class: middle

Data Science for Developers (Beginner)

@DrPhilWinder

DrPhilWinder

http://WinderResearch.com

http://TrainingDataScience.com

Schedule

- 09:00 Start
  - Introduction, Basics, Data Engineering
- 10:15 Morning break
  - Regression, Overfitting
- 12:00-13:00 Lunch
  - Classification
- 14:15 Afternoon break
  - Clustering, Contingency
- 16:30 End

Information

Name: Phil Winder

Title: CEO Winder Research

Occupation: Engineer (cloud software and data science)

This training: First of three parts

Other training: visit https://WinderResearch.com/training

Online Training: visit https://TrainingDataScience.com

Tweet! @DrPhilWinder

Email: phil@WinderResearch.com

.center[Winder Research]

Content and Questions

- I will email slides and workshops. To do this I need your email addresses.
- Please interrupt to ask questions. Data Science is all about asking the right questions.
- We're working on a cloud environment. You don't need to install anything (unless you want to).

Setting Expectations

- This *is* difficult
- There *is* a lot of content here
- We *will* move fast (this is intentional)
- Data Science is *chaotic*; learning is not as easy as A.B.C.

The Goal:

1. Overview. Understand what you don't know.
2. Go away and continue learning.

The goal is not to become an expert in 1/2/3 days

Models we will Learn Today

- Decision Tree
- Linear Regression
- Linear/Logistic/SVM Classification
- Nearest neighbour/K-NN
- K-Means

Concepts we will Learn Today

- Core part of business
- Minimalism/Simplicity
- It's all about the data
- Complexity vs. Generalisation

---

name: introduction

Introduction

This section:

- Why do we do data science?
- What does data science consist of?
- Is it magic?

???

Before we get into any technicalities, it is really important to have a grounding in why it is we do data science. Or why we do engineering in general.

This section introduces data science. It explains what it is and why we do it.

It covers, with broad brush strokes, the different aspects to data science and you might be surprised to see where all the time is eaten up.

Data Science is interesting. It is sexy. But there are a whole host of fields that we need to be experts in before we can do any real damage.

What is Data Science?

- Better at Software Engineering that Statisticians
- Better at Statistics than Software Engineers
- Better consultants than Employees

Involves:

- Information
- Automation

A.k.a:

- Analytics
- Machine learning
- Big data
- A.I. (marketing buzzword alert!)

Ubiquity of data

- Traditionally, separate teams would pour through data.
- Businesses can save money and time by automating these decisions.
- Software Engineers are best suited to implement this automation.

???

Traditionally a manual tasks, teams of analysts, DBAs, modellers and statisticians would pour through data.

Volume and variety of data prevents future manual analysis. Instead, we utilise computational power to automate these tasks away.

Software Engineers are best suited to implement this automation.

Use in business

Decades old, but limited to certain verticals.

- Marketing (advertising)
- Recommendations
- Credit scoring

Now spreading into all verticals.

Example: Target

> *"I had a talk with my daughter," he said. "It turns out there's been some activities in my house I haven't been completely aware of. She's due in August. I owe you an apology."*

Duhigg, Charles. "How Companies Learn Your Secrets." The New York Times, February 16, 2012. http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html

???

> *… a man walked into a Target outside Minneapolis and demanded to see the manager. He was clutching coupons that had been sent to his daughter, and he was angry…*
>
> *The manager apologized and then called a few days later to apologize again. On the phone, though, the father was somewhat abashed.*
>
> *"I had a talk with my daughter," he said. "It turns out there's been some activities in my house I haven't been completely aware of. She's due in August. I owe you an apology."*

Duhigg, Charles. "How Companies Learn Your Secrets." The New York Times, February 16, 2012. http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html

---

Example: Walmart

- 2004: Hurricane Frances

Think about what you could do given Wallmart's data?

- Bottled water?
- Tinned food?
- Drop in sales fresh veg?

Is that a guess?

???

> *HURRICANE FRANCES was on its way, barreling across the Caribbean, threatening a direct hit on Florida's Atlantic coast. Residents made for higher ground, but far away, in Bentonville, Ark., executives at Wal-Mart Stores decided that the situation offered a great opportunity for one of their newest data-driven weapons, something that the company calls predictive technology.*
>
> *A week ahead of the storm's landfall, Linda M. Dillman, Wal-Mart's chief information officer, pressed her staff to come up with forecasts based on what had happened when Hurricane Charley struck several weeks earlier. Backed by the trillions of bytes' worth of shopper history that is stored in Wal-Mart's computer network, she felt that the company could "start predicting what's going to happen, instead of waiting for it to happen," as she put it.*

Think about what you could do given Wallmart's data?

- Bottled water?
- Tinned food?
- Drop in sales fresh veg?

Is that a guess?

The results were intriguing:

- 7x increase in sales of Pop-tarts
- Top selling item: Beer.

Wallmart then sent out excess trucks filled with "toaster pastries and six-packs", following the forecast path of the hurricane.

Hays, Constance L. "What Wal-Mart Knows About Customers' Habits." The New York Times, November 14, 2004.
http://www.nytimes.com/2004/11/14/business/yourmoney/what-walmart-knows-about-customers-habits.html

---

Discovery

- Data Science is a journey of discovery
- The goal for the business is money.
    - Increase revenue
    - Optimise costs
    - Save time

How?

???

A key part of data science is discovery.

Businesses and increase revenue, optimise costs and save time with data science.

Customer churn (common): - Attracting customers is more expensive than keeping current ones. - There could be a significant marketing budget that pays for work. -

How to chose customers that are likely to leave? What incentive to give them?

How do we discover areas where we could use Data Science in our business?

Fish and chips: - The best fish and chips are freshly cooked, but I hate waiting for fish and chips - Can we preemptively cook the right amount at the right time?

Automation of Decisions

- Automation improves efficiency, reduces errors and costs less.
- It frees people from mundane tasks
- Business units should make decisions in a data-driven manner

???

We are currently beginning the mass automation of significant parts of the economy.

Automation improves efficiency, reduces errors and costs less. It frees people from mundane tasks to allow them to be more innovative and productive.

Automating business decisions requires data.

Data driven decision-making (not to be mistaken with domain driven design) is an emerging trend to divest strategic choices to the scientific method.

For example, knowing that one type of advertisement is going to do better than another makes the decision of which advert to use easy.

Data Science is an Asset

- Think of data science as an asset
- Some businesses are data science products
- Investment in data science results in good decisions

???

Data science is an asset to a business. Some data science offerings have become products in their own right (e.g. credit risk).

Good decisions <= good data science <= good data. An investment in good data and data science results in good decisions.

Example: Signet Bank

???

In the 80's credit profiling changed the way banks offered money to people. But they only offered uniform pricing. The banks worried the customer wouldn't accept "price discrimination" and they didn't have the data.

Some chaps went to work for a small bank in Virginia, US. Confident that they should be modelling profit, not default, they started randomly giving different terms to different types of customers.

Started off losing money after defaults, but these were viewed as "data investments". Given a few years they credit card part of the bank vastly surpassed traditional banking activities and was spun off int a new business called Capital One.

Their data is a valued asset. So valuable that they were willing to pay for it. The resultant data science has become an entire industry.

More examples: - Caesar's Entertainment (gambling) - Amazon - Google, Facebook, etc.

Talent shortage

Global Engineering shortage.

UK Stats for the whole of Engineering in 2017:

- 40k are UK nationals

- 40k are foreign nationals (Brexit?!?!)
- 20k deficit

Data Science shortages in the US:

- US: 100,000 shortage - Gartner
- US: 140,000-190,000 shortage - McKinsey
- US: 181,000 needed by 2018 (IDC)

???

In 2017 the UK engineering sector requires 100,000 graduate-level engineers per year. 40k are UK nationals. 40k are foreign nationals. 20k deficit.

The state of engineering, Engineering UK, 2017, http://www.engineeringuk.com/media/1355/enguk-report-2017.pdf

- Hampering the UK's presence in global engineering
- Brexit?
- Hard to get more detailed numbers.

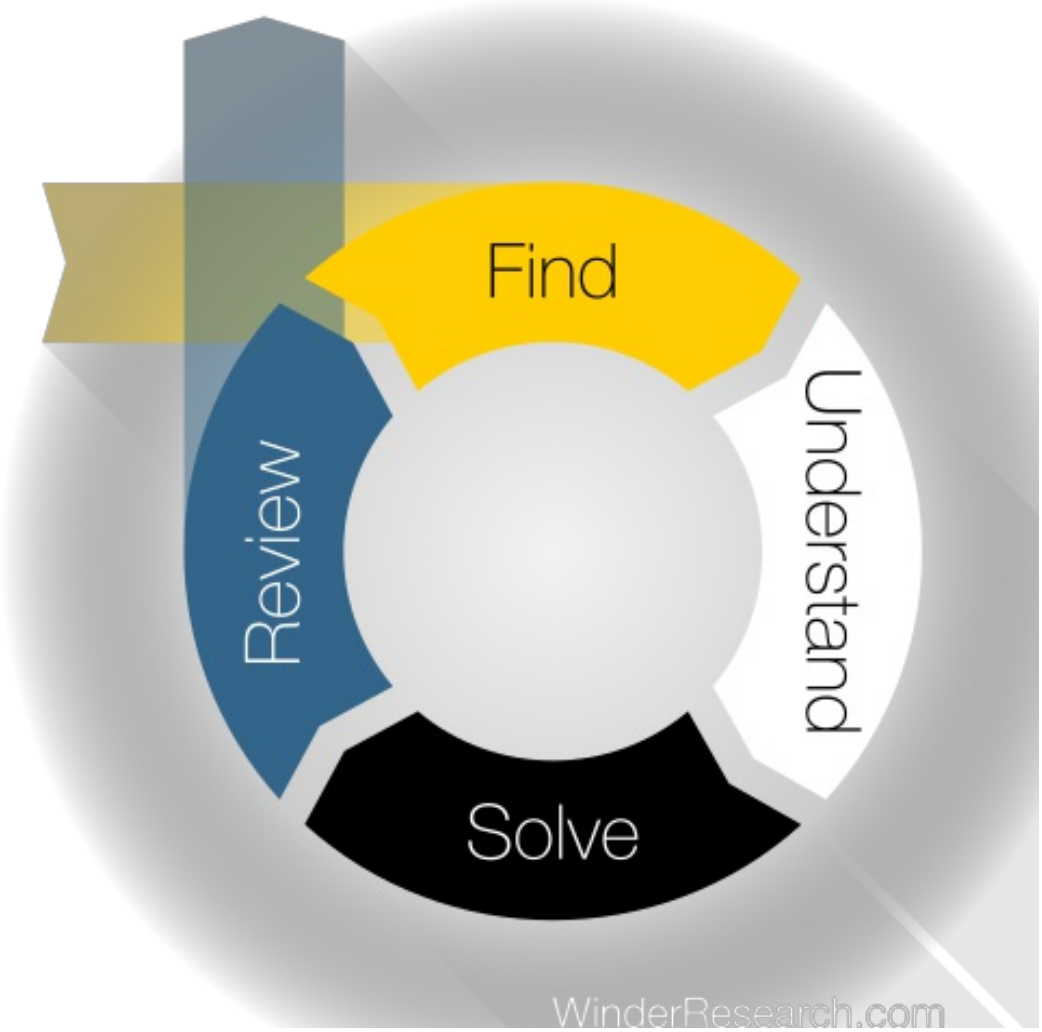
Data science: - UK: "Rare as unicorns" - Guardian - US: 100,000 shortage - Gartner - US: 140,000-190,000 shortage - McKinsey - US: 181,000 needed by 2018 (IDC)

Big numbers, but take with a pinch of salt.

- UK Median salary: £65k, vs. £45k for all of IT (https://www.itjobswatch.co.uk/jobs/uk/data%20scientist.do)

Data science as a process

Data Science gets its name from the fact that much of the work is research. Problems are not obvious and solutions are not ready made.



.center[ ]

Not quite software engineering

- Much of data science is software engineering
- Don't make the mistake of running data science projects like a software engineering project
- Scrum, XP, Agile, etc. Don't work well.

Data Science is:

- High risk
- High reward
- Difficult
- Unpredictable

???

We can learn a lot from Software Engineering and many skills are transferable.

However, data science is far more research than it is development. It is very hard to make estimates of when or how well a solution will perform.

This means that many software processes like Agile, XP, etc. do not fit well with data science projects.

This also means that data science projects are in general: - High risk - High reward - Difficult - Unpredictable

---

CRISP data mining process

.center[ ]

By Kenneth Jensen CC BY-SA 3.0, via Wikimedia Commons

???

Cross industry standard process for data mining

Business understanding

Most important step

This is the #1 reason why data science projects fail.

- What are the goals?

- Why are they the goals?
- Do the goals map to specific techniques in data science?

Often takes a number of iterations to figure out the right question

???

The most crucial step, and the one that most often goes wrong.

Think carefully about the business problem that is trying to be solved.

- What are the goals?
- Why are they the goals?
- Do the goals map to specific techniques in data science?

More often than not, it takes at least an full iteration to establish what the business *really* wants.

Accepting that the problem may be misunderstood

The solution: Assume you don't understand the problem

It is your job as a data scientist to figure out what is the real problem.

- Root cause analysis
- The Five Whys

???

Like most of engineering, data science is an iterative process.

Proposed problems and ideas may not represent what is really important to the business.

It is your job as a data scientist to figure out what is the real problem.

- Root cause analysis
- The Five Whys

For example:

> *"Who is the most profitable customer?"*

> *"What are the characteristics of a profitable customer?"*
>
> *"Given a prospective customer, will they be profitable? How can I target and choose new customers to maximise profit?"*

1. aggregation
2. clustering
3. classification and prediction

???

> *"Who is the most profitable customer?"*

Is that really useful? We could simply look at the accounts. We ask why?

> *"What are the characteristics of a profitable customer?"*

More interesting. Then we could look for similar customers that are not profitable and ask why.

But still, how do we change the buying patters of people that have already bought. We can't change the past.

We ask why again. What do we want to do differently with new customers?

> *"Given a prospective customer, will they be profitable? How can I target and choose new customers to maximise profit?"*

There we are. A truly actionable result. We can train on data we have, then test on new people.

Note that each question has profound effects on technology. The first question is a simple aggregation. The second is a clustering problem. The third is a classification and prediction problem.

Data Understanding

Second most important part of data science

- Must become a domain expert
- Real data science is about understanding the data, not producing complex

models

- Think of models as experiments to help understand the data
- Data is the "raw material"
- Find and remove sources of data leakage

???

Once we know what the business wants, do we have the data to achieve that?

The data is the "raw material" of the solution. If we don't have the right data, we can't produce a valid solution.

- Do we have the data?
- Do we have access?
- Will we need to acquire or purchase new data?
- Will it cost?

Data preparation

- Is the data in the right format?
- Is the data valid?
- Is the data reliable?
- Is it labelled?

Generally a process of:

- Finding and obtaining data
- Converting and selecting data
- Dealing with missing data
- Cleaning
- Normalisation
- Catagorisation

???

Once we have the data, is the data prepared for use?

- Is the data in the right format?

- Is the data valid?
- Is the data reliable?
- Is it labelled?

Very important not to leak temporal information. E.g. historical data might present data that is not available at the time. A big example is stock market prediction. It's quite easy to accidentally create models that correlate well with market movements, but when tested on live data, they don't work. Usually we trick ourselves by using data that isn't available at the time.

Generally:

- Converting and selecting data
- Dealing with missing data
- Cleaning
- Normalisation
- Catagorisation

Modelling

- Encoding your domain knowledge

Generally attacked for both sides:

- Develop model using your understanding of the data (deduction)
- Using the data to fine tune the model (induction)

???

We won't dwell on this now, since that is the topic of this course.

But this step represents implementing the techniques that you are about to learn to achieve the desired result.

Evaluation

Very important. Generally occurs iteratively.

- Retest the model in a realistic setting (but not in production!)
- "Smoke test" the work to weed out edge cases
- Does the resulting work satisfy the needs of the business?
- Has the research presented any new questions?
- Does the model perform to an expected level? (See probability of detection, false alarm rate, etc.)
- And how is the model to perform in the future?
- Can the assumptions made during the research change over time?
- Do we need to continuously reevaluate the model to ensure that performance doesn't degrade significantly?
- Do we need another iteration of the CRISP/research model?

???

We must spend time carefully evaluating our work to gain confidence in the results.

- Retest the model in a realistic setting (but not in production!)
- "Smoke test" the work to weed out edge cases

Does the resulting work satisfy the needs of the business? Has the research presented any new questions?

- Does the model perform to an expected level? (See probability of detection, false alarm rate, etc.)

And how is the model to perform in the future? Can the assumptions made during the research change over time? Do we need to continuously reevaluate the model to ensure that performance doesn't degrade significantly?

- Do we need another iteration of the CRISP/research model?

Remember that Data Science results are far more ambiguous than software projects. Data Science is certainly more research than it is development.

Deployment

Depends heavily on the results of the research. Very domain specific.

E.g.

- Lots of data, updated constantly? Streaming technologies.

- Lots of data, not updated frequently? Batch technologies.
- Not updated at all or very infrequently? Simple web-app.
- Complex data? Bespoke to model.

...

The new field of DataDevOps:

- Monitoring
- Performance
- Altering
- Implementation
- Scalability

???

Moving the model into production depends massively on the solved problem.

It could be anything: a new set of procedures to improve efficiency, a simple rule or a massively distributed system that serves millions of users.

The decision to deploy should be taken with all the stakeholders. Since deployment could have knock on effects to other people.

The deployment step is currently beyond the scope of this course, but we will touch on a few deployments whilst looking at some examples.

But don't "throw the model over the wall". You should be invested in all aspects of deployment, ideally with you being directly involved. Think about:

- Monitoring
- Performance
- Altering
- Implementation
- Scalability
- Costs

Implications

- Wider ramifications

- May affect large numbers of people
- Need buy in from other parts of the business

???

The result of a data science project can also have much wider ramifications for the business.

If the goal of the work was to improve processes, then this may affect large numbers of people.

If the goal of the work was to produce a new product, then other parts of the business will need to be frequently looped in.

Software development

> *"Your model isn't what the data scientists design, it's what the engineers build"*

I think that data scientists should be involved with the development of production algorithms.

Teams of multi-disciplinary engineers; all learning from each other.

DataDevOps

???

The maxim most often used is:

> *"Your model isn't what the data scientists design, it's what the engineers build"*

My opinion is slightly biased here. I consider myself an Engineer first, and a Data Scientist second.

I think there is a significant blurring of the disciplines and rightly so. From a business perspective, if you can't engineer a representation of the design, then the design is (often) useless.

Obviously there are exceptions to this, but business isn't likely to stick around to wait for .

Procedural development

- Results are often procedural in nature. E.g. if x > y then perform some action.

You'll need to inform the rest of the business.

???

If the result is more procedural in nature, e.g. only target customers that are , then these procedures need to be updated and communicated.

This could require interaction with other departments. E.g. is there some marketing angle, are there legal ramifications, etc.

Operational concerns

- Could alter the day-to-day running of a business
- Fewer people or new verticals
- Businesses need to be reactive

???

New procedures or products or features could alter the day to day running of the business.

It might take fewer people or open new lines of business.

The rest of the business should be prepared to react appropriately.

Human concerns

Most difficult part of data science.

Data Science automates decisions

- Similar to ramifications of process-automation through software engineering

- Personal consequences? Wider consequences? Privacy issues? Profiling?
- Solution: Have strong cultural and philosophical ideals within your business.

???

One of the greatest potential issues is the ethical conundrum that automation delivers.

Are humans no longer required in this part of the business? What will happen to these people?

Are there privacy issues? How will people react to the type of profiling that you are doing?

What are the wider consequences? If your business is the only employer in town, and you replace everyone with an automated equivilent, what happens to the town? Do you care? Should you care?

Here, the philosophy and culture of the business or the people that make it will have to make some tough decisions.

Conclusion

This section: Why Data Science?

Business goals: make money, save money or save time. Data Science generates profit.

Project justifications - you now know how they are judged:

- Alignment with Business Goals
- A well defined, testable requirement
- A robust plan
- Buy-in and integration with other parts of the business

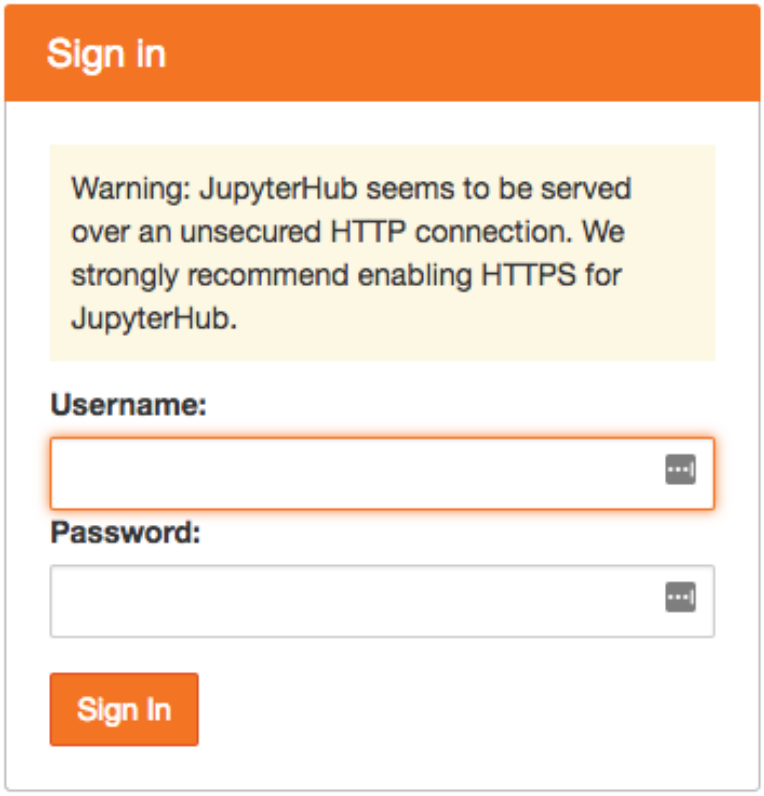
???

However, there are more philanthropic, scientific reasons for undertaking a project too. So these arguments may not directly apply to charitable causes or academia.

---

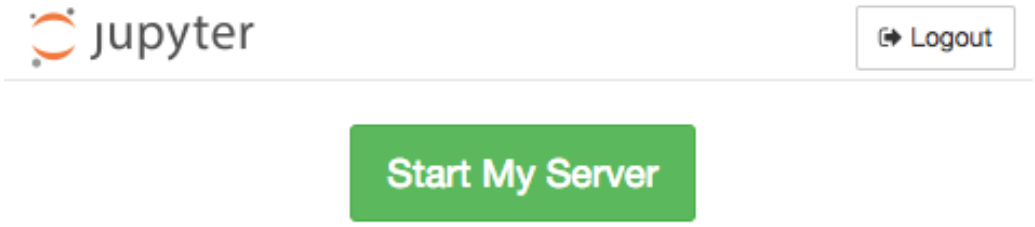
http://data.WinderResearch.com/

1. Log on to the cloud environment:
2. Pick and remember a username. No password. Click "Sign In".
3. Once you are on the control panel, click "Start My Server"
4. In a minute you will see a list of files. Read the `README.md`, then start `1-Introduction`.



.left-column[ ]

.right-column[



]

---

name: basics

Basics and Decision Trees

This section:

- Goals
- Terminology
- Derivation of first model

???

Now we have a firm understanding of how business problems map to solutions we need to learn the techniques to deliver the solutions.

This section introduces the basic terminology and concepts used in data science.

The ultimate goal

1. The goal is to make a decision or a prediction
2. Based upon Information
3. We want to improve a decision
4. The quality of a decision is represented by the certainty of the data

Think.

???

First lets discuss what the goal is. What is the goal?

- The goal is to make a decision or a prediction

Based upon what?

- Information

How can we improve the quality of the decision or prediction?

- The quality of the solution is defined by the certainty represented by the information.

Think about this for a moment. It's a key insight. Think about your projects. Your research. The decisions you make. They are all based upon some information. And you can make better decisions when you have more good quality information.

Information

Mathematical definition:

- Entropy: amount of information that can be stored in a limited number of bits

E.g. coin, die, random data

???

Claude Shannon defined exactly what information is. He defined a measure of the level of information contained within a variable and called it *entropy*.

Entropy is a measure of how much information is contained within an event. A coin

toss has lower entropy (less information) than the roll of a die. This is because the coin toss only has two possible outcomes. A die has 6.

A random variable with a wide *spread* has more entropy that one with narrow *spread*.

Uncertainty

Uncertainty represents the effects of entropy.

High entropy problems are highly uncertain.

Key Point: We want to be *certain* about the result.

Key Point: High entropy problems are harder to solve.

???

I.e. we cannot be certain about a solution if the data has high entropy.

Key Point: We want to be *certain* about the result.

Uncertainty reduction

To make a better decision, we need to reduce the uncertainty in the problem.

Everything we do is to reduce the amount of uncertainty

If we can be certain, we can make good decisions.

We will use this idea to derive a famous algorithm shortly.

???

Hence, the whole point of any modelling process is to reduce the amount of uncertainty in a decision or estimate that we make.

If we can make good decisions we can build good products and good businesses.

Reducing uncertainty by reducing the entropy measure of the data is the topic of

the next section.

Key Point: We need to reduce the uncertainty to improve our decision. The question of the entire course is: how?

Terminology

Data Science has emerged from many different disciplines.

Each has its own terms for the same thing.

I use a mix of statistical and computer science terminology.

???

Before we move any further, we need to talk terminology.

Data science is really bad for lots of different, complex words all meaning the same thing.

The reason for this is that data science has emerged from a number of different disciplines.

For example, statistic's L1-norm is the same as machine learning's Manhattan distance.

Also, terminology is quite personal to an individual's experience. Some of this terminology is my own.

Observations, samples and datasets

- Population
- Sample
- Observation

??? d An observation is a single measurement. It is often (even by me) referred to as an individual sample or data point. Words don't matter so long as you and your audience understand that you mean a single instance.

A sample is a chosen set of observations. But this isn't generally used because of it's confusion with an individual sample.

Instead, data scientists often use the word dataset to refer to a collection of observations.

How you *choose* a sample is very, very important. More about that later.

Features or attributes

A *feature* is one dimension of the measurement.

A finance dataset might have a `loan_amount` feature. A marketing dataset might have an `age` feature.

???

Other than observations, the next most important word is feature.

A *feature* is one dimension of the measurement. For example, a finance dataset might have a `loan_amount` feature. A marketing dataset might have an `age` feature.

An *attribute* is another word for a feature.

Labels

Labels represent the answers to the problem, if there are any.

A.k.a Targets

???

Labels are required for supervised machine learning tasks.

For example, in a classification problem, *labels* represent the correct class for an observation.

Labels are also often called targets.

Parameters and Hypterparameters

Parameters are part of the model. E.g. the

and

in

$$= \quad +$$

.

Hyperparameters are knobs that you can tweak to alter the behavior of a model.

