print(repr(arr))
```

In the following chapters, you'll learn all the necessary NumPy operations for data manipulation.

NumPy Arrays

Learn about NumPy arrays and how they're used.

Chapter Goals:

- Learn about NumPy arrays and how to initialize them
- Write code to create several NumPy arrays

A. Arrays

NumPy arrays are basically just Python lists with added features. In fact, you can easily convert a Python list to a Numpy array using the `np.array` function, which takes in a Python list as its required argument. The function also has quite a few keyword arguments, but the main one to know is `dtype`. The `dtype` keyword argument takes in a NumPy type and manually casts the array to the specified type.

The code below is an example usage of `np.array` to create a 2-D matrix. Note that the array is manually cast to `np.float32`.

```
import numpy as np

arr = np.array([[0, 1, 2], [3, 4, 5]],
               dtype=np.float32)
print(repr(arr))
```

When the elements of a NumPy array are mixed types, then the array's type will be *upcast* to the highest level type. This means that if an array input has mixed `int` and `float` elements, all the integers will be cast to their floating-point equivalents. If an array is mixed with `int`, `float`, and `string` elements, everything is cast to strings.

The code below is an example of `np.array` upcasting. Both integers are cast to

their floating-point equivalents.

```
arr = np.array([0, 0.1, 2])
print(repr(arr))
```

B. Copying

Similar to Python lists, when we make a reference to a NumPy array it doesn't create a different array. Therefore, if we change a value using the reference variable, it changes the original array as well. We get around this by using an array's inherent `copy` function. The function has no required arguments, and it returns the copied array.

In the code example below, `c` is a reference to `a` while `d` is a copy. Therefore, changing `c` leads to the same change in `a`, while changing `d` does not change the value of `b`.

```
a = np.array([0, 1])
b = np.array([9, 8])
c = a
print('Array a: {}'.format(repr(a)))
c[0] = 5
print('Array a: {}'.format(repr(a)))

d = b.copy()
d[0] = 6
print('Array b: {}'.format(repr(b)))
```

C. Casting

We cast NumPy arrays through their inherent `astype` function. The function's required argument is the new type for the array. It returns the array cast to the new type.

The code below shows an example of casting using the `astype` function. The `dtype` property returns the type of an array.

```
arr = np.array([0, 1, 2])
print(arr.dtype)
```

```
arr = arr.astype(np.float32)
print(arr.dtype)
```

D. NaN

When we don't want a NumPy array to contain a value at a particular index, we can use `np.nan` to act as a placeholder. A common usage for `np.nan` is as a filler value for incomplete data.

The code below shows an example usage of `np.nan`. Note that `np.nan` cannot take on an integer type.

```
arr = np.array([np.nan, 1, 2])
print(repr(arr))

arr = np.array([np.nan, 'abc'])
print(repr(arr))

# Will result in a ValueError
np.array([np.nan, 1, 2], dtype=np.int32)
```

E. Infinity

To represent infinity in NumPy, we use the `np.inf` special value. We can also represent negative infinity with `-np.inf`.

The code below shows an example usage of `np.inf`. Note that `np.inf` cannot take on an integer type.

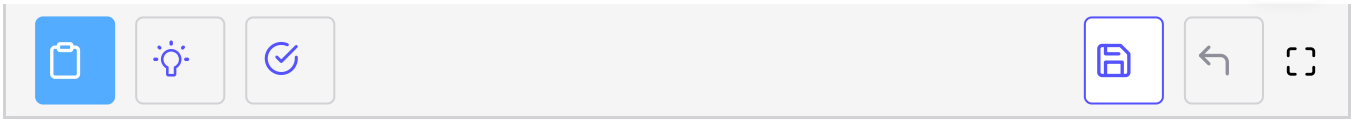
```
print(np.inf > 1000000)

arr = np.array([np.inf, 5])
print(repr(arr))

arr = np.array([-np.inf, 1])
print(repr(arr))

# Will result in an OverflowError
np.array([np.inf, 3], dtype=np.int32)
```

Time to Code!

The first array we'll create comes straight from a list of integers and `np.nan`. The list contains `np.nan` as the first element, and the integers from `2` to `5`, inclusive, as the next four elements.

Set `arr` equal to `np.array` applied to the specified list.

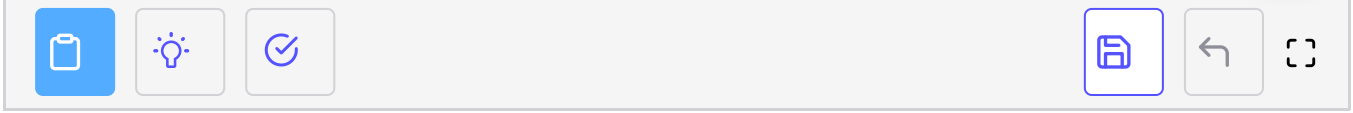
```
# CODE HERE
```



We now want to copy the array so we can change the first element to `10`. This way we don't modify the original array.

Set `arr2` equal to `arr.copy()`, then set the first element of `arr2` equal to `10`.

```
# CODE HERE
```



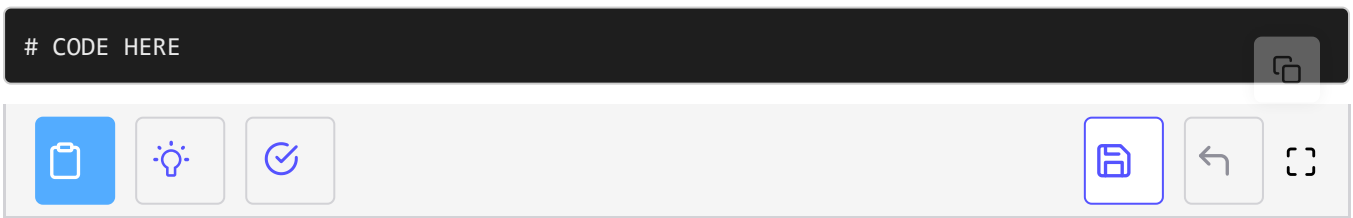
The next two arrays will use floating point numbers. The first array will be upcast to floating point numbers, while we manually cast the second array using `np.float32`.

For manual casting, we use an array's inherent `astype` function, which takes in the new type as an argument and returns the casted array.

Set `float_arr` equal to `np.array` applied to a list with elements `1`, `5.4`, and `3`, in that order.

Set `float_arr2` equal to `arr2.astype`, with argument `np.float32`.

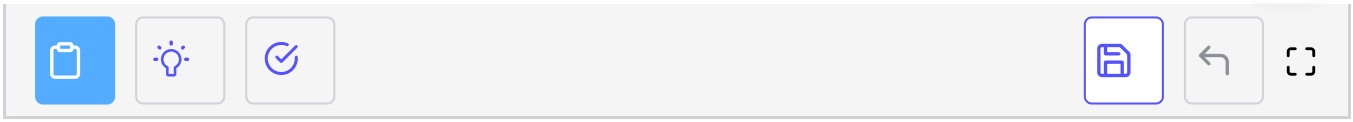
```
# CODE HERE
```



The final array will be a multi-dimensional array, specifically a 2-D matrix. The 2-D matrix will have the integers `1`, `2`, `3` in its first row, and the integers `4`, `5`, `6` in its second row. We'll also manually set its type to `np.float32`.

Set `matrix` equal to `np.array` with a list of lists (representing the specified 2-D matrix) as the first argument, and `np.float32` as the `dtype` keyword argument.

```
# CODE HERE
```



NumPy Basics

Perform basic operations to create and modify NumPy arrays.

Chapter Goals:

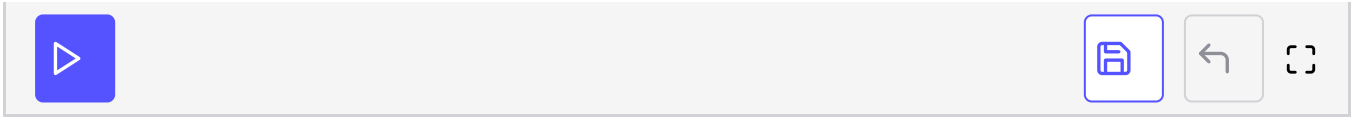
- Learn about some basic NumPy operations
- Write code using the basic NumPy functions

A. Ranged data

While `np.array` can be used to create any array, it is equivalent to hardcoding an array. This won't work when the array has hundreds of values. Instead, NumPy provides an option to create ranged data arrays using `np.arange`. The function acts very similar to the `range` function in Python, and will always return a 1-D array.

The code below contains example usages of `np.arange`.

```
arr = np.arange(5)
print(repr(arr))

arr = np.arange(5.1)
print(repr(arr))

arr = np.arange(-1, 4)
print(repr(arr))

arr = np.arange(-1.5, 4, 2)
print(repr(arr))
```



The output of `np.arange` is specified as follows:

- If only a single number, n, is passed in as an argument, `np.arange` will return an array with all the integers in the range $[0, n)$. **Note:** the lower end is inclusive while the upper end is exclusive.
- For two arguments, m and n, `np.arange` will return an array with all the

integers in the range [m, n).

- For three arguments, m, n, and s, `np.arange` will return an array with the integers in the range [m, n) using a step size of s.
- Like `np.array`, `np.arange` performs upcasting. It also has the `dtype` keyword argument to manually cast the array.

To specify the number of elements in the returned array, rather than the step size, we can use the `np.linspace` function.

This function takes in a required first two arguments, for the start and end of the range, respectively. The end of the range is inclusive for `np.linspace`, unless the keyword argument `endpoint` is set to `False`. To specify the number of elements, we set the `num` keyword argument (its default value is `50`).

The code below shows example usages of `np.linspace`. It also takes in the `dtype` keyword argument for manual casting.

```
arr = np.linspace(5, 11, num=4)
print(repr(arr))

arr = np.linspace(5, 11, num=4, endpoint=False)
print(repr(arr))

arr = np.linspace(5, 11, num=4, dtype=np.int32)
print(repr(arr))
```

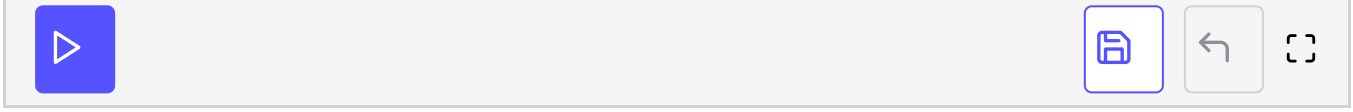
B. Reshaping data

The function we use to reshape data in NumPy is `np.reshape`. It takes in an array and a new shape as required arguments. The new shape must exactly contain all the elements from the input array. For example, we could reshape an array with 12 elements to `(4, 3)`, but we can't reshape it to `(4, 4)`.

We are allowed to use the special value of -1 in at most one dimension of the new shape. The dimension with -1 will take on the value necessary to allow the new shape to contain all the elements of the array.

The code below shows example usages of `np.reshape`.

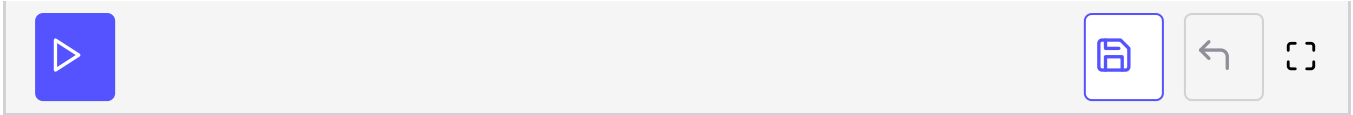
```
arr = np.arange(8)

reshaped_arr = np.reshape(arr, (2, 4))

print(repr(reshaped_arr))
print('New shape: {}'.format(reshaped_arr.shape))

reshaped_arr = np.reshape(arr, (-1, 2, 2))
print(repr(reshaped_arr))
print('New shape: {}'.format(reshaped_arr.shape))
```



While the `np.reshape` function can perform any reshaping utilities we need, NumPy provides an inherent function for flattening an array. Flattening an array reshapes it into a 1D array. Since we need to flatten data quite often, it is a useful function.

The code below flattens an array using the inherent `flatten` function.

```
arr = np.arange(8)
arr = np.reshape(arr, (2, 4))
flattened = arr.flatten()
print(repr(arr))
print('arr shape: {}'.format(arr.shape))
print(repr(flattened))
print('flattened shape: {}'.format(flattened.shape))
```



C. Transposing

Similar to how it is common to reshape data, it is also common to transpose data. Perhaps we have data that's supposed to be in a particular format, but some new data we get is rearranged. We can just transpose the data, using the `np.transpose` function, to convert it to the proper format.

The code below shows an example usage of the `np.transpose` function. The matrix rows become columns after the transpose.

```
arr = np.arange(8)
arr = np.reshape(arr, (4, 2))
transposed = np.transpose(arr)
print(repr(arr))
print('arr shape: {}'.format(arr.shape))
print(repr(transposed))
print('transposed shape: {}'.format(transposed.shape))
```

The function takes in a required first argument, which will be the array we want to transpose. It also has a single keyword argument called `axes`, which represents the new *permutation* of the dimensions.

The permutation is a tuple/list of integers, with the same length as the number of dimensions in the array. It tells us where to switch up the dimensions. For example, if the permutation had 3 at index 1, it means the old third dimension of the data becomes the new second dimension (since index 1 represents the second dimension).

The code below shows an example usage of the `np.transpose` function with the `axes` keyword argument. The `shape` property gives us the shape of an array.

```
arr = np.arange(24)
arr = np.reshape(arr, (3, 4, 2))
transposed = np.transpose(arr, axes=(1, 2, 0))
print('arr shape: {}'.format(arr.shape))
print('transposed shape: {}'.format(transposed.shape))
```

In this example, the old first dimension became the new third dimension, the old second dimension became the new first dimension, and the old third dimension became the new second dimension. The default value for `axes` is a dimension reversal (e.g. for 3-D data the default `axes` value is `[2, 1, 0]`).

D. Zeros and ones

Sometimes, we need to create arrays filled solely with 0 or 1. For example, since binary data is labeled with 0 and 1, we may need to create dummy datasets of strictly one label. For creating these arrays, NumPy provides the functions `np.zeros` and `np.ones`. They both take in the same arguments, which includes just one required argument, the array shape. The functions also allow for manual casting using the `dtype` keyword argument.

The code below shows example usages of `np.zeros` and `np.ones`.

```
arr = np.zeros(4)
print(repr(arr))

arr = np.ones((2, 3))
print(repr(arr))

arr = np.ones((2, 3), dtype=np.int32)
print(repr(arr))
```

If we want to create an array of 0's or 1's with the same shape as another array, we can use `np.zeros_like` and `np.ones_like`.

The code below shows example usages of `np.zeros_like` and `np.ones_like`.

```
arr = np.array([[1, 2], [3, 4]])
print(repr(np.zeros_like(arr)))

arr = np.array([[0., 1.], [1.2, 4.]])
print(repr(np.ones_like(arr)))
print(repr(np.ones_like(arr, dtype=np.int32)))
```

Time to Code!

Our initial array will just be all the integers from 0 to 11, inclusive. We'll also reshape it so it has three dimensions.

First, set `arr` equal to `np.arange` with `12` as the only argument.

Then, set `reshaped` equal to `np.reshape` with `arr` as the first argument and `(2, 3, 2)` as the second argument.

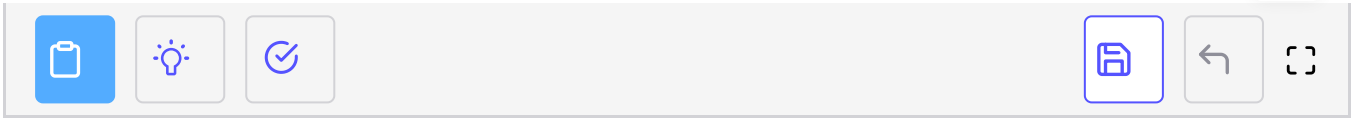
```
# CODE HERE
```

Next we want to get a flattened version of the reshaped array (the flattened version is equivalent to `arr`), as well as a transposed version. For the transposed version of `reshaped`, we use a permutation of `(1, 2, 0)`.

Set `flattened` equal to `reshaped.flatten()`.

Then set `transposed` equal to `np.transpose` with `reshaped` as the first argument and the specified permutation for the `axes` keyword argument.

```
# CODE HERE
```



We'll create an array of 5 elements, all of which are `0`. We'll also create an array with the same shape as `transposed`, but containing only `1` as the elements.

Set `zeros_arr` equal to `np.zeros` with `5` as the lone argument.

Then set `ones_arr` equal to `np.ones_like` with `transposed` as the lone argument.

```
# CODE HERE
```

The final array will contain 101 evenly spaced numbers between -3.5 and 1.5, inclusive. Since they are evenly spaced, the difference between adjacent numbers is 0.05.

Set `points` equal to `np.linspace` with `-3.5` and `1.5` as the first two arguments, respectively, as well as `101` for the `num` keyword argument.

```
# CODE HERE
```

Math

Understand how arithmetic and linear algebra work in NumPy.

Chapter Goals:

- Learn how to perform math operations in NumPy
- Write code using NumPy math functions

A. Arithmetic

One of the main purposes of NumPy is to perform multi-dimensional arithmetic. Using NumPy arrays, we can apply arithmetic to each element with a single operation.

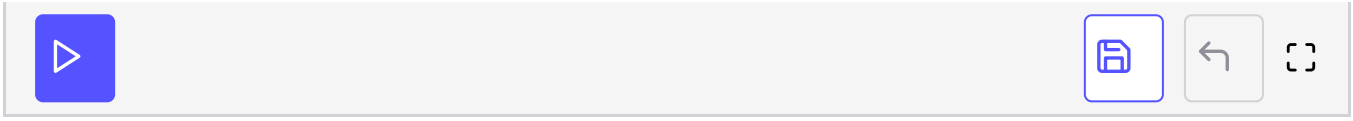
The code below shows multi-dimensional arithmetic with NumPy.

```
arr = np.array([[1, 2], [3, 4]])
# Add 1 to element values
print(repr(arr + 1))
# Subtract element values by 1.2
print(repr(arr - 1.2))
# Double element values
print(repr(arr * 2))
# Halve element values
print(repr(arr / 2))
# Integer division (half)
print(repr(arr // 2))
# Square element values
print(repr(arr**2))
# Square root element values
print(repr(arr**0.5))
```

Using NumPy arithmetic, we can easily modify large amounts of numeric data with only a few operations. For example, we could convert a dataset of Fahrenheit temperatures to their equivalent Celsius form.

The code below converts Fahrenheit to Celsius in NumPy.

```
def f2c(temps):
  return (5/9)*(temps-32)

fahrenheits = np.array([32, -4, 14, -40])
celsius = f2c(fahrenheits)
print('Celsius: {}'.format(repr(celsius)))
```



It is important to note that performing arithmetic on NumPy arrays **does not change the original array**, and instead produces a new array that is the result of the arithmetic operation.

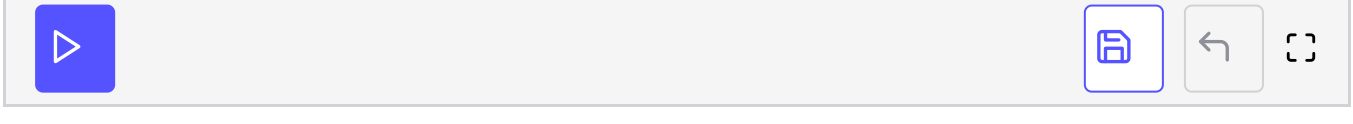
B. Non-linear functions

Apart from basic arithmetic operations, NumPy also allows you to use non-linear functions such as exponentials and logarithms.

The function `np.exp` performs a base *e* exponential on an array, while the function `np.exp2` performs a base 2 exponential. Likewise, `np.log`, `np.log2`, and `np.log10` all perform logarithms on an input array, using base *e*, base 2, and base 10, respectively.

The code below shows various exponentials and logarithms with NumPy. Note that `np.e` and `np.pi` represent the mathematical constants *e* and π, respectively.

```
arr = np.array([[1, 2], [3, 4]])
# Raised to power of e
print(repr(np.exp(arr)))
# Raised to power of 2
print(repr(np.exp2(arr)))

arr2 = np.array([[1, 10], [np.e, np.pi]])
# Natural logarithm
print(repr(np.log(arr2)))
# Base 10 logarithm
print(repr(np.log10(arr2)))
```

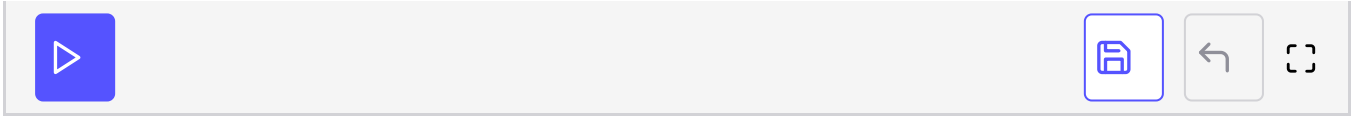


To do a regular power operation with any base, we use `np.power`. The first argument to the function is the base, while the second is the power. If the base or power is an array rather than a single number, the operation is applied to

of power is an array rather than a single number, the operation is applied to every element in the array.

The code below shows examples of using `np.power`.

```
arr = np.array([[1, 2], [3, 4]])
# Raise 3 to power of each number in arr
print(repr(np.power(3, arr)))
arr2 = np.array([[10.2, 4], [3, 5]])
# Raise arr2 to power of each number in arr
print(repr(np.power(arr2, arr)))
```



In addition to exponentials and logarithms, NumPy has various other mathematical functions, which are listed here.

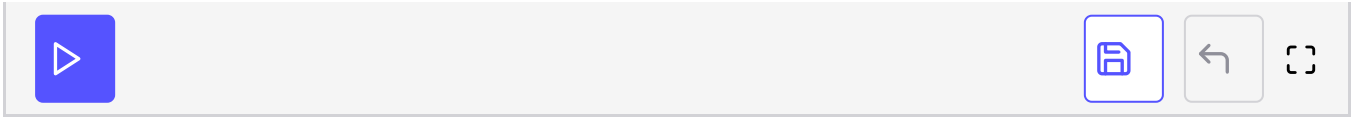
C. Matrix multiplication

Since NumPy arrays are basically vectors and matrices, it makes sense that there are functions for dot products and matrix multiplication. Specifically, the main function to use is `np.matmul`, which takes two vector/matrix arrays as input and produces a dot product or matrix multiplication.

The code below shows various examples of matrix multiplication. When both inputs are 1-D, the output is the dot product.

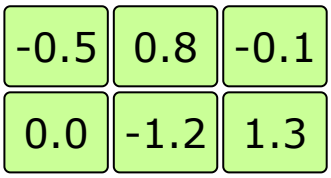
Note that the dimensions of the two input matrices must be valid for a matrix multiplication. Specifically, the second dimension of the first matrix must equal the first dimension of the second matrix, otherwise `np.matmul` will result in a `ValueError`.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([-3, 0, 10])
print(np.matmul(arr1, arr2))

arr3 = np.array([[1, 2], [3, 4], [5, 6]])
arr4 = np.array([[-1, 0, 1], [3, 2, -4]])
print(repr(np.matmul(arr3, arr4)))
print(repr(np.matmul(arr4, arr3)))
# This will result in ValueError
print(repr(np.matmul(arr3, arr3)))
```

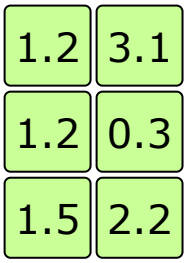


Time to Code!

We'll create a couple of matrix arrays to perform our math operations on. The first array will represent the matrix:



The second array will represent the matrix:



Set `arr` equal to `np.array` applied to a list of lists representing the first matrix.

Then set `arr2` equal to `np.array` applied to a list of lists representing the second matrix.

```
# CODE HERE
```

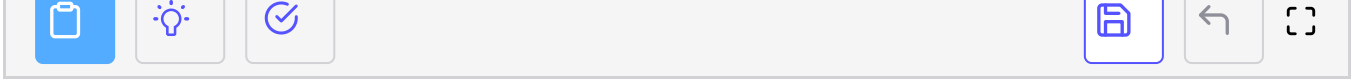
Next we'll apply some arithmetic to `arr`. Specifically, we'll do multiplication, addition, and squaring.

Set `multiplied` equal to `arr` multiplied by `np.pi`.

Then set `added` equal to the result of adding `arr` and `multiplied`.

Finally, set `squared` equal to `added` with each of its elements squared.

```
# CODE HERE
```

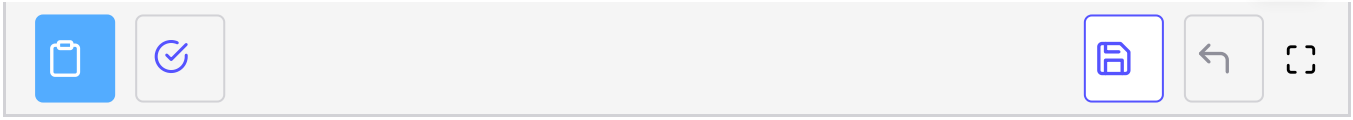


After the arithmetic operations, we'll apply the base *e* exponential and logarithm to our array matrices.

Set `exponential` equal to `np.exp` applied to `squared`.

Then set `logged` equal to `np.log` applied to `arr2`.

```
# CODE HERE
```

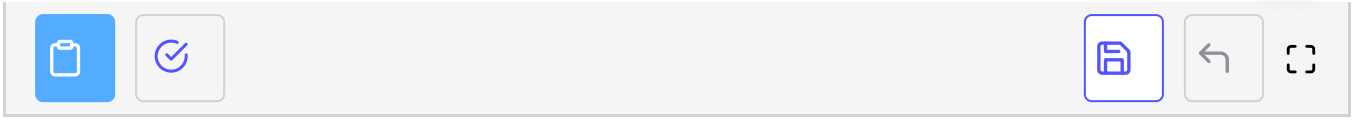


Note that `exponential` has shape `(2, 3)` and `logged` has shape `(3, 2)`. So we can perform matrix multiplication both ways.

Set `matmul1` equal to `np.matmul` with first argument `logged` and second argument `exponential`. Note that `matmul1` will have shape `(3, 3)`.

Then set `matmul2` equal to `np.matmul` with first argument `exponential` and second argument `logged`. Note that `matmul2` will have shape `(2, 2)`.

```
# CODE HERE
```



Random

Generate numbers and arrays from different random distributions.

Chapter Goals:

- Learn about random operations in NumPy
- Write code using the `np.random` submodule

A. Random integers

Similar to the Python `random` module, NumPy has its own submodule for pseudo-random number generation called `np.random`. It provides all the necessary randomized operations and extends it to multi-dimensional arrays. To generate pseudo-random integers, we use the `np.random.randint` function.

The code below shows example usages of `np.random.randint`.

```
print(np.random.randint(5))
print(np.random.randint(5))
print(np.random.randint(5, high=6))

random_arr = np.random.randint(-3, high=14,
                               size=(2, 2))
print(repr(random_arr))
```

The `np.random.randint` function takes in a single required argument, which actually depends on the `high` keyword argument. If `high=None` (which is the default value), then the required argument represents the upper (exclusive) end of the range, with the lower end being 0. Specifically, if the required argument is n, then the random integer is chosen uniformly from the range $[0, n)$.

If `high` is not `None`, then the required argument will represent the lower (inclusive) end of the range, while `high` represents the upper (exclusive) end.

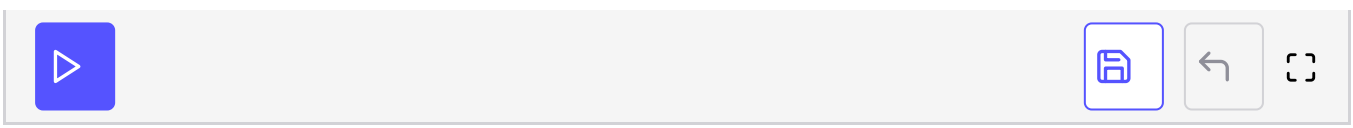
The `size` keyword argument specifies the size of the output array, where each integer in the array is randomly drawn from the specified range. As a default, `np.random.randint` returns a single integer.

B. Utility functions

Some fundamental utility functions from the `np.random` module are `np.random.seed` and `np.random.shuffle`. We use the `np.random.seed` function to set the random seed, which allows us to control the outputs of the pseudo-random functions. The function takes in a single integer as an argument, representing the random seed.

The code below uses `np.random.seed` with the same random seed. Note how the outputs of the random functions in each subsequent run are identical when we set the same random seed.

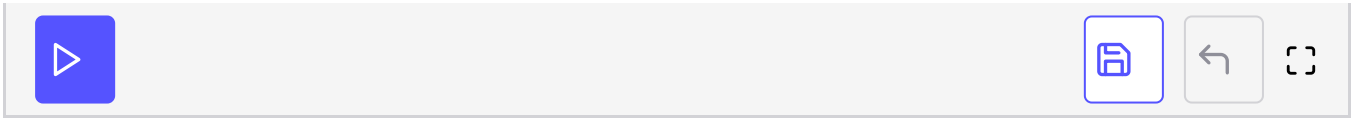
```
np.random.seed(1)
print(np.random.randint(10))
random_arr = np.random.randint(3, high=100,
                               size=(2, 2))
print(repr(random_arr))

# New seed
np.random.seed(2)
print(np.random.randint(10))
random_arr = np.random.randint(3, high=100,
                               size=(2, 2))
print(repr(random_arr))

# Original seed
np.random.seed(1)
print(np.random.randint(10))
random_arr = np.random.randint(3, high=100,
                               size=(2, 2))
print(repr(random_arr))
```



The `np.random.shuffle` function allows us to randomly shuffle an array. Note that the shuffling happens in place (i.e. no return value), and shuffling multi-dimensional arrays only shuffles the first dimension.

The code below shows example usages of `np.random.shuffle`. Note that only the rows of `matrix` are shuffled (i.e. shuffling along first dimension only).

```
vec = np.array([1, 2, 3, 4, 5])
np.random.shuffle(vec)
print(repr(vec))
np.random.shuffle(vec)
print(repr(vec))

matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
np.random.shuffle(matrix)
print(repr(matrix))
```

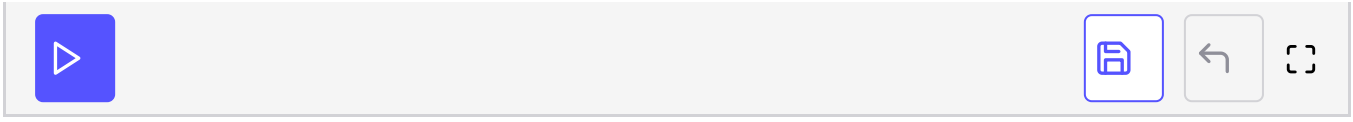


C. Distributions

Using `np.random` we can also draw samples from probability distributions. For example, we can use `np.random.uniform` to draw pseudo-random real numbers from a uniform distribution.

The code below shows usages of `np.random.uniform`.

```
print(np.random.uniform())
print(np.random.uniform(low=-1.5, high=2.2))
print(repr(np.random.uniform(size=3)))
print(repr(np.random.uniform(low=-3.4, high=5.9,
                             size=(2, 2))))
```



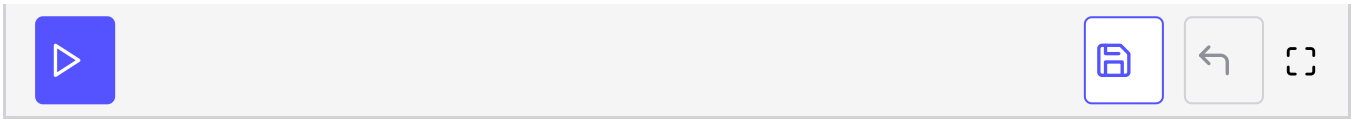
The function `np.random.uniform` actually has no required arguments. The keyword arguments, `low` and `high`, represent the inclusive lower end and exclusive upper end from which to draw random samples. Since they have default values of 0.0 and 1.0, respectively, the default outputs of `np.random.uniform` come from the range [0.0, 1.0).

The `size` keyword argument is the same as the one for `np.random.randint`, i.e. it represents the output size of the array.

Another popular distribution we can sample from is the normal (Gaussian) distribution. The function we use is `np.random.normal`.

The code below shows usages of `np.random.normal`.

```
print(np.random.normal())
print(np.random.normal(loc=1.5, scale=3.5))
print(repr(np.random.normal(loc=-2.4, scale=4.0,
                            size=(2, 2))))
```



Like `np.random.uniform`, `np.random.normal` has no required arguments. The `loc` and `scale` keyword arguments represent the mean and standard deviation, respectively, of the normal distribution we sample from.

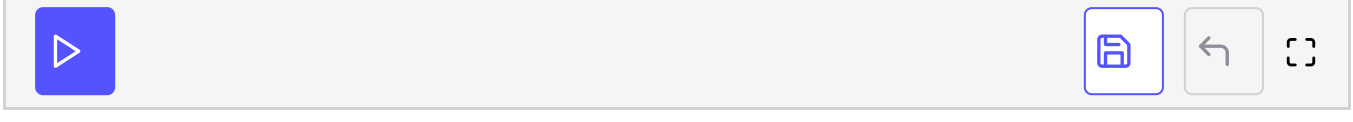
NumPy provides quite a few more built-in distributions, which are listed here.

D. Custom sampling

While NumPy provides built-in distributions to sample from, we can also sample from a custom distribution with the `np.random.choice` function.

The code below shows example usages of `np.random.choice`.

```
colors = ['red', 'blue', 'green']
print(np.random.choice(colors))
print(repr(np.random.choice(colors, size=2)))
print(repr(np.random.choice(colors, size=(2, 2),
                            p=[0.8, 0.19, 0.01])))
```



The required argument for `np.random.choice` is the custom distribution we sample from. The `p` keyword argument denotes the probabilities given to each element in the input distribution. Note that the list of probabilities for `p` must sum to 1.

In the example, we set `p` such that `'red'` has a probability of 0.8 of being chosen, `'blue'` has a probability of 0.19, and `'green'` has a probability of 0.01. When `p` is not set, the probabilities are equal for each element in the distribution (and sum to 1).

Time to Code!

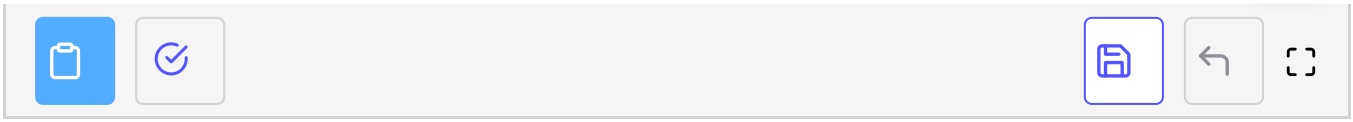
Note: it is important you do all the instructions in the order listed. We test your code by setting a fixed `np.random.seed`, so in order for your code to match the reference output, all the functions must be run in the correct order.

We'll start off by obtaining some random integers. The first integer we get will be randomly chosen from the range [0, 5). The remaining integers will be part of a 3x5 NumPy array, each randomly chosen from the range [3, 10).

Set `random1` equal to `np.random.randint` with `5` as the only argument.

Then set `random_arr` equal to `np.random.randint` with `3` as the first argument, `10` as the `high` keyword argument, and `(3, 5)` as the `size` keyword argument.

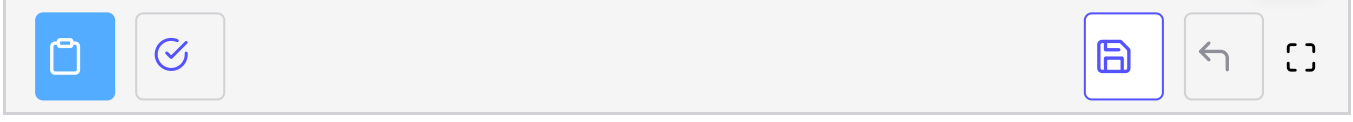
```
# CODE HERE
```



The next two arrays will be drawn randomly from distributions. The first will contain 5 numbers drawn uniformly from the range [-2.5, 1.5].

Set `random_uniform` equal to `np.random.uniform` with the `low` and `high` keyword arguments set to `-2.5` and `1.5`, respectively. The `size` keyword argument should be set to `5`.

```
# CODE HERE
```



The second array will contain 50 numbers drawn from a normal distribution with mean 2.0 and standard deviation 3.5.

Set `random_norm` equal to `np.random.normal` with the `loc` and `scale` keyword arguments set to `2.0` and `3.5`, respectively. The `size` keyword argument should be set to `(10, 5)`.

```
# CODE HERE
```

We'll now create our own distribution of strings and randomly select from it. The values for our distribution will be `'a'`, `'b'`, `'c'`, `'d'`.

To choose a value, we'll use a probability distribution of `[0.5, 0.1, 0.2, 0.2]`, i.e. `'a'` will have probability 0.5 of being chosen, `'b'` will have a probability of 0.1, etc.

Set `choices` equal to a list of the specified values, in the order given.

Set `choice` equal to `np.random.choice` with `choices` as the first argument and the specified probability distribution as the `p` keyword argument.

```
# CODE HERE
```

The last random operation we perform will be an in-place shuffle of a NumPy array.

Set `arr` equal to a NumPy array containing the integers from 1 to 5, inclusive.

Then apply `np.random.shuffle` to `arr`.

```
# CODE HERE
```

Indexing

Index into NumPy arrays to extract data and array slices.

Chapter Goals:

- Learn about indexing arrays in NumPy
- Write code for indexing and slicing arrays

A. Array accessing

Accessing NumPy arrays is identical to accessing Python lists. For multi-dimensional arrays, it is equivalent to accessing Python lists of lists.

The code below shows example accesses of NumPy arrays.

```
arr = np.array([1, 2, 3, 4, 5])
print(arr[0])
print(arr[4])

arr = np.array([[6, 3], [0, 2]])
# Subarray
print(repr(arr[0]))
```

B. Slicing

NumPy arrays also support slicing. Similar to Python, we use the colon operator (i.e. `arr[:]`) for slicing. We can also use negative indexing to slice in the backwards direction.

The code below shows example slices of a 1-D NumPy array.

```
arr = np.array([1, 2, 3, 4, 5])
print(repr(arr[:]))
print(repr(arr[1:]))
print(repr(arr[2:4]))
print(repr(arr[:-1]))
print(repr(arr[-2:]))
```

For multi-dimensional arrays, we can use a comma to separate slices across each dimension.

The code below shows example slices of a 2-D NumPy array.

```
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
print(repr(arr[:]))
print(repr(arr[1:]))
print(repr(arr[:, -1]))
print(repr(arr[:, 1:]))
print(repr(arr[0:1, 1:]))
print(repr(arr[0, 1:]))
```

C. Argmin and argmax

In addition to accessing and slicing arrays, it is useful to figure out the actual indexes of the minimum and maximum elements. To do this, we use the `np.argmin` and `np.argmax` functions.

The code below shows example usages of `np.argmin` and `np.argmax`. Note that the index of element `-6` is index `5` in the flattened version of `arr`.

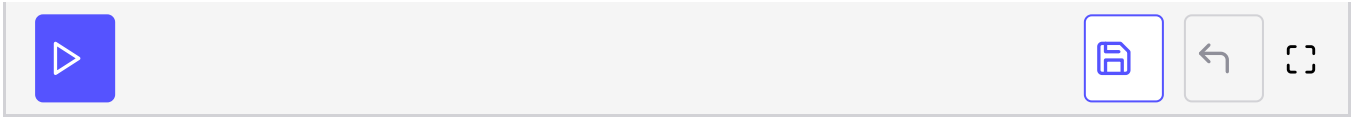
```
arr = np.array([[-2, -1, -3],
                [4, 5, -6],
                [-3, 9, 1]])
print(np.argmin(arr[0]))
print(np.argmax(arr[2]))
print(np.argmin(arr))
```

The `np.argmin` and `np.argmax` functions take the same arguments. The required argument is the input array and the `axis` keyword argument specifies which dimension to apply the operation on.

The code below shows how the `axis` keyword argument is used for these

functions.

```
arr = np.array([[-2, -1, -3],
                [4, 5, -6],
                [-3, 9, 1]])
print(repr(np.argmin(arr, axis=0)))
print(repr(np.argmin(arr, axis=1)))
print(repr(np.argmax(arr, axis=-1)))
```



In our example, using `axis=0` meant the function found the index of the minimum *row* element for each column. When we used `axis=1`, the function found the index of the minimum *column* element for each row.

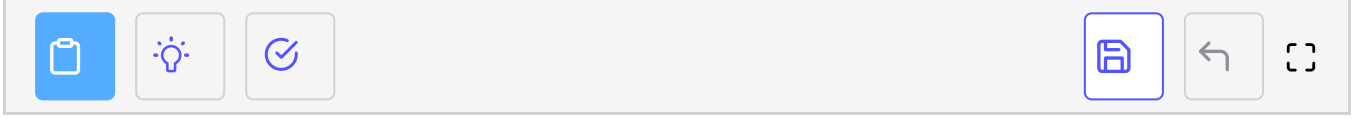
Setting `axis` to -1 just means we apply the function across the last dimension. In this case, `axis=-1` is equivalent to `axis=1`.

Time to Code!

Each coding exercise in this chapter will be to complete a small function that takes in a 2-D NumPy matrix (`data`) as input. The first function to complete is `direct_index`.

Set `elem` equal to the third element of the second row in `data` (remember that the first row is index 0). Then return `elem`.

```
def direct_index(data):
  # CODE HERE
  pass
```



The next function, `slice_data`, will return two slices from the input `data`.

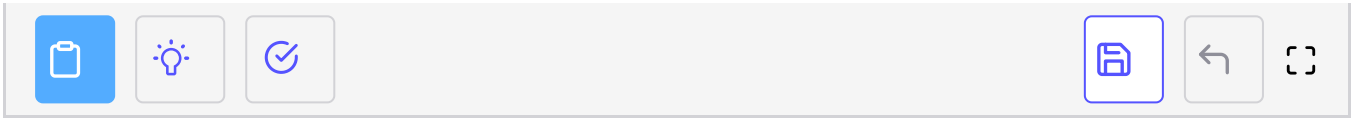
The first slice will contain all the rows, but will skip the first element in each row. The second slice will contain all the elements of the first three rows except the last two elements.

Set `slice1` equal to the specified first slice. Remember that NumPy uses a comma to separate slices along different dimensions.

Set `slice2` equal to the specified second slice.

Return a tuple containing `slice1` and `slice2`, in that order.

```
def slice_data(data):
  # CODE HERE
  pass
```



The next function, `argmin_data`, will find minimum indexes in the input `data`.

We can use `np.argmin` to find minimum points in the `data` array. First, we'll find the index of the overall minimum element.

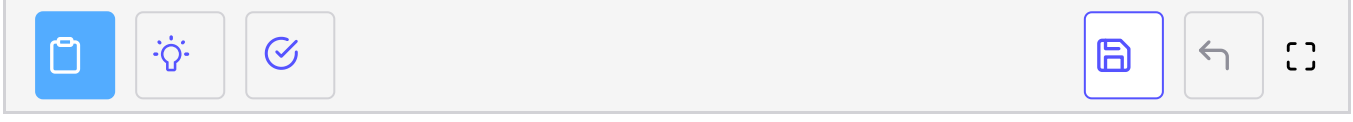
We can also return the indexes of each row's minimum element. This is equivalent to finding the minimum column for each row, which means our operation is done along axis `1`.

Set `argmin_all` equal to `np.argmin` with `data` as the only argument.

Set `argmin1` equal to `np.argmin` with `data` as the first argument and the specified `axis` keyword argument.

Return a tuple containing `argmin_all` and `argmin1`, in that order.

```
def argmin_data(data):
  # CODE HERE
  pass
```

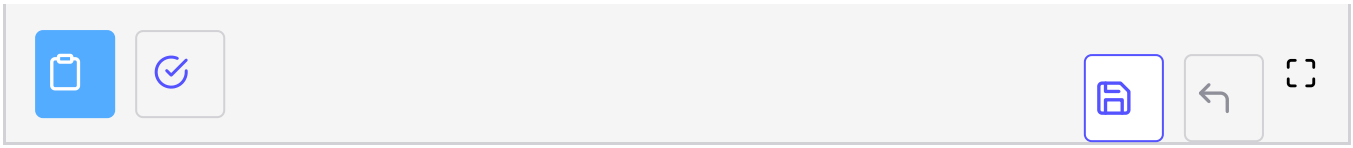


The final function, `argmax_data`, will find the index of each row's maximum element in `data`. Since there are only 2 dimensions in `data`, we can apply the operation along either axis `1` or `-1`.

Set `argmax_neg1` equal to `np.argmax` with `data` as the first argument and `-1` as the `axis` keyword argument. Then return `argmax_neg1`.

```
def argmax_data(data):
  # CODE HERE
```

```
pass
```



Filtering

Filter NumPy data for specific values.

Chapter Goals:

- Learn how to filter data in NumPy
- Write code for filtering NumPy arrays

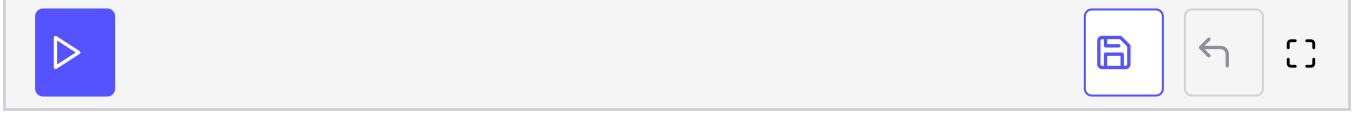
A. Filtering data

Sometimes we have data that contains values we don't want to use. For example, when tracking the best hitters in baseball, we may want to only use the batting average data above .300. In this case, we should *filter* the overall data for only the values that we want.

The key to filtering data is through basic relation operations, e.g. `==`, `>`, etc. In NumPy, we can apply basic relation operations element-wise on arrays.

The code below shows relation operations on NumPy arrays. The `~` operation represents a boolean negation, i.e. it flips each truth value in the array.

```
arr = np.array([[0, 2, 3],
                [1, 3, -6],
                [-3, -2, 1]])
print(repr(arr == 3))
print(repr(arr > 0))
print(repr(arr != 1))
# Negated from the previous step
print(repr(~(arr != 1)))
```



Something to note is that `np.nan` can't be used with any relation operation. Instead, we use `np.isnan` to filter for the location of `np.nan`.

The code below uses `np.isnan` to determine which locations of the array contain `np.nan` values.

```
arr = np.array([[0, 2, np.nan],
                [1, np.nan, -6],
                [np.nan, -2, 1]])
print(repr(np.isnan(arr)))
```

Each boolean array in our examples represents the location of elements we want to filter for. The way we perform the filtering itself is through the `np.where` function.

B. Filtering in NumPy

The `np.where` function takes in a required first argument, which is a boolean array where `True` represents the locations of the elements we want to filter for. When the function is applied with only the first argument, it returns a tuple of 1-D arrays.

The tuple will have size equal to the number of dimensions in the data, and each array represents the `True` indices for the corresponding dimension. Note that the arrays in the tuple will all have the same length, equal to the number of `True` elements in the input argument.

The code below shows how to use `np.where` with a single argument.

```
print(repr(np.where([True, False, True])))

arr = np.array([0, 3, 5, 3, 1])
print(repr(np.where(arr == 3)))

arr = np.array([[0, 2, 3],
                [1, 0, 0],
                [-3, 0, 0]])
x_ind, y_ind = np.where(arr != 0)
print(repr(x_ind)) # x indices of non-zero elements
print(repr(y_ind)) # y indices of non-zero elements
print(repr(arr[x_ind, y_ind]))
```

The interesting thing about `np.where` is that it must be applied with exactly 1 or 3 arguments. When we use 3 arguments, the first argument is still the

boolean array. However, the next two arguments represent the `True`

replacement values and the `False` replacement values, respectively. The output of the function now becomes an array with the same shape as the first argument.

The code below shows how to use `np.where` with 3 arguments.

```
np_filter = np.array([[True, False], [False, True]])
positives = np.array([[1, 2], [3, 4]])
negatives = np.array([[-2, -5], [-1, -8]])
print(repr(np.where(np_filter, positives, negatives)))

np_filter = positives > 2
print(repr(np.where(np_filter, positives, negatives)))

np_filter = negatives > 0
print(repr(np.where(np_filter, positives, negatives)))
```

Note that our second and third arguments necessarily had the same shape as the first argument. However, if we wanted to use a constant replacement value, e.g. `-1`, we could incorporate broadcasting. Rather than using an entire array of the same value, we can just use the value itself as an argument.

The code below showcases broadcasting with `np.where`.

```
np_filter = np.array([[True, False], [False, True]])
positives = np.array([[1, 2], [3, 4]])
print(repr(np.where(np_filter, positives, -1)))
```

C. Axis-wise filtering

If we wanted to filter based on rows or columns of data, we could use the `np.any` and `np.all` functions. Both functions take in the same arguments, and return a single boolean or a boolean array. The required argument for both functions is a boolean array.

The code below shows usage of `np.any` and `np.all` with a single argument.

```
arr = np.array([[-2, -1, -3],
                [4, 5, -6],

                [3, 9, 1]])
print(repr(arr > 0))
print(np.any(arr > 0))
print(np.all(arr > 0))
```

The `np.any` function is equivalent to performing a logical OR (`||`), while the `np.all` function is equivalent to a logical AND (`&&`) on the first argument. np.any returns true if even one of the elements in the array meets the condition and np.all returns true only if all the elements meet the condition. When only a single argument is passed in, the function is applied across the entire input array, so the returned value is a single boolean.

However, if we use a multi-dimensional input and specify the `axis` keyword argument, the returned value will be an array. The `axis` argument has the same meaning as it did for `np.argmin` and `np.argmax` from the previous chapter. Using `axis=0` means the function finds the index of the minimum *row* element for each column. When we used `axis=1`, the function finds the index of the minimum *column* element for each row.

Setting `axis` to -1 just means we apply the function across the last dimension.

The code below shows examples of using `np.any` and `np.all` with the `axis` keyword argument.

