



(2025, D. T. McGuiness, Ph.D)

Current version is SS.2025.

This document includes the contents of Data Science I - Tutorial taught at MCI. This document is the part of module.

All relevant code of the document is done using Python with heavy use of sklearn, matplotlib and numpy libraries.

This document was compiled with  $\text{LuaT}\bar{\text{E}}\text{X}$  and all editing were done using GNU Emacs using  $\text{AUCT}\bar{\text{E}}\text{X}$  and org-mode package.

This document is meant to be used for educational purposes only at MCI and not to be distributed for commercial purposes.

This document is based on the books on the topics of Data Science, Machine Learnign and Programming (Python), which includes *AI and Machine Learning for Coders* by L. Moroney (1<sup>st</sup> Edition), *Hands-on Machine Learning with Scikit-Learn Keras & TensorFlow* by A. Géron (2<sup>nd</sup> Edition) *Machine Learning with Python Cookbook* by C. Albon (1<sup>st</sup> Edition), *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2* by S. Raschka et. al. (3<sup>rd</sup> Edition), and *Neural Networks and Deep Learning* by C. C. Aggarwal (1<sup>st</sup> Edition).

The current maintainer of this work along with the primary lecturer is D. T. McGuiness, Ph.D () .

# Data Science I - Tutorial

---

D. T. McGuiness, Ph.D

SS.2025

MCI



# Table of Contents



## Document

- 1.** Machine Learning Landscape
- 2.** End-to-End ML Project
- 3.** Classification
- 4.** Training Models

# Machine Learning Landscape

---

# Table of Contents



## Defining ML

Learning Outcomes

## The Point of ML

Spam Filter Example

Application Examples

## Machine Learning Systems

Training Supervision

Batch v Online

Instance v. Model

## Challenges of ML

Lack of Training Data Quality

Non-representative Training Data

Poor Quality of Data

Irrelevant Features

Over-fitting Training Data

Under-Fitting Training Data

## Tests and Validations

Training and Testing Sets

Hyper-Parameter Tuning

Data Mismatch



## Learning Outcomes

- (LO1) An Introduction to ML,
- (LO2) Overview of Learning Methods,
- (LO3) Application of ML Algorithms,
- (LO4) General Problems of ML Training/Evaluations.





- Spam filters are a great place to start showing use of ML.
- A spam filter is a ML program where, given examples of spam emails (flagged by users) and examples of regular emails (non-spam, also called **ham**), can learn to flag spam.
- Examples the system uses to learn are called the **training set**.
- Each training example is called a training instance (i.e., sample).
- Part of the ML system learning and making predictions is called a **model**.
  - Neural networks and random forests are examples of models.

# Machine Learning Landscape



Figure 1: "Spam, Spam, Spam, Spam...Lovely Spam! Wonderful Spam!"



- Consider how you would write a spam filter using **traditional method**;
  1. Define spam.
    - Some words or phrases (such as `4U`, `credit card`, `free`, `amazing`) tend to come up a lot in subject.
    - You also notice sender email also looks **fishy**.
  2. Write an `if/else` statement for each case, and the program would flag emails as `spam` if a number of these patterns were detected.
  3. Test program and release it to the public.
  4. Profit!!

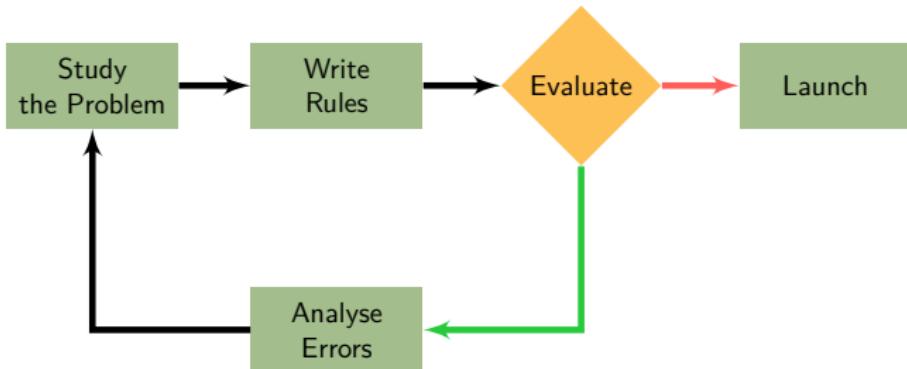


Figure 2: A block diagram on how to structure a spam filter using the traditional programming methods.



- As the problem is difficult, the traditional program will become a long list of complex rules.
  - This would be pretty hard to maintain.
- Whereas, a spam filter using ML automatically learns which **words** and **phrases** are good predictors of spam by detecting **unusually frequent patterns** of words in the spam examples compared to the ham examples [1].



- As the problem is difficult, the traditional program will become a long list of complex rules.
  - This would be pretty hard to maintain.
- Whereas, a spam filter using ML automatically learns which words

The program is much shorter, easier to maintain, and most likely more accurate.

# Machine Learning Landscape

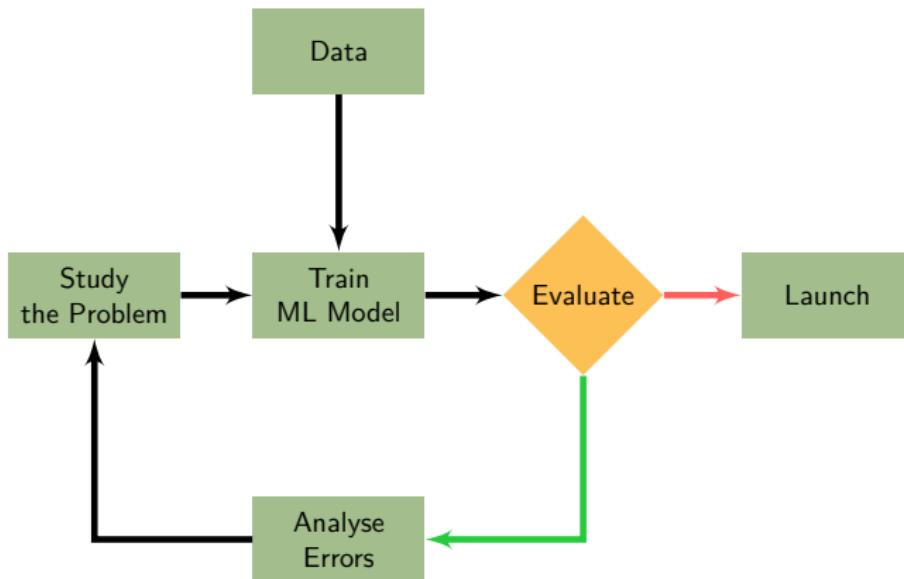


Figure 3: A block diagram on how to structure a spam filter using the ML approach.



- What if spammers notice that all their emails containing **4U** are blocked?
  - They might start writing **For U** instead.
- The traditional method needs to be updated to flag **For U** emails.
- If spammers keep working around your spam filter, you will need to keep writing new rules forever.

A spam filter based on ML automatically notices that **For U** has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention.



ML works best for problems too complex for traditional approaches or have no known algorithm.

- Consider **speech recognition** [2]:
- Say you want to write a program capable of distinguishing the words **one** and **two**.
- You notice the word **two** starts with a high-pitch sound **T**.
- You could hard-code an algorithm measuring high-pitch sound intensity and use it to distinguish ones and twos.
  - This technique **will not scale** to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages [3].
- The best solution is to write an algorithm that **learns by itself**.



- ML can also help humans learn.
- ML models can be inspected to see what they have learned.
  - Although for some models this can be tricky.
    - such as **black-box** models.
- i.e., once a spam filter has been trained on enough spam, it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam.
- Sometimes this will reveal unsuspected correlations or new trends, and thereby lead to a better understanding of the problem.
- Digging into large amounts of data to discover hidden patterns is called **data mining**, and ML excels at it [4].



- To summarise, ML is great for:
  - Problems where existing solutions require significant fine-tuning or long lists of rules.
  - Complex problems for which using a traditional approach gives no good solution.
  - Fluctuating environments.
  - Getting insights about complex problems and large amounts of data.



- Analysing images on production line items classify [5]:
  - Called **image classification**, performed using convolutional neural networks or sometimes transformers.
- Detecting tumours in scans [6]:
  - Called **semantic image segmentation**, where each pixel in the image is classified, to determine the exact location and shape of tumours, typically using CNNs or transformers.
- Automatically classifying news articles [7]:
  - Called **natural language processing** (NLP), more specifically text classification, tackled using **recurrent neural networks** (RNNs) and CNNs, but transformers work even better.



- Automatically flagging offensive comments on discussion forums [8]
  - Called **text classification**, using the same NLP tools.
- Summarising long documents automatically [9]:
  - Called **text summarising**, a sub-field of NLP.
- Creating a chat-bot or a personal assistant [10, 11]
  - Involves many NLP components, including:
    - Natural language understanding (NLU),
    - Question-Answering modules.



- Forecasting a company's revenue next year [12], based on many metrics.
  - Called **regression** using any regression model, such as:
    - linear regression,
    - polynomial regression,
    - regression support vector machine,
    - regression random forest,
    - artificial neural network.
  - If you want to take into account sequences of past performance metrics, RNNs, CNNs, or transformers may prove useful [13].



- Making your app react to voice commands.
  - Called **speech recognition**, requiring processing audio samples:
    - As they are long and complex sequences, they are typically processed using RNNs, CNNs, or transformers.
- Detecting credit card fraud:
  - Called **anomaly detection**, tackled using:
    - isolation forests,
    - Gaussian mixture models,
    - Auto-encoders.



- Classifying clients on their purchases to design a marketing strategy based on class [14].
  - Called **clustering**, which can be achieved using k-means, DBSCAN, and more.
- Representing a complex, high-dimensional dataset in a clear and insightful diagram.
  - Called data visualisation, which involves **dimensionality reduction** techniques.



- Recommending a product a client may be interested in, based on past purchases [15]:
  - Called a **recommender system**.
  - One approach is to feed past purchases to an ANN, and get it to output the most likely next purchase.
  - This ANN would typically be trained on past sequences of purchases across all clients.
- Building an intelligent bot for a game.
  - Tackled using **reinforcement learning**.
    - A branch of machine learning that trains agents to pick the actions that will maximise their rewards over time.
  - The famous AlphaGo program that beat the world champion at the game of Go was built using RL.



**Figure 4:** Deep Blue, computer chess-playing system designed by IBM in the early 1990s, playing against then current grand-master Garry Kasparov. It became the first computer winning against a world champion under tournament conditions [16].

# Machine Learning Landscape



**Figure 5:** AlphaGo was designed to play GO (a very complex game for computers to tackle) and was able to win a master which was deemed a milestone in ML.



- There are types of ML useful to classify data in broad categories:
  - Supervision during training:
    - Supervised,
    - Unsupervised,
    - Semi-supervised,
    - Self-supervised.
  - Whether or not they can learn incrementally on the fly:
    - Online v. Batch Learning,
  - Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model:
    - Instance v. Model based learning.



- There are two types of ML useful to classify data in broad categories:
  - Supervision during training:
    - Supervised,

These criteria are not exclusive. It is possible to combine them. For example, a state-of-the-art spam filter may learn on the fly using a DNN model trained using human-provided examples of spam and ham; this makes it an online, model-based, supervised learning system.

- Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model:
  - Instance v. Model based learning.



- ML can be classified based on the amount and type of supervision they get during training.
- There are many categories, but we'll discuss the main ones:
  - supervised learning,
  - unsupervised learning,
  - self-supervised learning,
  - semi-supervised learning,
  - reinforcement learning.



## Supervised learning

- The fed training set includes the desired solutions, called **labels**.
- A typical supervised learning task is **classification**.
- The spam filter is a good example of this:
  - Trained with many example emails along with their class, and it must learn how to classify new emails.
- Another typical task is to predict a target numeric value, such as the price of a car, given a set of features.
  - Called **regression** and to train it you need many examples of cars, including both their features and their targets.



## Supervised learning

- The fed training set includes the desired solutions, called **labels**.
- A typical supervised learning task is **classification**.

Some regression models can be used for classification as well, and vice versa. i.e., logistic regression is commonly used for classification [17], as it can output a value that corresponds to the probability of belonging to a given class (e.g., 20% chance of being spam).

price of a car, given a set of features.

- Called **regression** and to train it you need many examples of cars, including both their features and their targets.



## Supervised learning

- Target and label are generally treated as synonyms in supervised learning.
- But target is more common in regression tasks and label is more common in classification tasks.
- Features are sometimes called predictors or attributes.
- These terms may refer to individual samples, i.e.,  
**individual** this car's mileage feature is equal to 15,000,  
**all** the mileage feature is strongly correlated with price



## Unsupervised learning

- Training data is unlabelled where it tries to learn **without a teacher**.
- For example say you have a lot of data about your blog's visitors.
  - Run a clustering algorithm to try to detect groups of similar visitors.
  - At no point algorithm knows which group a visitor belongs to,
    - it finds those connections **without your help**.
    - i.e., it notices % of visitors are teenagers who love comic books and read your blog after school while % are adults who enjoy sci-fi and who visit during the weekends.
    - If you use a hierarchical clustering algorithm it may also subdivide each group into smaller groups.
    - This may help you target your posts for each group.



## Unsupervised learning

- Visualisation algorithms are also good examples.
- Feed them a lot of complex and unlabelled data and they output a 2D or 3D representation of your data that can easily be plotted.
- These algorithms try to preserve as much structure as they can.
  - Trying to keep separate clusters in the input space from overlapping in the visualisation.
- This would show how the data is organised and perhaps identify **un-suspected patterns**.



## Unsupervised learning

- A related task is **dimensionality reduction** where the goal is to simplify the data without losing too much information.
- A way is to merge several correlated features into one.
- i.e., a car's mileage may be strongly correlated with its age so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear.
- This is called **feature extraction**.



## Unsupervised learning

It is a good idea to reduce the number of dimensions in your training data using a dimensionality reduction algorithm before feeding it to another ML algorithm (such as a supervised learning algorithm)

It will run much faster the data will take up less disk and memory space and in some cases it may also perform better [18].



## Unsupervised learning

- Another task is **anomaly detection**.
  - Detecting unusual credit card transactions to prevent fraud,
  - Catching manufacturing defects,
  - Automatically removing outliers before feeding to another ML.
- Shown normal instances during training so it can recognise.  
When it sees a new instance it can tell whether it looks like a normal one or whether it is likely an **anomaly**.
- Another task is **novelty detection**.
  - Aims to detect new instances looking different from all instances in the training set.

This requires having a very clean training set devoid of any instance that you would like the algorithm to detect.



## Unsupervised learning

- For example if you have thousands of pictures of dogs:
  - % of these pictures are Chihuahuas then **novelty detection** should not treat new pictures of Chihuahuas as novelties.
  - Whereas **anomaly detection** may consider these dogs as so rare and so different from other dogs and classify them as anomalies.
- Finally another common unsupervised task is **association rule learning** digging into big data and discover interesting relations between attributes.
- For example suppose you own a supermarket.
  - Running association rule on logs reveal people who purchase barbecue sauce and potato chips also tend to buy steak,
  - Thus you may want to place these items close to one another.



## Semi-supervised Learning

- Labelling data is usually **time consuming** and **costly**.
  - Often have plenty of unlabelled and few labelled instances.
- Some algorithms can deal with data that's partially labelled.
  - This is called semi-supervised learning [19].
- Some photo-hosting services are good examples of this.
  - Once images are uploaded it recognises **person A** appears on many photos and will categorise them as a class.
  - This is the unsupervised part of the algorithm (clustering).
  - All the system needs is for you to tell it who these people are.



Most semi-supervised learning algorithms are combinations of unsupervised and supervised,

i.e., a clustering algorithm may be used to group similar instances together and then every unlabelled instance can be labelled with the most common label in its cluster,

Once the whole dataset is labelled it is possible to use any supervised learning algorithm.



## Self-Supervised Learning

- Involves actually generating a fully labelled dataset from a fully unlabelled one.
- Once the whole dataset is labelled any supervised learning algorithm can be used.
- i.e., if you have a large dataset of unlabelled images you can randomly mask a small part of each image and then train a model to recover the original image.



## Self-Supervised Learning

- During training the masked images are used as the inputs to the model and the original images are used as the labels.
- The resulting model may be quite useful in itself.
  - i.e., to repair damaged images or to erase unwanted objects from pictures
- Generally a model trained using self-supervised learning is **not the final goal**,
- You'll usually want to tweak and fine-tune the model for a slightly different task.
  - One that you actually care about.



## Self-Supervised Learning

- For example suppose you need a pet classification model.
  - Given a picture of a pet it will tell you what species it belongs.
- If you have a dataset of unlabelled photos you can start by training an image repairing model using self-supervised learning.
- Once performing, it should be able to distinguish different pet species.
  - Repairing masked cat image it must know not to add a dog.
- If model's architecture allows, it is possible to tweak the model so that it predicts pet species instead of repairing images.
- Final is to fine-tune the model on a labelled dataset,
  - Model already knows what cats and dogs look like.
  - Only needed so the model can learn the mapping between the species it already knows and the labels we expect from it.



## Self-Supervised Learning

- For example suppose you need a pet classification model.
  - Given a picture of a pet it will tell you what species it belongs.
- If you have a dataset of unlabelled photos you can start by training

Transferring knowledge from one task to another is called **transfer learning** and it's one of the most important techniques in machine learning today especially when using deep neural networks

that it predicts pet species instead of repairing images.

- Final is to fine-tune the model on a labelled dataset,
  - Model already knows what cats and dogs look like.
  - Only needed so the model can learn the mapping between the species it already knows and the labels we expect from it.



## Self-Supervised Learning

- Some consider self-supervised learning to be a part of unsupervised learning as it deals with **fully unlabelled** datasets,
- But self-supervised learning uses generated labels during training so it's closer to supervised learning.
- And the term **unsupervised learning** is generally used when dealing with tasks like clustering dimensionality reduction or anomaly detection.
- whereas self-supervised learning focuses on the same tasks as supervised learning mainly classification and regression.
- In short it's best to treat self-supervised learning as its own category.



## Reinforcement learning

- Reinforcement learning is a very different problem
- The learning method called an **agent** in this context
  - can observe the environment,
  - select and perform actions,
  - get rewards in return.
    - or penalties in the form of negative rewards
- It must then learn by itself what is the best strategy called a **policy** to get the most reward over time



## Reinforcement learning

- A policy defines what action the agent should choose when given a situation.
- i.e., robots implement reinforcement learning to learn how to walk.
- AlphaGo is also a good example of reinforcement learning.
  - it made the headlines in when it beat Ke Jie the number one ranked player in the world at the time at the game of Go.
  - It learned its winning policy by analysing millions of games and then playing many games against itself.

Learning was turned off during the games against the champion AlphaGo was just applying the policy it had learned.

- This is called offline learning.



- Another criterion used to classify machine learning systems is whether or not the system can learn incrementally from a stream of incoming data.
- The methods are:
  - Batch learning,
  - Online learning.



## Batch Learning

- The system is incapable of learning incrementally.
  - it must be trained using all the available data.
- This takes a lot of time and computing resources so it is typically done offline.
- First the system is trained and then it is launched into production and runs without learning anymore.
  - It just applies what it has learned.
- This is also called **offline learning**.



## Batch Learning

- Unfortunately a model's performance tends to decay slowly over time as the world continues to evolve while the model remains unchanged.
  - This phenomenon is often called model rot or data drift.
- The solution is to **regularly retrain the model** on up-to-date data.
- How often you need to do that depends on the use case.
  - if model classifies pictures of cats and dogs performance will decay very slowly.
  - if model deals with financial market then it is likely to decay quite fast.



## Batch Learning

- Unfortunately a model's performance tends to decay slowly over time as the world continues to evolve while the model remains unchanged.
  - This phenomenon is often called model rot or data drift

Even a model trained to classify pictures of cats and dogs may need to be retrained regularly due to cameras keep changing along with image formats sharpness brightness and size ratios

- decay very slowly.
- if model deals classifies financial market then it is likely to decay quite fast.



## Batch Learning

- If you want a batch learning system to know about new data you need to train a new version of the system from scratch on the **full dataset**.
  - not just the new data but also the old data.
- Finally replacing the old model with the new one.
- Fortunately the whole process of training evaluating and launching a machine learning system can be automated fairly easily so even a batch learning system can adapt to change.

Training using the full set of data can take many hours so you would typically train a new system only every few hours or even just weekly. If your system needs to adapt to rapidly changing data then you need a more reactive solution.



## Batch Learning

- Also training needs significant computing resources.
- If you have a lot of data and you automate your system to train from scratch every day it will end up costing you a lot of money.
- If the amount of data is huge it may even be impossible to use a batch learning algorithm.
- Finally if your system needs to be able to learn autonomously and it has limited resources then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.
- A better option in all these cases is to use algorithms that are capable of learning incrementally.
  - Called **online learning**.



## Online Learning

- Trains incrementally by feeding data instances sequentially.
  - Either individually or in small groups called mini batches.
- Each learning step is fast and cheap so the system can learn about new data on the fly as it arrives.
- Useful for systems needing to adapt to change extremely rapidly.
  - such as patterns in the stock market.
- It is also a good option if you have limited computing resources.
  - i.e., if the model is trained on a mobile device.
- Additionally online learning can be used to train models on huge datasets that cannot fit in one machine's main memory.
  - this is called out-of-core learning.
- The algorithm loads part of the data runs a training step on that data and repeats the process until it has run on all of the data.



## Online Learning

- One important parameter of online learning systems is how **fast they should adapt to changing data.**
  - This is called the **learning rate**.
- Setting a high learning rate then your system will rapidly adapt to new data but it will also tend to quickly forget the old data.
- Setting a low learning rate the system will have more inertia.
  - It will learn more slowly but it will also be less sensitive to noise in the new data or to non-representative data points (outliers).



## Online Learning

- A big problem is that if **bad data** is fed to the system the system's performance will decline possibly quickly.
  - For example, bad data could come from a bug.
  - It could come from someone trying to game the system.
- To reduce this risk, monitor the system closely and promptly switch learning off (and possibly revert) if you detect a drop in performance.
- You may also want to monitor the input data and react to abnormal data for example using an anomaly detection algorithm.