Parameters are directly optimised.

Hyperparameters are indirectly optimised (either manually or by brute force - or by more advanced algorithms like genetic algorithms)

Models

Models are a simplified version of reality.

We create them to be able to understand and act upon the underlying process.

- Life is messy: Noise
- Prefer simple models

???

Reality is messy. We use imprecise tools and equipment to sample an chaotic natural process.

The mess that is included in our measurements is called noise.

Noise masks the data that we are interested in.

Models attempt to simplify the measurement and ignore the noise.

Simpler models are easier to understand and easier to act upon.

Induction

The creation of models from data is called induction.

???

Contrast this to deduction, which is the process of formulating a model or theory from logical assertions.

Traditional science emphasised the importance of deductive reasoning, and is still preferred in many more traditional disciplines.

And in data science it still holds an important place in sanity checking. If the results are not what you expect, then either you don't understand what is going on, or you've made the wrong deductions.

Prediction

Prediction is often used for "predicting the future".

But in Data Science prediction means: estimate the most probable output.

So when our algorithm decides that this instance belongs to a class, we're making a prediction.

Types of learning

When talking about producing learning from data, there are two distinct forms of learning:

- Supervised
- Unsupervised

The type of learning is usually defined by whether the data has labels.

Supervised

- Requires labelled data
- Some algorithms work with categorical data, some with continuous data, some both.

Examples of supervised questions: - "What is likely to be next quarter's GDP? - "Will we continue to retain this customers account for the next 6 months?"

???

Supervised machine learning occurs when an algorithm is provided with data is labelled with a known outcome.

Supervised learning is then split into the type of label that is used. Some algorithms work with categorical data, some with continuous data, some both.

Examples of supervised questions: - "What is likely to be next quarter's GDP? - "Will we continue to retain this customers account for the next 6 months?"

Unsupervised

- No labelled data

Examples of unsupervised questions: - "Do our customers fall into specific groups?" - "What do similar customers like to buy?"

How do we quantify how well we perform?

???

Unsupervised problems arise when there are no labels indicated in the data.

Unsupervised problems often require some form of grouping or clustering to find similar types of data.

Examples of unsupervised questions: - "Do our customers fall into specific

groups?" - "What do similar customers like to buy?"

Combination approaches

Many domains have some labelled examples, but not all.

Generally go through an iterative process:

- Clustering
- Apply labels to similar types
- Investigate interesting data
- Repeat until you have a labelled dataset

???

Fairly often problems come with data that has some labels. For example some instances have been manually labelled by experts, but cannot possibly label all data.

In this case there are a subset of special techniques called semi-supervised algorithms.

These are often simply a combination of clustering followed by classification.

I.e. Similar instances can be labelled with the same label. The difficulty is where you draw the line.

Segmentation

This will seem quite complicated if you've never done anything like this before. That's ok!

- A direct example of making better decisions by reducing uncertainty
- At the end, we will have derived the decision tree algorithm

???

So let's walk through a very visual, intuitive example to help describe what all data

science algorithms are trying to do.

I want to do this to show you that all algorithms that you've every heard of have some very basic assumption of what they are trying to do.

At the end of this, we will have completely derived one very important type of classifier.

Problem

- Predict whether a new beer belongs to group `A` or group `B`
- Given some data
- How can we predict?
- First we can see if we can separate the groups *cleanly*

???

Imagine that we had a classification problem. We want perform model induction so that we can generate a predictive model to use to decide whether a new beer belongs to group `A` or group `B` (a binary classification problem).

We have a dataset where the targets have been specified, so this is a supervised task.

One way to solve this is to arbitrarily split the data according to some rule.

E.g. is `feature 1 > 0.5`

Then we can test to see how well we have split the data.

We do this by measuring how mixed the resulting classes are.

---

class: pure-table, pure-table-striped

Data

To make this more concrete, consider the following data:

Alc. Volume	Colour	Class
4.2	92	A
6.4	102	A
3.5	3	B
4.7	10	B

Note:

- the features
- the differing scales
- the classes

Choosing attributes

We can see a rule that will cleanly separate the data. But how do we quantify that?

- Segment the data so that the classes are *clean*

This relies on some measure of *purity*

???

So let's think how to generate a model. We want to segment the data so that the classes are *clean*.

By clean, I mean pure, homogeneous, clear cut.

If we sliced the data and we ended up with two classes, all A's in one class and all B's in the other, then the result is as pure as it can possibly be.

Obviously we can do this visually. We can *see* a *predictive model* that would solve the problem.

But real life isn't so easy.

Entropy

We have already seen a measure of purity. Entropy!

$$= -\sum(\quad \log_2(\quad))$$

Where is the probability that the observation belongs to class .

???

We need a way of measuring the purity of a group. This is known as a *purity measure*. Thankfully we've already encountered one purity measure. Entropy.

$$= -\sum(\quad \log_2(\quad))$$

Where is the probability that the observation belongs to class .

For example, if we have two classes:

$$= - \ _1 \log_2(\ _1) - \ _2 \log_2(\ _2)$$

Aside: Probability

Thought experiments:

- flip a coin
- throw a die
- pick a card from a pack

How did you calculate that probability?

A bit more difficult:

- What is the probability that you will next see a woman or a man?
- Given the previous data (two A's and two B's) what is the probability that you would pick an A? What about a B?

Consider the previous example where we had two classes, with two instances in each class:

$$= - \ _1 \log_2(\ _1) - \ _2 \log_2(\ _2)$$

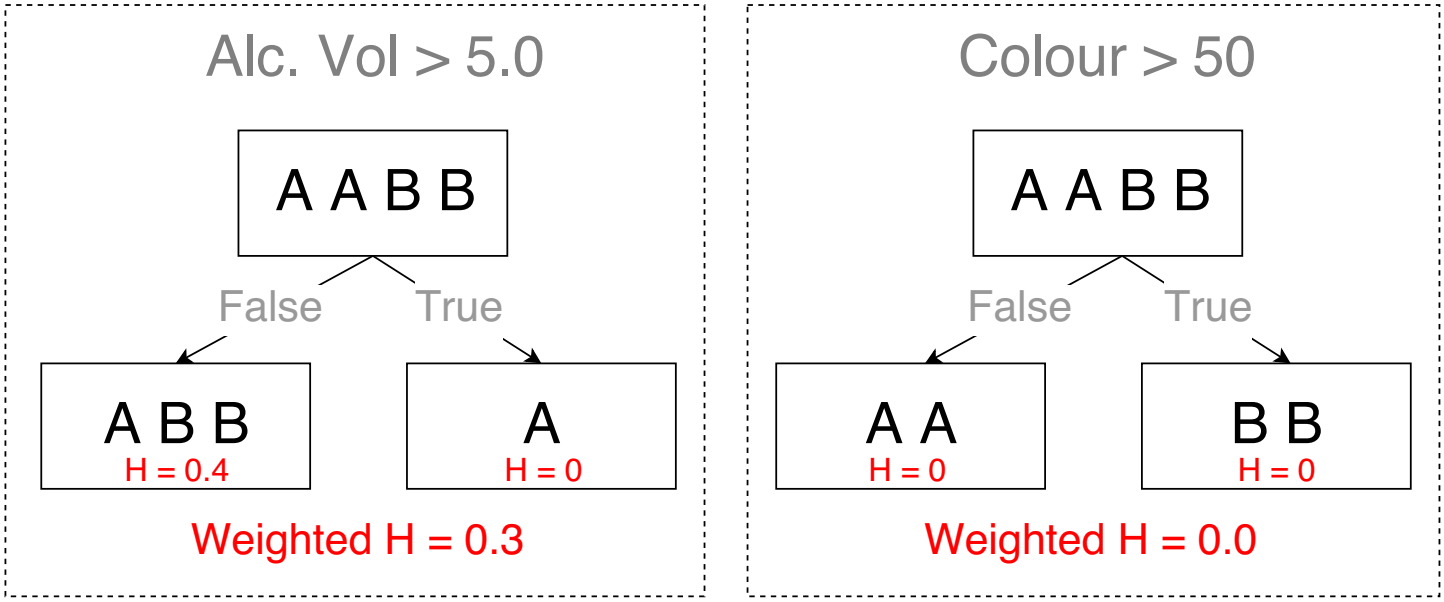
$$= -0.5 \log_2(0.5) - 0.5 \log_2(0.5)$$

$$= 1$$

This is a very impure dataset. Given the observation, this data has high entropy and contains a maximal amount of information.

Remember: We wanted to split the data to make the data *clean* or *pure*.

- Split the data and measure the new entropy of the splits
- If resulting splits are pure (entropy of 0), then we have achieved our goal



???

Entropy measures the amount of information in a given sample.

Looking back to the goal, we want to partition into groups so that when a new observation arrives we can guess which group that belongs to.

To choose the best partition we need to measure which split would give the best result.

We can do that by trying different splits and then measure the entropy again with the goal of making the groups more pure.

We can measure how much purer the new split becomes by computing the *information gain*

Information Gain

The information gain is defined as the parent entropy minus the weighted entropy of the subgroups.

$$(\quad , \quad) = \quad (\quad) - \quad (\quad) \quad (\quad 1) + \quad (\quad 2) \quad (\quad 2) + \ldots) \quad (1)$$

???

Reading the equation, this is the weighted entropy of each new group subtracted from the parent entropy.

I.e. we are calculating the reduction in entropy by creating two new classes.

---

class: pure-table, pure-table-striped

In the previous example the parent's entropy was 1. Lets try splitting the classes in two ways.

Alc. Volume	Colour	Class
4.2	92	A
6.4	102	A
3.5	3	B
4.7	10	B

First Let's split by an `Alc. Volume` of 5.0. This leaves the data:

< 5.0

Alc. Volume	Colour	Class
4.2	92	A
3.5	3	B
4.7	10	B

So, A = 1/3, B = 2/3

>= 5.0

Alc. Volume	Colour	Class
6.4	102	A

and, A = 1/1, B = 0/1

So, given the entropy of the parent (`1.0`) we can now calculate the entropy of the two children:

$$= - \ _1 \log_2(\ _1) - \ _2 \log_2(\ _2) \quad (2)$$

$$(\quad < 5.0) = -0.33 \log_2(0.33) - 0.66 \log_2(0.66) \quad (3)$$

$$(\quad < 5.0) = 0.92 \quad (4)$$

$$(\quad > 5.0) = -1 \log_2(1) \quad (5)$$

$$(\quad > 5.0) = 0 \quad (6)$$

And calculate the information gain as:

$$(\quad , \quad) = \quad (\quad) - \tag{7}$$

$$(\ (\ _1) \quad (\ _1) + \ (\ _2) \quad (\ _2) + \ldots) \tag{8}$$

$$= 1 - \left(\frac{3}{4} \times 0.92 + \frac{1}{4} \times 0 \right) \tag{9}$$

$$= 0.31 \tag{10}$$

We have an information gain of `0.31` with this split. Let's consider another split.

---

class: pure-table, pure-table-striped

Let's split by an `Colour` of 50. This leaves the data:

< 50

Alc. Volume	Colour	Class
3.5	3	B
4.7	10	B

So, A = 0/2, B = 2/2

>= 50

Alc. Volume	Colour	Class
4.2	92	A
6.4	102	A

and, A = 2/2, B = 0/2

$$= - \ _1 \log_2(\ _1) - \ \ _2 \log_2(\ _2) \tag{11}$$
$$(\quad < 5.0) = -1 \log_2(1) - 0 \tag{12}$$
$$(\quad < 5.0) = 0 \tag{13}$$
$$(\quad > 5.0) = -1 \log_2(1) \tag{14}$$
$$(\quad > 5.0) = 0 \tag{15}$$

And calculate the information gain as:

$$(\qquad , \qquad) = \qquad (\qquad) - \tag{16}$$
$$(\ (\ _1) \qquad (\ _1) + \ (\ _2) \qquad (\ _2) + \dots) \tag{17}$$
$$= 1 - \left(\frac{2}{4} \times 0 + \frac{2}{4} \times 0 \right) \tag{18}$$
$$= 1 \tag{19}$$

An information gain of `1`. Comparing the two splits, the first only increased the amount of information by `0.31`. This split increases the IG by a full `1.0`. Clearly, we would choose the second split to split out data.

---

class: pure-table, pure-table-striped

Example: Categorical Segmentation via Information Gain

- Mushroom dataset: interesting because it is purely categorical

poisonous	cap-shape	cap-surface	cap-color	bruises?
p	x	s	n	t
e	x	s	y	t
e	b	s	w	t

p poisonous	x cap-shape	y cap-surface	w cap-color	t bruises?
e	x	s	g	f

???

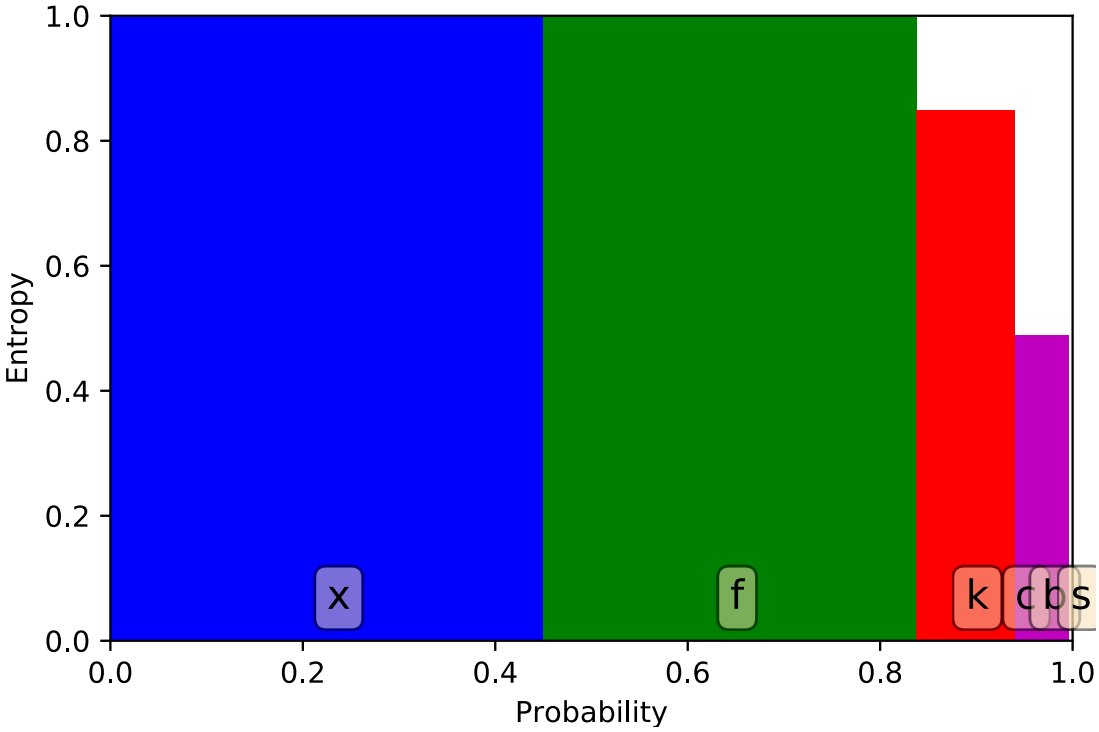
There's a fairly famous dataset called the "mushroom dataset".

It describes whether mushrooms are edible or not, depending on an array of features.

The nice thing about this dataset is that the features are all catagorical.

So we can go through and segment the data for each value in a feature.

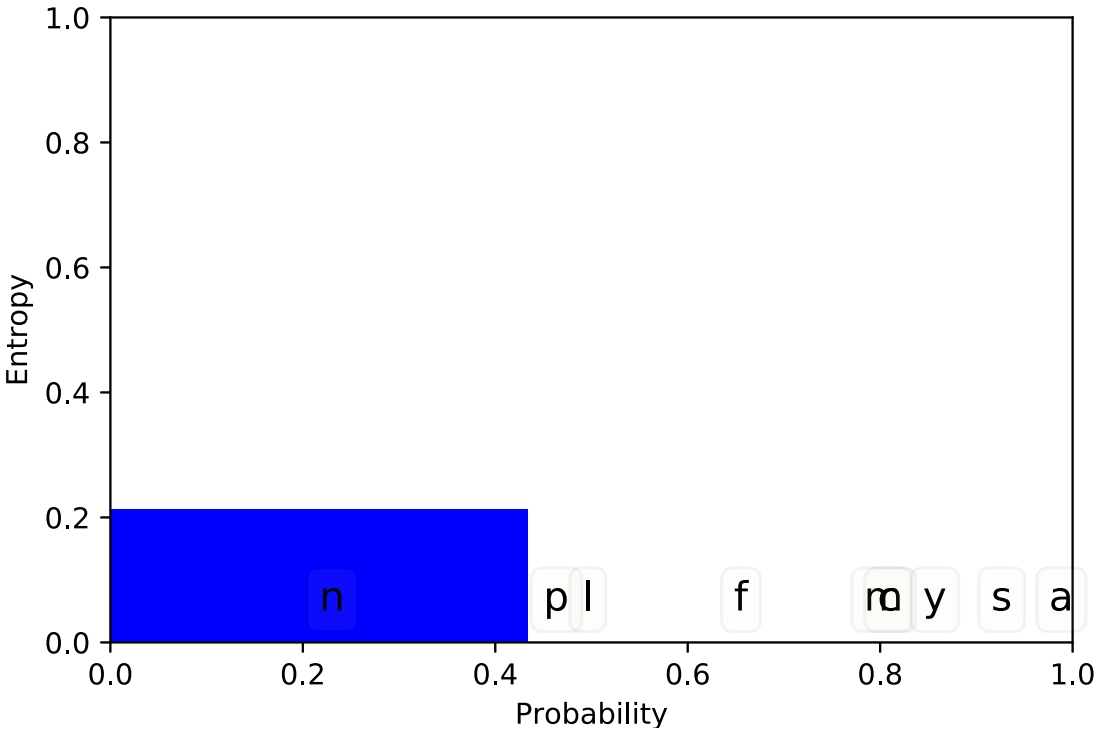
This is the information gain for the "cap-shape" feature:



???

For each feature, we can then plot the information gain visually, which helps. On the x axis we have the proportion of the sample given a feature value, vs the calculated entropy for the label (poisonous, not poisonous).

And this is the information gain for the "odour" feature:



???

We can visually see that the second feature has a much smaller `entropy*probability` area, hence the information gain is much greater.

We would certainly consider using this feature over the other.

Segmentation through trees

- What happens if it isn't possible to make split completely pure?
- Repeat the the segmentation for another feature until it is pure
- When it is stacked like this, it is known as a decision tree

???

Segmentation and quantitative measures of information are fundamental to data science.

If we select the most informative feature then we have segmented the data. It is then quite simple to use that segmentation rule (e.g. < or > 50) to predict the class of a new observation.

But what if a single split didn't completely separate the classes?

One simple solution is to stack segmentation rules. I.e. perform the split again until the classes are pure.

This is known as a tree.

A tree has several components:

- Root node: The starting point
- Interior node: A decision point
- Terminal node/Leaf: A pure node.

Trees are often used as predictive models.

- Given a new observation, follow the decisions made by the tree
- When you arrive at a leaf, that is your prediction

You've just derived the decision tree algorithm!

???

We can continue performing the splits and information gain calculations until all the terminal nodes are pure.

Then if we see a new observation, we can re-run the same rules for that new observation and *predict* it's class.

When trees are used to make a decision they are called *decision trees.* Our first classification algorithm!

Pat yourself on your back. Pat your neighbours back. You've just derived a very important algorithm!

They are popular because:

- They are easy to understand
- Simple and efficient to implement
- Work well with little tuning
- Work with both continuous and categorical data (more on this next)

???

Given some data, we can build an optimal tree structure for our problem. This is called *tree induction.*

The goal of the tree is to provide supervised segmentation; given some labelled data, find the rules, based upon their features, into subgroups that have similar values.

We performed this task manually when discussing information gain. We can iteratively select the best split that maximises the information gain.

Trees as rules

These rules can be easily encoded into software or procedures.

```
if COLOUR < 50 then Class=A else Class=B
```

???

One of the beauties of tree-based classification algorithms is that they are purely logical.

It is easy to convert simple trees into a set of rules.

For example, our beer example could be summarised as:

Example: Customer churn

- Mobile phone operator
- Trying to predict who is likely to leave their contract so they can entice people to stay

```
Index(['COLLEGE', 'INCOME', 'OVERAGE', 'LEFTOVER', 'HOUSE', 'HANDSET_PRICE',
       'OVER_15MINS_CALLS_PER_MONTH', 'AVERAGE_CALL_DURATION',
       'REPORTED_SATISFACTION', 'REPORTED_USAGE_LEVEL',
       'CONSIDERING_CHANGE_OF_PLAN'],
      dtype='object')
```

In general, these types of problems result in a simple procedural rule.

???

This is an interesting dataset describing whether users of a mobile phone service left for another provider. This is known as "customer churn".

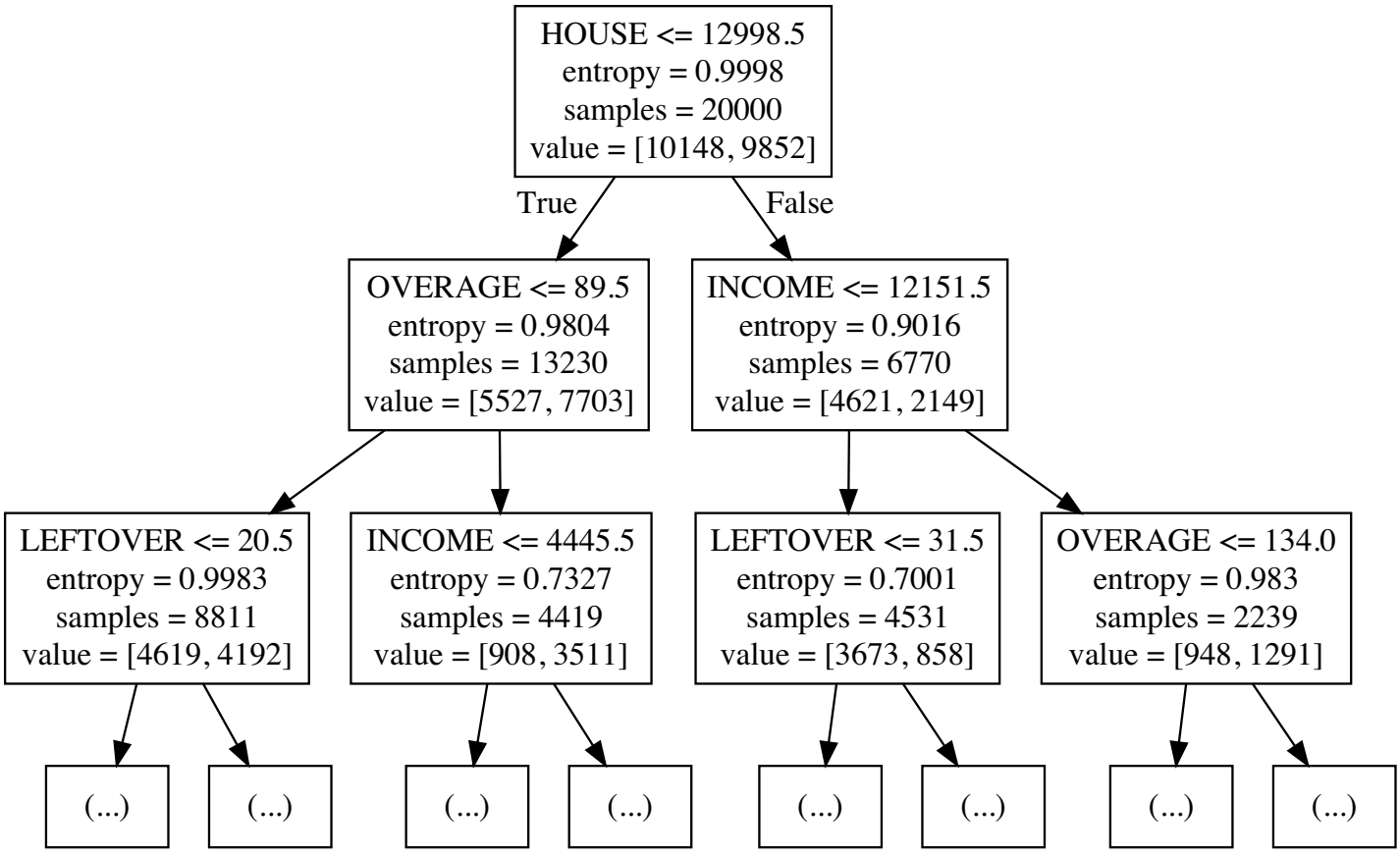
The goal of this dataset is to predict wheter a customer is going to leave or not from some other features.

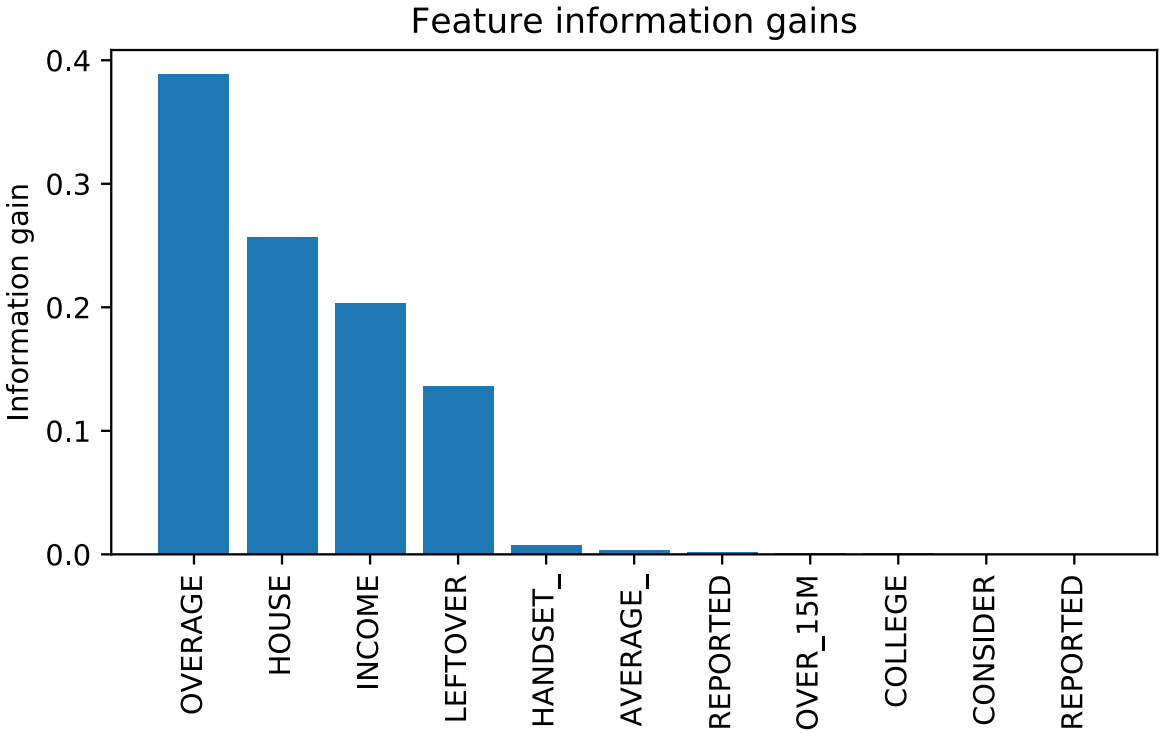
I won't go into details just yet, because we're not quite ready. But I pushed this data through an algorithm to create a decision tree, the calculated the information gain given each of the generated rules.