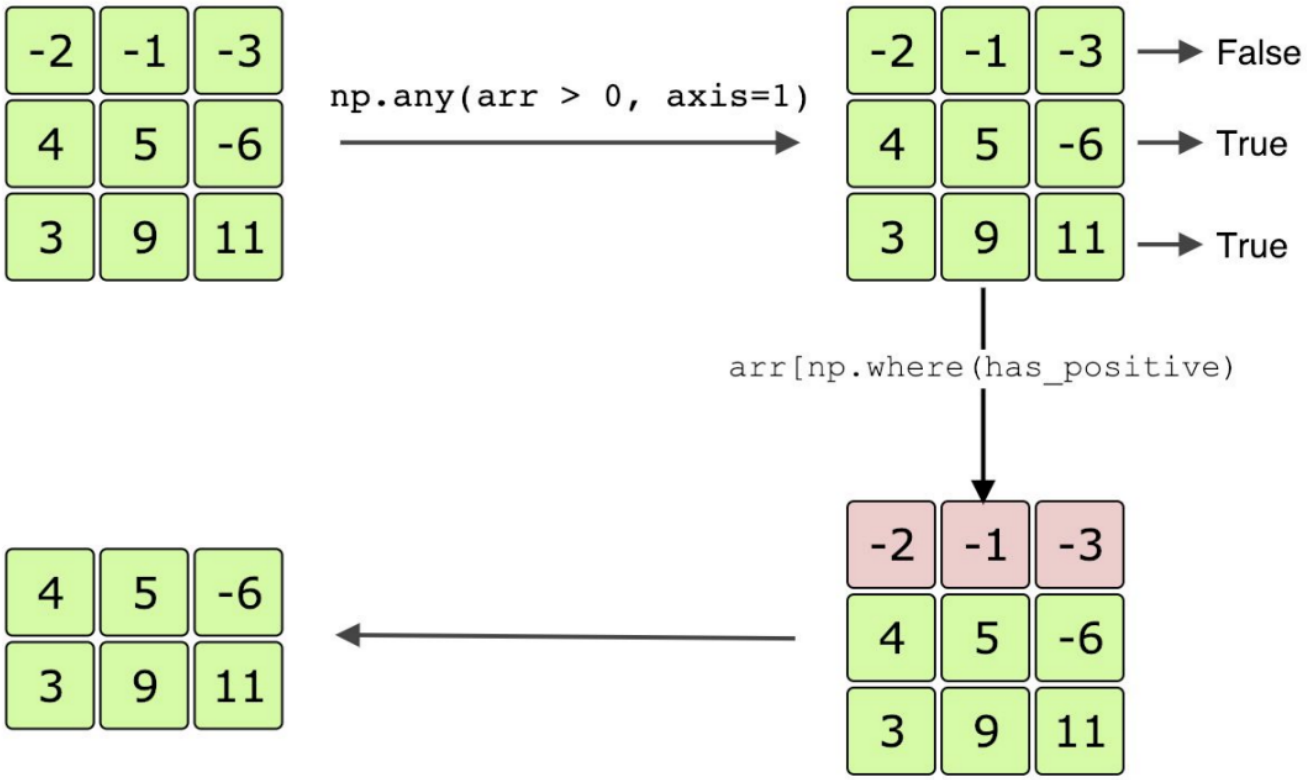
```
arr = np.array([[-2, -1, -3],
                [4, 5, -6],
                [3, 9, 1]])
print(repr(arr > 0))
print(repr(np.any(arr > 0, axis=0)))
print(repr(np.any(arr > 0, axis=1)))
print(repr(np.all(arr > 0, axis=1)))
```

We can use `np.any` and `np.all` in tandem with `np.where` to filter for entire rows or columns of data.

In the code example below, we use `np.any` to obtain a boolean array

representing the rows that have at least one positive number. We then use the

boolean array as the input to `np.where`, which gives us the actual indices of the rows with at least one positive number.




```
arr = np.array([[-2, -1, -3],
                [4, 5, -6],
                [3, 9, 1]])
has_positive = np.any(arr > 0, axis=1)
print(has_positive)
print(repr(arr[np.where(has_positive)]))
```

Time to Code!

Each coding exercise in this chapter will be to complete a small function that takes in a 2-D NumPy matrix (`data`) as input. The first function to complete is `get_positives`.

Set a tuple of `x_ind, y_ind` equal to the output of `np.where`, applied with the condition `data > 0`.

Then return `data[x_ind, y_ind]`.

```
def get_positives(data):
```

```
def get_positives(data):
  # CODE HERE
  pass
```

Next, we'll complete the function `replace_zeros`. The function replaces each of the non-positive elements in `data` with 0. We first create an array of all 0's, with the same shape as `data`.

Then we filter the `data` array and replace the non-positive elements with the corresponding element from `zeros` (which will be a 0).

Set `zeros` equal to `np.zeros_like` with `data` as the lone argument.

Set `zero_replace` equal to `np.where` with the condition of `data > 0`. The second argument will be `data` and the third argument will be `zeros`.

Return `zero_replace`.

```
def replace_zeros(data):
  # CODE HERE
  pass
```

The next function, `replace_neg_one`, will replace the non-positive elements of `data` with `-1`. Rather than creating a separate array, we'll use broadcasting.

Set `neg_one_replace` equal to `np.where` with the condition of `data > 0`. The second argument will be `data` and the third will be `-1`.

Return `neg_one_replace`.

```
def replace_neg_one(data):
  # CODE HERE
  pass
```

Our final function, `coin_flip_filter` will apply a filter using a boolean array as the condition. We'll first create a boolean coin flip array with the same

shape as `data`.

Then we filter `data` using `bool_coin_flips` as the condition. For the `False` values in `bool_coin_flips`, we replace the corresponding index in `data` with a 1.

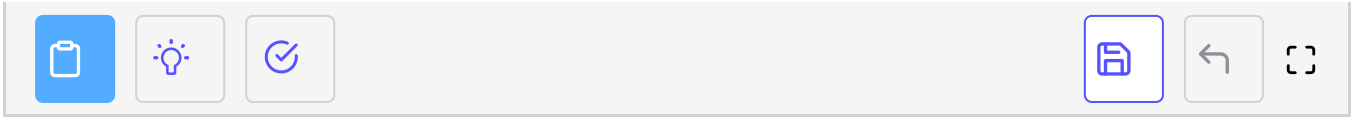
Set `coin_flips` equal to `np.random.randint` with `2` as the first argument and `data.shape` as the `size` keyword argument.

Set `bool_coin_flips` equal to `coin_flips`, cast as `np.bool` (using the `np.astype` function).

Set `one_replace` equal to `np.where` with `bool_coin_flips`, `data`, and `1` as the respective arguments.

Return `one_replace`.

```
def coin_flip_filter(data):
  # CODE HERE
  pass
```



Statistics

Learn how to apply statistical metrics to NumPy data.

Chapter Goals:

- Learn about basic statistical analysis in NumPy
- Write code to obtain statistics for NumPy arrays

A. Analysis

It is often useful to analyze data for its main characteristics and interesting trends. Though we will go more in-depth on data analysis in the section of this course titled Data Preprocessing with scikit-learn, there are still a few techniques in NumPy that allow us to quickly inspect data arrays.

For example, we can obtain minimum and maximum values of a NumPy array using its inherent `min` and `max` functions. This gives us an initial sense of the data's range, and can alert us to extreme outliers in the data.

The code below shows example usages of the `min` and `max` functions.

```
arr = np.array([[0, 72, 3],
                [1, 3, -60],
                [-3, -2, 4]])
print(arr.min())
print(arr.max())

print(repr(arr.min(axis=0)))
print(repr(arr.max(axis=-1)))
```

The `axis` keyword argument is identical to how it was used in `np.argmin` and `np.argmax` from the chapter on Indexing. In our example, we use `axis=0` to find an array of the minimum values in each column of `arr` and `axis=1` to find an array of the maximum values in each row of `arr`.

B. Statistical metrics

NumPy also provides basic statistical functions such as `np.mean`, `np.var`, and `np.median`, to calculate the mean, variance, and median of the data, respectively.

The code below shows how to obtain basic statistics with NumPy. Note that `np.median` applied without `axis` takes the median of the flattened array.

```
arr = np.array([[0, 72, 3],
                [1, 3, -60],
                [-3, -2, 4]])
print(np.mean(arr))
print(np.var(arr))
print(np.median(arr))
print(repr(np.median(arr, axis=-1)))
```

Each of these functions takes in the data array as a required argument and `axis` as a keyword argument. For a more comprehensive list of statistical functions (e.g. calculating percentiles, creating histograms, etc.), check out the NumPy statistics page.

Time to Code!

Each coding exercise in this chapter will be to complete a small function that takes in a 2-D NumPy matrix (`data`) as input. The first function to complete is `get_min_max`, which returns the overall minimum and maximum element in `data`.

Set `overall_min` equal to `data.min` applied with no arguments.

Set `overall_max` equal to `data.max` applied with no arguments.

Return a tuple of `overall_min` and `overall_max`, in that order.

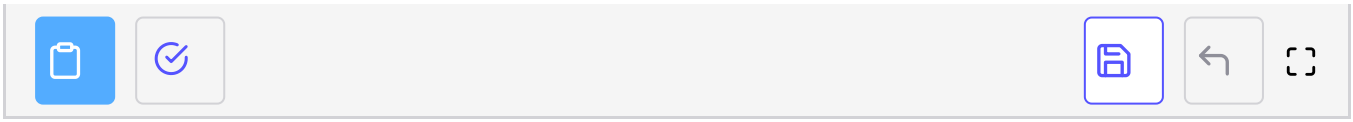
```
def get_min_max(data):
  # CODE HERE
  pass
```

Next, we'll complete `col_min`, which returns the minimums across each column of `data`.

Set `min0` equal to `data.min` with the `axis` keyword argument set to `0`.

Then return `min0`.

```
def col_min(data):
  # CODE HERE
  pass
```



The final function to complete is `basic_stats`, which returns the mean, median, and variance of `data`.

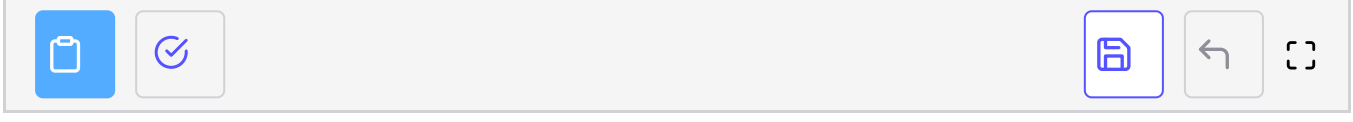
Set `mean` equal to `np.mean` applied to `data`.

Set `median` equal to `np.median` applied to `data`.

Set `var` equal to `np.var` applied to `data`.

Return a tuple containing `mean`, `median`, and `var`, in that order.

```
def basic_stats(data):
  # CODE HERE
  pass
```



Aggregation

Use aggregation techniques to combine NumPy data and arrays.

Chapter Goals:

- Learn how to aggregate data in NumPy
- Write code to obtain sums and concatenations of NumPy arrays

A. Summation

In the chapter on Math, we calculated the sum of individual values between multiple arrays. To sum the values within a single array, we use the `np.sum` function.

The function takes in a NumPy array as its required argument, and uses the `axis` keyword argument in the same way as described in previous chapters. If the `axis` keyword argument is not specified, `np.sum` returns the overall sum of the array.

The code below shows how to use `np.sum`.

```
arr = np.array([[0, 72, 3],
                [1, 3, -60],
                [-3, -2, 4]])
print(np.sum(arr))
print(repr(np.sum(arr, axis=0)))
print(repr(np.sum(arr, axis=1)))
```

In addition to regular sums, NumPy can perform cumulative sums using `np.cumsum`. Like `np.sum`, `np.cumsum` also takes in a NumPy array as a required argument and uses the `axis` argument. If the `axis` keyword argument is not specified, `np.cumsum` will return the cumulative sums for the flattened array.

The code below shows how to use `np.cumsum`. For a 2-D NumPy array, setting

`axis=0` returns an array with cumulative sums across each column, while

`axis=1` returns the array with cumulative sums across each row. Not setting `axis` returns a cumulative sum across all the values of the flattened array.

```
arr = np.array([[0, 72, 3],
                [1, 3, -60],
                [-3, -2, 4]])
print(repr(np.cumsum(arr)))
print(repr(np.cumsum(arr, axis=0)))
print(repr(np.cumsum(arr, axis=1)))
```

B. Concatenation

An important part of aggregation is combining multiple datasets. In NumPy, this equates to combining multiple arrays into one. The function we use to do this is `np.concatenate`.

Like the summation functions, `np.concatenate` uses the `axis` keyword argument. However, the default value for `axis` is `0` (i.e. dimension 0). Furthermore, the required argument for `np.concatenate` is a list of arrays, which the function combines into a single array.

The code below shows how to use `np.concatenate`, which aggregates arrays by joining them along a specific dimension. For 2-D arrays, not setting the `axis` argument (defaults to `axis=0`) concatenates the arrays vertically. When we set `axis=1`, the arrays are concatenated horizontally.

```
arr1 = np.array([[0, 72, 3],
                 [1, 3, -60],
                 [-3, -2, 4]])
arr2 = np.array([[-15, 6, 1],
                 [8, 9, -4],
                 [5, -21, 18]])
print(repr(np.concatenate([arr1, arr2])))
print(repr(np.concatenate([arr1, arr2], axis=1)))
print(repr(np.concatenate([arr2, arr1], axis=1)))
```

Time to Code!

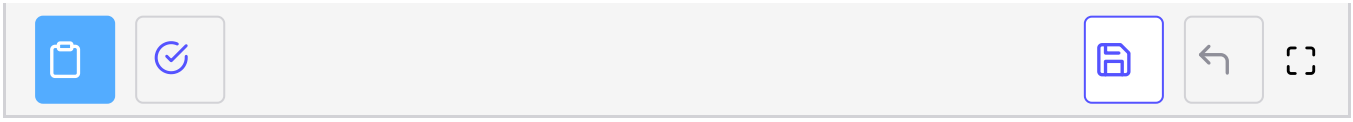
Each coding exercise in this chapter will be to complete a small function that takes in 2-D NumPy matrices as input. The first function to complete is `get_sums`, which returns the overall sum and column sums of `data`.

Set `total_sum` equal to `np.sum` applied to `data`.

Set `col_sum` equal to `np.sum` applied to `data`, with `axis` set to `0`.

Return a tuple of `total_sum` and `col_sum`, in that order.

```
def get_sums(data):
  # CODE HERE
  pass
```

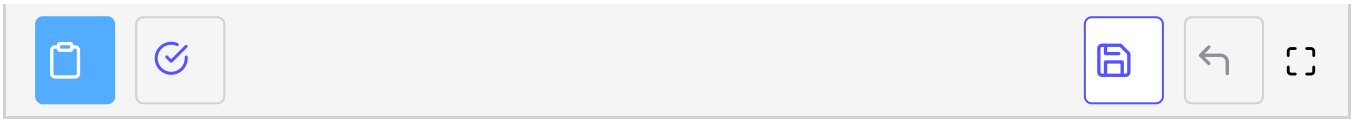


The next function to complete is `get_cumsum`, which returns the cumulative sums for each row of `data`.

Set `row_cumsum` equal to `np.cumsum` applied to `data` with `axis` set to `1`.

Then return `row_cumsum`.

```
def get_cumsum(data):
  # CODE HERE
  pass
```



The final function, `concat_arrays`, takes in two 2-D NumPy arrays as input. It returns the column-wise and row-wise concatenations of the input arrays.

Set `col_concat` equal to `np.concatenate` applied to a list of `data1`, `data2`, in that order.

Set `row_concat` equal to `np.concatenate` applied to a list of `data1`, `data2`, in that order. The `axis` keyword argument should be set to 1.

Return a tuple containing `col_concat` and `row_concat`, in that order.

```
def concat_arrays(data1, data2):
  # CODE HERE
  pass
```

Saving Data

Learn how to save and load NumPy data.

Chapter Goals:

- Learn how to save and load data in NumPy
- Write code to save NumPy data to a file

A. Saving

After performing data manipulation with NumPy, it's a good idea to save the data in a file for future use. To do this, we use the `np.save` function.

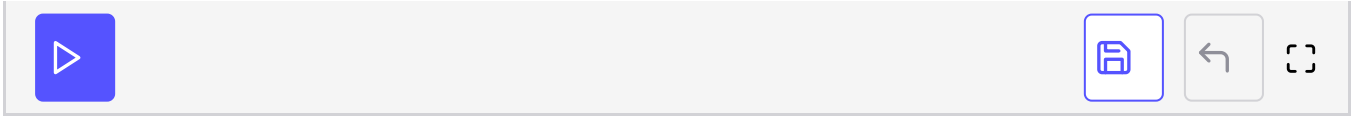
The first argument for the function is the name/path of the file we want to save our data to. The file name/path should have a ".npy" extension. If it does not, then `np.save` will append the ".npy" extension to it.

The second argument for `np.save` is the NumPy data we want to save. The function has no return value. Also, the format of the ".npy" files when viewed with a text editor is largely gibberish when viewed with a text editor.

If `np.save` is called with the name of a file that already exists, it will overwrite the previous file.

The code below shows examples of saving NumPy data.

```
arr = np.array([1, 2, 3])
# Saves to 'arr.npy'
np.save('arr.npy', arr)
# Also saves to 'arr.npy'
np.save('arr', arr)
```



B. Loading

After saving our data, we can load it again using `np.load`. The function's

required argument is the file name/path that contains the saved data. It returns the NumPy data exactly as it was saved.

Note that `np.load` will not append the ".npy" extension to the file name/path if it is not there.

The code below shows how to use `np.load` to load NumPy data.

```
arr = np.array([1, 2, 3])
np.save('arr.npy', arr)
load_arr = np.load('arr.npy')
print(repr(load_arr))

# Will result in FileNotFoundError
load_arr = np.load('arr')
```

Time to Code!

The coding exercise in this chapter will require you to complete the `save_points` function, which will save some randomly generated 2-D points in a file.

You'll generate 100 (x, y) points from a uniform distribution in the range [-2.5, 2.5), then save the points to `save_file`.

Set `points` equal to `np.random.uniform`, with the `low` and `high` keyword arguments representing the lower and upper ends of the range. The `size` keyword argument should be set to `(100, 2)`.

Call `np.save` with `save_file` as the first argument and `points` as the second argument.

```
def save_points(save_file):
  # CODE HERE
  pass
```

Quiz

1 What does the `arange` function do?

COMPLETED 0%

1 of 4

Introduction

An overview of data analysis with pandas.

In the **Data Processing** section, you will be using pandas to analyze Major League Baseball (MLB) data. The data comes courtesy of Sean Lahman, and contains statistics for every player, manager, and team in MLB history. The full database can be found and downloaded here.

A. Data analysis

Before doing any task with a dataset, it is a good idea to perform preliminary data analysis. Data analysis allows us to understand the dataset, find potential outlier values, and figure out which features of the dataset are most important to our application.

B. pandas

Since most machine learning frameworks (e.g. TensorFlow) are built on Python, it is beneficial to use a Python-based data analysis toolkit like pandas. pandas (all lowercase) is an excellent tool for processing and analyzing real world data, with utilities ranging from parsing multiple file formats to converting an entire data table into a NumPy matrix array.

In the following chapters we'll dive into the main data analysis functionalities of pandas. For a complete overview of the pandas toolkit, you can visit the official pandas website.

C. Matplotlib and pyplot

An essential part of data analysis is creating charts and plots to visualize the data. Similar to the saying, "a picture is worth a thousand words", data visualization can convey key data trends and correlations through a single figure.

The library we will use for data visualization in Python is Matplotlib.

Specifically, we'll be using the pyplot API of Matplotlib, which provides a variety of plotting tools from simple line plots to advanced visuals like heatmaps and 3-D plots. While we will only touch on the basic necessities for our data analysis (e.g. line plots, boxplots, etc.), a full overview of Matplotlib can be found at the official website.

Series

Learn about the pandas Series object for 1-D data.

Chapter Goals

- Learn about the pandas Series object and its basic utilities
- Write code to create several Series objects

A. 1-D data

Similar to NumPy, pandas frequently deals with 1-D and 2-D data. However, we use two separate objects to deal with 1-D and 2-D data in pandas. For 1-D data, we use the `pandas.Series` objects, which we'll refer to simply as a Series.

A Series is created through the `pd.Series` constructor, which takes in no required arguments but does have a variety of keyword arguments.

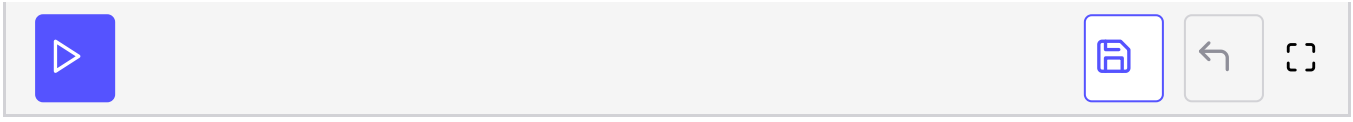
The first keyword argument is `data`, which specifies the elements of the Series. If `data` is not set, `pd.Series` returns an empty Series. Since the `data` keyword argument is almost always used, we treat it like a regular first argument (i.e. skip the `data=` prefix).

Similar to the `np.array` constructor, `pd.Series` also takes in the `dtype` keyword argument for manual casting.

The code below shows how to create pandas Series objects using `pd.Series`.

```
series = pd.Series()
# Newline to separate series print statements
print('{}\n'.format(series))

series = pd.Series(5)
print('{}\n'.format(series))

series = pd.Series([1, 2, 3])
print('{}\n'.format(series))

series = pd.Series([1, 2.2]) # upcasting
print('{}\n'.format(series))
```

```
arr = np.array([1, 2])
series = pd.Series(arr, dtype=np.float32)
print('{}\n'.format(series))

series = pd.Series([[1, 2], [3, 4]])
print('{}\n'.format(series))
```



In our examples, we initialized each Series with its values by setting the first argument using a scalar, list, or NumPy array. Note that `pd.Series` upcasts values in the same way as `np.array`. Furthermore, since Series objects are 1-D, the `ser` variable represents a Series with lists as elements, rather than a 2-D matrix.

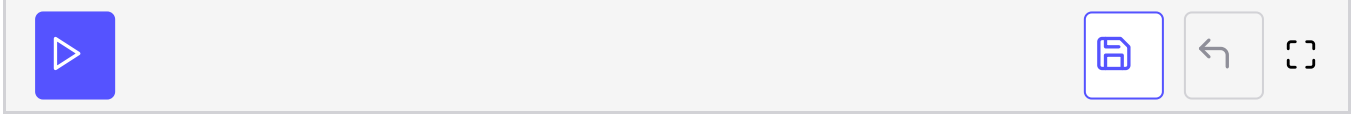
B. Index

In the previous examples, you may have noticed the zero-indexed integers to the left of the elements in each Series. These integers are collectively referred to as the *index* of a Series, and each individual index element is referred to as a *label*.

The default index is integers from 0 to n - 1, where n is the number of elements in the Series. However, we can specify a custom index via the `index` keyword argument of `pd.Series`.

The code below shows how to use the `index` keyword argument with `pd.Series`.

```
series = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
print('{}\n'.format(series))

series = pd.Series([1, 2, 3], index=['a', 8, 0.3])
print('{}\n'.format(series))
```



The `index` keyword argument needs to be a list or array with the same length as the `data` argument for `pd.Series`. The values in the `index` list can be any hashable type (e.g. integer, float, string).

C. Dictionary input

Another way to set the index of a Series is by using a Python dictionary for the `data` argument. The keys of the dictionary represent the index of the Series, while each individual key is the label for its corresponding value.

The code below shows how to use `pd.Series` with a Python dictionary as the first argument. In our example, we set `'a'`, `'b'`, and `'c'` as the Series index, with corresponding values `1`, `2`, and `3`.

```
series = pd.Series({'a':1, 'b':2, 'c':3})
print('{}\n'.format(series))

series = pd.Series({'b':2, 'a':1, 'c':3})
print('{}\n'.format(series))
```

Time to Code!

The coding exercise for this chapter involves creating various pandas Series objects.

The first Series we create will contain basic floating point numbers. The list we use to initialize the Series is `[1, 3, 5.2]`.

Set `s1` equal to `pd.Series` with the specified list as the only argument.

```
# CODE HERE
```

The second Series we create comes from performing elemental multiplication on `s1` using a separate list of floating point numbers.

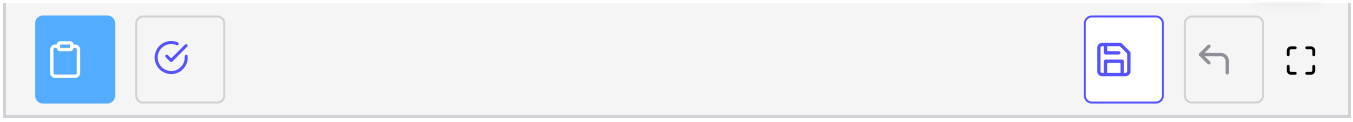
Set `s2` equal to `s1` multiplied by `[0.1, 0.2, 0.3]`.

```
# CODE HERE
```

We'll create another Series, this time with integers. The list we use to initialize this Series is `[1, 3, 8, np.nan]`. This Series will also have row labels, which will be `['a', 'b', 'c', 'd']`.

Set `s3` equal to `pd.Series` with the specified list of integers as the first argument and the list of labels as the `index` keyword argument.

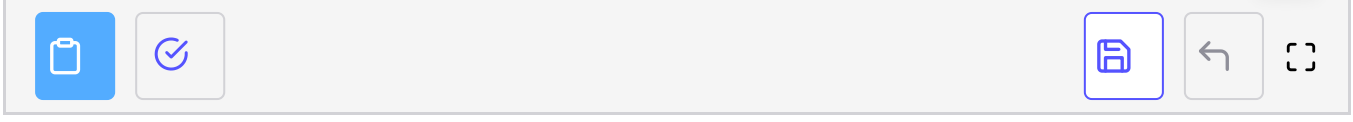
```
# CODE HERE
```



The final Series we create will be initialized from a Python dictionary. The dictionary will have key-value pairs `'a':0`, `'b':1`, and `'c':2`.

Set `s4` equal to `pd.Series` with a dictionary of the specified key-value pairs as the only argument.

```
# CODE HERE
```



DataFrame

Learn about the pandas DataFrame object for 2-D data.

Chapter Goals:

- Learn about the pandas DataFrame object and its basic utilities
- Write code to create and manipulate a pandas DataFrame

A. 2-D data

One of the main purposes of pandas is to deal with tabular data, i.e. data that comes from tables or spreadsheets. Since tabular data contains rows and columns, it is 2-D. For working with 2-D data, we use the `pandas.DataFrame` object, which we'll refer to simply as a DataFrame.

A DataFrame is created through the `pd.DataFrame` constructor, which takes in essentially the same arguments as `pd.Series`. However, while a Series could be constructed from a scalar (representing a single value Series), a DataFrame cannot.

Furthermore, `pd.DataFrame` takes in an additional `columns` keyword argument, which represents the labels for the columns (similar to how `index` represents the row labels).

The code below shows how to use the `pd.DataFrame` constructor.

```
df = pd.DataFrame()
# Newline added to separate DataFrames
print('{}\n'.format(df))

df = pd.DataFrame([5, 6])
print('{}\n'.format(df))

df = pd.DataFrame([[5,6]])
print('{}\n'.format(df))

df = pd.DataFrame([[5, 6], [1, 3]],
                  index=['r1', 'r2'],
                  columns=['c1', 'c2'])
print('{}\n'.format(df))
```

```
df = pd.DataFrame({'c1': [1, 2], 'c2': [3, 4]},

                  index=['r1', 'r2'])
print('{}\n'.format(df))
```

Note that when we use a Python dictionary for initialization, the DataFrame takes the dictionary's keys as its column labels.

B. Upcasting

When we initialize a DataFrame of mixed types, upcasting occurs on a per-column basis. The `dtypes` property returns the types in each column as a Series of types.

The code below shows how upcasting works in DataFrames. You'll notice that upcasting only occurs in the first column for the DataFrame below, because the second column's values are both integers.

```
upcast = pd.DataFrame([[5, 6], [1.2, 3]])
print('{}\n'.format(upcast))
# Datatypes of each column
print(upcast.dtypes)
```

C. Appending rows

We can append additional rows to a given DataFrame through the `append` function. The required argument for the function is either a Series or DataFrame, representing the row(s) we append.

Note that the `append` function returns the modified DataFrame but doesn't actually change the original. Furthermore, when we append a Series to the DataFrame, we either need to specify the `name` for the series or use the `ignore_index` keyword argument. Setting `ignore_index=True` will change the row labels to integer indexes.

The code below shows example usages of the `append` function.

```
df = pd.DataFrame([[5, 6], [1.2, 3]])
ser = pd.Series([0, 0], name='r3')

df_app = df.append(ser)
print('{}\n'.format(df_app))

df_app = df.append(ser, ignore_index=True)
print('{}\n'.format(df_app))

df2 = pd.DataFrame([[0,0],[9,9]])
df_app = df.append(df2)
print('{}\n'.format(df_app))
```

D. Dropping data

We can drop rows or columns from a given DataFrame through the `drop` function. There is no required argument, but the keyword arguments of the function gives us two ways to drop rows/columns from a DataFrame.

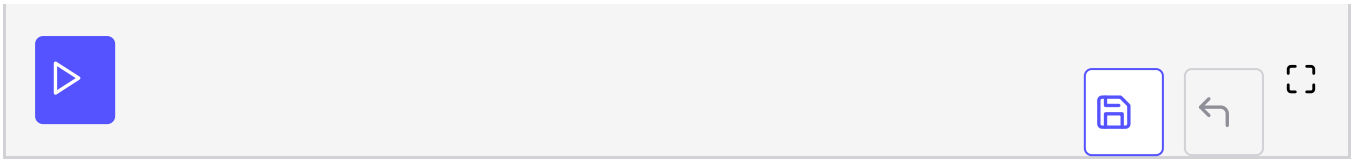
The first way is using the `labels` keyword argument to specify the labels of the rows/columns we want to drop. We use this alongside the `axis` keyword argument (which has default value of `0`) to drop from the rows or columns axis.

The second method is to directly use the `index` or `columns` keyword arguments to specify the labels of the rows or columns directly, without needing to use `axis`.

The code below shows examples on how to use the `drop` function.

```
df = pd.DataFrame({'c1': [1, 2], 'c2': [3, 4],
                   'c3': [5, 6]},
                  index=['r1', 'r2'])
# Drop row r1
df_drop = df.drop(labels='r1')
print('{}\n'.format(df_drop))

# Drop columns c1, c3
df_drop = df.drop(labels=['c1', 'c3'], axis=1)
print('{}\n'.format(df_drop))

df_drop = df.drop(index='r2')
print('{}\n'.format(df_drop))

df_drop = df.drop(columns='c2')
print('{}\n'.format(df_drop))

df.drop(index='r2', columns='c2')
```

```
print('{}\n'.format(df_drop))
```



Similar to `append`, the `drop` function returns the modified DataFrame but doesn't actually change the original.

Note that when using `labels` and `axis`, we can't drop both rows and columns from the DataFrame.

Time to Code!

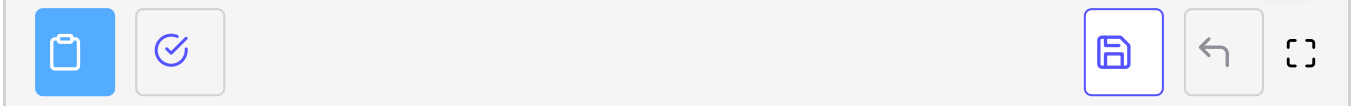
The coding exercise for this chapter involves creating various pandas DataFrame objects.

We'll first create a DataFrame from a Python dictionary. The dictionary will have key-value pairs `'c1':[0, 1, 2, 3]` and `'c2':[5, 6, 7, 8]`, in that order.

The index for the DataFrame will come from the list of row labels `['r1', 'r2', 'r3', 'r4']`.

Set `df` equal to `pd.DataFrame` with the specified dictionary as the first argument and the list of row labels as the `index` keyword argument.

```
# CODE HERE
```



We'll create another DataFrame, this one representing a single row. Rather than a dictionary for the first argument, we use a list of lists, and manually set the column labels to `['c1, 'c2']`.

Since there is only one row, the row labels will be `['r5']`.

Set `row_df` equal to `pd.DataFrame` with `[[9, 9]]` as the first argument, and the specified column and row labels for the `columns` and `index` keyword arguments.

```
# CODE HERE
```

After creating `row_df`, we append it to the end of `df` and drop row `'r2'`.

Set `df_app` equal to `df.append` with `row_df` as the only argument.

Then set `df_drop` equal to `df_app.drop` with `'r2'` as the `labels` keyword argument.

```
# CODE HERE
```

Combining

Combine multiple DataFrames through concatenation and merging.

Chapter Goals:

- Understand the methods used to combine DataFrame objects
- Write code for combining DataFrames

In the previous chapter, we discussed the `append` function for concatenating DataFrame rows. To concatenate multiple DataFrames along either rows or columns, we use the `pd.concat` function.

The code below shows example usages of `pd.concat`.

```
df1 = pd.DataFrame({'c1':[1,2], 'c2':[3,4]},
                   index=['r1','r2'])
df2 = pd.DataFrame({'c1':[5,6], 'c2':[7,8]},
                   index=['r1','r2'])
df3 = pd.DataFrame({'c1':[5,6], 'c2':[7,8]})

concat = pd.concat([df1, df2], axis=1)
# Newline to separate print statements
print('{}\n'.format(concat))

concat = pd.concat([df2, df1, df3])
print('{}\n'.format(concat))

concat = pd.concat([df1, df3], axis=1)
print('{}\n'.format(concat))
```

The `pd.concat` function takes in a list of pandas objects (normally a list of DataFrames) to concatenate. The function also takes in numerous keyword arguments, with `axis` being one of the more important ones. The `axis` argument specifies whether we concatenate the rows (`axis=0`, the default), or concatenate the columns (`axis=1`).

This works very similarly to concatenation in NumPy

In the code example, the final call to `pd.concat` resulted in a DataFrame with many `NaN` values. This is because the row labels for `df1` and `df3` did not match, so result was padded with `NaN` in locations where values did not exist.

B. Merging

Apart from combining DataFrames through concatenation, we can also merge multiple DataFrames. The function we use is `pd.merge`, which takes in two DataFrames for its two required arguments.

The code below shows how to use `pd.merge`.

```
mlb_df1 = pd.DataFrame({'name': ['john doe', 'al smith', 'sam black', 'john doe'],
                        'pos': ['1B', 'C', 'P', '2B'],
                        'year': [2000, 2004, 2008, 2003]})
mlb_df2 = pd.DataFrame({'name': ['john doe', 'al smith', 'jack lee'],
                        'year': [2000, 2004, 2012],
                        'rbi': [80, 100, 12]})

print('{}\n'.format(mlb_df1))
print('{}\n'.format(mlb_df1))

mlb_merged = pd.merge(mlb_df1, mlb_df2)
print('{}\n'.format(mlb_merged))
```

Without using any keyword arguments, `pd.merge` joins two DataFrames using all their common column labels. In the code example, the common labels between `mlb_df1` and `mlb_df2` were `name` and `year`.

The rows that contain the exact same values for the common column labels will be merged. Since `'john doe'` for year `2000` was in both `mlb_df1` and `mlb_df2`, its row was merged. However, `'john doe'` for year `2003` was only in `mlb_df1`, so its row was not merged.

The `pd.merge` function takes in many keyword arguments, but often none are needed to properly merge two DataFrames.

Time to Code!

The coding exercises for this chapter involve completing small functions that

take in two DataFrame objects as input.

The first function, `concat_rows` will concatenate the rows of the two DataFrames.

Set `row_concat` equal to `pd.concat` with `[df1, df2]` as the only argument. Then return `row_concat`.

```
def concat_rows(df1, df2):
  # CODE HERE
  pass
```

The next function, `concat_cols` will concatenate the columns of the two input DataFrames.

Set `col_concat` equal to `pd.concat` with `[df1, df2]` as the required argument. Also set the `axis` keyword argument to `1`.

Then return `col_concat`.

```
def concat_cols(df1, df2):
  # CODE HERE
  pass
```

The final function, `merge_dfs` will merge the two input DataFrames along their columns.

Set `merged_df` equal to `pd.merge` with `df1` and `df2` as the first and second arguments, respectively.

Then return `merged_df`.

```
def merge_dfs(df1, df2):
  # CODE HERE
  pass
```

Indexing

Understand how DataFrame values can be accessed via indexing.

Chapter Goals:

- Learn how to index a DataFrame to retrieve rows and columns
- Write code for indexing a DataFrame

A. Direct indexing

When indexing into a DataFrame, we can treat the DataFrame as a dictionary of Series objects, where each column represents a Series. Each column label then becomes a key, allowing us to directly retrieve columns using dictionary-like bracket notation.

The code below shows how to directly index into a DataFrame's columns.

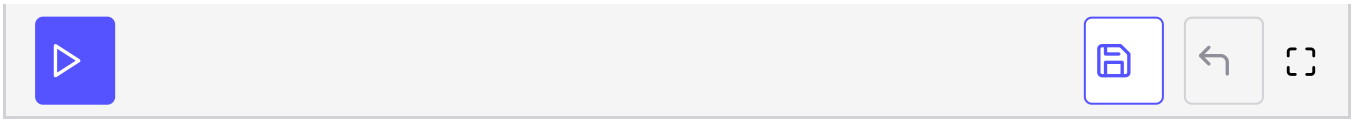
```
df = pd.DataFrame({'c1': [1, 2], 'c2': [3, 4],
                   'c3': [5, 6]}, index=['r1', 'r2'])
col1 = df['c1']
# Newline for separating print statements
print('{}\n'.format(col1))

col1_df = df[['c1']]
print('{}\n'.format(col1_df))

col23 = df[['c2', 'c3']]
print('{}\n'.format(col23))
```

Note that when we use a single column label inside the bracket (as was the case for `col1` in the code example), the output is a Series representing the corresponding column. When we use a list of column labels (as was the case for `col1_df` and `col23`), the output is a DataFrame that contains the corresponding columns.

We can also use direct indexing to retrieve a subset of the rows (as a

DataFrame). However, we can only retrieve rows based on slices, rather than specifying particular rows.
The code below shows how to directly index into a DataFrame's rows.

```
df = pd.DataFrame({'c1': [1, 2, 3], 'c2': [4, 5, 6],
                   'c3': [7, 8, 9]}, index=['r1', 'r2', 'r3'])

print('{}\n'.format(df))

first_two_rows = df[0:2]
print('{}\n'.format(first_two_rows))

last_two_rows = df['r2':'r3']
print('{}\n'.format(last_two_rows))

# Results in KeyError
df['r1']
```



You'll notice that when we used integer indexing for the rows, the end index was exclusive (e.g. `first_two_rows` excluded the row at index 2). However, when we use row labels, the end index is inclusive (e.g. `last_two_rows` included the row labeled `'r3'`).

Furthermore, when we tried to retrieve a single row based on its label, we received a `KeyError`. This is because the DataFrame treated `'r1'` as a column label.

B. Other indexing

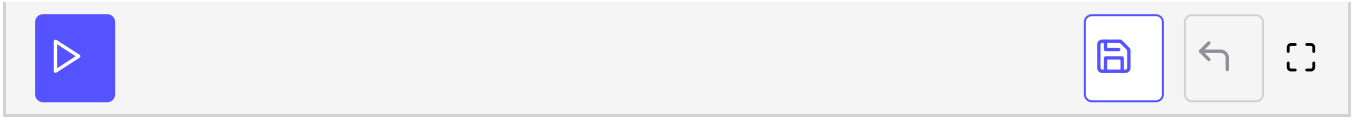
Apart from direct indexing, a DataFrame object also contains the `loc` and `iloc` properties for indexing.

We use `iloc` to access rows based on their integer index. Using `iloc` we can access a single row as a Series, and specify particular rows to access through a list of integers or a boolean array.

The code below shows how to use `iloc` to access a DataFrame's rows.

```
df = pd.DataFrame({'c1': [1, 2, 3], 'c2': [4, 5, 6],
                   'c3': [7, 8, 9]}, index=['r1', 'r2', 'r3'])

print('{}\n'.format(df))

print('{}\n'.format(df.iloc[1]))
```

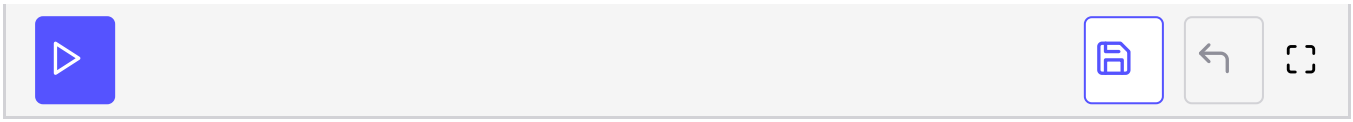
```
print('{}\n'.format(df.iloc[[0, 2]]))

bool_list = [False, True, True]
print('{}\n'.format(df.iloc[bool_list]))
```



The `loc` property provides the same row indexing functionality as `iloc`, but uses row labels rather than integer indexes. Furthermore, with `loc` we can perform column indexing along with row indexing, and set new values in a DataFrame for specific rows and columns.

The code below shows example usages of `loc`.

```
df = pd.DataFrame({'c1': [1, 2, 3], 'c2': [4, 5, 6],
                   'c3': [7, 8, 9]}, index=['r1', 'r2', 'r3'])

print('{}\n'.format(df))

print('{}\n'.format(df.loc['r2']))

bool_list = [False, True, True]
print('{}\n'.format(df.loc[bool_list]))

single_val = df.loc['r1', 'c2']
print('Single val: {}\n'.format(single_val))

print('{}\n'.format(df.loc[['r1', 'r3'], 'c2']))

df.loc[['r1', 'r3'], 'c2'] = 0
print('{}\n'.format(df))
```



You'll notice that the way we access rows and columns together with `loc` is similar to how we access 2-D NumPy arrays.

Since we can't access columns on their own with `loc` or `iloc`, we still use bracket indexing when retrieving columns of a DataFrame.

Time to Code!

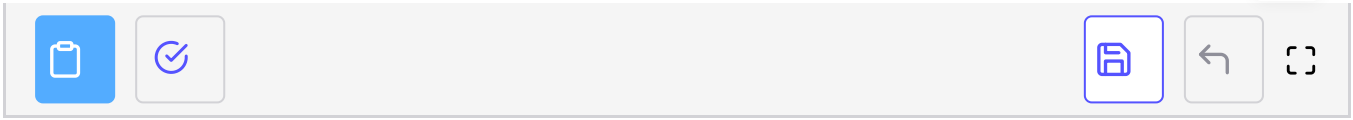
The coding exercises for this chapter involve directly indexing into a predefined DataFrame, `df`.

We'll initially use direct indexing to get the first column of `df` as well as the first two rows.

Set `col_1` equal to `df` directly indexed with `'c1'` as the key.

Set `row_12` equal to `df` directly indexed with `0:2` as the key.

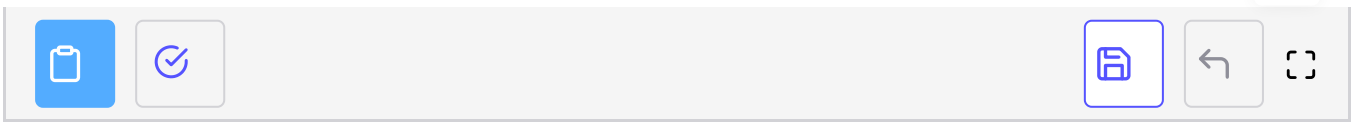
```
# CODE HERE
```



Next, we'll use `iloc` to retrieve the first and third rows of `df`.

Set `row_13` equal to `df.iloc` indexed with `[0, 2]` as the key.

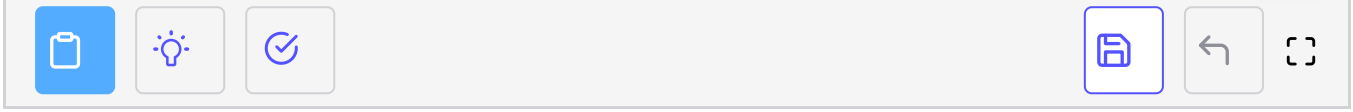
```
# CODE HERE
```



Finally, we use `loc` to set each value of the second column, in the third and fourth rows, equal to 12. The row key we use for indexing will be `['r3','r4']`, while the column key will be `'c2'`.

Set `df.loc`, indexed with the specified row and column keys, equal to `12`.

```
# CODE HERE
```



File I/O

Read from and write to different types of files in pandas.

Chapter Goals:

- Learn how to handle file input/output using pandas
- Write code for processing data files

A. Reading data

One of the most important features in pandas is the ability to read from data files. pandas accepts a variety of file formats, ranging from CSV and Excel spreadsheets to SQL and even HTML. A full list of the available file formats for pandas can be found here.

In this chapter we'll focus on three of the most common file types: CSV, XLSX (Microsoft Excel), and JSON. For reading data from a file, we use either the `read_csv`, `read_excel`, or `read_json` function, depending on the file type.

Each of the file reading functions takes in a file path as the only required argument. Each function has numerous keyword arguments, so we won't get into most of them. However, we'll still discuss a couple of the more commonly used keyword arguments.

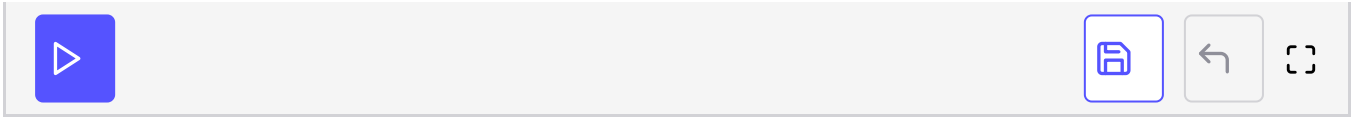
CSV

A CSV file is pretty straightforward; it's just comma-separated column names and values. When we don't use any keyword arguments, `pd.read_csv` returns a DataFrame with integer indexes as row labels, and each comma-separated column name as the column labels.

However, when we set the `index_col` keyword argument, we specify which column we want to use as the row labels. In our example, we used the first and second column as row labels.

The code below shows how to use `pd.read_csv`.

```
# data.csv contains baseball data
df = pd.read_csv('data.csv')
# Newline to separate print statements
print('{}\n'.format(df))

df = pd.read_csv('data.csv', index_col=0)
print('{}\n'.format(df))

df = pd.read_csv('data.csv', index_col=1)
print('{}\n'.format(df))
```



Excel

An Excel spreadsheet is similar to a CSV in its usage of rows and columns. However, the file path for `pd.read_excel` normally specifies an Excel workbook, which can contain multiple spreadsheets.

When we don't use any keyword arguments, the returned DataFrame from `pd.read_excel` contains the first sheet of the Excel workbook. However, when we set the `sheet_name` keyword argument, we can obtain a specific spreadsheet by passing in its integer index or name.

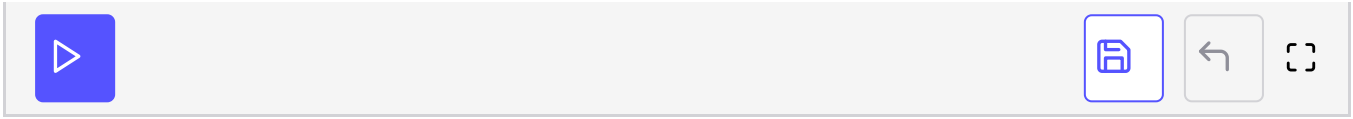
Furthermore, we obtain an ordered dictionary of spreadsheets by passing in a list of integers or sheet names. Setting `sheet_name=None` returns all the sheets in an ordered dictionary.

Like `pd.read_csv`, we can also specify the `index_col` argument in `pd.read_excel`.

The code below shows how to use `pd.read_excel`.

```
# data.csv contains baseball data
df = pd.read_excel('data.xlsx')
# Newline to separate print statements
print('{}\n'.format(df))

print('Sheet 1 (0-indexed) DataFrame:')
df = pd.read_excel('data.xlsx', sheet_name=1)
print('{}\n'.format(df))

print('MIL DataFrame:')
df = pd.read_excel('data.xlsx', sheet_name='MIL')
print('{}\n'.format(df))
```

```
# Sheets 0 and 1
df_dict = pd.read_excel('data.xlsx', sheet_name=[0, 1])
print('{}\n'.format(df_dict[1]))

# All Sheets
df_dict = pd.read_excel('data.xlsx', sheet_name=None)
print(df_dict.keys())
```



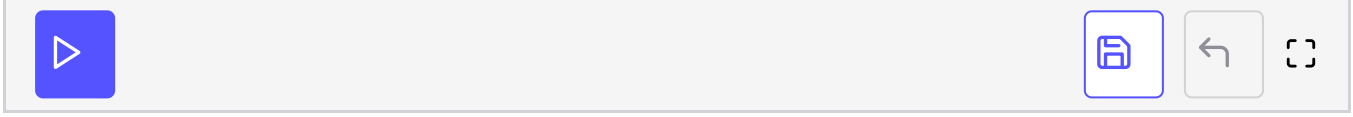
JSON

JSON data is pretty similar to a Python dictionary. In fact, you can use the `json` module (part of the Python standard library) to convert between dictionaries and JSON data. The file path for `pd.read_json` specifies a file containing JSON data.

When we don't use any keyword arguments, `pd.read_json` treats each outer key of the JSON data as a column label and each inner key as a row label. In the code example below, you can see `df1` treats the player's names as column labels.

However, when we set `orient='index'`, the outer keys are treated as row labels and the inner keys are treated as column labels.

```
# data is the JSON data (as a Python dict)
print('{}\n'.format(data))

df1 = pd.read_json('data.json')
print('{}\n'.format(df1))

df2 = pd.read_json('data.json', orient='index')
print('{}\n'.format(df2))
```



B. Writing to files

We can also use pandas to write data to a file. Focusing again on CSV, Excel, and JSON, the functions we use to write to files are `to_csv`, `to_excel`, and `to_json`.