- A way to categorise ML systems is by how they **generalise**.
- Most ML tasks are about making predictions.
  - This means that given a number of training examples the system needs to be able to make good predictions for examples it has never seen before.
- Having a good performance measure on the training data is good but insufficient.
  - The true goal is to perform well on new instances.
- There are two (**2**) main approaches to generalisation:
  1. Instance-based learning.
  2. Model-based learning.



## Instance-Based Learning

- The system learns the examples by heart then generalises to new cases by using a similarity measure to compare them to the learned examples.
- For example, flagging emails that are identical to known spam emails very similar to known spam emails.
- This requires a measure of similarity between two emails.
- Similarity measure between two emails could be to count the number of words they have in common.

## Model-Based Learning

- Generalise from a set of examples is to build a model of examples and then use that model to make predictions.

# Machine Learning Landscape



- For example you want to know if money makes people **happy**.
- You look at the graph below.

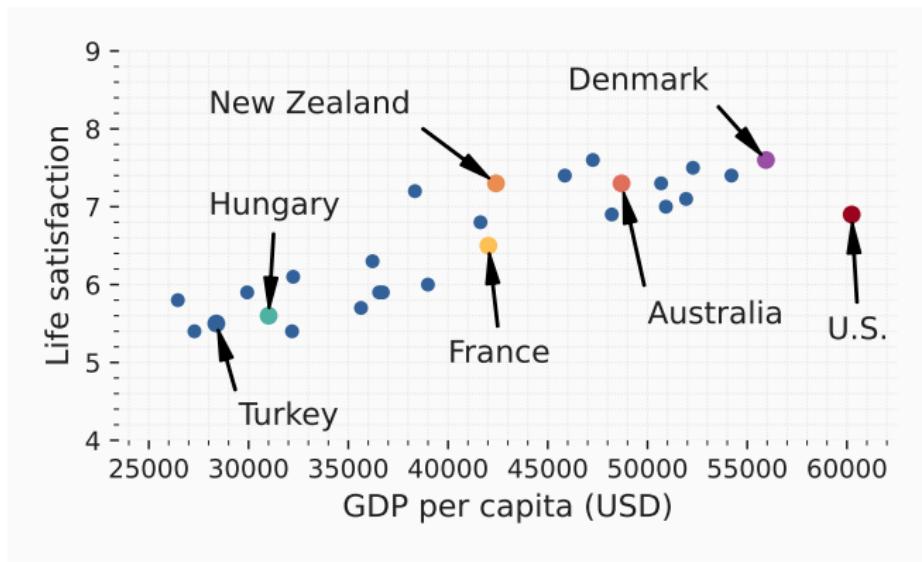


Figure 6: There seems to be something here.



- Looks like life satisfaction goes up more or less linearly as the country's GDP per capita.
- We can model life satisfaction as a linear function of GDP per capita.
- This step is called model selection you selected a linear model of life satisfaction with just one attribute, GDP per capita.

$$\text{Life Satisfaction} = \theta_0 + \theta_1 \text{GDP per Capita.}$$

- Before modelling, define the parameter values  $\theta_0$ ,  $\theta_1$ .
- Which values will make your model perform best?

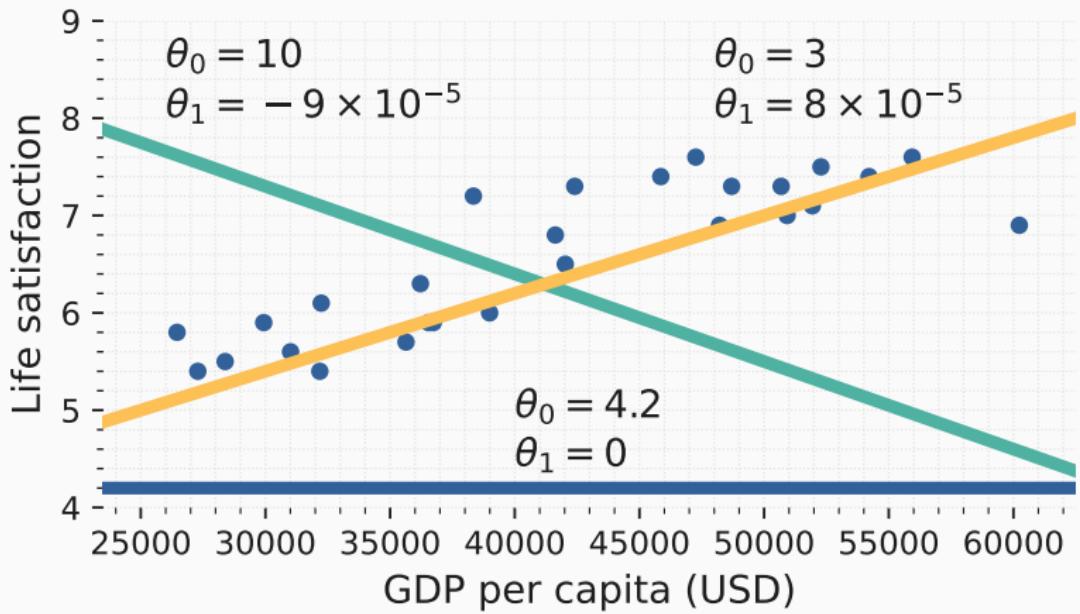


Figure 7: Possible linear models.



- To answer this question, specify a **performance measure**.
- Either define a utility function (or fitness function) measuring how good your model is or define a cost function measures how bad it is.
- For linear regression problems people typically use a cost function that measures the distance between the linear model's predictions and the training example.

The goal is to minimise this distance.

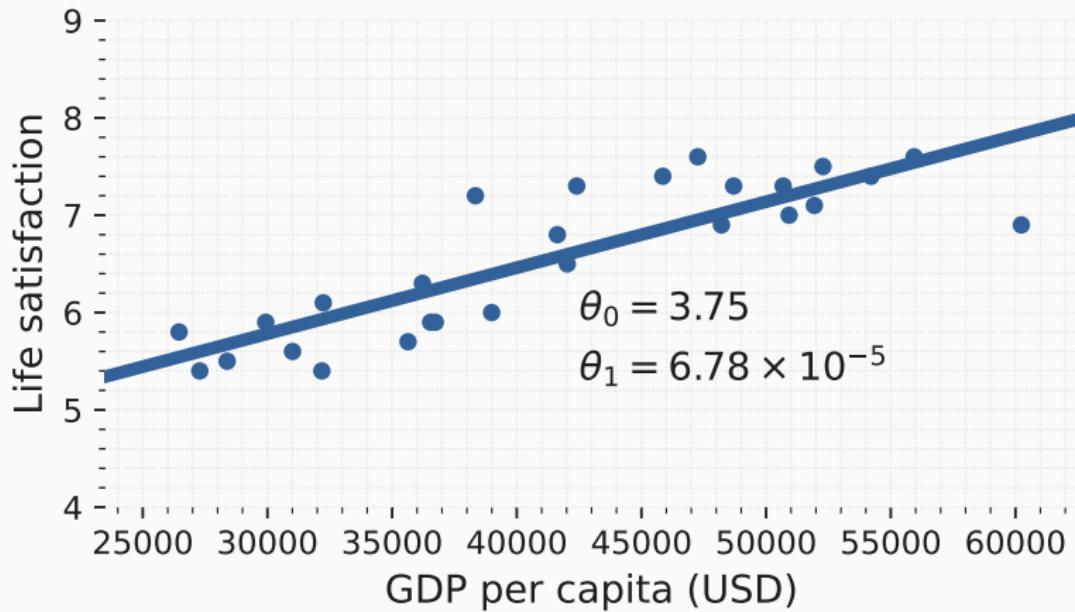


Figure 8: Best fit to the training set.



- As the main task is to select a model and train it on some data the two (2) things that can go wrong are:
  - bad model,
  - bad data.
  
- Let's start with examples of bad data.



- For a toddler to learn an apple all it takes is for you to point to an apple and say “apple”.
- Now the child is able to recognise apples in all sorts of colours and shapes.
- ML is not quite there yet.
  - It takes a lot of data for most machine learning algorithms to work properly.
- For very simple problems you need thousands of examples.
- For complex problems such as image or speech recognition you may need millions of examples.



- To generalise well, it is crucial the training data be representative of the new cases you want to generalise.
- This is true whether you use **instance-based** learning or **model-based** learning
- For example the set of countries earlier for training the linear model was not perfectly representative.
  - It did not contain any country with a GDP per capita lower than \$ 23.500,00 or higher than 62.500,00 \$

# Machine Learning Landscape

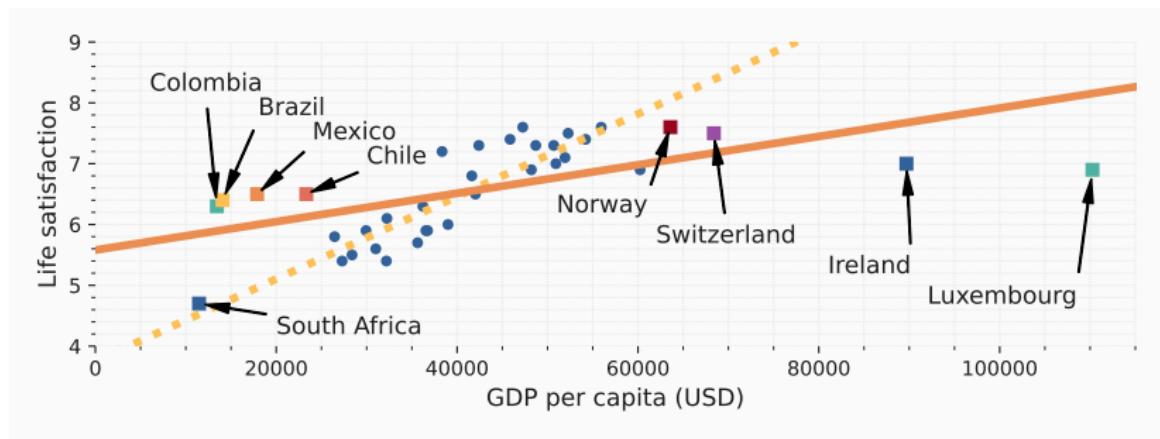


Figure 9: A more representative training sample.



- If you train a linear model on this data you get the solid line,
- while the old model is represented by the dotted line.
- Adding missing countries significantly alter the model and shows a simple linear model would not work well.

By using a non-representative training set you trained a model that is unlikely to make accurate predictions especially for very poor and very rich countries.

- It is crucial to use a training set that is representative.
- If the method is flawed, it is called **sampling bias**.



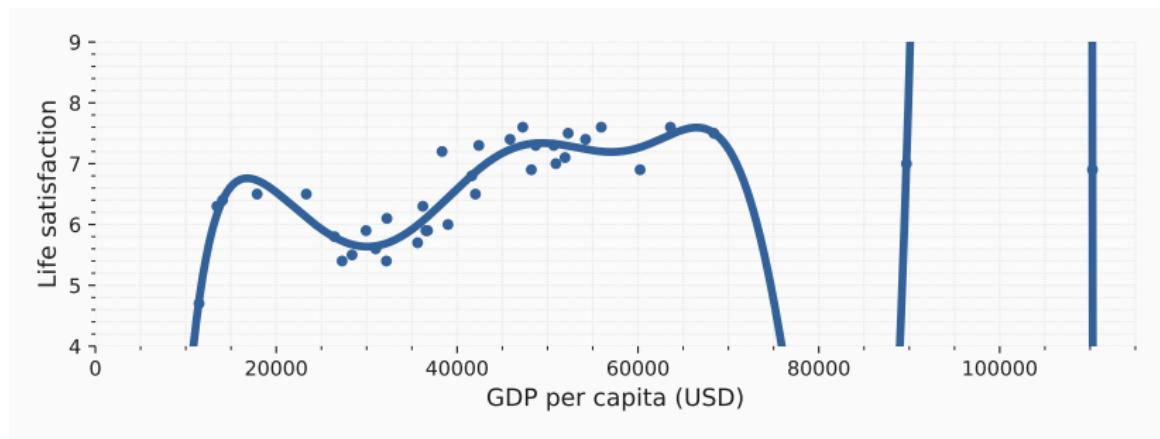
- If your training data is full of errors outliers and noise, it will make it harder for the system to detect the underlying patterns.
- It is worth cleaning up the training data.
  - If some instances are clearly outliers it may help to simply discard them or try to fix the errors manually
  - If some instances are missing a few features decide whether to:
    - ignore this attribute altogether,
    - ignore these instances,
    - fill in the missing values,
    - train one model with the feature and one model without it.



- System will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones.
- A critical part of the success of a ML project is coming up with a good set of features to train on.
- This process called feature engineering involves the following steps:
  1. **Feature selection** selecting the most useful features to train
  2. **Feature extraction** combining existing features to produce a more useful one
  3. **Creating new features** by gathering new data



- Say you are visiting a foreign country and the taxi driver rips you off.
- You might be tempted to say that all taxi drivers in that country are thieves.
- Overgeneralising is something that we humans do all too often. and unfortunately machines can fall into the same trap if we are not careful.
- In ML this is called **over-fitting**.
  - The model performs well on the training data but it does not generalise well.



**Figure 10:** Overfitting the training data.



- The figure shows an example of a high-degree polynomial life satisfaction model that strongly over-fits the training data.
- Even though it performs much better on the training data than the simple linear model would you really trust its predictions?

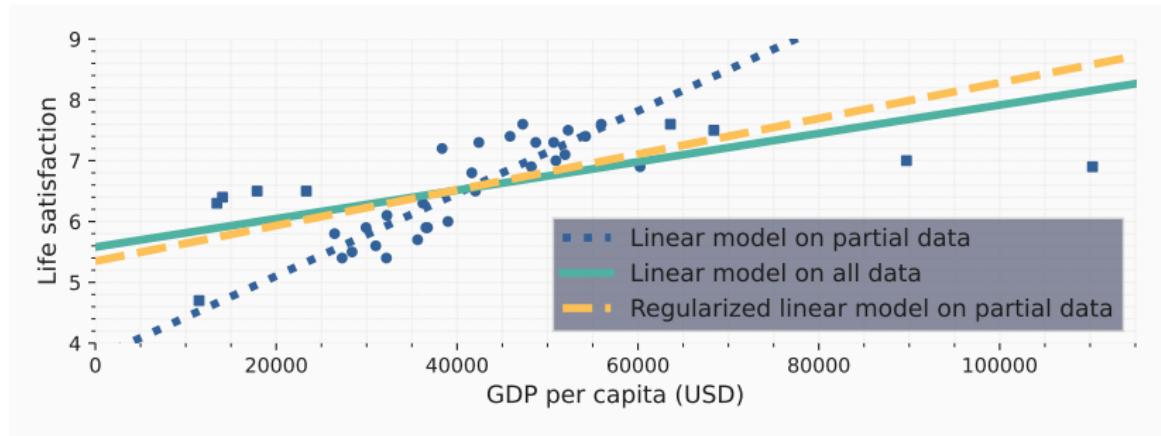


- Complex models such as DNNs can detect subtle patterns.
  - If the training set is noisy or has sampling noise then the model is likely to detect patterns in the noise itself.
- Obviously these patterns will not generalise to new instances.
- For example feeding life satisfaction model with attributes.
  - including uninformative ones such as the country's name.
- A complex model may detect patterns like the fact that all countries in the training data with a **w** in their name have a life satisfaction greater than 7.
  - New Zealand (7.3)
  - Norway (7.6)
  - Sweden (7.3)
  - Switzerland (7.5)



- Constraining a model to make it simpler and reduce the risk of over-fitting is called **regularisation**.
- For example, the linear model has two parameters  $\theta_0, \theta_1$ .
- This gives the learning algorithm two degrees of freedom.
  - Forcing  $\theta_1 = 0$  make the model have a line that can only go up or down.
  - Limiting  $\theta_1$  will keep the DoF between 1 and 2.
- The goal is to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalise well.

# Machine Learning Landscape



**Figure 11:** Regularisation reduces the risk of over-fitting.



- The amount of regularisation to apply during learning can be controlled by a hyper-parameter.
- A hyper-parameter is a parameter of a learning algorithm (not of the model).
- It is not affected by the learning algorithm itself.
- It must be set prior to training and remains constant during training.
- If you set the regularisation hyper-parameter to a very large value you will get an almost flat model (a slope close to zero) the learning algorithm will almost certainly not over-fit the training data but it will be less likely to find a good solution.



- Under-fitting is the opposite of over-fitting.
  - Occurs when your model is too simple to learn the underlying structure of the data.
- For example a linear model of life satisfaction is prone to under-fit.
  - Reality is just more complex than the model so its predictions are bound to be inaccurate even on the training examples.
- Here are the main options for fixing this problem [20]:
  - Select a more powerful model with more parameters,
  - Feed better features to the learning algorithm,
  - Reduce the constraints on the model.



- To know how well a model will generalise to new cases is to actually try it out on new cases.
- One way to do that is to put your model in production and monitor how well it performs.
- This works well but if the model is bad, users will complain.
- A better option is to split your data into two sets:
  - Training set,
  - Test set.
- Train the model using the training set and test it using the test set.
- The error rate on new cases is called the generalisation error.
- Evaluating the model on test set gives an estimate of this error.
- This value tells how well the model will perform on instances it has never seen before.
- If the training error is low, but the generalisation error is high it means that your model is over-fitting the training data.



- Evaluate a model by using a test set.
- Suppose there is hesitation between two types of models.
  - linear v. poly.
  - How can you decide between them?
- An option is to train both and compare how well they generalise using the test set



- Suppose the linear model generalises better but you want to apply some regularisation to avoid over-fitting.
- How do you choose the value of the regularisation hyper-parameter?
- An option is to train different models using different values for this hyper-parameter.
- Suppose you find the best hyper-parameter value that produces a model with the lowest generalisation error.
- You launch this model into production but unfortunately it does not perform as well as expected and produces more errors.
- What just happened?



- The problem is the generalisation error was measured multiple times on the test set and adapted the model and hyper-parameters to produce the best model for that particular set.
- This means the model is unlikely to perform as well on new data.
- A solution to this problem is called **holdout validation**.
- Simply hold out part of the training set to evaluate several candidate models and select the best one
- The new held-out set is called the **validation set**.
  1. Train multiple models with various hyper-parameters on the reduced training set.
  2. Select the model that performs best on the validation set
  3. After this holdout validation process Train the best model on the full training set.



- Having a large amount of data for training does not make it better if the data does not represent the application.
- For example, a mobile app to take pictures of flowers and automatically determine their species.
- Download millions of pictures of flowers from the web but won't be representative pictures (i.e., actually taken with the app)
- The most important rule to remember is that both the validation set and the test set must be as representative as possible.

# End-to-End ML Project

---

# Table of Contents



## Introduction

Learning Outcomes

First Steps

Places to Find Data

## Our Task

Understanding The Data

Selecting a Performance Measure

Check Assumptions

## Getting the Data

Downloading Data

Creating a Test Set

Visualising Geographical Data

Look for Correlations

## Prepare the Data for ML

Clean the Data

Handling Text and Categorical Attributes

Feature Scaling and Transformation

Transformation Pipelines

## Select and Train a Model

Train and Evaluate on the Training Set

Evaluate Your System on the Test Set

## Launch, Monitor, Maintain

End-to-End ML Project



## Learning Outcomes

- (LO1) Following a practical ML Project,
- (LO2) Analysing Data,
- (LO3) Training Models,
- (LO4) Fine-tuning Models.





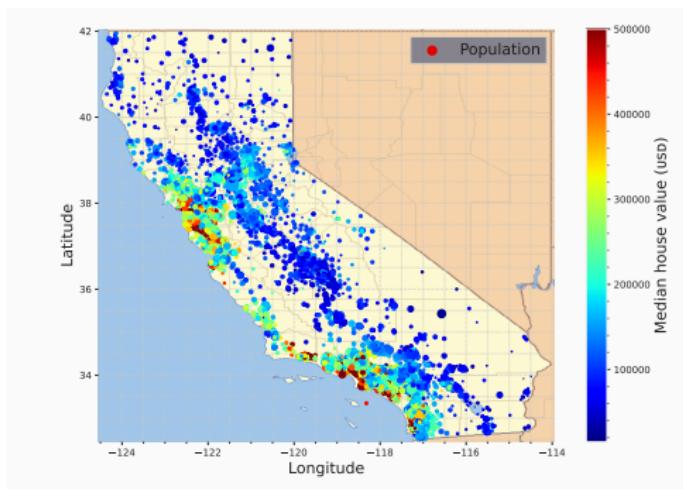
- Let's start learning ML by beginning with a project.
- We will conduct the following steps:
  1. See the big picture,
  2. Retrieve the data,
  3. Discover and visualise data,
  4. Prepare the data for analysis,
  5. Selecting a model and train it,
  6. Fine tuning the model,
  7. Present solution,



- Popular open data repositories:
  - UC Irvine Machine Learning Repository,
  - Kaggle datasets,
  - Amazon AWS datasets.
- Meta Portals:
  - Data Portals,
  - OpenDataMonitor,
  - Quandl,
- Other Sources:
  - Wikipedia's list of ML datasets.



- For this exercise we will work with [California Housing Prices](#) dataset from the [StatLib](#) repository.
- This is based on data from 1990 California census.



**Figure 12:** Housing density of the state of California.



- Our task will be to use the census data to build a model of housing prices within the state.
- The data we have includes numerous metrics such as:
  - population,
  - median income,
  - median housing price for each **block group**.

Block group is a term to describe the smallest geographical unit US Census Bureau publishes data (typical range is 600-3000 people.)



- Our first task should be to define **our goal**.
- Building the model is not the endgame.

We need to answer why we are building the model.

- You are told the model's output (a district's median housing price predictions) will be fed to another ML system, with other signals.
- This downstream system will determine whether it is worth investing in a given area of the problem or not.

Getting this right is critical, as it directly affects revenue.



- The next question is what the current solution looks like (if any).

The current situation will often give you a reference for performance, as well as insights on how to solve the problem.