E.g. "Give customer an offer. If they call and ask for a discount, give it to them."

Very annoying for the customer. But has produced lots of profit.

The resulting rules look like...





.center[ ]

---

class: middle, center

Workshop 2

---

name: data-and-feature-engineering

Data and Feature engineering

Your job depends on your data

The goal of this section is to:

- Talk about what data is and the context provided by your domain
- Discover how to massage data to produce the best results
- Find out how and where we can discover new data

???

If you have inadequate data you will not be able to succeed in any data science task.

More generally, I want you to focus on your data. It is necessary to understand your

data before building a model.

Each algorithm has different characteristics that work better with certain types of data.

Data in your domain

- Engineers are often not domain experts
- You need to be

The best models are the simplest models

Tip: Domain experts already use models, find out what they are

???

One important aspect that is often overlooked is experience within a domain.

Engineers are often not domain experts. This affects their ability to question data, build models and assess results.

For example, is a 53% accuracy (just better than flipping a coin) in predicting the direction of the stock market good or bad? (Hint: it's good)

When you are not an expert in the domain of the data, either learn to be an expert or make sure you find someone who is.

Think of data as an asset

Data, and the ability to extract valuable knowledge from data, should be regarded as a strategic asset

???

Too often, data science projects are started on the premise of extracting value from existing data.

Current data might not be the best data to perform a new task.

Always consider whether it would be easier and/or cheaper to collect new data, rather than trying to hammer circular data into a square hole.

Companies that are driven by data are more efficient; hence more profitable more quickly.

Investments in data, data science and engineers pays dividends.

Can be costly and time consuming

Do not underestimate the cost (usually in time) that it takes to collect, parse and understand the data.

???

The cost of acquiring data can vary. Some data is free, some data will need to be purchased. Some data will need a bespoke process to collect.

It is worth spending time to ascertain whether the costs and benefits of collecting and using data are worthwhile.

Collecting and investigating data is a significant, if not one of the largest uses of time in data science. Because the data are so fundamental to a problem, large gains can be found at a very low level. For example, adding, altering or removing features.

Must understand the data

Once the data is collected, it must be understood.

- What are you expecting? (your priors)
- When is that data available? (data leakage)
- What type of data is it? (continuous or categorical, stochastic or set?)

Tip: Do things manually first. Then you can validate automation.

???

You cannot expect to run data you do not understand through an algorithm and get

a result.

First, you need to know what sort of result you should be expecting. Only you can tell whether the results are promising or misleading; no number of metrics can do that (yet!).

Furthermore, some algorithms are better suited to certain types of data.

For example, you wouldn't trying to push complex categorical data through a linear model. And you should notice if one of the observations are clearly erroneous.

All of this is only possible if you understand the data that is going into the model.

Bias in data collection

- Is the data representative?
- Is common-knowledge really that common?
- How was the data recorded? Has that created a subset?

Tip: Assume that you are biased. Assume that the data is biased.

???

Do not be fooled into thinking that the data represents the objective truth.

Like "there is no such thing as 100%", there is also no such thing as a representative dataset.

One common example is the black swan theory. The word "black swan" was a common expression up to 17th century, which had a similar meaning to what "when pigs fly" means today. In 1697 Dutch explorers found a black swan in Australia.

More routinely though, biases creep into data collection. For example, if we were handed a dataset from a bank, with information about people that were given a loan, a common question might be "what sort of people should we give a loan to".

But of course, the data is biased. It only contains information about loans decided upon by a previous person or policy.

Be sure that your data is a good (enough) representation of the entire population,

not just a subset.

Preprocessing data

- Black boxes don't work. Expert models might work.
- No free lunch

Preprocessing data is the task of taking the raw data and attempting to fit the data to those assumptions so the algorithm can do its job more effectively.

???

Once the data has been collected we cannot simply throw the data into a black box and expect to produce something meaningful.

A theory exists called "no free lunch" which states (in a highly mathematical way) that no one technique (algorithm, optimisation function, etc.) can perform equally well on all kinds of data.

We create models to simplify the real world. To distill a set of measurements into a form that is easier to understand. All models make certain assumptions during this simplification.

Preprocessing data is the task of taking the raw data and attempting to fit the data to those assumptions so the algorithm can do its job more effectively.

Characteristics of your data

- Statistics 101
- All data has some "character"
- We need to visualise that character to improve our understanding of the data
- Histograms are one way of doing this

The *mean* and *standard deviation* are measures of that character.

???

Before we start massaging the data into shape, lets have a quick statistics lesson.

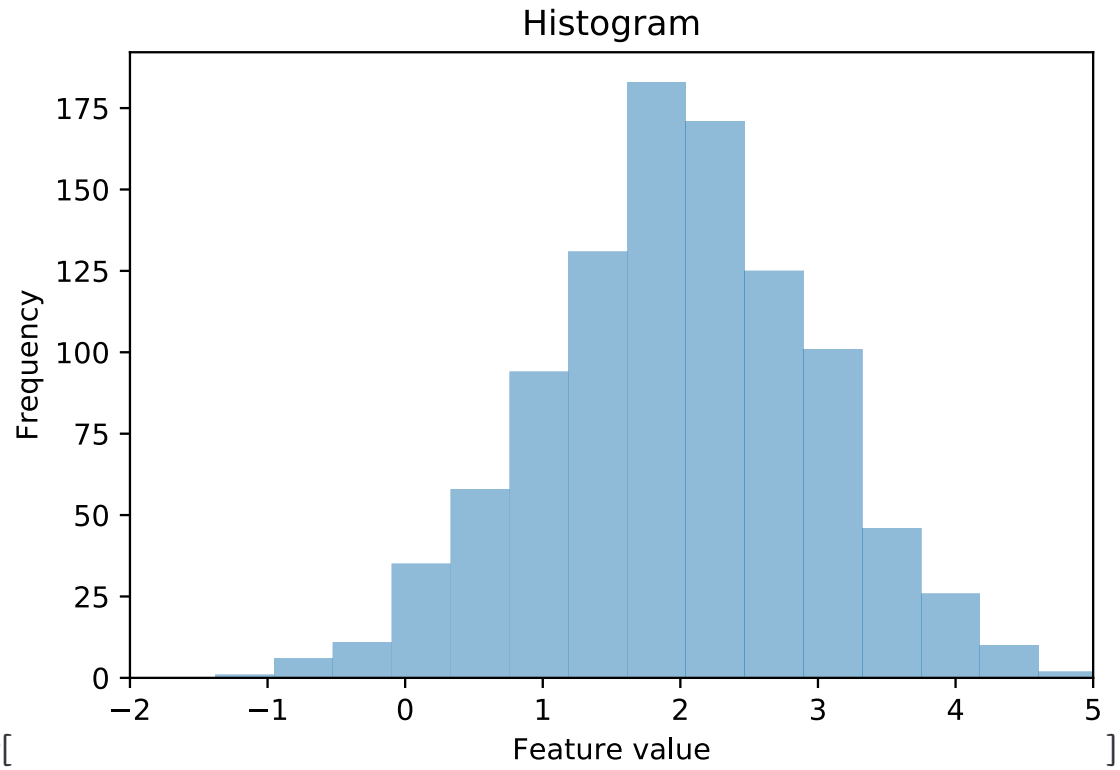
Each feature that we sample has a character. It produces values in very characteristic ways. It is uncommon for values to be produced that are uncharacteristic.

If we take the values of the feature, and assign them to bins, we can count the number of times a value falls into a bin. This is called a histogram.

Histograms expose the true nature of the data, that the data is formed by some natural process. The underlying model that produces this data is called a distribution. It perfectly describes any new potential value that could arise from that process in only a handful of numbers.

---

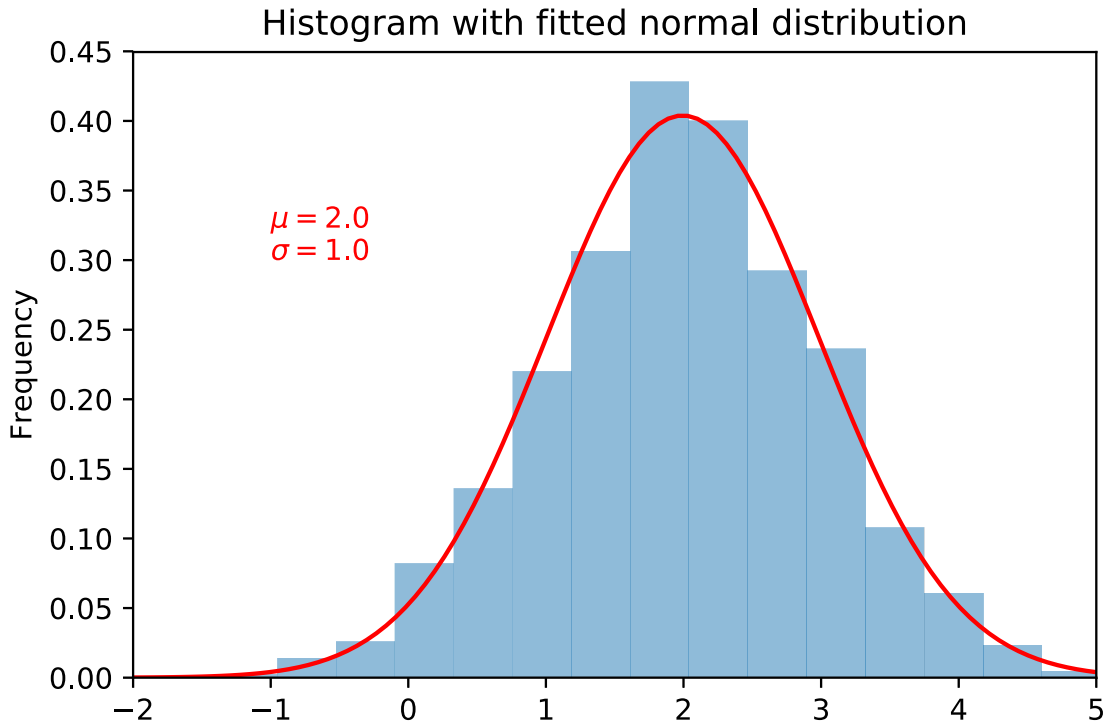
class: middle



.center[ ]

---

class: middle



.center[ ]

---

class: middle

.center[

]

The bad news

Many algorithms, from optimisers to learning algorithms, assume that your data is normally distributed

???

- Spend time trying to normalise your data (performing tricks to make the data look more normal)
- Be aware of when your data isn't normal, bare that in mind when running through algorithms
- Some types of data might have a different distribution and that distribution is inherent to the problem. Don't try to normalise this data.
- Plotting histograms of the features allows you to "get a feeling" for the data. Even if you don't use it to improve the data, it is essential for data understanding.
- Histograms might reveal important characteristics within the data. For example, are there groups of data?
- Histograms are also an important first step of visualising problem data

---

Scaling

Nearly all algorithms expect your data to have equal scales.

- Due to scale being interpreted as "important"

(Decision trees are one of the only algorithms that you can ignore scale)

???

Many (in fact, all except for decision trees) algorithms expect the data in a certain format. Chief among these is the requirement for all features to have the same scale.

Imagine we are fitting a two dimensional linear model to some data. Linear algorithms do their job by minimising the error between the predicted result and the data.

If one of the features is ten times larger than the other, then the errors produced by a prediction would also be 10^2 larger. Hence, the optimisation algorithm trying to minimise that error will spend more time trying to reduce the larger error, because that is the larger "slope".

Obviously this is wrong, we don't want to treat any features as more important than another (unless we do, and if we do there's better ways of doing it).

To fix the problem, we need to scale the larger feature so it has similar

characteristics to the smaller.

---

class: pure-table, pure-table-striped

Scaling options

Name	Result	When?
StandardScaler	Zero mean and variance of one	Most of the time
MinMaxScaler, MaxAbsScaler	Rescale feature to lie between zero and one	When data really can be zero, catagorical values
RobustScaler	A version of StandardScaler that is less sensitive to outliers	When there are outliers

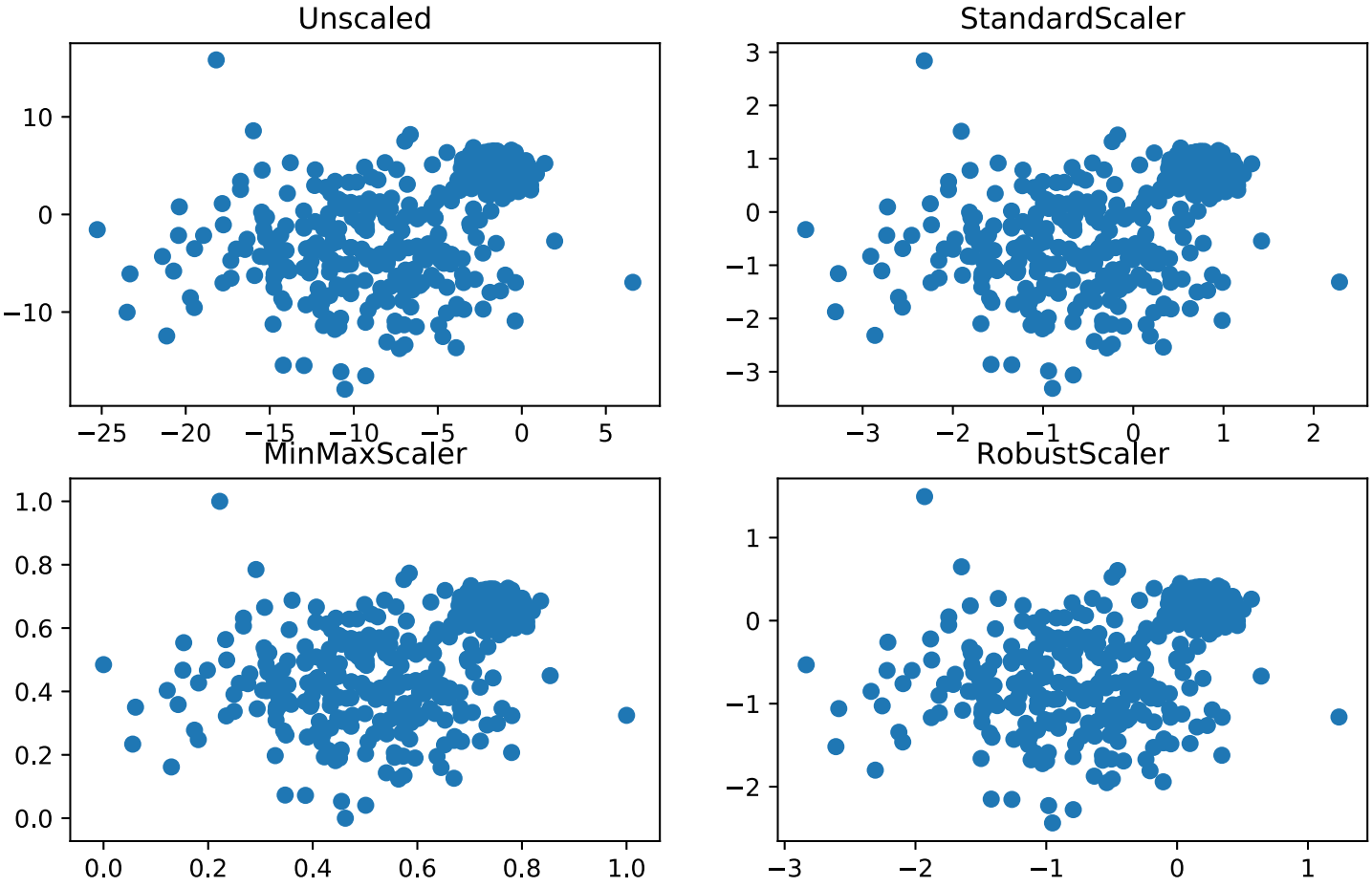
Most of the time we will use the `StandardScaler`. Lets look at an example...

---

class: middle

Example

.center[



]

Missing values

Most if not all algorithms expect data to be numeric and that they have meaning.

- Remove rows or columns
- *Impute* missing values

???

Most real world data is not perfect. It has missing or invalid values like blanks, NaNs, or Nulls.

The first strategy, and the simplest if you can afford it, is to drop the observations that have invalid data.

If you can't drop data, we can attempt to fill or *impute* the values; i.e. infer them from the known part of the data.

```
>>> import numpy as np
```

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
Imputer(axis=0, copy=True, missing_values='NaN', strategy='mean', verbose=0)
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[ 4.          2.        ]
 [ 6.          3.666...]
 [ 7.          6.        ]]
```

class: pure-table, pure-table-striped

Categorical features

If necessary, we can convert categorical features into continuous ones:

- Create new columns for each possible category

```
X = [
    {'sex': 'female', 'location': 'Europe', 'age': 33},
    {'sex': 'male', 'location': 'US', 'age': 65},
    {'sex': 'female', 'location': 'Asia', 'age': 48},
]
```

age	location=Asia	location=Europe	location=US	sex=female	sex=male
33.0	0.0	1.0	0.0	1.0	0.0
65.0	0.0	0.0	1.0	0.0	1.0
48.0	1.0	0.0	0.0	1.0	0.0

???

When data is presented as a dictionary, we can use a `DictVectorizer`.

```
from sklearn.feature_extraction import DictVectorizer
```

```
X = [
    {'sex': 'female', 'location': 'Europe', 'age': 33},
    {'sex': 'male', 'location': 'US', 'age': 65},
    {'sex': 'female', 'location': 'Asia', 'age': 48},
]
vec = DictVectorizer()
X_v = vec.fit_transform(X).toarray()
df = pd.DataFrame(X_v, columns=vec.feature_names_)
print(pandas_df_to_markdown_table(df).data)
```

age	location=Asia	location=Europe	location=US	sex=female	sex=male
33.0	0.0	1.0	0.0	1.0	0.0
65.0	0.0	0.0	1.0	0.0	1.0
48.0	1.0	0.0	0.0	1.0	0.0

Also see: `pandas.get_dummies()` performs a similar function.

Sometimes data is not presented as a dictionary, but still catagorical. For example, consider the previous example where we have used a numerical value that represents the different characteristics of the people (0 = 'female', 1 = 'male', etc.)

```
from sklearn.preprocessing import OneHotEncoder

X_v = [[33, 0, 0], [65, 1, 1], [48, 0, 2]]
enc = OneHotEncoder()
X_e = enc.fit_transform(X_v)
print("One hot encoded\n", X_e.toarray())

One hot encoded
 [[ 1.  0.  0.  1.  0.  1.  0.  0.]
 [ 0.  0.  1.  0.  1.  0.  1.  0.]
 [ 0.  1.  0.  1.  0.  0.  0.  1.]]
```

class: pure-table, pure-table-striped

Creating Categories

Sometimes it helps to add categories, rather than working with continuous variables.

- Create bins, like a histogram, and use a 1 to signify the bin

age	0–10	10–20	20–30	30–40	40–50
33.0	0.0	1.0	0.0	0.0	0.0
65.0	0.0	0.0	1.0	0.0	0.0
48.0	0.0	0.0	0.0	1.0	0.0

Save your preprocessors!

A quick note: remember to keep track of your preprocessors!

Because you've trained your models on this preprocessed data, the model will only work on data that has also been preprocessed. I.e. you also need to perform the same preprocessing on new data.

Engineering features

You want to do this because:

- Reduces the number of features without losing information
- Better features than the original
- Make data more suitable for training

???

Another part of the data wrangling challenge is to create better features from current ones.

Distribution/Model specific rescaling

Most models expect normally distributed data. If you can, transform the data to be normal.

- Infer the distribution from the histogram (and confirm by fitting distributions)
- Apply the inverse transformation to re-normalise

Some example types of data that follow these distributions: - natural-log: The length of comments posted in Internet discussion forums - natural-log: Time to repair a maintainable system - power-law: The Severity of violence - power-law: The relationship between a CPU's cache size and the number of cache misses

???

We touched upon this earlier, but you may be able to transform the data directly into normally distributed data.

Very common in physical problems, log scaling is required in situations where the underlying process is affected by a power law (e.g. exponential growth in y for a linear increase in x).

You may ask "how am I supposed to know that!?". This is where domain knowledge comes into play. But most often these things are most obvious when plotting the histogram.

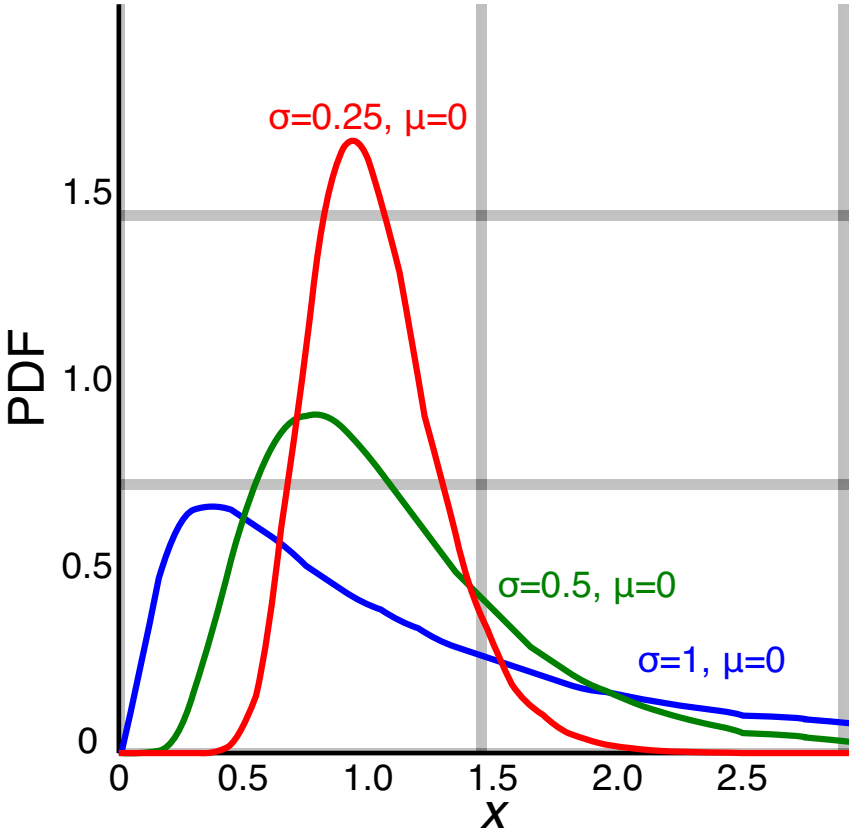
Other types of scaling include: square, square root, natural-log.

Some example types of data that follow these distributions: - natural-log: The length of comments posted in Internet discussion forums - natural-log: Time to repair a maintainable system - power-law: The Severity of violence - power-law: The relationship between a CPU's cache size and the number of cache misses

---

Data affected by a power law (exponential distribution):

Data affected by a natural logarithm (log-normal distribution):



Combining features

- Many domains have already encoded important combinations of features:
  - velocity, body mass index, price-earnings ratio, queries per second, etc.

Looking for new combinations can be lucrative.

???

Rescaling has become a standard pre-processing technique in data science. But feature combination is less prevalent; presumably because it requires some domain knowledge.

But this is at odds with the rest of the world, because there are many, many examples of combined features being used in everyday life: velocity, body mass index, price-earnings ratio, queries per second, etc.

For domain experts, many of these types of features may be known. However, looking for new combinations or transformations can be lucrative.

Types of combinations

- Multiplication or division (ratio)
- Change over time, or rate
- Subtraction of a baseline
- Normalisation: Normalising one variable with respect to another. E.g. the number of failures in itself is probably not that useful. However a failure rate as a percentage could be very useful. (i.e. number of failures / total requests)

???

There are many ways in which to combine features, but these are the most common:

When creating new variables, always bear in mind the goal. The purpose of creating new features is to provide new information that make the goal easier to achieve.

Example: Polynomial features

`sklearn` provides us with one easy feature generation method called `PolynomialFeatures`.

This gnerates high-order interaction terms with each of the features. For example, the dataset:

$$\mathbf{X} = [\quad,\quad]$$

becomes:

$$\mathbf{X}_\mathrm{p} = \left[\quad,\quad,\quad\times\quad,\quad^2,\quad^2,\ldots\right]$$

???