Similar to the file reading functions, each of the writing functions has dozens of keyword arguments. Therefore, we'll only go over a few of the commonly used ones.

used ones.

CSV

Note that when we don't use any keyword arguments, `to_csv` will write the row labels as the first column in the CSV file. This is fine if the row labels are meaningful, but if they are just integers we don't really want them in the CSV file. In that case, we set `index=False`, to specify that we don't write the row labels into the CSV file.

The code below shows how to use `to_csv`.

```
# Predefined mlb_df
print('{}\n'.format(mlb_df))

# Index is kept when writing
mlb_df.to_csv('data.csv')
df = pd.read_csv('data.csv')
print('{}\n'.format(df))

# Index is not kept when writing
mlb_df.to_csv('data.csv', index=False)
df = pd.read_csv('data.csv')
print('{}\n'.format(df))
```

Excel

The basic `to_excel` function will only write a single DataFrame to a spreadsheet. However, if we want to write multiple spreadsheets in an Excel workbook, we first load the Excel file into a `pd.ExcelWriter` then use the ExcelWriter as the first argument to `to_excel`.

When we don't specify the `sheet_name` keyword argument, the Excel spreadsheet we write to is named `'Sheet1'`. We can pass in custom names into `sheet_name` to avoid constantly writing to `'Sheet1'`.

```
# Predefined DataFrames
print('{}\n'.format(mlb_df1))
print('{}\n'.format(mlb_df2))

with pd.ExcelWriter('data.xlsx') as writer:
  mlb_df1.to_excel(writer, index=False, sheet_name='NYY')
  mlb_df2.to_excel(writer, index=False, sheet_name='BOS')

df_dict = pd.read_excel('data.xlsx', sheet_name=None)
print(df_dict.keys())
print('{}\n'.format(df_dict['BOS']))
```

JSON

The `to_json` function also uses the `orient` keyword argument that was part of `pd.read_json`. Like in `pd.read_json`, setting `orient='index'` will set the outer keys of the JSON data to the row labels and the inner keys to the column labels.

The code below shows how to use `to_json`.

```
# Predefined df
print('{}\n'.format(df))

df.to_json('data.json')
df2 = pd.read_json('data.json')
print('{}\n'.format(df2))

df.to_json('data.json', orient='index')
df2 = pd.read_json('data.json')
print('{}\n'.format(df2))
df2 = pd.read_json('data.json', orient='index')
print('{}\n'.format(df2))
```

Time to Code!

The coding exercises in this chapter reading from CSV files containing baseball data, manipulating the data, then writing the resulting data into a new CSV file.

First, we'll read from the two CSV files `'stats.csv'` and `'salary.csv'`. These files contain the stats and salaries, respectively, of various baseball players.

Set `stats_df` equal to `pd.read_csv` applied to `'stats.csv'`.

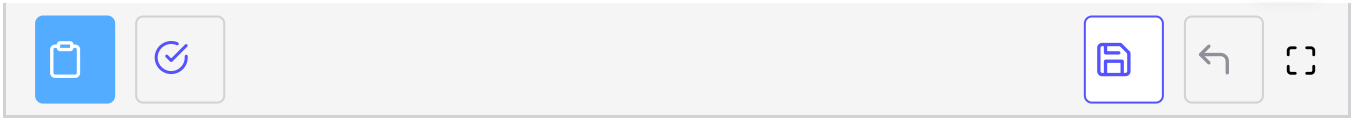
Set `salary_df` equal to `pd.read_csv` applied to `'salary.csv'`.

```
# CODE HERE
```

Rather than having two separate DataFrames, we want a single DataFrame that contains the yearly stats and salaries for each player. Therefore, we can just merge the `stats_df` and `salary_df` DataFrames.

Set `df` equal to `pd.merge` with `stats_df` and `salary_df` as the first two arguments, in that order.

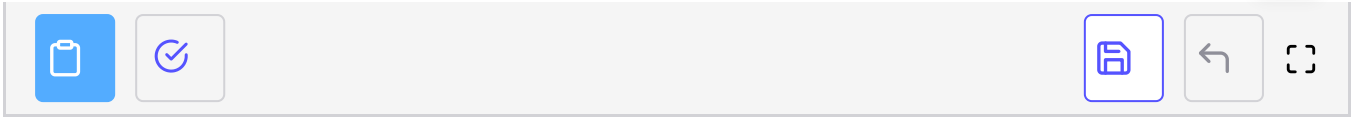
```
# CODE HERE
```



Finally, we write the merged DataFrame into the file named `'out.csv'`. Since the original CSV files didn't label the rows, we'll make sure not to label the rows of `'out.csv'`.

Call `df.to_csv` with `'out.csv'` as the first argument and `False` for the `index` keyword argument.

```
# CODE HERE
```



Grouping

Learn how DataFrames can be grouped based on particular columns.

Chapter Goals:

- Learn how to group DataFrames by columns
- Write code to retrieve home run statistics through DataFrame grouping

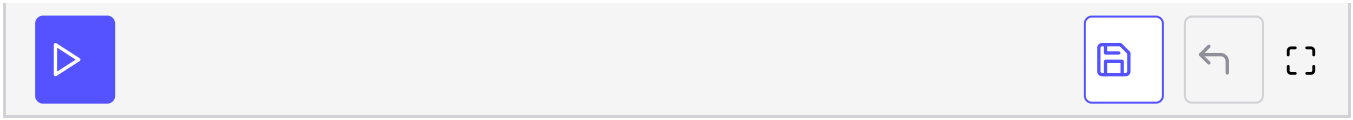
A. Grouping by column

When dealing with large amounts of data, it is usually a good idea to group the data by common categories. For example, we could group a large dataset of MLB player statistics by year, so we can deal with each year's data separately.

With pandas DataFrames, we can perform dataset grouping with the `groupby` function. A common usage of the function is to group a DataFrame by values from a particular column, e.g. a column representing years.

The code below shows how to use the `groupby` function, with the example of grouping MLB data by year.

```
# Predefined df of MLB stats
print('{}\n'.format(df))

groups = df.groupby('yearID')
for name, group in groups:
  print('Year: {}'.format(name))
  print('{}\n'.format(group))

print('{}\n'.format(groups.get_group(2016)))
print('{}\n'.format(groups.sum()))
print('{}\n'.format(groups.mean()))
```



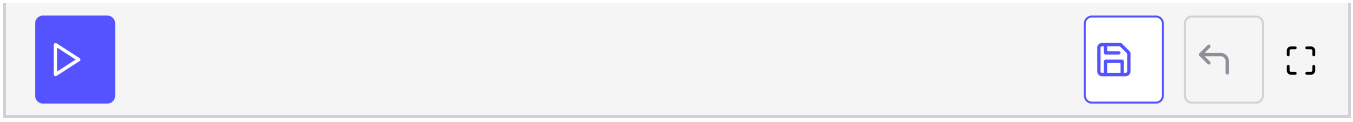
The grouping code example produced three DataFrames for the years 2015, 2016, and 2017. The three DataFrame groups are contained in the `groups`

variable, and we used its `sum` and `mean` functions to retrieve the total and average per-year statistics.

In addition to aggregation functions like `sum` and `mean`, we can also filter the groups using `filter`. The `filter` function takes in another function as its required argument, which specifies how we want to filter the groups. The output of `filter` is the concatenation of all the groups that pass the filter.

The code below shows how to use the `filter` function.

```
no2015 = groups.filter(lambda x: x.name > 2015)
print(no2015)
```



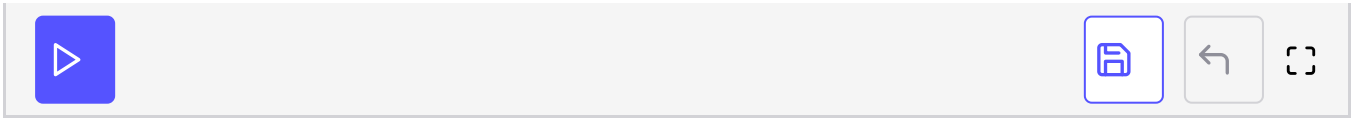
In the above code example, the lambda function passed into `filter` returns `True` if the group (represented as `x`) represents a year greater than 2015. The output is the concatenation of the 2016 and 2017 groups.

B. Multiple columns

DataFrame grouping is not just limited to a single column. Rather than passing a single column label into `groupby`, we can use a list of column labels to specify grouping by multiple columns.

Grouping by multiple columns can be useful when multiple data features have many different values. For example, if our dataset consisted of MLB players, grouping by both team and year would give us an organized way to view a team's roster throughout the years.

```
# player_df is predefined
groups = player_df.groupby(['yearID', 'teamID'])

for name, group in groups:
  print('Year, Team: {}'.format(name))
  print('{}\n'.format(group))

print(groups.sum())
```



In the code above, we grouped the MLB data by both year and team, resulting

in each group's name being a tuple of year and team. Using the `sum` function, we obtained the annual total hits for each team.

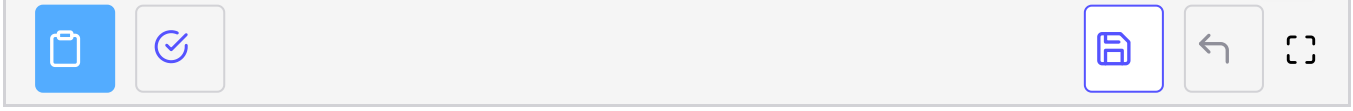
Time to Code!

The coding exercises for this chapter involve performing grouping operations on `df`, which contains all MLB batting data from 1871-2017. Using `df`, our goal is to retrieve home run (HR) statistics for 2017.

To do this, we need to calculate the total number of home runs hit each year. This involves first grouping `df` by year.

Set `year_group` equal to `df.groupby` with `'yearID'` as the lone argument.

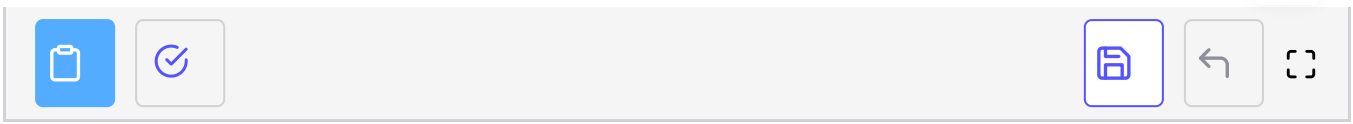
```
# CODE HERE
```



The yearly stats can be obtained from summing the values across the year-separated groups.

Set `year_stats` equal to `year_group.sum` applied with no arguments.

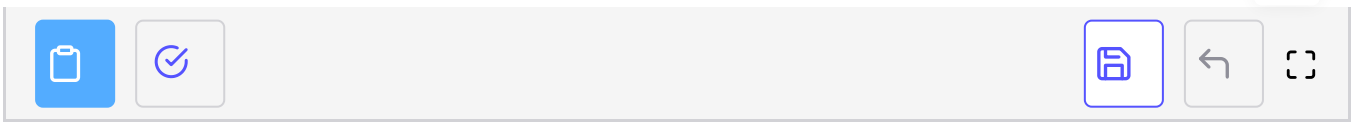
```
# CODE HERE
```



The `year_stats` DataFrame represents the total value for each stat per year. The row labels are the years and the column labels are the stat categories, e.g. home runs. Using the `loc` property, we'll retrieve the home run total for 2017.

Set `hr_2017` equal to `year_stats.loc` with `2017` as the row index and `'HR'` as the column index.

```
# CODE HERE
```

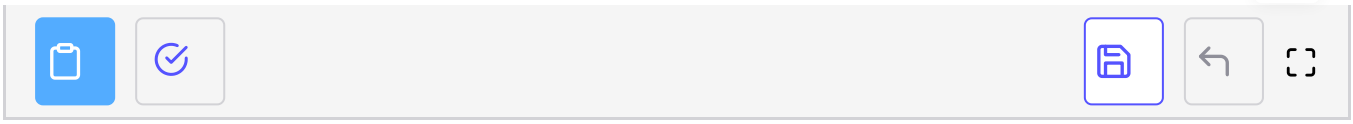


Next we want to get the yearly totals for each batting statistic *per team*. To do

this, we group the data by both the year and team.

Set `year_team_group` equal to `df.groupby` applied with the list `['yearID', 'teamID']`.

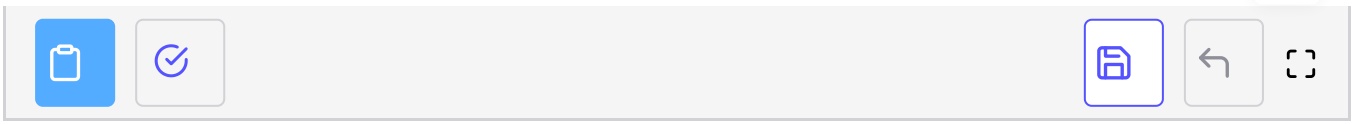
```
# CODE HERE
```



Once again, to obtain the yearly stats we just sum over all the groups.

Set `year_team_stats` equal to `year_team_group.sum` applied with no arguments.

```
# CODE HERE
```



Features

Learn about the different feature types that can be part of a dataset.

Chapter Goals:

- Understand the difference between quantitative and categorical features
- Learn methods to manipulate features and add them to a DataFrame
- Write code to add MLB statistics to a DataFrame

A. Quantitative vs. categorical

We often refer to the columns of a DataFrame as the *features* of the dataset that it represents. These features can be quantitative or categorical.

A quantitative feature, e.g. height or weight, is a feature that can be measured numerically. These are features we could calculate the sum, mean, or other numerical metrics for.

A categorical feature, e.g. gender or birthplace, is one where the values are categories that could be used to group the dataset. These are the features we would use with the `groupby` function from the previous chapter.

Some features can be both quantitative or categorical, depending on the context they are used. For example, we could use year of birth as a quantitative feature if we were trying to find statistics such as the average birth year for a particular dataset. On the other hand, we could also use it as a categorical feature and group the data by the different years of birth.

B. Quantitative features

In the previous chapter, we focused on grouping a dataset by its categorical features. We'll now describe methods for dealing with quantitative features.

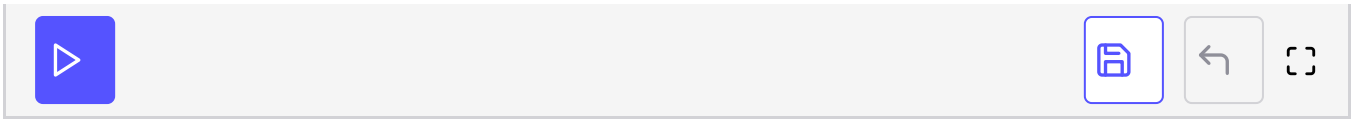
Two of the most important functions to use with quantitative features are `sum` and `mean`. In the previous chapter we also introduced `sum` and `mean` functions, which were used to aggregate quantitative features for each a

group.

However, while the functions from the previous chapter were applied to the output of `groupby`, the ones we use in this chapter are applied to individual DataFrames.

The code below shows example usages of `sum` and `mean`. The `df` DataFrame represents three different speed tests (columns) for three different processors (rows). The data values correspond to the seconds taken for a given speed test and processor.

```
df = pd.DataFrame({
  'T1': [10, 15, 8],
  'T2': [25, 27, 25],
  'T3': [16, 15, 10]})

print('{}\n'.format(df))

print('{}\n'.format(df.sum()))

print('{}\n'.format(df.sum(axis=1)))

print('{}\n'.format(df.mean()))

print('{}\n'.format(df.mean(axis=1)))
```



Neither function takes in a required argument. The most commonly used keyword argument for both functions is `axis`. The `axis` argument specifies whether to aggregate over rows (`axis=0`, the default), or columns (`axis=1`).

In the code example, we used a DataFrame representing speed tests for three different processors (measured in seconds). When we used no argument, equivalent to using `axis=0`, the `sum` and `mean` functions calculated total and average times for each test. When we used `axis=1`, the `sum` and `mean` functions calculated total and average test times (across all three tests) for each processor.

C. Weighted features

Along with aggregating quantitative features, we can also apply weights to them. We do this through the `multiply` function.

The `multiply` function takes in a list of weights or a constant as its required argument. If a constant is used, the constant is multiplied across all the rows or columns (depending on the value of `axis`). If a list is used, then the position of each weight in the list corresponds to which row/column it is multiplied to.

In contrast with `sum` and `mean`, the default `axis` for `multiply` is the columns axis. Therefore, to multiply weights along the rows of a DataFrame, we need to explicitly set `axis=0`.

The code below shows example usages of `multiply`. The `df` DataFrame represents three different speed tests (columns) for two different processors (rows).

```
df = pd.DataFrame({
  'T1': [0.1, 150.],
  'T2': [0.25, 240.],
  'T3': [0.16, 100.]})

print('{}\n'.format(df))

print('{}\n'.format(df.multiply(2)))

df_ms = df.multiply([1000, 1], axis=0)
print('{}\n'.format(df_ms))

df_w = df_ms.multiply([1,0.5,1])
print('{}\n'.format(df_w))
print('{}\n'.format(df_w.sum(axis=1)))
```

In the code above, the test times for processor `'p1'` were measured in seconds, while the times for `'p2'` were in milliseconds. So we made all the times in milliseconds by multiplying the values of `'p1'` by `1000`.

Then we multiplied the values in `'T2'` by `0.5`, since those tests were done with two processors rather than one. This makes the final `sum` a *weighted sum* across the three columns.

Time to Code!

The code exercises for this chapter involves calculating various baseball statistics based on the values of other statistics. The `mlb_df` DataFrame is predefined, and contains all historic MLB hitting statistics.

We also provide a `col_list_sum` function. This is a utility function to calculate the sum of multiple columns across a DataFrame.

```
def col_list_sum(df, col_list, weights=None):
    col_df = df[col_list]
    if weights is not None:
        col_df = col_df.multiply(weights)
    return col_df.sum(axis=1)
```

The `df` argument is the input DataFrame, while the `col_list` argument is a list of column labels representing the columns we want to sum in `df`.

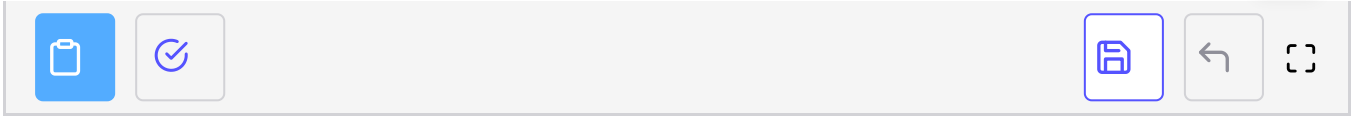
The `weights` keyword argument represents the weight coefficients we use for a weighted column sum. Note that if `weights` is not `None`, it must have the same length as `col_list`.

The `mlb_df` doesn't contain one of the key stats in baseball, batting average. Therefore, we'll calculate the batting average and add it as a column in `mlb_df`.

To calculate the batting average, simply divide a player's hits (`'H'`) by their number of at-bats (`'AB'`).

Set `mlb_df['BA']` equal to `mlb_df['H']` divided by `mlb_df['AB']`.

```
# CODE HERE
```



Though `mlb_df` contains columns for doubles, triples, and home runs (labeled `'2B'`, `'3B'`, `'HR'`), it does not contain a column for singles.

However, we can calculate singles by subtracting doubles, triples, and home runs from the total number of hits. We'll label the singles column as `'1B'`.

Set `other_hits` equal to `col_list_sum` with `mlb_df` as the first argument. The second argument should be a list of the column labels for doubles, triples, and home runs.

Set `mlb_df['1B']` equal to `mlb_df['H']` subtracted by `other_hits`.

```
# CODE HERE
```

Now that `mlb_df` contains columns for all four types of hits, we can calculate slugging percentage (column label `'SLG'`). The formula for slugging percentage is:

$$\text{SLG} = \frac{1\text{B} + 2 \cdot 2\text{B} + 3 \cdot 3\text{B} + 4 \cdot \text{HR}}{\text{AB}}$$

Therefore, the numerator represents a weighted sum with column labels `'1B'`, `'2B'`, `'3B'`, `'HR'`.

Set `weighted_hits` equal to `col_list_sum` with `mlb_df` as the first argument and a list of numerator column labels as the second argument. The `weights` keyword argument should be a list of integers from 1 to 4, inclusive.

```
# CODE HERE
```

We can now calculate the slugging percentage by dividing the weighted sum by the number of at-bats.

Set `mlb_df['SLG']` equal to `weighted_hits` divided by `mlb_df['AB']`.

```
# CODE HERE
```

Filtering

Filter DataFrames for values that fit certain conditions.

Chapter Goals:

- Understand how to filter a DataFrame based on filter conditions
- Write code to filter a dataset of MLB statistics

A. Filter conditions

In the Data Manipulation section, we used relation operations on NumPy arrays to create *filter conditions*. These filter conditions returned boolean arrays, which represented the locations of the elements that pass the filter.

In pandas, we can also create filter conditions for DataFrames. Specifically, we can use relation operations on a DataFrame's column features, which will return a boolean Series representing the DataFrame rows that pass the filter.

The code below demonstrates how to use relation operations as filter conditions.

```
df = pd.DataFrame({
  'playerID': ['bettsmo01', 'canoro01', 'cruzne02', 'ortizda01', 'cruzne02'],
  'yearID': [2016, 2016, 2016, 2016, 2017],
  'teamID': ['BOS', 'SEA', 'SEA', 'BOS', 'SEA'],
  'HR': [31, 39, 43, 38, 39]})

print('{}\n'.format(df))

cruzne02 = df['playerID'] == 'cruzne02'
print('{}\n'.format(cruzne02))

hr40 = df['HR'] > 40
print('{}\n'.format(hr40))

notbos = df['teamID'] != 'BOS'
print('{}\n'.format(notbos))
```

In the code above, we created filter conditions for `df` based on the columns labeled `'playerID'`, `'HR'`, and `'teamID'`. The boolean Series outputs have `True` for the rows that pass the filter, and `False` for the rows that don't.

B. Filters from functions

Apart from relation operations, pandas provides various functions for creating specific filter conditions. For columns with string values, we can use `str.startswith`, `str.endswith`, and `str.contains` to filter for specific strings. These functions work the exact same as their namesakes from the Python standard library.

The code below shows various examples of string filter conditions. In the final example using `str.contains`, we prepend the `~` operation, which negates the filter condition. This means our final filter condition checked for player IDs that *do not* contain `'o'`.

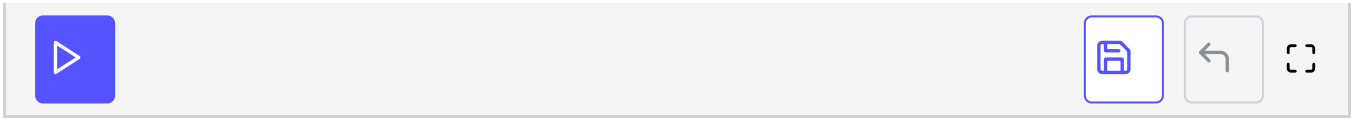
```
df = pd.DataFrame({
  'playerID': ['bettsmo01', 'canoro01', 'cruzne02', 'ortizda01', 'cruzne02'],
  'yearID': [2016, 2016, 2016, 2016, 2017],
  'teamID': ['BOS', 'SEA', 'SEA', 'BOS', 'SEA'],
  'HR': [31, 39, 43, 38, 39]})

print('{}\n'.format(df))

str_f1 = df['playerID'].str.startswith('c')
print('{}\n'.format(str_f1))

str_f2 = df['teamID'].str.endswith('S')
print('{}\n'.format(str_f2))

str_f3 = ~df['playerID'].str.contains('o')
print('{}\n'.format(str_f3))
```

We can also create filter conditions that check for values in a specific set, by using the `isin` function. The function only takes in one argument, which is a list of values that we want to filter for.

The code below demonstrates how to use the `isin` function for filter conditions.

```
df = pd.DataFrame({
  'playerID': ['bettsmo01', 'canoro01', 'cruzne02', 'ortizda01', 'cruzne02'],
```

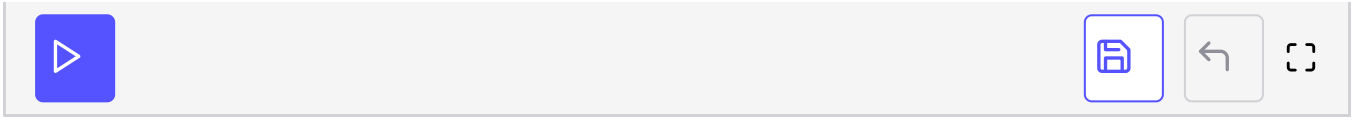
```
  'yearID': [2016, 2016, 2016, 2016, 2017],
  'teamID': ['BOS', 'SEA', 'SEA', 'BOS', 'SEA'],
  'HR': [31, 39, 43, 38, 39]})

print('{}\n'.format(df))

isin_f1 = df['playerID'].isin(['cruzne02',
                               'ortizda01'])
print('{}\n'.format(isin_f1))

isin_f2 = df['yearID'].isin([2015, 2017])
print('{}\n'.format(isin_f2))
```



In pandas, when a Series or DataFrame has a missing value at a location, it is represented by `NaN`. The `NaN` value in pandas is equivalent to `np.nan` in NumPy.

Similar to Numpy, we cannot use a relation operation to create a filter condition for `NaN` values. Instead, we use the `isna` and `notna` functions.

```
df = pd.DataFrame({
  'playerID': ['bettsmo01', 'canoro01', 'doejo01'],
  'yearID': [2016, 2016, 2017],
  'teamID': ['BOS', 'SEA', np.nan],
  'HR': [31, 39, 99]})

print('{}\n'.format(df))

isna = df['teamID'].isna()
print('{}\n'.format(isna))

notna = df['teamID'].notna()
print('{}\n'.format(notna))
```



The `isna` function returns `True` in the locations that contain `NaN` and `False` in the locations that don't, while the `notna` function does the opposite.

C. Feature filtering

It is really easy to filter a DataFrame's rows based on filter conditions. Similar to direct indexing of a DataFrame, we use square brackets. However, the inside of the square brackets will now contain a filter condition.

When applying filter conditions within square brackets, we retrieve the rows of the DataFrame that pass the filter condition (i.e. the rows for which the filter condition is `True`).

The code below shows how to filter using square brackets and filter conditions.

```
df = pd.DataFrame({
  'playerID': ['bettsmo01', 'canoro01', 'cruzne02', 'ortizda01', 'bettsmo01'],
  'yearID': [2016, 2016, 2016, 2016, 2015],
  'teamID': ['BOS', 'SEA', 'SEA', 'BOS', 'BOS'],
  'HR': [31, 39, 43, 38, 18]})

print('{}\n'.format(df))

hr40_df = df[df['HR'] > 40]
print('{}\n'.format(hr40_df))

not_hr30_df = df[~(df['HR'] > 30)]
print('{}\n'.format(not_hr30_df))

str_df = df[df['teamID'].str.startswith('B')]
print('{}\n'.format(str_df))
```

Time to Code!

In this chapter's code exercises, we'll apply various filters to a predefined DataFrame, `mlb_df`, which contains MLB statistics.

We'll first filter `mlb_df` for the top MLB hitting seasons in history, which we define as having a batting average above .300.

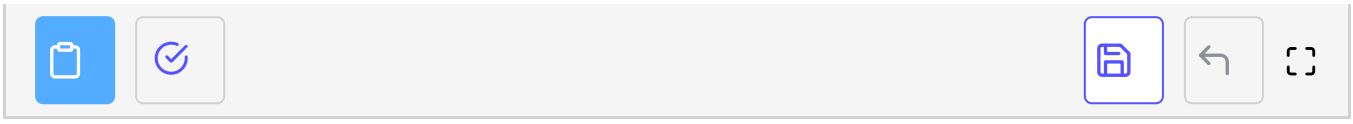
Set `top_hitters` equal to `mlb_df[]` applied with `mlb_df['BA'] > .300` as the filter condition.

```
# CODE HERE
```

Next we filter for the players whose player ID *does not* start with the letter **a**.

Set `exclude_a` equal to `mlb_df[]` applied with the negation of `mlb_df['playerID'].str.startswith('a')` as the filter condition.

```
# CODE HERE
```

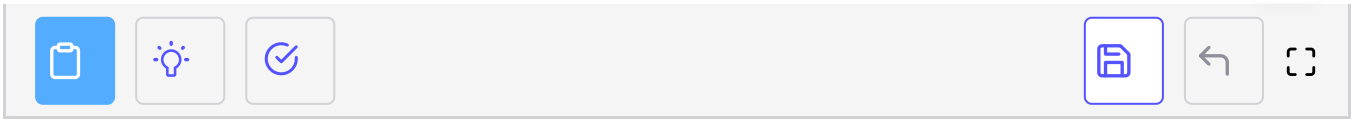


We'll now retrieve the statistics for two specific players. Their player IDs are `'bondsba01'` and `'troutmi01'`.

Set `two_ids` equal to a list containing the two specified player IDs.

Set `two_players` equal to `mlb_df[]` applied with `mlb_df['playerID'].isin(two_ids)` as the filter condition.

```
# CODE HERE
```



Sorting

Sort DataFrames based on their column features.

Chapter Goals:

- Learn how to sort a DataFrame by its features
- Write code to sort an MLB player's statistics

A. Sorting by feature

When we deal with a dataset that has many features, it is often useful to sort the dataset. This makes it easier to view the data and spot trends in the values.

In pandas, the `sort_values` function allows us to sort a DataFrame by one or more of its columns. The first argument is either a column label or a list of column labels to sort by.

The `ascending` keyword argument allows us to specify whether to sort in ascending or descending order (default is ascending order, i.e. `ascending=True`).

The code below demonstrates how to use `sort_values` with a single column label. The first example sorts by `'yearID'` in ascending order, while the second sorts `'playerID'` in descending lexicographic (alphabetical) order.

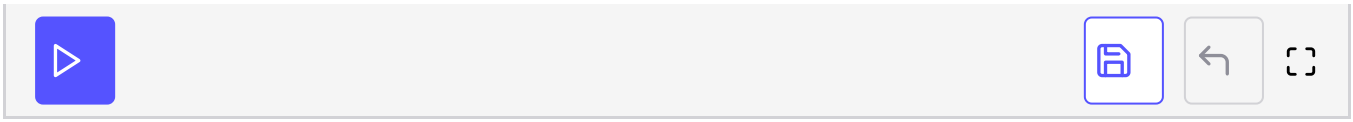
```
# df is predefined
print('{}\n'.format(df))

sort1 = df.sort_values('yearID')
print('{}\n'.format(sort1))

sort2 = df.sort_values('playerID', ascending=False)
print('{}\n'.format(sort2))
```

When sorting with a list of column labels, each additional label is used to

break ties. Specifically, label *i* in the list acts as a tiebreaker for label *i - 1*.

The code below demonstrates how to sort with a list of column labels.

```
# df is predefined
print('{}\n'.format(df))

sort1 = df.sort_values(['yearID', 'playerID'])
print('{}\n'.format(sort1))

sort2 = df.sort_values(['yearID', 'HR'],
                       ascending=[True, False])
print('{}\n'.format(sort2))
```



When using two column labels to sort, the list's first label represents the main sorting criterion, while the second label is used to break ties. In the example with sorting by `'yearID'` and `'playerID'`, the DataFrame is first sorted by year (in ascending order). For identical years, we sort again by player ID (in ascending order).

For multi-label inputs to `sort_values`, we are allowed to specify different sorting orders for each column label. In our second example, we specified that `'yearID'` would be sorted in ascending order, while `'HR'` would be sorted in descending order.

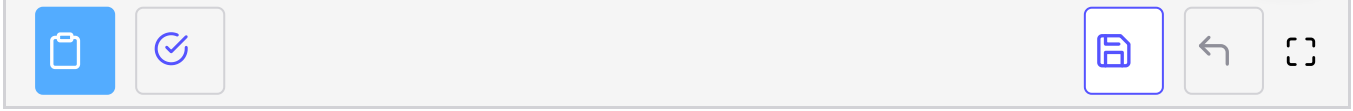
Time to Code!

The code exercises in this chapter involve sorting a DataFrame of yearly MLB player stats, `yearly_stats_df`.

We'll sort `yearly_stats_df` using two different methods. The first method sorts by `'yearID'` in ascending order.

Set `by_year` equal to `yearly_stats_df.sort_values` with `'yearID'` as the only argument.

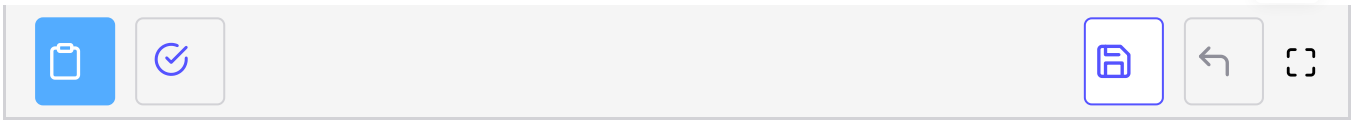
```
# CODE HERE
```



The next sorting method will sort by `'HR'` in descending order.

Set `best_hr` equal to `yearly_stats_df.sort_values` with `'HR'` as the first argument and the `ascending` keyword argument set to `False`.

```
# CODE HERE
```



The final sorting method will again sort `yearly_stats_df` by `'HR'` in descending order, but this time we break ties with `'SO'` in ascending order.

Set `best_hr_so` equal to `yearly_stats_df.sort_values` with a list of the specified column labels, in the order provided. The `ascending` keyword argument should be set to `[False, True]`.

```
# CODE HERE
```

Metrics

Use pandas to obtain statistical metrics for data.

Chapter Goals:

- Understand the common metrics used to summarize numeric data
- Learn how to describe categorical data using histograms

A. Numeric metrics

When working with numeric features, we usually want to calculate metrics such as mean, standard deviation, etc. These metrics give us more insight into the type of data we're working with, which benefits our overall analysis of the dataset.

Rather than calculating several different metrics separately, pandas provides the `describe` function to obtain a summary of a DataFrame's numeric data.

The metrics included in the output summary of `describe` are

Metric	Description
count	The number of rows in the DataFrame
mean	The mean value for a feature
std	The standard deviation for a feature
min	The minimum value in a feature
25%	The 25^{th} percentile of a feature

50%	The 50th percentile of a feature. Note that this is identical to the median
75%	The 75th percentile of a feature
max	The maximum value in a feature

The code below shows how to use the `describe` function.

```
# df is predefined
print('{}\n'.format(df))

metrics1 = df.describe()
print('{}\n'.format(metrics1))

hr_rbi = df[['HR','RBI']]
metrics2 = hr_rbi.describe()
print('{}\n'.format(metrics2))
```

Using `describe` with a DataFrame will return a summary of metrics for each of the DataFrame's numeric features. In our example, `df` had three features with numerical values: `yearID`, `HR`, and `RBI`.

Since we normally treat `yearID` as a categorical feature, the second time we used `describe` was with the `hr_rbi` DataFrame, which only included the `HR` and `RBI` features.

To have `describe` return specific percentiles, we can use the `percentiles` keyword argument. The `percentiles` argument takes in a list of decimal percentages, representing the percentiles we want returned in the summary.

```
metrics1 = hr_rbi.describe(percentiles=[.5])
print('{}\n'.format(metrics1))

metrics2 = hr_rbi.describe(percentiles=[.1])
print('{}\n'.format(metrics2))

metrics3 = hr_rbi.describe(percentiles=[.2,.8])
print('{}\n'.format(metrics3))
```

Note that the 50th percentile, i.e. the median, is always returned. The values specified in the `percentiles` list will replace the default 25th and 75th percentiles.

B. Categorical features

With categorical features, we don't calculate metrics like mean, standard deviation, etc. Instead, we use *frequency counts* to describe a categorical feature.

The frequency count for a specific category of a feature refers to how many times that category appears in the dataset. In pandas, we use the `value_counts` function to obtain the frequency counts for each category in a column feature.

The code below uses the `value_counts` function to get frequency counts of the `'playerID'` feature.

```
p_ids = df['playerID']
print('{}\n'.format(p_ids.value_counts()))

print('{}\n'.format(p_ids.value_counts(normalize=True)))

print('{}\n'.format(p_ids.value_counts(ascending=True)))
```

Using `value_counts` without any keyword arguments will return the frequency counts for each category, sorted in descending order.

Setting `normalize=True` returns the frequency proportions, rather than counts, for each category (note that the sum of all the proportions is 1). We can also set `ascending=True` to get the frequencies sorted in ascending order.

If we just want the names of each unique category in a column, rather than the frequencies, we use the `unique` function.

```
unique_players = df['playerID'].unique()
print('{}\n'.format(repr(unique_players)))
```

```
unique_teams = df['teamID'].unique()
print('{}\n'.format(repr(unique_teams)))
```

So far we've focused on categorical features with string values. However, categorical features can also have integer values. For example, we can use `yearID` as a categorical feature with each unique year as a separate category.

```
y_ids = df['yearID']
print('{}\n'.format(y_ids))

print('{}\n'.format(repr(y_ids.unique())))
print('{}\n'.format(y_ids.value_counts()))
```

Time to Code!

The coding exercises for this chapter involve getting metrics from a DataFrame of MLB players, `player_df`.

First, we'll get a summary of all the statistics in `player_df`.

Set `summary_all` equal to `player_df.describe` with no arguments.

```
# CODE HERE
```

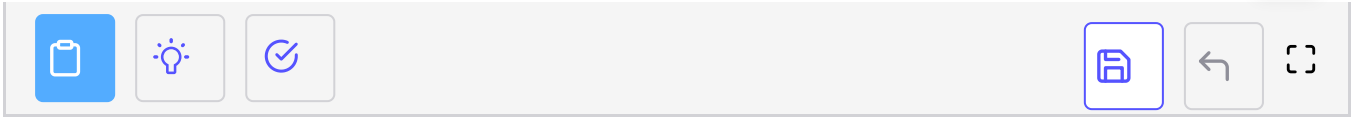
Next, we want to get summaries specifically for the home run totals. The first summary will contain the default metrics from `describe`, while the second summary will contain the 10^{th} and 90^{th} percentiles.

Set `hr_df` equal to `player_df[]` directly indexed with `'HR'`.

Set `summary_hr` equal to `hr_df.describe` with no arguments.

Set `low_high_10` equal to `hr_df.describe` with `[.1,.9]` as the `percentiles` keyword argument.

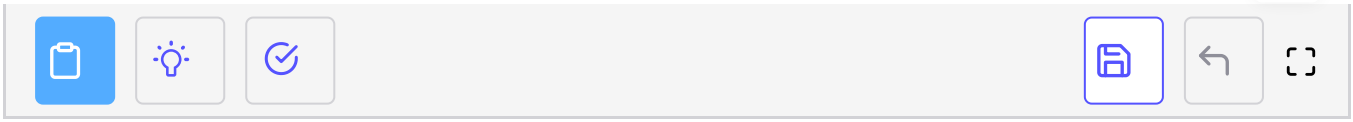
```
# CODE HERE
```



Finally, we'll treat the `'HR'` feature as a categorical variable, with each unique home run total as a separate category. We then get the frequency counts for each category.

Set `hr_counts` equal to `hr_df.value_counts` with no arguments.

```
# CODE HERE
```



Plotting

Learn how to plot DataFrames using the pyplot API from Matplotlib.

Chapter Goals:

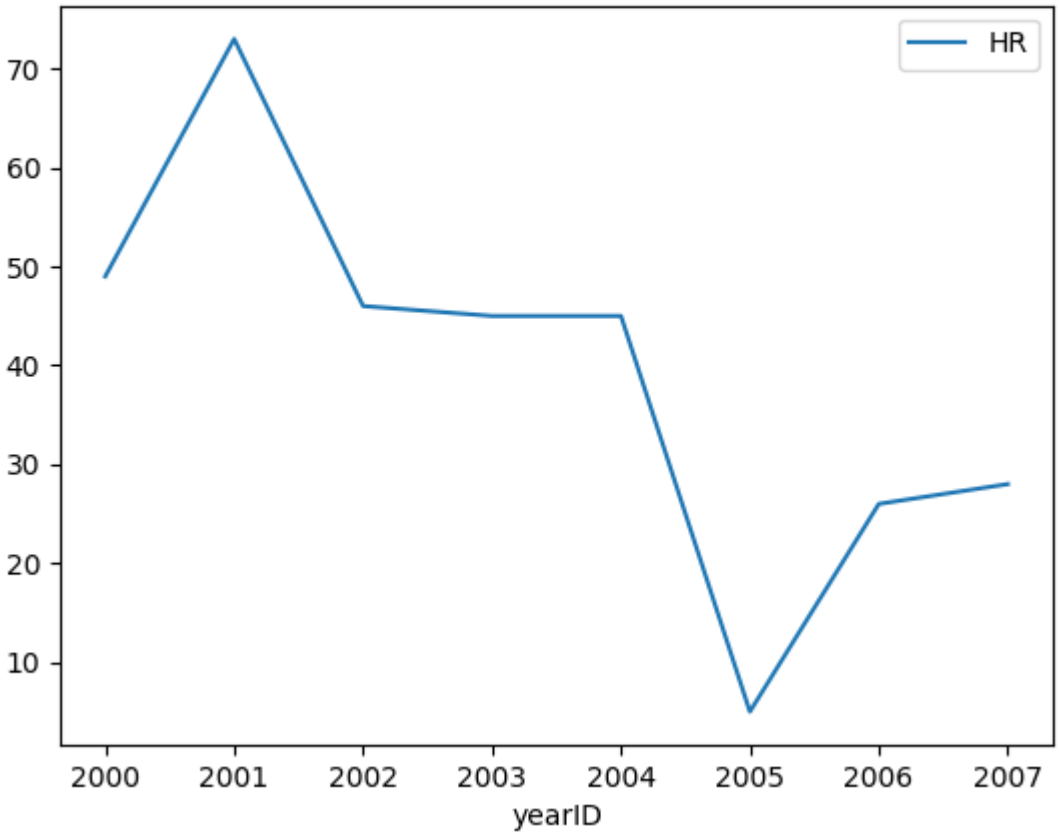
- Learn how to plot DataFrames using the pyplot API

A. Basics

The main function used for plotting DataFrames is `plot`. This function is used in tandem with the `show` function from the pyplot API, to produce plot visualizations. We import the pyplot API with the line:
`import matplotlib.pyplot as plt`

```
# predefined df
print('{}\n'.format(df))

df.plot()
plt.show()
```

The above code results in this plot:



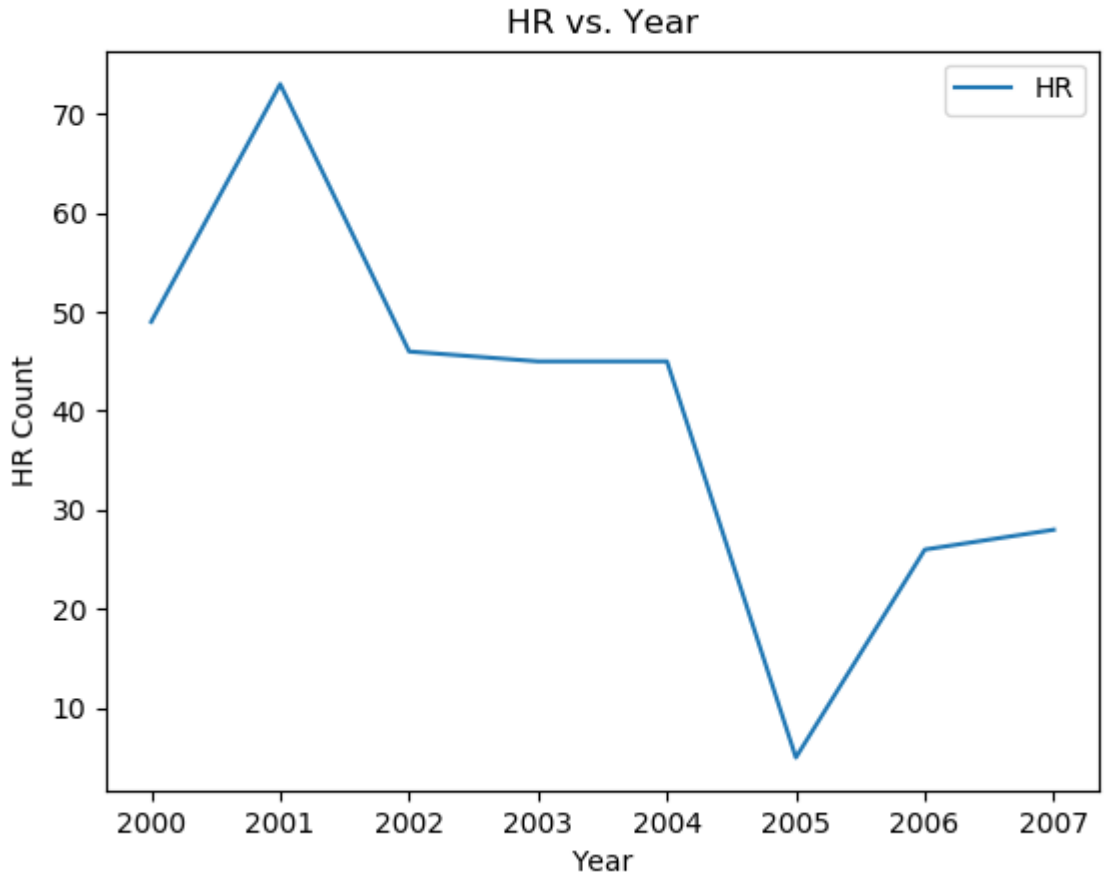
After calling `df.plot`, which creates our line plot, we then use `plt.show` to open a separate window containing the visualization of the plot. You can also use `plt.savefig` to save the plot to a PNG or PDF file.

```
# predefined df
print('{}\n'.format(df))

df.plot()
plt.savefig('df.png')  # save to PNG file
```

The plot we created has no title or *y*-axis label. We can manually set the plot's title and axis labels using the pyplot API.

```
# predefined df
print('{}\n'.format(df))

df.plot()
plt.title('HR vs. Year')
plt.xlabel('Year')
plt.ylabel('HR Count')
plt.show()
```

The above code results in this plot:



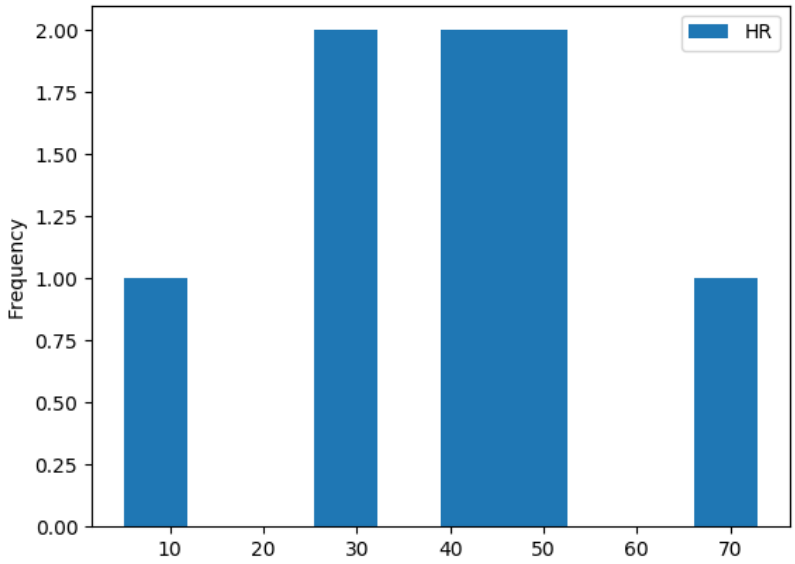
We use the `title` function to set the title of our plot, and the `xlabel` and `ylabel` functions to set the axis labels.

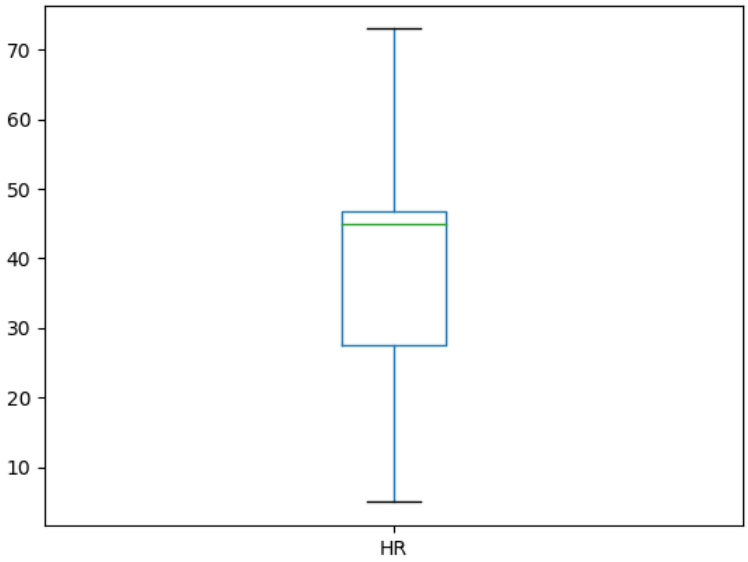
B. Other plots

In addition to basic line plots, we can create other plots like histograms or boxplots by setting the `kind` keyword argument in `plot`.

```
# predefined df
print('{}\n'.format(df))

df.plot(kind='hist')
df.plot(kind='box')
plt.show()
```

The above code results in these plots:





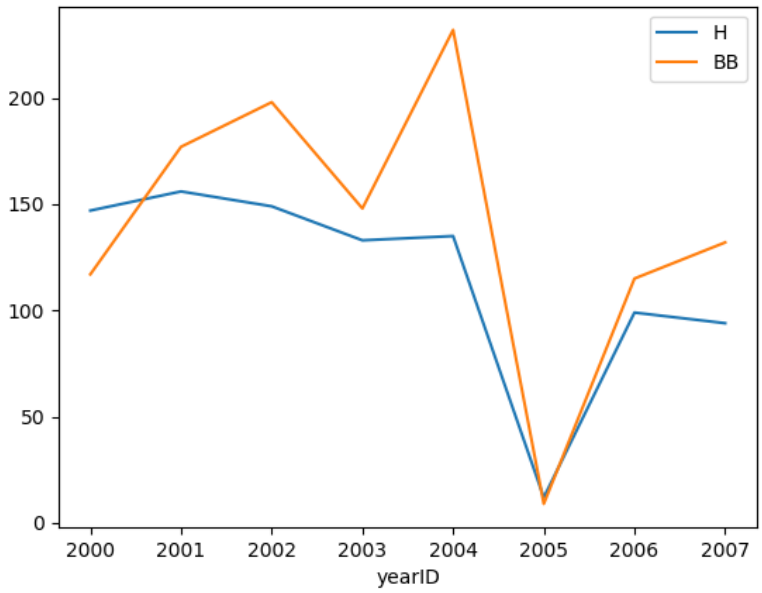
There are numerous different kinds of plots we can create by setting the `kind` keyword argument. A list of the accepted values for `kind` can be found in the documentation for `plot`.