- You are answered that the district housing prices are currently estimated manually by experts.
  - A team gathers up-to-date information, and when there is no median housing price, they estimate it using complex rules.
- This is costly and time-consuming, and their estimates are not great.
- In cases of finding out the actual median housing price, they often realise that their estimates were off by more than 30%.



- First task is to determine the model of training.
- Let's analyse our tasks and data:
  - It is **supervised learning**, as we have data with **labels**
  - It is **regression task**, as we are trying to predict a value.
  - It is **batch learning**, as there is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory.



- First step is to select a **performance measure**.
- A standard in regression is Root-Mean Square Error (RMSE).
  - It gives an idea on error a system makes in predictions.
  - This also gives a **higher** weight for large errors.

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2},$$

where  $m$  is the number of instances,  $\mathbf{x}^{(i)}$  is a vector of all the feature values of the  $i^{\text{th}}$  instance in the datasets, and  $y^{(i)}$  is its label.



- While RMSE is the preferred performance measure for regression tasks, in some you may prefer to use another.
- For example, suppose that there are many outlier districts.
- In that case, consider using the Mean Absolute Error (MAE).

It also called the average absolute deviation.

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

- Both RMSE and MAE are ways to measure the distance between two vectors: the vector of **predictions** and the vector of **target values**.



## Generalised Performance measure

- RMSE corresponds to the Euclidean norm:
  - It is also called the  $\ell_2$  norm.
- Computing the sum of absolutes MAE corresponds to the  $\ell_1$  norm.
- This is sometimes called the Manhattan norm because it measures the distance between two points in a city if you can only travel along orthogonal city blocks [21].

Higher norm index focuses on large values and neglects small ones.

This is why the RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.



- Our final pre-code checklist is to list and verify assumptions.
- Let's list and verify the assumptions that have been made so far.
- This can help you catch serious issues early on.
- For example, district prices your system outputs are going to be fed into a downstream machine learning system, and you assume that these prices are going to be used as such.

What if downstream system converts the prices into categories ([cheap](#), [medium](#), or [expensive](#)) and then uses those categories instead of the prices themselves?

- In this case, getting the price perfectly right is not important at all.
- your system just needs to get the category right (i.e., classification).



- Below is our function called `load_housing_data` which downloads our data and turns it into a `pandas` Dataframe.

```
def load_housing_data():
    # path to save the file
    tarball_path = Path("datasets/housing.tgz")
    # check if the path exists, if not create one

    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url =
            "https://github.com/dTmC0945/L-MCI-BSc-Data-Science-II/raw/main/da
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))
```



- With this function present load the data:

```
housing = load_housing_data()
```

- We can see the data below:

```
print(housing.head())
```

```
      longitude  latitude    ...  median_house_value  ocean_proximity
0       -122.23     37.88    ...           452600.0      NEAR BAY
1       -122.22     37.86    ...           358500.0      NEAR BAY
2       -122.24     37.85    ...           352100.0      NEAR BAY
3       -122.25     37.85    ...           341300.0      NEAR BAY
4       -122.25     37.85    ...           342200.0      NEAR BAY
[5 rows x 10 columns]
```



- Each row represents one district which have ten (10) attributes:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   longitude        20640 non-null   float64
 1   latitude         20640 non-null   float64
 2   housing_median_age 20640 non-null   float64
 3   total_rooms       20640 non-null   float64
 4   total_bedrooms    20433 non-null   float64
 5   population        20640 non-null   float64
 6   households        20640 non-null   float64
 7   median_income     20640 non-null   float64
 8   median_house_value 20640 non-null   float64
 9   ocean_proximity   20640 non-null   object  
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
None
```



- As seen previously, `.info()` method from pandas is useful to get **quick description** of data.
- There are **20,640** instances in the dataset.
- This is fairly small by ML standards, but it's perfect to get started.
- Notice that the `total_bedrooms` attribute has only **20,433** non-null values, meaning that **207** districts are missing this feature.

We need to take care of this later.



- All attributes are **numerical**, except the **ocean\_proximity** field.
- It's an object, so it could hold any kind of Python object.
- But since you loaded this data from a **.csv** file, it must text.
- When you looked at the top five rows, you probably noticed that the values in the **ocean\_proximity** column were repetitive, which means it is probably a categorical attribute.
- You can find out what categories exist and how many districts belong to each category by using the **value\_counts()** method:

```
print(housing["ocean_proximity"].value_counts())
```



- All attributes are **numerical**, except the `ocean_proximity` field.
- It's an object, so it could hold any kind of Python object.
- But since you loaded this data from a `.csv` file, it must text.

```
ocean_proximity
<1H OCEAN      9136
INLAND         6551
NEAR OCEAN     2658
NEAR BAY        2290
ISLAND          5
Name: count, dtype: int64
```

```
print(housing["ocean_proximity"].value_counts())
```



- Have a look at other fields.
  - The `.describe()` method shows a summary of numerical methods.

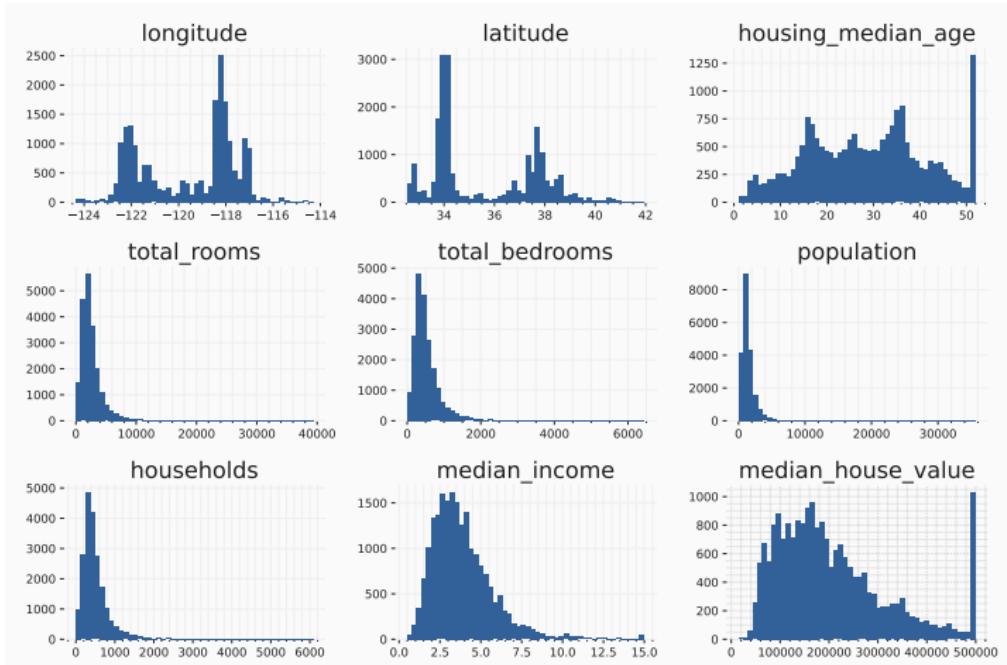
```
print(housing.describe)
```

The null values (`NaN`) are ignored.



- Another method of understanding the data is to plot a **histogram** for each numerical attribute.
- Either plot one attribute at a time, or call `.hist()` method, and it will plot a histogram for **each numerical attribute**.

```
import matplotlib.pyplot as plt  
  
housing.hist(bins=50, figsize=(12, 8))  
  
cp.store_fig("attribute-histogram-plots", close=True)
```



**Figure 13:** A histogram for each numerical attribute.



There are two (2) things of interest:

1. The median income is **not expressed in US dollars**.
  - After checking with the team, you are told that the data has been **scaled and capped** at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes.
  - The numbers represent roughly tens of thousands of dollars.
    - i.e., 3 means roughly 30,000 United States Dollar (USD).

Working with preprocessed attributes is common in ML, and it is not necessarily a problem, but you should try to understand how the data was computed.



There are a few things of interest:

2. The housing **median age** and **value** were also capped.
  - **value** may be a serious problem as it is the target attribute.
  - The ML algorithm may learn prices never go beyond that limit.
  - You need to check with your client to see if this is a problem or not.

If they tell you that they need precise predictions even beyond 500,000 USD, you have two (2) options:

- Collect proper labels for the districts whose labels were capped.
- Remove those districts from the training set



- Creating a test set is theoretically simple; pick some instances randomly, typically 20% of the dataset, and set them aside:

```
import numpy as np

# Define a function to shuffle and split data
def shuffle_and_split_data(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

- Afterwards use the function as follows:

```
train_set, test_set = shuffle_and_split_data(housing, 0.2)
```



- While this works, it's not perfect.
- If you run the program again, it will generate a different test set.
- Over time, the ML algorithm will get to see the whole dataset, which is something to avoid.
- A solution is to save the test set on the first run and then load it in subsequent runs.
- Another option is to set the **random number generator's seed**.

```
np.random.seed(42)
```



- `sklearn` has few functions to split datasets into multiple subsets.
- The simplest function is `train_test_split()`, which does pretty much the same thing as the `shuffle_and_split_data()` function defined earlier, with a couple of additional features.
- First, there is a `random_state` parameter that allows you to set the random generator seed.
- Second, you can pass it multiple datasets with an identical number of rows, and it will split them on the same indices.

This is very useful if you have a separate DataFrame for labels.

```
from sklearn.model_selection import train_test_split  
  
train_set, test_set = train_test_split(housing, test_size=0.2,  
                                      random_state=42)
```



- We taken a quick glance at the data to get a general understanding of the kind of data to manipulate.
- Now the goal is to go into a little more depth.
- First, put the test set aside to only explore the training set.
- If training set is large, sample an **exploration set**, to make manipulations easy and fast during the exploration phase.
- Now, the training set is quite small, so work directly on the full set.
- Since you're going to experiment with various transformations of the full training set, make a copy of the original so you can revert to it afterwards.

```
housing = strat_train_set.copy()
```



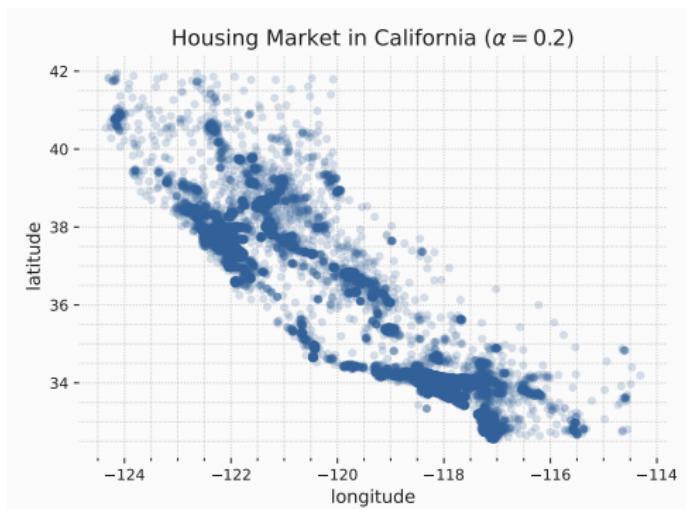
- As the dataset includes geographical information (latitude and longitude), it is a good idea to create a scatterplot of all the districts to visualise the data:



Figure 14: A geographical scatterplot of the data.



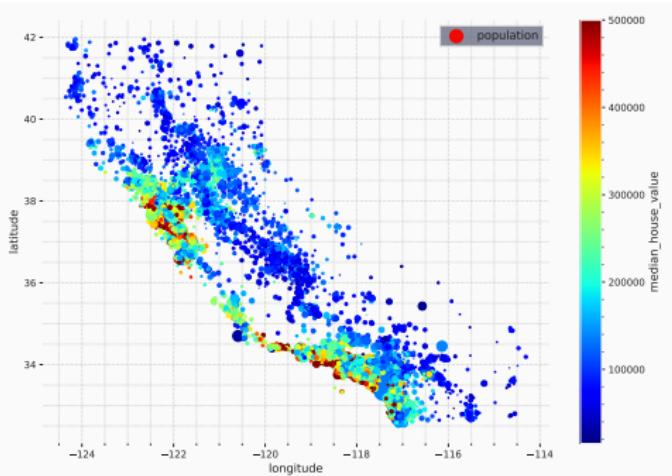
- It is hard to see any **particular pattern**.
- Setting the alpha option to **0.2** makes it much easier to visualise the places where there is a high density of data points.



**Figure 15:** A better visualisation that highlights high-density areas.



- Next, look at the housing prices. The radius of each circle represents the district's population, and the color represents the price.



**Figure 16:** California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population.



- It tells you the housing prices are very much related to the location (e.g., close to the ocean) and to the population density,
  - This makes sense.
- A **clustering algorithm** should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centres.

The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.



- As the dataset is large, the standard correlation coefficient (i.e., Pearson's) between every pair of attributes using the `.corr()` method:

```
corr_matrix = housing.corr(numeric_only=True)
corr_matrix["median_house_value"].sort_values(ascending=False)
```



- As the dataset is large, the standard correlation coefficient (i.e., Pearson's) between every pair of attributes using the `.corr()` method:

```
median_house_value      1.000000
median_income          0.688380
total_rooms            0.137455
housing_median_age     0.102175
households              0.071426
total_bedrooms          0.054635
population             -0.020153
longitude              -0.050859
latitude                -0.139584
Name: median_house_value, dtype: float64
```

```
corr_matrix = housing.corr(numeric_only=True)
corr_matrix["median_house_value"].sort_values(ascending=False)
```



- The correlation coefficient ranges from -1 to 1.
- Close to 1 means a strong positive correlation.
- Close to -1, it means a strong negative correlation.
- Close to 0 mean no linear correlation.
  
- Another way to check is to use the Pandas `scatter_matrix()` function.
  - Plots every numerical attribute against every other numerical attribute.
- As there are 11 numerical attributes, you would get 121 plots.
- Focus on a few promising attributes most correlated with the median housing value.

# End-to-End ML Project

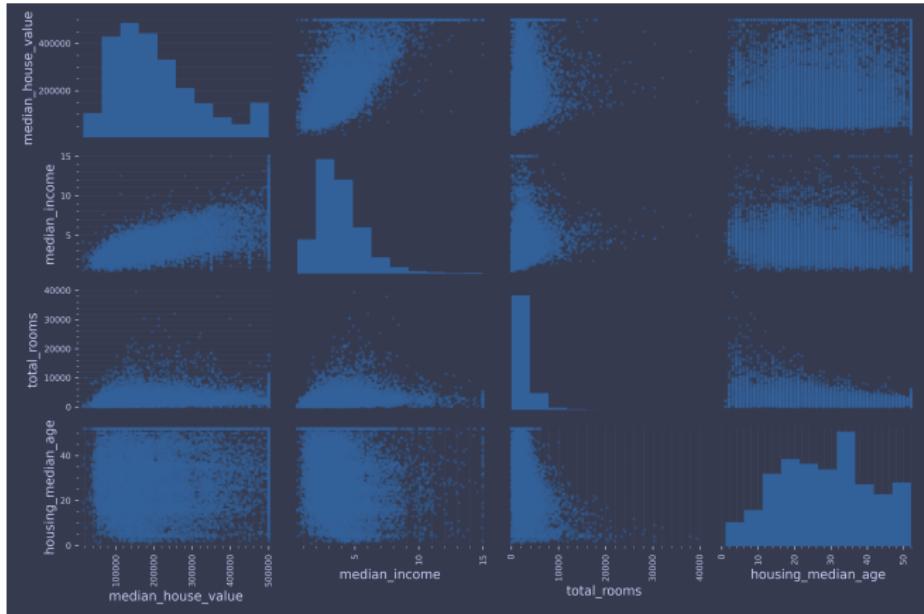


Figure 17: Correlation results of the `scatter_matrix()` function.

# End-to-End ML Project



```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]

scatter_matrix(housing[attributes], figsize=(12, 8))
cp.store_fig("scatter-matrix-plot", close=True) # extra code
```

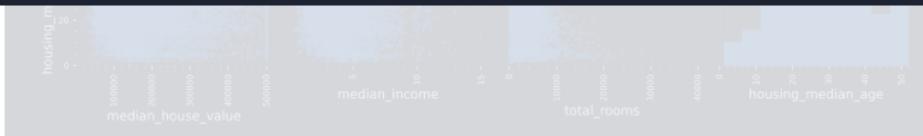


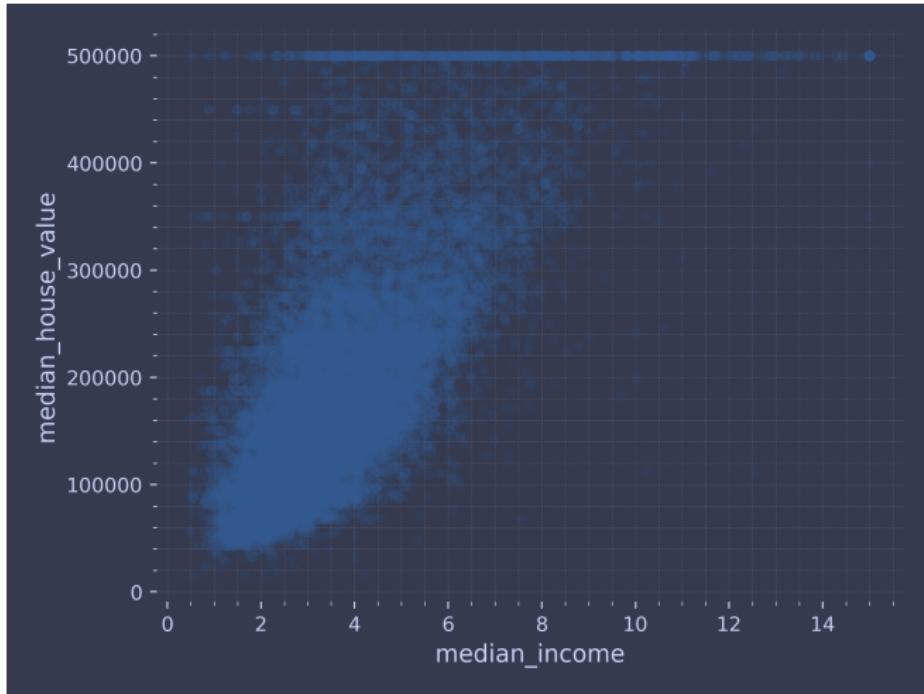
Figure 17: Correlation results of the `scatter_matrix()` function.



- The Pandas displays a histogram of each attribute.
- Looking at the correlation scatterplots, it seems like the most promising attribute to predict the `median_house_value` is the `median_income`, so you zoom in on their scatterplot.

```
housing.plot(kind="scatter",
              x="median_income",
              y="median_house_value",
              alpha=0.1, grid=True)

cp.store_fig("income-vs-house-value-scatterplot", close=True)
```



**Figure 18:** Median income versus median house value.



- This plot reveals a few things.
- The correlation is indeed quite strong;
  - see the upward trend, and the points are not too dispersed.
- Second, the price cap you noticed earlier is clearly visible as a horizontal line at 500,000 USD.
- But the plot also reveals other less obvious straight lines: a horizontal line around 450,000 USD, another around 350,000 USD, perhaps one around 280,000 USD, and a few more below that.

You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.



- Time to prepare the data for your ML algorithms.
- Instead of doing this manually, write functions for this purpose, for several good reasons:
  1. Allows you to reproduce these transformations easily on any dataset.
  2. Gradually build a library of transformation functions you can reuse in future projects.
  3. Use these functions in your live system to transform the new data before feeding it to your algorithms.
  4. Makes it possible to easily try various transformations and see which combination of transformations works best.



- First revert to a clean training set.
- Also separate the predictors and the labels, as you don't necessarily want to apply the same transformations to the predictors and the target values.

note that `.drop()` creates a copy of the data and does not affect `strat_train_set`.

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```



- Most ML algorithms cannot work with missing features, so we need to clean.
- For example, you noticed earlier the `total_bedrooms` attribute has some missing values.
- You have three options to fix this:
  1. Get rid of the corresponding districts,
  2. Get rid of the whole attribute,
  3. Set the missing value to some value (zero, mean, median)
    - This is called **imputation**.



- You can accomplish these easily using the Pandas DataFrame's methods:
  - `.dropna()`,
  - `.drop()`,
  - `.fillna()`,



```
housing_option1 = housing.copy()
housing_option1.dropna(subset=["total_bedrooms"], inplace=True)
housing_option1.loc>null_rows_idx].head()
```

```
housing_option2 = housing.copy()
housing_option2.drop("total_bedrooms", axis=1, inplace=True)
housing_option2.loc>null_rows_idx].head()
```

```
housing_option3 = housing.copy()
median = housing["total_bedrooms"].median()
housing_option3["total_bedrooms"].fillna(median, inplace=True)

housing_option3.loc>null_rows_idx].head()
```



- Go for option 3 since it is the least destructive.
- Instead of the preceding code, use `sklearn` class: `SimpleImputer`.
- The benefit is it will store the median value of each feature:
  - Makes it possible to impute missing values not only on the training set, but also on the validation set, the test set, and any new data fed to the model.
- To use it, first create a `SimpleImputer` instance, specifying to replace each attribute's missing values with the `median` of that attribute:

```
from sklearn.impute import SimpleImputer  
  
imputer = SimpleImputer(strategy="median")
```



- As the median can only be computed on numerical attributes, create a copy of the data with **only numerical attributes**.