```
>>> import numpy as np
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[  1.,   0.,   1.,   0.,   0.,   1.],
       [  1.,   2.,   3.,   4.,   6.,   9.],
       [  1.,   4.,   5.,  16.,  20.,  25.]])
```

Feature Selection

Tip: Minimise the number of features

But how do we select which features to use?

???

The problem with generating lots of features is that this increases the changes of *overfitting* (which we will discuss in the next section). To mitigate this we want to reduce the number of features.

Assuming we're not domain experts quite yet (or even if we are), how do we decide which features to include in our training dataset?

How can we tell which features are the most important given our goal?

In the previous section we saw a technique to partition data based upon information gain. We will see this again soon…

Correlation

Correlating features in datasets are bad.

- Algorithms think that duplicate information represents importance
- Correlating features don't add any more information

???

Recap: Correlation is a measure of the statistical dependence between two quantities. I.e. if one features changes, does another feature change in lock-step?

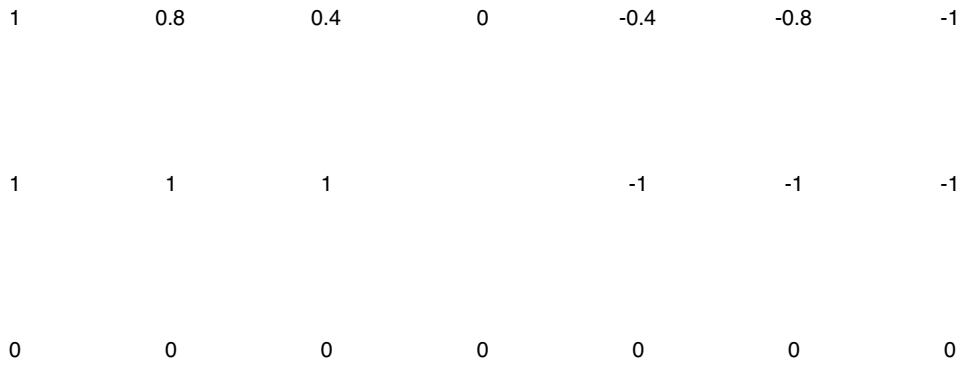
At worst, the feature doesn't add any more information. We can obtain the same amount of information from one feature alone.

In reality, correlating features tend to reduce performance due to the optimisation problem we saw in the scaling section. The optimisation function will expend effort trying to optimise these redundant variables.

At best, it will simply take a lot longer to reach the same result (this is only true for "sparse" optimisers - those that penalise large numbers of features).

At worst, the model will start overfitting because of the extra features (i.e. because you have two features saying the same thing, that must be REALLY important!)

This is a plot of the *Pearson Correlation Coefficient* for different 2D datasets. The datasets with a coefficient near 1 or -1 are highly correlated and will affect the performance of your model.



.center[ ]

When features do correlate with each other, they are said to be *collinear*. I.e. they are linearly dependent on each other.

When comparing the colinearity between all features (i.e. first with second, first with third, etc.) this is called testing for *multicollinearity*.

- Simple: visualise the data
- Automated: Inspect the *eigenvalues* of the correlation matrix.

???

Eigenvector Analysis

Our goal is to pick features that do not correlate. (remember the 2D plot here).

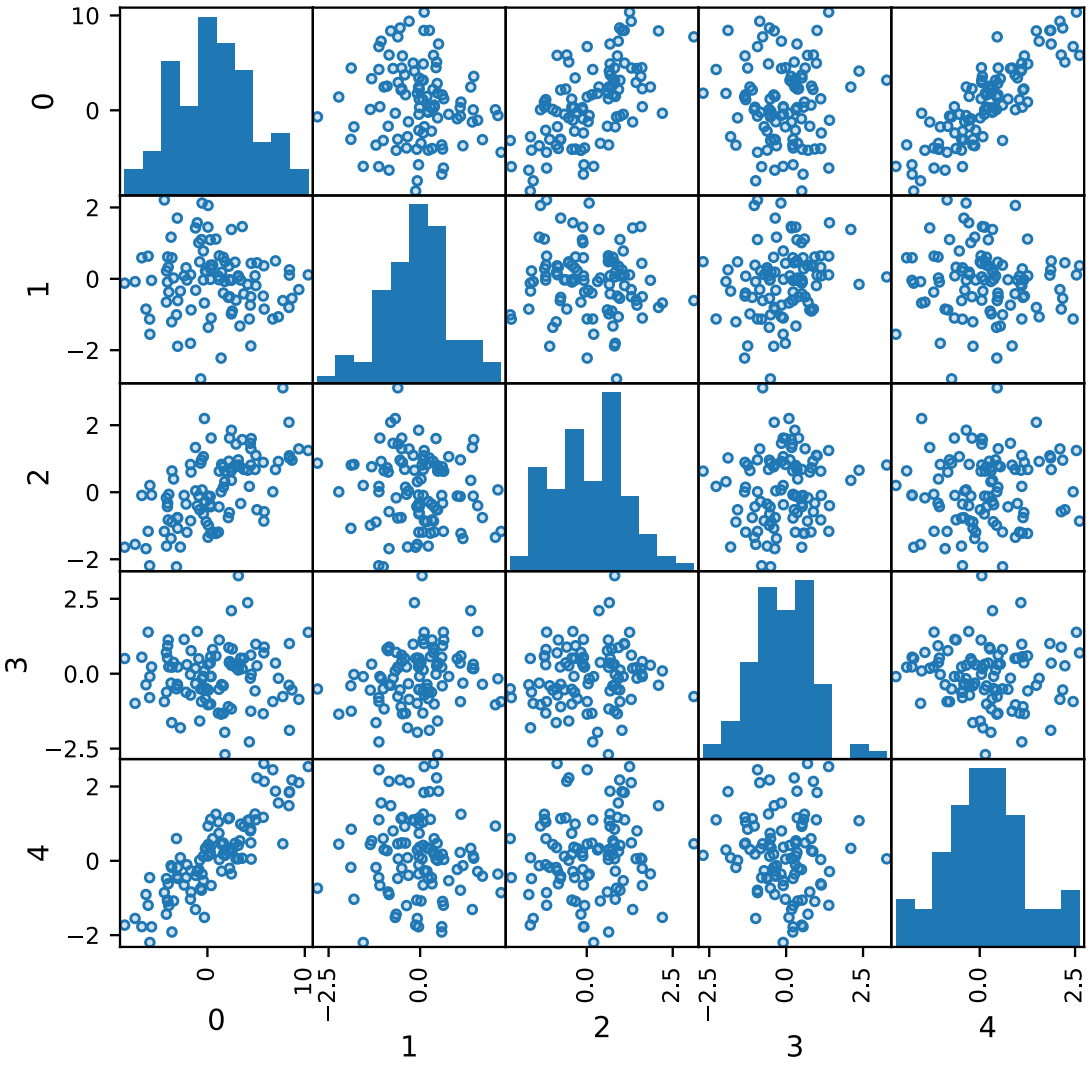
Hence, if we look for very low eigenvalues, this tells us exactly which features have highly correlating features.

Once we know which features are correlating we can look at the eigenvector for that eigenvalue to show us which of the features it correlated with!

This is a bit complex, so let's look at a concrete example...

Example: Correlating features

```
X = np.random.randn(100,5)
noise = np.random.randn(100)
X[:,0] = 2*X[:,2] + 3*X[:, 4] + 0.5*noise
```



Let's calculate the correlation, then the eigenvalues for this correlation matrix:

```
corr = np.corrcoef(X, rowvar=0)
w, v = np.linalg.eig(corr)
print('Eigenvalues of features in X')
print(w)

Eigenvalues of features in X
[ 2.13215129  0.00826567  1.20093744  0.97602299  0.68262261]
```

Here, we have an low valued eigenvalue. This is saying, in one of the feature combinations, there is little variance (i.e. little information).

Note that the order of the eigenvalues is not guaranteed; it does not represent a feature.

Now, let's view the eigenvectors for that eigenvalue...

```
print('Eigenvector for eigenvalue 1')
```

```
print(v[:, 1])

Eigenvector for eigenvalue 1
[ 0.72545976 -0.00837951 -0.37350552  0.00075081 -0.57804064]
```

This array is stating that the direction in which we have little variance is in the 0th, 2nd and 4th features. (remember the eigenvector is the axis that skewers the data in a direction).

So, we need to remove either one (or more) of these features.

(Probably the first, since that appears to have the largest correlation with the other two)...

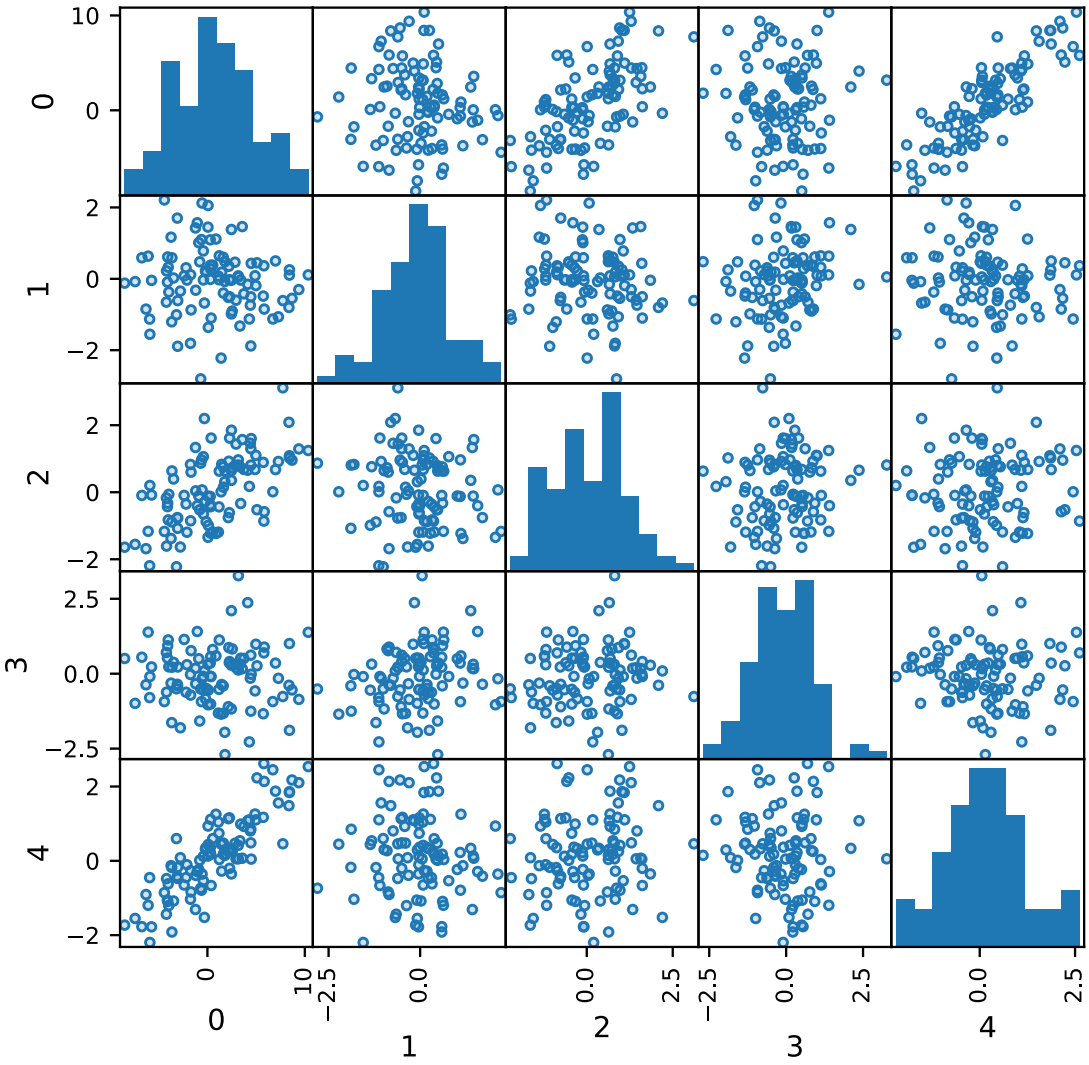
```
corr = np.corrcoef(X[:,1:], rowvar=0)
w, v = np.linalg.eig(corr)
print('Eigenvalues of features in X')
print(w)

Eigenvalues of features in X
[ 0.68138777  0.92924905  1.25915791  1.13020527]
```

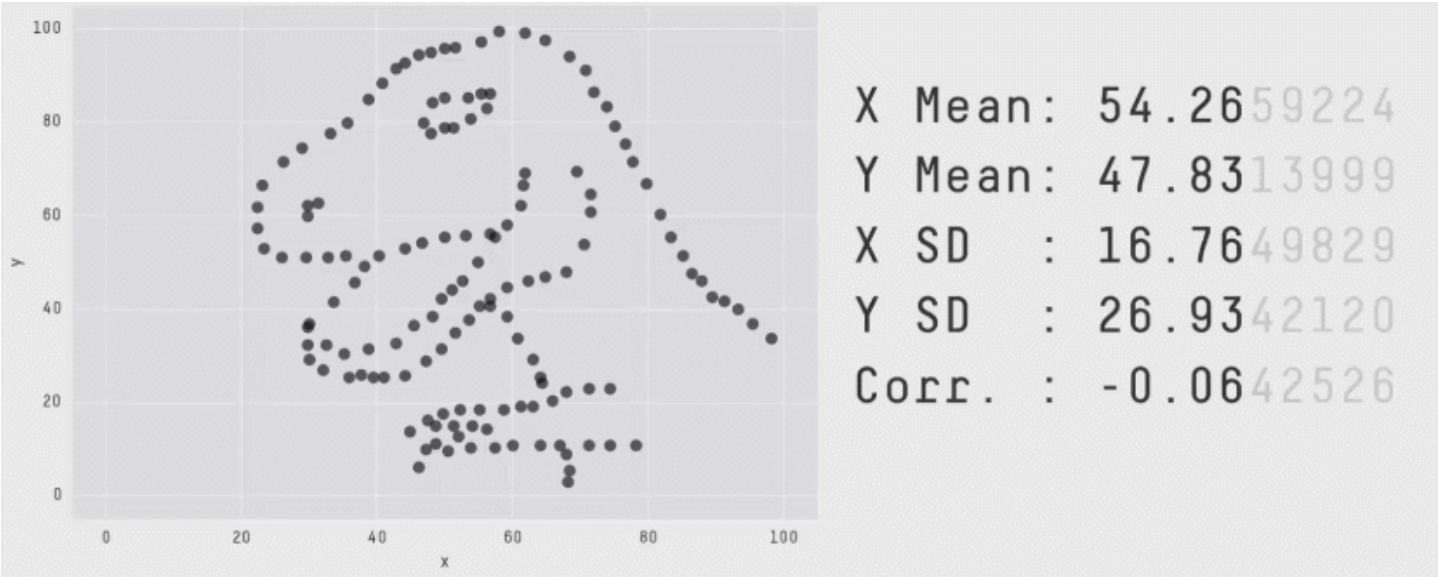
No more correlating features!

Example: Correlating features

```
X = np.random.randn(100,5)
noise = np.random.randn(100)
X[:,0] = 2*X[:,2] + 3*X[:, 4] + 0.5*noise
```



class: center, middle



> *Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing* – *Justin Matejka*, *George Fitzmaurice*

Feature importance

Two general ways to infer importance:

- Brute force: test your model performance using different combinations of features
- Infer importance from another type of score

???

After we've removed correlating features, we're well on out way to defining the most informative features.

However, how can we tell which features are the most important?

One way is to use a single feature to generate a score. The features that generate the highest scores can be said to be the most *informative*.

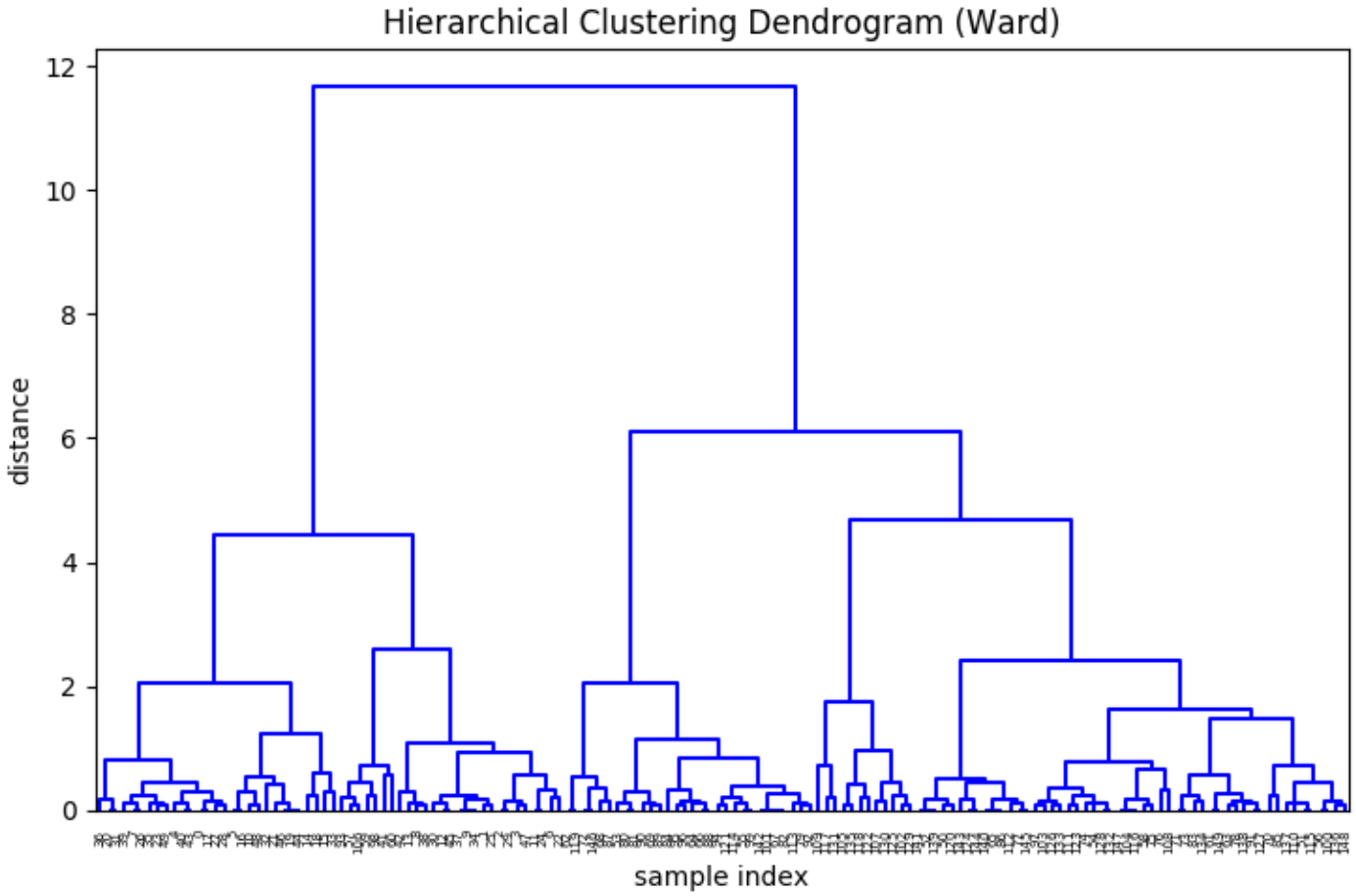
`sklearn` has one dedicated method using trees...

Feature importance with trees

We've just touched upon one type of classification/regression algorithm, decision trees.

We saw that the tree attempts to segment the data by increasing the purity of the classes.

Remember that trees build up their segmentation tree agglomeratively. I.e. highest levels in the tree make the broadest decisions. We can then say that the highest levels of the tree are the most informative.



Example: feature importance with random forests

Finally, we can't rely on one decision tree. Decision trees tend to overfit data. What we can do is to create lots of trees, and randomise the data going into each tree. This gives us some statistical estimate of how stable the feature importance is.

Let's give this a try on some synthetic data...

```
# Build a classification task using 3 informative features
X, y = make_classification(n_samples=1000,
                           n_features=10,
                           ...

# Build a forest and compute the feature importances
forest = ExtraTreesClassifier(n_estimators=250,
                              random_state=0)

forest.fit(X, y)
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_],
             axis=0)
```

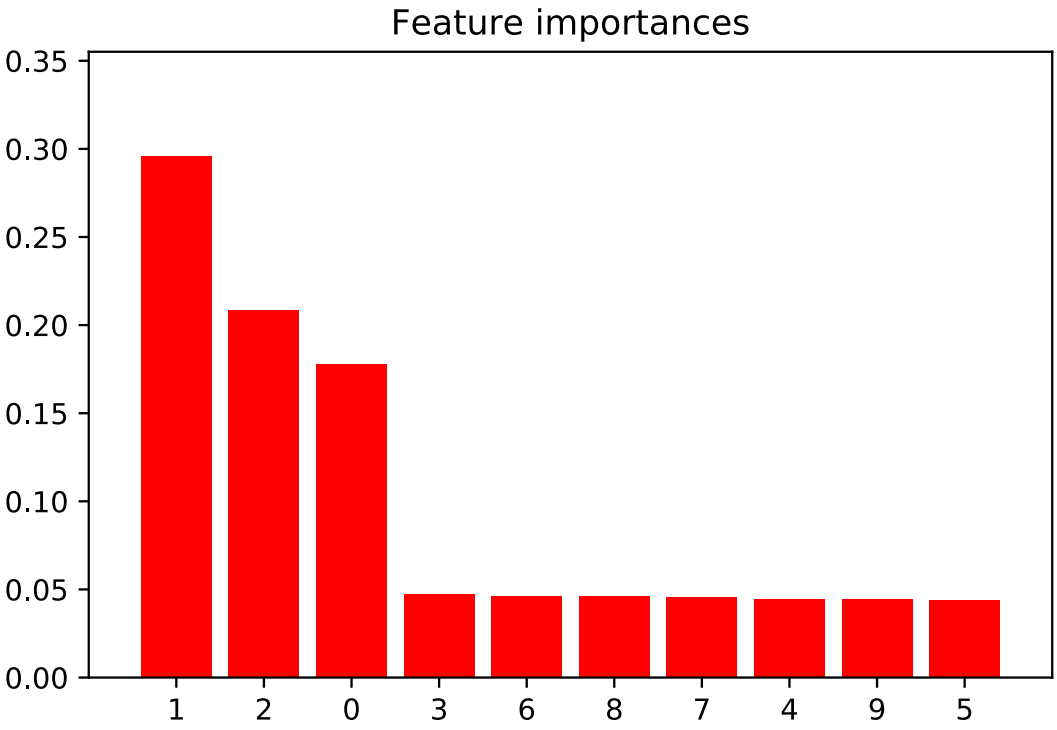
```
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(X.shape[1]):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))