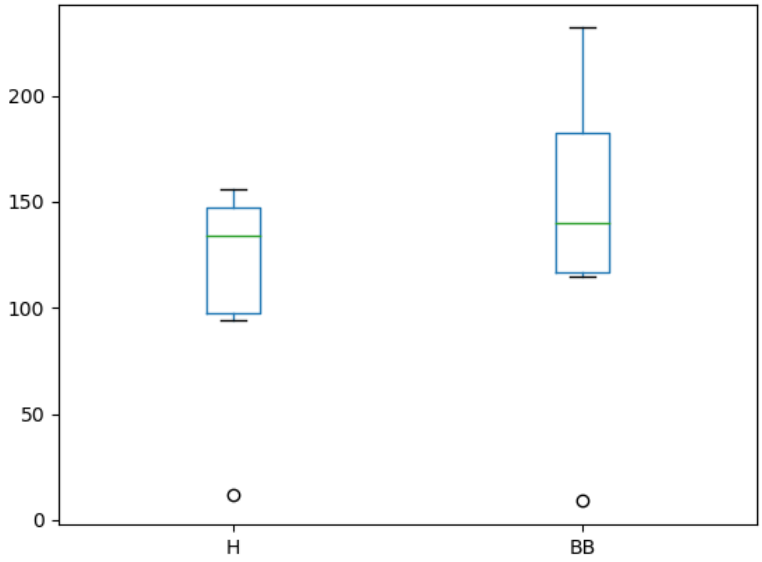
C. Multiple features

We can also plot multiple features on the same graph. This can be extremely useful when we want visualizations to compare different features.

```
# predefined df
print('{}\n'.format(df))

df.plot()
df.plot(kind='box')
plt.show()
```

The above code results in these plots:





These are a line plot and boxplot showing both hits (`H`) and walks (`BB`). Note that the circles in the boxplot represent outlier values.

To NumPy

Understand how DataFrames can be converted to 2-D NumPy arrays.

Chapter Goals:

- Learn how to convert a DataFrame to a NumPy matrix
- Write code to modify an MLB dataset and convert it to a NumPy matrix

A. Machine learning

The DataFrame object is great for storing a dataset and performing data analysis in Python. However, most machine learning frameworks (e.g. TensorFlow), work directly with NumPy data. Furthermore, the NumPy data used as input to machine learning models must solely contain quantitative values.

Therefore, to use a DataFrame's data with a machine learning model, we need to convert the DataFrame to a NumPy matrix of quantitative data. So even the categorical features of a DataFrame, such as gender and birthplace, must be converted to quantitative values.

B. Indicator features

When converting a DataFrame to a NumPy matrix of quantitative data, we need to find a way to modify the categorical features in the DataFrame.

The easiest way to do this is to convert each categorical feature into a set of *indicator features* for each of its categories. The indicator feature for a specific category represents whether or not a given data sample belongs to that category.

The code below shows a DataFrame with indicator features.

```
# predefined non-indicator DataFrame
print('{}\n'.format(df))

# predefined indicator Dataframe
```

```
print('{}\n'.format(indicator_df))
```

In the code above, the DataFrame `df` has a single categorical feature called `Color`. The corresponding indicator features for `Color` are shown in `indicator_df`.

Note that an indicator feature contains `1` when the row has that particular category, and `0` if the row does not.

C. Converting to indicators

In pandas, we convert each categorical feature of a DataFrame to indicator features with the `get_dummies` function. The function takes in a DataFrame as its required argument, and returns the DataFrame with each of its categorical features converted to indicator features.

The code below demonstrates how to use the `get_dummies` function.

```
# predefined df
print('{}\n'.format(df))

converted = pd.get_dummies(df)
print('{}\n'.format(converted.columns))

print('{}\n'.format(converted[['teamID_BOS',
                               'teamID_PIT']]))
print('{}\n'.format(converted[['lgID_AL',
                               'lgID_NL']]))
```

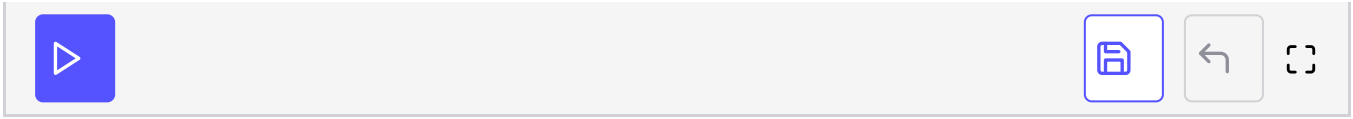
Note that the indicator features have the original categorical feature's label as a prefix. This makes it easy to see where each indicator feature originally came from.

D. Converting to NumPy

After converting all the categorical features to indicator features, the DataFrame should have all quantitative data. We can then convert to a NumPy matrix using the `values` function.

The code below converts a DataFrame, `df` into a NumPy matrix.

```
# predefined indicator df
print('{}\n'.format(df))

n_matrix = df.values
print(repr(n_matrix))
```



The rows and columns of the output matrix correspond to the rows and columns of the same position in the DataFrame. In the code above, the first column of the NumPy matrix represents `HR`, the second column represents `teamID_BOS`, and the third column represents `teamID_PIT`.

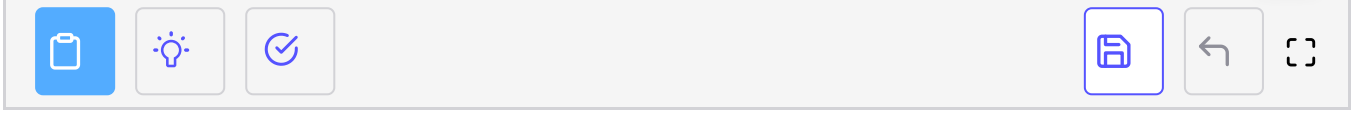
Time to Code!

The code exercise for this chapter will be to convert a DataFrame of MLB statistics (`df`) into a NumPy matrix.

We only want the data in `df` to be from the current century, so we need to first apply a filter.

Filter `df` for rows where `'yearID'` is at least `2000`, then reset `df` equal to the filtered output.

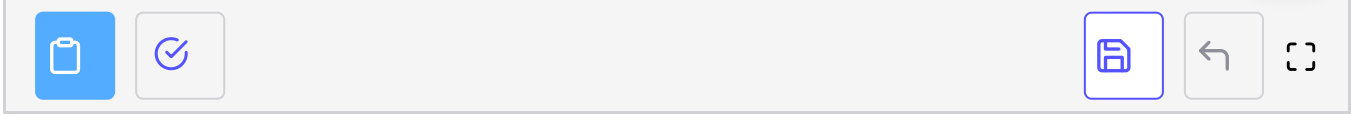
```
# CODE HERE
```



We also don't want any of the NaN values in our data. We can filter those out using the special `dropna` function.

Set `df` equal to `df.dropna` applied with no arguments.

```
# CODE HERE
```



Finally, we want to convert each categorical feature into a set of indicator features for each of its categories.

Then we can convert `df` into a NumPy matrix.

Set `df` equal to `pd.get_dummies` with `df` as the only argument.

Set `matrix` equal to `df.values`.

Quiz

1 Which of the following are methods for indexing into a DataFrame?

COMPLETED 0%

1 of 4

Introduction

An overview of industry data science and the scikit-learn API.

In the **Data Preprocessing** section, you will learn data preprocessing techniques with scikit-learn, one of the most popular frameworks used for industry data science.

A. ML engineering vs. data science

In industry, there is quite a bit of overlap between machine learning engineering and data science. Both jobs involve working with data, such as data analysis and data preprocessing.

The main task for machine learning engineers is to first analyze the data for viable trends, then create an efficient input pipeline for training a model. This process involves using libraries like NumPy and pandas for handling data, along with machine learning frameworks like TensorFlow for creating the model and input pipeline. For more information on ML engineering and the NumPy and pandas libraries, check out the previous two sections in this course.

While the NumPy and pandas libraries are also used in data science, the **Data Preprocessing** section will cover one of the core libraries that is specific to industry-level data science: scikit-learn. Data scientists tend to work on smaller datasets than machine learning engineers, and their main goal is to analyze the data and quickly extract usable results. Therefore, they focus more on traditional data inference models (found in scikit-learn), rather than deep neural networks.

The scikit-learn library includes tools for data preprocessing and data mining. It is imported in Python via the statement `import sklearn`.

Standardizing Data

Learn about data standardization and implement it with scikit-learn.

Chapter Goals:

- Learn about data standardization

A. Standard data format

Data can contain all sorts of different values. For example, Olympic 100m sprint times will range from 9.5 to 10.5 seconds, while calorie counts in large pepperoni pizzas can range from 1500 to 3000 calories. Even data measuring the exact same quantities can range in value (e.g. weight in kilograms vs. weight in pounds).

When data can take on any range of values, it makes it difficult to interpret. Therefore, data scientists will convert the data into a standard format to make it easier to understand. The standard format refers to data that has 0 mean and unit variance (i.e. standard deviation = 1), and the process of converting data into this format is called *data standardization*.

Data standardization is a relatively simple process. For each data value, x, we subtract the overall mean of the data, μ, then divide by the overall standard deviation, σ. The new value, z, represents the standardized data value. Thus, the formula for data standardization is:

$$z = \frac{x - \mu}{\sigma}$$

B. NumPy and scikit-learn

For most scikit-learn functions, the input data comes in the form of a NumPy array.

Note: The array's rows represent individual data observations, while each column represents a particular feature of the data, i.e. the same format as a

spreadsheet data table.

The scikit-learn data preprocessing module is called `sklearn.preprocessing`. One of the functions in this module, `scale`, applies data standardization to a given axis of a NumPy array.

```
# predefined pizza data
# Newline to separate print statements
print('{}\n'.format(repr(pizza_data)))

from sklearn.preprocessing import scale
# Standardizing each column of pizza_data
col_standardized = scale(pizza_data)
print('{}\n'.format(repr(col_standardized)))

# Column means (rounded to nearest thousandth)
col_means = col_standardized.mean(axis=0).round(decimals=3)
print('{}\n'.format(repr(col_means)))

# Column standard deviations
col_stds = col_standardized.std(axis=0)
print('{}\n'.format(repr(col_stds)))
```

We normally standardize the data independently across each feature of the data array. This way, we can see how many standard deviations a particular observation's feature value is from the mean.

For example, the second data observation in `pizza_data` has a net weight of 1.6 standard deviations above the mean pizza weight in the dataset.

If for some reason we need to standardize the data across rows, rather than columns, we can set the `axis` keyword argument in the `scale` function to 1. This may be the case when analyzing data within observations, rather than within a feature. An example of this would be analyzing a particular student's test scores in terms of standard deviations from that student's average test score.

Time to Code!

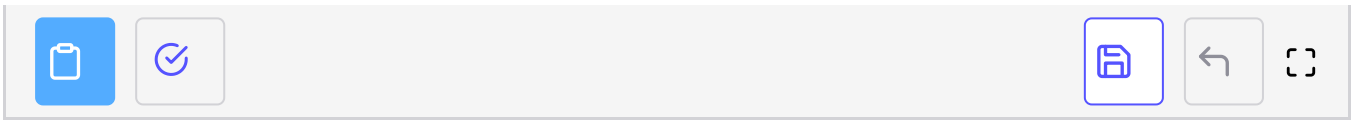
The coding exercise in this chapter is to complete a generic data standardization function, `standardize_data`.

This function will standardize the input NumPy array, `data`, by using the

`scale` function (imported in the backend).

Set `scaled_data` equal to `scale` applied with `data` as the only argument. Then return `scaled_data`.

```
def standardize_data(data):
  # CODE HERE
  pass
```



Data Range

Create a function to compress data into a specific range of values.

Chapter Goals:

- Learn how to compress data values to a specified range

A. Range scaling

Apart from standardizing data, we can also scale data by compressing it into a fixed range. One of the biggest use cases for this is compressing data into the range [0, 1]. This allows us to view the data in terms of proportions, or percentages, based on the minimum and maximum values in the data.

The formula for scaling based on a range is a two-step process. For a given data value, x, we first compute the proportion of the value with respect to the min and max of the data d_{min} and d_{max}, respectively).

$$x_{prop} = \frac{x - d_{min}}{d_{max} - d_{min}}$$

The formula above computes the proportion of the data value, x_{prop}. Note that this only works if not all the data values are the same (i.e. $d_{\text{max}} \neq d_{\text{min}}$).

We then use the proportion of the value to scale to the specified range, [r_{min}, r_{max}]. The formula below calculates the new scaled value, x_{scale}.

$$x_{scale} = x_{prop} \cdot (r_{max} - r_{min}) + r_{min}$$

B. Range compression in scikit-learn

The scikit-learn library provides a variety of *transformers*, modules that perform transformations on data. While in the previous chapter we used a single function, `scale`, to perform the data standardization, the remaining chapters will focus on using these transformer modules.

The `MinMaxScaler` transformer performs the range compression using the previous formula. Specifically, it scales each feature (column) of the data to a given range (where the default range is [0, 1]).

The code below shows how to use the `MinMaxScaler` (with the default range and a custom range).

The `MinMaxScaler` object contains a function called `fit_transform`, which allows it to take in the input data array and then output the scaled data. The function is a combination of the object's `fit` and `transform` functions, where the former takes in an input data array and the latter transforms a (possibly different) array based on the data from the input to the `fit` function.

```
# predefined data
print('{}\n'.format(repr(data)))

from sklearn.preprocessing import MinMaxScaler
default_scaler = MinMaxScaler() # the default range is [0,1]
transformed = default_scaler.fit_transform(data)
print('{}\n'.format(repr(transformed)))

custom_scaler = MinMaxScaler(feature_range=(-2, 3))
transformed = custom_scaler.fit_transform(data)
print('{}\n'.format(repr(transformed)))
```

Now lets run the `fit` and `transform` functions separately and compare them with the `fit_transform` function. `fit` takes in an input data array and `transform` transforms a (possibly different) array based on the data from the input to the fit function.

```
# predefined new_data
print('{}\n'.format(repr(new_data)))

from sklearn.preprocessing import MinMaxScaler
default_scaler = MinMaxScaler() # the default range is [0,1]
transformed = default_scaler.fit_transform(new_data)
print('{}\n'.format(repr(transformed)))

default_scaler = MinMaxScaler()  # new instance
default_scaler.fit(data)  # different data value fit
transformed = default_scaler.transform(new_data)
print('{}\n'.format(repr(transformed)))
```

The code above scales the `new_data` array to the range [0, 1], based on the (column-wise) minimum and maximum values from the `data` array in the original code example.

Time to Code!

The coding exercise in this chapter uses `MinMaxScaler` (imported in backend) to complete the `ranged_data` function.

The function will compress the input NumPy array, `data`, into the range given by `value_range`.

Set `min_max_scaler` equal to `MinMaxScaler` initialized with `value_range` for the `feature_range` keyword argument.

Set `scaled_data` equal to `min_max_scaler.fit_transform` applied with `data` as the only argument. Then return `scaled_data`.

```
def ranged_data(data, value_range):
  # CODE HERE
  pass
```

Robust Scaling

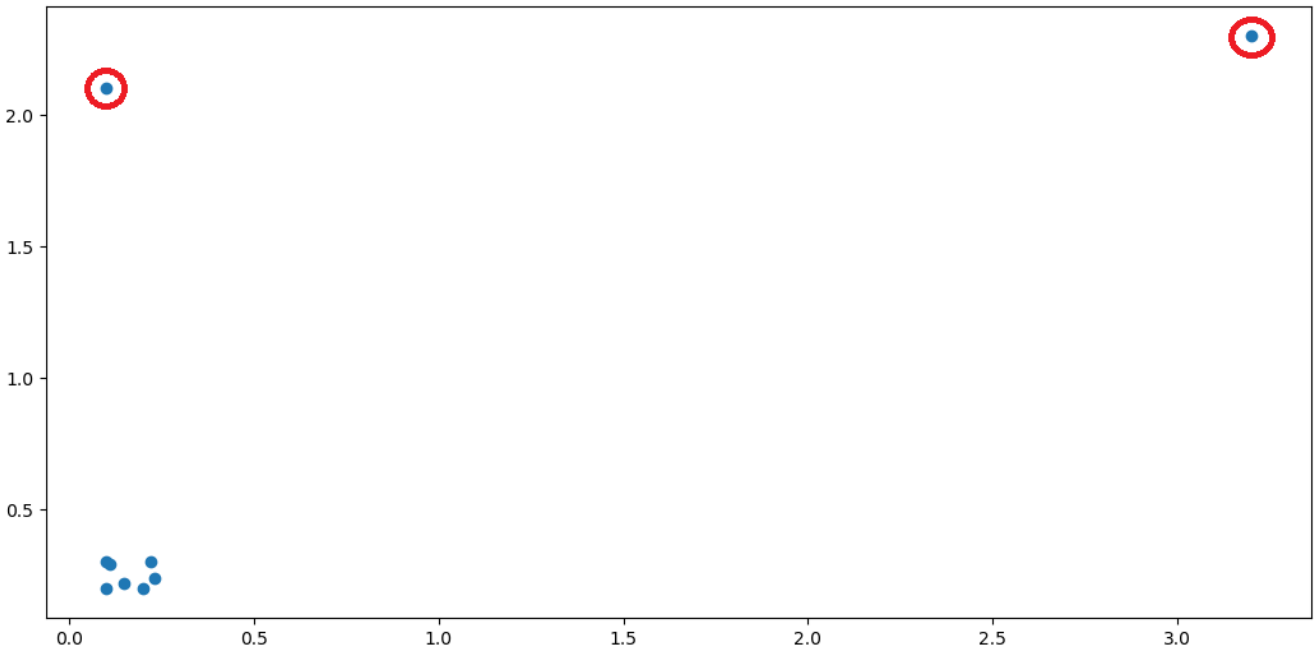
Understand how outliers can affect data and implement robust scaling.

Chapter Goals:

- Learn how to scale data without being affected by outliers

A. Data outliers

An important aspect of data that we have to deal with is *outliers*. In general terms, an outlier is a data point that is significantly further away from the other data points. For example, if we had watermelons of weights 5, 4, 6, 7, and 20 pounds, the 20 pound watermelon is an outlier.



A 2-D data plot with the outlier data points circled. Note that the outliers in this plot are exaggerated, and in real life outliers are not usually this far from the non-outlier data.

The data scaling methods from the previous two chapters are both affected by outliers. Data standardization uses each feature's mean and standard deviation, while ranged scaling uses the maximum and minimum feature values, meaning that they're both susceptible to being skewed by outlier values.

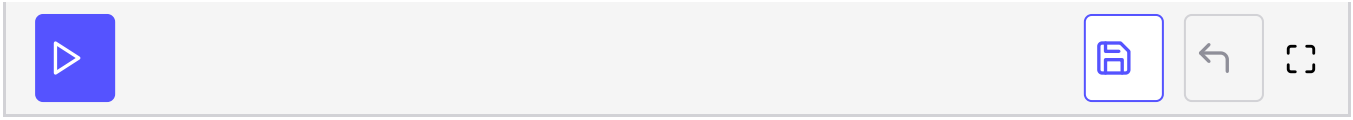
We can robustly scale the data, i.e. avoid being affected by outliers, by using use the data's median and Interquartile Range (IQR). Since the median and IQR are percentile measurements of the data (50% for median, 25% to 75% for the IQR), they are not affected by outliers. For the scaling method, we just subtract the median from each data value then scale to the IQR.

B. Robust scaling with scikit-learn

In scikit-learn, we perform robust scaling with the `RobustScaler` module. It is another transformer object, with the same `fit`, `transform`, and `fit_transform` functions described in the previous chapter.

The code below shows how to use the `RobustScaler`.

```
# predefined data
print('{}\n'.format(repr(data)))

from sklearn.preprocessing import RobustScaler
robust_scaler = RobustScaler()
transformed = robust_scaler.fit_transform(data)
print('{}\n'.format(repr(transformed)))
```



Time to Code!

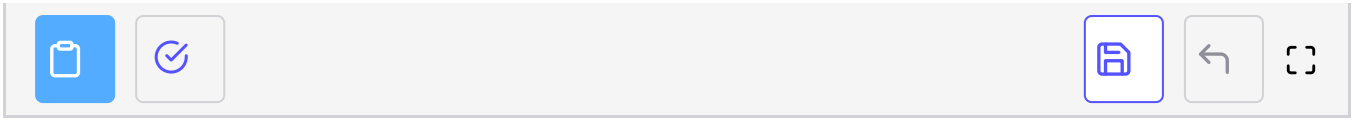
The coding exercise in this chapter uses `RobustScaler` (imported in backend) to complete the `robust_scaling` function.

The function will apply outlier-independent scaling to the input NumPy array, `data`.

Set `robust_scaler` equal to `RobustScaler` initialized without any parameters.

Set `scaled_data` equal to `robust_scaler.fit_transform` applied with `data` as the only argument. Then return `scaled_data`.

```
def robust_scaling(data):
  # CODE HERE
  pass
```



Normalizing Data

Learn about data normalization and implement a normalization function.

Chapter Goals:

- Learn how to apply L2 normalization to data

L2 normalization

So far, each of the scaling techniques we've used has been applied to the data features (i.e. columns). However, in certain cases we want to scale the individual data observations (i.e. rows). For instance, when clustering data we need to apply L2 normalization to each row, in order to calculate cosine similarity scores. The **Clustering** section will cover data clustering and cosine similarities in greater depth.

L2 normalization applied to a particular row of a data array will divide each value in that row by the row's L2 norm. In general terms, the L2 norm of a row is just the square root of the sum of squared values for the row.

$$X = [x_1, x_2, ..., x_m]$$

$$X_{L2} = \left[\frac{x_1}{\ell}, \frac{x_2}{\ell}, ..., \frac{x_m}{\ell}\right], \text{where } \ell = \sqrt{\sum_{i=1}^{m} x_i^2}$$

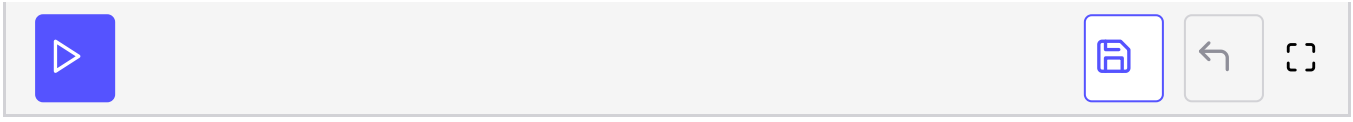
The above formula demonstrates L2 normalization applied to row X to obtain the normalized row of values, X_{L2}.

In scikit-learn, the transformer module that implements L2 normalization is the `Normalizer`.

The code below shows how to use the `Normalizer`.

```
# predefined data
```

```
print('{}\n'.format(repr(data)))

from sklearn.preprocessing import Normalizer

normalizer = Normalizer()
transformed = normalizer.fit_transform(data)
print('{}\n'.format(repr(transformed)))
```



Time to Code!

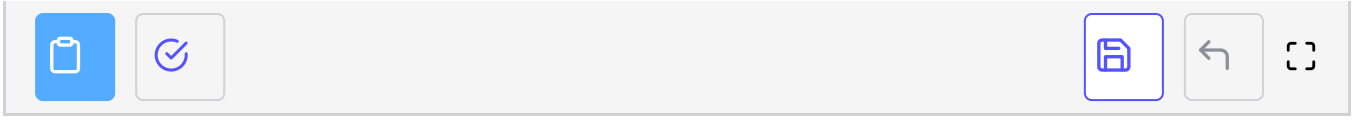
The coding exercise in this chapter uses `Normalizer` (imported in backend) to complete the `normalize_data` function.

The function will apply L2 normalization to the input NumPy array, `data`.

Set `normalizer` equal to `Normalizer` initialized without any parameters.

Set `norm_data` equal to `normalizer.fit_transform` applied with `data` as the only argument. Then return `norm_data`.

```
def normalize_data(data):
  # CODE HERE
  pass
```



Data Imputation

Learn about data imputation and the various methods to accomplish it.

Chapter Goals:

- Learn different methods for imputing data

A. Data imputation methods

In real life, we often have to deal with data that contains missing values. Sometimes, if the dataset is missing too many values, we just don't use it. However, if only a few of the values are missing, we can perform data imputation to substitute the missing data with some other value(s).

There are many different methods for data imputation. In scikit-learn, the `SimpleImputer` transformer performs four different data imputation methods.

The four methods are:

- Using the mean value
- Using the median value
- Using the most frequent value
- Filling in missing values with a constant

The code below shows how to perform data imputation using mean values from each column.

```
# predefined data
print('{}\n'.format(repr(data)))

from sklearn.impute import SimpleImputer
imp_mean = SimpleImputer()
transformed = imp_mean.fit_transform(data)
print('{}\n'.format(repr(transformed)))
```

In NumPy arrays, missing data is represented by the `np.nan` value. In the above example, we replaced each missing value with the mean of the values in its column.

The default imputation method for `SimpleImputer` is using the column means. By using the `strategy` keyword argument when initializing a `SimpleImputer` object, we can specify a different imputation method.

The code below demonstrates various initialization strategies for `SimpleImputer`.

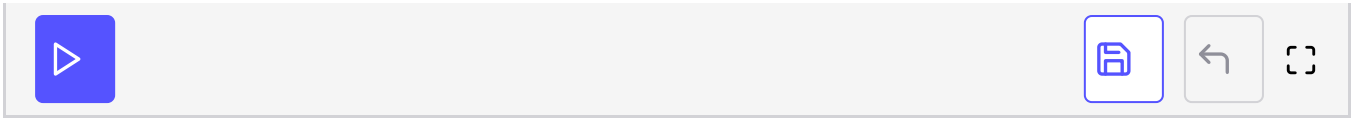
```
# predefined data
print('{}\n'.format(repr(data)))

from sklearn.impute import SimpleImputer
imp_median = SimpleImputer(strategy='median')
transformed = imp_median.fit_transform(data)
print('{}\n'.format(repr(transformed)))

imp_frequent = SimpleImputer(strategy='most_frequent')
transformed = imp_frequent.fit_transform(data)
print('{}\n'.format(repr(transformed)))
```

The `'median'` strategy fills in missing data with the median from each column, while the `'most_frequent'` strategy uses the value that appears the most for each column.

The final imputation method that `SimpleImputer` provides is to fill in missing values with a specified constant. This can be useful if there is already a suitable substitute for missing data (e.g. 0 or -1).

The code below demonstrates how to fill in missing data with a specific constant. The `fill_value` keyword argument is used when initializing the `SimpleImputer` object, to specify the constant.

```
# predefined data
print('{}\n'.format(repr(data)))

from sklearn.impute import SimpleImputer
imp_constant = SimpleImputer(strategy='constant',
                             fill_value=-1)
transformed = imp_constant.fit_transform(data)
print('{}\n'.format(repr(transformed)))
```



B. Other imputation methods

The `SimpleImputer` object only implements the four imputation methods shown in section **A**. However, data imputation is not limited to those four methods.

There are also more advanced imputation methods such as k-Nearest Neighbors (filling in missing values based on similarity scores from the kNN algorithm) and MICE (applying multiple chained imputations, assuming the missing values are randomly distributed across observations).

In most industry cases these advanced methods are not required, since the data is either perfectly cleaned or the missing values are scarce. Nevertheless, the advanced methods could be useful when dealing with open source datasets, since these tend to be more incomplete.

PCA

Learn about PCA and why it's useful for data preprocessing.

Chapter Goals:

- Learn about principal component analysis and why it's used

A. Dimensionality reduction

Most datasets contain a large number of features, some of which are redundant or not informative. For example, in a dataset of basketball statistics, the total points and points per game for a player will (most of the time) tell the same story about the player's scoring prowess.

When a dataset contains these types of correlated numeric features, we can perform principal component analysis (PCA) for dimensionality reduction (i.e. reducing the number of columns in the data array).

PCA extracts the *principal components* of the dataset, which are an uncorrelated set of latent variables that encompass most of the information from the original dataset. Using a smaller set of principal components can make it a lot easier to use the dataset in statistical or machine learning models (especially when the original dataset contains many correlated features).

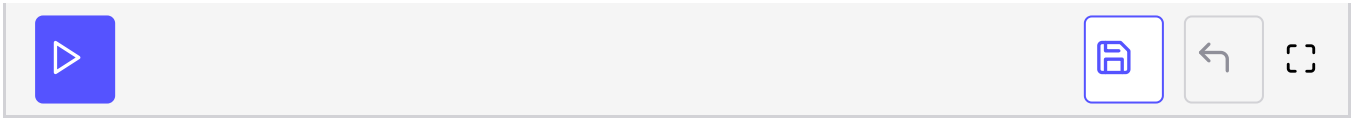
B. PCA in scikit-learn

Like every other data transformation, we can apply PCA to a dataset in scikit-learn with a transformer, in this case the `PCA` module. When initializing the `PCA` module, we can use the `n_components` keyword to specify the number of principal components. The default setting is to extract m - 1 principal components, where m is the number of features in the dataset.

The code below shows examples of applying PCA with various numbers of principal components.

```
# predefined data
```

```
print('{}\n'.format(repr(data)))

from sklearn.decomposition import PCA
pca_obj = PCA() # The value of n_component will be 4. As m is 5 and default is always m-1
pc = pca_obj.fit_transform(data).round(3)
print('{}\n'.format(repr(pc)))

pca_obj = PCA(n_components=3)
pc = pca_obj.fit_transform(data).round(3)
print('{}\n'.format(repr(pc)))

pca_obj = PCA(n_components=2)
pc = pca_obj.fit_transform(data).round(3)
print('{}\n'.format(repr(pc)))
```



In the code output above, notice that when PCA is applied with 4 principal components, the final column (last principal component) is all 0's. This means that there are actually only a maximum of three uncorrelated principal components that can be extracted.

Time to Code!

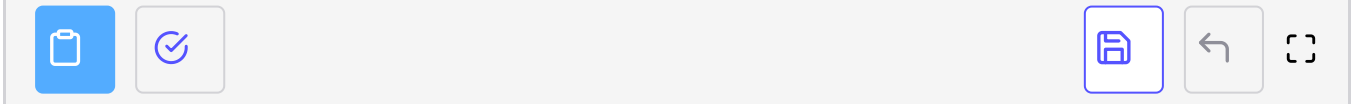
The coding exercise in this chapter uses `PCA` (imported in backend) to complete the `pca_data` function.

The function will apply principal component analysis (PCA) to the input NumPy array, `data`.

Set `pca_obj` equal to `PCA` initialized with `n_components` for the `n_components` keyword argument.

Set `component_data` equal to `pca_obj.fit_transform` applied with `data` as the only argument. Then return `component_data`.

```
def pca_data(data, n_components):
  # CODE HERE
  pass
```



Labeled Data

Separate the PCA components of a dataset by class.

Chapter Goals:

- Learn about labeled datasets
- Separate principle component data by class label

A. Class labels

A big part of data science is classifying observations in a dataset into separate categories, or *classes*. A popular use case of data classification is in separating a dataset into "good" and "bad" categories. For example, we can classify a dataset of breast tumors as either malignant or benign.

The code below separates a breast cancer dataset into malignant and benign categories. The `load_breast_cancer` function is part of the scikit-learn library, and its data comes from the Breast Cancer Wisconsin dataset.

```
from sklearn.datasets import load_breast_cancer
bc = load_breast_cancer()
print('{}\n'.format(repr(bc.data)))
print('Data shape: {}\n'.format(bc.data.shape))

# Class labels
print('{}\n'.format(repr(bc.target)))
print('Labels shape: {}\n'.format(bc.target.shape))

# Label names
print('{}\n'.format(list(bc.target_names)))

malignant = bc.data[bc.target == 0]
print('Malignant shape: {}\n'.format(malignant.shape))

benign = bc.data[bc.target == 1]
print('Benign shape: {}\n'.format(benign.shape))
```

In the example above, the `bc.data` array contains all the dataset values, while

the `bc.target` array contains the class ID labels for each row in `bc.data`. A

class ID of 0 corresponds to a malignant tumor, while a class ID of 1 corresponds to a benign tumor.

Using the `bc.target` class IDs, we separated the dataset into malignant and benign data arrays. In other words, the `malignant` array contains the rows of `bc.data` corresponding to the indexes in `bc.target` containing 0, while the `benign` array contains the rows of `bc.data` corresponding to the indexes in `bc.target` containing 1. There are 212 malignant data observations and 357 benign observations.

Time to Code!

The coding exercise in this chapter involves completing the `separate_components` function, which will separate principal component data by class.

To do this, we first need to complete a helper function, `get_label_info`, which returns the label name and data for an input class label.

The `component_data` input represents the principal component data.

The `labels` input is a 1-D array containing the class label IDs corresponding to each row of `component_data`. We can use it to separate the principle components by class.

The `class_label` input represents a particular class label ID.

The `label_names` input represents all the string names for the class labels.

Set `label_name` equal to the string at index `class_label` of `label_names`.

Set `label_data` equal to the rows of `component_data` corresponding to the indexes where `labels` equals `class_label`. Then return the tuple `(label_name, label_data)`.

```
def get_label_info(component_data, labels,
                   class_label, label_names):
  # CODE HERE
  pass
```

Now, inside the main `separate_data` function, we'll iterate through each label in the `label_names` list.

Set `separated_data` equal to an empty list.

Create a `for` loop that iterates variable `class_label` through `range(len(label_names))`.

Inside the `for` loop, we can use our helper function to obtain the separated data for each class.

Inside the `for` loop, call `get_label_info` with inputs `component_data`, `labels`, `class_label`, and `label_names`. Append the function's output to `separated_data`.

After finalizing the list of principle components separated by class, we return it.

Outside the `for` loop, return `separated_data`.

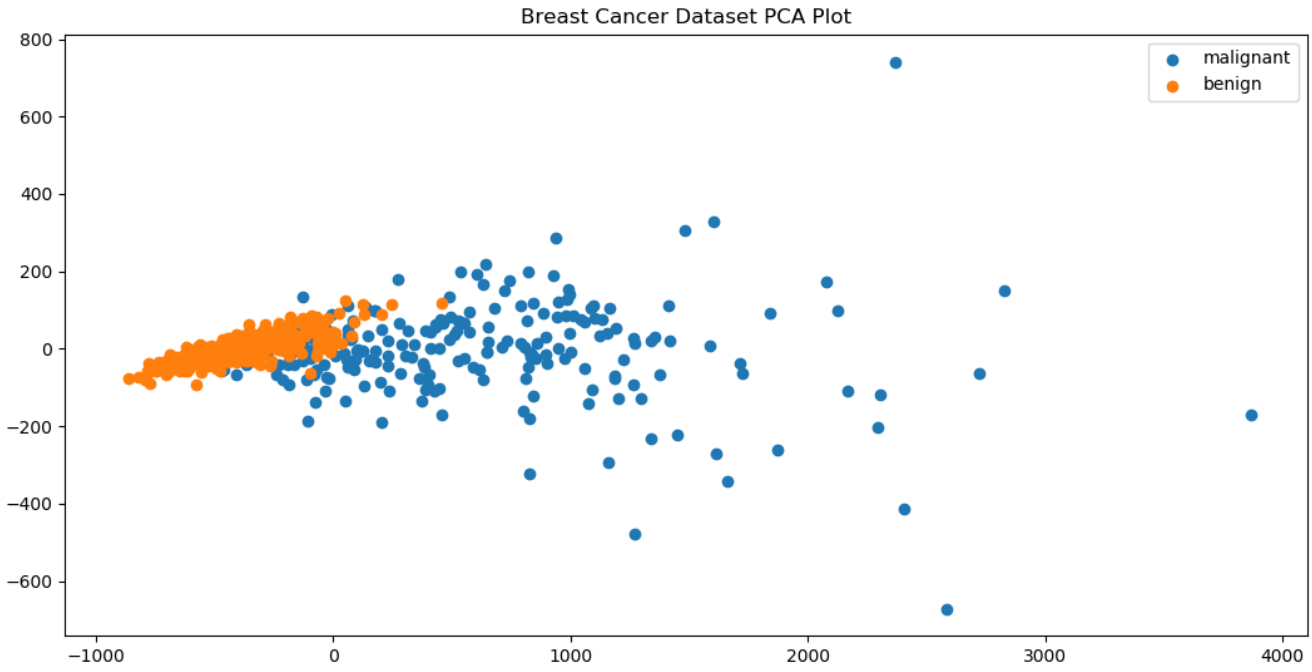
```
def separate_data(component_data, labels,
                  label_names):
  # CODE HERE
  pass
```

The `separate_data` function is incredibly useful for visualizing the components. We can use matplotlib to create nice plots of the separated data (shown in the code below).

```
from sklearn.datasets import load_breast_cancer
from sklearn.decomposition import PCA
bc = load_breast_cancer()
pca_obj = PCA(n_components=2)
component_data = pca_obj.fit_transform(bc.data)
labels = bc.target
label_names = bc.target_names
# Using the completed separate_data function
separated_data = separate_data(component_data,
                               labels, label_names)

# Plotting the data
```

```
import matplotlib.pyplot as plt
for label_name, label_data in separated_data:
  col1 = label_data[:, 0]  # 1st column (1st pr. comp.)
  col2 = label_data[:, 1]  # 2nd column (2nd pr. comp.)
  plt.scatter(col1, col2, label=label_name) # scatterplot
plt.legend()  # adds legend to plot
plt.title('Breast Cancer Dataset PCA Plot')
plt.show()
```

The result of the above code is this plot:



Quiz

1 What is the main purpose of standardizing data?

COMPLETED 0%

1 of 4

Introduction

An overview of the main models used in scikit-learn.

In the **Data Modeling** section, you will be creating a variety of models for linear regression and classifying data. You will also learn how to perform hyperparameter tuning and model evaluation through cross-validation.

A. Creating models for data

The main job of a data scientist is analyzing data and creating models for obtaining results from the data. Oftentimes, data scientists will use simple statistical models for their data, rather than machine learning models like neural networks. This is because data scientists tend to work with smaller datasets than machine learning engineers, so they can quickly extract good results using statistical models.

The scikit-learn library provides many statistical models for linear regression. It also provides a few good models for classifying data, which will be introduced in later chapters.

When creating these models, data scientists need to figure out the optimal hyperparameters to use. Hyperparameters are values that we set when creating a model, e.g. certain constant coefficients used in the model's calculations. We'll talk more about hyperparameter tuning, the process of finding the optimal hyperparameter settings, in later chapters.

Linear Regression

Learn about basic linear regression and how it's used.

Chapter Goals:

- Create a basic linear regression model based on input data and labels

A. What is linear regression?

One of the main objectives in both machine learning and data science is finding an equation or distribution that best fits a given dataset. This is known as data modeling, where we create a model that uses the dataset's features as independent variables to predict output values for some dependent variable (with minimal error). However, it is incredibly difficult to find an optimal model for most datasets, given the amount of noise (i.e. random errors/fluctuations) in real world data.

Since finding an optimal model for a dataset is difficult, we instead try to find a good approximating distribution. In many cases, a linear model (a linear combination of the dataset's features) can approximate the data well. The term *linear regression* refers to using a linear model to represent the relationship between a set of independent variables and a dependent variable.

$$y = ax_1 + bx_2 + cx_3 + d$$

The above formula is example linear model which produces output y (dependent variable) based on the linear combination of independent variables *x_1*, *x_2*, *x_3*. The coefficients *a*, *b*, *c* and intercept *d* determine the model's fit.

B. Basic linear regression

The simplest form of linear regression is called least squares regression. This strategy produces a regression model, which is a linear combination of the

independent variables, that minimizes the sum of squared residuals between the model's predictions and actual values for the dependent variable.

In scikit-learn, the least squares regression model is implemented with the `LinearRegression` object, which is a part of the `linear_model` module in `sklearn`. The object contains a `fit` function, which takes in an input dataset of features (independent variables) and an array of labels (dependent variables) for each data observation (rows of the dataset).

The code below demonstrates how to fit a `LinearRegression` model to a dataset of 5 different pizzas (`pizza_data`) and corresponding pizza prices. The first column of `pizza_data` represents the number of calories and the second column represents net weight (in grams).

```
# predefined pizza data and prices
print('{}\n'.format(repr(pizza_data)))
print('{}\n'.format(repr(pizza_prices)))

from sklearn import linear_model
reg = linear_model.LinearRegression()
reg.fit(pizza_data, pizza_prices)
```

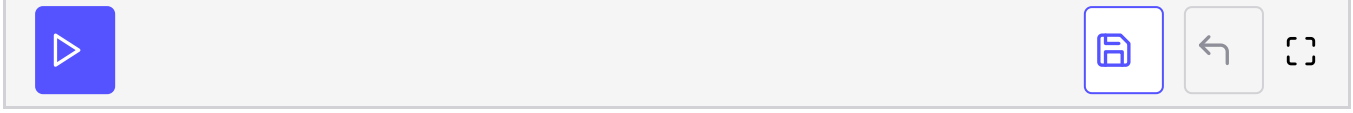
After calling the `fit` function, the model is ready to use. The `predict` function allows us to make predictions on new data.

We can also get the specific coefficients and intercept for the linear combination using the `coef_` and `intercept_` properties, respectively.

Finally, we can retrieve the coefficient of determination (or R^2 value) using the `score` function applied to the dataset and labels. The R^2 value tells us how close of a fit the linear model is to the data, or in other words, how good of a fit the model is for the data.

```
# new pizza data
new_pizzas = np.array([[2000,  820],
                       [2200,  830]])

price_predicts = reg.predict(new_pizzas)
print('{}\n'.format(repr(price_predicts)))

print('Coefficients: {}\n'.format(repr(reg.coef_)))
print('Intercept: {}\n'.format(reg.intercept_))
```

```
# Using previously defined pizza_data, pizza_prices
r2 = reg.score(pizza_data, pizza_prices)
print('R2: {}\n'.format(r2))
```



The traditional R^2 value is a real number between 0 and 1. In scikit-learn it ranges from -∞ to 1, where lower values denote a poorer model fit to the data. The closer the value is to 1, the better the model's fit on the data. In the example above, we see that the model is a near perfect fit for the pizza data.

Time to Code!

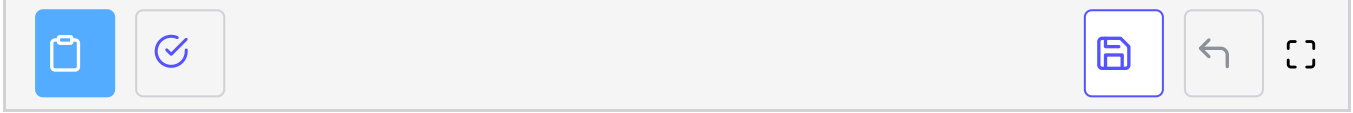
The coding exercise in this chapter uses the `LinearRegression` object of the `linear_model` module (imported in backend) to complete the `linear_reg` function.

The function will fit a basic least squares regression model to the input data and labels.

Set `reg` equal to `linear_model.LinearRegression` initialized with no input arguments.

Call `reg.fit` with `data` and `labels` as the two input arguments. Then return `reg`.

```
def linear_reg(data, labels):
  # CODE HERE
  pass
```



Ridge Regression

Understand the need for regularization in linear regression.

Chapter Goals:

- Learn about regularization in linear regression
- Learn about hyperparameter tuning using cross-validation
- Implement a cross-validated ridge regression model in scikit-learn

While ordinary least squares regression is a good way to fit a linear model onto a dataset, it relies on the fact that the dataset's features are each independent, i.e. uncorrelated. When many of the dataset features are linearly correlated, e.g. if a dataset has multiple features depicting the same price in different currencies, it makes the least squares regression model highly sensitive to noise in the data.

Because real life data tends to have noise, and will often have some linearly correlated features in the dataset, we combat this by performing *regularization*. For ordinary least squares regression, the goal is to find the weights (coefficients) for the linear model that minimize the sum of squared residuals:

$$\sum_{i=1}^{n}(\mathbf{x}_i \cdot w - y_i)^2$$

where each x_i represents a data observation and y_i is the corresponding label.

For regularization, the goal is to not only minimize the sum of squared residuals, but to do this with coefficients as small as possible. The smaller the coefficients, the less susceptible they are to random noise in the data. The most commonly used form of regularization is ridge regularization.

With ridge regularization, the goal is now to find the weights that minimize the following quantity:

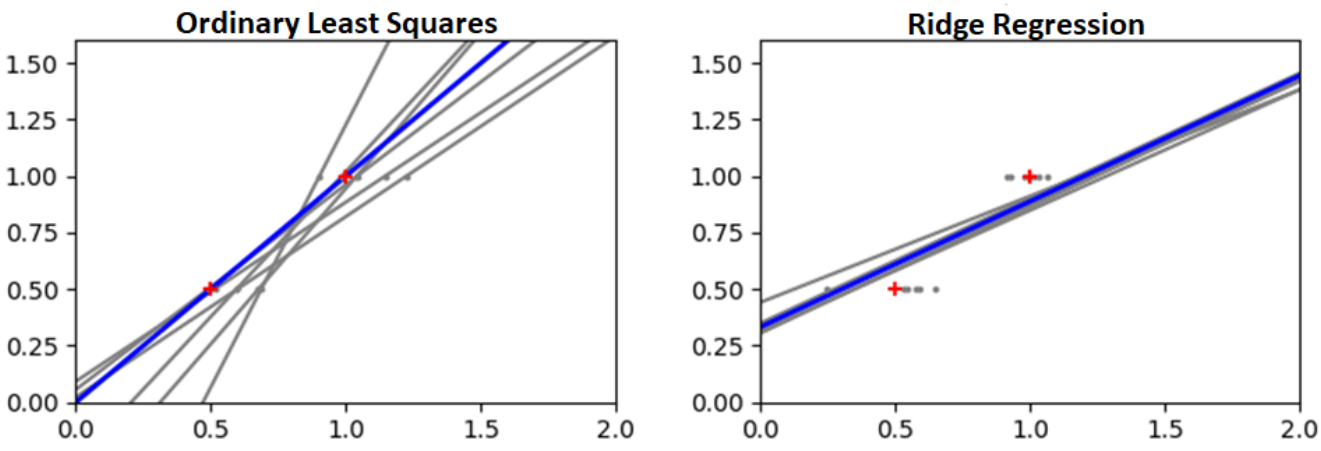
the following quantity:

$$\alpha||w||_2^2 + \sum_{i=1}^{n}(\mathbf{x}_i \cdot w - y_i)^2$$

where α is a non-negative real number hyperparameter and $||w||_2$ represents the L2 norm of the weights. The additional

$$\alpha||w||_2^2$$

is referred to as the *penalty term*, since it penalizes larger weight values. Larger quantities of α will put greater emphasis on the penalty term, forcing the model to have even smaller weight values.



The plot above shows an example of ordinary least squares regression models vs. ridge regression models. The two red crosses mark the points (0.5, 0.5) and (1, 1), and the blue lines are the regression lines for those two points. Each of the grey lines are the regression lines for the original points with added noise (which is signified by the grey points).

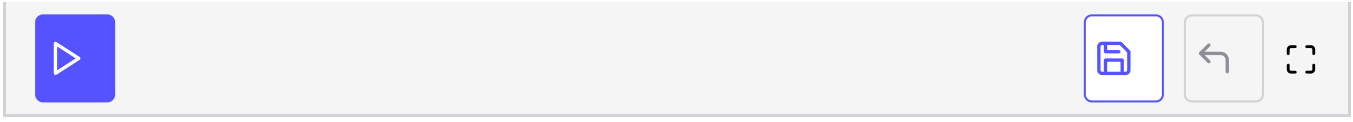
The ordinary least squares regression is much more susceptible to being influenced by the added noise, as there is a much larger degree of variance in the grey regression lines compared to the ridge regression.

B. Choosing the best alpha

In scikit-learn, we implement ridge regression in essentially the same way we implement ordinary least squares regression. We use the `Ridge` object (part of the `linear_model` module) to implement ridge regression.

The code below fits a `Ridge` object on the pizza dataset from the previous chapter.

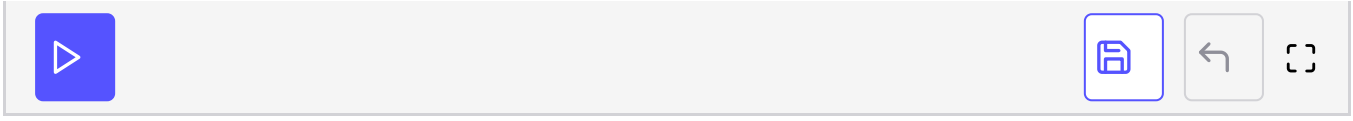
```
from sklearn import linear_model
reg = linear_model.Ridge(alpha=0.1)
reg.fit(pizza_data, pizza_prices)
print('Coefficients: {}\n'.format(repr(reg.coef_)))
print('Intercept: {}\n'.format(reg.intercept_))
r2 = reg.score(pizza_data, pizza_prices)
print('R2: {}\n'.format(r2))
```



We can specify the value of the α hyperparameter when initializing the `Ridge` object (the default is 1.0). However, rather than manually choosing a value, we can use *cross-validation* to help us choose the optimal α from a list of values.