This will exclude the text attribute `ocean_proximity`.

```
housing_num = housing.select_dtypes(include=[np.number])
```

- Now fit the imputer instance to the training data using `.fit()` method:

```
imputer.fit(housing_num)
```



- The `imputer` computed the median of each attribute and stored the result in its `statistics_` instance variable.
- Only the `total_bedrooms` attribute had missing values.
- To make sure there are no missing values in new data after the system goes live, apply the imputer to all the numerical attributes:

```
print(imputer.statistics_)
```

```
print(housing_num.median().values)
```



- The `imputer` computed the median of each attribute and stored the result in its `statistics_` instance variable.
- Only the `total_bedrooms` attribute had missing values.
- To make sure there are no missing values in new data after the system goes live, apply the imputer to all the numerical attributes:

```
print(imputer.statistics_)
```

```
print(housing_num.median().values)
```



- The imputer computed the median of each attribute and stored the result in its `statistics_` instance variable.
- Only the `total_bedrooms` attribute had missing values.
- To make sure there are no missing values in new data after the system

If you check the imputer's `statistics_` attribute, you will see that:

```
[-118.51      34.26      29.       2125.       434.       1167.       408.  
 3.5385]
```

```
print(housing_num.median().values)
```



- The `imputer` computed the median of each attribute and stored the result in its `statistics_` instance variable.
- Only the `total_bedrooms` attribute had missing values.
- To make sure there are no missing values in new data after the system goes live, apply the imputer to all the numerical attributes:

```
print(imputer.statistics_)
```

```
print(housing_num.median().values)
```



- The imputer computed the median of each attribute and stored the result in its `statistics_` instance variable.
- Only the `total_bedrooms` attribute had missing values.
- To make sure there are no missing values in new data after the system

```
[-118.51      34.26      29.       2125.       434.       1167.       408.  
 3.5385]
```

```
print(housing_num.median().values)
```



- You can use this **trained imputer** to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
# to see the name of the columns
print(imputer.feature_names_in_)
```

- Missing values can also be replaced with:
  - mean value **strategy="mean"**,
  - most frequent value **strategy="most\_frequent"**,
  - a constant value **strategy="constant"**, **fill\_value=....**

The last two strategies support non-numerical data.



- Till now we only dealt with numerical attributes, but data may also contain text attributes.
- In this dataset, there is just one: the `ocean_proximity` attribute.
- Let's look at its value for the first few instances:

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(8)
```



- Till now we only dealt with numerical attributes, but data may also contain text attributes.

In this section, we will learn how to handle categorical attributes.

	ocean_proximity
13096	NEAR BAY
14973	<1H OCEAN
3785	INLAND
14689	INLAND
20507	NEAR OCEAN
1286	INLAND
18078	<1H OCEAN
4396	NEAR BAY

```
housing_cat = housing[["ocean_proximity"]]
housing_cat.head(8)
```



- It's not arbitrary text:
- There are a limited number of possible values.
  - As each represents a **category**.
- So this attribute is a **categorical attribute**.
- ML learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers.
- For this, we can use **sklearn's OrdinalEncoder** class:

Ordinal encoding works by mapping each unique category value to a different integer. Typically, integers start at 0 and increase by 1 for each additional category.



```
from sklearn.preprocessing import OrdinalEncoder  
  
ordinal_encoder = OrdinalEncoder()  
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

- Here's what the first few encoded values in `housing_cat_encoded` look like:

```
[[3.]  
 [0.]  
 [1.]  
 [1.]  
 [4.]  
 [1.]  
 [0.]  
 [3.]]
```



- You can get the list of categories using the `categories_` instance variable.
- It is a list containing a 1D array of categories for each categorical attribute.
- In this case, a list containing a single array since there is just one categorical attribute.

```
print(ordinal_encoder.categories_)
```



- You can get the list of categories using the `categories_` instance variable.
- It is a list containing a 1D array of categories for each categorical attribute.
- In this case, it contains five categories for the attribute:

```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```

```
print(ordinal_encoder.categories_)
```



- An issue is ML algorithms will assume that two nearby values are more similar than two distant values.
- This may be fine in some cases
  - i.e., for ordered categories such as `bad`, `average`, `good`, and `excellent`
- but it is obviously not the case for the `ocean_proximity` column.
  - i.e., categories 0 and 4 are clearly more similar than categories 0 and 1.



- To fix this, create one binary attribute per category:
  - One attribute equal to 1 when the category is `<1H_OCEAN` (and 0 otherwise), another attribute equal to 1 when category is `INLAND` (and 0 otherwise), and so on.
- This is called **one-hot encoding**, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold).
- The new attributes are sometimes called **dummy attributes**.
- `sklearn` provides a `OneHotEncoder` class to convert categorical values into one-hot vectors.



- To fix this, create one binary attribute per category:
  - One attribute equal to 1 when the category is <1H\_OCEAN (and 0 otherwise), another attribute equal to 1 when category is INLAND (and 0 otherwise), and so on.

```
from sklearn.preprocessing import OneHotEncoder  
  
cat_encoder = OneHotEncoder()  
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

- sklearn provides a OneHotEncoder class to convert categorical values into one-hot vectors.



- By default, the output of a `OneHotEncoder` is a SciPy sparse matrix, instead of a `numpy` array.

Sparse matrix is an efficient representation which contains mostly zeros. Internally it only stores the nonzero values and positions.

- When a categorical attribute has thousand categories, one-hot encoding results in a matrix full of `0s` except for a single `1` per row.
- A sparse matrix is useful for this as it will save plenty of memory and speed up computations.
- You can use a sparse matrix like a normal 2D array, but to convert it to a (dense) NumPy array, call the `toarray()` method:

```
print(housing_cat_1hot.toarray())
```



- One of the most important transformations you need to apply to your data is **feature scaling**.
- With few exceptions, ML algorithms don't perform well when the input numerical attributes have different scales.
- This is the case for the housing data:
  - The total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15.

Without any scaling, most models will be biased toward ignoring the median income and focusing more on the number of rooms.



- There are two (2) common ways to get all attributes to have the same scale

## Min-Max Scaling (Normalisation)

- For each attribute, the values are shifted and rescaled so that they end up ranging from 0 to 1.
- This is performed by subtracting the min value and dividing by the difference between the min and the max.
- `sklearn` provides a transformer called `MinMaxScaler` for this.
- It has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0-1.
- i.e., neural networks work best with zero-mean inputs, so a range of -1 to 1 is preferable.



## Standardisation

- First subtracts the mean value.
  - so standardised values have a zero mean.
- Then divides the result by the standard deviation.
- This makes the values have a standard deviation equal to 1.

Standardisation does not restrict values to a specific range

- However, standardisation is much less affected by outliers.
- i.e., suppose a district has a median income equal to 100 (by mistake), instead of the usual 0-15. Min-max scaling to the 0-1 range would map this outlier down to 1 and it would crush all the other values down to 0-0.15,
- `sklearn` provides `StandardScaler` for standardisation:



- If a feature's distribution has a heavy tail both min-max scaling and standardisation will squash most values into a small range.
- ML models generally don't like this at all.
- So before you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the **distribution roughly symmetrical**.
- For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its square root (or raise the feature to a power between 0 and 1).
- If the feature has a really long and heavy tail, such as a power law distribution, then replacing the feature with its logarithm may help.



- So far we've only looked at the input features, but the target values may also need to be transformed.
- For example, if the target distribution has a heavy tail, you may choose to replace the target with its logarithm.
- But if you do, the regression model will now predict the log of the median house value, not the median house value itself.
- You will need to compute the exponential of the model's prediction if you want the predicted median house value.
- Luckily, most of `sklearn`'s transformers have an `inverse_transform()` method, making it easy to compute the inverse of their transformations.



- As you can see, there are many data transformation steps that need to be executed in the **right order**.
- Fortunately, `sklearn` provides the `Pipeline` class to help with such sequences of transformations.
- Here is a small pipeline for numerical attributes, which will first impute then scale the input features:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])

```



- Pipeline takes a list of name/estimator pairs (2-tuples) defining a sequence of steps.
- The names can be anything you like, as long as they are unique and don't contain double underscores (\_\_).

They will be useful later, when we discuss hyperparameter tuning.

- The estimators must all be transformers.
- i.e., they must have a `fit_transform()` method, except for the last one, which can be anything:
- a transformer, a predictor, or any other type of estimator.



- If you don't want to name the transformers, use the `make_pipeline()` instead
- It takes transformers as **positional arguments** and creates a **Pipeline** using the names of the transformers' classes, in lowercase and without underscores (e.g., "`SimpleImputer`"):

```
from sklearn.pipeline import make_pipeline
num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                           StandardScaler())
```



- We framed the problem, we got the data and explored it,
- Sampled a training set and a test set, and a preprocessing pipeline to
- Automatically clean up and prepare your data for machine learning algorithms.
- You are now ready to select and train a machine learning model.



- Due to all these previous steps, things are now going to be easy.
- You decide to train a very basic linear regression model to get started:

```
from sklearn.linear_model import LinearRegression  
  
lin_reg = make_pipeline(preprocessing, LinearRegression())  
print(lin_reg.fit(housing, housing_labels))
```

- Done! You now have a working linear regression model.
- The model could be of course improved, but that is a different topic for a different time.



- You are ready to evaluate the final model on the test set.
- There is nothing special about this process.
- Just get the predictors and the labels from your test set and run your `final_model` to transform the data and make predictions, then evaluate these predictions:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = mean_squared_error(y_test, final_predictions,
                                squared=False)
final_rmse
```



- Now need to get your solution ready for production.
  - i.e., cleaning up the code, writing tests and documentation.
- You can then deploy your model.
- A basic way to do this is to save the best model you trained, transfer the file to your production, and load it.
- To save the model, you can use the `joblib` library like this:

```
import joblib  
  
joblib.dump(final_model, "my_california_housing_model.pkl")
```

- Once your model is in production, you can load it and use it.
- For this, first import any custom classes and functions the model relies on (which means transferring the code to production), then load the model using `joblib` and use it to make predictions:



```
import joblib

# extra code - excluded for conciseness
from sklearn.cluster import KMeans
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.metrics.pairwise import rbf_kernel

def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

#class ClusterSimilarity(BaseEstimator, TransformerMixin):
#    [...]

final_model_reloaded =
    joblib.load("my_california_housing_model.pkl")

new_data = housing.iloc[:5] # pretend these are new districts
predictions = final_model_reloaded.predict(new_data)
print(predictions)
```



- i.e., perhaps the model will be used within a [website](#):
  - User will type in some data about a district and click Estimate Price.
  - This sends a query containing the data to the web server,
  - Which will forward it to the web application.
  - Finally your code will call the model's `.predict()` method.

You want to load the model upon server startup, rather than every time the model is used.



- Alternatively, you can wrap the model within a dedicated web service that your web application can query through.
- This makes it easier to upgrade the model to new versions **without interrupting the main application**.
- It also simplifies scaling, since you can start as many web services as needed and load-balance the requests coming from your web application across these web services.
- Moreover, it allows your web application to use any programming language, not just Python.



- But deployment is not the end of the story.
- You also need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
- It may drop very quickly, for example if a component breaks in your infrastructure, but be aware that it could also decay very slowly, which can easily go unnoticed for a long time. This is quite common because of model rot: if the model was trained with last year's data, it may not be adapted to today's data.



- So, you need to monitor your model's live performance.
- The action you take depends on the application.
- In some cases, the model's performance can be inferred from downstream metrics.
- For example, if your model is part of a recommender system and it suggests products that the users may be interested in, then it's easy to monitor the number of recommended products sold each day. If this number drops (compared to non-recommended products), then the prime suspect is the model. This may be because the data pipeline is broken, or perhaps the model needs to be retrained on fresh data (as we will discuss shortly).



- However, you may also need human analysis to assess the model's performance.
- For example, suppose you trained an image classification model to detect various product defects on a production line.
- How can you get an alert if the model's performance drops, before thousands of defective products get shipped to your clients? One solution is to send to human raters a sample of all the pictures that the model classified (especially pictures that the model wasn't so sure about).
- Depending on the task, the raters may need to be experts, or they could be nonspecialists, such as workers on a crowdsourcing platform (e.g., Amazon Mechanical Turk).
- In some applications they could even be the users themselves, responding, for example, via surveys or repurposed captchas.

# Classification

---

# Table of Contents



## Introduction

Learning Outcomes

MNIST

## Training a Binary Classifier

Five is the Loneliest Number after  
Six

## Performance Measures

Measuring Accuracy using Cross-  
Validation

Confusion Matrices

Precision and Recall

Precision/Recall Trade-off

ROC Curve

## Multi-Class Classification



## Learning Outcomes

- (LO1) Working with Binary Classifiers,
- (LO2) Defining Precision and Recall,
- (LO3) Determining good metric for goodness of model,
- (LO4) Error analysis.





- Classification is a **supervised** Machine Learning (ML) method.
  - The model tries to predict the correct label of a given input data.
- In classification, the model is fully trained using the training data, and then it is evaluated on test data before being used to perform prediction on **new unseen data**.

i.e., to predict whether an email is spam or ham.



- We will be using the Modified National Institute of Standards and Technology (MNIST) dataset, which is a set of **70,000** small images of digits **handwritten** by high school students and employees of the United States (US) Census Bureau.

Each image is labeled with the digit it represents.

This set is often called the **hello world** of ML. Whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST. Anyone who learns ML tackles this dataset sooner or later.



- `sklearn` has functions to download popular datasets (i.e., MNIST)

```
from sklearn.datasets import fetch_openml  
  
mnist = fetch_openml('mnist_784', as_frame=False)
```

- `sklearn` containing three (3) types of functions:
  - `fetch_*` downloads real-life datasets,
  - `load_*` load small toy datasets bundled with `sklearn`,
  - `make_*` generate fake datasets, useful for tests.

Generated datasets are usually returned as an  $(X, y)$  tuple containing the input data and the targets, both as `numpy` arrays.



- Other datasets are returned as `sklearn.utils.Bunch` objects,
  - Dictionaries whose entries can also be accessed as attributes.
- They generally contain the following entries:

**descr** Description of the dataset,

**data** The input data, usually as a 2D `numpy` array,

**target** The labels, usually as a 1D `numpy` array.

The `fetch_openml()` by default returns as a `pd.DataFrame` and the labels as a `pd.Series`. But the MNIST dataset contains images, and `pd.DataFrame` aren't ideal for that, so it's preferable to set `as_frame=False` to get the data as `numpy` arrays instead.



```
X, y = mnist.data, mnist.target  
print(X)
```

```
print(X.shape)
```

```
print(y.shape)
```

- There are 70,000 images, and each image has 784 features.
  - Each image is  $28 \times 28$  pixels where each feature simply represents one pixel's intensity, from 0 (white) to 255 (black).
- Grab an instance's feature vector, `reshape` it to a  $28 \times 28$  array, and display it using `matplotlib's imshow()` function:



Figure 19: Example of an MNIST image.

```
print(y[0])
```

1

5

text



5	0	4	1	9	2	1	3	1	4
3	5	3	6	1	7	2	8	6	9
4	0	9	1	1	2	4	3	2	7
3	8	6	9	0	5	6	0	7	6
1	8	7	9	3	9	8	5	9	3
3	0	7	4	9	8	0	9	4	1
4	4	6	0	4	5	6	1	0	0
1	7	1	6	3	0	2	1	1	7
9	0	2	6	7	8	3	9	0	4
6	7	4	6	8	0	7	8	3	1

Figure 20: Digits from the MNIST dataset



- Before working on the dataset, **create a test set** and set it aside.
- The MNIST dataset returned by `fetch_openml()` is already split into a training set and a test set (first 60k / last 10k):

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000],  
→ y[60000:]
```

- The training set is already shuffled which guarantees cross-validation.
- Some learning algorithms are sensitive to training instance order, and they perform poorly if they get many similar instances in a row.

Shuffling the dataset ensures that this won't happen.



- Let's simplify the problem.
- For now we only try to identify one (1) digit.
  - Let's pick the number 5.
- This 5-detector will be an example of a **binary classifier**, capable of distinguishing between just two classes:
  1. a number which is 5,
  2. a number which is **not** 5.
- First we'll create the target vectors for this classification task:

```
y_train_5 = (y_train == '5') # True for all 5s, False for all  
# other digits  
y_test_5 = (y_test == '5')
```



- Time to pick a classifier and train it.
- A good start is with a Stochastic Gradient Descent (SGD) classifier,

This classifier is capable of handling very large datasets efficiently.

- This is due to SGD deals with training instances **independently**, one at a time, which also makes SGD well suited for online learning.
- Create a **SGDClassifier** and train it on the **whole training set**.

```
from sklearn.linear_model import SGDClassifier  
  
sgd_clf = SGDClassifier(random_state=42)  
sgd_clf.fit(X_train, y_train_5)
```



- Now we can use it to detect images of the number 5:

```
sgd_clf.predict([some_digit])
```

1 [ True]

text

- The classifier guesses that this image represents a 5 (True).
- Looks like it guessed right in this particular case.
- Now, let's evaluate this model's performance.



- Evaluating a classifier is trickier than evaluating a regressor,
- A good way to evaluate a model is to use **cross-validation**.
- Use `cross_val_score()` to evaluate the `SGDClassifier` model, using **k-fold** cross-validation with three (3) folds.

k-fold means splitting the training set into k folds, then training k times, holding out a different fold each time for evaluation:

```
from sklearn.model_selection import cross_val_score  
  
cross_val_score(sgd_clf, X = X_train,y = y_train_5, cv=3,  
    scoring="accuracy")
```

1 [0.95035 0.96035 0.9604 ]

text



- We get 95% accuracy on all cross-validation folds.
- Before getting too excited, let's look at a **dummy classifier**.

It classifies every single image in the most frequent class, which in this case is the negative class (i.e., non 5).

```
cross_val_score(dummy_clf, X_train, y_train_5, cv=3,  
    scoring="accuracy")
```

1 [0.90965 0.90965 0.90965]

text



- It has over 90% accuracy.
- This is due to 10% of the images are 5s, so if you always guess an image is **not a 5**, you will be right about 90% of the time.
- This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with skewed datasets
  - i.e., when some classes are much more frequent than others

A much better way to evaluate the performance of a classifier is to look at Confusion Matrix (CM).



- The idea of CM is to count the number of times instances of class A are classified as class B, for all A/B pairs.
- To compute CM, first have a set of predictions compared to the actual targets.

You could make predictions on the test set, but it's best to keep that untouched for now.

- Instead, Use `cross_val_predict()`:

```
from sklearn.model_selection import cross_val_predict  
  
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```



- Like `cross_val_score()`, `cross_val_predict()` performs k-fold cross-validation.
  - Instead of returning the evaluation scores, it returns the predictions made on each test fold.
- This means, you get a **clean** prediction for each instance in training set.
  - i.e., making predictions on data that it never saw during training.
- Time to get CM using `confusion_matrix()`.



- Just pass:
  1. the target classes (`y_train_5`),
  2. the predicted classes (`y_train_pred`).

```
from sklearn.metrics import confusion_matrix  
  
cm = confusion_matrix(y_train_5, y_train_pred)  
cm
```

```
1 [[53892    687]                                     text  
2   [ 1891   3530]]
```



- Each row in CM represents an actual class, while each column represents a predicted class.
- First row considers non-5 images (the negative class):
  - 53,892 were correctly classified (called True Negative (TN)),
  - 687 were wrongly classified (called False Positive (FP)).
- The second row considers the images of 5s (the positive class):
  - 1,891 were wrongly classified (called False Negative (FN)),
  - 3,530 were correctly classified (called True Positive (TP)).

A perfect classifier has only true positives and true negatives, so the CM would have nonzero values only on its main diagonal.



- The CM gives a lot of information, but sometimes a more concise metric is preferred.
- An interesting one to look at is the accuracy of the positive predictions.
- This is called the **precision** of the classifier.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}},$$

where TP is the number of true positives, and FP is the number of false positives.

- An easy way to have perfect precision is to create a classifier which always makes negative predictions, except for one single positive prediction on the instance it's most confident about.

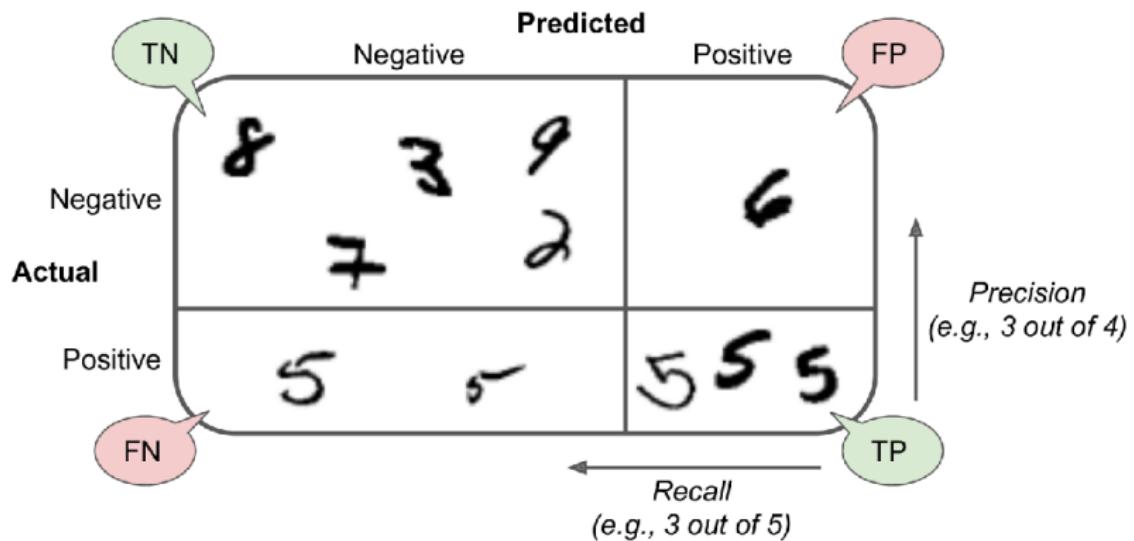


- Obviously, this would be useless, as it would ignore all but one positive instance.
- Precision is used along with another metric called **recall** (a.k.a sensitivity, True Positive Rate (TPR))
- This is the ratio of positive instances that are correctly detected by the classifier.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FN}},$$

where FN is the number of false negatives.

# Classification



**Figure 21:** An illustrated confusion matrix shows examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)



- `sklearn` provides functions to compute classifier metrics, including precision and recall:

```
from sklearn.metrics import precision_score, recall_score  
  
precision_score(y_train_5, y_train_pred) # == 3530 / (687 + 3530)
```

1 0.8370879772350012 text

```
print(recall_score(y_train_5, y_train_pred))
```

1 0.6511713705958311 text



- Our 5-detector is not good as its accuracy promised.
- When it claims an image is 5, it is correct only 83.7% of the time.
- Moreover, it only detects 65.1% of the 5s.
- It is convenient to combine precision and recall: called the  $F_1$  score,
  - Especially when you need a single metric to compare two classifiers.
- The  $F_1$  score is the **harmonic mean** of precision and recall.
- While regular mean treats all values equally, the harmonic mean gives much more weight to low values.
  - As a result, the classifier will only get a high  $F_1$  score if both recall and precision are high.

$$F_1 = \frac{TP}{TP + 0.5(FN + FP)}$$



- To compute the  $F_1$  score, simply call `f1_score()`:

```
from sklearn.metrics import f1_score  
  
print(f1_score(y_train_5, y_train_pred))
```

1 0.7325171197343847

text

- The  $F_1$  score favours classifiers having **similar** precision and recall.
- Sometime precision only matters, and sometimes it is recall.
- If you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product.



- On the other hand, suppose you train a classifier to detect shoplifters in surveillance images:
- it is probably fine if your classifier only has 30% precision as long as it has 99% recall.
- The security guards will get a few false alerts, but almost all shoplifters will get caught.

You can't have it both ways: increasing precision reduces recall, and vice versa. This is called the precision/recall trade-off.



- Let's look at how `SGDClassifier` makes its classification decisions.
- For each instance, it computes a score based on a `decision function`.
  - If score is greater than a threshold, it assigns the instance to the **positive** class;
  - Otherwise it assigns it to the **negative** class.

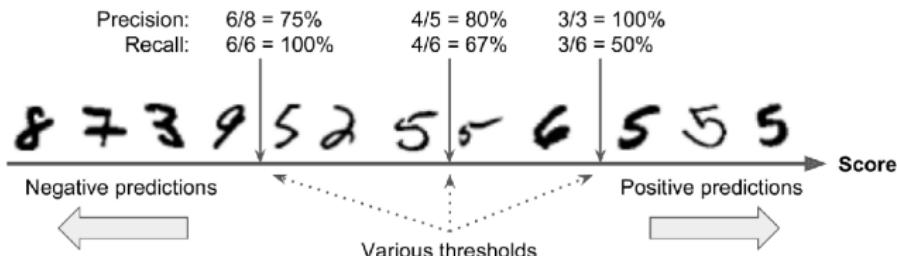


Figure 22: The precision/recall trade-off.



- Figure shows a few digits positioned from the lowest score on left to the highest score on right.
- Suppose the decision threshold is positioned at the central arrow:
  - you will find 4 true positives (actual 5s) on the right of that threshold,
  - and 1 false positive (actually a 6)
- Therefore, with that threshold, the precision is 80% (4 out of 5).
  - But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6).
- Raising the threshold, the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%.
- Conversely, lowering the threshold increases recall and reduces precision.



- `sklearn` does not let you set the threshold directly,
  - But gives access to decision scores it uses to make predictions.
- Instead of calling the `predict()` method, call its `decision_function()` method, which returns a score for each instance, and use any threshold you want to make predictions based on those scores:

```
y_scores = sgd_clf.decision_function([some_digit])
print(y_scores)
```

1 [2164.22030239]

text



- `sklearn` does not let you set the threshold directly,
  - But gives access to decision scores it uses to make predictions.
- Instead of calling the `predict()` method, call its `decision_function()` method, which returns a score for each instance, and use any threshold you want to make predictions based on those scores:

```
threshold = 0
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

1

[ True]

text



- `SGDClassifier` uses a threshold equal to `0`, so previous code returns the same result as the `.predict()` (i.e., `True`).

```
threshold = 3000
y_some_digit_pred = (y_scores > threshold)
print(y_some_digit_pred)
```

1 [False]

text

- This confirms that raising the threshold decreases recall.

The image actually represents a `5`, and the classifier detects it when the threshold is `0`, but it misses it when the threshold is increased to `3,000`.

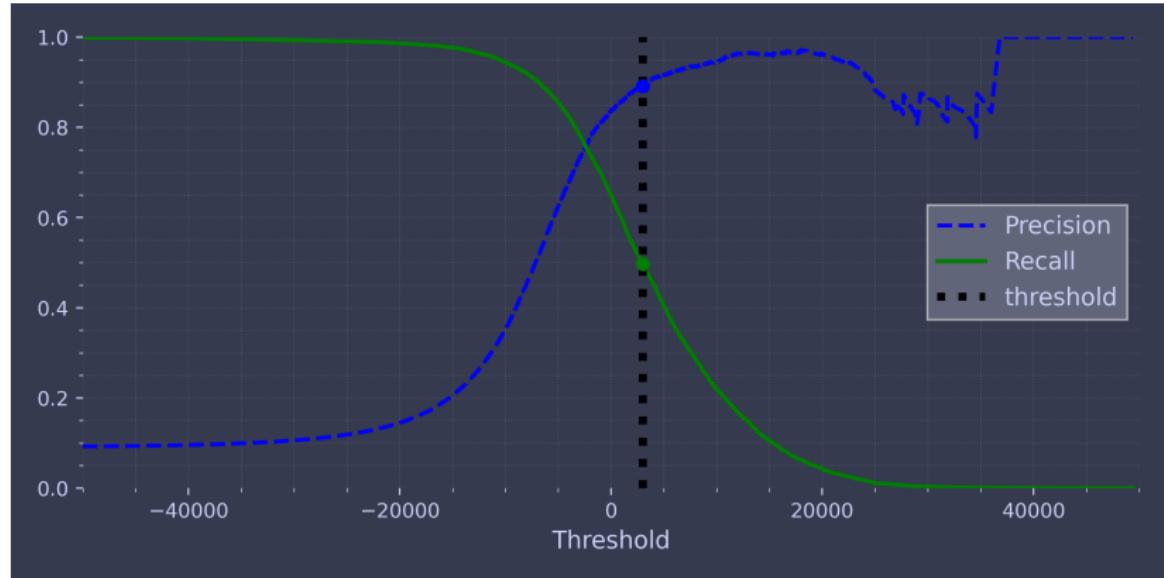


- How to decide which threshold to use?
- First, use `cross_val_predict()` to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

- Now, use the `precision_recall_curve()` to compute precision and recall for all possible thresholds (the function adds a last precision of 0 and a last recall of 1, corresponding to an infinite threshold):

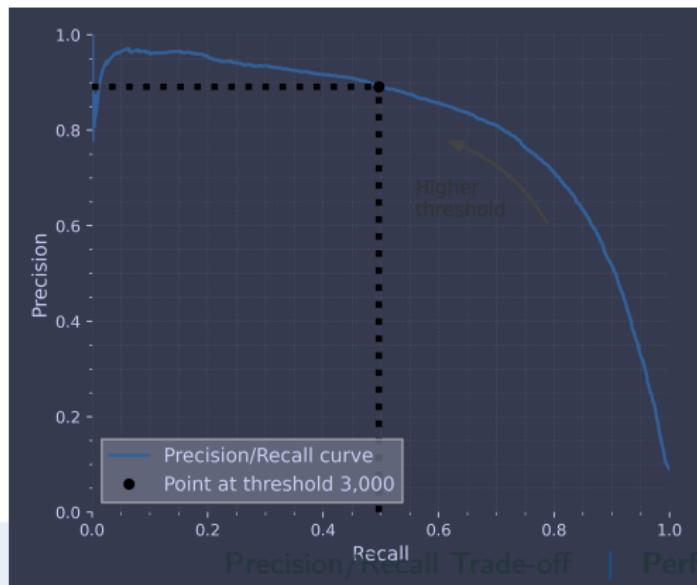
```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds = precision_recall_curve(y_train_5,
    → y_scores)
```



**Figure 23:** Precision and recall versus the decision threshold.



- At this threshold value, precision is near 90% and recall is around 50%.
- Another way to select a good precision/recall trade-off is to plot precision directly against recall.





- Precision really starts to fall sharply at around 80% recall.
- You will probably want to select a precision/recall trade-off just before the drop—for example, at around 60% recall.
- The choice depends on your project.
- Suppose you decide to aim for 90% precision.



- You could use the first plot to find the threshold you need to use, but that's not very precise.
- Alternatively, you can search for the lowest threshold that gives you at least 90% precision.
- For this, you can use the `numpy` array's `argmax()` method.
- This returns the first index of the maximum value, which in this case means the first `True` value:

```
idx_for_90_precision = (precisions >= 0.90).argmax()  
threshold_for_90_precision = thresholds[idx_for_90_precision]  
print(threshold_for_90_precision)
```



- To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)  
print(precision_score(y_train_5, y_train_pred_90))
```

- Let's check these predictions' precision and recall:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)  
print(precision_score(y_train_5, y_train_pred_90))
```

1 0.9000345901072293

text



- To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)  
print(precision_score(y_train_5, y_train_pred_90))
```

- Let's check these predictions' precision and recall:

```
recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)  
print(recall_at_90_precision)
```

1 0.4799852425751706

text



- We have a 90% precision classifier.
- As you can see, it is fairly easy to create a classifier with virtually any precision you want.
  - just set a high enough threshold, and you're done.

A high-precision classifier is not very useful if its recall is too low!  
For many applications, 48% recall wouldn't be great at all.

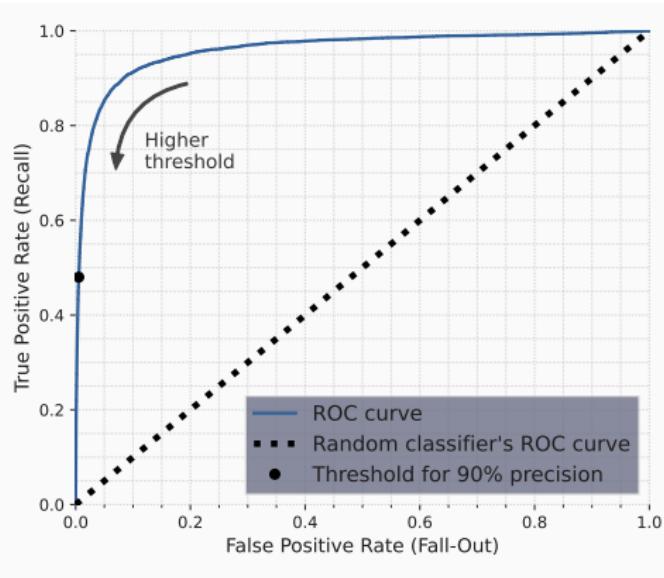


- The Receiver Operating Characteristic (ROC) curve is another common tool used with binary classifiers.
- Similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the true positive rate (another name for recall) against the False Positive Rate (FPR).
- The FPR (also called the fall-out) is the ratio of negative instances that are incorrectly classified as positive.
- It is equal to  $1 - \text{True Negative Rate (TNR)}$ , which is the ratio of negative instances that are correctly classified as negative.
- The TNR is also called specificity.
- Hence, the ROC curve plots sensitivity (recall) versus  $1 - \text{specificity}$ .



- To plot the ROC curve, you first use the `roc_curve()` function to compute the TPR and FPR for various threshold values:

```
from sklearn.metrics import roc_curve  
  
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```



**Figure 25:** A ROC curve plotting the false positive rate against the true positive rate for all possible thresholds; the black circle highlights the chosen ratio (at 90% precision and 48% recall)



- Once again there is a **trade-off**:
  - the higher the recall (TPR), the more false positives (FPR) the classifier produces.
- The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible
  - i.e, toward the top-left corner.



- One way to compare classifiers is to measure the area under the curve (AUC).
- A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to estimate the ROC AUC:

```
from sklearn.metrics import roc_auc_score  
  
print(roc_auc_score(y_train_5, y_scores))
```



- Now create a `RandomForestClassifier`, whose PR curve and  $F_1$  score we can compare to those of the `SGDClassifier`:

```
from sklearn.ensemble import RandomForestClassifier  
  
forest_clf = RandomForestClassifier(random_state=42)
```

- The `precision_recall_curve()` expects `labels` and `scores` for each instance.
  - We need to train the random forest classifier and assign a score to each instance.

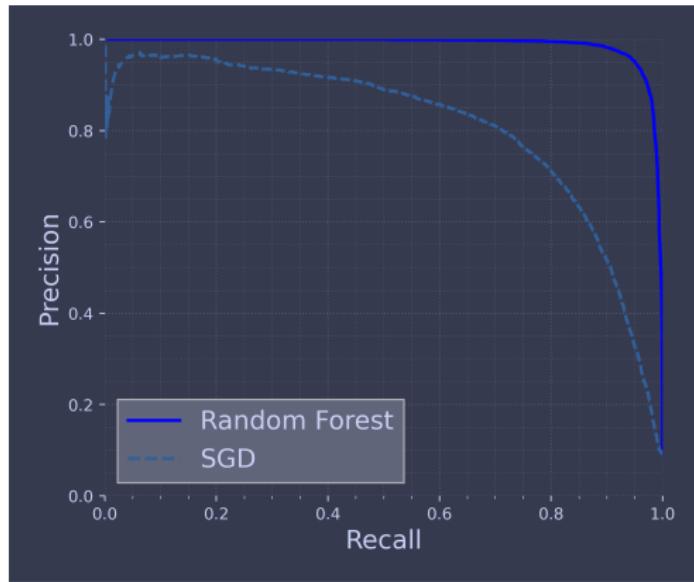
`RandomForestClassifier` has no `decision_function()`.



- Luckily, it has `.predict_proba()`, returning class probabilities for each instance.
  - Just use the probability of the positive class as a score.
- Call `cross_val_predict()` to train `RandomForestClassifier` using cross-validation and make it predict class probabilities for every image:



- Time to look at the class probabilities for the first two images in the training set:
  1. The model predicts the first image is positive with 89% probability,
  2. it predicts that the second image is negative with 99% probability.
- Since each image is either positive or negative, the probabilities in each row add up to 100%.
- The 2<sup>nd</sup> column contains the estimated probabilities for the positive class, so pass them to `precision_recall_curve()`:
- Now we're ready to plot the PR curve. It is useful to plot the first PR curve as well to see how they compare



**Figure 26:** Comparing PR curves: the random forest classifier is superior to the SGD classifier because its PR curve is much closer to the top-right corner, and it has a greater AUC



- `RandomForestClassifier` PR curve is better than `SGDClassifier`
- it comes much closer to the top-right corner. Its F1 score and ROC AUC score are also significantly better:
- Try measuring the precision and recall scores: you should find about 99.1% precision and 86.6% recall. Not too bad!
- You now know how to train binary classifiers, choose the appropriate metric for your task, evaluate your classifiers using cross-validation, select the precision/recall trade-off that fits your needs, and use several metrics and curves to compare various models. You're ready to try to detect more than just the 5s.



- While binary classifiers distinguish between two (2) classes, multiclass classifiers can distinguish between more than two classes.
- Some `sklearn` classifiers (e.g., `LogisticRegression`, `RandomForestClassifier` and `GaussianNB`) are capable of handling multiple classes natively.
- Others are strictly binary classifiers (e.g., `SGDClassifier` and `SVC`).

However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.



- A way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit.
  - a 0-detector, a 1-detector, a 2-detector, and so on.
- When you want to classify an image, you get the decision score from each classifier and select the class whose classifier outputs the **highest score**.
- This is called the One v. Rest (OvR) strategy.
- Another strategy is to train a binary classifier for every pair of digits:
  - One to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on.
- This is called the One v. One (OvO) strategy.



- If there are N classes, you need to train:

$$\frac{N \times (N - 1)}{2}$$

- For the MNIST problem, this means training 45 binary classifiers.
- When you want to classify an image, you have to run the image through all MNIST classifiers and see which class wins the most duels.
- The main advantage of OvO is that each classifier only needs to be trained on the part of the training set containing the two classes that it must distinguish.



- Some algorithms (such as Support Vector Machines (SVM)) scale poorly with the size of the training set.
- For these algorithms OvO is preferred as it is faster to train many classifiers on small training sets than to train few classifiers on large training sets.
- For most binary classification algorithms, however, OvR is preferred.
- `sklearn` detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvR or OvO, depending on the algorithm.
- Let's try this with a SVM classifier using the `sklearn.svm.SVC` class.
- We'll only train on the first 2,000 images, or else it will take a very long time:



```
from sklearn.svm import SVC  
  
svm_clf = SVC(random_state=42)  
svm_clf.fit(X_train[:2000], y_train[:2000])
```

- That was easy.
- We trained the SVM using the original target classes from 0 to 9 (`y_train`), instead of the 5-versus-the-rest target classes (`y_train_5`).
- As there are 10 classes (i.e., more than 2), `sklearn` used the OvO strategy and trained 45 binary classifiers.
- Now let's make a prediction on an image:

```
print(svm_clf.predict([some_digit]))
```



1 ['5']

text

- That's correct.
- The code actually made 45 predictions.
  - One per pair of classes,
- It then selected the class that won the most duels.
- Calling `.decision_function()` will see that it returns 10 scores per instance: one per class.
- Each class gets a score equal to the number of won duels plus or minus a small tweak ( $\max \pm 0.33$ ) to break ties, based on the classifier scores:

```
some_digit_scores = svm_clf.decision_function([some_digit])
print(some_digit_scores.round(2))
```



```
1 [[ 3.79  0.73  6.06  8.3   -0.29  9.3    1.75  2.77  7.21 text
   ↵  4.82]]
```

- The highest score is 9.3, and it's indeed it's the class 5:
- When a classifier is trained, it stores the list of target classes in `classes_`, ordered by value.
- In the case of MNIST, the index of each class in the `classes_` array conveniently matches the class itself.
  - i.e., the class at index 5 happens to be class '5'.

```
print(svm_clf.classes_)
```



- To force `sklearn` to use OvO or OvR, use `OneVsOneClassifier` or `OneVsRestClassifier` classes.
- Simply create an instance and pass a classifier to its constructor (it doesn't even have to be a binary classifier).
- For example, this code creates a multiclass classifier using the OvR strategy, based on an SVM:

```
from sklearn.multiclass import OneVsRestClassifier  
  
ovr_clf = OneVsRestClassifier(SVC(random_state=42))  
ovr_clf.fit(X_train[:2000], y_train[:2000])
```



- Let's make a prediction, and check the number of trained classifiers:

```
print(ovr_clf.predict([some_digit]))  
print(len(ovr_clf.estimators_))
```

```
1      ['5']                                         text  
2      10
```

- Training an `SGDClassifier` on a multiclass dataset and using it to make predictions is just as easy:

```
sgd_clf = SGDClassifier(random_state=42)  
sgd_clf.fit(X_train, y_train)  
print(sgd_clf.predict([some_digit]))
```



```
sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train)
print(sgd_clf.predict([some_digit]))
```

- That's incorrect.
- Prediction errors do happen!
- This time `sklearn` used the OvR strategy under the hood: as there are 10 classes, it trained 10 binary classifiers.
- `.decision_function()` now returns one value per class.
- Let's look at the scores that the `SGDClassifier` assigned:

```
sgd_clf.decision_function([some_digit]).round()
```



```
1 [[-31893. -34420. -9531.  1824. -22320. -1386. -26189_text  
  ↳ -16148. -4604. -12051.]]
```

- The classifier is not very confident about its prediction:
- almost all scores are very negative, while class 3 has a score of **+1,824**, and class 5 is not too far behind at **-1,386**.
- You'll want to evaluate this classifier on more than one image.
- Since there are roughly the same number of images in each class, the accuracy metric is fine.



- Use the `cross_val_score()` function to evaluate the model:

```
cross_val_score(sgd_clf, X_train, y_train, cv=3,  
    → scoring="accuracy")
```

```
1 [0.87365 0.85835 0.8689 ]
```

text

## Training Models

---

# Table of Contents



## Introduction

## Linear Regression

The Normal Equation

Computational Complexity

## Gradient Descent

Introduction

Batch Gradient Descent

Stochastic Gradient Descent

Mini-Batch Gradient Descent

## Polynomial Regression

A Simple Example

Learning Curves

## Regularised Linear Models

Ridge Regression

Lasso Regression

Elastic Net Regression

Right Model For the Job

Early Stopping

## Logistic Regression

Introduction

Estimating Boundaries

Training and Cost Function

Decision Boundaries

Softmax Regression



## Learning Outcomes

- (LO1) Cost Functions
- (LO2) Linear Regression,
- (LO3) Polynomial Regression,
- (LO4) Decision Boundaries.





- Up to now, we treated the models as **black boxes**.

Knowing how things work can help you deciding the **right** model, training algorithm, and hyper-parameters for your application.

- There are two (2) different ways to train a *Linear Regression Model*:
  1. A direct closed-form equation, directly computing the model parameters best fitting the model to the training set.
  2. An iterative optimisation approach called Gradient Descent (GD) which gradually tweaks the model parameters to **minimise the cost function** over the training set.

Second method eventually converges to the same set of parameters as the first method.



- Next, we will look at Polynomial Regression, a more complex model which can fit **non-linear datasets**.
- As this model has more parameters than Linear Regression (LR), it is more prone to over-fitting the training data.
- To avoid it, we will look at how to detect whether or not this is the case using learning curves, and then we will look at several regularisation techniques which can reduce the risk of over-fitting the training set.



- Previously, we looked at a simple regression model of life satisfaction:

$$\text{life satisfaction} = \theta_0 + \theta_1 \times \text{GDP per capita}$$

- This is just a linear function of the input feature `GDP per capita`.
- $\theta_0$  and  $\theta_1$  are the model's parameters.

a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term<sup>1</sup>.

---

<sup>1</sup>intercept term



- Looking at a simple equation:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

where:

$\hat{y}$  predicted value,

$n$  the number of features

$x_i$  the  $i^{\text{th}}$  feature,

$\theta_j$  the  $j^{\text{th}}$  model parameter



- Of course, this can be written more terse in a vectorised form:

$$\hat{y} = h(\mathbf{x}) = \theta \cdot \mathbf{x}$$

where:

- $h_{\theta}$  The hypothesis function, using the model parameters  $\theta$ .
  - $\theta$  The model's parameter vector, containing the bias term  $\theta_0$  and the feature weights  $\theta_1$  to  $\theta_n$ .
  - $\mathbf{x}$  the instance feature vector, containing [0] to [n] with [0] always being equal to 1.
- $\theta \cdot \mathbf{x}$  is the dot product of the vectors  $\theta$  and  $\mathbf{x}$ , which equals to:

$$\theta \cdot \mathbf{x} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$



The main question is of **training**.