# Plot the feature importances of the forest
...
```

```
Feature ranking:
1. feature 1 (0.295902)
2. feature 2 (0.208351)
3. feature 0 (0.177632)
4. feature 3 (0.047121)
5. feature 6 (0.046303)
6. feature 8 (0.046013)
7. feature 7 (0.045575)
8. feature 4 (0.044614)
9. feature 9 (0.044577)
10. feature 5 (0.043912)
```



Note that the information gain does not represent how "important" each feature is.

Just how well we can separate classes with simple rules.

???

The y-axis can be interpreted as the proportion of importance over the classification in the dataset.

It does not represent how much the feature "explains the data". The scores are simply a ratio of the numbers of classifications that were made when using that feature alone.

One caveat is related to the implementation of the tree optimisation. Tree algorithms pick the next best feature to segment the data. This means that if there are correlating features, only one will be used to segment the data, since once segmented, the same correlating feature will not produce any purer classes.

The result is that of the correlating features, only one will be shown as important. The others will be ranked as not important because they are not used.

Another caveat worth mentioning is the choice of score being used in the tree algorithm. Because we're aiming to promote more informative features, we should chose a score that is based on a measure of information; i.e. `entropy`.

However, if the goal is to produce better separating classes, then consider using `gini` metric.

Sequential feature selection

1. Iteratively add or remove features whilst measuring performance.
2. Remove features that don't improve performance

Pros:

- Related to your chosen performance metric

Cons:

- Computationally expensive

???

A similar method of choosing features is to iteratively add or remove features from a dataset and recompute the score.

For example, if we had the iris dataset with three features, we would start by testing each feature as a classifier. We would pick the best scoring feature and use that as our "single feature" score.

Next, we would use the first feature and concatenate a second feature from the remaining two. Again the best combination will be saved as the best "two feature" score.

And so on.

We could then make a choice to remove features that were not increasing the score significantly.

This algorithm isn't difficult to code, but is not available in `sklearn`.

The next best recommended library for implementing this is the `mlxtend` library: https://github.com/rasbt/mlxtend. It's also got some handy plotting functions.

Sparse models through regularisation

- Some models can automatically penalise features
- They do this by setting feature weights to zero
- Robust, because it is within the context of the model

Still remove features manually. It improves understanding and computational efficiency.

???

One final thing I want to mention is sparse regularisation inside the learning algorithms.

Some algorithms implement *regularisation*, which is the act of penalising the use of large numbers of features in the model.

This is a very robust method of removing uninformative features, because they are removing them within the context of the model.

For example, features selected by a decision tree (a simple, linear thresholding like algorithm) might not choose the best features for a highly complex nonlinear deep learning algorithm that is capable of extraordinary combinations of features.

However, in my experience it is certainly worth trying to remove very uninformative and correlating features, purely to reduce the computation load of having lots of features.

If you can get it down to two or three dimensions, then this helps with plotting too.

Obtaining data

I was going to talk about how to get data. But the reality is that it depends entirely on the domain that you are working in.

There are a huge amount of test and dummy datasets available freely on the internet and you can get access to streaming data from the likes of twitter, etc.

But the reality is that it depends entirely on your problem.

From my experience, the most valuable data is proprietary, and specific to the business. These are the types of data that tend to produce the most valuable outcomes.

For everything else, search for it.

---

class: middle, center

Workshop 3

---

name: regression

Regression and Linear Classifiers

Traditional linear regression (a.k.a. Ordinary Least Squares) is the simplest and

classic form of regression.

Given a linear model in the form of:

$$(\mathbf{x}) = \quad _0 + \quad _1 \ _1 + \quad _2 \ _2 + \cdots \tag{20}$$

$$= \mathbf{w} \quad \cdot \mathbf{x} \tag{21}$$

Linear regression finds the parameters w that minimises the *mean squared error* (MSE)…

The MSE is the sum of the squared values between the predicted value and the actual value.

$$(\mathbf{w}) = \frac{1}{} \sum_{=1} (\mathbf{w} \quad \cdot \mathbf{x}^{(\ )} - \mathbf{y}^{(\ )})^2 \tag{22}$$

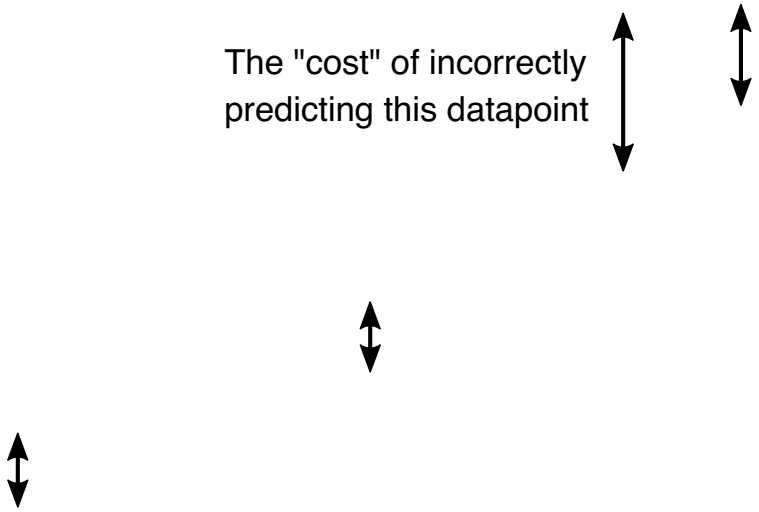
Where is the number of observations.

Cost function

- The MSE is known as a *cost function*.

We can also use a measure of how good a fit is (*utility* or *fitness function*).

- We can then tweak the parameters to minimise the cost function



???

The typical use for a cost function is to measure how badly a model fits or classifies the data. The model is then tweaked to minimise the cost function. E.g. For linear regression we often use the distance between the predictions from the model and the real data; the MSE in the previous example.

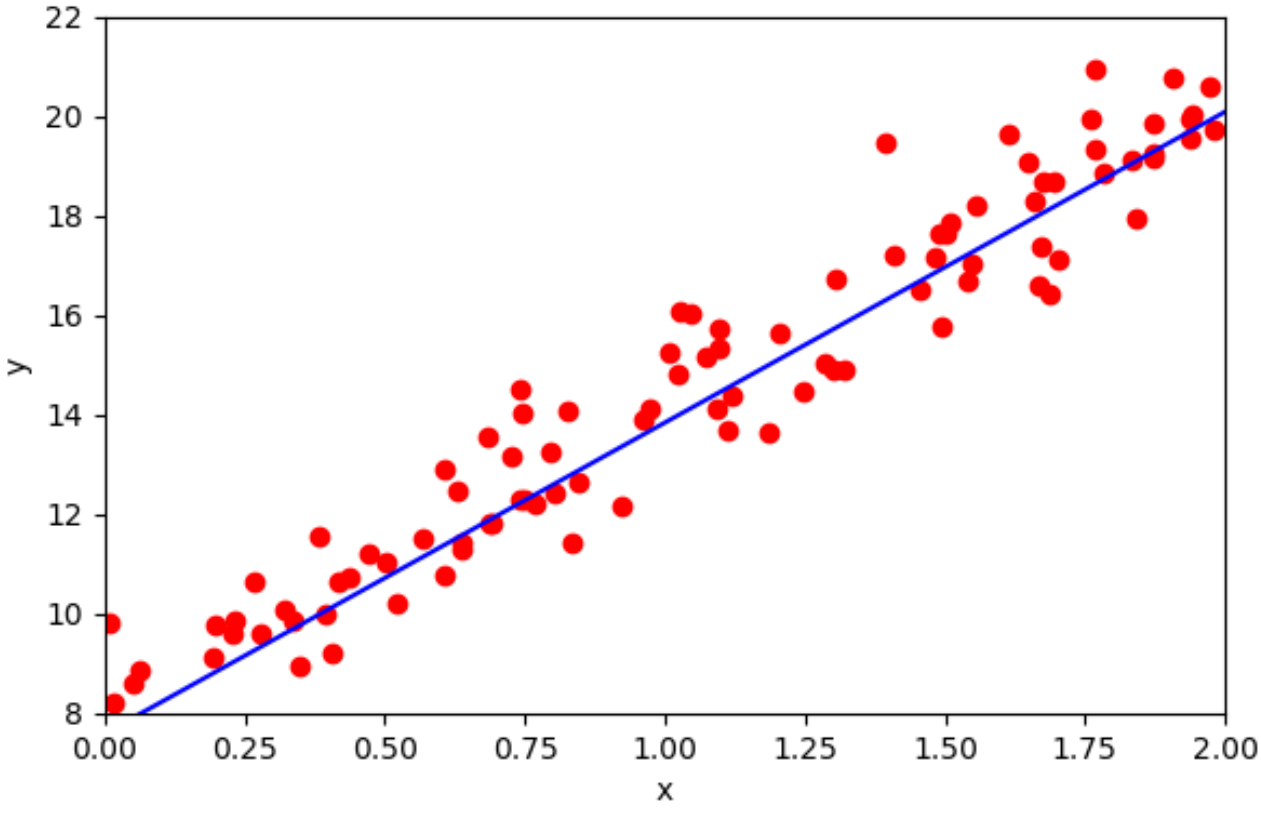
Normal equation

There is a closed solution to find the optimal MSE for a given data. It's called the Normal equation.

$$\hat{} = \left(\mathrm{x} \quad \cdot \mathrm{x} \right)^{-1} \cdot \mathrm{x} \quad \cdot \mathrm{y} \tag{23}$$

Let's generate some synthetic data and run through this manually.

Linear regression



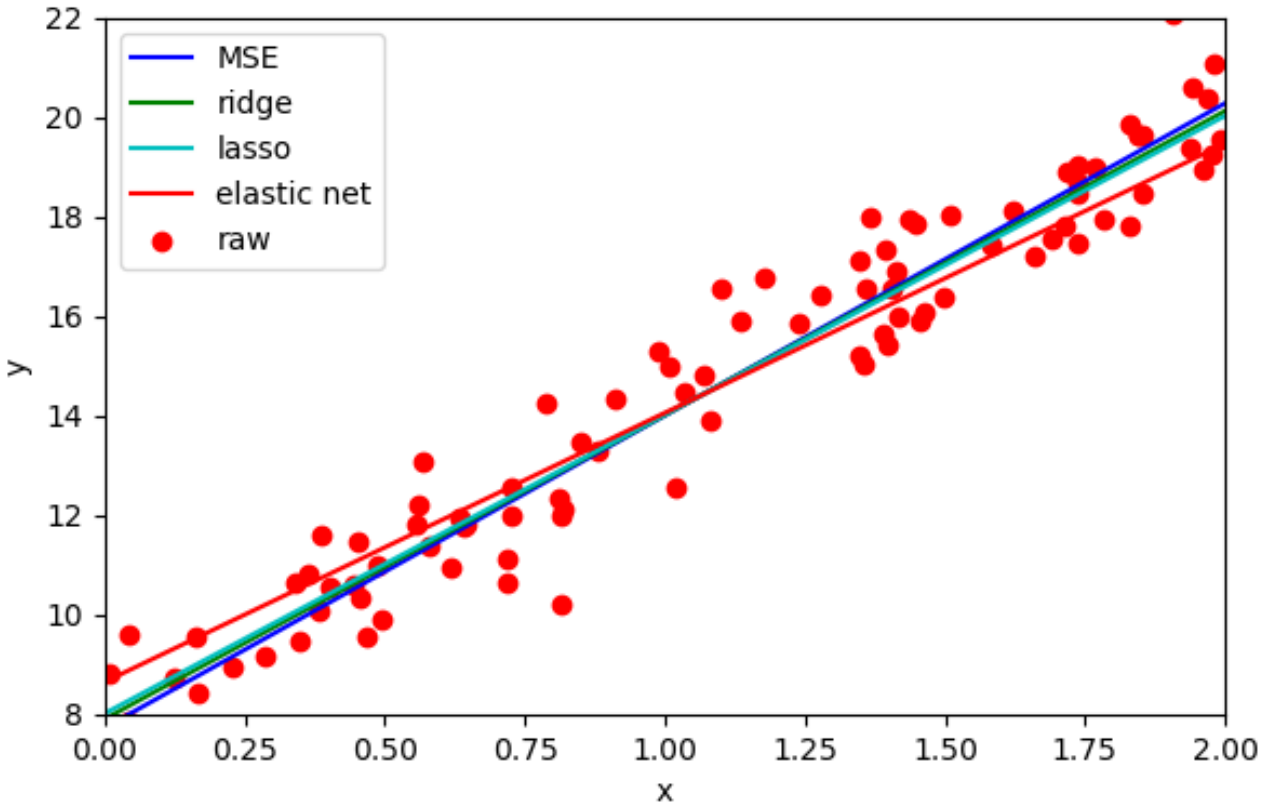
Most of the time you will use a library, but sometimes you need to dig deeper.

???

Most standard models and optimisers are already implemented. During your day-to-day work you will use these. But it aids understanding and occasionally you will need custom implementations.

Other Regression

- Ridge regression (penalises the squared weights - regularisation)
- Lasso regression (penalises absolute magnitudes of the weights)
- Elastic Net regression (combination of Ridge and Lasso)



???

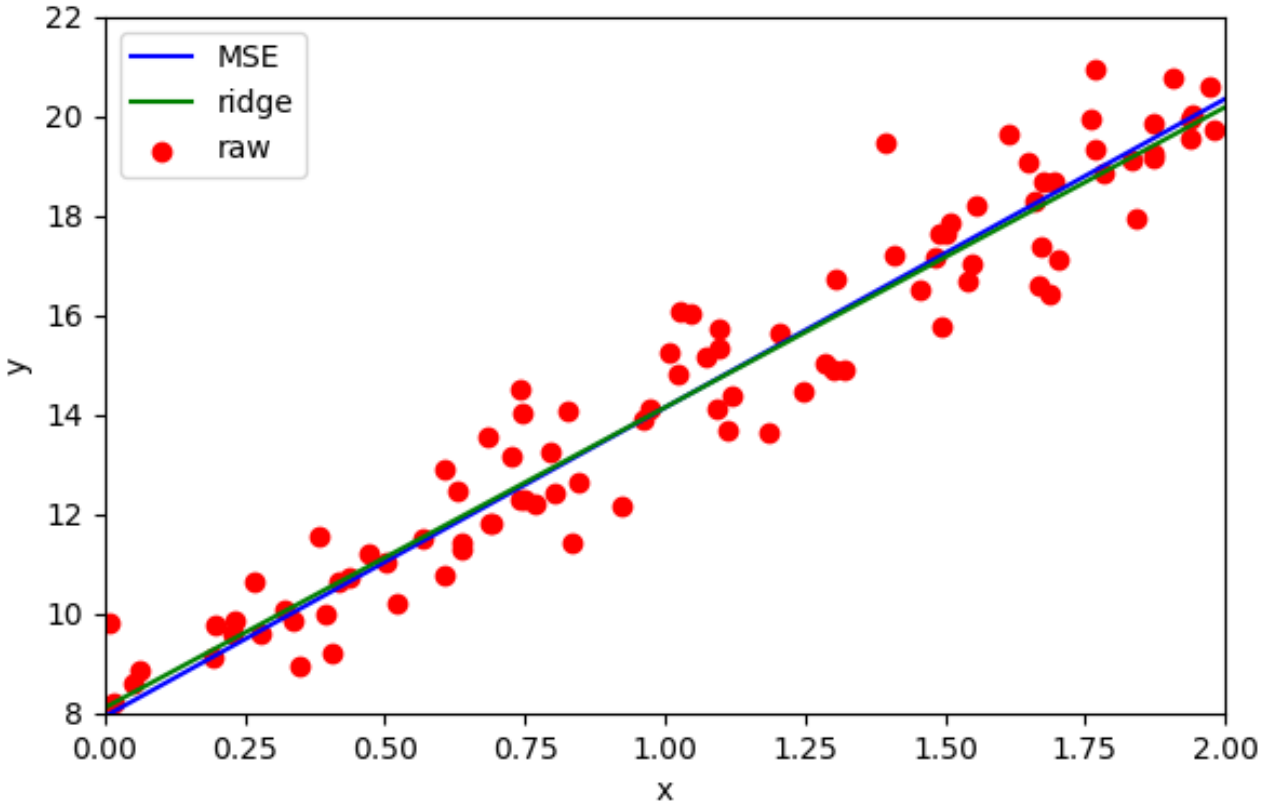
Ridge regression

Other linear regression implementations exist. *Ridge* regression attempts to

optimise the same parameters but penalises the squared weights.

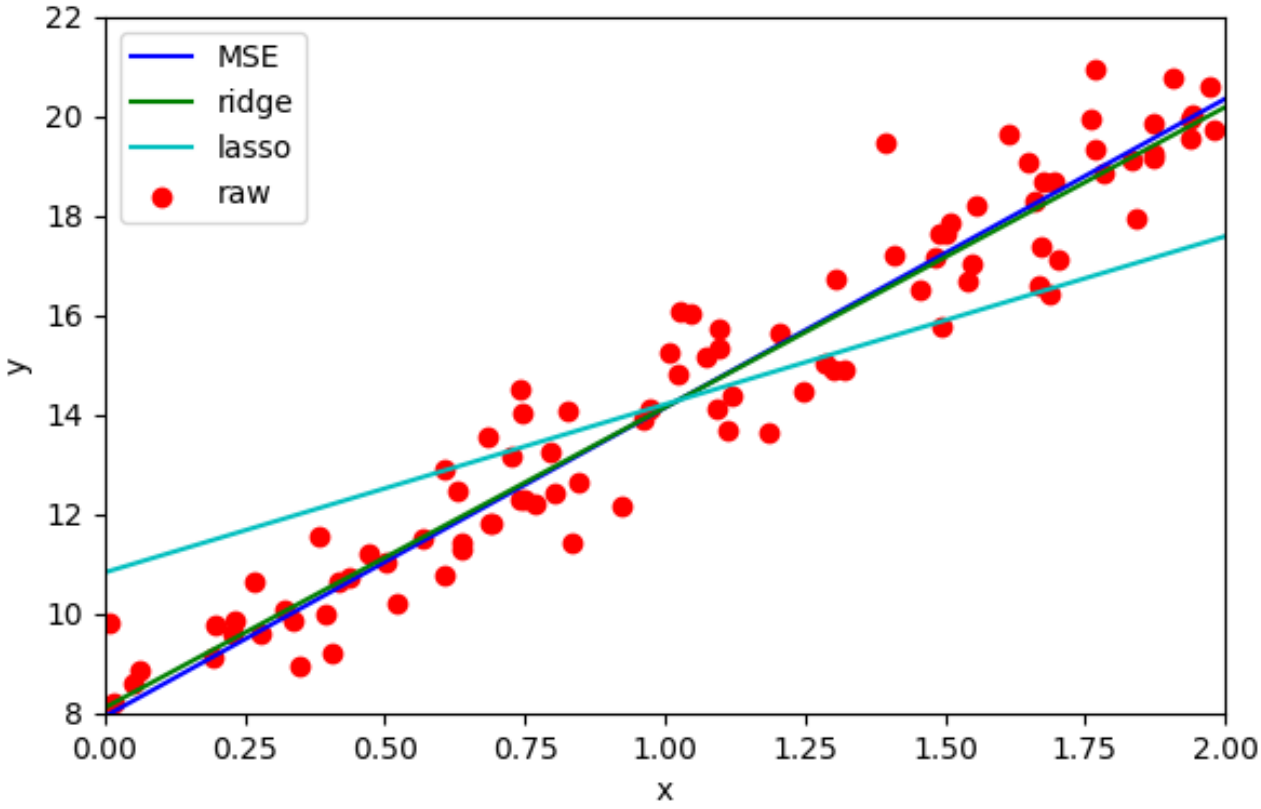
Effectively it attempts to minimise the use of features. This is called *regularisation*.

Ridge regression is most useful when there are large numbers of features.

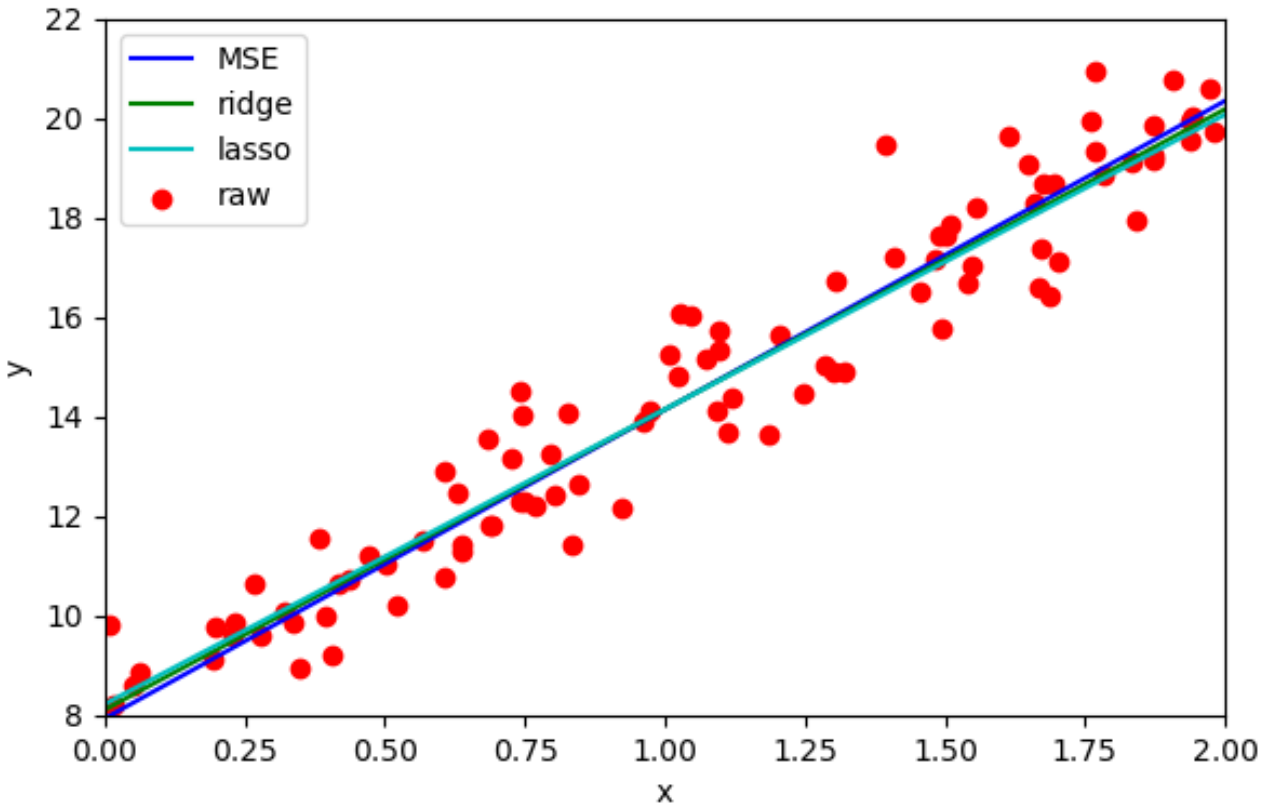


Lasso regression

Another type of regression is called *Lasso*. This is similar to Ridge regression but penalises L1 (absolute) magnitudes of the weights.



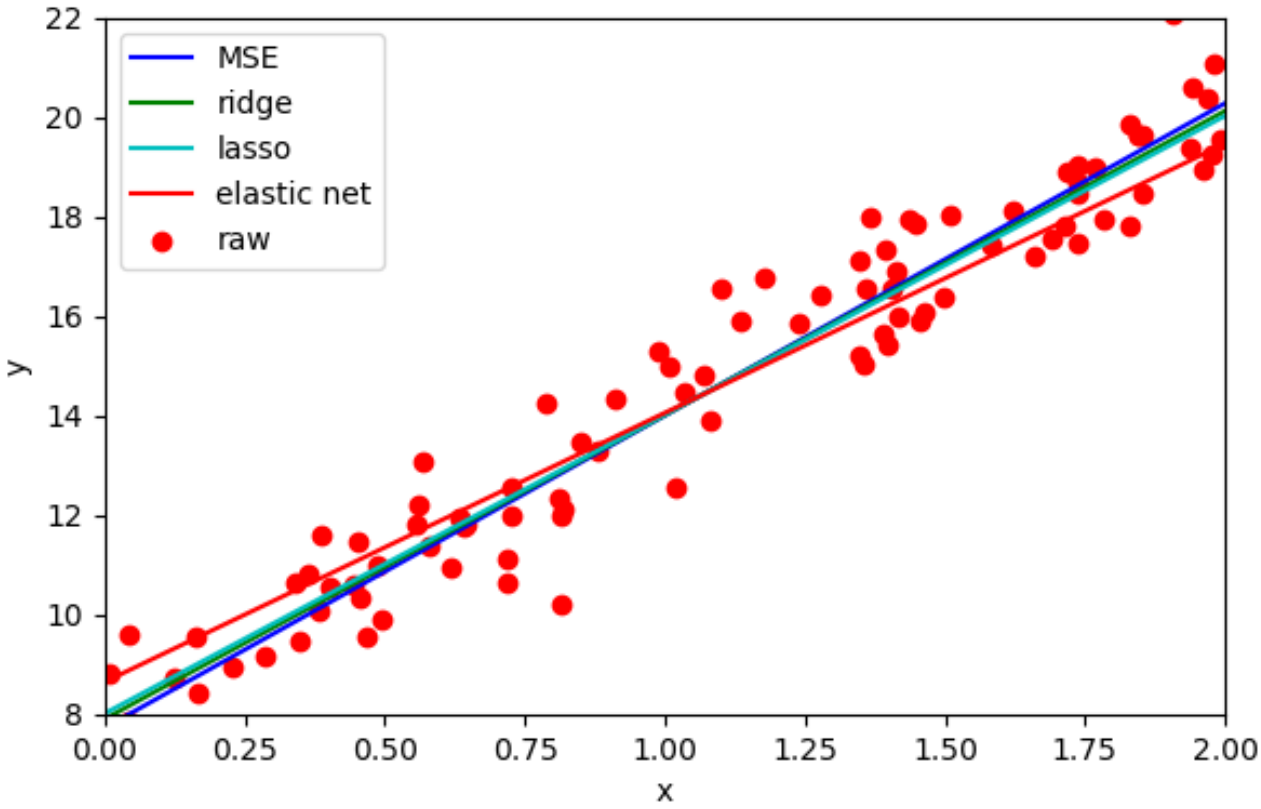
We can see that the penalty is too aggressive. We can alter the `alpha` parameter to reduce the the aggressiveness of the penalisation (same for Ridge).



Generally, Ridge regression is simpler to use but assumes Gaussian noise. Lasso performs better when there are outliers (due to the L1 penalisation, rather than L2).

Elastic Net

There is one final common form of regularisation and that is called *Elastic Net*. In essence, it is a combination of both ridge and lasso regression, where you can control the level of influence from each.



We can see that the effect is quite pronounced.

RANSAC

RANSAC randomly picks a selection of observations from the sample and then performs regression as normal.

The most common set of parameters are chosen as the final result.

This algorithm works really well when there are outliers in the data.

When would you use what?

- Avoid plain linear regression
- Ridge is a good default
- Avoid ridge if the data is grossly non-normal
- But if only a few features are useful, prefer lasso or elastic
- Avoid lasso and use elastic with large numbers of features, or correlating features
- RANSAC whenever you have outliers that you don't want to get rid of.

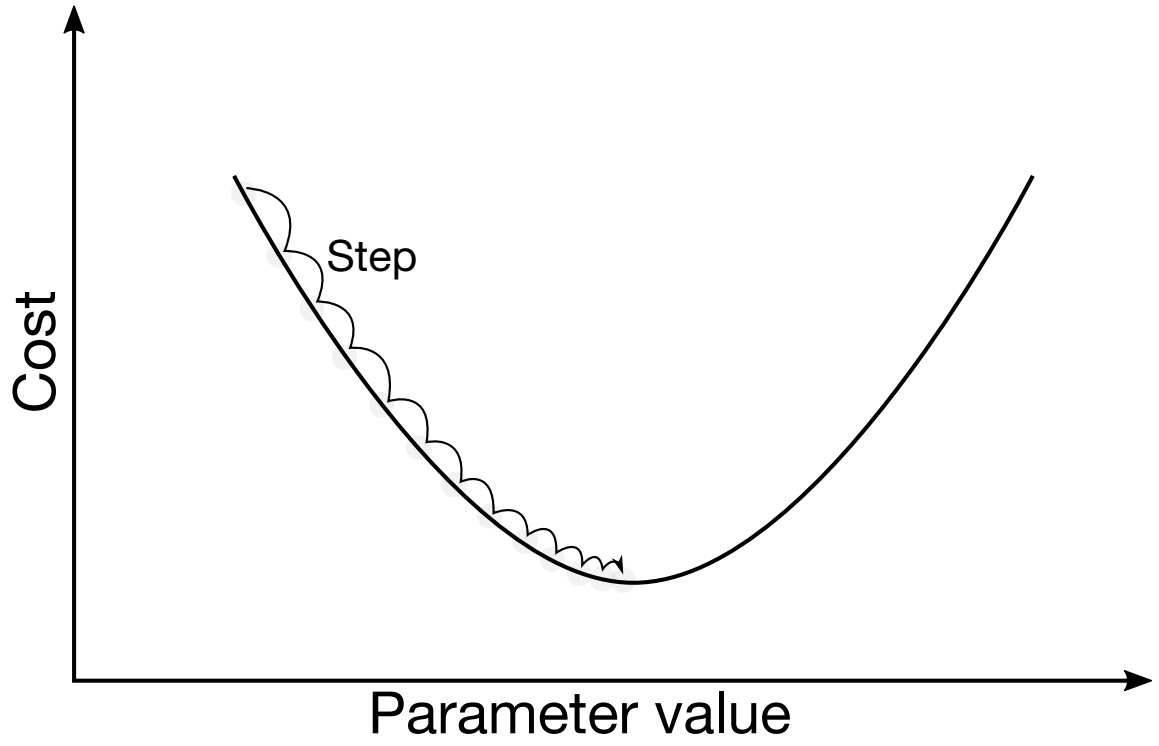
Optimisation

When discussing regression we found that these have closed solutions. I.e. solutions that can be solved directly. For many other algorithms there is no closed solution available.

In these cases we need to use an *optimisation algorithm*. The goals of these algorithms is to iteratively step towards the correct result.

Gradient descent

Given a cost function, the gradient decent algorithm calculates the gradient of the last step and move in the direction of that gradient.



Important: Step size, stopping.

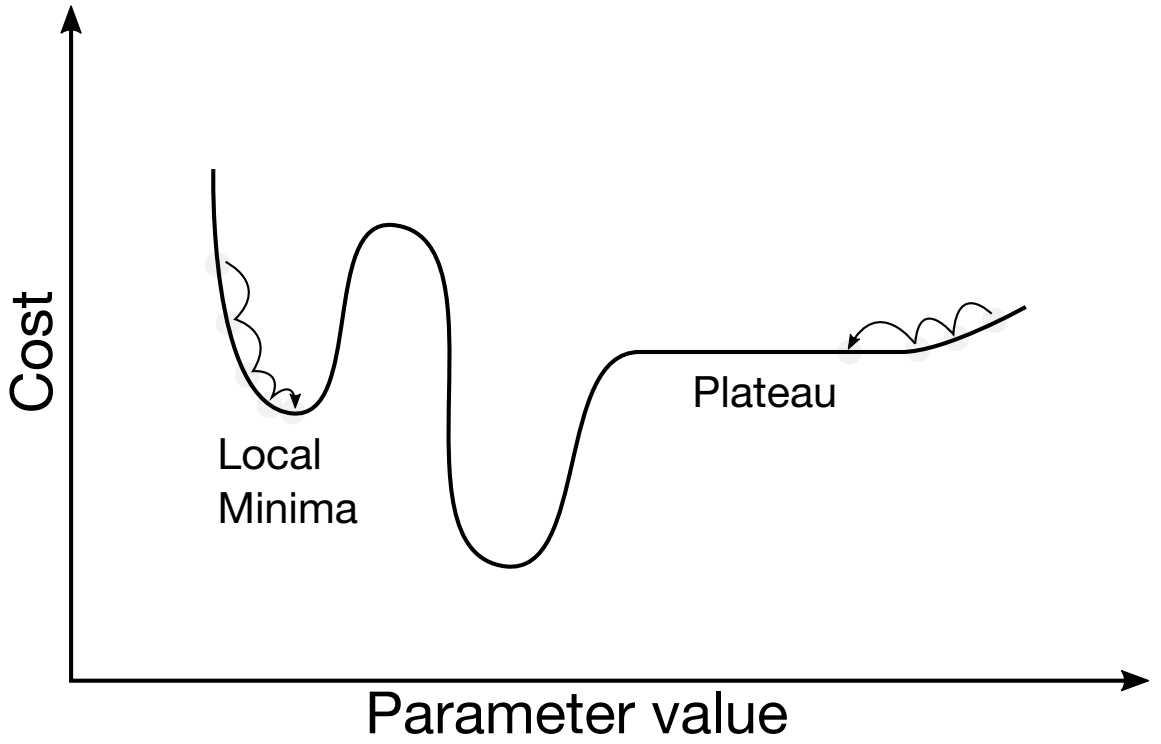
???

Once the gradient approaches zero, then we stop. The step size is an important parameter. Too low, it will take too long. Too high and it will overshoot the minima.

Issues with gradient descent

The gradient of linear regression produces a single, closed solution.

But generally, most data science algorithms have many minima (e.g. a quadratic has two solutions).



Implementation of gradient descent

1. Calculate the partial derivative of the cost function
2. Iteratively update the values of the weights to traverse down the slope

???

We need to calculate the gradient of the cost function. I.e. we need to calculate the difference between a step of a single parameter. This is known as the partial derivative:

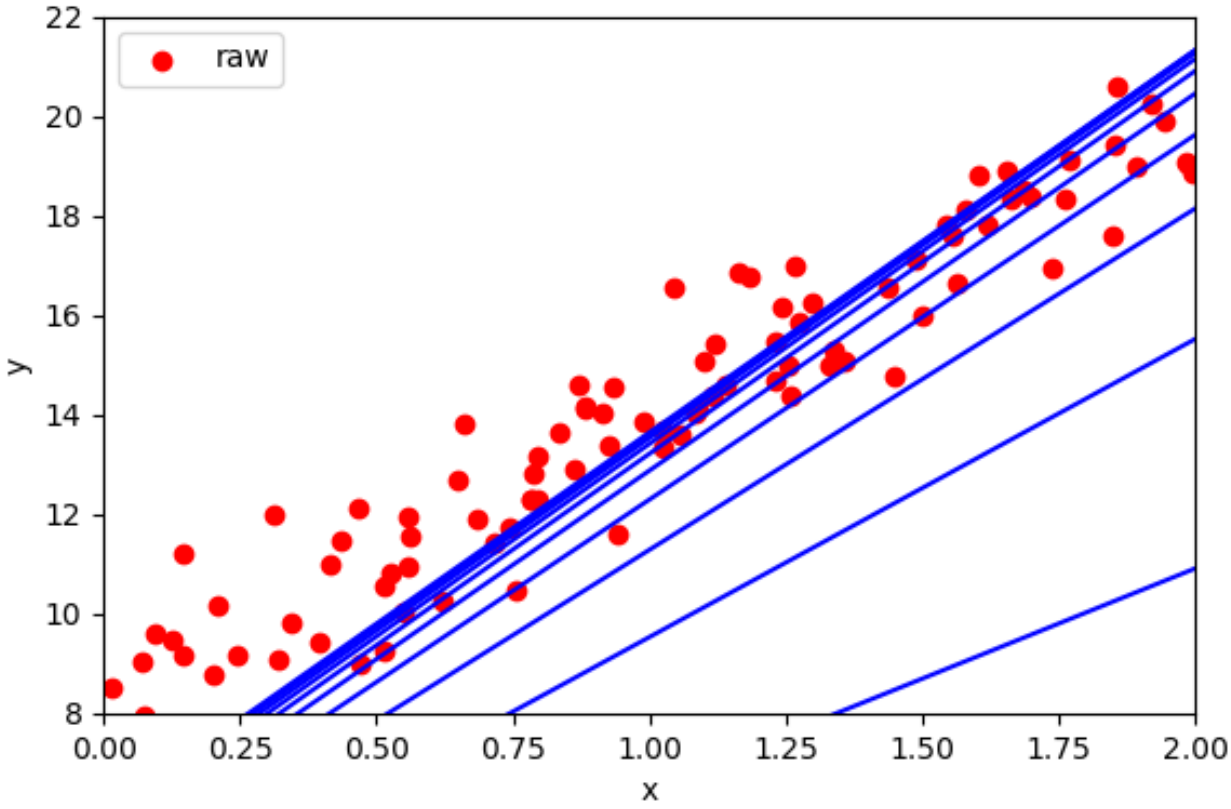
$$\frac{\partial}{\partial \mathrm{w}} \qquad (\mathrm{w}) = \frac{2}{} \sum_{=1} \left(\mathrm{w} \quad \cdot \mathrm{x}^{(\,)} - \mathrm{y}^{(\,)} \right) \mathrm{x}^{(\,)}$$

Instead of calculating the derivatives for the parameters individually, we can compute them in one fell swoop with the gradient vector:

$$\nabla_{\mathrm{w}} \qquad (\mathrm{w}) = \frac{2}{} \mathrm{x} \quad \cdot (\mathrm{x} \cdot \mathrm{w} - \mathrm{y})$$

Note how this uses *every* element in the dataset.

Implementation:



???