We'll discuss the specifics of cross-validation in the chapters Cross-Validation, Applying CV to Decision Trees, and Exhaustive Tuning, but for now just know that we can implement a cross-validated ridge regression using the `RidgeCV` object.

```
from sklearn import linear_model
alphas = [0.1, 0.2, 0.3]
reg = linear_model.RidgeCV(alphas=alphas)
reg.fit(pizza_data, pizza_prices)
print('Coefficients: {}\n'.format(repr(reg.coef_)))
print('Intercept: {}\n'.format(reg.intercept_))
print('Chosen alpha: {}\n'.format(reg.alpha_))
```



Time to Code!

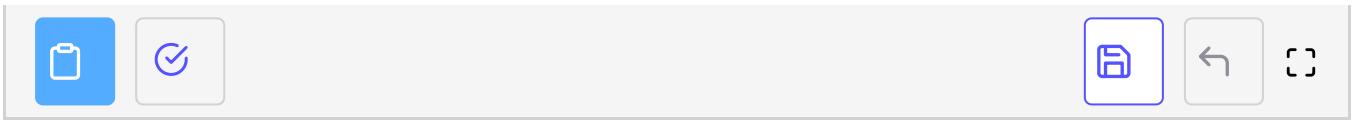
The coding exercise in this chapter uses the `RidgeCV` object of the `linear_model` module (imported in backend) to complete the `cv_ridge_reg` function.

The function will fit a ridge regression model to the input data and labels. The model is cross-validated to choose the best α value from the input list `alphas`.

Set `reg` equal to `linear_model.RidgeCV` initialized with the input list, `alphas`, for the `alphas` keyword argument.

Call `reg.fit` with `data` and `labels` as the two input arguments. Then return `reg`.

```
def cv_ridge_reg(data, labels, alphas):
  # CODE HERE
  pass
```



LASSO Regression

Apply regularization with LASSO regression.

Chapter Goals:

- Learn about sparse linear regression via LASSO

A. Sparse regularization

While ridge regularization uses an L2 norm penalty term, another regularization method called LASSO uses an L1 norm for the weights penalty term. Specifically, LASSO regularization will find the optimal weights to minimize the following quantity:

$$\alpha||w||_1 + \sum_{i=1}^{n}(\mathbf{x}_i \cdot w - y_i)^2$$

where $||w||_1$ represents the L1 norm of the weights.

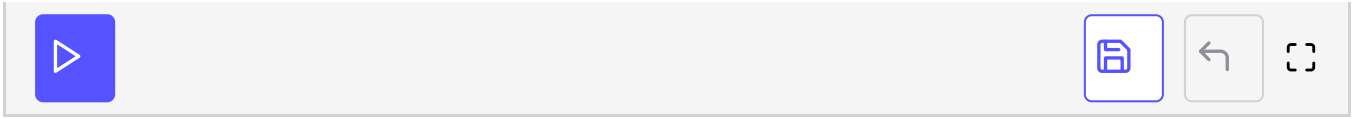
LASSO regularization tends to prefer linear models with fewer parameter values. This means that it will likely zero-out some of the weight coefficients. This reduces the number of features that the model is actually dependent on (since some of the coefficients will now be 0), which can be beneficial when some features are completely irrelevant or duplicates of other features.

In scikit-learn, we implement LASSO using the `Lasso` object, which is part of the `linear_model` module. Like the `Ridge` object, it takes in the model's α value with the `alpha` keyword argument (default is 1.0).

The code below demonstrates how to use the `Lasso` object on a dataset with 150 observations and 4 features.

```
# predefined dataset
print('Data shape: {}\n'.format(data.shape))
print('Labels shape: {}\n'.format(labels.shape))
```

```
from sklearn import linear_model
reg = linear_model.Lasso(alpha=0.1)
reg.fit(data, labels)
print('Coefficients: {}\n'.format(repr(reg.coef_)))
print('Intercept: {}\n'.format(reg.intercept_))
print('R2: {}\n'.format(reg.score(data, labels)))
```



In the example above, note that a majority of the weights are 0, due to the LASSO sparse weight preference.

There is also a cross-validated version in the form of the `LassoCV` object, which works in essentially the same way as the `RidgeCV` object.

Time to Code!

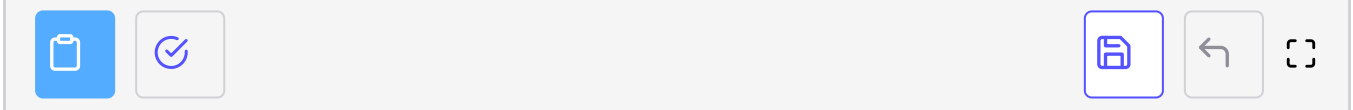
The coding exercise in this chapter uses the `Lasso` object of the `linear_model` module (imported in backend) to complete the `lasso_reg` function.

The function will fit a LASSO regression model to the input data and labels. The α hyperparameter for the model is provided to the function via the `alpha` input argument.

Set `reg` equal to `linear_model.Lasso` initialized with `alpha` for the `alpha` keyword argument.

Call `reg.fit` with `data` and `labels` as the two input arguments. Then return `reg`.

```
def lasso_reg(data, labels, alpha):
  # CODE HERE
  pass
```



Bayesian Regression

Learn about Bayesian regression techniques.

Chapter Goals:

- Learn about Bayesian regression techniques

A. Bayesian techniques

So far, we've discussed hyperparameter optimization through cross-validation. Another way to optimize the hyperparameters of a regularized regression model is with Bayesian techniques.

In Bayesian statistics, the main idea is to make certain assumptions about the probability distributions of a model's parameters *before* being fitted on data. These initial distribution assumptions are called *priors* for the model's parameters.

In a Bayesian ridge regression model, there are two hyperparameters to optimize: α and λ. The α hyperparameter serves the same exact purpose as it does for regular ridge regression; namely, it acts as a scaling factor for the penalty term.

The λ hyperparameter acts as the precision of the model's weights. Basically, the smaller the λ value, the greater the variance between the individual weight values.

B. Hyperparameter priors

Both the α and λ hyperparameters have gamma distribution priors, meaning we assume both values come from a gamma probability distribution.

There's no need to know the specifics of a gamma distribution, other than the fact that it's a probability distribution defined by a shape parameter and scale parameter.

Specifically, the α hyperparameter has prior:

Specifically, the α hyperparameter has prior:

$$\Gamma(\alpha_1, \alpha_2)$$

and the λ hyperparameter has prior:

$$\Gamma(\lambda_1, \lambda_2)$$

where Γ(k, θ) represents a gamma distribution with shape parameter k and scale parameter θ.

C. Tuning the model

When finding the optimal weight settings of a Bayesian ridge regression model for an input dataset, we also concurrently optimize the α and λ hyperparameters based on their prior distributions and the input data.

This can all be done with the `BayesianRidge` object (part of the `linear_model` module). Like all the previous regression objects, this one can be initialized with no required arguments.

```
# predefined dataset from previous chapter
print('Data shape: {}\n'.format(data.shape))
print('Labels shape: {}\n'.format(labels.shape))

from sklearn import linear_model
reg = linear_model.BayesianRidge()
reg.fit(data, labels)
print('Coefficients: {}\n'.format(repr(reg.coef_)))
print('Intercept: {}\n'.format(reg.intercept_))
print('R2: {}\n'.format(reg.score(data, labels)))
print('Alpha: {}\n'.format(reg.alpha_))
print('Lambda: {}\n'.format(reg.lambda_))
```

We can manually specify the α_1 and α_2 gamma parameters for α with the `alpha_1` and `alpha_2` keyword arguments when initializing `BayesianRidge`. Similarly, we can manually set λ_1 and λ_2 with the `lambda_1` and `lambda_2` keyword arguments. The default value for each of the four gamma parameters is 10^{-6}.

Time to Code!

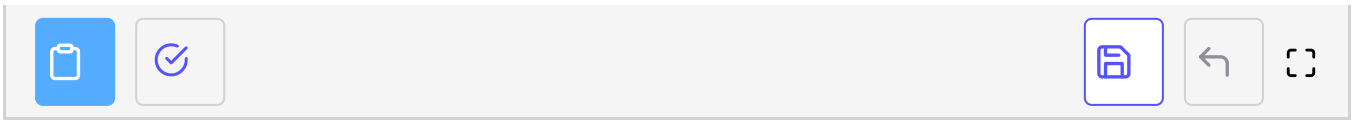
The coding exercise in this chapter uses the `BayesianRidge` object of the `linear_model` module (imported in backend) to complete the `bayes_ridge` function.

The function will fit a Bayesian ridge regression model to the input data and labels.

Set `reg` equal to `linear_model.BayesianRidge`, initialized with no input arguments.

Call `reg.fit` with `data` and `labels` as the two input arguments. Then return `reg`.

```
def bayes_ridge(data, labels):
  # CODE HERE
  pass
```



Logistic Regression

Implement logistic regression for classification tasks.

Chapter Goals:

- Learn about logistic regression for linearly separable datasets

A. Classification

Thus far we've learned about several linear regression models and implemented them with scikit-learn. The logistic regression model, despite its name, is actually a linear model for *classification*. It is called logistic regression because it performs regression on logits, which then allows us to classify the data based on model probability predictions.

For a more detailed explanation of logistic regression, check out the **Intro to Deep Learning** section of this course, which implements logistic regression via a single layer perceptron model in TensorFlow.

We implement logistic regression with the `LogisticRegression` object (part of the `linear_model` module). The default setting for `LogisticRegression` is *binary classification*, i.e. classifying data observations that are labeled with either a 0 or 1.

```
# predefined dataset
print('Data shape: {}\n'.format(data.shape))
# Binary labels
print('Labels:\n{}\n'.format(repr(labels)))

from sklearn import linear_model
reg = linear_model.LogisticRegression()
reg.fit(data, labels)

new_data = np.array([
  [  0.3,  0.5, -1.2,  1.4],
  [ -1.3,  1.8, -0.6, -8.2]])
print('Prediction classes: {}\n'.format(
  repr(reg.predict(new_data))))
```

The code above created a logistic regression model from a labeled dataset. The model predicts 1 and 0, respectively, as the labels for the observations in `new_data`.

For *multiclass classification*, i.e. when there are more than two labels, we initialize the `LogisticRegression` object with the `multi_class` keyword argument. The default value is `'ovr'`, which signifies a One-Vs-Rest strategy. In multiclass classification, we want to use the `'multinomial'` strategy.

The code below demonstrates multiclass classification. Note that to use the `'multinomial'` strategy, we need to choose a proper solver (see below for details on solvers). In this case, we choose `'lbfgs'`.

```
# predefined dataset
print('Data shape: {}\n'.format(data.shape))
# Multiclass labels
print('Labels:\n{}\n'.format(repr(labels)))

from sklearn import linear_model
reg = linear_model.LogisticRegression(
  solver='lbfgs',
  multi_class='multinomial')
reg.fit(data, labels)

new_data = np.array([
  [ 1.8, -0.5, 6.2, 1.4],
  [ 3.3,  0.8, 0.1, 2.5]])
print('Prediction classes: {}\n'.format(
  repr(reg.predict(new_data))))
```

B. Solvers

The `LogisticRegression` object uses a *solver* to obtain the optimal weight settings based on the input data and labels. The five solvers and their various properties are shown in the table below (which comes from the scikit-learn official website):

	Solvers				
Penalties	**'liblinear'**	**'lbfgs'**	**'newton-cg'**	**'sag'**	**'saga'**
Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

Table of the five solvers and their properties.

By default, the logistic regression is regularized through the L2 norm of weights. We can manually specify whether to use the L1 or L2 norm with the `penalty` keyword argument, by setting it as either `'l1'` or `'l2'`.

We can choose a particular solver using the `solver` keyword argument. The default solver is currently `'liblinear'` (although it will change to `'lbfgs'` in future version). For the `'newton-cg'`, `'sag'`, and `'lbfgs'` solvers, we can also set the maximum number of iterations the solver takes until the model's weights converge using the `max_iter` keyword argument. Since the default `max_iter` value is 100, we may want to let the solver run for a higher number of iterations in certain applications.

The code below demonstrates usage of the `solver` and `max_iter` keyword arguments.

```
from sklearn import linear_model
reg = linear_model.LogisticRegression(
  solver='lbfgs', max_iter=1000)
```

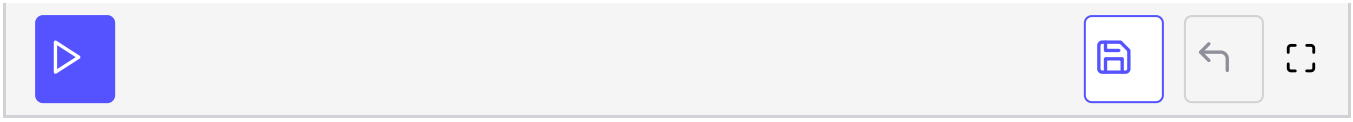
C. Cross-validated model

Like the ridge and LASSO regression models, the logistic regression model comes with a cross-validated version in scikit-learn. The cross-validated logistic regression object, `LogisticRegressionCV`, is initialized and used in the same way as the regular `LogisticRegression` object.

The code below demonstrates usage of the `LogisticRegressionCV` object.

```
from sklearn import linear_model
```

```
reg = linear_model.LogisticRegressionCV(
  solver='multinomial', max_iter=1000)
```



Time to Code!

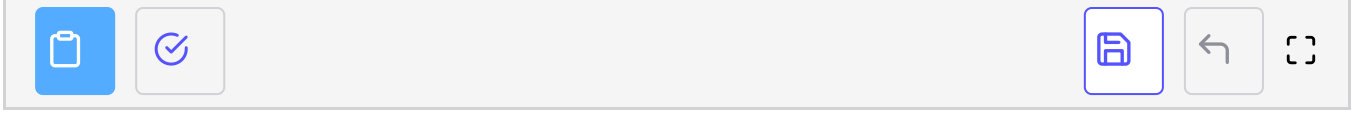
The coding exercise in this chapter uses the `LogisticRegression` object of the `linear_model` module (imported in backend) for multiclass classification.

The function `multiclass_lr` will fit a logistic regression model to a dataset with multiclass labels.

Set `reg` equal to `linear_model.LogisticRegression` with the `solver` and `max_iter` keyword arguments set as `'lbfgs'` and `max_iter`, respectively. Also have the `multi_class` keyword argument set to `'multinomial'`.

Call `reg.fit` with `data` and `labels` as the two input arguments. Then return `reg`.

```
def multiclass_lr(data, labels, max_iter):
  # CODE HERE
  pass
```



Decision Trees

Learn about decision trees and how they're used.

Chapter Goals:

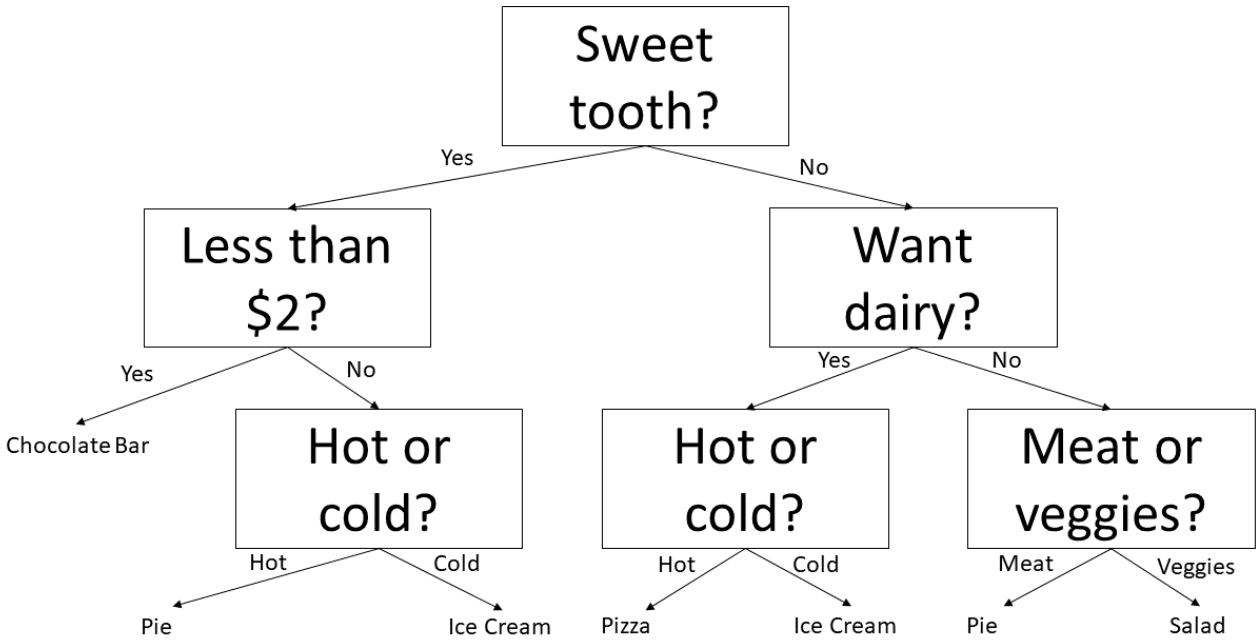
- Learn about decision trees and how they are constructed
- Learn how decision trees are used for classification and regression

A. Making decisions

Each model we've looked at so far is based on creating an optimal linear combination of dataset features for either regression or classification. However, another popular model in data science for both classification and regression is the decision tree. It is a binary tree where each node of the tree decides on a particular feature of the dataset, and we descend to the node's left or right child depending on the feature's value.

If a feature is boolean, we go left or right from the node depending on if the feature value is true or false. If a feature is numeric, we can decide to go left or right based on a decision boundary for the feature (e.g. go left if the feature value is less than 1, otherwise go right).

The leaves of the decision tree determine the class label to predict (in classification) or the real number value to predict (in regression).



A decision tree for deciding what to eat. This is an example of multiclass classification.

In scikit-learn, we implement classification decision trees with the `DecisionTreeClassifier` object, and regression trees with the `DecisionTreeRegressor` object. Both objects are part of the `tree` module in scikit-learn.

The code below demonstrates how to create decision trees for classification and regression. Each decision tree uses the `fit` function for fitting on data and labels.

```
from sklearn import tree
clf_tree1 = tree.DecisionTreeClassifier()
reg_tree1 = tree.DecisionTreeRegressor()
clf_tree2 = tree.DecisionTreeClassifier(
  max_depth=8)  # max depth of 8
reg_tree2 = tree.DecisionTreeRegressor(
  max_depth=5)  # max depth of 5

# predefined dataset
print('Data shape: {}\n'.format(data.shape))
# Binary labels
print('Labels:\n{}\n'.format(repr(labels)))
clf_tree1.fit(data, labels)
```

The `max_depth` keyword argument lets us manually set the maximum number of layers allowed in the decision tree (i.e. the tree's maximum depth). The default value is `None`, meaning that the decision tree will continue to be constructed until no nodes can have anymore children. Since large decision trees are prone to overfit data, it can be beneficial to manually set a maximum depth for the tree.

B. Choosing features

Since a decision tree makes decisions based on feature values, the question now becomes how we choose the features to decide on at each node. In general terms, we want to choose the feature value that "best" splits the remaining dataset at each node.

How we define "best" depends on the decision tree algorithm that's used. Since scikit-learn uses the CART algorithm, we use Gini Impurity, MSE (mean squared error), and MAE (mean absolute error) to decide on the best feature at each node.

Specifically, for classification trees we choose the feature at each node that minimizes the remaining dataset observations' Gini Impurity. For regression trees we choose the feature at each node that minimizes the remaining dataset observations' MSE or MAE, depending on which you choose to use (the default for `DecisionTreeRegressor` is MSE).

Time to Code!

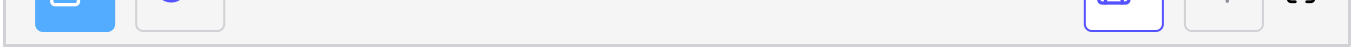
The coding exercises in this chapter use the `DecisionTreeRegressor` object of the `tree` module for regression modeling.

You will create a decision tree with max depth equal to 5, then fit the tree on (predefined) `data` and `labels`.

Set `d_tree` equal to `tree.DecisionTreeRegressor` initialized with `5` for the `max_depth` keyword argument.

Call `d_tree.fit` on `data` and `labels`.

```
# CODE HERE
```



Training and Testing

Separate a dataset into training and testing sets.

Chapter Goals:

- Learn about splitting a dataset into training and testing sets

A. Training and testing sets

We've discussed in depth how to fit a model on data and labels. However, once we fit the model, how do we evaluate it? It is a bad idea to evaluate a model solely on the same dataset it was fitted on, because the model's parameters are already tuned for that dataset. Instead, we need to split the original dataset into two datasets: one for *training* and one for *testing*.

The training set is used for fitting the model on data (i.e. training the model), while the testing set is used for evaluating the model. Therefore, the training set is much larger than the testing set. Exactly how much larger depends on the application and requirements.

Increasing the size of the training set will give more data for the model to be fitted on, which can increase the model's performance. However, because this decreases the size of the testing set, there's a higher chance that the testing set may not be representative of the original dataset (which can lead to inaccurate evaluation).

In general, the testing set is around 10-30% of the original dataset, while the training set makes up the remaining 70-90%.

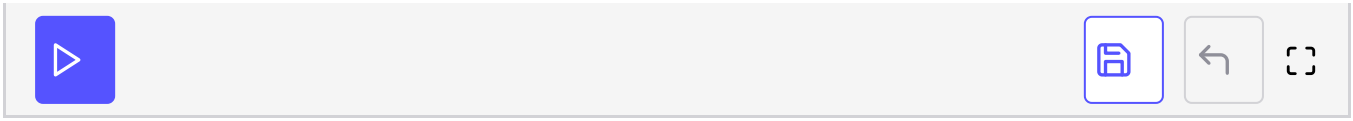
B. Splitting the dataset

The scikit-learn library provides a nice utility function, called `train_test_split` (which is part of the `model_selection` module) that handles the dataset splitting for us.

The code below demonstrates how to split a dataset into training and testing

sets.

```
data = np.array([
  [10.2 ,  0.5 ],
  [ 8.7 ,  0.9 ],
  [ 9.3 ,  0.8 ],
  [10.1 ,  0.4 ],
  [ 9.5 ,  0.77],
  [ 9.1 ,  0.68],
  [ 7.7 ,  0.9 ],
  [ 8.3 ,  0.8 ]])
labels = np.array(
  [1.4, 1.2, 1.6, 1.5, 1.6, 1.3, 1.1, 1.2])

from sklearn.model_selection import train_test_split
split_dataset = train_test_split(data, labels)
train_data = split_dataset[0]
test_data = split_dataset[1]
train_labels = split_dataset[2]
test_labels = split_dataset[3]

print('{}\n'.format(repr(train_data)))
print('{}\n'.format(repr(train_labels)))
print('{}\n'.format(repr(test_data)))
print('{}\n'.format(repr(test_labels)))
```

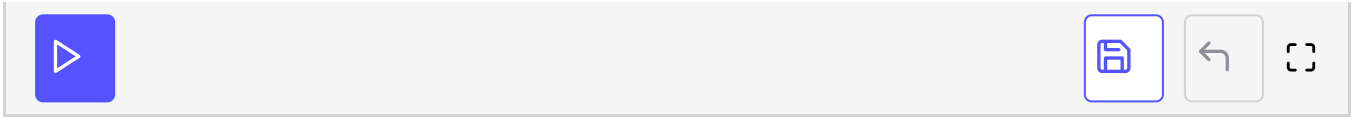


Note that the `train_test_split` function randomly shuffles the dataset and corresponding labels prior to splitting. This is good practice to remove any systematic orderings in the dataset, which could potentially impact the model into training on the orderings rather than the actual data.

The default size of the testing set is 25% of the original dataset. We can use the `test_size` keyword argument to manually specify the proportion of the original dataset that will go into the testing set.

```
data = np.array([
  [10.2 ,  0.5 ],
  [ 8.7 ,  0.9 ],
  [ 9.3 ,  0.8 ],
  [10.1 ,  0.4 ],
  [ 9.5 ,  0.77],
  [ 9.1 ,  0.68],
  [ 7.7 ,  0.9 ],
  [ 8.3 ,  0.8 ]])
labels = np.array(
  [1.4, 1.2, 1.6, 1.5, 1.6, 1.3, 1.1, 1.2])

from sklearn.model_selection import train_test_split
split_dataset = train_test_split(data, labels,
                                 test_size=0.375)
```

```
test_size=0.375)
train_data = split_dataset[0]
test_data = split_dataset[1]
train_labels = split_dataset[2]
test_labels = split_dataset[3]

print('{}\n'.format(repr(train_data)))
print('{}\n'.format(repr(train_labels)))
print('{}\n'.format(repr(test_data)))
print('{}\n'.format(repr(test_labels)))
```



In later chapters, we'll discuss how we use the testing set to evaluate a trained (fitted) model.

Time to Code!

The coding exercise for this chapter will be to finish a utility function called `dataset_splitter`, which will be used in future chapters.

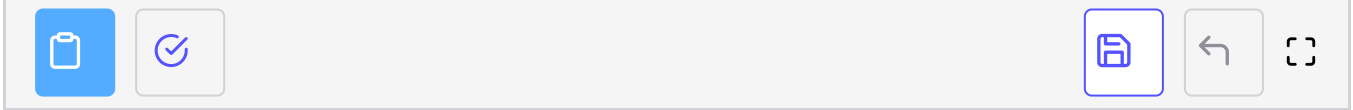
The function will split the input dataset into training and testing sets, and then group the data and labels based on type of set.

Set `split_dataset` equal to `train_test_split` applied with `data` and `labels` as required arguments, as well as `test_size` for the `test_size` keyword argument.

Set `train_set` equal to a tuple containing the first and third elements of `split_dataset`. Also set `test_set` equal to a tuple containing the second and fourth elements of `split_dataset`.

Return a tuple containing `train_set` and `test_set`, in that order.

```
def dataset_splitter(data, labels, test_size=0.25):
  # CODE HERE
  pass
```



Cross-Validation

Learn about K-Fold cross-validation and why it's used.

Chapter Goals:

- Learn about the purpose of cross-validation
- Implement a function that applies the K-Fold cross-validation algorithm to a model

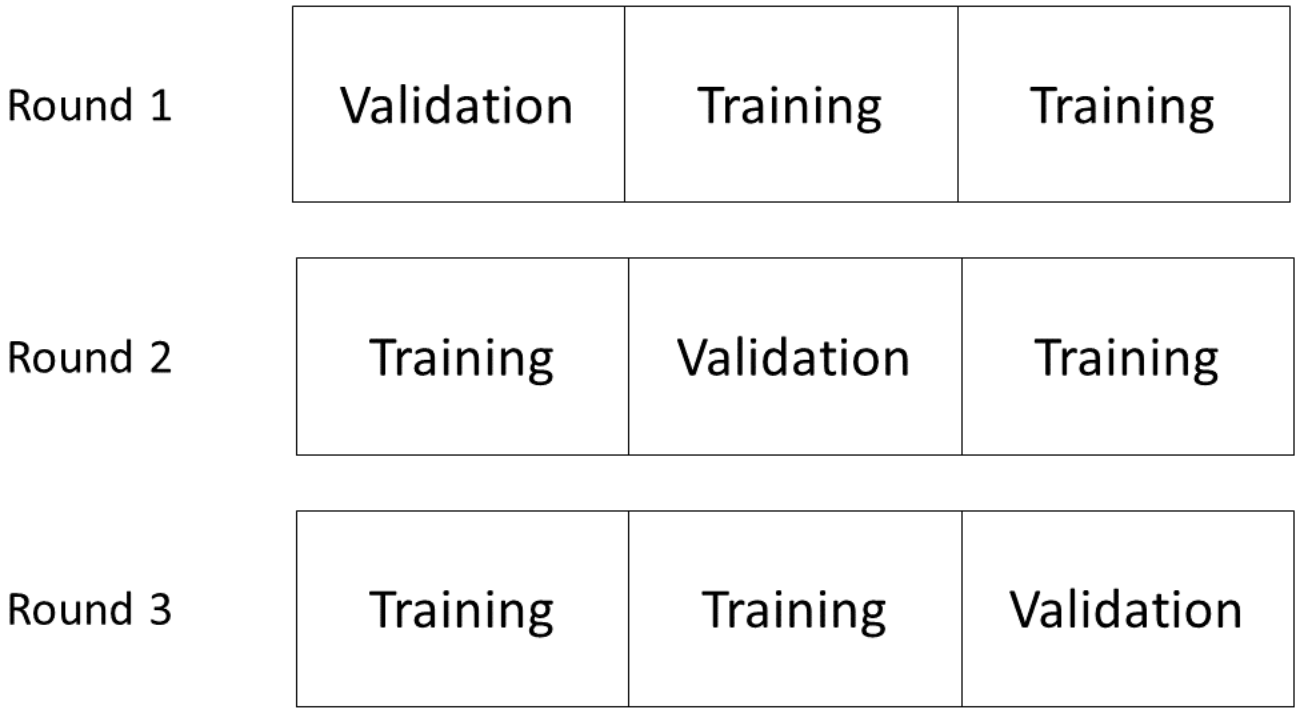
A. Additional evaluation datasets

Sometimes, it's not enough to just have a single testing set for model evaluation. Having additional sets of data for evaluation gives us a more accurate measurement of how good the model is for the original dataset.

If the original dataset is big enough, we can actually split it into three subsets: training, testing, and validation. The validation set is about the same size as the testing set, and it is used for evaluating the model after training. The testing set is then used for final evaluation once the model is done training and tuning.

However, partitioning the original dataset into three distinct sets will cut into the size of the training set. This can reduce the performance of the model if our original dataset is not large enough. A solution to this problem is cross-validation (CV).

Cross-validation creates synthetic validation sets by partitioning the training set into multiple smaller subsets. One of the most common algorithms for cross-validation, K-Fold CV, partitions the training set into **k** approximately equal sized subsets (referred to as *folds*). There are **k** "rounds" of the algorithm, and each "round" chooses one of the **k** subsets for the validation set (a different subset is chosen each round), while the remaining **k - 1** subsets are aggregated into the round's training set and used to train the model.



The K-Fold cross-validation process with 3 folds (k=3)

Each round of the K-Fold algorithm, the model is trained on that round's training set (the combined training folds) and then evaluated on the single validation fold. The evaluation metric depends on the model. For classification models, this is usually classification accuracy on the validation set. For regression models, this can either be the model's mean squared error, mean absolute error, or R^2 value on the validation set.

B. Scored cross-validation

In scikit-learn, we can easily implement K-Fold cross-validation with the `cross_val_score` function (also part of the `model_selection` module). The function returns an array containing the evaluation score for each round.

The code below demonstrates K-Fold CV with 3 folds for classification. The evaluation metric is classification accuracy.

```
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
clf = linear_model.LogisticRegression()
# Predefined data and labels
cv_score = cross_val_score(clf, data, labels, cv=3)  # k = 3

print('{}\n'.format(repr(cv_score)))
```

The code below demonstrates K-Fold CV with 4 folds for regression. The evaluation metric is R^2 value.

```
from sklearn import linear_model
from sklearn.model_selection import cross_val_score
reg = linear_model.LinearRegression()
# Predefined data and labels
cv_score = cross_val_score(reg, data, labels, cv=4)  # k = 4

print('{}\n'.format(repr(cv_score)))
```

Note that we don't call `fit` with the model prior to using `cross_val_score`. This is because the `cross_val_score` function will use `fit` for training the model each round.

For classification models, the `cross_val_score` function will apply a special form of the K-Fold algorithm called *stratified* K-Fold. This just means that each fold will contain approximately the same class distribution as the original dataset. For example, if the original dataset contained 60% class **0** data observations and 40% class **1**, each fold of the stratified K-Fold algorithm will have about the same 60-40 split between class **0** and class **1** data observations.

While cross-validation gives us a better measurement of the model's fit on the original dataset, it can be very time-consuming when used on large datasets. For large enough datasets, it is better to just split it into training, validation, and testing sets, and then use the validation set for evaluating the model before it is finalized.

Applying CV to Decision Trees

Apply K-Fold cross-validation to decision trees.

Chapter Goals:

- Apply K-Fold cross-validation to a decision tree

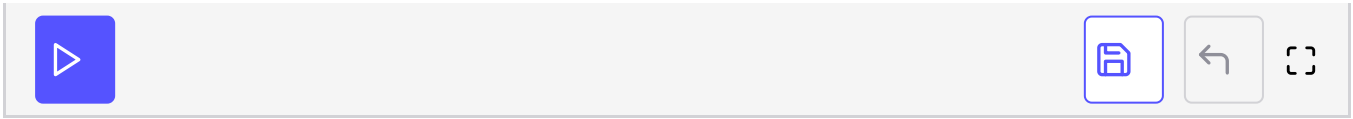
A. Decision tree depth

We've previously discussed cross-validation for tuning hyperparameters such as the α value for regularized regression. For decision trees, we can tune the tree's maximum depth hyperparameter (`max_depth`) by using K-Fold cross-validation.

K-Fold cross-validation gives an accurate measurement of how good the decision tree is for the dataset. We can use K-Fold cross-validation with different values of the `max_depth` hyperparameter and see which one gives the best cross-validation scores.

The code below demonstrates how to apply K-Fold CV to tune a decision tree's maximum depth. It uses the `cv_decision_tree` function that you will implement later in this chapter.

```
is_clf = True  # for classification
for depth in range(3, 8):
  # Predefined data and labels
  scores = cv_decision_tree(
    is_clf, data, labels, depth, 5)  # k = 5
  mean = scores.mean()  # Mean acc across folds
  std_2 = 2 * scores.std()  # 2 std devs
  print('95% C.I. for depth {}: {} +/- {:.2f}\n'.format(
    depth, mean, std_2))
```



In the above code, we use the `cv_decision_tree` function to apply 5-Fold cross-validation to a classification decision tree. We tune its maximum depth

hyperparameter across depths of 3, 4, 5, 6, and 7. For each `max_depth` value,

we print the 95% confidence interval for the cross-validated scores across the 5 folds.

For the most part, the maximum depth of 4 produces the best 95% confidence interval of cross-validated scores. This would be the value of `max_depth` that we choose for the final decision tree.

If the confidence interval had consistently continued to improve for maximum depths of 5, 6 and 7, we would have continued applying the cross-validation process to evaluate larger maximum depth values.

Time to Code!

The coding exercise for this chapter is to complete the aforementioned `cv_decision_tree` function. The function's first argument defines whether the decision tree is for classification/regression, the next two arguments represent the data/labels, and the final two arguments represent the tree's maximum depth and number of folds, respectively.

First, we'll create the decision tree (using the `tree` module imported in the backend).

Initialize `d_tree` with `tree.DecisionTreeClassifier` if `is_clf` is `True`, otherwise use `tree.DecisionTreeRegressor`. In either case, initialize with keyword argument `max_depth` set to `max_depth`.

Then we'll use the `cross_val_score` function (imported in the backend) to obtain the CV scores.

Set `scores` equal to `cross_val_score` applied with `d_tree`, `data`, and `labels` for the first three arguments. Use `cv=cv` for the keyword argument, then return `scores`.

```
def cv_decision_tree(is_clf, data, labels,
                     max_depth, cv):
  # CODE HERE
  pass
```

Evaluating Models

Learn how to evaluate classification and regression models.

Chapter Goals:

- Learn how to evaluate regression and classification models

A. Making predictions

Each of the models we've worked with has a `predict` function, which is used to predict values for new data observations (i.e. data observations not in the training set).

The code below shows an example of making predictions with a regression decision tree.

```
reg = tree.DecisionTreeRegressor()
# predefined train and test sets
reg.fit(train_data, train_labels)
predictions = reg.predict(test_data)
```

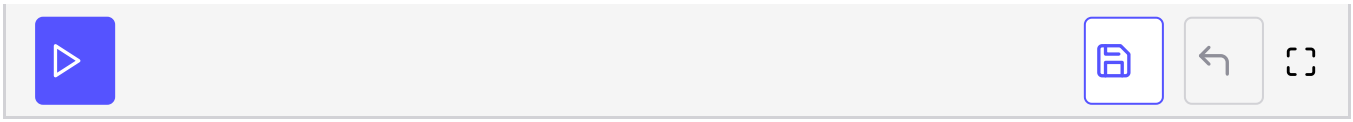
B. Evaluation metrics

For classification models, we use the classification accuracy on the test set as the evaluation metric. For regression models, we normally use either the R^2 value, mean squared error, or mean absolute error on the test set. The most commonly used regression metric is mean absolute error, since it represents the natural definition of error. We use mean squared error when we want to penalize really bad predictions, since the error is squared. We use the R^2 value when we want to evaluate the fit of the regression model on the data.

The `metrics` module of scikit-learn provides functions for each of these metrics. Each of the evaluation functions takes in the actual testing labels as the first argument and the predictions as the second argument.

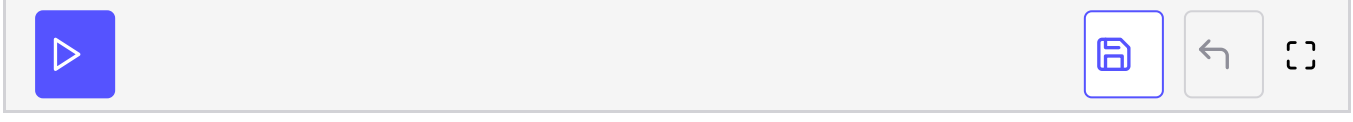
The code below evaluates a regression model's predictions based on the testing labels.

```
reg = tree.DecisionTreeRegressor()
# predefined train and test sets
reg.fit(train_data, train_labels)
predictions = reg.predict(test_data)

from sklearn import metrics
r2 = metrics.r2_score(test_labels, predictions)
print('R2: {}\n'.format(r2))
mse = metrics.mean_squared_error(
  test_labels, predictions)
print('MSE: {}\n'.format(mse))
mae = metrics.mean_absolute_error(
  test_labels, predictions)
print('MAE: {}\n'.format(mae))
```



The code below evaluates a classification model's predictions based on the testing labels.

```
clf = tree.DecisionTreeClassifier()
# predefined train and test sets
clf.fit(train_data, train_labels)
predictions = clf.predict(test_data)

from sklearn import metrics
acc = metrics.accuracy_score(test_labels, predictions)
print('Accuracy: {}\n'.format(acc))
```



Exhaustive Tuning

Use exhaustive grid search techniques for hyperparameter tuning.

Chapter Goals:

- Learn how to use grid search cross-validation for exhaustive hyperparameter tuning

A. Grid-search cross-validation

If our application requires us to absolutely obtain the best hyperparameters of a model, and if the dataset is small enough, we can apply an exhaustive grid search for tuning hyperparameters. For the grid search cross-validation, we specify possible values for each hyperparameter, and then the search will go through each possible combination of the hyperparameters and return the model with the best combination.

We implement grid search cross-validation with the `GridSearchCV` object (part of the `model_selection` module).

```
reg = linear_model.BayesianRidge()
params = {
  'alpha_1':[0.1,0.2,0.3],
  'alpha_2':[0.1,0.2,0.3]
}
reg_cv = GridSearchCV(reg, params, cv=5, iid=False)
# predefined train and test sets
reg_cv.fit(train_data, train_labels)
print(reg_cv.best_params_)
```

In the code example above, we searched through each possible pair of α_1 and α_2 values based on the two lists in the `params` dictionary. The search resulted in an α_1 value of 0.3 and an α_2 value of 0.1. For each of the models we've covered, you can take a look at their respective scikit-learn code documentation pages to determine the model's hyperparameters that can be

used as the `params` argument for `GridSearchCV`.

The `cv` keyword argument represents the number of folds used in the K-Fold cross-validation for grid search. The `iid` keyword argument relates to how the cross-validation score is calculated. We use `False` to match the standard definition of cross-validation. Note that in later updates of scikit-learn, the `iid` argument will be removed from `GridSearchCV`.

Since exhaustive grid search performs cross-validation on each possible hyperparameter value combination, it can be incredibly slow for larger datasets. It should only be used if the dataset is reasonably small and it is important to choose the best hyperparameter combination.

Quiz

1 Why is regularization important in regression modeling?

COMPLETED 0%

1 of 4

Introduction

An overview of unsupervised learning and clustering.

In the **Clustering** section, you will be using unsupervised learning methods, i.e. methods of extracting insights from unlabeled datasets. Specifically, you will learn about different *clustering* algorithms and how they are able to group together similar data observations.

A. Unsupervised learning

So far, we've only used supervised learning methods, since we've exclusively been dealing with labeled datasets. However, in the real world many datasets are completely unlabeled, since labeling datasets involves additional work and foresight. Rather than just ignoring all these unlabeled datasets, we can still extract meaningful insights using unsupervised learning.

Since we only have data observations to work with, and no labels, unsupervised learning methods are centered around finding similarities/differences between data observations and making inferences based on those findings. The most commonly used form of unsupervised learning is clustering. As the name suggests, clustering algorithms will cluster the data into distinct groups (clusters), where each cluster consists of similar data observations.

Clustering is used in many different applications, from anomaly detection (i.e. detecting real vs. fraudulent data) to market research (e.g. grouping customers together based on their purchase history). In the upcoming chapters, you'll learn about a variety of commonly used clustering algorithms in data science, as well as other tools for finding similarities between data observations.

Cosine Similarity

Learn about the cosine similarity metric and how it's used.

Chapter Goals:

- Understand how the cosine similarity metric measures the similarity between two data observations

A. What defines similarity?

To find similarities between data observations, we first need to understand how to actually measure similarity. The most common measurement of similarity is the cosine similarity metric.

A data observation with numeric features is essentially just a vector of real numbers. Cosine similarity is used in mathematics as a similarity metric for real-valued vectors, so it makes sense to use it as a similarity metric for data observations. The cosine similarity for two data observations is a number between -1 and 1. It specifically measures the *proportional* similarity of the feature values between the two data observations (i.e. the ratio between feature columns).

Cosine similarity values closer to 1 represent greater similarity between the observations, while values closer to -1 represent more divergence. A value of 0 means that the two data observations have no correlation (neither similar nor dissimilar).

B. Calculating cosine similarity

The cosine similarity for two vectors, u and v, is calculated as the dot product between the L2-normalization of the vectors. The exact formula for cosine similarity is:

$$\text{cossim}(u, v) = \frac{u}{||u||_2} \cdot \frac{v}{||v||_2}$$

where $||u||_2$ represents the L2 norm of u and $||v||_2$ represents the L2 norm of v.
In scikit-learn, cosine similarity is implemented via the `cosine_similarity` function (which is part of the `metrics.pairwise` module). It calculates the cosine similarities for pairs of data observations in a single dataset, or pairs of data observations between two datasets.

The code below computes cosine similarities between pairs of observations in a 2-D dataset.

```
from sklearn.metrics.pairwise import cosine_similarity
data = np.array([
  [ 1.1,  0.3],
  [ 2.1,  0.6],
  [-1.1, -0.4],
  [ 0. , -3.2]])
cos_sims = cosine_similarity(data)
print('{}\n'.format(repr(cos_sims)))
```

When we only pass in one dataset into `cosine_similarity`, the function will compute cosine similarities between pairs of observations within the dataset. In the code above, we passed in `data` (which contains 4 data observations), so the output of `cosine_similarity` is a 4x4 array of cosine similarity values.

The value at index (i, j) of `cos_sims` is the cosine similarity between data observations i and j in `data`. Since cosine similarity is symmetric, the `cos_sims` array contains the same values at index (i, j) and (j, i).

Note that the cosine similarity between a data observation and itself is 1, unless the data observation contains only 0's as feature values (in which case the cosine similarity is 0).

If we decide to pass in two datasets (with equal numbers of columns) into `cosine_similarity`, the function will compute the cosine similarities for pairs of data observations between the two datasets.

```
from sklearn.metrics.pairwise import cosine_similarity
data = np.array([
  [ 1.1,  0.3],
  [ 2.1,  0.6],
  [-1.1, -0.4],
```

```
  [ 0. , -3.2]])
data2 = np.array([
  [ 1.7,  0.4],
  [ 4.2, 1.25],
  [-8.1,  1.2]])
cos_sims = cosine_similarity(data, data2)
print('{}\n'.format(repr(cos_sims)))
```

In the code above, the value at index (i, j) of `cos_sims` is the cosine similarity between data observation i in `data` and data observation j in `data2`. Note that `cos_sims` is a 4x3 array, since `data` contains 4 data observations and `data2` contains 3.

Time to Code!

The code exercise for this chapter will be to use the `cosine_similarity` function to compute the most similar data observations for each data observation in `data`. Both `cosine_similarity` and `data` are imported/initialized in the backend.

First, we need to calculate the pairwise cosine similarities for each data observation.

Set `cos_sims` equal to `cosine_similarity` applied to `data`.

```
# CODE HERE
```

The diagonal in `cos_sims` represents the similarities between each observation and itself. We'll substitute each diagonal value with 0, so that for each data observation we find the most similar observation *besides itself*.

In NumPy, the function `fill_diagonal` allows us to fill the diagonal of an array with a specified value.

Call `np.fill_diagonal` with `cos_sims` as the first argument and `0` as the second argument.

```
# CODE HERE
```

For each row of `cos_sims`, the column containing the largest cosine similarity score represents the most similar data observation. We can find the column indexes with the largest value by using the `argmax` function of `cos_sims`.

We set the keyword argument `axis` equal to 1 to specify the largest column indexes for each row.

Set `similar_indexes` equal to the output of `cos_sims.argmax` with the `axis` keyword argument equal to `1`.

```
# CODE HERE
```

Nearest Neighbors

Understand the purpose of finding nearest neighbors for data points.

Chapter Goals:

- Learn how to find the nearest neighbors for a data observation

A. Finding the nearest neighbors

In Chapter 1, we mentioned that clustering is a method for grouping together similar data observations. Another method for finding similar data observations is the *nearest neighbors* approach. With this approach, we find the *k* most similar data observations (i.e. neighbors) for a given data observation (where *k* represents the number of neighbors).

In scikit-learn, we implement the nearest neighbors approach with the `NearestNeighbors` object (part of the `neighbors` module).

The code below finds the 5 nearest neighbors for a new data observation (`new_obs`) based on its fitted dataset (`data`).

```
data = np.array([
  [5.1, 3.5, 1.4, 0.2],
  [4.9, 3. , 1.4, 0.2],
  [4.7, 3.2, 1.3, 0.2],
  [4.6, 3.1, 1.5, 0.2],
  [5. , 3.6, 1.4, 0.2],
  [5.4, 3.9, 1.7, 0.4],
  [4.6, 3.4, 1.4, 0.3],
  [5. , 3.4, 1.5, 0.2],
  [4.4, 2.9, 1.4, 0.2],
  [4.9, 3.1, 1.5, 0.1]])

from sklearn.neighbors import NearestNeighbors
nbrs = NearestNeighbors()
nbrs.fit(data)
new_obs = np.array([[5. , 3.5, 1.6, 0.3]])
dists, knbrs = nbrs.kneighbors(new_obs)

# nearest neighbors indexes
print('{}\n'.format(repr(knbrs)))
# nearest neighbor distances
print('{}\n'.format(repr(dists)))
```

```
only_nbrs = nbrs.kneighbors(new_obs,

                            return_distance=False)
print('{}\n'.format(repr(only_nbrs)))
```

The `NearestNeighbors` object is fitted with a dataset, which is then used as the pool of possible neighbors for new data observations. The `kneighbors` function takes in new data observation(s) and returns the *k* nearest neighbors along with their respective distances from the input data observations. Note that the nearest neighbors are the neighbors with the smallest distances from the input data observation. We can choose not to return the distances by setting the `return_distance` keyword argument to `False`.

The default value for *k* when initializing the `NearestNeighbors` object is 5. We can specify a new value using the `n_neighbors` keyword argument.

```
data = np.array([
  [5.1, 3.5, 1.4, 0.2],
  [4.9, 3. , 1.4, 0.2],
  [4.7, 3.2, 1.3, 0.2],
  [4.6, 3.1, 1.5, 0.2],
  [5. , 3.6, 1.4, 0.2],
  [5.4, 3.9, 1.7, 0.4],
  [4.6, 3.4, 1.4, 0.3],
  [5. , 3.4, 1.5, 0.2],
  [4.4, 2.9, 1.4, 0.2],
  [4.9, 3.1, 1.5, 0.1]])

from sklearn.neighbors import NearestNeighbors
nbrs = NearestNeighbors(n_neighbors=2)
nbrs.fit(data)
new_obs = np.array([
  [5. , 3.5, 1.6, 0.3],
  [4.8, 3.2, 1.5, 0.1]])
dists, knbrs = nbrs.kneighbors(new_obs)

# nearest neighbors indexes
print('{}\n'.format(repr(knbrs)))
# nearest neighbor distances
print('{}\n'.format(repr(dists)))
```

In the code above, the first row of `knbrs` and `dists` correspond to the first data observation in `new_obs`, while the second row of `knbrs` and `dists`

correspond to the second observation in `new_obs`.

K-Means Clustering

Learn about the K-Means clustering algorithm and how it works.

Chapter Goals:

- Learn about K-means clustering and how it works
- Understand why mini-batch clustering is used for large datasets

A. K-means algorithm

The idea behind clustering data is pretty simple: partition a dataset into groups of similar data observations. How we go about finding these clusters is a bit more complex, since there are a number of different methods for clustering datasets.