- This means setting its parameters so that the model best fits the training set.
- First, we need a measure of how well (or poorly) the model fits the training data.
- Previously we have looked at RMSE

to train a LR, find the value of  $\theta$  minimising RMSE.

In practice, it is simpler to minimise Mean Square Error (MSE) than RMSE and it leads to same result.



- The MSE of LR hypothesis  $h$  on a training set  $\mathbf{X}$  is calculated using

$$\text{MSE}(\mathbf{X}, h_\theta) = \frac{1}{m} \sum_i^m \left( \theta^\top \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

- The only difference is that we write  $h$  instead of just  $h$  to make it clear that the model is parameterised by the vector  $\theta$ .

To simplify notations, write  $\text{MSE}(\theta)$  instead of  $\text{MSE}(X, h)$ .



- To find the value of  $\theta$  minimising the cost function, there is a closed-form solution.

A mathematical equation that gives the result directly.

- This is called the **Normal Equation** with the following:

$$\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

where:

- the value of  $\theta$  that minimises the cost function.
- the vector of target values containing  $y^{(1)}$  to  $y^{(m)}$ .



- Time to work on some code!
- Start with generating some linear<sup>1</sup> data.

```
import numpy as np

np.random.seed(42) # to make this code example reproducible
m = 100 # number of instances
X = 2 * np.random.rand(m, 1) # column vector
y = 4 + 3 * X + np.random.randn(m, 1) # column vector
```

# Training Models

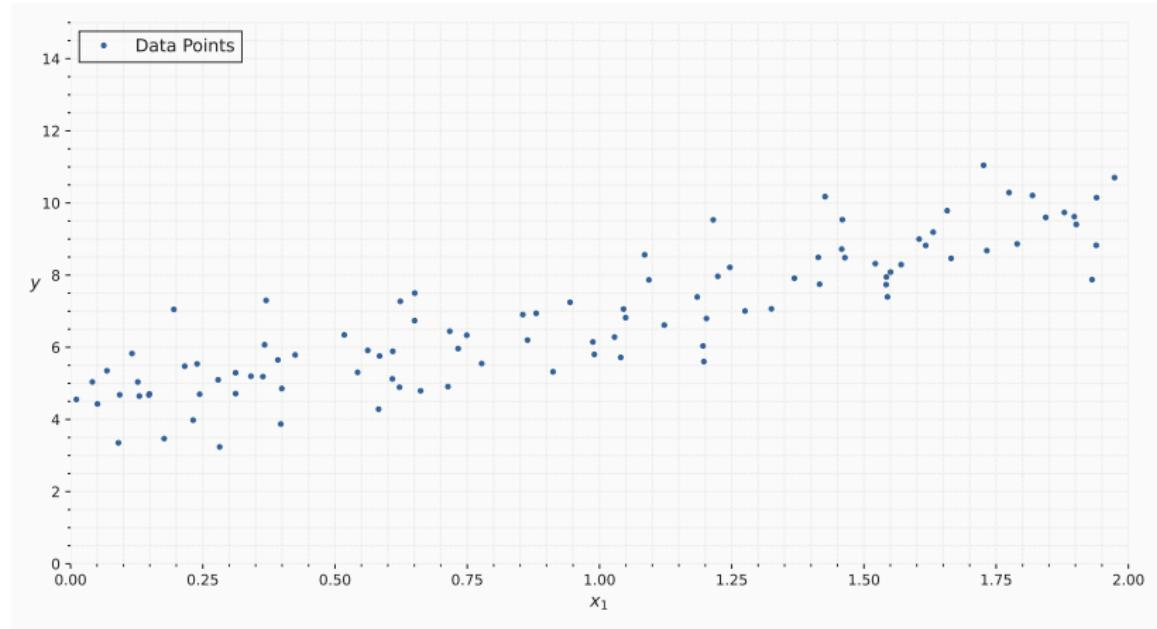


Figure 27: Randomly generated linear dataset.



- Compute  $\hat{\theta}$  using the normal equation.
- We will use the `inv()` function from `numpy's linalg` module to compute the inverse of a matrix and use `@` operator.

```
from sklearn.preprocessing import add_dummy_feature  
  
X_b = add_dummy_feature(X) # add x0 = 1 to each instance  
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

`@` performs matrix multiplication. If `A` and `B` are `numpy` arrays, then `A @ B` is equivalent to `np.matmul(A, B)`. Other libraries, support `@` operator as well. However, you cannot use `@` on pure Python arrays (i.e., lists of lists).



- The function that we used to generate the data is:

$$y = 4 + 3[1] + \mathcal{N}(\mu, \sigma^2)$$

- Let's see what the equation found:

```
print(theta_best)
```

```
[[4.21509616]
 [2.77011339]]
```

- Ideally it should be  $\theta_0 = 4$ ,  $\theta_1 = 3$  however as data has noise it is not possible to perfectly reconstruct the original values.

The smaller and noisier the dataset, the harder it gets.



- Now we can make predictions using  $\hat{\theta}$ :

```
X_new = np.array([[0], [2]])
X_new_b = add_dummy_feature(X_new) # add x0 = 1 to each instance
y_predict = X_new_b @ theta_best
print(y_predict)
```

```
[[4.21509616]
 [9.75532293]]
```

- Now we got our prediction, we can compare it with the original data.

# Training Models

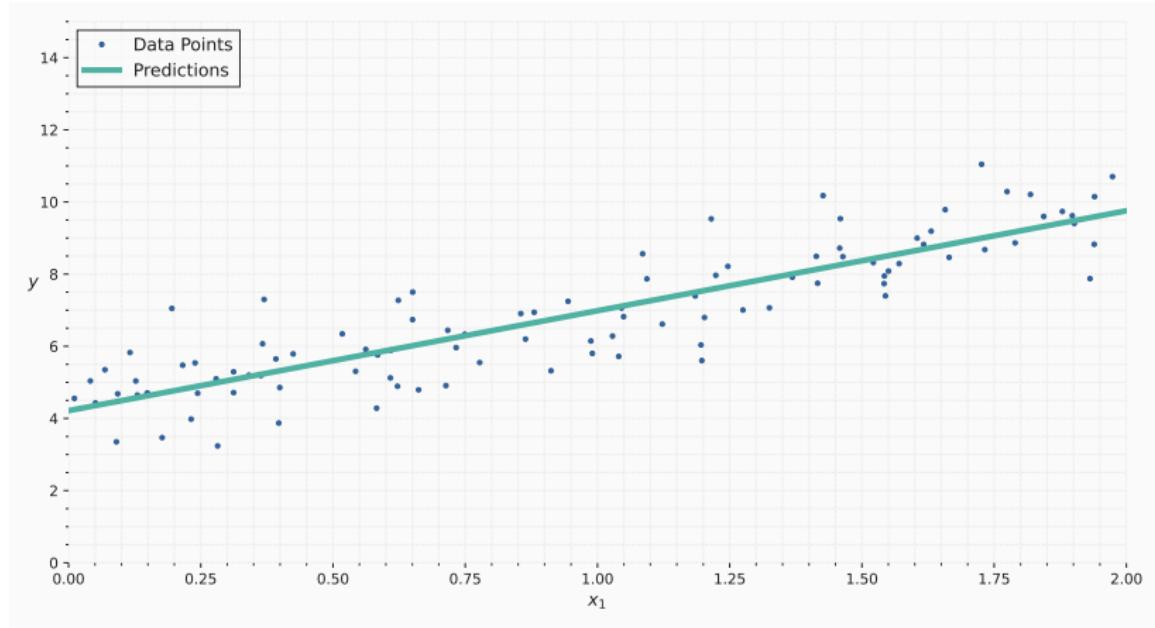


Figure 28: Linear regression model predictions.



- Performing this using `sklearn` is relatively straight forward:

```
from sklearn.linear_model import LinearRegression  
  
lin_reg = LinearRegression()  
lin_reg.fit(X, y)  
  
print("Estimated coefficients: ", lin_reg.coef_)  
print("The b0 value is: ", lin_reg.intercept_)
```

```
Estimated coefficients: [[2.77011339]]  
The b0 value is: [4.21509616]
```

`sklearn` separates the bias term (`intercept_`) from the feature weights (`coef_`).



- The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function, which you could call directly:

```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y,  
                                                 rcond=1e-6)
```

- The function calculates  $\hat{\theta} = \mathbf{X}^+ \mathbf{y}$  where  $\mathbf{X}^+$  is the **pseudo-inverse** of  $\mathbf{X}$  (a.k.a [Moore-Penrose inverse](#)). This operation is done simply by `np.linalg.pinv()`:

```
np.linalg.pinv(X_b) @ y
```



- The normal equation computes the inverse of  $\mathbf{X}^T \mathbf{X}$ .
  - This is a  $(n + 1) \times (n + 1)$  matrix, where  $n$  is the number of features.
- The computational complexity of inverting such a matrix is typically about  $\mathcal{O}(n^{2.4})$ , depending on the implementation.
- The Singular Value Decomposition (SVD) approach used by `sklearn's LinearRegression` class is about  $\mathcal{O}(n^2)$ .
- Once you have a LR model, predictions are very fast:
  - The computational complexity is **linear** with regard to:
    1. The number of instances you want to make predictions on
    2. The number of features.



- A generic optimisation algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea of GD is to tweak parameters iteratively in order to minimise a cost function.
- You start by filling  $\theta$  with random values.
  - this is called random initialisation.
- Then you improve it gradually, taking one step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum.



- An important parameter is **step size**, determined by the *learning rate* hyper-parameter.

**Too small** The algorithm will have to go through many iterations to converge, which will take a long time.

**Too high** You might jump across the minimum and end up on the other side, possibly even higher up than you were before.

Choosing a bad learning rate can make the algorithm diverge, with larger and larger values, failing to find a good solution

Not all cost functions have a **single** local minimum.

- There can be holes, ridges, plateaus, and all sorts of irregular terrains, making convergence to the minimum difficult.



- Luckily, the MSE cost function for a LR model happens to be a convex function, which means that if you pick any two points on the curve, the line segment joining them never crosses the curve [22].
- This implies that there are no local minima, just one global minimum .
- It is also a continuous function with a slope that never changes abruptly.
- These two facts have a great consequence:

GD is guaranteed to approach arbitrarily close the global minimum (if you wait long enough and if the learning rate is not too high).



- All the training data is taken into consideration to take a single step.
- We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters.
- So that's just one step of gradient descent in one epoch.

It is great for convex or relatively smooth error manifolds. In this case, we move somewhat directly towards an optimum solution.



- Let's have a quick look at it's implementation.

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances

np.random.seed(42)
theta = np.random.randn(2, 1) # randomly initialized model
# parameters
# Each iteration over the training set is called an epoch.
for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients

print(theta)
```

```
[[4.21509616]
 [2.77011339]]
```



- This is the same as the Normal Equation values.
- What if we used a different learning rate  $\eta$  ?

# Training Models

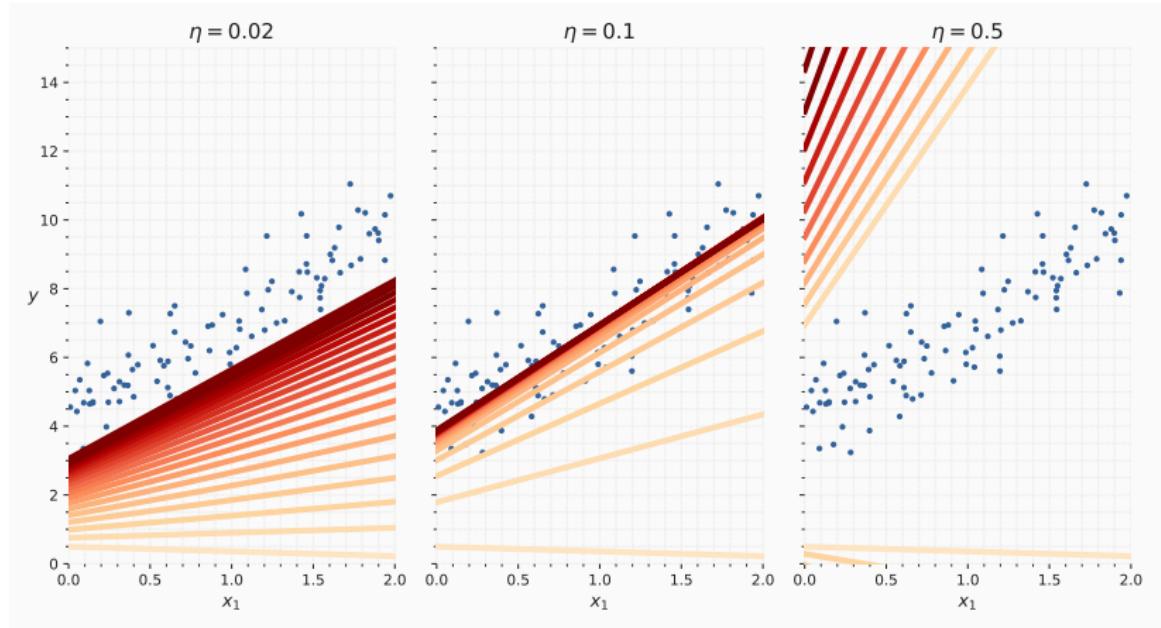


Figure 29: Gradient descent with various learning rates



- When  $\eta = 0.002$ , the learning rate is too low.
  - The algorithm will eventually reach the solution, but will take a long time.
- In the middle ( $\eta = 0.1$ ), the learning rate looks good.
  - In just a few iterations, it has already converged to the solution.
- When ( $\eta = 0.5$ ), the learning rate is too high.
  - The algorithm **diverges**, jumping all over the place and actually getting further and further away from the solution at every step.



- To find a good learning rate, you can use **grid search**.
- However, you may want to limit the number of iterations so grid search can eliminate models that take too long to converge.

## Setting Iteration Number

- Too low, it will be far away from the optimal solution when the algorithm stops.
- Too high, you will waste time while the model parameters do not change anymore.
- A simple solution is to set a very large number of iterations but to interrupt the algorithm when the gradient vector becomes tiny.
- When its norm becomes smaller than a tiny number  $\epsilon$ .



## Introduction

- Problem with Batch GD is, it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large.
- Stochastic GD picks a random instance in the training set at every step and computes the gradients based only on that single instance.
- Working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration.

Stochastic GD makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.



## Limitations

- On the other hand, due to its stochastic (i.e., random) nature, Stochastic GD is less regular than Batch GD.
  - Instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average.
- Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down.

Once Stochastic GD stops, the final parameter values are good, but not optimal.



- If the cost function is very irregular, this can actually help the algorithm jump out of **local minima**.

Stochastic GD has a better chance of finding the global minimum than Batch GD Descent does.

- Randomness can be a good strategy to escape from local optima.
  - But it also means that the algorithm can never settle at the minimum.



- One solution is to **gradually reduce the learning rate**.
- Steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.
- The function determining the learning rate at each iteration is called the **learning schedule** which can go two (2) ways:

**Too quick** You may get stuck in a local minimum, or even end up frozen halfway to the minimum.

**Too slow** You may jump around the minimum for a long time and end up with a sub-optimal solution if you halt training too early.

# Training Models

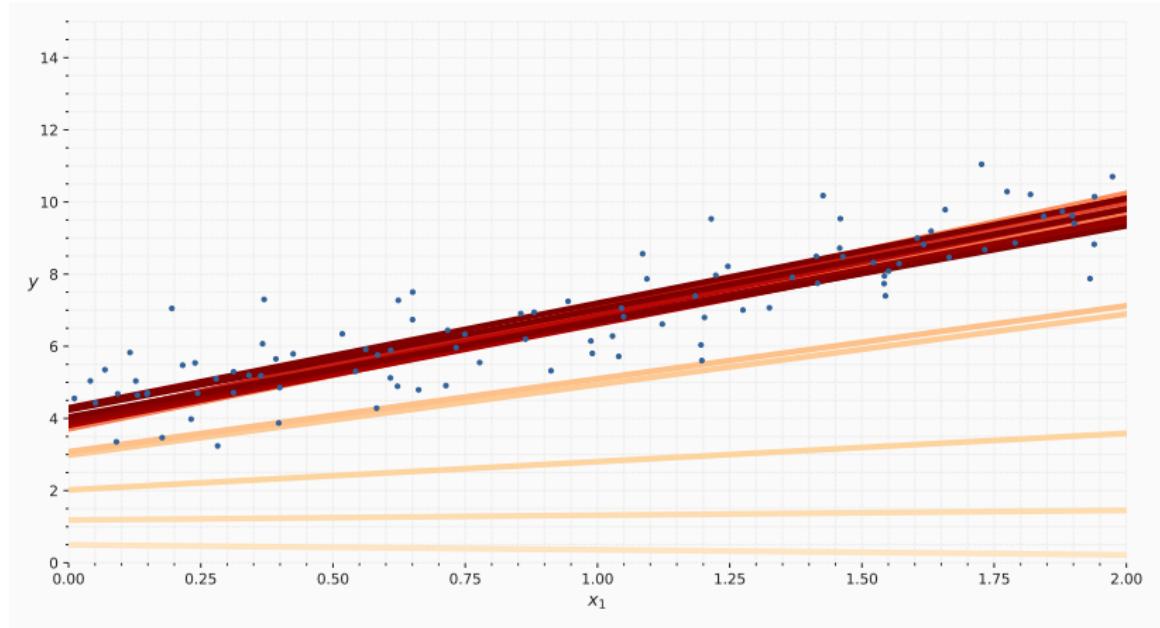


Figure 30: The first 20 steps of stochastic gradient descent



- At each step, instead of computing the gradients based on the full training set (i.e., Batch GD) or based on just one instance (i.e., Stochastic GD), mini-batch GD computes the gradients on small random sets of instances called **mini-batches**.
- The main advantage of mini-batch GD over Stochastic GD is that you can get a performance boost from hardware optimisation of matrix operations, especially when using GPUs.

# Training Models

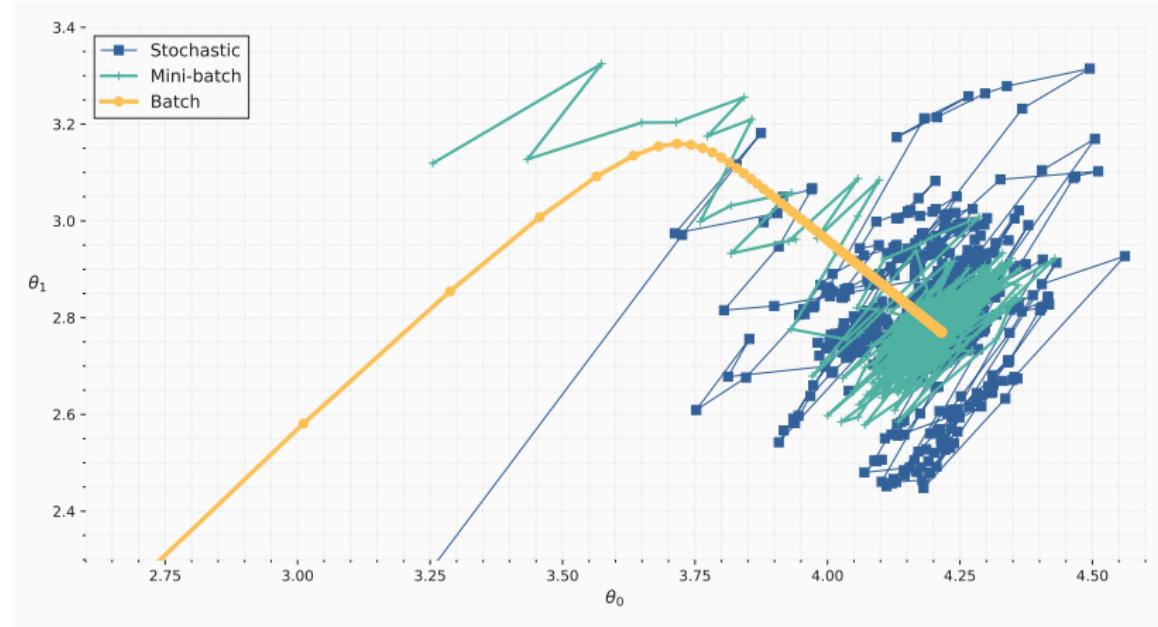


Figure 31: Gradient descent paths in parameter space.



- Sometimes data comes in non-linear fashion.
- You can use a linear model to fit nonlinear data.
- A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features.

This technique is called Polynomial Regression (PR).



- Let's generate a quadratic polynomial with added noise.
- This quadratics is in the form:

$$y = ax^2 + bx + c$$

```
np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m, 1)
```

# Training Models

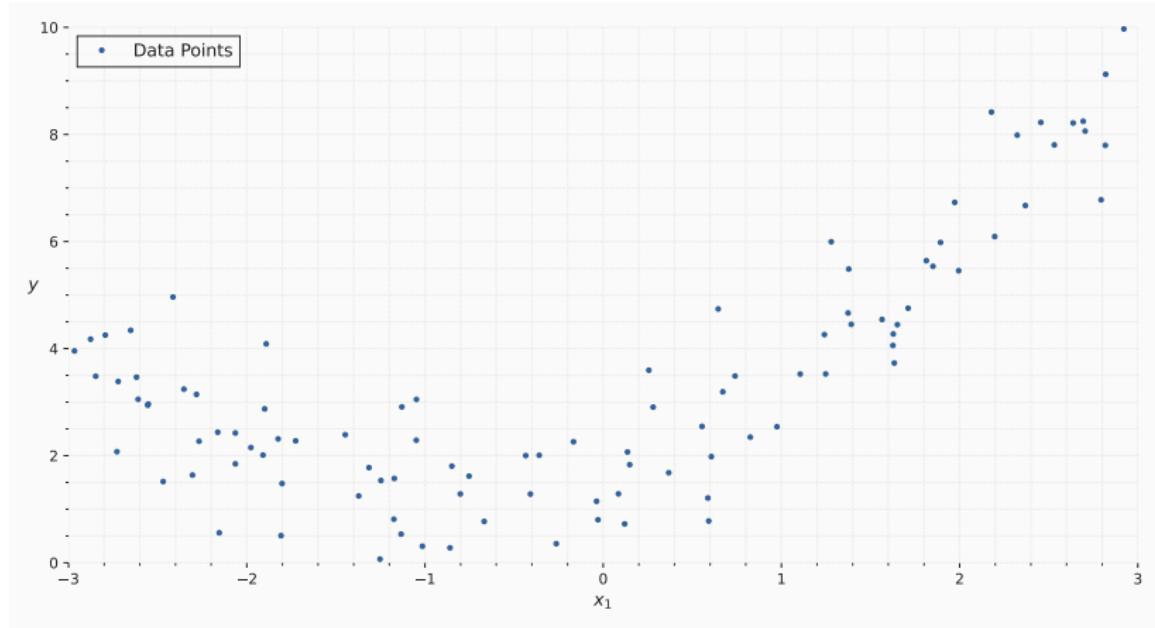


Figure 32: Generated non-linear noise.