```
eta = 0.1 # learning rate
n_iterations = 1000 # number of iterations
m=100 # number of observations

w = np.random.randn(2,1) # random initialization of parameters
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(w) - y)
    w = w - eta * gradients
```

Stochastic Gradient Descent

Standard gradient descent:

- Uses all data for each update (RAM limited)
- Gets stuck on plateaus and in local minima

Stochastic gradient descent is a fancy name for:

- Updating the weights in smaller batches (minibatches), often one observation at a time

Need to randomise data

???

So there are several issues with standard gradient descent:

- Uses all the data at once, probably won't fit in RAM
- Gets stuck in local minima and plateaus

One thing we can do is run through the data in a loop, rather than running it as one big matrix equation. Or subsets of the data to gain some of the optimisations made by matrix math libraries (this is called *minibatch gradient descent*).

But the result would be that we would be fitting to an ordered version of the data. So each time we need to either randomise the data, or simply pick a observation at random.

Despite this, it is actually remarkably useful, because the "jitter" causes it to jump out of local minima and plateaus. But this means it never really settles at a minima. We can control this by reducing the learning rate as we go.

```
Foreach iteration
    Foreach observation
        pick a random index
        calculate gradient for single observation
        reduce the step size slightly
        update w
```

Implementation

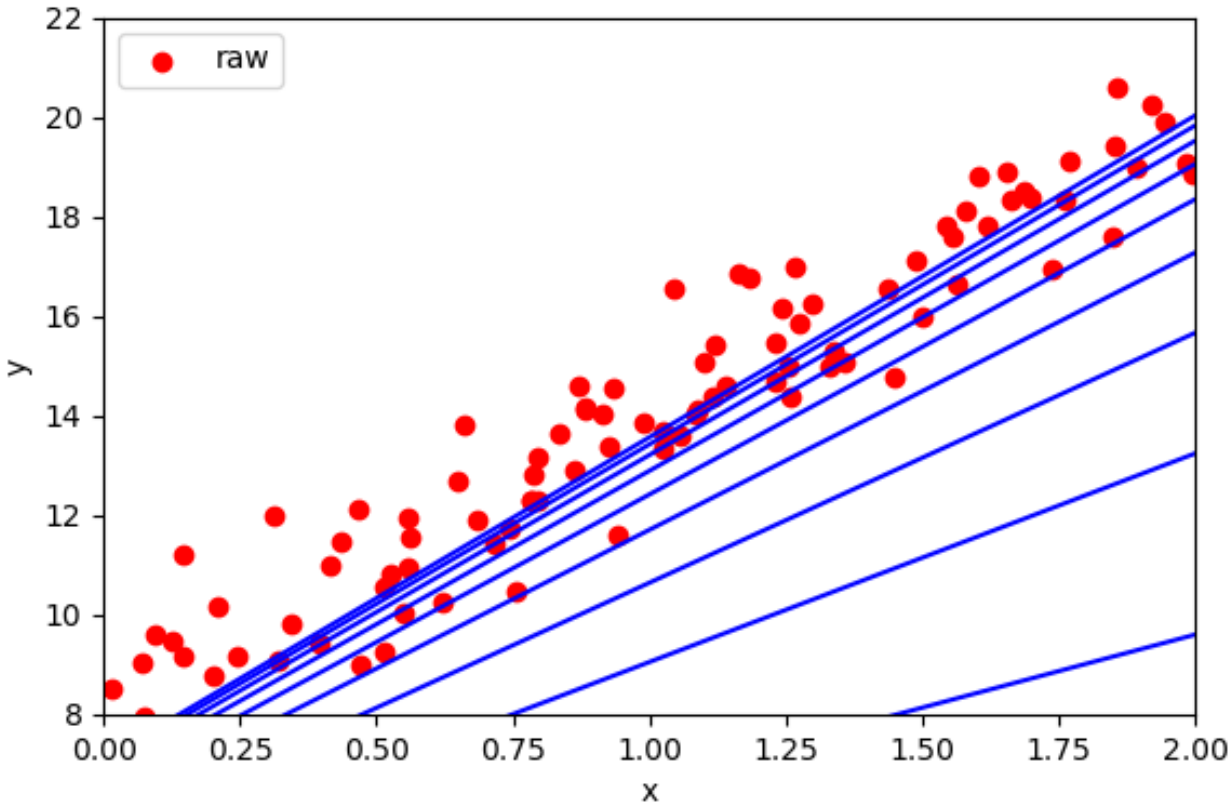
```
n_iterations = 10
m=100 # number of observations
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0/(t+t1)

w = np.random.randn(2,1) # random initialization
for iteration in range(n_iterations):
    for i in range(m):
        random_index = np.random.randint(m)
```