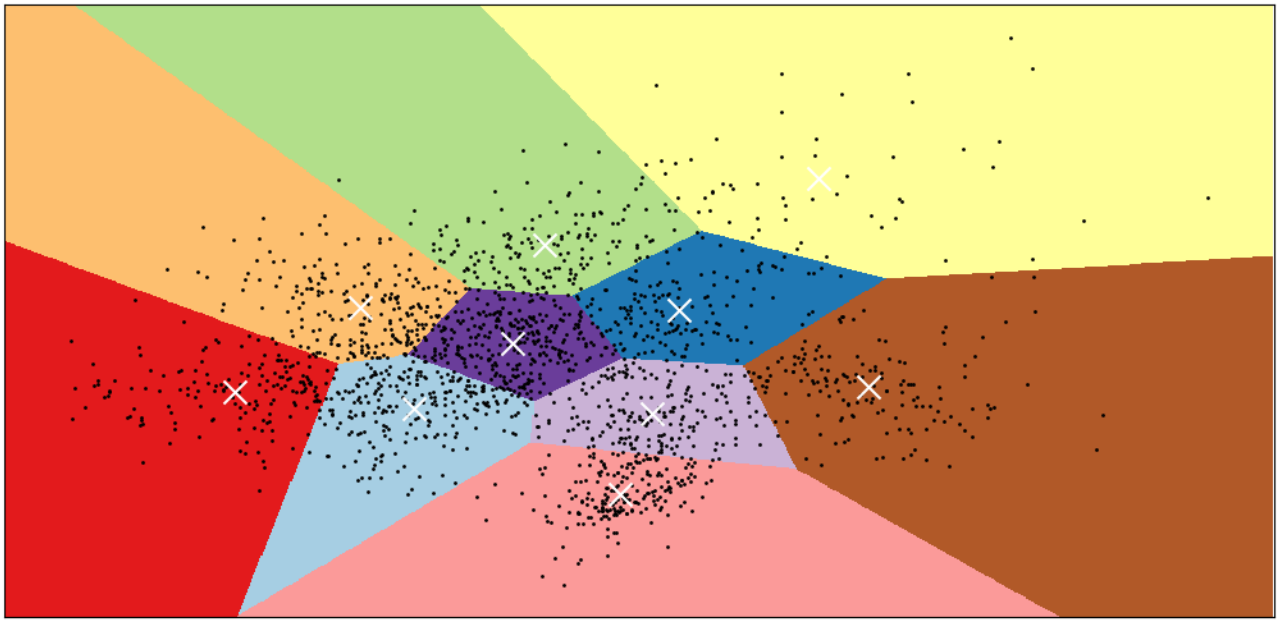
The most well-known clustering method is K-means clustering. The K-means clustering algorithm will separate the data into *K* clusters (the number of clusters is chosen by the user) using cluster means, also known as *centroids*.

These centroids represent the "centers" of each cluster. Specifically, a cluster's centroid is equal to the average of all the data observations within the cluster.

```
cluster = np.array([
  [ 1.2, 0.6],
  [ 2.4, 0.8],
  [-1.6, 1.4],
  [ 0. , 1.2]])
print('Cluster:\n{}\n'.format(repr(cluster)))

centroid = cluster.mean(axis=0)
print('Centroid:\n{}\n'.format(repr(centroid)))
```

The K-means clustering algorithm is an iterative process. Each iteration, the algorithm will assign each data observation to the cluster with the closest centroid to the observation (using the regular distance metric).

Then it updates each centroid to be equal to the new average of the data observations in the cluster. Note that at the beginning of the algorithm, the cluster centroids are either randomly initialized or (better) initialized using the K-means++ algorithm. The clustering process stops when there are no more changes in cluster assignment for any data observation.



An example of K-means clustering on a dataset with 10 clusters chosen (K = 10). Clusters are distinguished by color. The white crosses represent the centroids of each cluster.

In scikit-learn, K-means clustering is implemented using the `KMeans` object (part of the `cluster` module).

The code below demonstrates how to use the `KMeans` object (with 3 clusters).

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3)
# predefined data
kmeans.fit(data)

# cluster assignments
print('{}\n'.format(repr(kmeans.labels_)))

# centroids
print('{}\n'.format(repr(kmeans.cluster_centers_)))

new_obs = np.array([
  [5.1, 3.2, 1.7, 1.9],
  [6.9, 3.2, 5.3, 2.2]])
# predict clusters
print('{}\n'.format(repr(kmeans.predict(new_obs))))
```

The `KMeans` object uses K-means++ centroid initialization by default. The `n_clusters` keyword argument lets us set the number of clusters we want. In the example above, we applied clustering to `data` using 3 clusters.

The `labels_` attribute of the object tells us the final cluster assignments for each data observation, and the `cluster_centers_` attribute represents the final centroids. We use the `predict` function to assign new data observations to one of the clusters.

B. Mini-batch clustering

When working with very large datasets, regular K-means clustering can be quite slow. To reduce the computation time, we can perform *mini-batch* K-means clustering, which is just regular K-means clustering applied to randomly sampled subsets of the data (mini-batches) at a time.

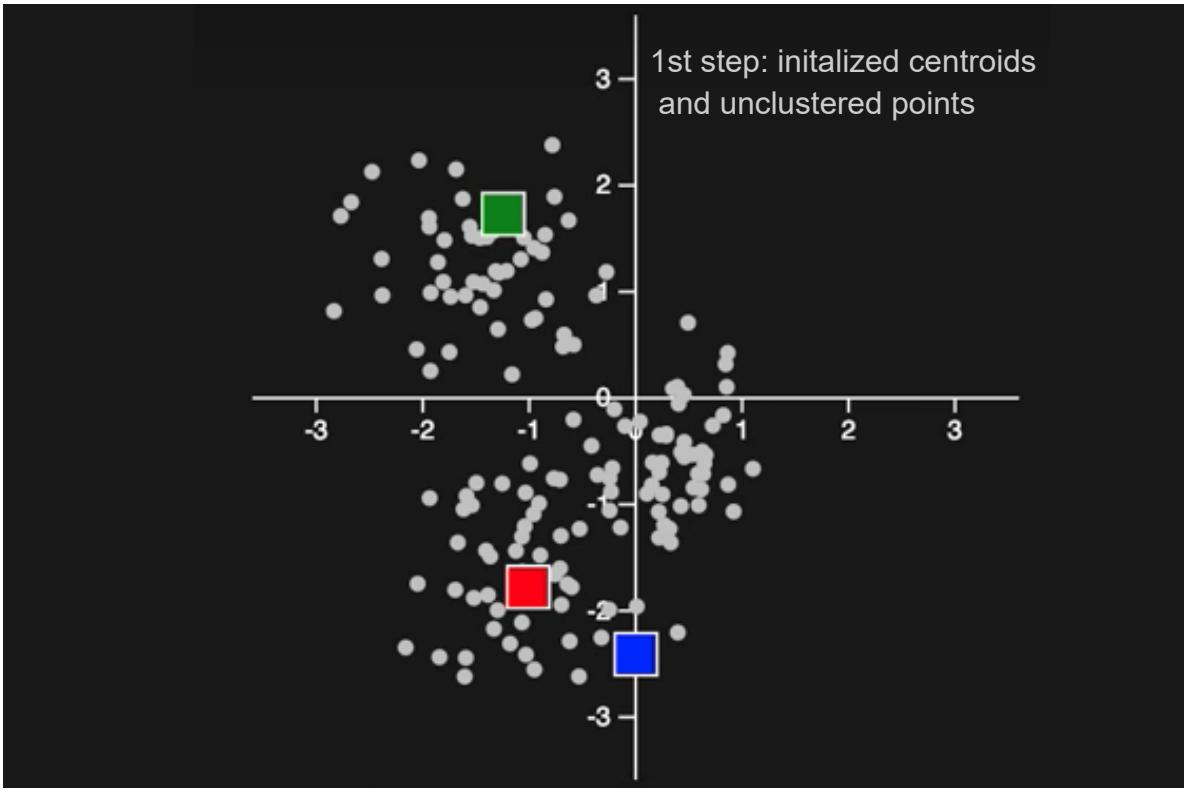
There is a trade-off in using mini-batch clustering, as the results may not be as good as regular K-means clustering. However, in practice the difference in quality is negligible, so mini-batch clustering is usually the choice when dealing with large datasets.

In scikit-learn, mini-batch K-means clustering is implemented using the `MiniBatchKMeans` object (also part of the `cluster` module). It is used in the same way as the regular `KMeans` object, with an additional `batch_size` keyword argument during initialization that allows us to specify the size of each mini-batch.

```
from sklearn.cluster import MiniBatchKMeans
kmeans = MiniBatchKMeans(n_clusters=3, batch_size=10)
# predefined data
kmeans.fit(data)

# cluster assignments
print('{}\n'.format(repr(kmeans.labels_)))

# centroids
print('{}\n'.format(repr(kmeans.cluster_centers_)))

new_obs = np.array([
  [5.1, 3.2, 1.7, 1.9],
  [6.9, 3.2, 5.3, 2.2]])
# predict clusters
```

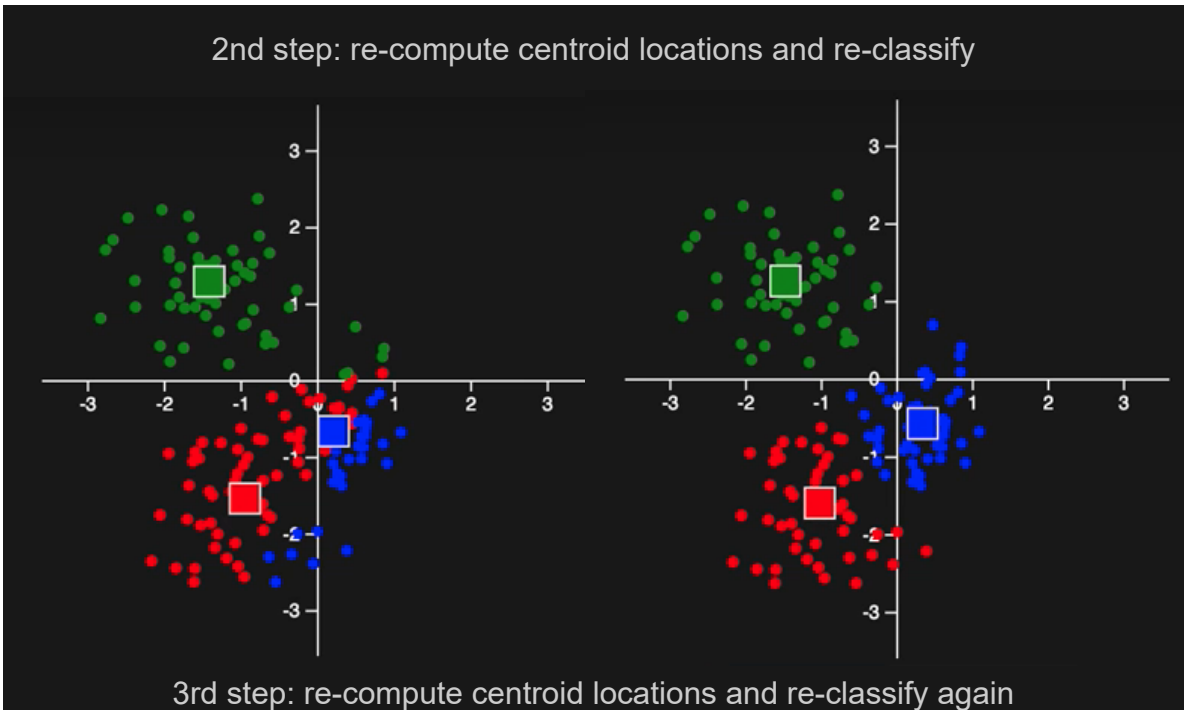
```
print('{}\n'.format(repr(kmeans.predict(new_obs))))
```

Note that the clusterings can have different permutations, i.e. different cluster labelings (0, 1, 2 vs. 1, 2, 0). Otherwise, the cluster assignments for both `KMeans` and `MiniBatchKMeans` should be relatively the same.



Example of the K-Means algorithm's multiple steps.

1 of 2



The most noticeable difference is which points are in the red and blue clusters

Time to Code!

The coding exercise for this chapter will be to complete the `kmeans_clustering` function, which will use either `KMeans` or `MiniBatchKMeans` for clustering `data`.

Which object we use for clustering depends on the value of `batch_size`.

If `batch_size` is `None`, set `kmeans` equal to `KMeans`, initialized with `n_clusters` for the `n_clusters` argument. Otherwise, set `kmeans` equal to `MiniBatchKMeans`, initialized with `n_clusters` for the `n_clusters` argument and `batch_size` for the `batch_size` argument.

After setting up `kmeans`, we fit it on the `data` and return it.

Call `kmeans.fit` with `data` as the only argument. Then return `kmeans`.

```
def kmeans_clustering(data, n_clusters, batch_size):
  # CODE HERE
  pass
```

Hierarchical Clustering

Learn about hierarchical clustering via the agglomerative approach.

Chapter Goals:

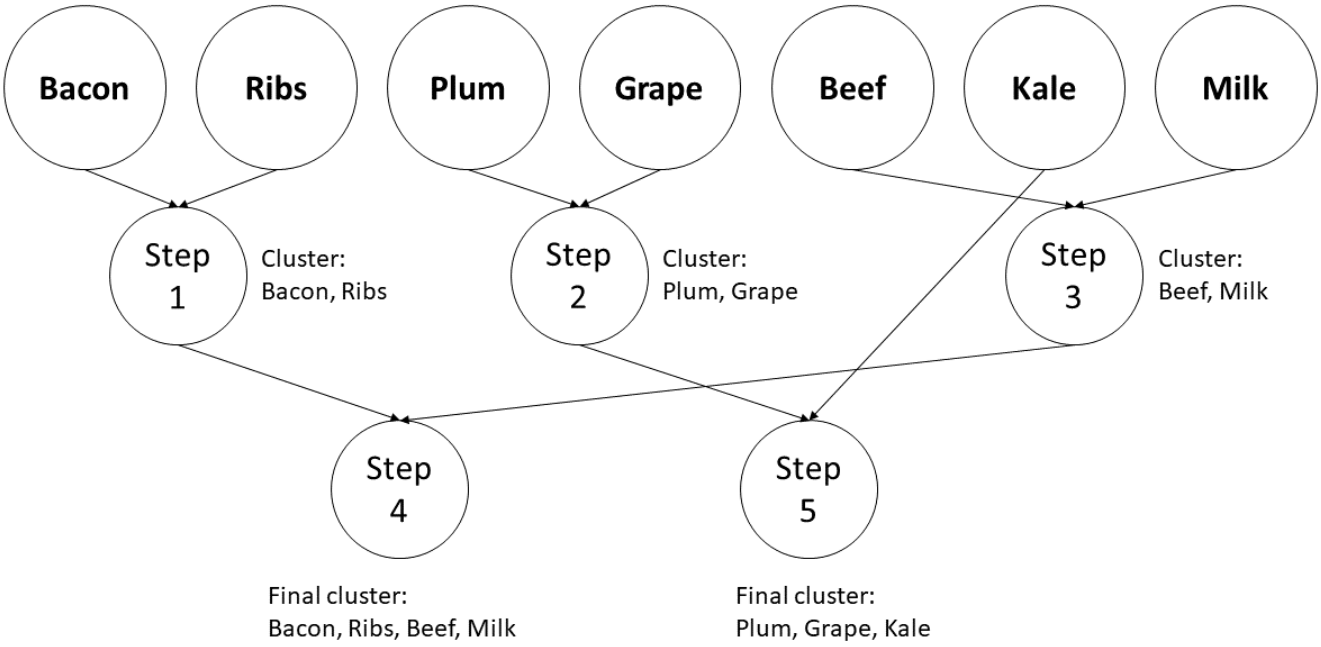
- Learn about hierarchical clustering using the agglomerative approach

A. K-means vs. hierarchical clustering

A major assumption that the K-means clustering algorithm makes is that the dataset consists of *spherical* (i.e. circular) clusters. With this assumption, the K-means algorithm will create clusters of data observations that are circular around the centroids. However, real life data often does not contain spherical clusters, meaning that K-means clustering might end up producing inaccurate clusters due to its assumption.

An alternative to K-means clustering is hierarchical clustering. Hierarchical clustering allows us to cluster any type of data, since it doesn't make any assumptions about the data or clusters.

There are two approaches to hierarchical clustering: bottom-up (divisive) and top-down (agglomerative). The divisive approach initially treats all the data as a single cluster, then repeatedly splits it into smaller clusters until we reach the desired number of clusters. The agglomerative approach initially treats each data observation as its own cluster, then repeatedly merges the two most similar clusters until we reach the desired number of clusters.



Agglomerative clustering for seven different food items into two final clusters. Each step the two most similar clusters are merged.

In practice, the agglomerative approach is more commonly used due to better algorithms for the approach. Therefore, we'll focus on using agglomerative clustering in this chapter.

B. Agglomerative clustering

In scikit-learn, agglomerative clustering is implemented through the `AgglomerativeClustering` object (part of the `cluster` module). Similar to the `KMeans` object from the previous chapter, we specify the number of clusters with the `n_clusters` keyword argument.

The code below demonstrates how to use the `AgglomerativeClustering` object (with 3 clusters).

```
from sklearn.cluster import AgglomerativeClustering
agg = AgglomerativeClustering(n_clusters=3)
# predefined data
agg.fit(data)

# cluster assignments
print('{}\n'.format(repr(agg.labels_)))
```

Since agglomerative clustering doesn't make use of centroids, there's no `cluster_centers_` attribute in the `AgglomerativeClustering` object. There's also no `predict` function for making cluster predictions on new data (since K-means clustering makes use of its final centroids for new data predictions).

Mean Shift Clustering

Use mean shift clustering to determine the optimal number of clusters.

Chapter Goals:

- Learn about the mean shift clustering algorithm

A. Choosing the number of clusters

Each of the clustering algorithms we've used so far require us to pass in the number of clusters. This is fine if we already know the number of clusters we want, or have a good guess for the number of clusters. However, if we don't have a good idea of what the actual number of clusters for the dataset should be, there exist algorithms that can automatically choose the number of clusters for us.

One such algorithm is the mean shift clustering algorithm. Like the K-means clustering algorithm, the mean shift algorithm is based on finding cluster centroids. Since we don't provide the number of clusters, the algorithm will look for "blobs" in the data that can be potential candidates for clusters.

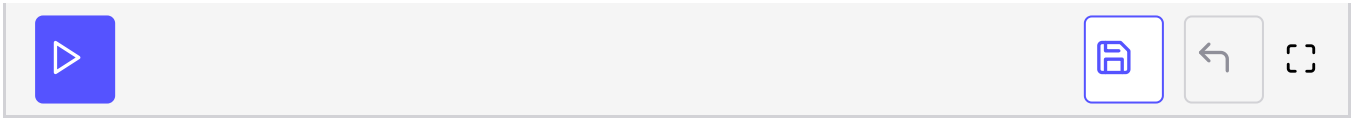
Using these "blobs", the algorithm finds a number of candidate centroids. It then removes the candidates that are basically duplicates of others to form the final set of centroids. The final set of centroids determines the number of clusters as well as the dataset cluster assignments (data observations are assigned to the nearest centroid).

In scikit-learn, the mean shift algorithm is implemented with the `MeanShift` object (part of the `cluster` module). Since the algorithm doesn't require us to pass in the number of clusters, we can initialize the `MeanShift` with no arguments.

The code below demonstrates how to use the `MeanShift` object.

```
from sklearn.cluster import MeanShift
mean_shift = MeanShift()
```

```
mean_shift = MeanShift()
# predefined data
mean_shift.fit(data)

# cluster assignments
print('{}\n'.format(repr(mean_shift.labels_)))

# centroids
print('{}\n'.format(repr(mean_shift.cluster_centers_)))

new_obs = np.array([
  [5.1, 3.2, 1.7, 1.9],
  [6.9, 3.2, 5.3, 2.2]])
# predict clusters
print('{}\n'.format(repr(mean_shift.predict(new_obs))))
```



Since mean shift is a centroid-based algorithm, the `MeanShift` object contains the `cluster_centers_` attribute (the final centroids) and the `predict` function.

DBSCAN

Learn about the DBSCAN clustering algorithm.

Chapter Goals:

- Learn about the DBSCAN algorithm

A. Clustering by density

The mean shift clustering algorithm in the previous chapter usually performs sufficiently well and can choose a reasonable number of clusters. However, it is not very scalable due to computation time and still makes the assumption that clusters have a "blob"-like shape (although this assumption is not as strong as the one made by K-means).

Another clustering algorithm that also automatically chooses the number of clusters is DBSCAN. DBSCAN clusters data by finding *dense* regions in the dataset. Regions in the dataset with many closely packed data observations are considered *high-density* regions, while regions with sparse data are considered *low-density* regions.

The DBSCAN algorithm treats high-density regions as clusters in the dataset, and low-density regions as the area between clusters (so observations in the low-density regions are treated as noise and not placed in a cluster).

High-density regions are defined by *core samples*, which are just data observations with many neighbors. Each cluster consists of several core samples and all the observations that are neighbors to a core sample.

Unlike the mean shift algorithm, the DBSCAN algorithm is both highly scalable and makes no assumptions about the underlying shape of clusters in the dataset.

B. Neighbors and core samples

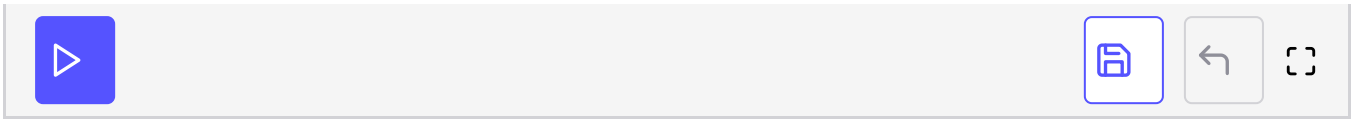
The exact definition of "neighbor" and "core sample" depends on what we

want in our clusters. We specify the maximum distance, ε, between two data observations that are considered neighbors. Smaller distances result in smaller and more tightly packed clusters. We also specify the minimum number of points in the neighborhood of a data observation for the observation to be considered a core sample (the neighborhood consists of the data observation and all its neighbors).

In scikit-learn, we implement DBSCAN with the `DBSCAN` object (part of the `cluster` module). The object is initialized with the keyword arguments `eps` (representing the value of ε) and `min_samples` (representing the minimum size of a core sample's neighborhood).

The code below demonstrates how to use the `DBSCAN` object, with ε equal to 1.2 and a minimum size of 30 for a core sample's neighborhood.

```
from sklearn.cluster import DBSCAN
dbscan = DBSCAN(eps=1.2, min_samples=30)
# predefined data
dbscan.fit(data)

# cluster assignments
print('{}\n'.format(repr(dbscan.labels_)))

# core samples
print('{}\n'.format(repr(dbscan.core_sample_indices_)))
num_core_samples = len(dbscan.core_sample_indices_)
print('Num core samples: {}\n'.format(num_core_samples))
```



In the code above, we used `DBSCAN` to cluster the 150 data observations in `data`. The algorithm found two clusters. In this case all the data observations fit in a cluster, but in general the non-cluster observations would be labeled with `-1`.

The `core_sample_indices_` attribute represents the core sample data observations in `data` (specified by row index).

Evaluating Clusters

Learn how to evaluate the performance of clustering algorithms.

Chapter Goals:

- Learn how to evaluate clustering algorithms

A. Evaluation metrics

When we don't have access to any true cluster assignments (labels), the best we can do to evaluate clusters is to just take a look at them and see if they make sense with respect to the dataset and domain. However, if we do have access to the true cluster labels for the data observations, we can apply a number of metrics to evaluate our clustering algorithm.

One popular evaluation metric is the adjusted Rand index. The regular Rand index gives a measurement of similarity between the true clustering assignments (true labels) and the predicted clustering assignments (predicted labels). The adjusted Rand index (ARI) is a corrected-for-chance version of the regular one, meaning that the score is adjusted so that random clustering assignments will not have a good score.

The ARI value ranges from -1 to 1, inclusive. Negative scores represent bad labelings, random labelings will get a score near 0, and perfect labelings get a score of 1.

In scikit-learn, ARI is implemented through the `adjusted_rand_score` function (part of the `metrics` module). It takes in two required arguments, the true cluster labels and the predicted cluster labels, and returns the ARI score.

```
from sklearn.metrics import adjusted_rand_score
true_labels = np.array([0, 0, 0, 1, 1, 1])
pred_labels = np.array([0, 0, 1, 1, 2, 2])

ari = adjusted_rand_score(true_labels, pred_labels)
print('{}\n'.format(ari))

# symmetric
```

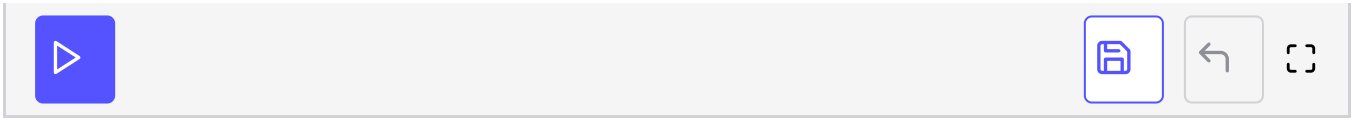
```
ari = adjusted_rand_score(pred_labels, true_labels)
print('{}\n'.format(ari))

# Perfect labeling
perf_labels = np.array([0, 0, 0, 1, 1, 1])
ari = adjusted_rand_score(true_labels, perf_labels)
print('{}\n'.format(ari))

# Perfect labeling, permuted
permuted_labels = np.array([1, 1, 1, 0, 0, 0])
ari = adjusted_rand_score(true_labels, permuted_labels)
print('{}\n'.format(ari))

renamed_labels = np.array([1, 1, 1, 3, 3, 3])
# Renamed labels to 1, 3
ari = adjusted_rand_score(true_labels, renamed_labels)
print('{}\n'.format(ari))

true_labels2 = np.array([0, 1, 2, 0, 3, 4, 5, 1])
# Bad labeling
pred_labels2 = np.array([1, 1, 0, 0, 2, 2, 2, 2])
ari = adjusted_rand_score(true_labels2, pred_labels2)
print('{}\n'.format(ari))
```

Note that the `adjusted_rand_score` function is symmetric. This means that changing the order of the arguments will not affect the score. Furthermore, permutations in the labeling or changing the label names (i.e. `0` and `1` vs. `1` and `3`) does not affect the score.

Another common clustering evaluation metric is adjusted mutual information (AMI). It is implemented in scikit-learn with the `adjusted_mutual_info_score` function (also part of the `cluster` module). Like `adjusted_rand_score`, the function is symmetric and oblivious to permutations and renamed labels.

```
from sklearn.metrics import adjusted_mutual_info_score
true_labels = np.array([0, 0, 0, 1, 1, 1])
pred_labels = np.array([0, 0, 1, 1, 2, 2])

ami = adjusted_mutual_info_score(true_labels, pred_labels)
print('{}\n'.format(ami))

# symmetric
ami = adjusted_mutual_info_score(pred_labels, true_labels)
print('{}\n'.format(ami))

# Perfect labeling
perf_labels = np.array([0, 0, 0, 1, 1, 1])
ami = adjusted_mutual_info_score(true_labels, perf_labels)
print('{}\n'.format(ami))

# Perfect labeling, permuted
```

```
# Perfect labeling, permuted
permuted_labels = np.array([1, 1, 1, 0, 0, 0])
ami = adjusted_mutual_info_score(true_labels, permuted_labels)
print('{}\n'.format(ami))

renamed_labels = np.array([1, 1, 1, 3, 3, 3])
# Renamed labels to 1, 3
ami = adjusted_mutual_info_score(true_labels, renamed_labels)
print('{}\n'.format(ami))

true_labels2 = np.array([0, 1, 2, 0, 3, 4, 5, 1])
# Bad labeling
pred_labels2 = np.array([1, 1, 0, 0, 2, 2, 2, 2])
ami = adjusted_mutual_info_score(true_labels2, pred_labels2)
print('{}\n'.format(ami))
```



The ARI and AMI metrics are very similar. They both assign a score of 1.0 to perfect labelings, a score near 0.0 to random labelings, and negative scores to poor labelings.

A general rule of thumb of when to use which: ARI is used when the true clusters are large and approximately equal sized, while AMI is used when the true clusters are unbalanced in size and there exist small clusters.

Feature Clustering

Use agglomerative clustering for feature dimensionality reduction.

Chapter Goals:

- Learn how to use agglomerative clustering for feature dimensionality reduction

A. Agglomerative feature clustering

In the **Data Preprocessing** section, we used PCA to perform feature dimensionality reduction on datasets. We can also perform feature dimensionality reduction using agglomerative clustering. By merging common features into clusters, we reduce the number of total features while still maintaining most of the original information from the dataset.

In scikit-learn, we perform agglomerative clustering on features using the `FeatureAgglomeration` object (part of the `cluster` module). When initializing the object, `n_clusters` keyword argument (which represents the number of final clusters) is used to specify the new feature dimension of the data.

The code below demonstrates how to use the `FeatureAgglomeration` object to reduce feature dimensionality from 4 to 2. We use the object's `fit_transform` function to fit the clustering model on the data, then subsequently apply the feature reduction on the data.

```
# predefined data
print('Original shape: {}\n'.format(data.shape))
print('First 10:\n{}\n'.format(repr(data[:10])))

from sklearn.cluster import FeatureAgglomeration
agg = FeatureAgglomeration(n_clusters=2)
new_data = agg.fit_transform(data)
print('New shape: {}\n'.format(new_data.shape))
print('First 10:\n{}\n'.format(repr(new_data[:10])))
```

Quiz

1 What does cosine similarity measure (in terms of a dataset)?

COMPLETED 0%

1 of 4

Introduction

An intro to XGBoost and gradient boosted decision trees.

In the **Gradient Boosting** section, you will be learning about XGBoost, a library for highly efficient gradient boosted decision trees. It is one of the premier libraries used in data science for classification and regression.

A. XGBoost vs. scikit-learn

The previous 3 sections of this course used scikit-learn for a variety of tasks. In this section, we cover XGBoost, a state-of-the-art data science library for performing classification and regression. XGBoost makes use of gradient boosted decision trees, which provides better performance than regular decision trees.

In addition to the performance boost, XGBoost implements an extremely efficient version of gradient boosted trees. The XGBoost models train much faster than scikit-learn models, while still providing the same ease of use.

For data science and machine learning competitions that use small to medium sized datasets (e.g. Kaggle), XGBoost is always among the top performing models.

B. Gradient boosted trees

The problem with regular decision trees is that they are often not complex enough to capture the intricacies of many large datasets. We could continuously increase the maximum depth of a decision tree to fit larger datasets, but decision trees with many nodes tend to overfit the data.

Instead, we make use of gradient boosting to *combine* many decision trees into a single model for classification or regression. Gradient boosting starts off with a single decision tree, then iteratively adds more decision trees to the overall model to correct the model's errors on the training dataset.

The XGBoost API handles the gradient boosting process for us, which produces a much better model than if we had used a single decision tree.

XGBoost Basics

Learn about the basics of using XGBoost.

Chapter Goals:

- Learn about the XGBoost data matrix
- Train a `Booster` object in XGBoost

A. Basic data structures

The basic data structure for XGBoost is the `DMatrix`, which represents a data matrix. The `DMatrix` can be constructed from NumPy arrays.

The code below creates `DMatrix` objects with and without labels.

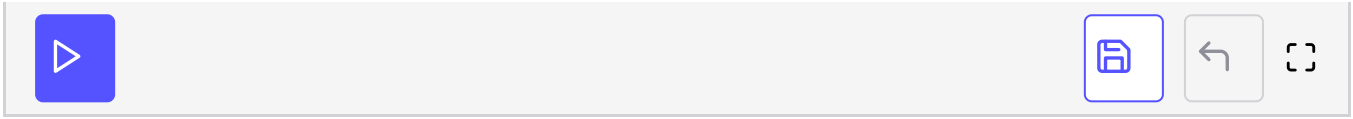
```
data = np.array([
  [1.2, 3.3, 1.4],
  [5.1, 2.2, 6.6]])

import xgboost as xgb
dmat1 = xgb.DMatrix(data)

labels = np.array([0, 1])
dmat2 = xgb.DMatrix(data, label=labels)
```

The `DMatrix` object can be used for training and using a `Booster` object, which represents the gradient boosted decision tree. The `train` function in XGBoost lets us train a `Booster` with a specified set of parameters.

The code below trains a `Booster` object using a predefined labeled dataset.

```
# predefined data and labels
print('Data shape: {}'.format(data.shape))
print('Labels shape: {}'.format(labels.shape))
dtrain = xgb.DMatrix(data, label=labels)

# training parameters
```

```
params = {
  'max_depth': 0,
  'objective': 'binary:logistic'

}
print('Start training')
bst = xgb.train(params, dtrain)  # booster
print('Finish training')
```

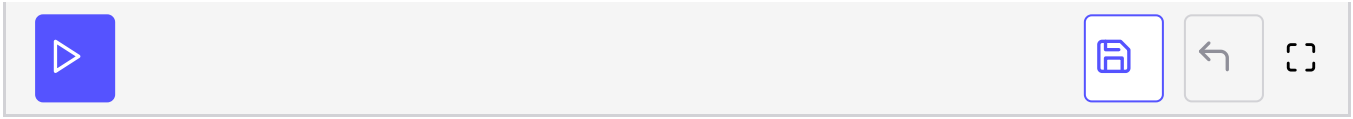


A list of the possible parameters and their values can be found here. In the example above, we set the `'max_depth'` parameter to `0` (which means no limit on the tree depths, equivalent to `None` in scikit-learn). We also set the `'objective'` parameter (the objective function) to binary classification via logistic regression. For the remaining available parameters, we used their default settings (so we didn't include them in `params`).

B. Using a `Booster`

After training a `Booster`, we can evaluate it and use it to make predictions.

```
# predefined evaluation data and labels
print('Data shape: {}'.format(eval_data.shape))
print('Labels shape: {}'.format(eval_labels.shape))
deval = xgb.DMatrix(eval_data, label=eval_labels)

# Trained bst from previous code
print(bst.eval(deval))  # evaluation

# new_data contains 2 new data observations
dpred = xgb.DMatrix(new_data)
# predictions represents probabilities
predictions = bst.predict(dpred)
print('{}\n'.format(predictions))
```



The evaluation metric used for binary classification (`eval-error`) represents the classification error, which is the default `'eval_metric'` parameter for binary classification `Booster` models.

Note that the model's predictions (from the `predict` function) are probabilities, rather than class labels. The actual label classifications are just the rounded probabilities. In the example above, the `Booster` predicts classes of 0 and 1, respectively.

Time to Code!

The coding exercise for this chapter will be to train a `Booster` object on input `data` and `labels` (predefined in the backend).

The first thing to do is set up a `DMatrix` for training.

Set `dtrain` equal to `xgb.DMatrix` initialized with `data` as the required argument and `labels` as the `label` keyword argument.

```
# CODE HERE
```

The input dataset contains 3 classes, so we'll perform multiclass classification with the `Booster`. The dataset is also relatively small, so we limit the decision tree's maximum depth to 2.

This means that the parameters for the `Booster` object will have `'max_depth'` set to `2`, `'objective'` set to `'multi:softmax'`, and `'num_class'` set to `3`.

Set `params` equal to a dictionary with the specified keys and values.

```
# CODE HERE
```

Using the data matrix and parameters, we'll train the `Booster`.

Set `bst` equal to `xgb.train` applied with `params` and `dtrain` as the required arguments.

```
# CODE HERE
```

Cross-Validation

Use cross-validation to evaluate parameters for XGBoost.

Chapter Goals:

- Learn how to cross-validate parameters in XGBoost

A. Choosing parameters

Since there are many parameters in XGBoost and several possible values for each parameter, it is usually necessary to *tune* the parameters. In other words, we want to try out different parameter settings and see which one gives us the best results.

We can tune the parameters using cross-validation (for a detailed explanation of cross-validation, see the **Data Modeling** section). In XGBoost, the `cv` function performs cross-validation for a set of parameters on a given training dataset.

The code below demonstrates cross-validation in XGBoost.

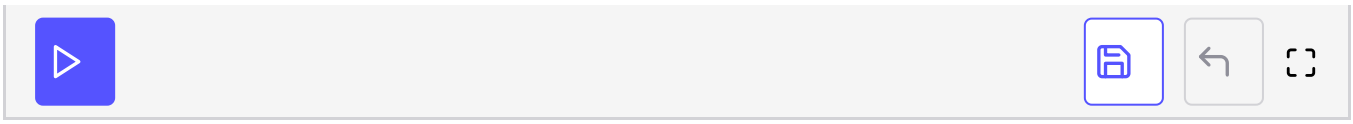
```
# predefined data and labels
dtrain = xgb.DMatrix(data, label=labels)
params = {
  'max_depth': 2,
  'lambda': 1.5,
  'objective':'binary:logistic'
}
cv_results = xgb.cv(params, dtrain)
print('CV Results:\n{}'.format(cv_results))
```

The output of `cv` is a pandas DataFrame (see the **Data Processing** section for details). It contains the training and testing results (mean and standard deviation) of a K-fold cross-validation applied for a given number of boosting iterations. The value of K for the K-fold cross-validation is set with the `nfold`

keyword argument (default is 3).

The keyword argument `num_boost_round` specifies the number of boosting iterations. Each boosting iteration will try to improve the model through gradient boosting. The default number of iterations is 10.

```
# predefined data and labels
dtrain = xgb.DMatrix(data, label=labels)
params = {
  'max_depth': 2,
  'lambda': 1.5,
  'objective':'binary:logistic'
}
cv_results = xgb.cv(params, dtrain, num_boost_round=5)
print('CV Results:\n{}'.format(cv_results))
```



Storing Boosters

Save and load Booster objects using XGBoost binary files.

Chapter Goals:

- Learn how to save and load `Booster` models in XGBoost

A. Saving and loading binary data

After finding the best parameters for a `Booster` and training it on a dataset, we can save the model into a binary file. Each `Booster` contains a function called `save_model`, which saves the model's binary data into an input file.

The code below saves a trained `Booster` object, `bst`, into a binary file called *model.bin*.

```
# predefined data and labels
dtrain = xgb.DMatrix(data, label=labels)
params = {
  'max_depth': 3,
  'objective':'binary:logistic'
}
bst = xgb.train(params, dtrain)

# 2 new data observations
dpred = xgb.DMatrix(new_data)
print('Probabilities:\n{}'.format(
  repr(bst.predict(dpred))))

bst.save_model('model.bin')
```

We can restore a `Booster` from a binary file using the `load_model` function. This requires us to initialize an empty `Booster` and load the file's data into it.

The code below loads the previously saved `Booster` from *model.bin*.

```
# Load saved Booster
new_bst = xgb.Booster()
```

```
new_bst = xgb.Booster()
new_bst.load_model('model.bin')

# Same dpred from before
print('Probabilities:\n{}'.format(
  repr(new_bst.predict(dpred))))
```

XGBoost Classifier

Create an XGBoost classifier object.

Chapter Goals:

- Learn how to create a scikit-learn style classifier in XGBoost

A. Following the scikit-learn API

While XGBoost provides a more efficient model than scikit-learn, using the model can be a bit convoluted. For people who are used to scikit-learn, XGBoost provides wrapper APIs around its model for classification and regression. These wrapper APIs allow us to use XGBoost's efficient model in the same style as scikit-learn.

For classification, the XGBoost wrapper model is called `XGBClassifier`. Like regular scikit-learn models, it can be trained with a simple call to `fit` with NumPy arrays as input arguments.

```
model = xgb.XGBClassifier()
# predefined data and labels
model.fit(data, labels)

# new_data contains 2 new data observations
predictions = model.predict(new_data)
print('Predictions:\n{}'.format(repr(predictions)))
```

Note that the `predict` function for `XGBClassifier` returns actual predictions (not probabilities).

All the parameters for the original `Booster` object are now keyword arguments for the `XGBClassifier`. For instance, we can specify the type of classification, i.e. the `'objective'` parameter for `Booster` objects, with the `objective` keyword argument (the default is binary classification).

```
model = xgb.XGBClassifier(objective='multi:softmax')
# predefined data and labels (multiclass dataset)
model.fit(data, labels)

# new_data contains 2 new data observations
predictions = model.predict(new_data)
print('Predictions:\n{}'.format(repr(predictions)))
```

XGBoost Regressor

Create an XGBoost regressor object.

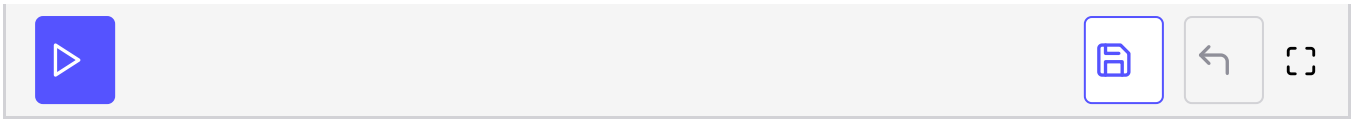
Chapter Goals:

- Learn how to create a scikit-learn style regression model in XGBoost

A. XGBoost linear regression

In addition to scikit-learn style classifiers, XGBoost also provides a scikit-learn style linear regression model with the `XGBRegressor` object.

```
model = xgb.XGBRegressor(max_depth=2)
# predefined data and labels (for regression)
model.fit(data, labels)

# new_data contains 2 new data observations
predictions = model.predict(new_data)
print('Predictions:\n{}'.format(repr(predictions)))
```



Just like the `XGBClassifier` object, we can specify the model's parameters with keyword arguments. In the code above, we set the `max_depth` parameter (representing the depth of the boosted decision tree) to 2.

Feature Importance

Learn about feature importance in making model predictions.

Chapter Goals:

- Understand how to measure each dataset feature's importance in making model predictions
- Use the matplotlib pyplot API to save a feature importance plot to a file

A. Determining important features

Not every feature in a dataset is used equally for helping a boosted decision tree make predictions. Certain features are more important than others, and it is useful to figure out which features are the most important.

After training an XGBoost model, we can view the relative (proportional) importance of each dataset feature using the `feature_importances_` property of the model.

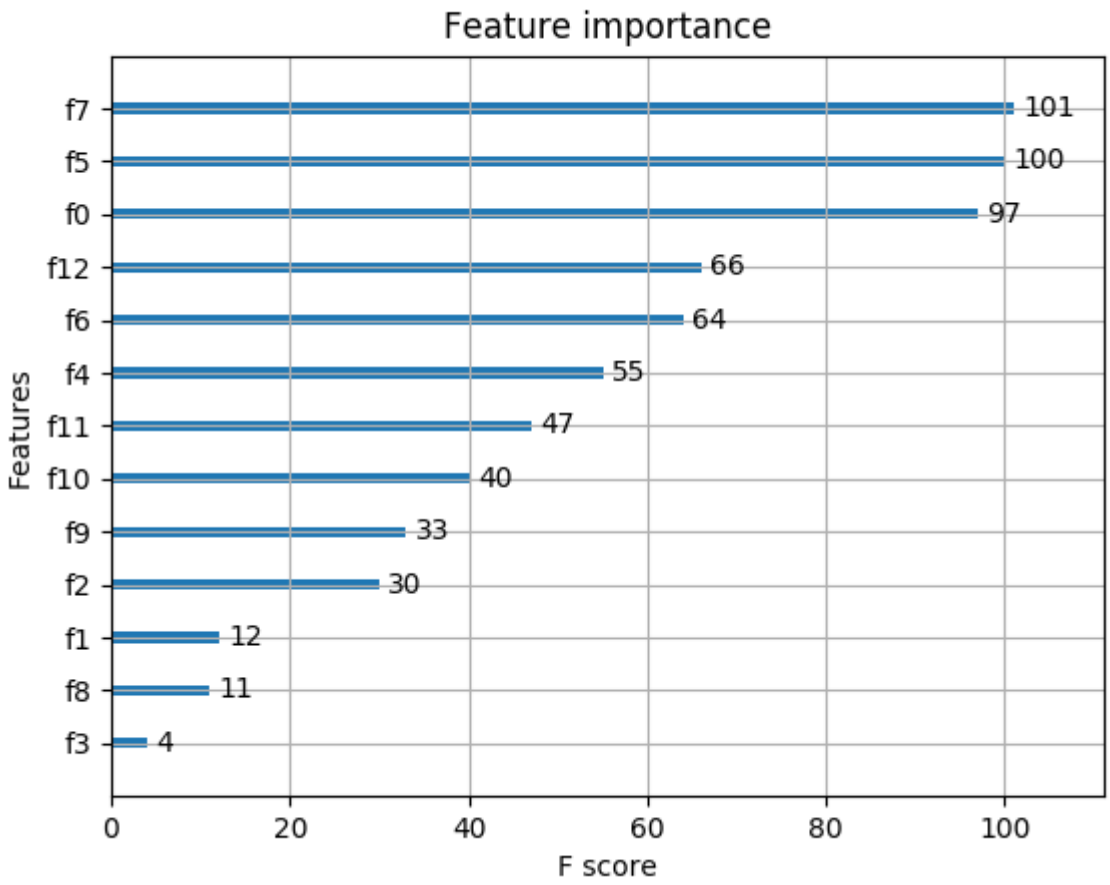
The code below prints out the relative feature importances of a model trained on a dataset of 4 features.

```
model = xgb.XGBClassifier()
# predefined data and labels
model.fit(data, labels)

# Array of feature importances
print('Feature importances:\n{}'.format(
  repr(model.feature_importances_)))
```

B. Plotting important features

We can plot the feature importances for a model using the `plot_importance` function.

```
model = xgb.XGBRegressor()
# predefined data and labels (for regression)
model.fit(data, labels)

xgb.plot_importance(model)
plt.show() # matplotlib plot
```



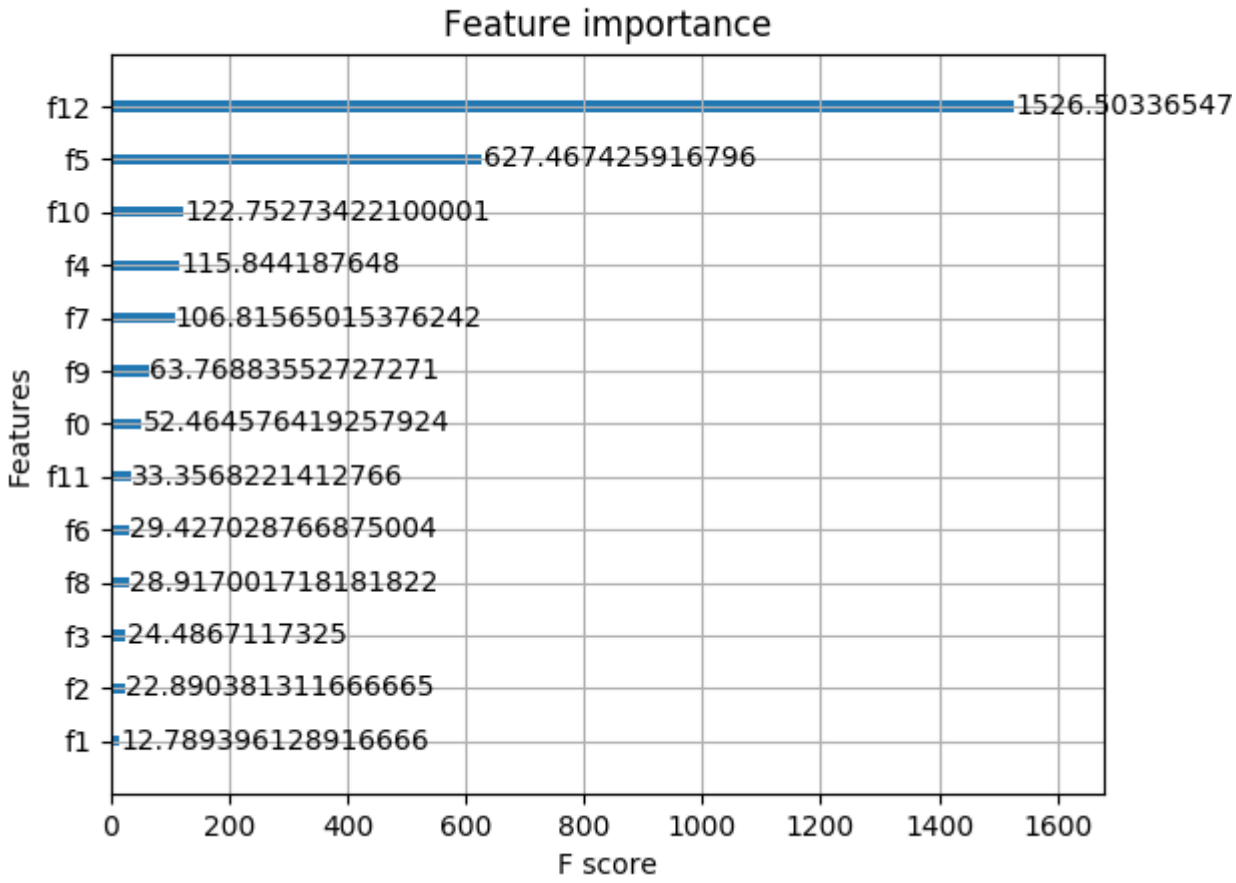
Plotting the feature importances and showing the plot using matplotlib.pyplot (plt).

The resulting plot is a bar graph of the F-scores (F_1-scores) for each feature (the number next to each bar is the exact F-score). Note that the features are labeled as "f*N*", where *N* is the index of the column in the dataset. The F-score is a standardized measurement of a feature's importance, based on the specified importance metric.

By default, the `plot_importance` function uses feature *weight* as the importance metric, i.e. how often the feature appears in the boosted decision tree. We can manually choose a different importance metric with the `importance_type` keyword argument.