- As can be seen, a linear line will never properly fit the data.
- We can use `sklearn`'s class `PolynomialFeatures` which can transform the training data.

```
from sklearn.preprocessing import PolynomialFeatures  
  
poly_features = PolynomialFeatures(degree=2, include_bias=False)  
X_poly = poly_features.fit_transform(X)  
  
print(X[0])  
print(X_poly[0])
```

```
[-0.75275929]  
[-0.75275929  0.56664654]
```



- `X_poly` now contains the original feature of `X` plus the square of this feature.
- Now we can fit a `LinearRegression` model to this extended training data:

```
lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
print(lin_reg.intercept_, lin_reg.coef_)

X_new = np.linspace(-3, 3, 100).reshape(100, 1)
X_new_poly = poly_features.transform(X_new)
y_new = lin_reg.predict(X_new_poly)
```

```
[1.78134581] [[0.93366893 0.56456263]]
```

# Training Models

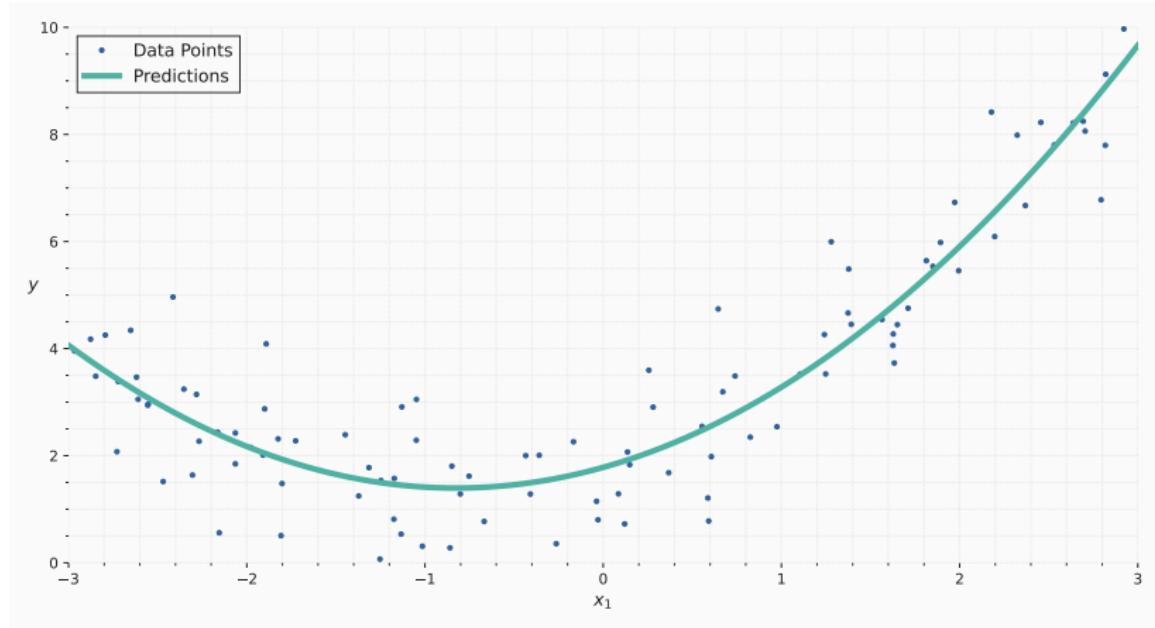


Figure 33: Polynomial regression model predictions.



- The model estimates:

$$\hat{y} = 0.56x^2 + 0.93x + 1.78,$$

- the original function was:

$$\hat{y} = 0.5x^2 + x + 2 + \mathcal{N}(\mu, \sigma^2).$$

- When there are **multiple features**, PR is capable of finding relationships between features, which is something a plain LR model cannot do.
- This is due to **PolynomialFeatures** also adding **all combinations of features** up to the given degree.
- For example, if there were two features  $a$  and  $b$ , **PolynomialFeatures** with **degree=3** would not only add the features  $a^2$ ,  $a^3$ ,  $b^2$  and  $b^3$ , but also the combinations  $ab$ ,  $A^2b$ ,  $ab^2$



- If you perform **high-degree** PR, you will likely fit the training data much better than with plain LR.
- For example, applying a **300**-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial).

Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

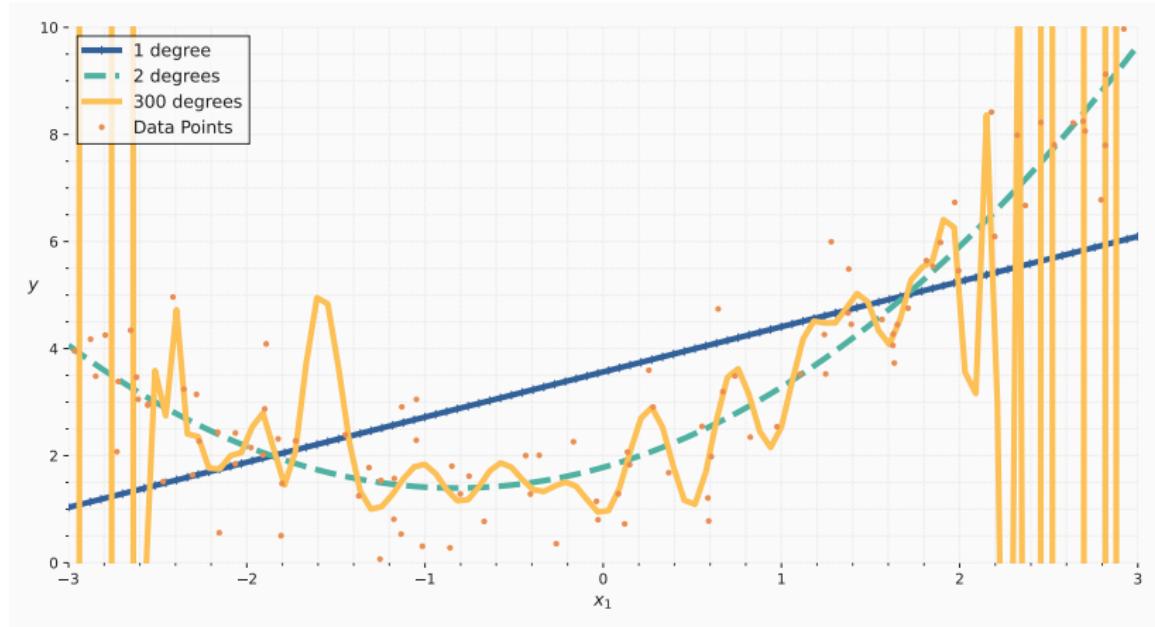


Figure 34: Comparison of different degrees of polynomial fit.



- The high-degree PR model is **severely over-fitting** the training data, while the LR model is **under-fitting** it.
- The model that will generalise best in this case is the quadratic model, which makes sense because the data was generated using a quadratic model.
- But in general you won't know what function generated the data.

How can you decide how complex your model should be and How can you tell that your model is over-fitting or under-fitting the data?



- A way to tell is to **look at the learning curves**.
- These are plots of the model's training error and validation error as a function of the training iteration:
  - Just evaluate the model regular intervals during training on both the training set and the validation set, and plot the results.
- If the model cannot be trained incrementally (i.e., if it does not support `partial_fit()` or `warm_start`), then you must train it several times on gradually larger subsets of the training set.



- `sklearn` has a useful `learning_curve()` function to help with this.
- It trains and evaluates the model using cross-validation.
- By default it retrains the model on growing subsets of the training set.

If the model supports incremental learning you can set `exploit_incremental_learning=True` when calling `learning_curve()` and it will train the model incrementally instead.

- The function returns the training set sizes at which it evaluated the model, and the training and validation scores it measured for each size and for each cross-validation fold.

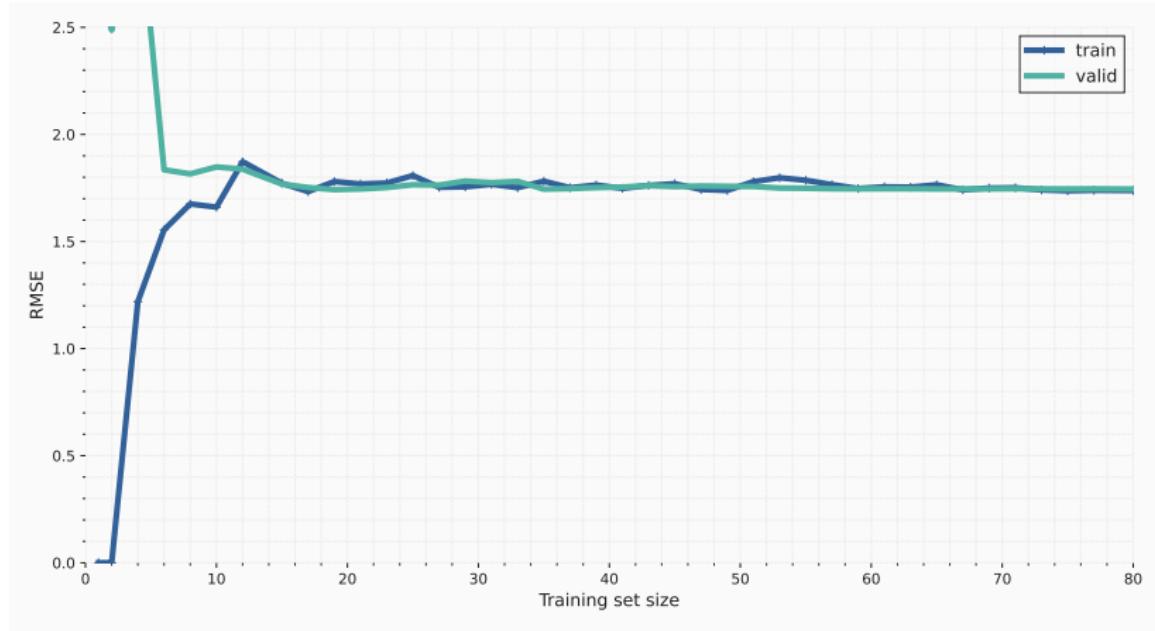


Figure 35: Learning Curves.



- This model is **under-fitting** as the training error tells.
- When there are just one or two instances in the training set, the model can fit them perfectly as RMSE is zero.
- As new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, as data is noisy.
- So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse.



- Now look at the validation error.
- When the model is trained on very few training instances, it is **incapable of generalizing properly**,
  - Which is why the validation error is initially quite large.
- As the model is shown more training examples, it learns, and thus the validation error slowly goes down.
- However, once again a straight line cannot do a good job of modeling the data, so the error ends up at a plateau, very close to the other curve.
- These learning curves are typical of a model that's under-fitting.

Both curves have reached a plateau as they are close and high.

# Training Models

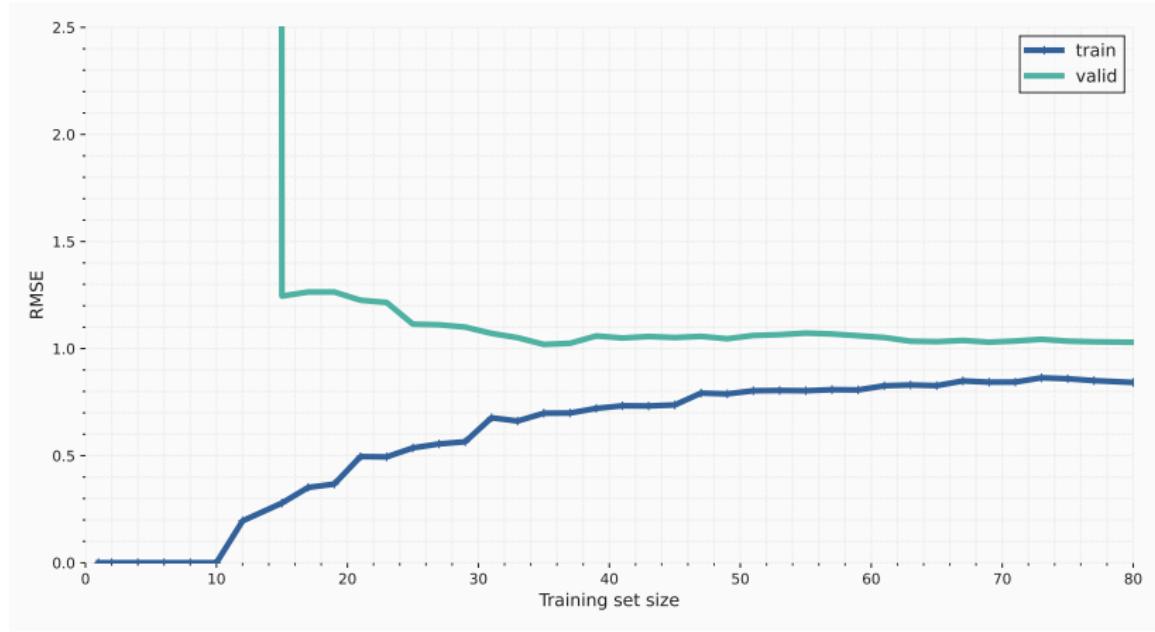


Figure 36: Learning curves of a 10<sup>th</sup> degree polynomial.



- These learning curves look a bit like the previous ones.
- However, there are two (2) very important differences:
  1. The error on the training data is much lower than before.
  2. There is a **gap between the curves**.  
This means that the model performs significantly better on the training data than on the validation data.

The hallmark of an overfitting model. If you used a much larger training set, however, the two curves would continue to get closer.



- A regularised version of linear regression:
  - a regularization term equal to  $\sum_{i=1}^m \lambda w_i^2$  is added to the MSE.
- This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible.

Ridge uses  $\ell_2$  regularisation.

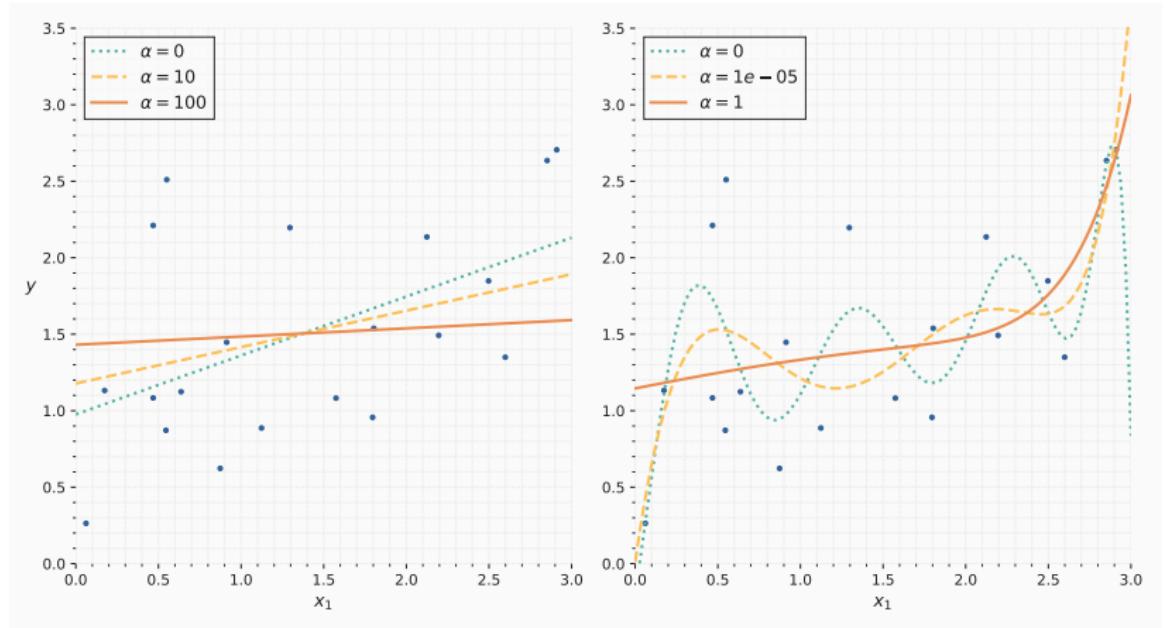
The regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularised MSE (or RMSE) to evaluate the model's performance.



- The hyper-parameter  $\alpha$  controls **how much** to regularise the model.
- For a linear model:
  - Set too low (i.e.,  $\alpha = 0$ ), it becomes a standard LR.
  - Set too high (i.e.,  $\alpha = 100$ ), it becomes a flat line.

The bias term  $\theta_0$  is not regularised.

# Training Models



**Figure 37:** Linear (left) and a polynomial (right) models, both with various levels of ridge regularisation.



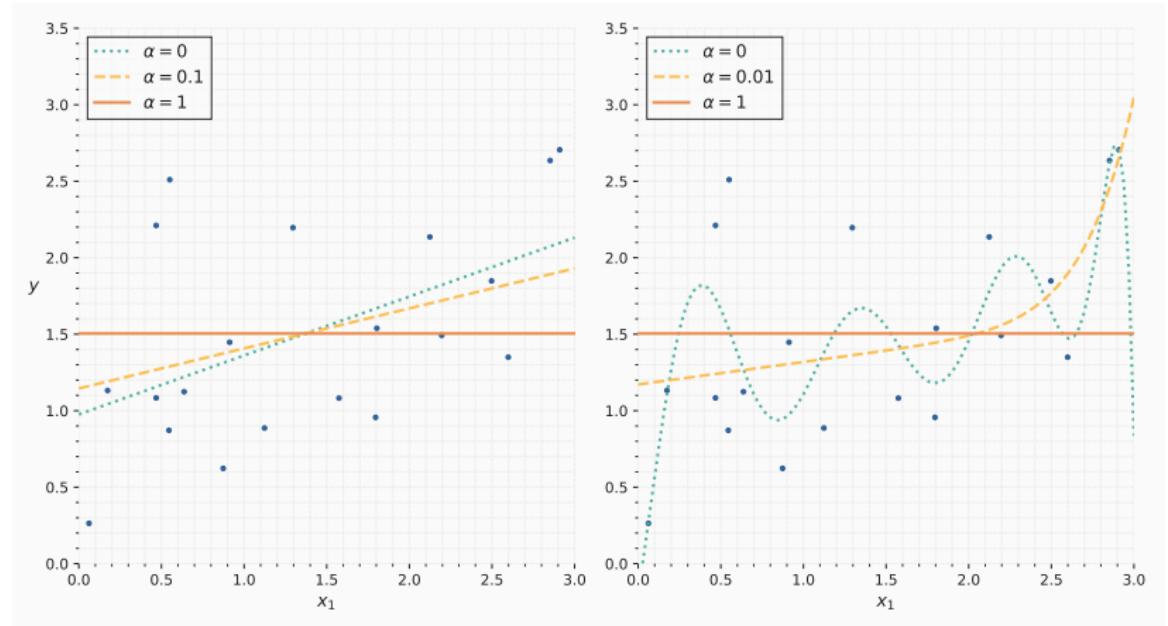
- Called Least Absolute Shrinkage and Selection Operator (LASSO) is another regularised version of LR.
- Similar to ridge regression, it adds a regularisation term to the cost function,

It uses  $\ell_1$  instead of  $\ell_2$  norm (i.e., Ridge Regression).

- Notice that the  $\ell_1$  norm is multiplied by  $2\alpha$ , whereas the  $\ell_2$  norm was multiplied by  $\alpha/m$  in ridge regression.

These factors ( $\alpha, m$ ) were chosen to ensure that the optimal  $\beta$  value is independent from the training set size

# Training Models



**Figure 38:** Linear (left) and polynomial (right) models, both using various levels of lasso regularisation.



- An important characteristic of LASSO is it eliminates the weights of the least important features (i.e., set them to zero).
- For example, the dashed line ( - - ) in the right-hand plot (with  $\alpha = 0.01$ ) looks roughly cubic:
- All the weights for the high-degree polynomial features are equal to zero.

The higher the norm index, the more it focuses on large values and neglects small ones.

LASSO regression automatically performs feature selection and outputs a sparse model with few nonzero feature weights.

# Training Models

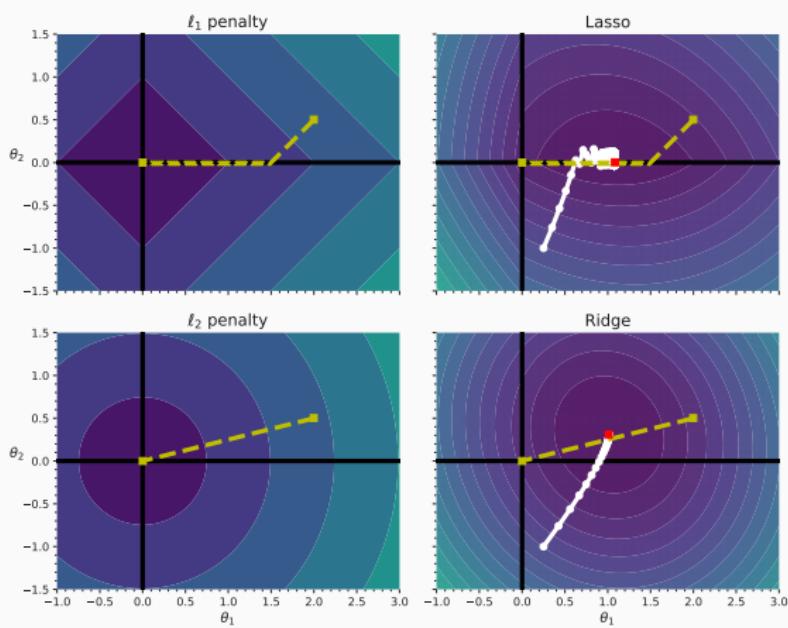


Figure 39: Lasso versus ridge regularisation.



- A middle ground between ridge regression and lasso regression.
- The regularization term is a weighted sum of both ridge and lasso's regularization terms, and you can control the mix ratio  $r$ .
  - When  $r = 0$ , elastic net is equivalent to ridge regression,
  - When  $r = 1$ , it is equivalent to lasso regression



- It is almost always preferable to have at least a little bit of regularisation, so avoid plain linear regression.
- Ridge is a good default, but if you suspect that only a few features are useful, you should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as discussed earlier.
- In general, elastic net is preferred over lasso because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.



- A way to regularise iterative learning algorithms such as GD is to stop training as soon as the validation error reaches a minimum.
- This is called early stopping
- For a model, as the epochs go by, the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set.
- After a while, though, the validation error stops decreasing and starts to go back up.
- This indicates that the model has started to over-fit the training data.
- With early stopping you just stop training as soon as the validation error reaches the minimum.

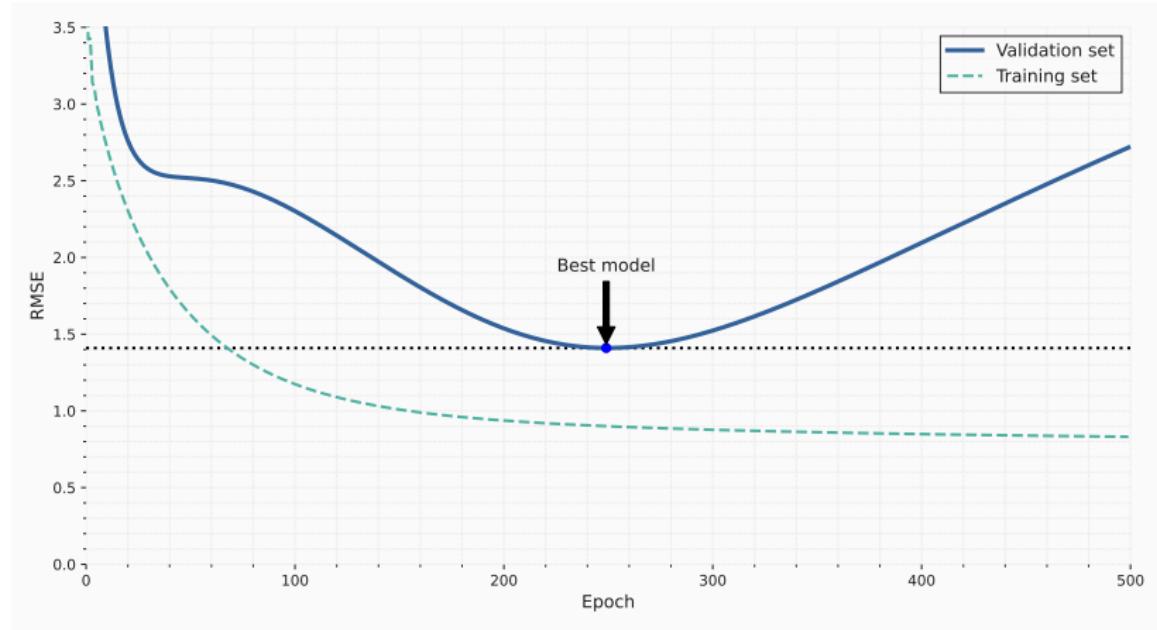


Figure 40: Regularising the early stopping.



- Some regression algorithms can be used for classification (and vice versa).
- Logistic regression (also called logit regression) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?).
- If the estimated probability is greater than a given threshold (typically 50%), then the model predicts that the instance belongs to that class (called the positive class, labelled “1”),
- and otherwise it predicts that it does not (i.e., it belongs to the negative class, labelled “0”). This makes it a binary classifier.



- So how does logistic regression work? Just like a linear regression model, a logistic regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the linear regression model does, it outputs the logistic of this result

$f$

- The logistic-noted ( $\cdot$ ) is a sigmoid function (i.e., S-shaped) that outputs a number between 0 and 1. It is defined as shown

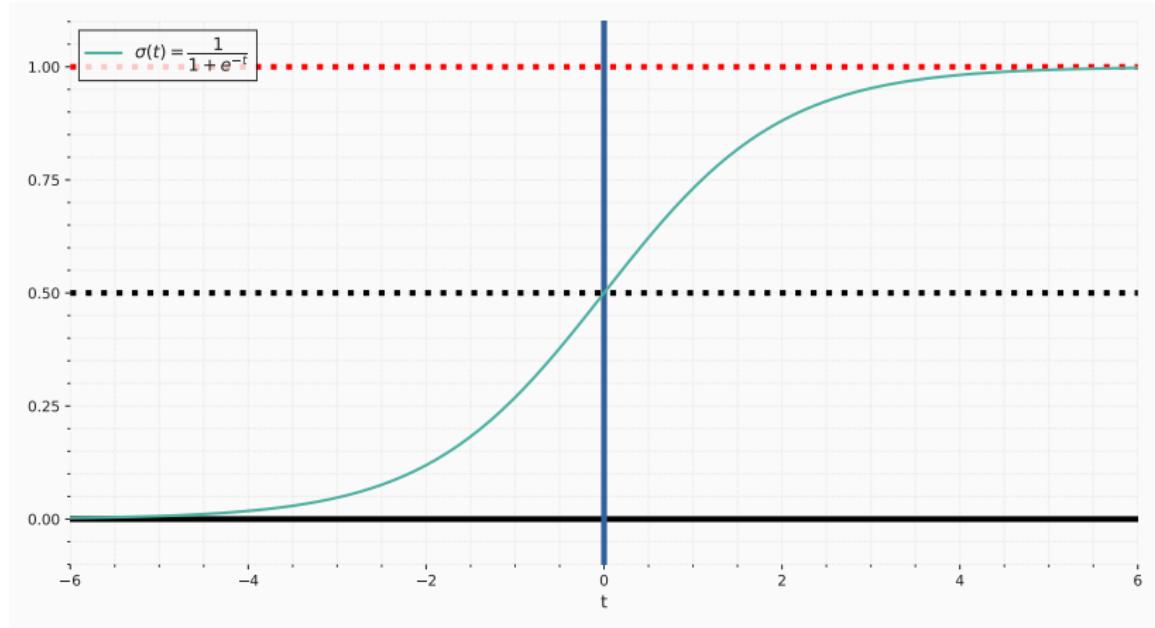


Figure 41: An example of a logistic function.



- The question now is how to train a **logistic** model.
- The objective of training is to set the parameter vector  $\theta$  so that the model estimates high probabilities for positive instances ( $y = 1$ ) and low probabilities for negative instances ( $y = 0$ ).

$$c(\theta) = -\log \hat{p} \quad \text{if } y = 1,$$

$$c(\theta) = 1 - \log(1 - \hat{p}) \quad \text{if } y = 0.$$

- $\log$  grows very large as  $t \rightarrow 0$ , making cost large if the model estimates a probability close to 0 for a positive instance.
- Same behaviour when a probability close to 1 for a negative instance.
- Inverse behaviour for when  $-\log(t)$  as  $t \rightarrow 0$ .



- A problem is there is no known **close-form** equation to compute  $\theta$ .
- But the good news is that this cost function is **convex**, so gradient descent (or any other optimisation algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough).



- Let's use the iris dataset to illustrate *logistic regression*.
- This is a famous dataset containing the sepal and petal length and width of 150 iris flowers of three different species:
  - Iris setosa,
  - Iris versicolor,
  - Iris virginica.
- Let's try to build a classifier to detect the Iris virginica type based only on the petal width feature.

```
from sklearn.datasets import load_iris  
  
iris = load_iris(as_frame=True)  
print(list(iris))
```



Figure 42: One of the flowers from the iris dataset: *Iris versicolor*.



- Next we'll split the data and train a logistic regression model on the training set:

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y,
→ random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)
```

- Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm.

# Training Models

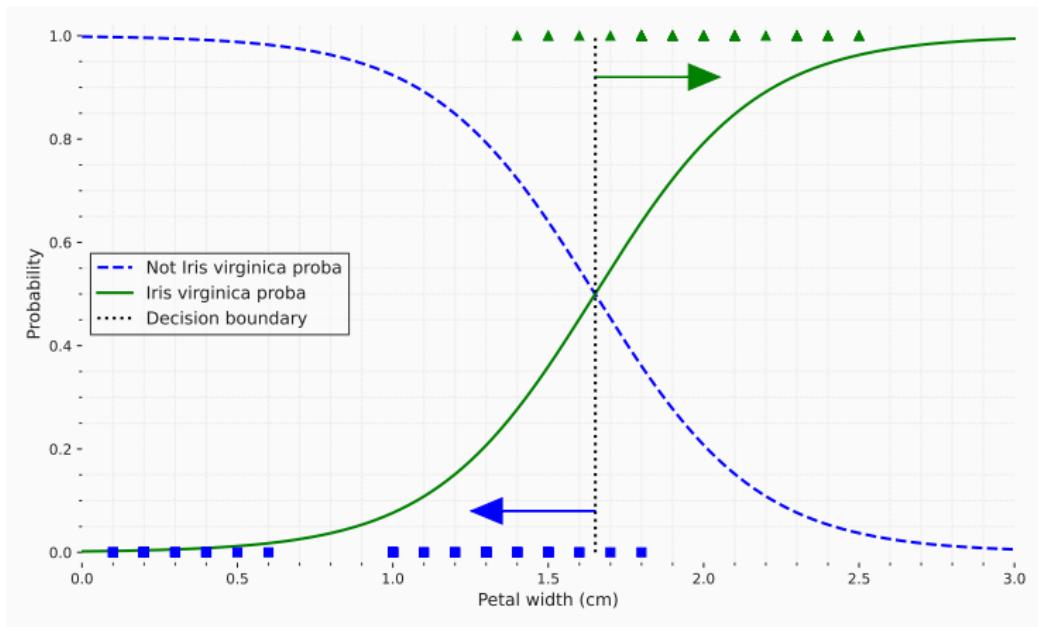


Figure 43: Estimated probabilities and decision boundary.



- The petal width of *Iris virginica* (triangles) ranges from 1.4 cm to 2.5 cm, while the other irises (squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm.

Notice that there is a bit of overlap.

- Above about 2 cm the classifier is highly confident that the flower is an *Iris virginica* (it outputs a high probability for that class).
- Whereas below 1 cm it is highly confident that it is not an *Iris virginica* (high probability for the “Not Iris virginica” class).



- In between these extremes, the classifier is **unsure**.
- If you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely.
- Therefore, there is a decision boundary at around 1.6 cm where both probabilities are equal to 50%:

If petal width is greater than 1.6 cm classifier will predict the flower is an *Iris virginica*, otherwise it will predict it is not (even if it is not very confident):

# Training Models

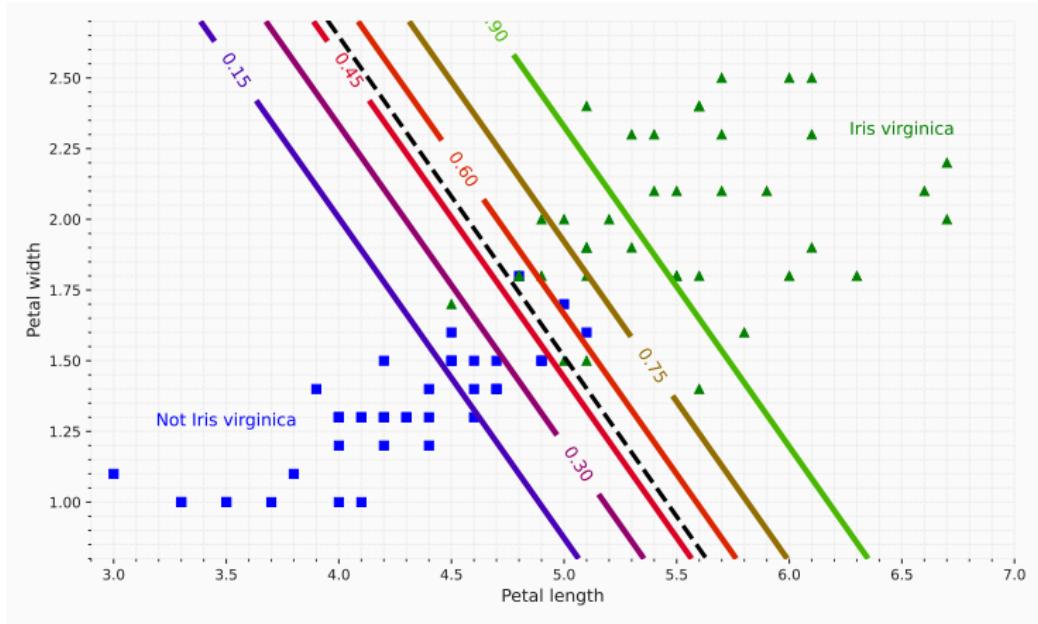


Figure 44: Linear decision boundary.



- Previous figure shows the same dataset, but this time displaying two (2) features:
  1. petal width,
  2. petal length.
- Once trained, the logistic regression classifier can, based on these two features, estimate the probability that a new flower is an Iris virginica.
- The dashed line (---) represents the points where the model estimates a 50% probability:

This is the model's decision boundary with each parallel line representing the points where the model outputs a specific probability.



# Training Models

- The LR model can be generalised to support multiple classes directly, without having to train and combine multiple binary classifiers.
- This is called softmax regression, or multinomial logistic regression.
- When given an instance  $x$ , the softmax regression model first computes a score  $s_x(x)$  for each class  $k$ , then estimates the probability of each class by applying the softmax function (also called the normalised exponential) to the scores.

Just like the logistic regression classifier, by default the softmax regression classifier predicts the class with the highest estimated probability.

# Training Models

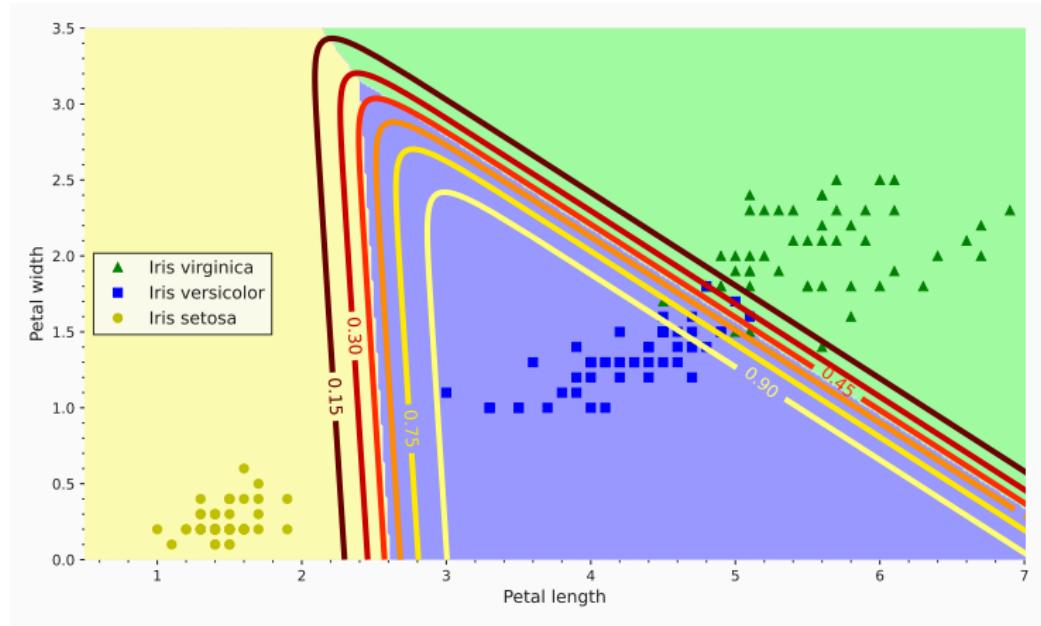


Figure 45: Softmax regression decision boundaries.



- [1] MD Islam and Morshed Chowdhury. “**Spam filtering using ML algorithms**”. In: (2005).
- [2] Santosh K Gaikwad, Bharti W Gawali, and Pravin Yannawar. “**A review on speech recognition technique**”. In: *International Journal of Computer Applications* 10.3 (2010), pp. 16–24.
- [3] MA Anusuya and Shriniwas K Katti. “**Speech recognition by machine, a review**”. In: *arXiv preprint arXiv:1001.2267* (2010).
- [4] What Is Data Mining. ***Introduction to data mining***. Springer, 2006.
- [5] Ziqiu Kang, Cagatay Catal, and Bedir Tekinerdogan. “**Machine learning applications in production lines: A systematic literature review**”. In: *Computers & Industrial Engineering* 149 (2020), p. 106773.
- [6] Ye-Jiao Mao et al. “**Breast tumour classification using ultrasound elastography with machine learning: A systematic scoping review**”. In: *Cancers* 14.2 (2022), p. 367.
- [7] Jeelani Ahmed and Muqeem Ahmed. “**Online news classification using machine learning techniques**”. In: *IIUM Engineering Journal* 22.2 (2021), pp. 210–225.



- [8] Aditya Gaydhani et al. “**Detecting hate speech and offensive language on twitter using machine learning: An n-gram and tfidf based approach**”. In: *arXiv preprint arXiv:1809.08651* (2018).
- [9] Joel Larocca Neto, Alex A Freitas, and Celso AA Kaestner. “**Automatic text summarization using a machine learning approach**”. In: *Advances in Artificial Intelligence: 16th Brazilian Symposium on Artificial Intelligence, SBIA 2002 Porto de Galinhas/Recife, Brazil, November 11–14, 2002 Proceedings 16*. Springer. 2002, pp. 205–215.
- [10] Prissadang Suta et al. “**An overview of machine learning in chatbots**”. In: *International Journal of Mechanical Engineering and Robotics Research* 9.4 (2020), pp. 502–510.
- [11] Soufyane Ayanouz, Boudhir Anouar Abdelhakim, and Mohammed Benhmed. “**A smart chatbot architecture based NLP and machine learning for health care assistance**”. In: *Proceedings of the 3rd international conference on networking, information systems & security*. 2020, pp. 1–6.
- [12] Helmut Wasserbacher and Martin Spindler. “**Machine learning for financial forecasting, planning and analysis: recent developments and pitfalls**”. In: *Digital Finance* 4.1 (2022), pp. 63–88.



- [13] Jiajing Wang et al. “**Predicting stock market trends using lstm networks: overcoming RNN limitations for improved financial forecasting**”. In: *Journal of Computer Science and Software Applications* 4.3 (2024), pp. 1–7.
- [14] Patel Monil et al. “**Customer segmentation using machine learning**”. In: *International Journal for Research in Applied Science and Engineering Technology (IJRASET)* 8.6 (2020), pp. 2104–2108.
- [15] Jatin Sharma et al. “**Product recommendation system a comprehensive review**”. In: *IOP conference series: materials science and engineering*. Vol. 1022. 1. IOP Publishing. 2021, p. 012021.
- [16] The Editors of Encyclopaedia. Britannica. **Deep Blue**. 2024. URL: <https://www.britannica.com/topic/Deep-Blue>.
- [17] Abdallah Bashir Musa. “**Comparative study on classification performance between support vector machine and logistic regression**”. In: *International Journal of Machine Learning and Cybernetics* 4 (2013), pp. 13–24.



- [18] Laurens Van Der Maaten, Eric Postma, Jaap Van den Herik, et al. **“Dimensionality reduction: a comparative”**. In: *J Mach Learn Res* 10.66-71 (2009).
- [19] Jesper E Van Engelen and Holger H Hoos. **“A survey on semi-supervised learning”**. In: *Machine learning* 109.2 (2020), pp. 373–440.
- [20] H Jabbar and Rafiqul Zaman Khan. **“Methods to avoid over-fitting and under-fitting in supervised machine learning (comparative study)”**. In: *Computer Science, Communication and Instrumentation Devices* 70.10.3850 (2015), pp. 978–981.
- [21] Latifa Greche et al. **“Comparison between Euclidean and Manhattan distance measure for facial expressions classification”**. In: *2017 International conference on wireless technologies, embedded and intelligent systems (WITS)*. IEEE. 2017, pp. 1–4.
- [22] Ayush Bhatnagar (<https://math.stackexchange.com/users/537077/ayush-bhatnagar>). **Convexity of MSE in Neural Networks?** Mathematics Stack Exchange. URL:<https://math.stackexchange.com/q/4263311> (version: 2021-09-29). eprint: <https://math.stackexchange.com/q/4263311>. URL: <https://math.stackexchange.com/q/4263311>.



**CM** Confusion Matrix. 181–183, 185, 186

**FN** False Negative. 185, 187

**FP** False Positive. 185, 186

**FPR** False Positive Rate. 206, 207, 209

**GD** Gradient Descent. 231, 248, 250, 257–259, 262, 290

**LASSO** Least Absolute Shrinkage and Selection Operator. 284, 286

**LR** Linear Regression. 232, 236, 237, 247, 250, 270, 271, 273, 282, 284, 305

**MAE** Mean Absolute Error. 94, 95



**ML** Machine Learning. 168, 169

**MNIST** Modified National Institute of Standards and Technology.  
169–171, 175, 218, 222

**MSE** Mean Square Error. 236, 237, 248, 250, 281

**OvO** One v. One. 217–219, 223

**OvR** One v. Rest. 217, 219, 223, 225

**PR** Polynomial Regression. 264, 270, 271, 273

**RMSE** Root-Mean Square Error. 93–95, 236, 277, 281, 290

**ROC** Receiver Operating Characteristic. 206, 207, 209

**SGD** Stochastic Gradient Descent. 177



**SVD** Singular Value Decomposition. 247

**SVM** Support Vector Machines. 219, 220, 223

**TN** True Negative. 185

**TNR** True Negative Rate. 206

**TP** True Positive. 185, 186

**TPR** True Positive Rate. 187, 207, 209

**US** United States. 169

**USD** United States Dollar. 106, 107, 123