```
model = xgb.XGBRegressor()
```

```
# predefined data and labels (for regression)
model.fit(data, labels)


xgb.plot_importance(model, importance_type='gain')
plt.show() # matplotlib plot
```



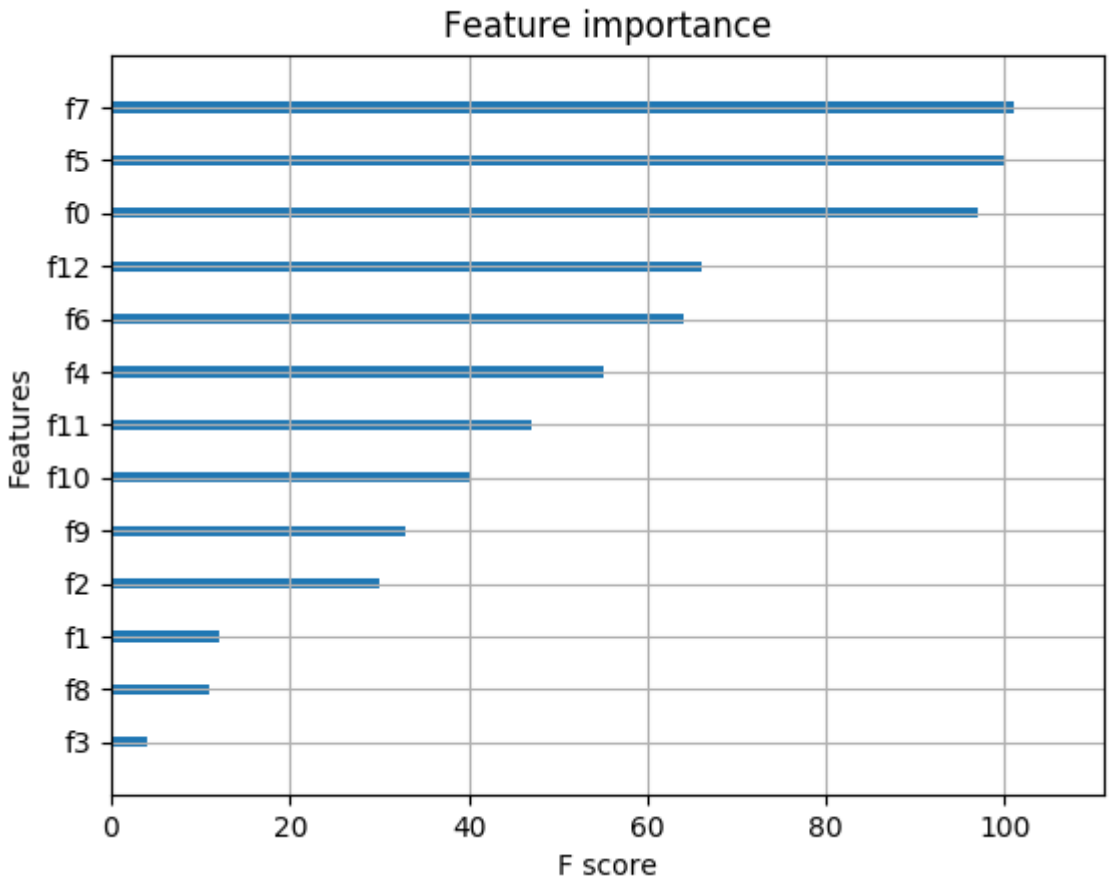
Plotting the feature importances with information gain as the importance metric

In the code above, we set `importance_type` equal to `'gain'`, which means that we use information gain as the importance metric. Information gain is a commonly used metric for determining how good a feature is at differentiating the dataset, which is important in making predictions with a decision tree.

Finally, if we don't want to show the exact F-score next to each bar, we can set the `show_values` keyword argument to `False`.

```
model = xgb.XGBRegressor()
# predefined data and labels (for regression)
model.fit(data, labels)

xgb.plot_importance(model, show_values=False)
```

```
plt.savefig('importances.png') # save to PNG file
```



Plotting the feature importances without the exact F-score.

Hyperparameter Tuning

Apply grid search cross-validation to XGBoost models.

Chapter Goals:

- Apply grid search cross-validation to an XGBoost model

A. Using scikit-learn's `GridSearchCV`

One of the benefits of using XGBoost's scikit-learn style models is that we can use the models with the actual scikit-learn API. A common scikit-learn object used with XGBoost models is the `GridSearchCV` wrapper. For more on `GridSearchCV` see the **Data Modeling** section.

The code below applies grid search cross-validation to a binary classification XGBoost model.

```
model = xgb.XGBClassifier()
params = {'max_depth': range(2, 5)}

from sklearn.model_selection import GridSearchCV
cv_model = GridSearchCV(model, params, cv=4, iid=False)

# predefined data and labels
cv_model.fit(data, labels)
print('Best max_depth: {}\n'.format(
  cv_model.best_params_['max_depth']))

# new_data contains 2 new data observations
print('Predictions:\n{}'.format(
  repr(cv_model.predict(new_data))))
```

In the code above, we applied grid search cross-validation to a binary classification XGBoost model to find the optimal `'max_depth'` parameter (in the range from 2 to 4, inclusive). The *K*-fold cross-validation (the default for grid search) uses 4 folds. Note that the cross-validation process works the same for an `XGBRegressor` object.

After calling `fit` on `data` and `labels`, `cv_model` represents the cross-validated classification model trained on the dataset. The grid search cross-validation automatically chose the best performing `'max_depth'` parameter, which in this case was 4. The `best_params_` attribute contains the best performing hyperparameters after cross-validation.

The official XGBoost documentation provides a list of the possible parameters we can tune for in a model. A couple commonly tuned parameters are `'max_depth'` and `'eta'` (the learning rate of the boosting algorithm).

Model Persistence

Save and load XGBoost models using joblib.

Chapter Goals:

- Save and load XGBoost models with the joblib API

A. The joblib API

Since the `XGBClassifier` and `XGBRegressor` models follow the same format as scikit-learn models, we can save and load them in the same way. This means using the joblib API.

With the joblib API, we save and load models using the `dump` and `load` functions, respectively. See here for specific examples on model persistence with scikit-learn models.

Quiz

1 What type of model does XGBoost use?

COMPLETED 0%

1 of 4

Introduction

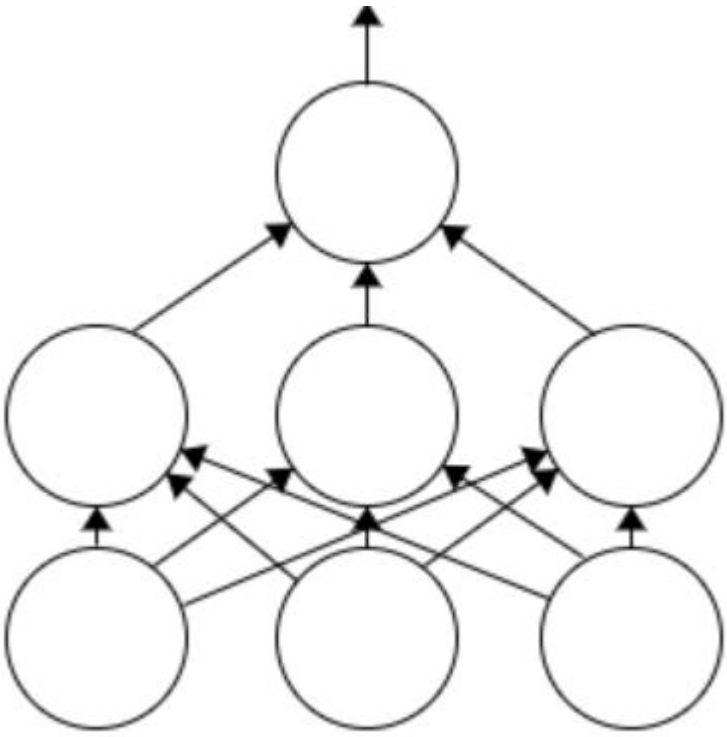
An overview of the multilayer perceptron neural network and deep learning in TensorFlow.

In the **Intro to Deep Learning** section, you will learn about one of the most essential neural networks used in deep learning, the multilayer perceptron.

A. Multilayer perceptron

The multilayer perceptron (MLP) is used for a variety of tasks, such as stock analysis, spam detection, and election voting predictions. In the following chapters you will learn how to code your own MLP and apply it to the task of classifying 2-D points in the Cartesian plane.

You will also learn the basics of creating a *computation graph*, i.e. the structure of a neural network. The structure of a neural network can be viewed in layers of *neurons*:



Multilayer perceptron

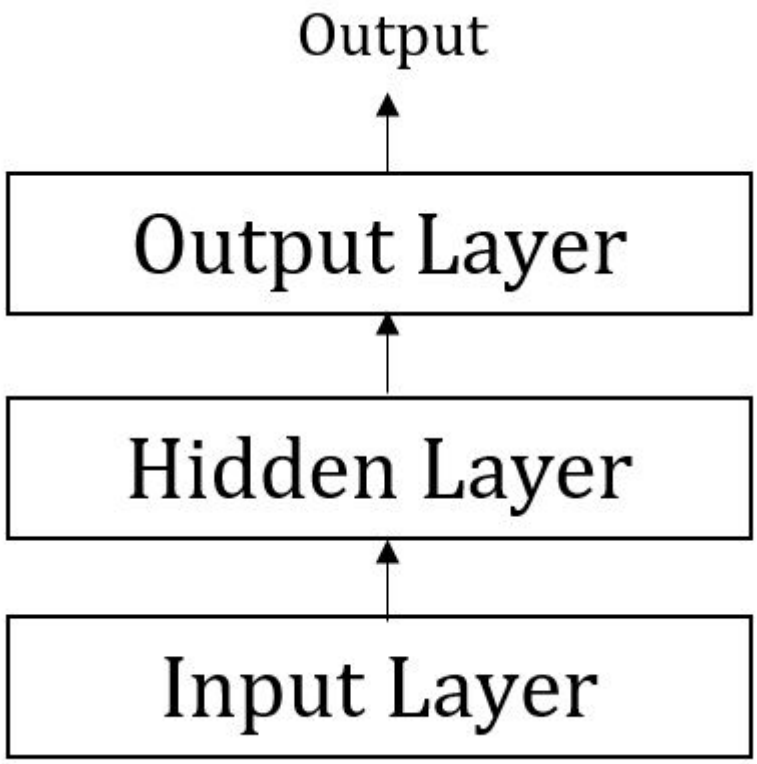
In the diagram, the circles represent neurons and the *connections* are the

arrows going from neurons in one layer to the next. There are three layers in this diagram's neural network:

- *Input layer*: The first (bottom) layer.
- *Output layer*: The last (top) layer.
- *Hidden layer(s)*: The layer(s) between the input and output layers. In the diagram, there is 1.

The number of hidden layers represents how "deep" a model is, and you'll see the power of adding hidden layers to a model.

When diagramming the computation graph, it is common to eschew the neurons and connections:



This makes it easier to view computation graphs of complex neural networks with dozens of layers.

B. TensorFlow

To code our neural network model, we will be using TensorFlow, one of the most popular deep learning frameworks. The name for TensorFlow is derived from *tensors*, which are basically multidimensional (i.e. generalized) vectors/matrices. When writing the code, it may be easier to think of anything with numeric values as being a tensor.

In TensorFlow, we first create the computation graph structure (which we will

do in chapters 2-5), and then train and evaluate the model with input data and labels.

Something quick to note is that TensorFlow is normally imported via the statement `import tensorflow as tf`. This provides a shorthand way to call the module.

Model Initialization

Learn about the input and output layers of a neural network.

Chapter Goals:

- Define a class for an MLP model
- Initialize the model

A. Placeholder

When building the computation graph of our model, `tf.placeholder` acts as a "placeholder" for the input data and labels. Without the `tf.placeholder`, we would not be able to train our model on real input data.

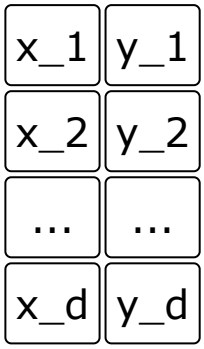
A `tf.placeholder` takes in a required first argument and two keyword arguments called `shape` and `name`.

The required argument for the placeholder is its type. In **Deep Learning with TensorFlow**, our input data is pairs of (x, y) points, so `self.inputs` has type `tf.float32`. The labels (which are explained in a later chapter) have type `tf.int32`.

The `name` keyword argument allows us to give a custom name to our placeholder, making it easier for debugging.

B. Shapes and dimensions

The `shape` argument is a tuple of integers representing the size of each of the placeholder tensor's dimensions. In **Deep Learning with TensorFlow**, and many real world problems, the shape of the input data will be a two integer tuple. If we view the input data as coming from a data table, the `shape` is akin to the dimensions of the table.



The shape of this data is (d, 2). There are d data points, each of dimension 2.

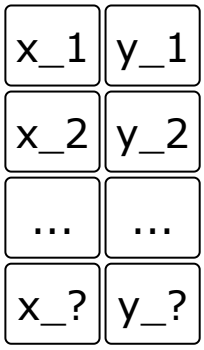
The first integer represents the number of data points we pass in (i.e. number of rows in the data table). When training the model, we refer to the first dimension as the *batch size*.

The second integer represents the number of features in the dataset (i.e. number of columns). In the remaining chapters of **Deep Learning with TensorFlow**, our input data will have two features: the *x* and *y* coordinates of the data point. So `input_size` is 2.

Each data point also has a label, which is used to identify and categorize the data. The labels we use for our model's data will have a two dimension shape, with `output_size` as the second dimension (the first dimension is still the batch size). The `output_size` refers to the number of possible *classes* a label can have (explained in a later chapter).

C. Different amounts of data

One thing to note about TensorFlow is the use of `None` in place of a dimension size. When we use `None` in the `shape` tuple, we are allowing that dimension to take on any size.



The shape of this data is (None, 2). There can be variable number of data points, each of dimension 2.

This is particularly useful because we will use our neural network on input data with different input sizes.

When we input multiple input data for our neural network, we don't actually use multiple neural networks. Rather, we use the same neural network on each of the points simultaneously to obtain the network's output for each point.

Time to Code!

The code for this chapter will focus on initializing the placeholders for the MLP model.

First you'll define the placeholder for the model's input data, in the function `init_inputs`. Note that TensorFlow is already imported in the backend via the import statement `import tensorflow as tf`.

Set `inputs` equal to a `tf.placeholder` with data type `tf.float32` and keyword arguments `shape=(None, input_size)` and `name='inputs'`.

Then return `inputs`.

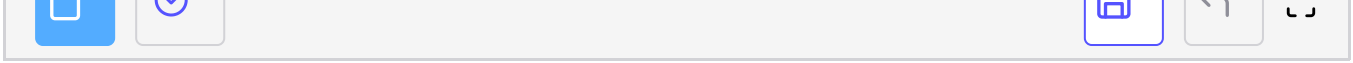
```
def init_inputs(input_size):
  # CODE HERE
  pass
```

Next, we'll define placeholders for the labels, in the function `init_labels`.

Set `labels` equal to a `tf.placeholder` with data type `tf.int32` and keyword arguments `shape=(None, output_size)` and `name='labels'`.

Then return `labels`.

```
def init_labels(output_size):
  # CODE HERE
  pass
```



Logits

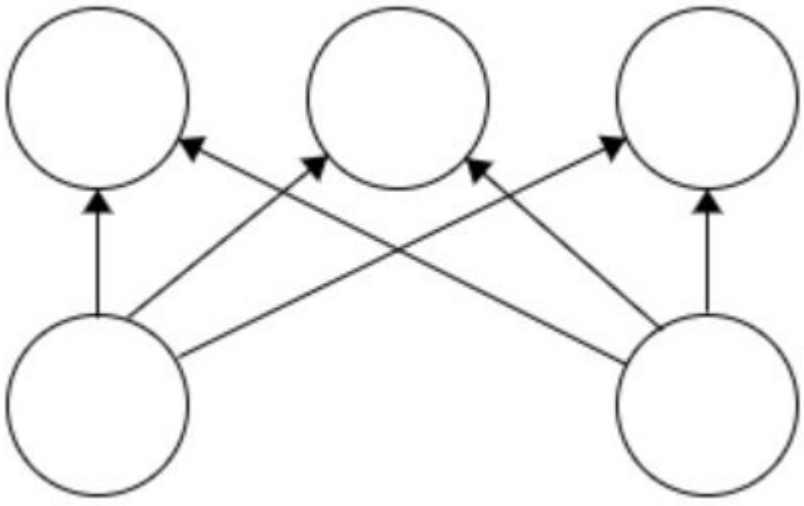
Dive into the inner layers of a neural network and understand the importance of logits.

Chapter Goals:

- Build a single fully-connected model layer
- Output logits, AKA log-odds

A. Fully-connected layer

Before we can get into multilayer perceptrons, we need to start off with a single layer perceptron. The single fully-connected layer means that the input layer, i.e. `self.inputs`, is directly connected to the output layer, which has `output_size` neurons. Each of the `input_size` neurons in the input layer has a connection to each neuron in the output layer, hence the fully-connected layer.

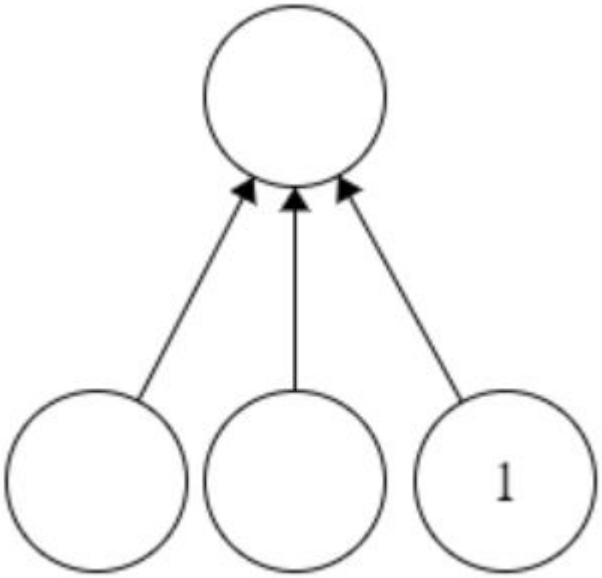


Fully-connected layer with input_size = 2, output_size = 3

In TensorFlow, this type of fully-connected neuron layer is implemented using `tf.layers.dense`, which takes in a neuron layer and output size as required arguments, and adds a fully-connected output layer with the given size to the computation graph.

In addition to the input layer neurons, `tf.layers.dense` adds another neuron

called the *bias*, which always has a value of 1 and has full connections to the output layer.

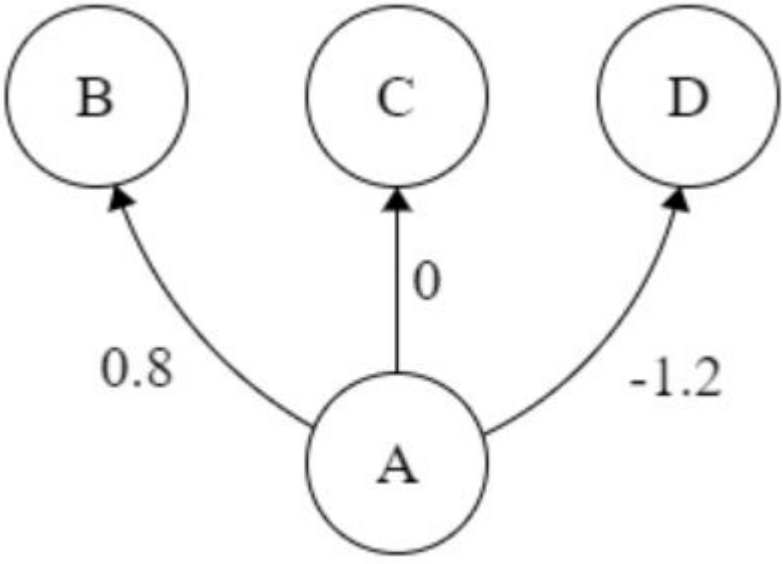


Fully-connected layer with input_size = 2, output_size = 1, and a bias neuron.

The bias neuron helps our neural network produce better results, by allowing each fully-connected layer to model a true linear combination of the input values.

B. Weighted connections

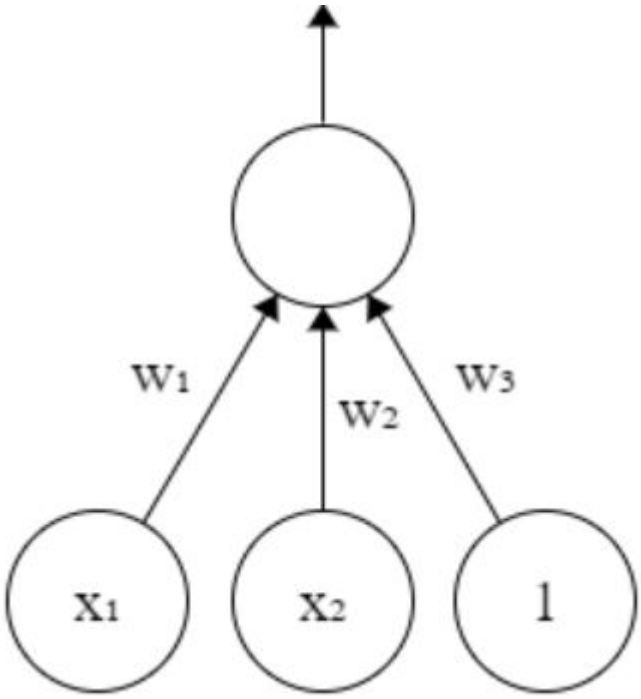
The forces that drive a neural network are the real number weights attached to each connection. The weight on a connection from neuron **A** into neuron **B** tells how strongly **A** affects **B** as well as whether that effect is positive or negative, i.e. direct vs. inverse relationship.



The diagram above has three weighted connections:

$\boldsymbol{A} \rightarrow \boldsymbol{B}$: Direct relationship.
$\boldsymbol{A} \rightarrow \boldsymbol{C}$: No relationship.
$\boldsymbol{A} \rightarrow \boldsymbol{D}$: Inverse relationship.

The weighted connections allow fully-connected layers to model a linear combination of the inputs. Let's take a look at an example fully-connected layer with weights:



Fully-connected layer with input neuron values x1 and x2, a bias neuron, and weight values w1, w2, and w3

The output of this fully-connected layer is now a linear combination of the input neuron values:

$$w_1 \cdot x_1 + w_2 \cdot x_2 + w_3$$

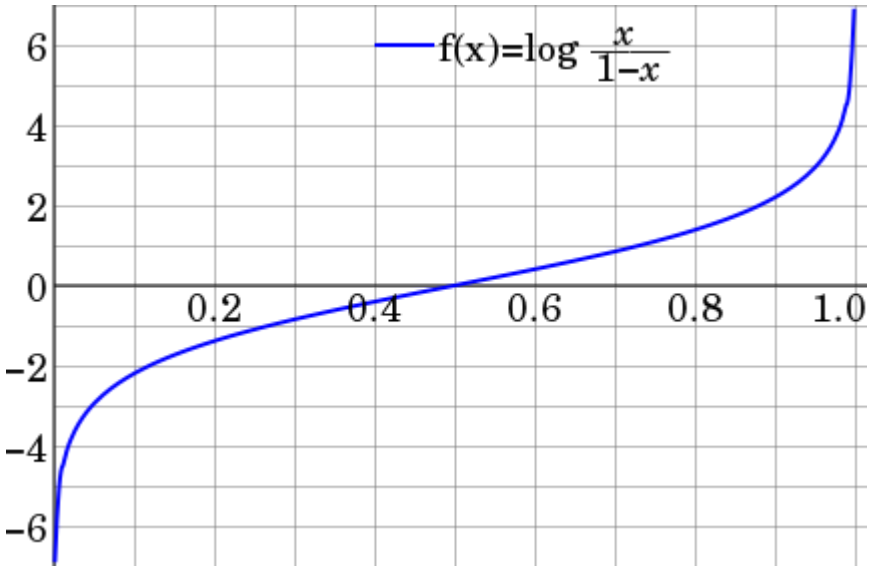
The logits produced by our single layer perceptron are therefore just a linear combination of the input data feature values.

Connection weights can be optimized through training (which we will cover in a later chapter), so that the logits produced by the neural network allow the model to make highly accurate predictions based on the input data.

C. Logits

So what exactly are logits? In classification problems they represent log-odds, which maps a probability between 0 and 1 to a real number. When

`output_size = 1`, our model outputs a single logit per data point. The logits will then be converted to probabilities representing how likely it is for the data point to be labeled 1 (as opposed to 0).



In the above diagram, the *x*-axis represents the probability and the *y*-axis represents the logit.
Note the vertical asymptotes at *x = 0* and *x = 1*.

We want our neural network to produce logits with large absolute values, since those represent probabilities close to 0 (meaning we are very sure the label is 0/False) or probabilities close to 1 (meaning we are very sure the label is 1/True).

D. Regression

In the next chapter, you'll be producing actual probabilities from the logits. This makes our single layer perceptron model equivalent to logistic regression. Despite the name, logistic regression is used for classification, not regression problems. If we wanted a model for regression problems (i.e. predicting a real number such as a stock price), we would have our model directly output the logits rather than convert them to probabilities. In this case, it would be better to rename `logits`, since they don't map to a probability anymore.

Time to Code!

The code for this chapter will build a single layer perceptron, whose output is

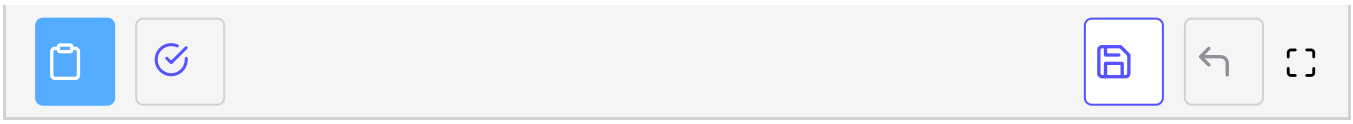
`logits`. The code goes inside the `model_layers` function.

We're going to obtain the `logits` by applying a dense layer to `inputs` (the placeholder from Chapter 2) to return a tensor with shape `(None, self.output_size)`.

Set `logits` equal to `tf.layers.dense` with required arguments `inputs` and `output_size`, and keyword argument `name='logits'`.

Then return `logits`.

```
def model_layers(inputs, output_size):
  # CODE HERE
  pass
```



Metrics

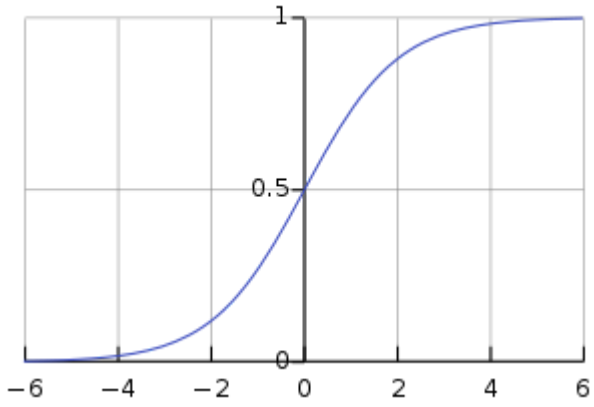
Discover the most commonly used metrics for evaluating a neural network.

Chapter Goals:

- Convert logits to probabilities
- Obtain model predictions from probabilities
- Calculate prediction accuracy

A. Sigmoid

As discussed in the previous chapter, our model outputs logits based on the input data. These logits represent real number mappings from probabilities. Therefore, if we had the inverse mapping we could obtain the original probabilities. Luckily, we have exactly that, in the form of the sigmoid function.



The above plot shows a sigmoid function graph. The *x*-axis represents logits, while the *y*-axis represents probabilities.

> **Note** the horizontal asymptotes at $y = 0$ and $y = 1$.

Using the sigmoid function (`tf.nn.sigmoid` in TensorFlow), we can extract probabilities from the logits. In binary classification, i.e. `output_size = 1`, this

represents the probability the model gives to the input data being labeled 1. A probability closer to 0 or 1 means the model is pretty sure in its prediction,

while a probability closer to 0.5 means the model is unsure (0.5 is equivalent to a random guess of the label's value).

B. Predictions and accuracy

A probability closer to 1 means the model is more sure that the label is 1, while a probability closer to 0 means the model is more sure that the label is 0. Therefore, we can obtain model predictions just by rounding each probability to the nearest integer, which would be 0 or 1. Then our prediction accuracy would be the number of correct predictions divided by the number of labels.

In TensorFlow, there is a function called `tf.reduce_mean` which produces the overall mean of a tensor's numeric values. We can use this to calculate prediction accuracy as the mean number of correct predictions across all input data points.

We use these metrics to evaluate how good our model is both during and after training. This way we can experiment with different computation graphs, such as different numbers of neurons or layers, and find which structure works best for our model.

Time to Code!

The coding exercise for this chapter focuses on obtaining predictions and accuracy based on model logits.

In the backend, we've loaded the `logits` tensor from the previous chapter's `model_layers` function. We'll now obtain probabilities from the `logits` using the sigmoid function.

Set `probs` equal to `tf.nn.sigmoid` applied to `logits`.

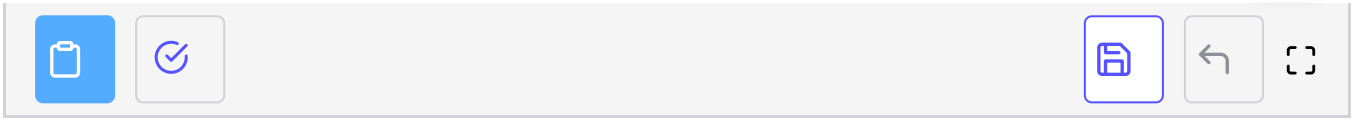
```
# CODE HERE
```

We can calculate label predictions by rounding each probability to the nearest

integer (0 or 1). We'll use `tf.round` to first round the probabilities to 0.0 or 1.0.

Set `rounded_probs` equal to `tf.round` with `probs` as the lone argument.

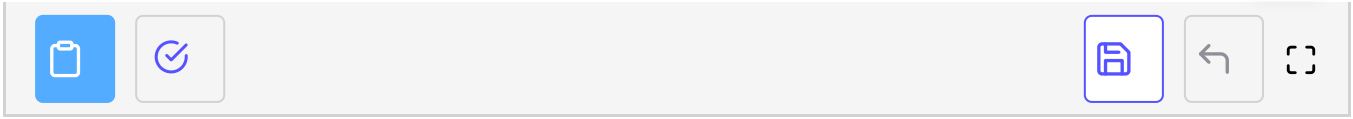
```
# CODE HERE
```



The problem with `rounded_probs` is that it's still type `tf.float32`, which doesn't match the type of the `labels` placeholder. This is an issue, since we need the types to match to compare the two. We can fix this problem using `tf.cast`.

Set `predictions` equal to `tf.cast` with `rounded_probs` as the first argument and data type `tf.int32` as the second argument.

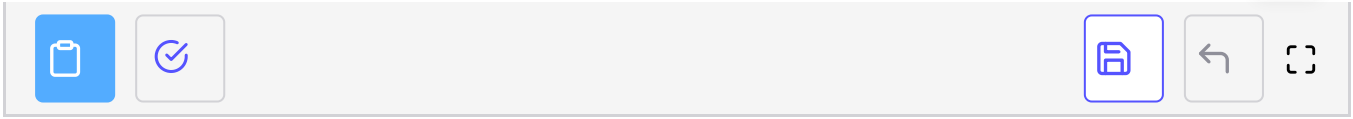
```
# CODE HERE
```



The final metric we want is the accuracy of our predictions. We'll directly compare `predictions` to `labels` by using `tf.equal`, which returns a tensor that has `True` at each position where our prediction matches the label and `False` elsewhere. Let's call this tensor `is_correct`.

Set `is_correct` equal to `tf.equal` with `predictions` as the first argument and `labels` as the second argument.

```
# CODE HERE
```

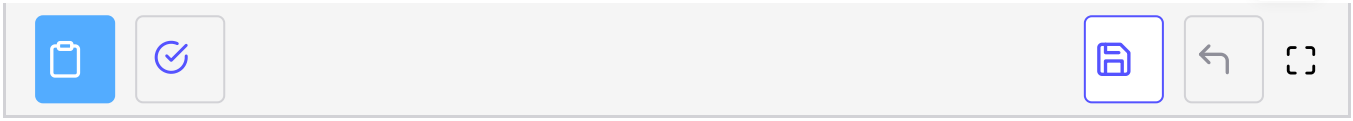


The neat thing about `is_correct` is that the number of `True` values divided by the number of total values in the tensor gives us our accuracy. We can use `tf.reduce_mean` to do this calculation. We just need to cast `is_correct` to type `tf.float32`, which converts `True` to 1.0 and `False` to 0.0.

Set `is_correct_float` equal to `tf.cast` with `is_correct` as the first argument and data type `tf.float32` as the second argument.

Set `accuracy` equal to `tf.reduce_mean` applied to `is_correct_float`.

```
# CODE HERE
```



Optimization

Learn about loss functions and optimizing neural network weights.

Chapter Goals:

- Know the relationship between training, weights, and loss
- Understand the intuitive definition of loss
- Obtain the model's loss from logits
- Write a training operation to minimize the loss

A. What is training?

In Chapter 3, we discussed the weights associated with connections between neurons. These weights determine what a neural network outputs based on the input data. However, these weights are what we call *trainable variables*, meaning that we need to train our neural network to find the optimal weights for each connection.

For any neural network, training involves setting up a *loss function*. The loss function tells us how bad the neural network's output is compared to the actual labels.

Since a larger loss means a worse model, we want to train the model to output values that minimize the loss function. The model does this by *learning* the optimal weight settings. Remember, the weights are just real numbers, so the model is essentially just figuring out the best numbers to set the weights to.

B. Loss as error

In regression problems, common loss functions are the L1 norm:

$$\sum_i |actual_i - predicted_i|$$

and the L2 norm:

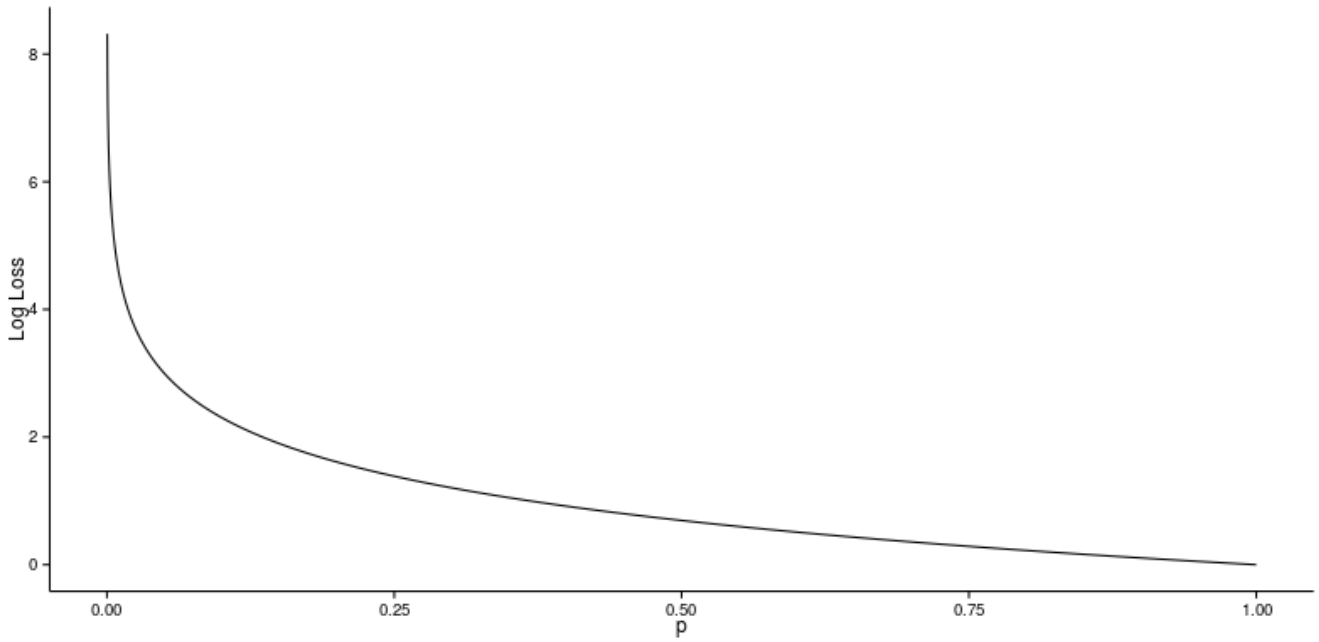
$$\sum_i (actual_i - predicted_i)^2$$

These provide an error metric for how far the predictions are from the labels, so the goal is to minimize the prediction error by minimizing the L1 and L2 norm.

In classification problems there's no good error measurement between predictions and labels, since the labels are discrete values. For example, in regression if we predict a stock's price was $99 but the actual value was $100, our prediction is still really good even though it was incorrect. However, in classification a prediction is either right or wrong, without any sense of how close it is to the actual label.

C. Cross entropy

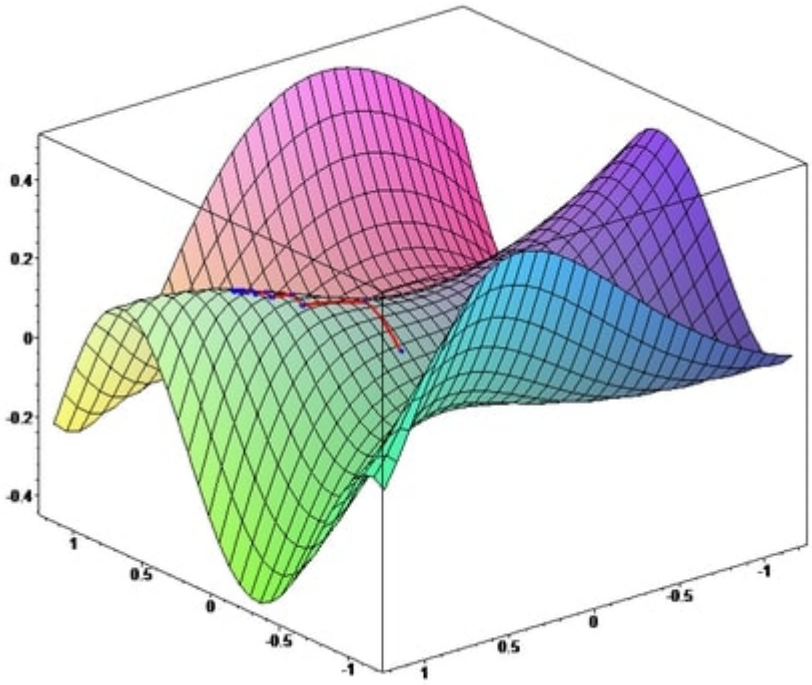
Rather than defining error as being right or wrong in our prediction, we can instead define it in terms of probability. Therefore, we want a loss function that is small when the probability is close to the label (i.e. a probability of 0.99 for a label of 1) and large when the probability is far from the label (i.e. a probability of 0.99 for a label of 0). The loss function that achieves this is known as cross entropy, also referred to as *log loss*.



Cross entropy (log loss) for a label of 1. Thex-axis represents the probability and the y-axis represents the log loss.

D. Optimization

Now we can just minimize the cross entropy based on the model's logits and labels to get our optimal weights. We do this through gradient descent, where the model updates its weights based on a *gradient* of the loss function until it reaches the minimum loss (at which point the weights converge to the optimum). We use backpropagation to find the optimal gradient for the model to follow. Gradient descent is implemented as an object in TensorFlow, called `tf.train.GradientDescentOptimizer`.



In the above graph, the colored shape represents values of the loss function, and the *x* and *y* axes represent weight values in the model. The model follows a gradient (red line) towards the minimum of the loss function.

The size of the gradient depends on something called the *learning rate*. A larger learning rate means the model could potentially reach the minimum loss quicker, but could also overshoot the minimum. Smaller learning rates are more likely to reach the minimum, but may take longer. Usually we test out learning rates between 0.001 to 0.1 to find the best one for model training. You can set the learning rate via the `learning_rate` argument when initializing a TensorFlow `Optimizer` (e.g. `GradientDescentOptimizer`).

Regular gradient descent has trouble minimizing complex loss functions, so we usually use better optimization methods for training. A popular and effective optimization method is Adam, which is implemented in TensorFlow

as `tf.train.AdamOptimizer`. It has default values already set for its parameters (e.g. `learning_rate`), so in our code we initialize the object with no arguments.

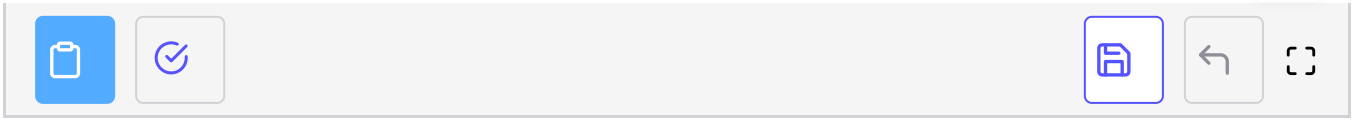
Time to Code!

The coding exercises for this chapter build on top of the code from the previous chapter. Specifically, the optimization code for this chapter is only needed for model training (not evaluation or testing).

We'll first set up the loss parameter, a variable named `loss`. Since `loss` will be a floating-point number, we'll need to cast `labels` to type `tf.float32`.

Set `labels_float` equal to `tf.cast` with `labels` as the first argument and data type `tf.float32` as the second argument.

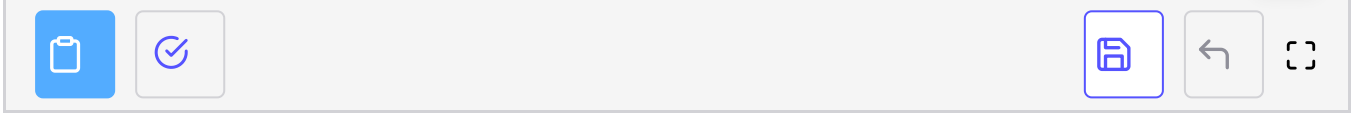
```
# CODE HERE
```



We'll now calculate the sigmoid-based cross entropy between `labels_float` and `logits`.

Set `cross_entropy` equal to `tf.nn.sigmoid_cross_entropy_with_logits` with keyword arguments `labels=labels_float` and `logits=logits`.

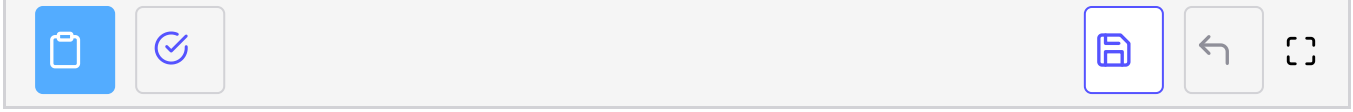
```
# CODE HERE
```



Since `cross_entropy` represents the sigmoid cross entropy for each input data label (so its shape is `(None, 1)`), we'll use the overall mean of the cross entropy as our loss.

Set `loss` equal to `tf.reduce_mean` applied to `cross_entropy`.

```
# CODE HERE
```



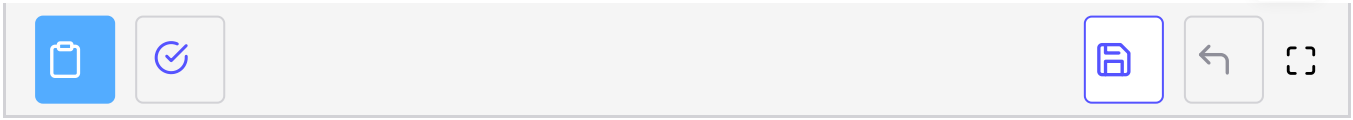
We'll now use the Adam optimization algorithm to set our training operation. First, we'll initialize an Adam optimizer object called `adam`.

Set `adam` equal to `tf.train.AdamOptimizer()`.

Then we'll use `adam` to minimize `loss`, which becomes our training operation (i.e. how we train the model's weights).

Set `train_op` equal to `adam.minimize` applied to `loss`.

```
# CODE HERE
```



Training

Initialize and train a TensorFlow neural network using actual training data.

Chapter Goals:

- Learn how to feed data values into a neural network
- Understand how to run a neural network using input values
- Train the neural network on batched input data and labels

A. Running the model

In this chapter and the next, you will be running your model on input data, using a `tf.Session` object and the `tf.placeholder` variables from the previous chapters.

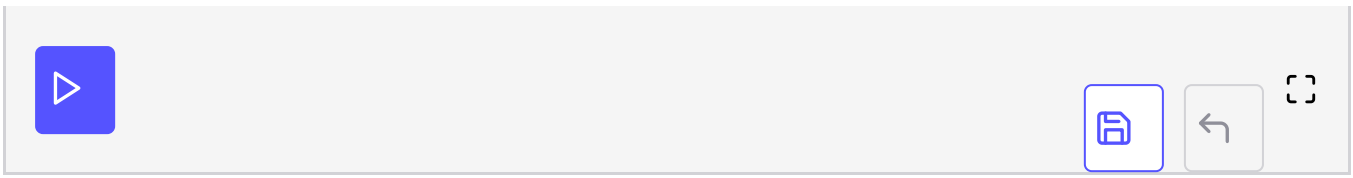
The `tf.Session` object has an extremely important function called `run`. All the code written in the previous chapters was to build the computation graph of the neural network, i.e. its layers and operations. However, we can only train or evaluate the model on real input data using `run`. The function takes in a single required argument and a few keyword arguments.

B. Using `run`

The required argument is normally either a single tensor/operation or a list/tuple of tensors and operations. Calling `run` on a tensor returns the value of that tensor after executing our computation graph. The output of `run` with a tensor input is a NumPy array.

The code below shows usages of `run` on `tf.constant` tensors.

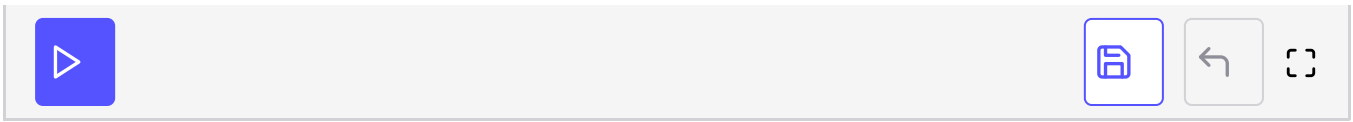
```
t = tf.constant([1, 2, 3])
sess = tf.Session()
arr = sess.run(t)
print('{}\n'.format(repr(arr)))

t2 = tf.constant(4)
tup = sess.run((t, t2))
print('{}\n'.format(repr(tup)))
```



Of the keyword arguments for `run`, the important one for most applications is `feed_dict`. The `feed_dict` is a python dictionary. Each key is a *tensor* from the model's computation graph. The key's value can be a Python scalar, list, or NumPy array.

We use `feed_dict` to pass values into certain tensors in the computation graph. In the code below, we pass in a value for `inputs`, which is a `tf.placeholder` object.

```
inputs = tf.placeholder(tf.float32, shape=(None, 2))
feed_dict = {
  inputs: [[1.1, -0.3],
           [0.2, 0.1]]
}
sess = tf.Session()
arr = sess.run(inputs, feed_dict=feed_dict)
print('{}\n'.format(repr(arr)))
```



Each `tf.placeholder` object used in the model execution must be included as a key in the `feed_dict`, with the corresponding value's shape and type matching the placeholder's.

C. Initializing variables

When we call `run`, every tensor in the model's computation graph must either already have a value or must be fed in a value through `feed_dict`. However, when we start training from scratch, none of our variables (e.g. weights) have values yet. We need to initialize all the variables using `tf.global_variables_initializer`. This returns an operation that, when used as the required argument in `run`, initializes all the variables in the model.

In the code below, the variables that are initialized are part of `tf.layers.dense`. The variable initialization process is defined internally by the function. In this case, the variables are initialized in a way that results in zero logits.

```
inputs = tf.placeholder(tf.float32, shape=(None, 2))
feed_dict = {
  inputs: [[1.1, -0.3],
           [0.2, 0.1]]
}
logits = tf.layers.dense(inputs, 1, name='logits')
init_op = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init_op) # variable initialization
arr = sess.run(logits, feed_dict=feed_dict)
print('{}\n'.format(repr(arr)))
```

D. Training logistics

The `num_steps` argument represents the number of iterations we use to train our model. Each iteration we train the model on a *batch* of data points. So `input_data` is essentially a large dataset divided into chunks (i.e. batches), and each iteration we train on a specific batch of points and their corresponding labels.

```
# predefined dataset
print('Input data:')
print('{}\n'.format(repr(input_data)))

print('Labels:')
print('{}\n'.format(repr(input_labels)))
```

Example dataset with a batch size of 3.

The batch size determines how the model trains. Larger batch sizes usually result in faster training but less accurate models, and vice-versa for smaller batch sizes. Choosing the batch size is a speed-precision tradeoff.

When training a neural network, it's usually a good idea to print out the loss every so often, so you know that the model is training correctly and to stop the training when the loss has converged to a minimum.

Time to Code!

The coding exercise for this chapter sets up the training utility, using the ADAM training operation and model layers from the previous chapters.

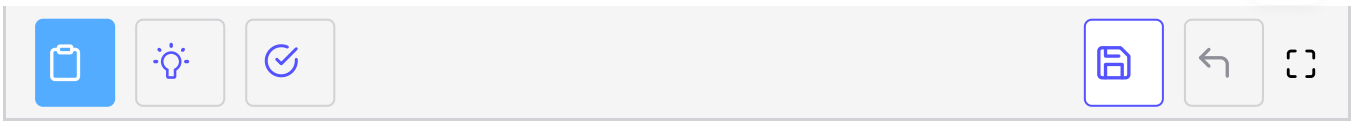
We first need to initialize all the model variables (e.g. the variables via `tf.layers.dense`).

To do this, we'll use `tf.Session` and `tf.global_variables_initializer`.

Set `init_op` equal to `tf.global_variables_initializer()`.

Create a `tf.Session` object named `sess`, and call its `run` function on `init_op`.

```
# CODE HERE
```



After initializing the variables, we can run our model training. We'll run the training for 1000 steps. We provide a `for` loop for you, which iterates 1000 steps.

The rest of the code for this chapter goes inside the `for` loop.

We provide the input dataset (`input_data`) and labels (`input_labels`) as NumPy arrays, initialized in the backend. The placeholders for the data and labels, `inputs` and `labels`, are also predefined in the backend (using your code from Chapter 2).

Each iteration of the `for` loop we will feed in the i^{th} data observation (and its corresponding label) into the model.

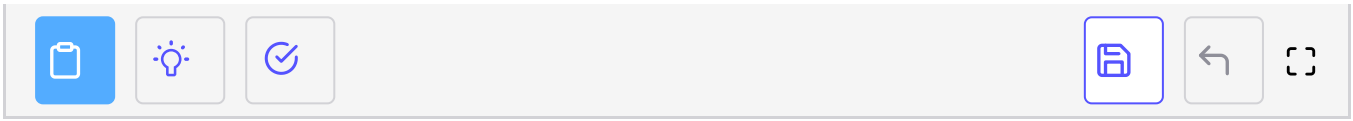
Set `feed_dict` equal to a python dictionary with key-value pairs `inputs: input_data[i]` and `labels: input_labels[i]`.

The ADAM training operation (`train_op`) is defined in the backend, using the code from Chapter 5. Using `sess.run`, we can run training on the input data and labels.

Using the `sess` object defined earlier, call the `run` function with first

argument `train_op` and keyword argument `feed_dict=feed_dict`.

```
# input_data, input_labels, inputs, labels, train_op
# are all predefined in the backend
for i in range(1000):
  # CODE HERE
  pass
```



Evaluation

Evaluate a fully trained neural network using the model accuracy as the evaluation metric.

Chapter Goals:

- Evaluate model performance on a test set

A. Evaluating using accuracy

After training a model, it is a good idea to evaluate its performance. We do this by using a *test set* (i.e. data points not used in model training) and observe the model's prediction accuracy on the test set.

The code for this chapter makes use of the `accuracy` metric defined in Chapter 4. The accuracy represents the classification accuracy of an already trained model, i.e. the proportion of correct predictions the model makes on a test set.

B. Different amounts of data

Since we used `None` when defining our placeholder shapes, it allows us to run training or evaluation on any number of data points, which is super helpful since we normally want to evaluate on many more data points than the training batch size.

It is good practice to split up a dataset into a three sets:

- *Training set (~80% of dataset)*: Used for model training and optimization
- *Validation set (~10% of dataset)*: Used to evaluate the model in between training runs, e.g. when tweaking model parameters like batch size
- *Test set (~10% of dataset)*: Used to evaluate the final model, usually through some accuracy metric

Time to Code!

The coding exercise for this chapter uses the `accuracy` metric from Chapter 4 (which is initialized in the backend). We also provide the test set's data and

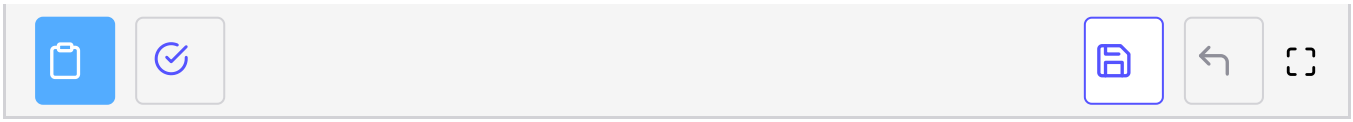
labels (`test_data` and `test_labels`) as NumPy arrays initialized in the backend, as well as the `inputs` and `labels` placeholders.

We've taken the liberty of loading a pretrained model in the backend using a `tf.Session` object called `sess`. You'll be evaluating the accuracy of the pretrained model on the test data and labels.

Set `feed_dict` equal to a python dictionary with key-value pairs `inputs: test_data` and `labels: test_labels`.

Set `eval_acc` equal to the output of `sess.run`, with first argument `accuracy` and a keyword argument `feed_dict=feed_dict`.

```
# test_data, test_labels, inputs, labels, accuracy
# are all predefined in the backend
# CODE HERE
```



Linear Limitations

An overview of the limitations of a single layer perceptron model.

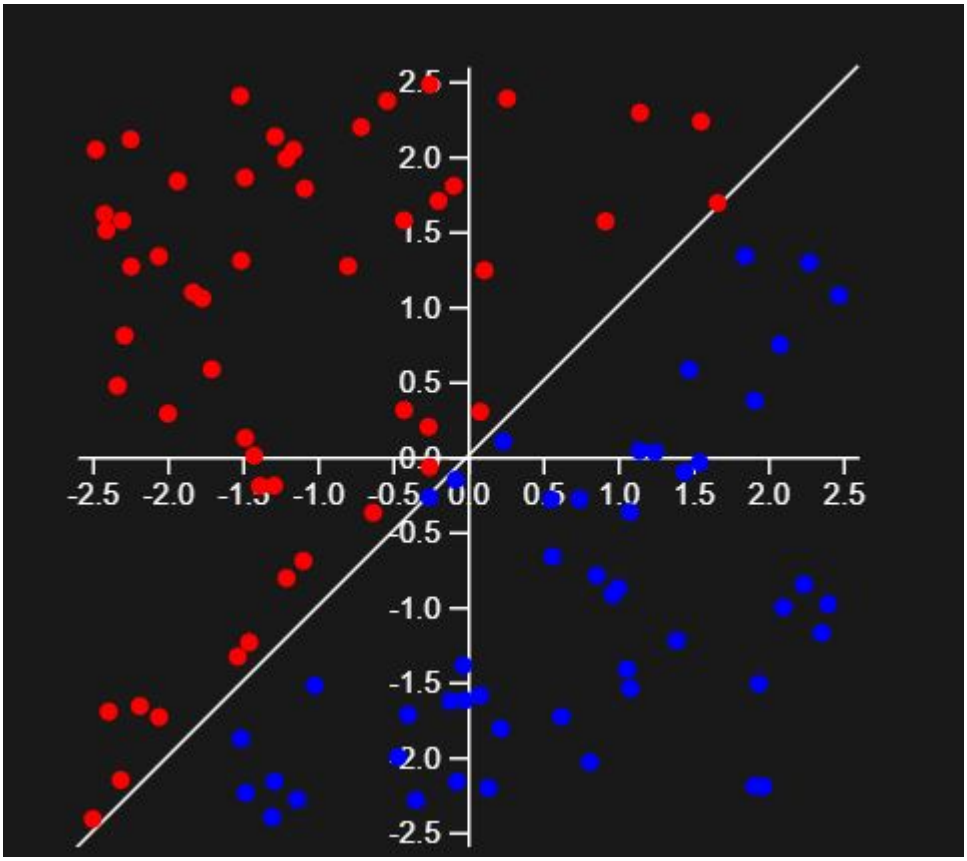
Chapter Goals:

- Understand the limitations of a single layer perceptron model

A. Linear decision boundary

The input data we've been using to train and evaluate the single layer perceptron model has been pairs of (x, y) points with labels indicating whether the point is above (labeled **1**) or below (labeled **0**) the $y = x$ line. We trained the pretrained model on the generated data for 100,000 iterations at a batch size of 50.

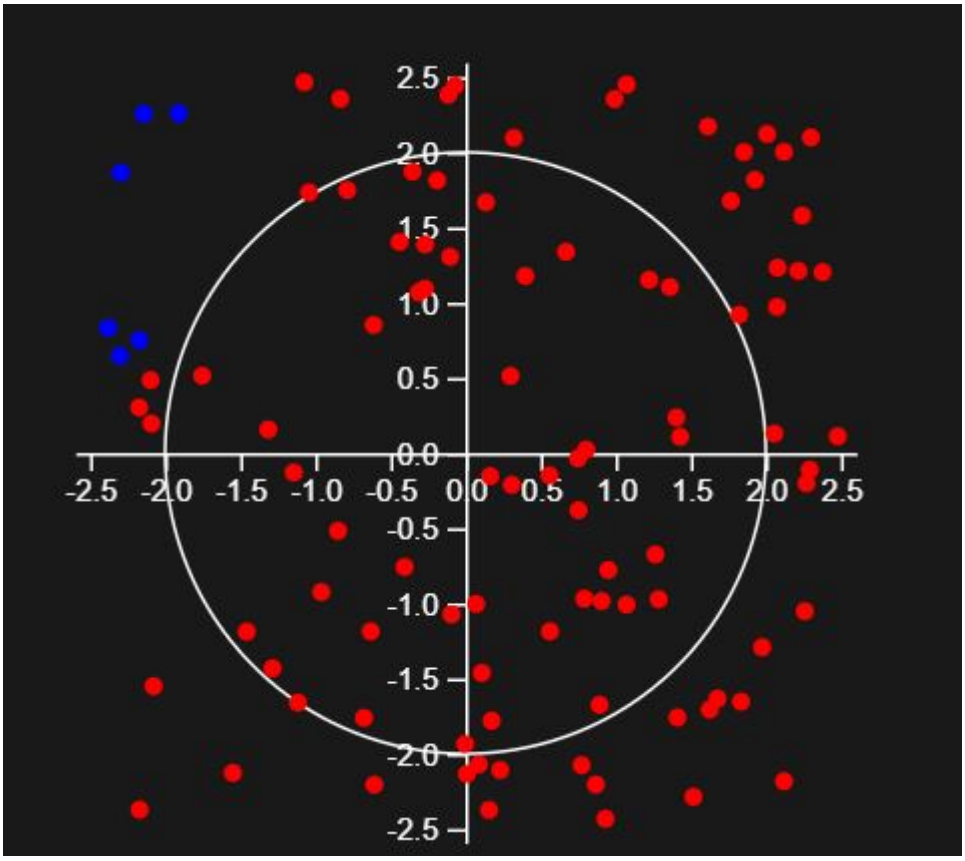
Running the model on some randomly generated 2-D points will give a plot like this:



In the plot above, there are 100 randomly generated 2-D points. The model

was tasked with classifying whether or not each point is above or below the *y* = *x* line (red for above, blue for below). As you can see, the model classifies each of the points correctly.

However, if we instead train the single layer perceptron on 2-D points being inside or outside a circle, the plot will look something like this:



In this case, red means the model thinks the point is inside the circle, while blue means the model thinks the point is outside the circle. In this case, the single layer perceptron does a very poor job of classifying the points, despite being trained on the circle dataset until the loss converged.

The lack of performance from the model is not due to undertraining, but instead due to an inherent limitation in the abilities of a single layer perceptron. With just a single fully-connected layer, the model is only able to learn linear decision boundaries. So for any set of points that are divided by a line in the 2-D plane, the model can be trained to correctly classify those points. However, for non-linear decision boundaries, such as this circle example, no matter how much you train the model it will not be able to perform well in classification.

In the next chapter, we delve into how we can create perceptron models that

are able to learn non-linear decision boundaries.

Hidden Layer

An overview of the limitations of a single layer perceptron model.

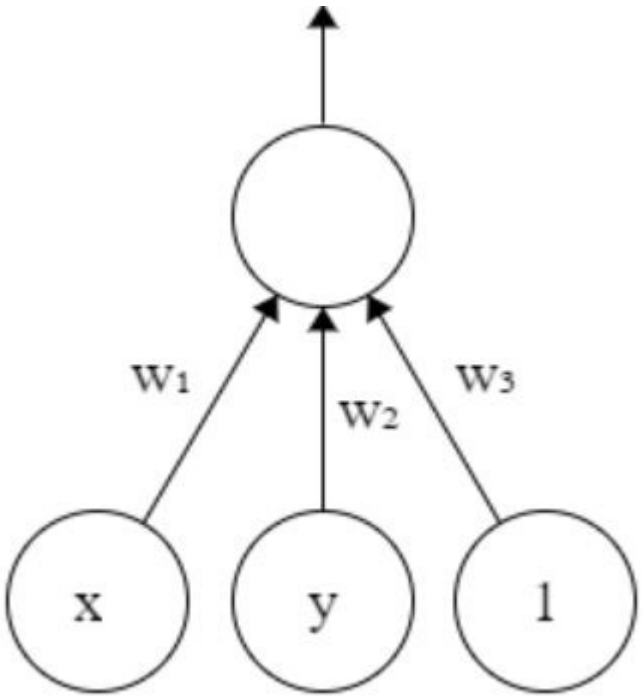
Chapter Goals:

- Add a hidden layer to the model's layers
- Understand the purpose of non-linear activations
- Learn about the ReLU activation function

A. Why a single layer is limited

In the previous chapter we saw that the single layer perceptron was unable to classify points as being inside or outside a circle centered around the origin. This was because the output of the model (the logits) only had connections directly from the input features.

Why exactly is this a limitation? Think about how the connections work in a single layer neural network. The neurons in the output layer have a connection coming in from each of the neurons in the input layer, but the connection weights are all just real numbers.



The single layer neural network architecture. The weight values are denoted w1, w2, and w3.

Based on the diagram, the logits can be calculated as a linear combination of the input layer and weights:

$$logits = w_1 \cdot x + w_2 \cdot y + w_3$$

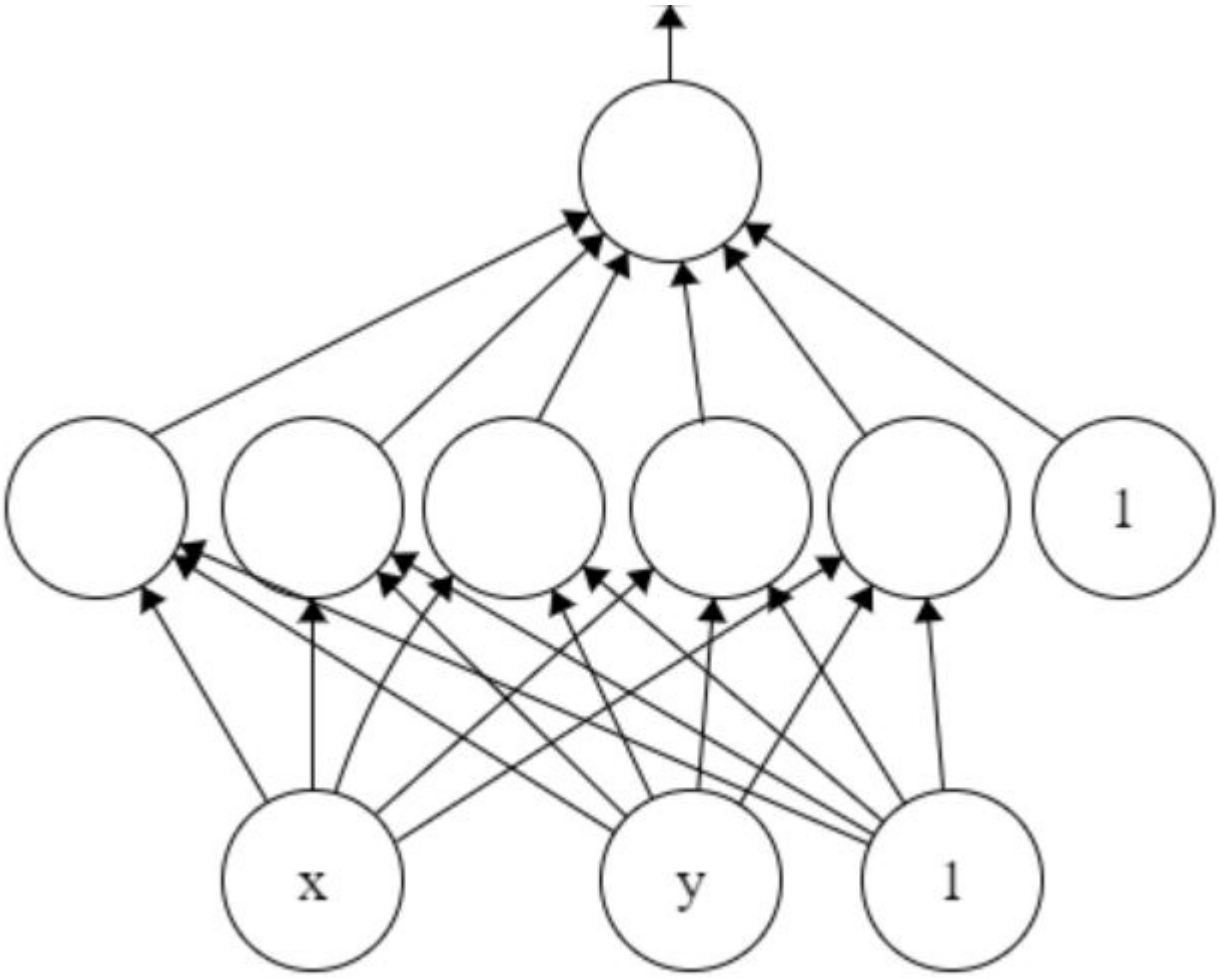
The above linear combination shows the single layer perceptron can model any linear boundary. However, the equation of the circle boundary is:

$$x^2 + y^2 = r^2$$

This is not an equation that can be modeled by a single linear combination.

B. Hidden layers

If a single linear combination doesn't work, what can we do? The answer is to add more linear combinations, as well as *non-linearities*. We add more linear combinations to our model by adding more layers. The single layer perceptron has only an input and output layer. We will now add an additional *hidden layer* between the input and output layers, officially making our model a multilayer perceptron. The hidden layer will have 5 neurons, which means that it will add an additional 5 linear combinations to the model's computation.



The multilayer perceptron architecture.

The 5 neuron hidden layer is more than enough for our circle example. However, for very complex datasets a model could have multiple hidden layers with hundreds of neurons per layer. In the next chapter, we discuss some tips on choosing the number of neurons and hidden layers in a model.

C. Non-linearity

We add non-linearities to our model through *activation functions*. These are non-linear functions that are applied within the neurons of a hidden layer. You've already seen an example of a non-linear activation function, the sigmoid function. We used this after the output layer of our model to convert the logits to probabilities.

The 3 most common activation functions used in deep learning are tanh, ReLU, and the aforementioned sigmoid. Each has its uses in deep learning, so it's normally best to choose activation functions based on the problem. However, the ReLU activation function has been shown to work well in most general-purpose situations, so we'll apply ReLU activation for our hidden

general-purpose situations, so we'll apply ReLU activation for our hidden layer.

D. ReLU

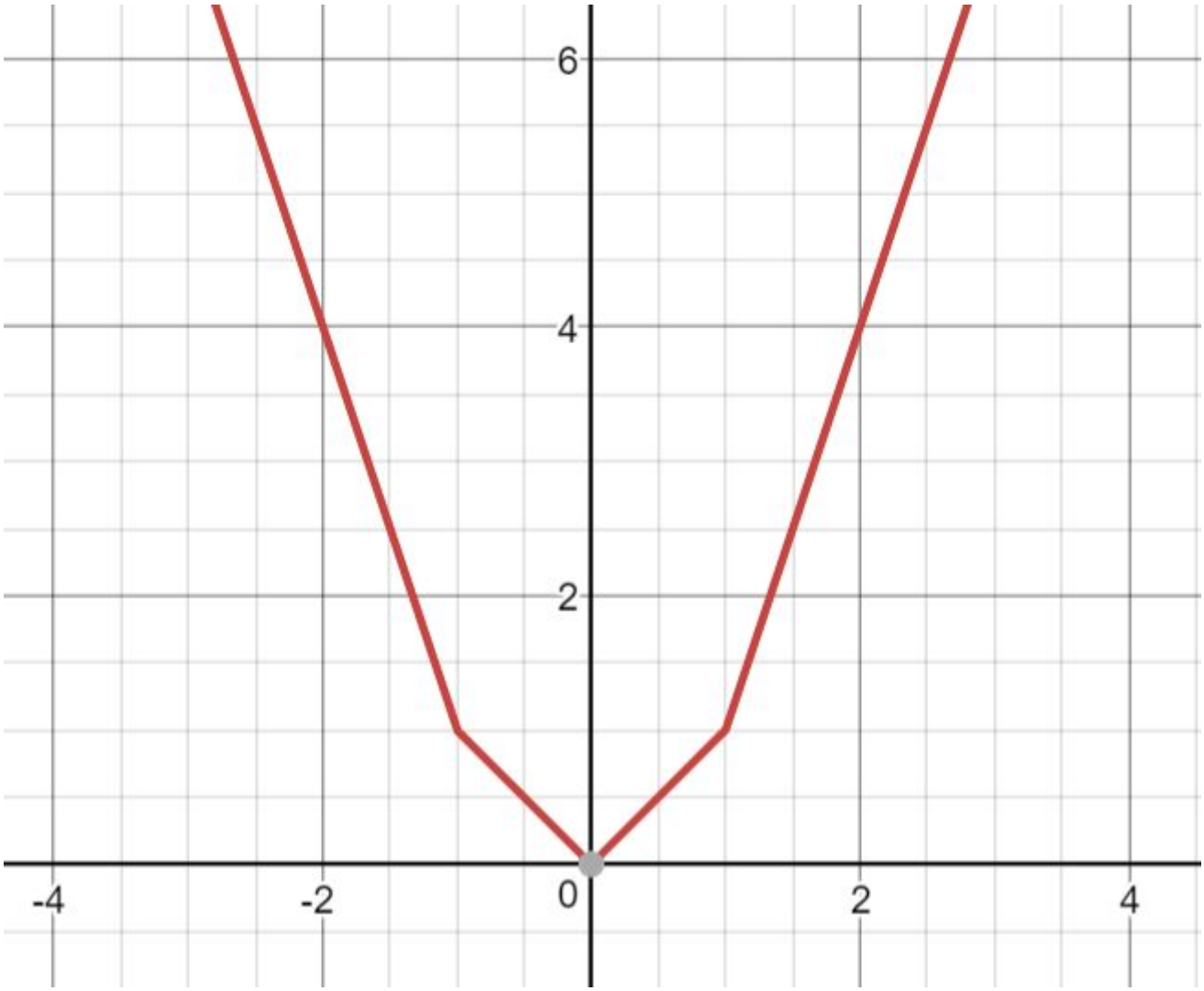
The equation for ReLU is very simple:

$$ReLU(x) = max(0, x)$$

You might wonder why ReLU even works. While tanh and sigmoid are both inherently non-linear, the ReLU function seems pretty linear (it's just $f(x) = 0$ for $x < 0$ and $f(x) = x$ for $x \geq 0$). However, let's take a look at the following function:

$$f(x) = ReLU(x) + ReLU(-x) + ReLU(2x - 2) + ReLU(-2x - 2)$$

This is just a linear combination of ReLU. However, the graph it produces looks like this:



Though a bit rough on the edges, it looks somewhat like the quadratic function, $f(x) = x^2$. In fact, with enough linear combinations and ReLU

function, $f(x) = x^2$. In fact, with enough linear combinations and ReLU

activations, a model can easily learn the quadratic transformation. This is exactly how our multilayer perceptron can learn the circle decision boundary.

We've shown that ReLU is capable of being a non-linear activation function. So what makes it work well in general purpose situations? Its aforementioned simplicity. The simplicity of ReLU, specifically with respect to its gradient, allows it to avoid the vanishing gradient problem. Furthermore, the fact that it maps all negative values to 0 actually helps the model train faster and avoid overfitting (discussed in the next chapter).

Time to Code!

The coding exercise for this chapter involves modifying the `model_layers` function from Chapter 3. You will be adding an additional hidden layer to the model, to change it from a single layer to a multilayer perceptron.

The additional hidden layer will go directly before the `'logits'` layer.

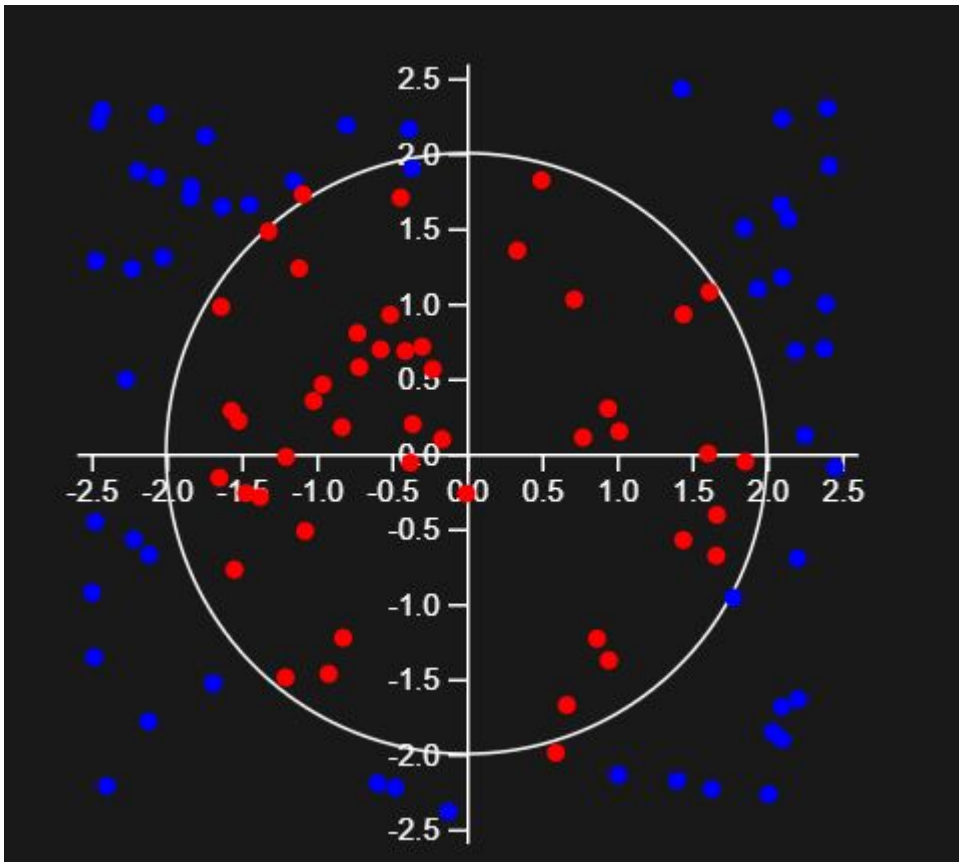
Set `hidden1` equal to `tf.layers.dense` with required arguments `inputs` and `5`, as well as keyword arguments `activation=tf.nn.relu` and `name='hidden1'`.

We also need to update the layer which produces the logits, so that it takes in `hidden1` as input.

Change the first argument of `tf.layers.dense` for `logits` to be `hidden1`.

```
def model_layers(inputs, output_size):
  logits = tf.layers.dense(inputs, output_size,
                           name='logits')
  return logits
```

After adding in the hidden layer to the model, the multilayer perceptron will be able to classify the 2-D circle dataset. Specifically, the classification plot will now look like:



Blue represents points the model thinks is outside the circle and red represents points the model thinks is inside. As you can see, the model is a lot more accurate now.

Multiclass

Understand the differences between binary and multiclass classification.

Chapter Goals:

- Learn about multiclass classification
- Understand the purpose of multiple hidden layers
- Learn the pros and cons of adding hidden layers

A. Multiclass classification

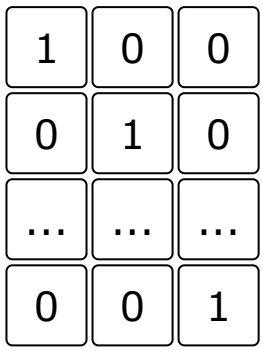
In the previous chapters we focused on binary classification, labeling whether or not an input data point has some class attribute (e.g. if it is in a circle or not). Now, we will attempt to classify input data points when there are multiple possible classes and the data point belongs to exactly one. This is referred to as multiclass classification.

The example is an extension of the previous circle example, but now there is an additional circle with radius 1 centered at the origin. The classes are now:

- **0**: Outside both circles
- **1**: Inside the smaller circle
- **2**: Outside the smaller circle but inside the larger circle

B. One-hot

Instead of representing each label as a single number, we use a *one-hot vector*. A one-hot vector has length equal to the number of classes, which is the same as `output_size`. The vector contains a single 1 at the index of the class that the data point belongs to, i.e. the *hot index*, and 0's at the other indexes. The labels now become a set of one-hot vectors for each data point:



An example set of labels. In this case there are 3 possible classes, exactly one of which is the hot index.

Another way to think about one-hot vectors is as multiple binary classification. The actual class of the data point is labeled as 1 (True), while the other classes are labeled as 0 (False).

C. Adding hidden layers

Since there are now multiple decision boundaries, it would be beneficial to either increase the size of our model's current hidden layer, `hidden1`, or add another hidden layer. Given that the decision boundaries are still relatively trivial, both methods would lead to successful models eventually. However, adding an extra hidden layer may decrease the number of training iterations needed for convergence compared to maintaining a single hidden layer.

When deciding how many hidden layers a model needs (i.e. how deep it is) and how many neurons are at each hidden layer, it is a good idea to base the decision on the problem itself. There are a few general rules of thumb, but they do not apply to every scenario. For example, it is common not to need more than 3 hidden layers in a neural network, but if you are working on a complicated problem you would most likely need more (Google's Alpha Go needed more than a dozen layers).

If you don't have much domain knowledge for the particular problem you're working on, it's usually best to only add extra layers or neurons when they're needed. The fewer layers and neurons, the faster your model trains, and the quicker you can evaluate how good it is. It then becomes easier to optimize the number of layers and neurons in your model through experimentation.

D. Overfitting

One thing to note is that the more hidden layers or neurons a neural network has, the more prone it is to overfitting the training data. Overfitting refers to the model becoming very accurate in classifying the training data, but then performing poorly on other data. Since we want models that can generalize well and accurately classify data it has never seen before, it is best to avoid going overboard in adding hidden layers.

Time to Code!

The coding exercise for this chapter involves modifying the `model_layers` function from the previous chapter. You will be adding an additional hidden layer to the model, bringing the total number of hidden layers to 2.

The additional hidden layer will go directly before the `'logits'` layer.

Set `hidden2` equal to `tf.layers.dense` with required arguments `hidden1` and `5`, as well as keyword arguments `activation=tf.nn.relu` and `name='hidden2'`.

We also need to update the layer which produces the logits, so that it takes in `hidden2` as input.

Change the first argument of `tf.layers.dense` for `logits` to be `hidden2`.

```
def model_layers(inputs, output_size):
  hidden1 = tf.layers.dense(inputs, 5,
                            activation=tf.nn.relu,
                            name='hidden1')
  logits = tf.layers.dense(hidden1, output_size,
                           name='logits')
  return logits
```

Softmax

Use the softmax function to convert a neural network from binary to multiclass classification.

Chapter Goals:

- Update the model to use the softmax function
- Perform multiclass classification

A. The softmax function

To convert the model to multiclass classification, we need to make a few changes to the metrics and training parameters. Previously, we used the sigmoid function to convert logits to probabilities, then rounded those probabilities to get a predicted label. However, now that there are multiple possible classes, we need to use the generalization of the sigmoid function, known as the softmax function.

The softmax function takes in a vector of numbers (logits for each class), and converts the numbers to a probability distribution. This means that the sum of the probabilities across all the classes equals 1, and each class's individual probability is based on how large its logit was relative to the sum of all the classes's logits.

The code below demonstrates how to use the TensorFlow `tf.nn.softmax` function to apply softmax to a tensor.

```
t = tf.constant([[0.4, -0.8, 1.3],
                 [0.2, -1.2, -0.4]])
softmax_t = tf.nn.softmax(t)
sess = tf.Session()
print('{}\n'.format(repr(sess.run(t))))
print('{}\n'.format(repr(sess.run(softmax_t))))
```

When training our model, we also replace the sigmoid cross entropy with a

softmax cross entropy, for the same reason as stated above. The cross entropy is now calculated *for each class* and then averaged at the end.

B. Predictions

Our model's prediction now becomes the class with the highest probability. Since we label each class with a unique index, we need to return the index with the maximum probability. TensorFlow provides a function that lets us do this, called `tf.argmax`.

The function takes in a required input tensor, as well as a few keyword arguments. One of the more important keyword arguments is `axis`, which tells us which dimension to retrieve the maximum index from. Setting `axis=-1` uses the final dimension, which in this case corresponds to retrieving the column index.

The code below generates multiclass predictions from probabilities using `tf.argmax`. The prediction for each row of probabilities is just the column index with the highest probability.

```
probs = tf.constant([[0.4, 0.3, 0.3],
                     [0.2, 0.7, 0.1]])
preds = tf.argmax(probs, axis=-1)
sess = tf.Session()
print('{}\n'.format(repr(sess.run(probs))))
print('{}\n'.format(repr(sess.run(preds))))
```

Time to Code!

The coding exercises for this chapter calculates multiclass predictions from logits, and then applies training with softmax cross entropy loss.

We'll first calculate the probabilities from `logits` (predefined in the backend), using the softmax function. Then we can use the `tf.argmax` function to generate the predictions.

Set `probs` equal to `tf.nn.softmax` applied to `logits`.

Set `predictions` equal to `tf.argmax` applied to `probs`, with keyword argument `axis=-1`.

```
# CODE HERE
```

Since our `labels` are one-hot vectors (predefined in the backend), we need to convert them back to class indexes to calculate our accuracy.

Set `class_labels` equal to tf.argmax applied to `labels`, with keyword argument `axis=-1`.

Set `is_correct` equal to tf.equal applied with `predictions` and `class_labels` as input arguments.

```
# CODE HERE
```

From this point, the calculation of the model's accuracy (using the `is_correct` variable is the same as in Chapter 4.

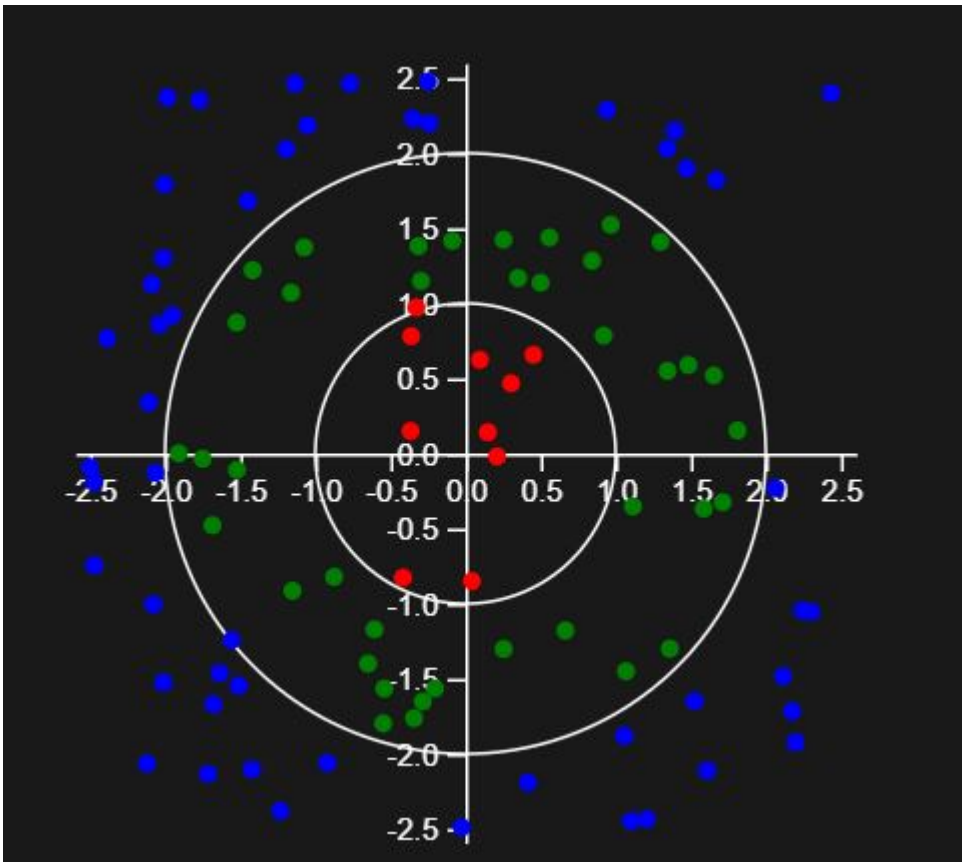
For training the model, the main change we need to make is going from sigmoid cross entropy to softmax cross entropy. In TensorFlow, softmax cross entropy is applied via the `tf.nn.softmax_cross_entropy_with_logits_v2` function.

Set `cross_entropy` equal to `tf.nn.softmax_cross_entropy_with_logits_v2`, applied with `labels` and `logits` for the `labels` and `logits` keyword arguments, respectively.

```
# CODE HERE
```

The remainder of the training optimization code is the same as in Chapter 5.

Now if we run the trained 2-hidden layer MLP model on a multiclass circle dataset, the plot will look like this:



Blue represents points that the model believes is outside both circles, green represents points the model believes is between the two circles, and red represents points the model believes is inside both circles.

As you can see, the multiple hidden layer MLP model performs quite well on this basic multiclass classification task.

Quiz

1 What is a benefit to using more hidden layers in a neural network?

COMPLETED 0%

1 of 4

Introduction

An overview of the Keras API and how it compares to TensorFlow.

In the **Intro to Keras** section, you will learn about the Keras API, a simple and compact API for creating neural networks. You will use Keras to build a multilayer perceptron model for multiclass classification.

A. The Keras API

The most popular deep learning framework in the world is TensorFlow. It is incredibly powerful, efficient, and widely used in industry. However, a downside to TensorFlow is that the code can be a bit complex, especially when setting up a model for training or evaluation.

A simpler alternative to TensorFlow is Keras. The Keras API is easier to use than TensorFlow, allowing us to create, train, and evaluate a deep learning model with considerably less code. Interestingly, Keras is often run on top of TensorFlow, acting as a wrapper API to make the coding simpler.

While Keras is excellent for building small deep learning projects, TensorFlow is still the preferred framework for industry-level projects since it provides more utilities and efficient training mechanisms.

B. Multilayer perceptron

The MLP model is one of the most important neural networks for deep learning. It is a relatively simple model, but versatile enough for a variety of different applications. In this section, we'll be focusing on the Keras implementation of an MLP, rather than go into details on how it works or what it can be used for.

For specific details on the MLP model, check out the **Intro to Deep Learning** section.

Sequential Model

Learn how a neural network is built in Keras.

Chapter Goals:

- Initialize an MLP model in Keras

A. Building the MLP

In Keras, every neural network model is an instance of the `Sequential` object. This acts as the container of the neural network, allowing us to build the model by stacking multiple layers inside the `Sequential` object.

The most commonly used Keras neural network layer is the `Dense` layer. This represents a fully-connected layer in the neural network, and it is the most important building block of an MLP model.

When building a model, we start off by initializing a `Sequential` object. We can initialize an empty `Sequential` object and add layers onto the model using the `add` function, or we can directly initialize the `Sequential` object with a list of layers.

```
model = Sequential()
layer1 = Dense(5, input_dim=4)
model.add(layer1)
layer2 = Dense(3, activation='relu')
model.add(layer2)
```

Adding two Dense layers to a Sequential model.

```
layer1 = Dense(5, input_dim=4)
layer2 = Dense(3, activation='relu')
model = Sequential([layer1, layer2])
```

Initializing a Sequential model with two Dense layers.

The `Dense` object takes in a single required argument, which is the number of neurons in the fully-connected layer. The `activation` keyword argument specifies the activation function for the layer (the default is no activation). In the code snippets above, we used no activation for `layer1` and ReLU activation for `layer2`.

The first layer of the `Sequential` model represents the input layer. Therefore, in the first layer we need to specify the feature dimension of the input data for the model, which we do with the `input_dim` keyword argument.

In the code snippets above, we set the input feature dimension to 4, meaning that the input data has shape `(batch_size, 4)` (where `batch_size` is the data's batch size, decided at runtime).

Time to code!

The coding exercise for this chapter involves setting up a Keras `Sequential` model with a single `Dense` layer. We start off with an empty initialized `Sequential` object.

Set `model` equal to `Sequential` initialized with no arguments.

```
# CODE HERE
```

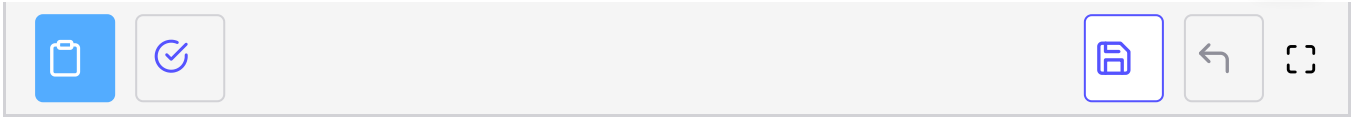
We'll build a three layer MLP model. The first layer will consist of 5 neurons and use ReLU activation. It will also act as the input layer for the model.

To create the input layer, we'll initialize a `Dense` object with the requisite number of neurons and activation. We'll also set the `input_dim` keyword argument to `2`, which represents the feature dimension of the input data for the model.

Set `layer1` equal to a `Dense` with `5` as the required argument, `'relu'` for the `activation` keyword argument, and `2` for the `input_dim` keyword argument.

Then call `model.add` on `layer1`.

```
# CODE HERE
```



Model Output

Complete a multilayer perceptron model in Keras.

Chapter Goals:

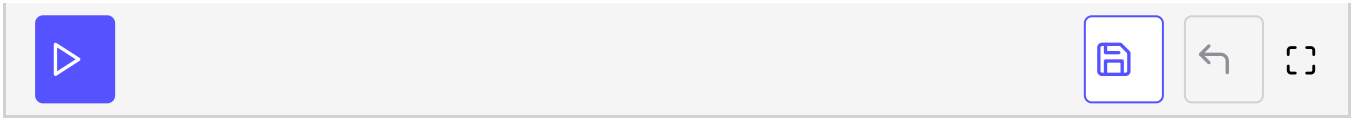
- Add the final layers to the MLP for multiclass classification

A. Final layer activation

In the **Intro to Deep Learning** section, we built the MLP classification models such that each model produced logits. This is because the TensorFlow cross-entropy loss functions applied the sigmoid/softmax function to the output of the MLP.

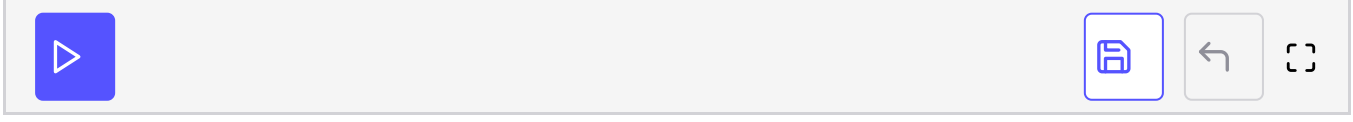
In Keras, the cross-entropy loss functions only calculate cross-entropy, without applying the sigmoid/softmax function to the MLP output. Therefore, we can have the model directly output class probabilities instead of logits (i.e. we apply sigmoid/softmax activation to the output layer).

```
model = Sequential()
layer1 = Dense(5, activation='relu', input_dim=4)
model.add(layer1)
layer2 = Dense(1, activation='sigmoid')
model.add(layer2)
```



Creating an MLP model for binary classification (sigmoid activation).

```
model = Sequential()
layer1 = Dense(5, input_dim=4)
model.add(layer1)
layer2 = Dense(3, activation='softmax')
model.add(layer2)
```



Creating an MLP model for multiclass classification with 3 classes (softmax activation)

Creating an MLP model for multiclass classification with 3 classes (softmax activation).

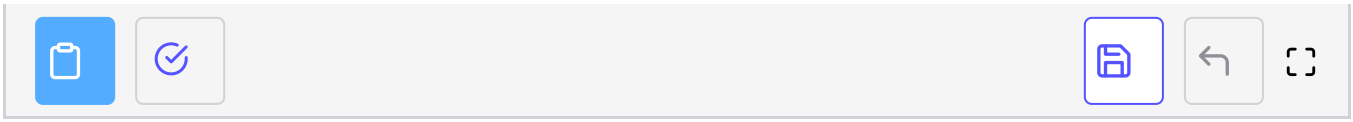
Time to code!

The coding exercise will complete the Keras `Sequential` model that was set up in the previous chapter. Note that the output size of the model will be 3 (there are 3 possible classes for each data observation).

Set `layer2` equal to a `Dense` with `5` as the required argument and `'relu'` for the `activation` keyword argument. Then call `model.add` on `layer2`.

Set `layer3` equal to a `Dense` with `3` as the required argument and `'softmax'` for the `activation` keyword argument. Then call `model.add` on `layer3`.

```
model = Sequential()
layer1 = Dense(5, activation='relu', input_dim=2)
model.add(layer1)
# CODE HERE
```



Model Configuration

Configure the Keras model for training.

Chapter Goals:

- Learn how to configure a Keras model for training

A. Configuring for training

When it comes to configuring the model for training, Keras shines with its simplicity. A single call to the `compile` function allows to set up all the training requirements for the model.

The function takes in a single required argument, which is the optimizer to use, e.g. ADAM. A full list of optimizers can be found here. A shorthand way to specify the optimizer is to use a (lowercase) string of the optimizer's name (e.g. `'adam'`, `'sgd'`, `'adagrad'`).

The two main keyword arguments to know are `loss` and `metrics`. The `loss` keyword argument specifies the loss function to use. For binary classification, we set the value to `'binary_crossentropy'`, which is the binary cross-entropy function. For multiclass classification, we set the value to `'categorical_crossentropy'`, which is the multiclass cross-entropy function.

The `metrics` keyword argument takes in a list of strings, representing metrics we want to track during training and evaluation. For classification, we only need to track the model loss (which is tracked by default) and the classification accuracy.

```
model = Sequential()
layer1 = Dense(5, activation='relu', input_dim=4)
model.add(layer1)
layer2 = Dense(1, activation='sigmoid')
model.add(layer2)
model.compile('adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

The code example above creates an MLP model for binary classification and configures it for training. We specified classification accuracy as the metric to track.

Time to code!

The coding exercise will complete the Keras multiclass classification model for training.

To configure the model for training, we need to use the `compile` function. The function sets up the model's loss, optimizer, and evaluation metrics.

For our model, we'll use the ADAM optimizer. Since the model performs multiclass classification, we'll use `'categorical_crossentropy'` for the loss.

The only metric we need to know during training and evaluation is the model's classification accuracy.

Call `self.model.compile` with `'adam'` as the required argument. Use `'categorical_crossentropy'` and `['accuracy']` for the `loss` and `metrics` keyword arguments.

```
model = Sequential()
layer1 = Dense(5, activation='relu', input_dim=2)
model.add(layer1)
layer2 = Dense(5, activation='relu')
model.add(layer2)
layer3 = Dense(3, activation='softmax')
model.add(layer3)
# CODE HERE
```

Model Execution

Learn how to train, evaluate, and make predictions with a Keras model.

Chapter Goals:

- Understand the facets of model execution for Keras models

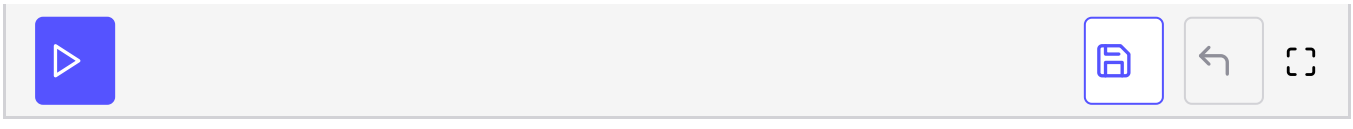
A. Training

After configuring a Keras model for training, it only takes a single line of code to actually perform the training. We use the `Sequential` model's `fit` function to train the model on input data and labels.

The first two arguments of the `fit` function are the input data and labels, respectively. Unlike TensorFlow (where we need to use tensor objects for any sort of data), we can simply use NumPy arrays as the input arguments for the `fit` function.

The training batch size can be specified using the `batch_size` keyword argument (the default is a batch size of 32). We can also specify the number of epochs, i.e. number of full run-throughs of the dataset during training, using the `epochs` keyword argument (the default is 1 epoch).

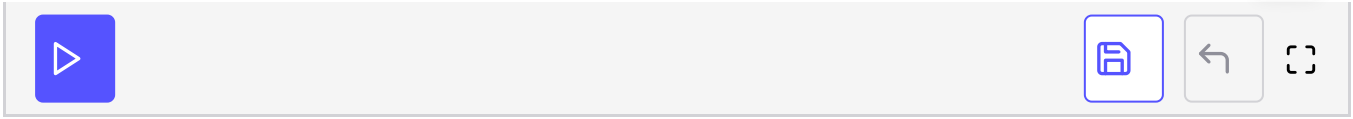
```
model = Sequential()
layer1 = Dense(200, activation='relu', input_dim=4)
model.add(layer1)
layer2 = Dense(200, activation='relu')
model.add(layer2)
layer3 = Dense(3, activation='softmax')
model.add(layer3)
model.compile('adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# predefined multiclass dataset
train_output = model.fit(data, labels,
                         batch_size=20, epochs=5)
```



The console output of the training is logged for each epoch. Notice that both the loss and classification accuracy per epoch is measured (since we configured the model to track classification accuracy).

The output of the `fit` function is a `History` object, which records the training metrics. The object's `history` attribute is a dictionary that contains the metric values at each epoch of training.

```
print(train_output.history)
```



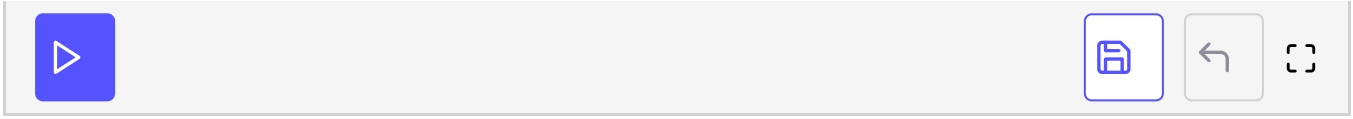
The history for the train_output object from the previous code snippet.

B. Evaluation

Evaluating a trained Keras model is just as simple as training it. We use the `Sequential` model's `evaluate` function, which also takes in data and labels (NumPy arrays) as its first two arguments.

Calling the `evaluate` function will evaluate the model over the entire input dataset and labels. The function returns a list containing the evaluation loss as well as the values for each metric specified during model configuration.

```
# predefined eval dataset
print(model.evaluate(eval_data, eval_labels))
```



Evaluating the previously trained model.

C. Predictions

Finally, we can make predictions with a Keras model using the `predict` function. The function takes in a NumPy array dataset as its required argument, which represents the data observations that the model will make predictions for.

The output of the `predict` function is the output of the model. That means for

classification, the `predict` function is the model's class probabilities for each data observation.

```
# 3 new data observations
print('{}'.format(repr(model.predict(new_data))))
```

Class probabilities for multiclass new data observations.

In the code above, the model returned class probabilities for the 3 new data observations. Based on the probabilities, the first observation would be classified as class **0**, the second observation would be classified as class **2**, and the third observation would be classified as class **1**.

Quiz

1 What is the main benefit of Keras over TensorFlow?

COMPLETED 0%

1 of 3

Course Conclusion

A. Closing thoughts

Machine Learning (ML) is one of the hottest fields in technology right now, and is inherently relevant to artificial intelligence and data science. From healthcare and agriculture to manufacturing and retail, many companies across a variety of different industries are leveraging these technologies to get ahead.

B. Course recap

This course has provided an overview of how to write useful code and impactful machine learning applications. You'll be able to take the practical lessons and actionable insights from this course and apply them to your projects.

C. Topics covered

- Data analysis/visualization
- Feature engineering
- Supervised learning
- Unsupervised learning
- Deep learning

Industry standard frameworks used in this course

- NumPy
- pandas
- scikit-learn
- XGBoost
- TensorFlow
- Keras