```
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(w) - yi)
        eta = learning_schedule(epoch * m + i)
        w = w - eta * gradients
```

This type of learning schedule is known as *simulated annealing* (from metalurgy, where annealing is the process of metal slowly cooling).



Classification via a model

- Decision trees created a one-dimensional decision boundary
- We could easily imagine using a linear model to define a decision boundary

???

Previously we used fixed decision boundaries to segment the data based upon how informative the segmentation would be.

The decision boundary represents a one-dimensional rule that separates the data. We could easily increase the number or complexity of the parameters used to define

the boundary.

Leaving the decision boundary parameters unspecified, we can allow the parameters to be defined by some theoretical model or by some regression-like process.

Modelling

In the extreme, we can use any arbitrary function to define a decision boundary.

But some interesting questions arise:

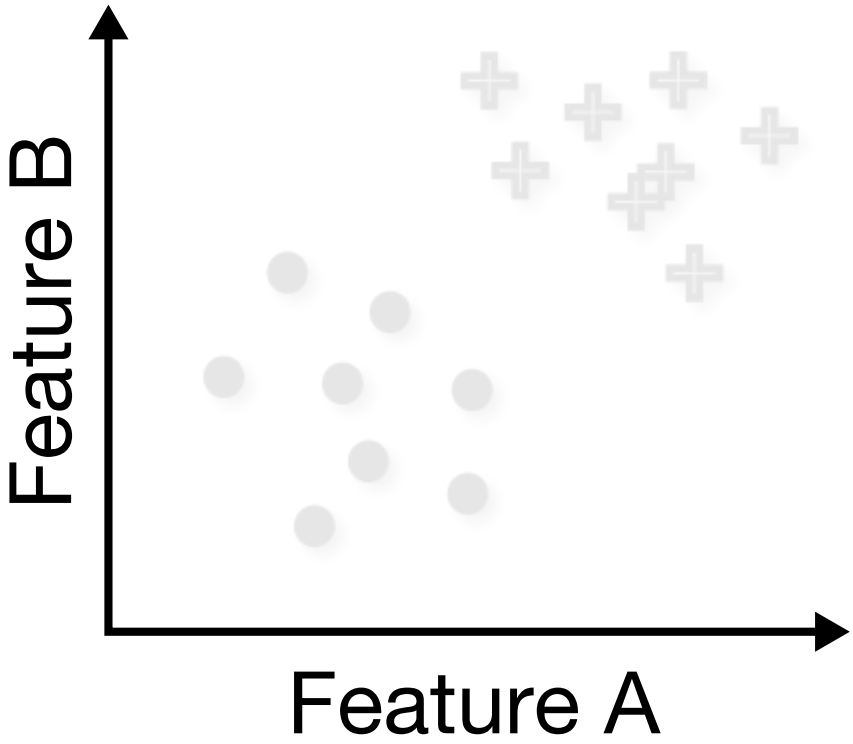
- What is the optimal location for the boundary?
- What do we mean by optimal?
- How can we justify a boundary, in business terms?

???

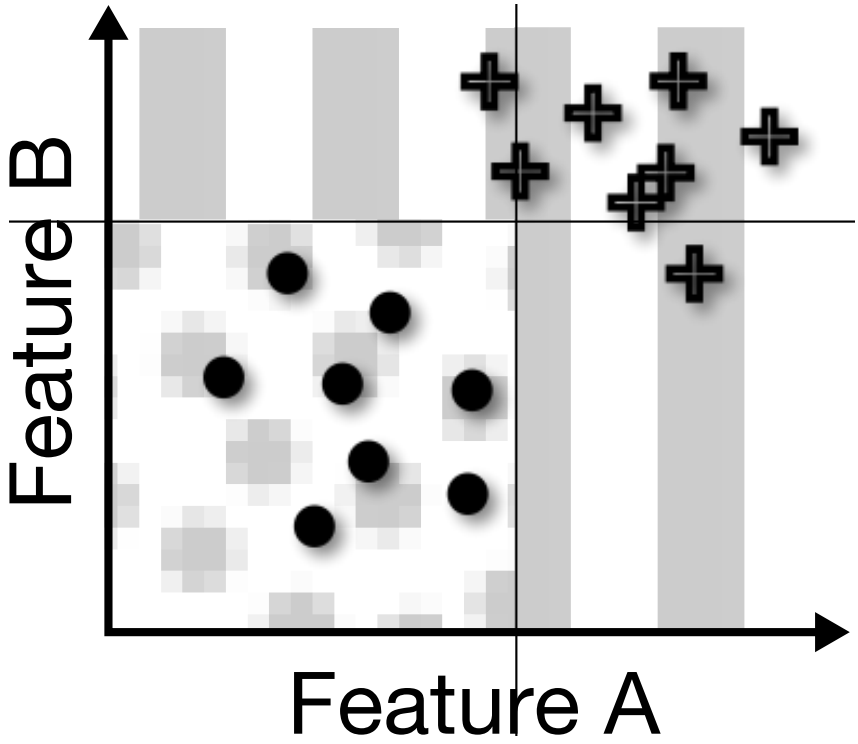
Many types of problems can be defined in this way and there are a range of linear and nonlinear methods that are able to define the decision boundary.

Classification via mathematical functions

Consider the image below. The data has two features.

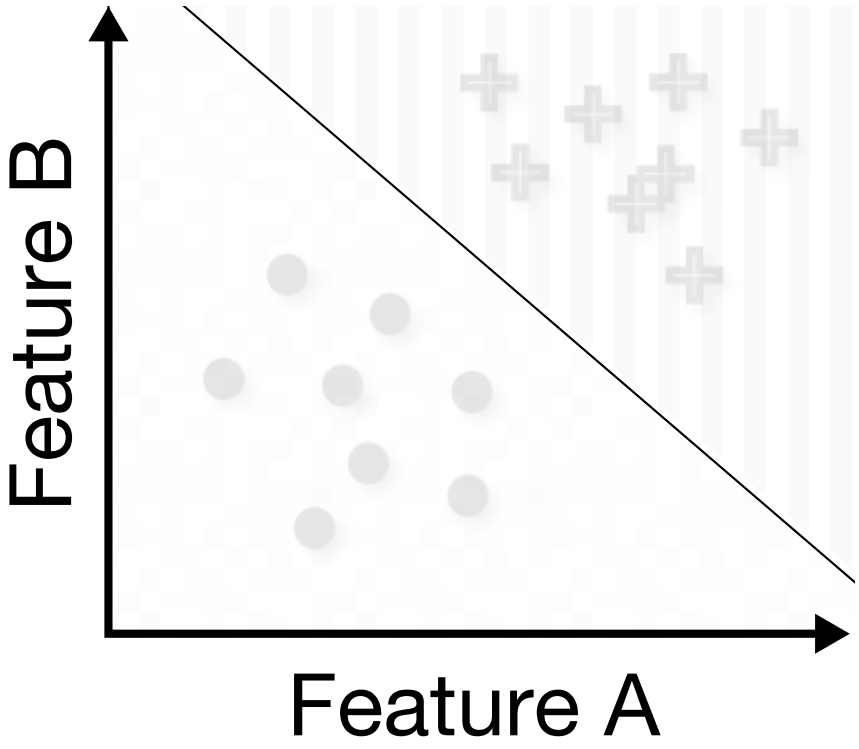


If we used one-dimensional rules, like in a decision tree, we would end up with a decision boundary looking like this:



Linear function

Instead, we could create a decision boundary that is a mathematical function. Like this:



This type of decision boundary is called a linear classifier because the hyperplane is a weighted combination of linear features. E.g. $= \quad +$

More generally, a linear function is defined as:

$$(\mathbf{x}) = \quad_0 + \quad_1 \ _1 + \quad_2 \ _2 + \cdots \tag{24}$$

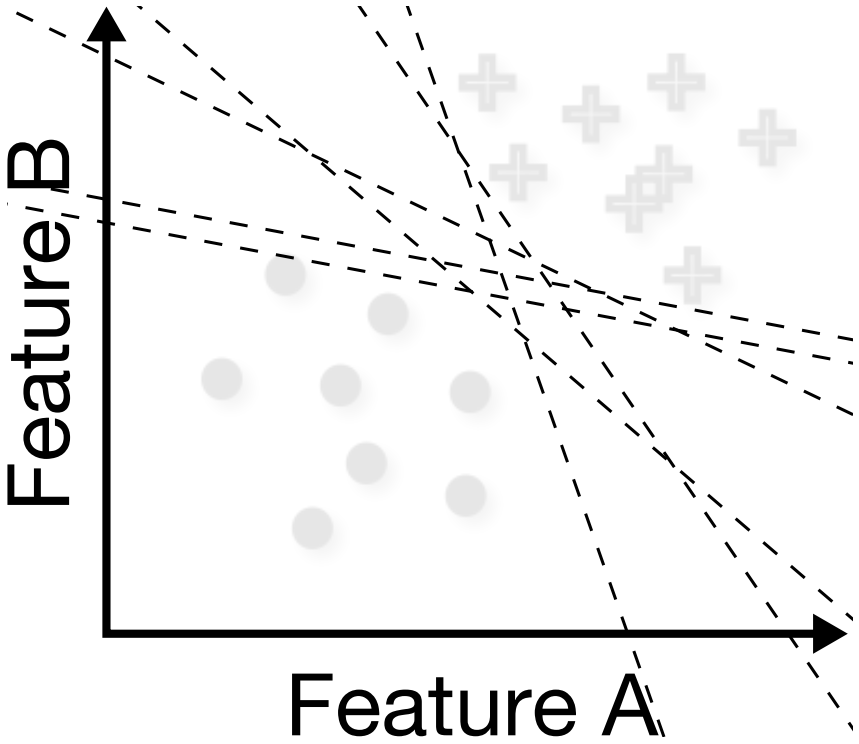
$$= \mathbf{w} \quad \cdot \mathbf{x} \tag{25}$$

This is the same as linear regression!

The goal, however is different. The goal is to alter the parameters $\mathbf{w}$ so that it correctly classifies all classes in the data $\mathbf{x}$.

But this begs the question, what is the best location for the decision boundary?

We could place the decision boundary in any number of locations:



Which is best? This leads us to one of the most important questions in data science, one that is often overlooked, but is the key to advanced applications…

Loss Function

- What is the goal or *objective* when choosing the parameters?

The general procedure is to define an objective, then optimise the parameters to fulfil that objective (as best as it can).

Note: The premise is the same as a cost function, but now we're talking about categorical classes.

???

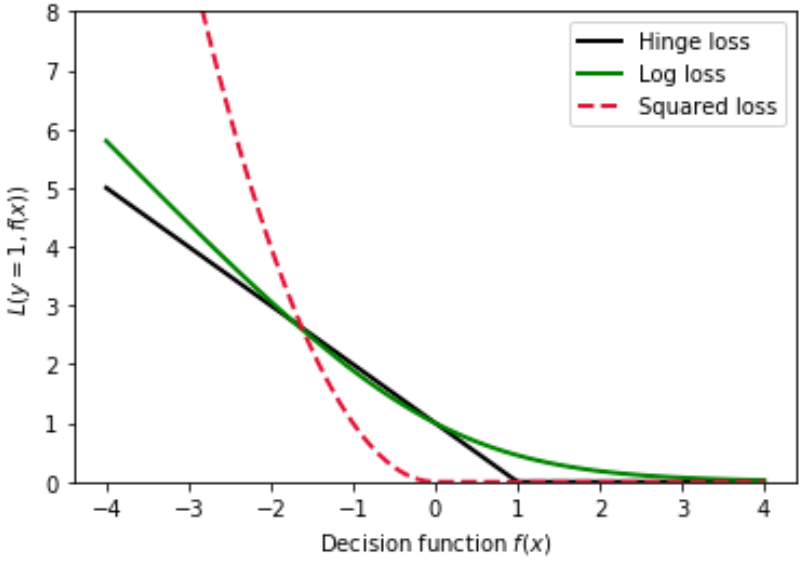
For example, we could choose the objective function to optimally separate classes or we could be interested in being as far away as one class as possible (to avoid false positives, for example).

The choice of loss function is often quite hard to define and a range of "standard" objective functions have emerged: - Linear - Support Vector Machines - Logistic

These three are the workhorses of data science and differ only in their objective.

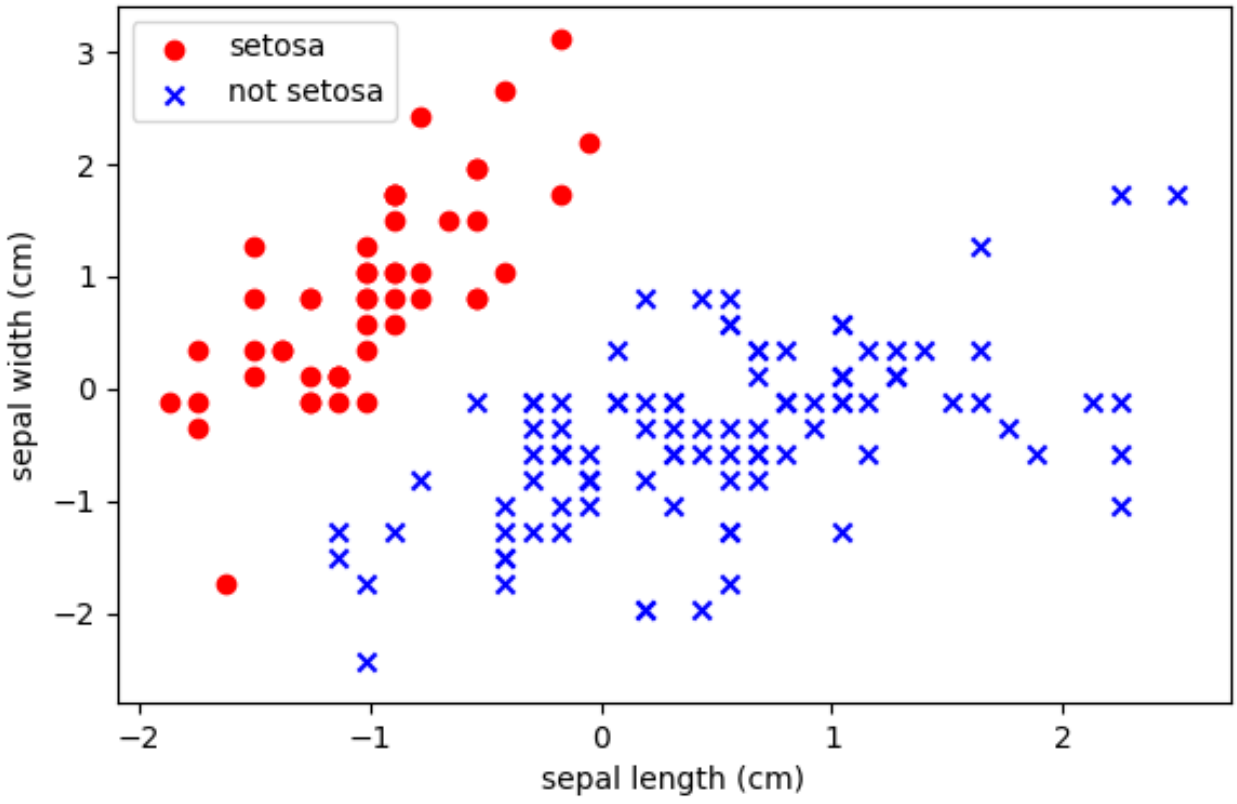
---

class: center, middle



Example: Petal/Sepal

The *Iris* dataset is one of a number of "standard" data science datasets.



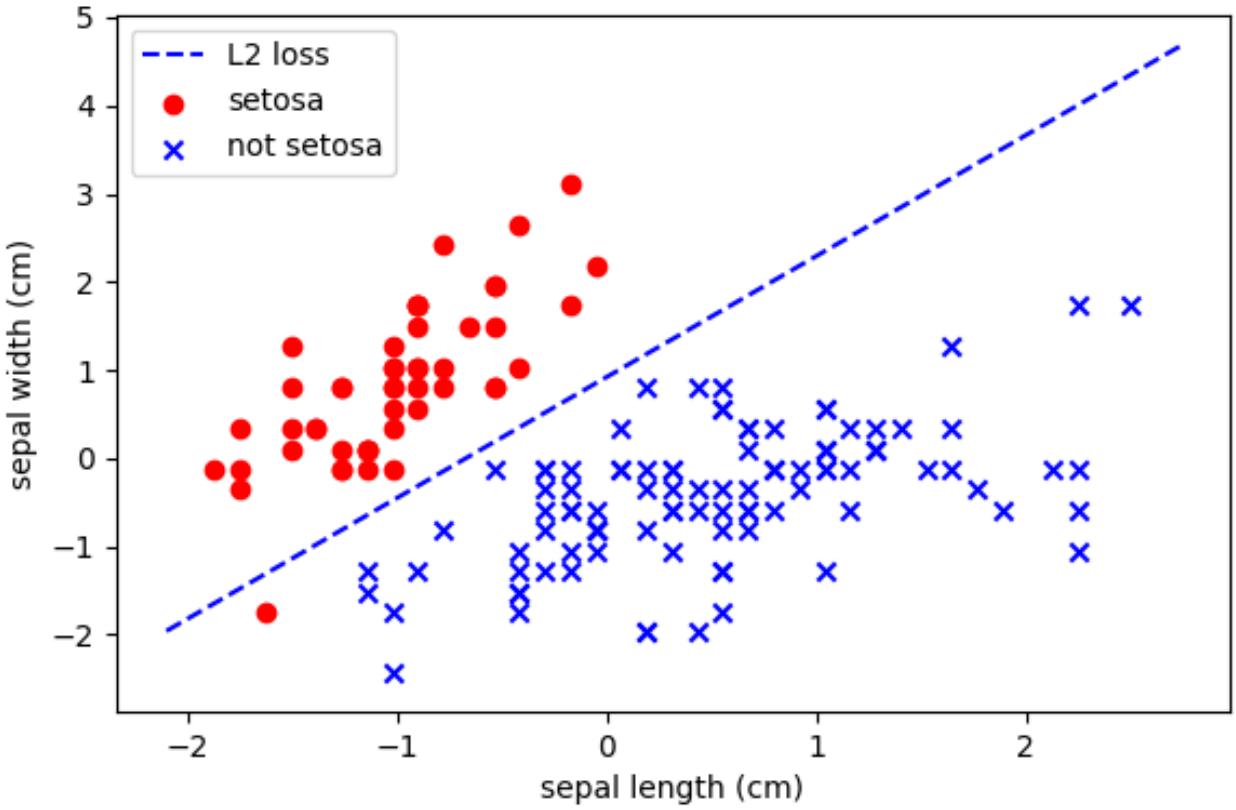
???

It is a dataset that measures different parts of an iris flower and is labeled by the species.

Two of the features are plotted below for two of the species.

Linear Classification

This measures the squared error between the distance from the boundary to a misclassification.

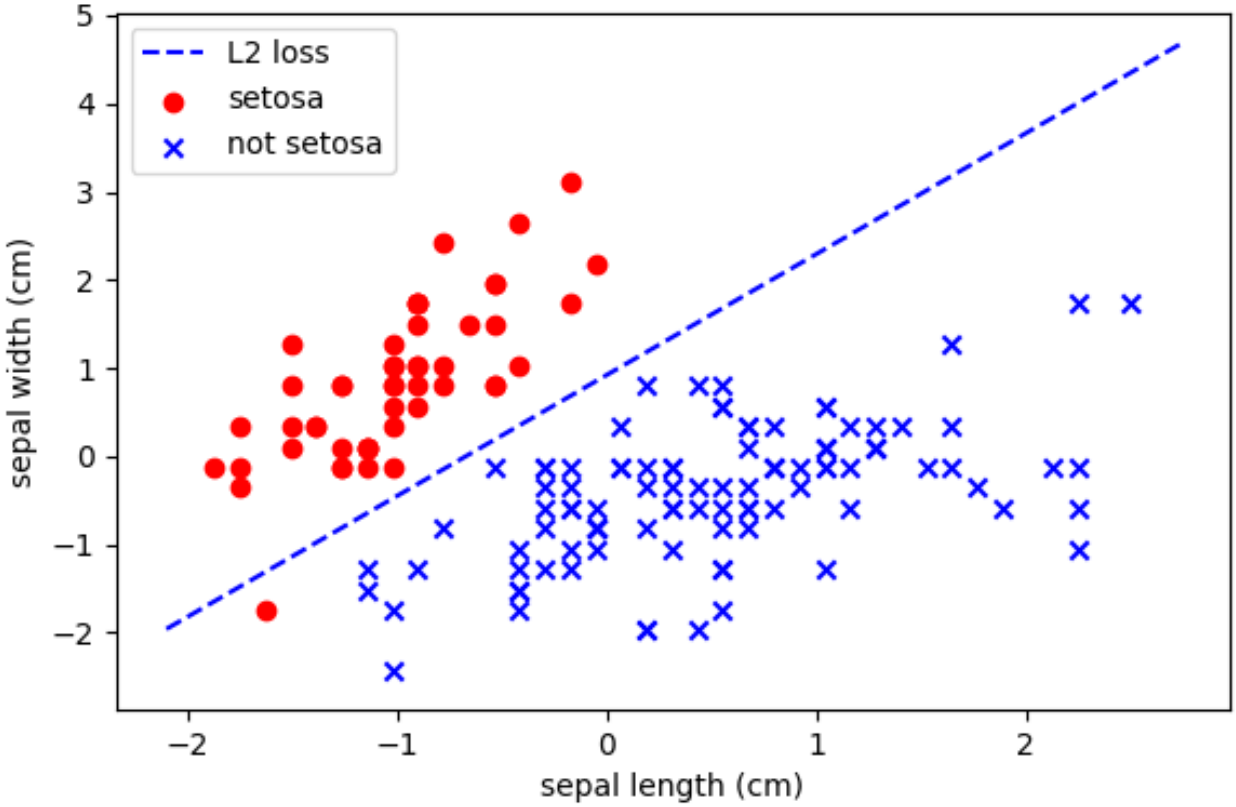


???

This attempts to fit a line that best reduces the squared error of two or more classes. For these examples we will use stochastic gradient descent.

We can see that there are some issues with this classification.

- Skewed due to larger numbers of one class
- Sensitive to outliers (assumes gaussian distribution)
- Does the chosen objective match to the business requirements? E.g. are those misclassifications costly?



Logistic regression

- Logistic regression is often used because it also estimates class probabilities.

Logistic regression fits a binomial distribution between two classes. A probability of 0.5 is used as the decision boundary.

???

Logistic regression (a.k.a. *logit regression*) is commonly used to estimate the probability that a new observation belongs to a class of interest. For example, if the goal is to reduce the amount of money lost to fraud, then we not only need to know the occurance of fraud, but also the probability of it occurring.

Logistic regression is a linear model that instead of outputting expected value directly, it outputs the *logistic* of this result:

$$\hat{} = \quad (\mathrm{x}) = \quad (\mathrm{w} \quad \cdot \mathrm{x})$$

The *logistic*, (a.k.a. *logit*) is a *sigmoid function* that outputs a number between 0 and 1.

$$( ) = \frac{1}{1 + \exp(-)}$$

Once the model has estimated the probability of the class $\hat{}$ it can threshold at a value of 0.5 to perform classification.

Training

Assuming binary classification, we need a cost function that produces a small value when the regression correctly predicts a class and a large value when it gets it wrong. Commonly, this is used:

$$( ) = \begin{cases} -\log (\hat{}), & \text{if y} = 1 \\ \log (1 - \hat{}), & \text{if y} = 0 \end{cases}$$

The cost function is then the average log error over the entire dataset. This is a common loss function and so has a name, the *log loss*:

$$( ) = \frac{1}{} \sum_{=1} \left({}^{( )} \log \left(\hat{}^{( )} \right) + (1 - {}^{( )}) \log (1 - \hat{}) \right)$$

Unfortunately there is no closed solution so we must use an optimisation algorithm to find the solution. Thankfully though it is convex so the solution guaranteed to find the global minimum. It also looks very similar to the MSE's partial derivative.

$$\frac{\partial}{\partial \mathrm{w}} \ (\mathrm{w}) = \frac{2}{} \sum_{=1} \left(\ \left(\mathrm{w} \ \cdot \mathrm{x}^{( )} - \mathrm{y}^{( )} \right) \right) \mathrm{x}^{( )}$$

This is the result of logistic regression (a.k.a. fitting a binomial distribution).

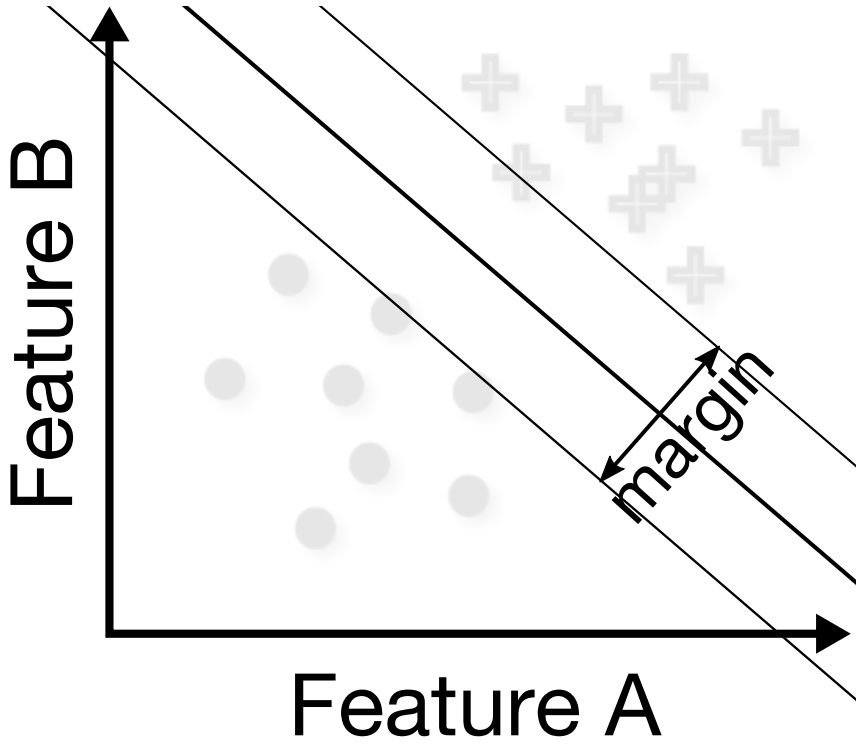
Support vector machines

SVMs have an annoyingly vague name, but their performance is impressive.

They have one goal:

- Maximise the boundary between classes

It tries to separate the classes by the largest possible margin:



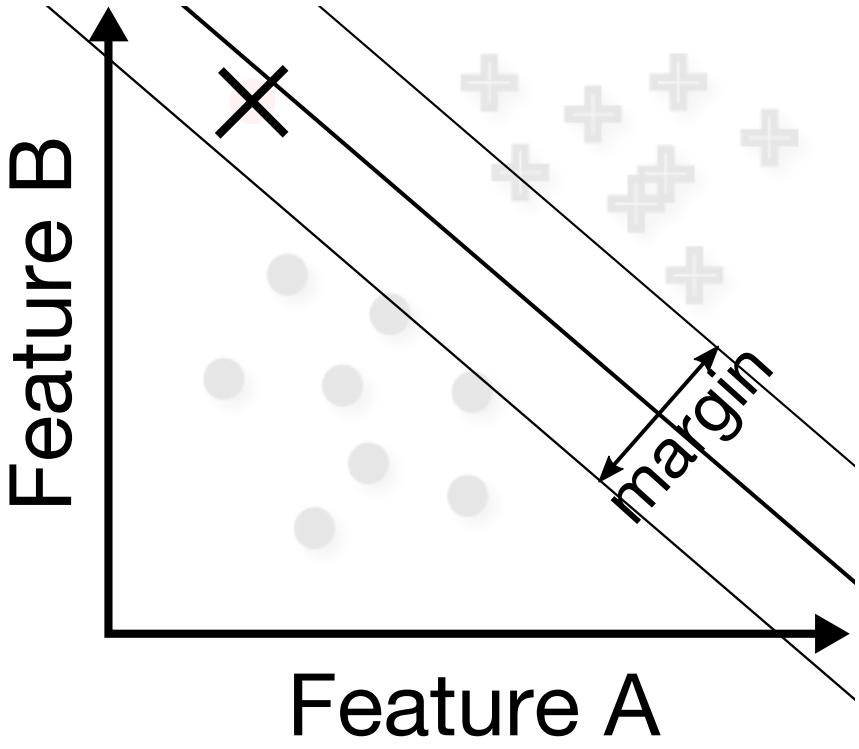
???

First, it classifies using a hard margin. I.e. anything above the threshold has no penalty and anything below it has increasing penalty.

The loss function that achieves this is a loss of 0 at correct classifications and an increasing loss at at incorrect classifications. This is very similar to a magnitude loss (L1) except there is a classification discontinuity at 0.

The issue with this is that because it's discontinuous, it isn't differentiable. Which means we can't use any of the gradient descent methods. Instead, it resorts to a pretty complex form of linear algebra called *Quadratic Programming*.

It allows some points into the margin by using a Hinge loss (i.e. it continues to penalise elements within the margin)

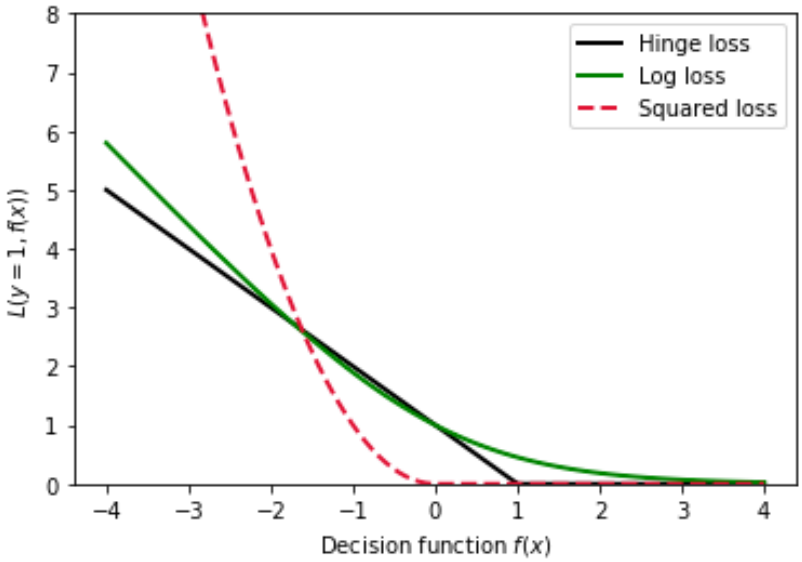


We can see that it has positioned the threshold to produce correct classifications, but has allowed observations to enter the margin.

Note, however, that because of the outlier in the bottom left, it is still skewing the decision boundary to be slightly higher than (in my opinion) it should. And this is why many people prefer logistic regression…

And this is ultimately because of the cost function…

Remember: Loss Function



Remember that the only difference between these three algorithms is the function used to calculate the loss.

Nonlinear functions

- Sometimes data cannot be separated by a simple threshold or linear boundary.

We can also use nonlinear functions as a decision boundary.

???

To represent more complex data, we can introduce nonlinearities. Before we do, bear in mind:

- More complex interactions between features yield solutions that overfit data; to compensate we will need more data.
- More complex solutions take a greater amount of computational power
- Anti-KISS

The simplest way of adding a nonlinearities is to add various permutations of the original features. For example, some feature squared.

Polynomial Features for Nonlinear Logistic Regression

1. Create new features, polynomials of the original data
2. Perform logistic regression using all of the new features

Still a linear classifier, we're just using more complex features.

???

This is a logistic classifier with a polynomial expansion to a degree of 3. E.g. before we just had two features, $_1$ and $_2$. Now we have nine:

$$_1 + \ _2 + \ _1 \ _2 + \ ^2_1 + \ ^2_2 + \ ^2_1 \ _2 + \ _1 \ ^2_2 + \ ^3 + \ ^3$$

The beauty is that this is still a linear problem and therefore fast and guaranteed to optimise.

Nonlinear SVMs

A similar polynomial trick can be performed with SVMs:

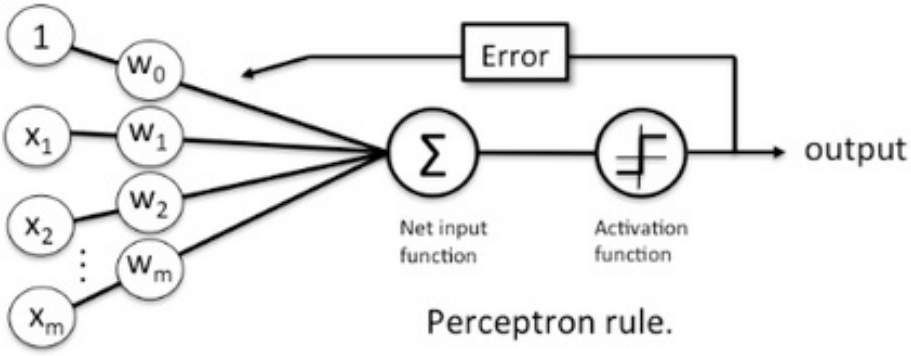
Kernel trick

Briefly, one final strategy to note is called the *kernel trick*. Essentially this convolves a kernel (of any shape) over the data. This is a data transformation. We are mapping the data from one domain into another.

Schematic of Linear Classification

It's sometimes easier to visualise an algorithm as a schematic or a graph.

For each of the linear classifiers, we ere altering the activation function.



Key Point: This is actually an image of a Perceptron, the base unit in Neural Networks.

All of Deep Learning is based upon lots of linear classifiers!!!

???

Neural networks

The ultimate progression of nonlinear classifiers has resulted in stacked neural networks.

Neurons are fairly simple nonlinear classifiers. They are based upon a range of *activation functions* which are essentially cost functions for a single feature combination.

The functions are very familiar, with logistic-like, hinge-like and one-zero-like activation equivalents.

But the real advantage is that the neurons can be stacked in any number of formations to provide incredibly nonlinear functions.

Functions so nonlinear that they approach the capacity of the human brain in a number of specific domains (e.g. image classification).

A really hot topic, but bewilderingly huge. We delve into neural networks in the third workshop (advanced).

---

class: middle, center

Workshop 4

---

name: generalisation-and-overfitting

Generalisation and overfitting

> *"enough rope to hang yourself with"*

- We can create classifiers that have a decision boundary of *any* shape.

Very easy to *overfit* the data.

This section is all about what *overfitting* is and why it is bad.

???

Speaking generally, we can create classifiers that correspond to *any* shape. We have so much flexibility that we could end up *overfitting* the data.

This is where chance data, data that is noise, is considered a valid part of the model.

In reality, we are interested in models that are able to generalise; models that are able to classify observations that have not been observed thus far.

Generalisation

Given some data, think of a simple way to accurately classify that data?

- Build a lookup table

Lookup tables cannot generalise to new examples.

Tip: Beware of 100%

???

Imagine we had a historical dataset of customers that did or did not click on a range of image based adverts. An advertising client has asked whether we can predict whether a customer would click on the advert.

- We go away and report back that we have 100% accuracy.
- We tell them that we have created a lookup table for every user for every advert.
- We have successfully classified all instances of whether a user will click on an advert.

Clearly this is a ridiculous example; we couldn't claim that it predicted anything.

This is why we need to *generalise* our models, to cope with new data.

Overfitting

- Complex models produce lookup tables

Overfitting is the word used to describe when models are too complex.

E.g. Decision trees could create a decision rule for every observation.

Key point: Data Science is all about compromise

???

Despite the the previous example being so contrived, every ML algorithm has the capability of overfitting.

For example, decision trees can very easily create decision rules that segment every single piece of data.

And this occurs far more often than you might think.

The fundamental problem is that most real-life data is complex. We need complex models to cope with the complexity.

So there is a trade-off between how well we can model the problem and how much we overfit.

Why is overfitting bad?

- Noise

You don't want your decision boundary to hug observations, because they are probably noisy.

???

All data is inherently noisy. Even the simplest of experiments contain noise.

Data with more features have compounding noise sources.

With more complex data it is very common to have noise that is not distributed according to a random process. For example, some data is missing, or invalid.

At best, we end up training our model on the noise, rather than the signal of interest.

At the worse, we could end up using spurious, factually incorrect data, to create our

model.

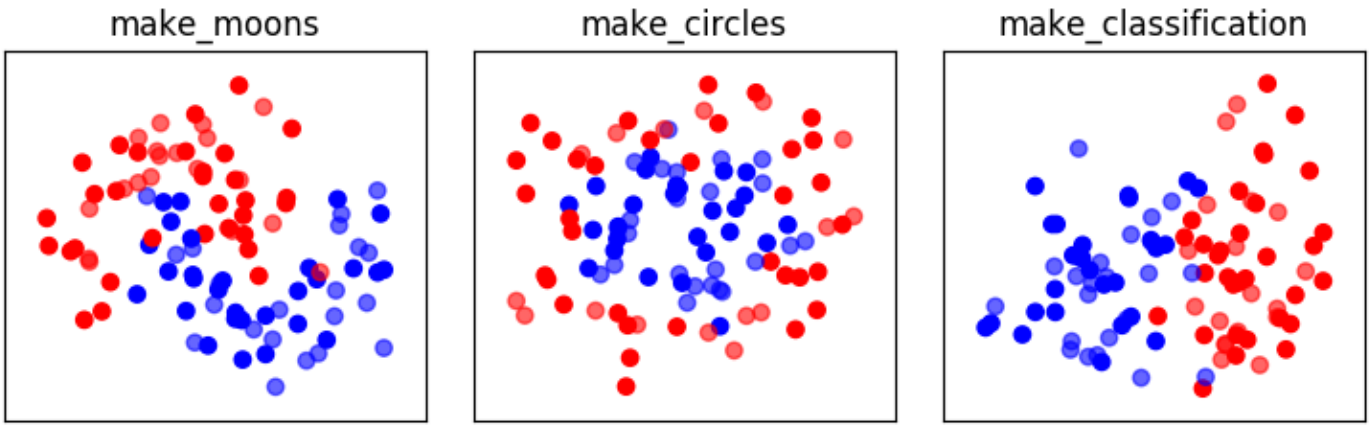
Any new observations that arrive within these bounds will be classified incorrectly.

Underfitting

Underfitting is a lack of model complexity.

This usually occurs in domains where there are inherently large numbers of features, e.g. images.

2D example:



Spotting overfitting can be hard

- How do we visualise under/overfitting?
- It can be difficult.

Let's look at an example using the `make_circles` data...

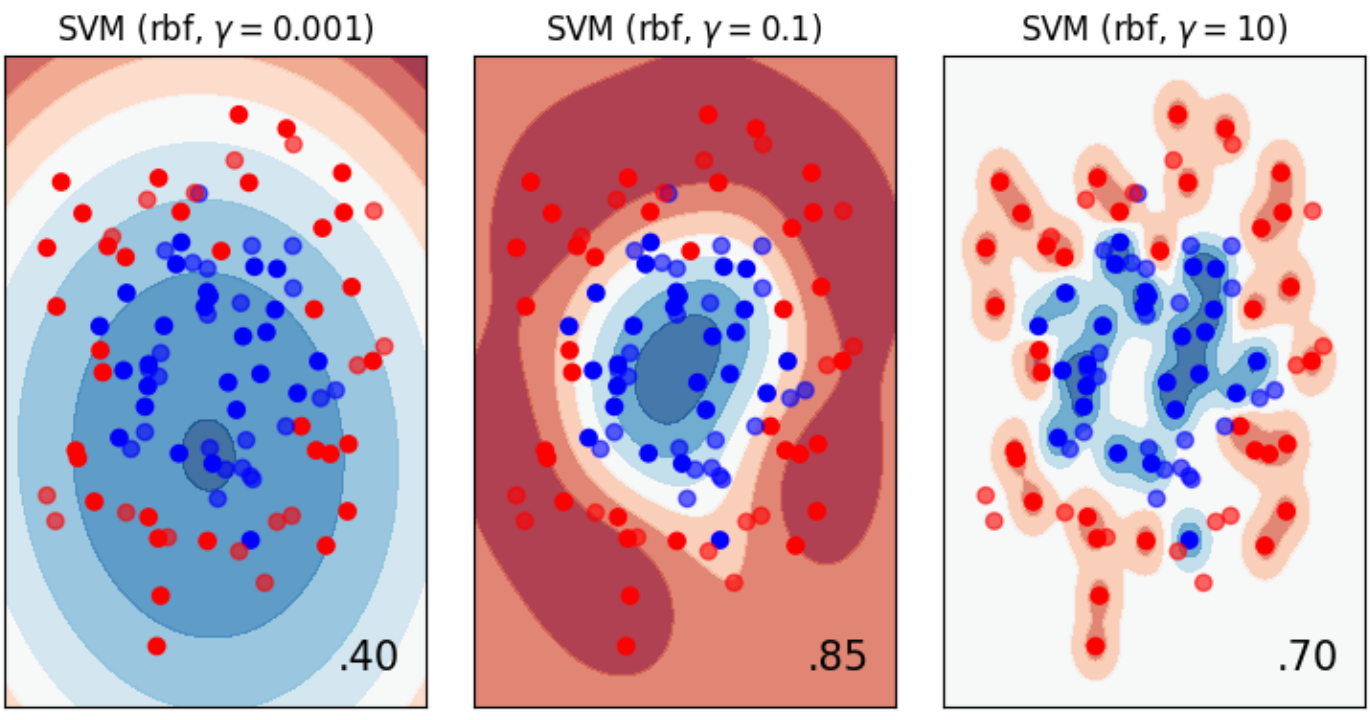
???

The problem with overfitting is that it is very hard to tell when you have overfitted. Especially when it's hard to plot the features/decision surface (e.g. many features).

For example, using the `make_circles` data, let's try and plot the accuracy of of a model vs a parameter...

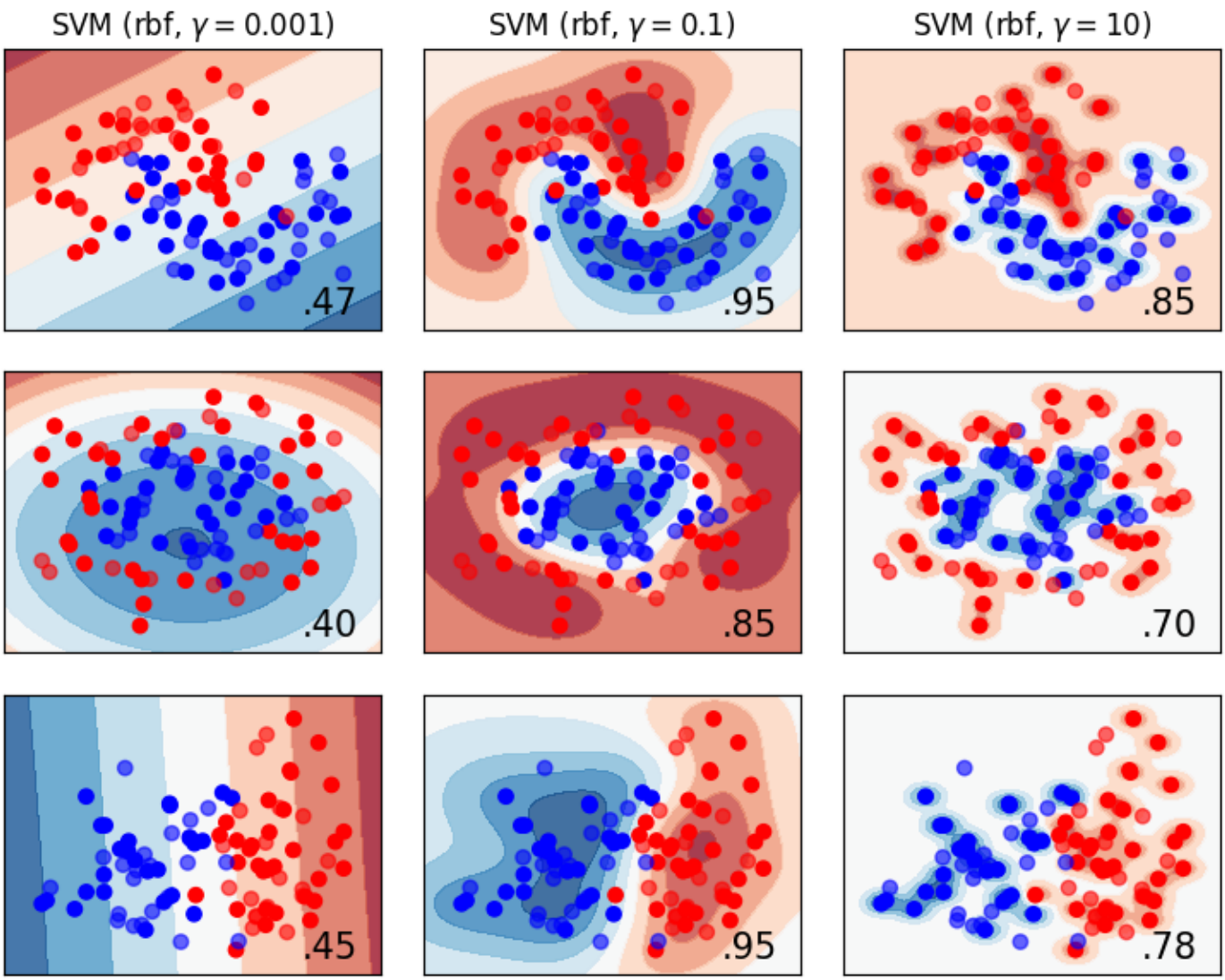
We can see that as we alter the value of the accuracy increases. We might assume that we have a better model.

But we would be wrong. Let's look at a plot of the decision boundary (this isn't usually possible due to the number of features!)...



If we look at the right most plot, we can see that we are not generalising any more. We're fitting too closely to the data. Essentially, we've created a very complex look-up table!

Let's look at this over some more datasets...



So how can we make sure that we don't produce misleading scores?

This is probably the most important part of data science, so pay attention!

Holdout

We have been using:

- Training data

Not representative of production.

We want to pretend like we are seeing new data:

- Hold back some data

???

When we train the model, we do so on some data. This is called *training data*.

Up to now, we have been using the same training data to measure our accuracy.

If we create a lookup table, our accuracy will be 100%. But this doesn't generalise to new examples.

So instead we want to *pretend* like we have new examples and use that to test our model.

In other words, we hold back some data.

How does holdout work?

Separate the data into a *training set* and a *test set*.

Test set size approx. 10-40%.

Minimum size depends on number of features and complexity of model.

???

When we obtain a dataset, we would separate the data into a *training set* and a *test set*.

The size of the test set is usually somewhere between 10-40% of the size of the whole dataset.

Generally, the more data you have, the smaller the test set can become.

This way, we can get an accurate estimate of performance if the algorithm was to see new data (assuming that the random elements of the data are stationary!)

Issues

However, there are issues with a simple holdout technique like this.

Put simply, think really hard about the test data - Is it independent from the training? - Does it represent realistic data?

Randomising data

Common structures found within data: - Ordering - Time - Key (e.g. when obtaining data from a database) - Value or label (e.g. all elements of class 0 first, then class 1, ...) - Geography - Only sampled from certain geographies. Doesn't scale to other geographies - Language

Thankfully the fix is simple. Always randomise data before training.

???

One issue is that the data in a set often has structure. E.g. it could be ordered or collected in such a way that when we pick a observation to train or test against, it doesn't represent the population.

Hyperparameter tuning

Imagine trying to tune a hyperparameter...

Can you see the issue?

We're using the test set to train hyperparameters!

???

We saw above that a common task is to alter some parameter of the model to improve performance.

If we repeatedly alter the *hyperparameter* to maximise the test set score, we're not really finding the best model. We are tuning the hyperparameters to best represent the test set.

Can you see the issue here?

We are using the test set to train out hyperparameters! Over time we would overfit our model to fit the test set!

The simplest fix for this problem is to introduce another holdout set called the validation set.

Validation set

The validation dataset is a second holdout set that is to be used when computing final accuracies.

Validation set issues

- Significantly reduces the amount of data available for training

???

The main issue with using a validation set is obvious from the image. It significantly reduces the amount of data that can be used to train the model.

This will ultimately affect model performance. Since more data usually means better performance.

The simplest fix is to retrain the best model using all of the training and test data put together...

But we're still not using the data in the test set to train the model. The data in the test dataset might be important to train the model.

So…

Cross-validation

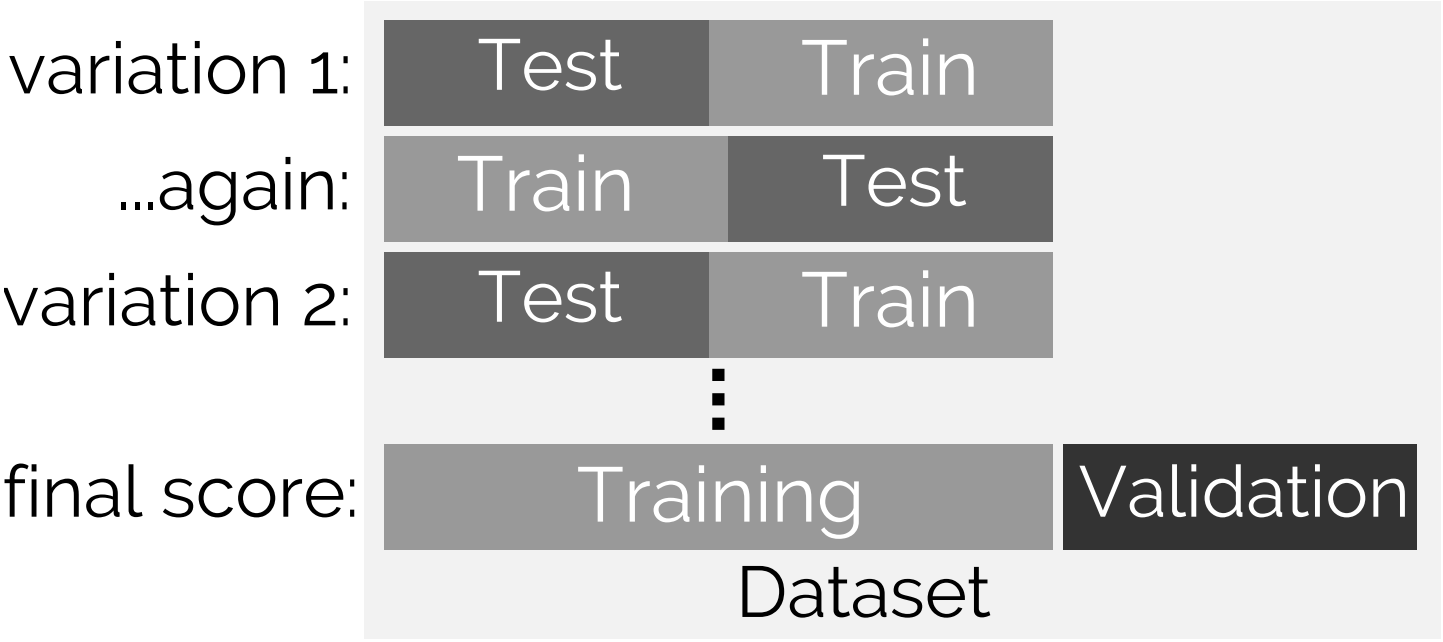
- For each new training run, pick a new subset of the data to train/test against.

???

Cross-validation is a process where we repeatedly perform a fitting procedure but each time pick a new test set to train against.

This way we use all of the test set to train the model with, but we are still able to pick the best model before final validation on an independent dataset.

How does it work?



The choice of the number of iterations is called the number of *folds*. In the example above there are `2` folds.

Benefit of Cross Validation

- Run through all the folds per training run

Then we have statistics about how consistent our model is over the various folds.

I.e. we can calculate the mean and standard deviation of our score.

Issues with cross validation

The main issue is the additional time required to repeat the training process for each fold.

This is increasingly problematic for complex models like deep learning.

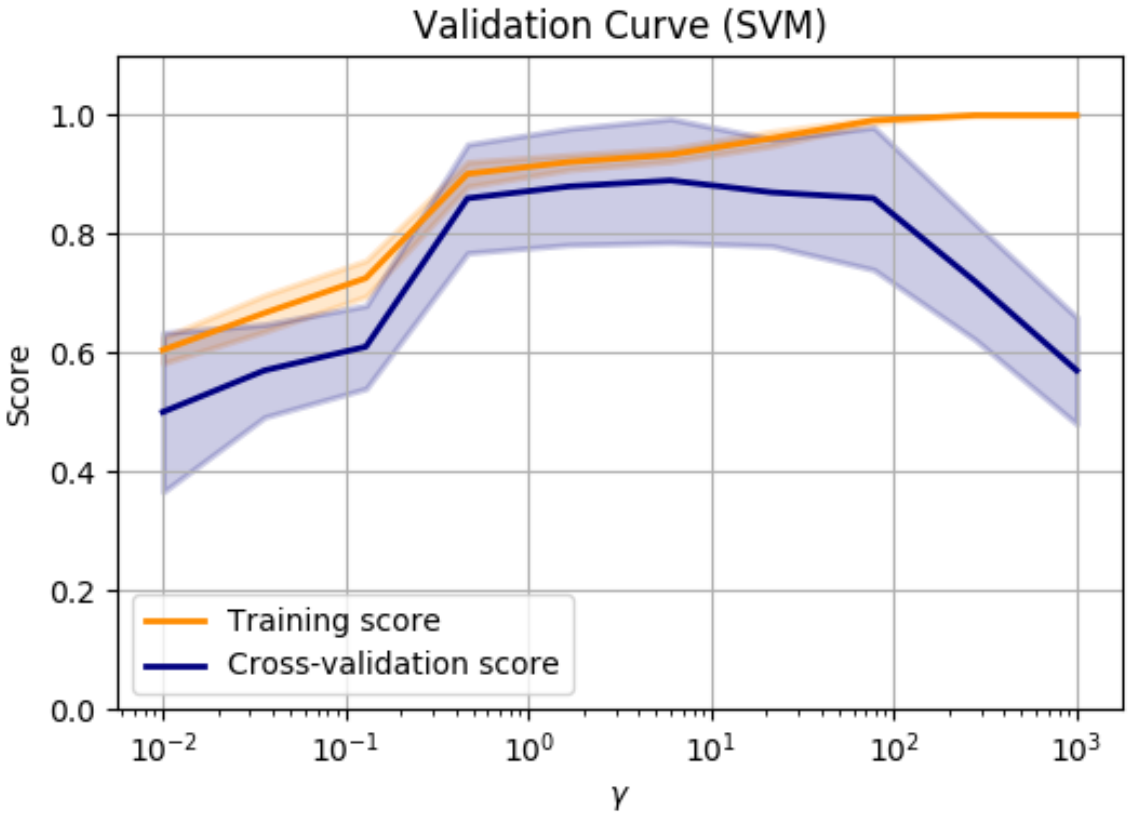
Let's talk about visualising overfitting

Validation curve

One simple method of visualising overfitting is with a *validation curve*, (a.k.a fitting curve).

This is a plot of a score (e.g. accuracy) verses some parameter in the model.

Let's compare the `make_circles` dataset again and vary the SVM->RBF->gamma value.



???

Performance of the SVM->RBF algorithm when altering the parameters of the RBF.

We can see that we are underfitting at low values of . So we can make the model more complex by allowing the SVM to fit smaller and smaller kernels.

But when we get to a value of about `10`, we can see that the performance on our validation set drops off dramatically.

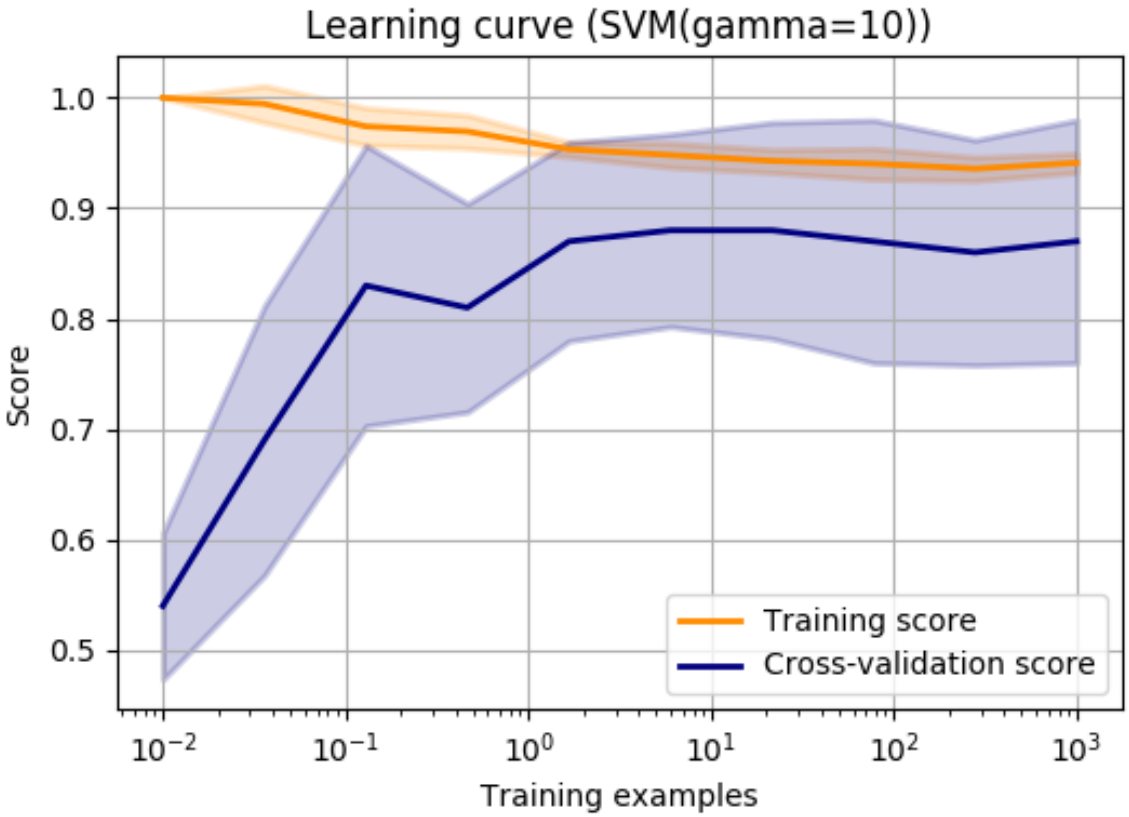
Learning curves

Another way to observe overfitting is with a *learning curve*.

A learning curve is a plot of the score vs. the amount of data used in the model.

(Or occasionally, in iterative training like deep learning, a plot of score vs. the iteration number is plotted with a similar result)

Again, let's look at the `make_circles` dataset again and vary the SVM->RBF->gamma value.



???

This curve shows us a few things.

Firstly, we can see that the final scores are approximately the same as the validation curve for this value of gamma (always good to sanity check!).

Next, we can see that the training score isn't quite 1.0.

I.e. it's not quite modelling the test data exactly. This could mean one of a few things.

1. It could mean that we aren't using a complex enough model to capture all the complexity in the data. This is call model *bias*.
2. Alternatively, it could mean that the problem isn't separable. There are some

instances that really look like the other class.

E.g. A person really was worthy of giving a loan, but they've had an accident and aren't able to work.

3. It could just be due to noisy data. Maybe you're IoT sensor doesn't give perfect readings.

In the first case, we could try a more complex model. It's always worth a try.

For second and third case, you may benefit from more data or this result might be typical for the domain.

One final use is that we can use the learning curve to decide whether adding more data is likely to improve the results.

If the validation score is much lower than the training score, then we can improve results by adding more data.

However, if the validation score matches the training data, then we know that we are performing at the best of our abilities.

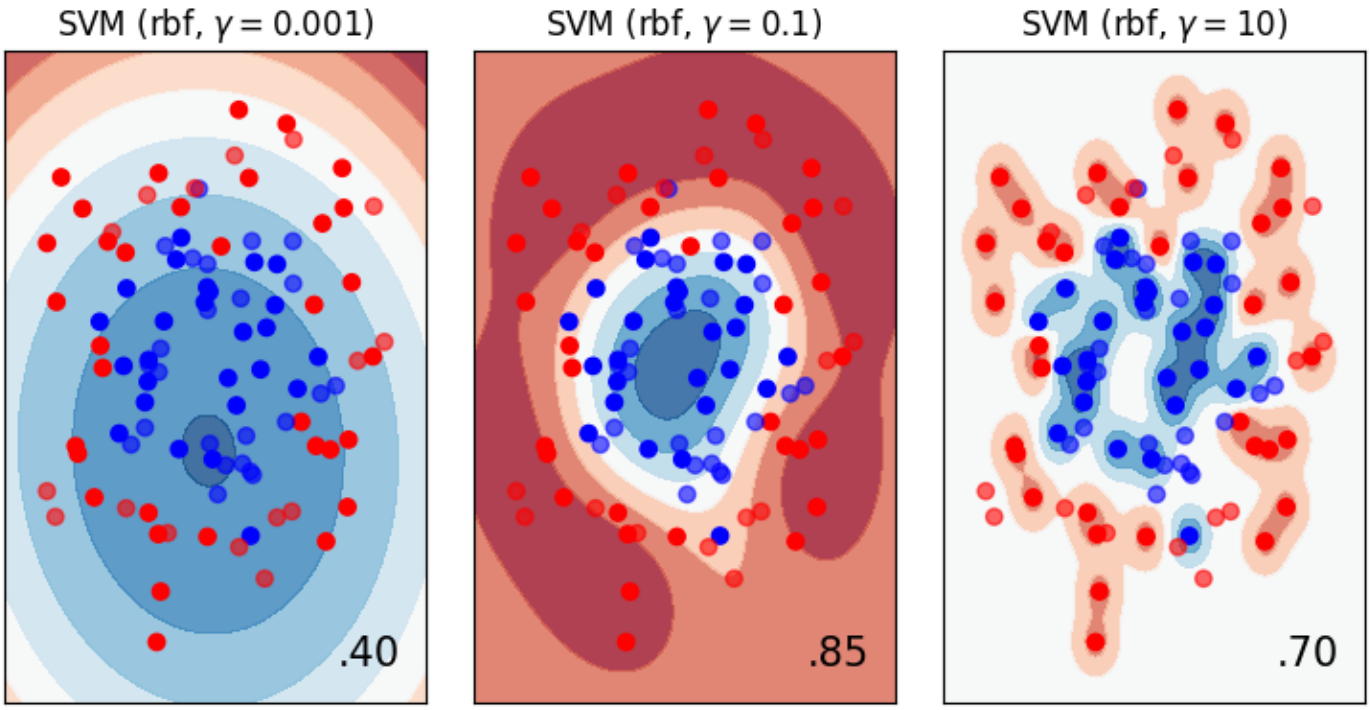
No amount of new data will improve the score with the given model.

Let's take another look at the SVM data vs. gamma...

From the validation and learning curves, we can draw these conclusions:

- We are overfitting at high values of gamma
- We need more data to make better generalisations
- There is some noise in the dataset

We can confirm these conclusions by plotting the data directly (which often isn't possible!)



More data usually means better generalisation

- As a general rule of thumb, more data means better generalisation.

We need more data to "fill in the gaps".

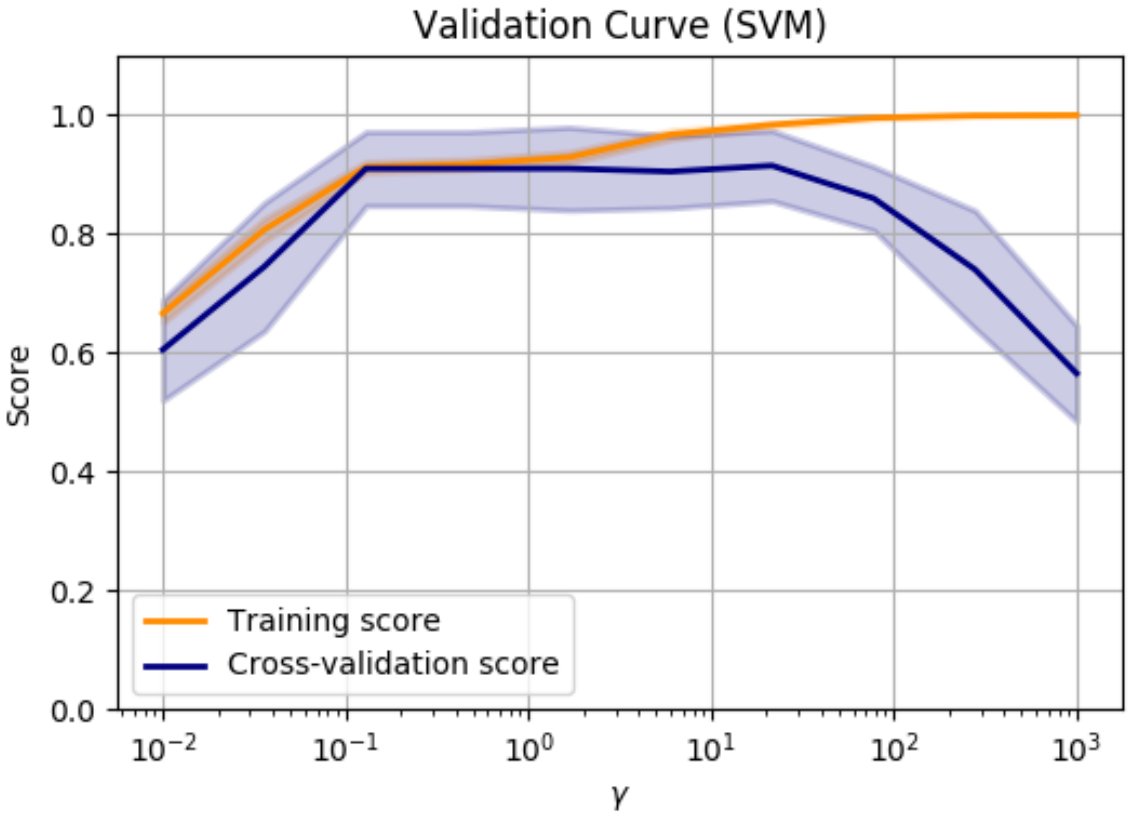
This compensates for more complex models.

???

We now have models that can model arbitrary complexities (e.g. deep learning, k-nearest neighbours) so we need more and more data to ensure that we continue to generalise to new observations.

For example, let's increase the number of datapoints in the circles dataset and re-plot the validation and fitting curves...

The same `make_circle` data with 200 observations (as opposed to 100).



- More data has improved the validation score at higher values of gamma.
- Our hunch about overfitting at high values of gamma was correct. Around a value of `1` the training and validation scores diverge.

Now, with a new value of `gamma=1` and more data, we get much closer to the training data (better generalisation) without overfitting.

Model complexity

A quick note on complexity (we've mentioned it a few times).

Different model types have differing abilitys to model complex data. For exaxmple a polynomial classifier has more complexity that a linear one.

This is refered to as the *variance* of the model. (I prefer the term complexity)

Clearly, there is a trade-off between generalisation and complexity, (a.k.a. bias and variance).

You can try to fit data better with a more complex model, but you will probably won't generalise well to new models.

This is a careful balancing act.

Parameters vs. hyperparameters

Parameters define a model that are learned during the training process.

- e.g. linear coefficients, decision hyperplanes

Hyperparameters are higher level concepts about the model that cannot be learned from the data directly

- e.g. Learning rate, number of leaves in a tree, gamma in RBF-SVM

Hyperparameters often require repeated runs to obtain optimal values.

Hyperparameter optimisation

We've seen how model parameters can be learned by minimising a cost function.

But hyperparameters cannot be learned directly from the data.

Instead we often perform a *search* to find the best parameters.

The simplest form is called a *grid search*. This is a brute force approach where we try all combinations of hyperparameters and pick (maybe by visualising) the best combination.

Regularisation

Finally, one note about how most models attempt to avoid overfitting.

They introduce *regularisation*.

This is the practice of actively penalising models with too much complexity.

How regularisation is implemented is different for each model, but the gist is the same:

Occam's Razor / Law of parsimony

Prefer the simplest model

---

class: middle, center

Workshop 5

---

name: nearest-neighbour

Nearest Neighbour

This section introduces the idea of "similarity".

Why?:

- Simplicity
- Many business tasks require a measure of "similarity"
- Works well

Business reasoning

Why would businesses want to use a measure of similarity? What business problems map well to similarity classifiers?

- Find similar companies on a CRM
- Find similar people in an online dating app
- Find similar configurations of machines in a data centre
- Find pictures of cats that look like this cat
- Recommend products to buy from similar customers
- Find similar wines

Similarity

What is similarity?

- We can say that two wines are similar if they have the same colour, alcohol

content, tastes, etc.

- We can say that configurations of machines are similar if they have the same RAM, CPU, HDD, etc.

In other words, we are comparing the features of the observation.

Observations are similar if they have matching features.

Distance

What is the best way of reducing the similarity into a single *distance* measurement?

The simplest conversion would be to use the Euclidean distance (a.k.a. L2 norm, Pythagoras' Theorem):

$$(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}|| = \sqrt{(\ _1 - \ _1)^2 + (\ _2 - \ _2)^2 + \ldots}$$

???

I.e. small distances are very similar, large distances are dissimilar.

Note that although we use the word *distance*, this measurement has no units. We're potentially comparing multiple types of features, so a real "distance" doesn't make sense.

Nearest Neighbour Algorithm

1. Calculate distance to all observations
2. Find the next closest observation

- Recommendations: List the next nearest
- Classification: Predict the same class as the nearest observations
- Regression: Predict the same value as the nearest observations

???

Now we have a measure of distance, we can perform the nearest neighbour

algorithm!

If we wanted to find the next similar wine for example, we'd simply calculate the distance between the current wine and all other wines.

Our next wine would be the one with the smallest distance!

If we wanted to perform classification, then we'd do the same but predict a classification using the nearest neighbour's class.

If we wanted to perform regression, then we'd do the same but pick the same value as the closest neighbour.

Simple!

---

class: pure-table, pure-table-striped

Example: Whiskey recommendations

Distillery	Body	Sweetness	Smoky	Medicinal
Aberfeldy	2	2	2	0
Aberlour	3	3	1	0
AnCnoc	1	3	2	0
Ardbeg	4	1	4	4
Ardmore	2	2	2	0

Columns:

```
['Distillery', 'Body', 'Sweetness', 'Smoky', 'Medicinal', 'Tobacco', 'Honey', 'Spicy', 'Winey', 'Nutty', 'Malty', 'Fruity', 'Floral', 'Postcode', ' Latitude', ' Longitude']
```

???

Do you like whiskey? (Say yes!)

If we had a dataset detailing whiskey characteristics, then we could take your favourite whiskey and return the most similar whiskeys as a personalised recommendation!

Algorithm

```
given favourite whiskey
foreach whiskey:
    dist = 0
    foreach feature:
        dist += (favorite[feature] - whiskey[feature])^2
    neighbours[whiskey] = dist
sort(neighbours by value)
print(first 5 neighbours)
```

Results

So, let's go for a super-smoke: Laphroig. The results:

```
[
    (array([4, 2, 4, 4, 1, 0, 0, 1, 1, 1, 0, 0, 'Laphroig'], dtype=object), 0.0),
    (array([4, 1, 4, 4, 1, 0, 1, 2, 1, 1, 1, 0, 'Lagavulin'], dtype=object), 2.0),
    (array([4, 1, 4, 4, 0, 0, 2, 0, 1, 2, 1, 0, 'Ardbeg'], dtype=object), 3.0),
    (array([3, 2, 3, 3, 1, 0, 2, 0, 1, 1, 2, 0, 'Clynelish'], dtype=object), 3.4641016151377544),
    (array([3, 1, 4, 2, 1, 0, 2, 0, 2, 1, 1, 1, 'Caol Ila'], dtype=object), 3.7416573867739413)
]
```

???

We can see that Laphroig matches itself perfectly, as it should.

Next up is Lagavulin and Ardbeg.

According to the smoke classifications (the third column), and others, these are pretty good recommendations.

However, this points out how important your data is. I've definitely have non-

smokey Ardbeg's before.

More than just similarities

- Classification: Predict the same class as the nearest observations
- Regression: Predict the same value as the nearest observations

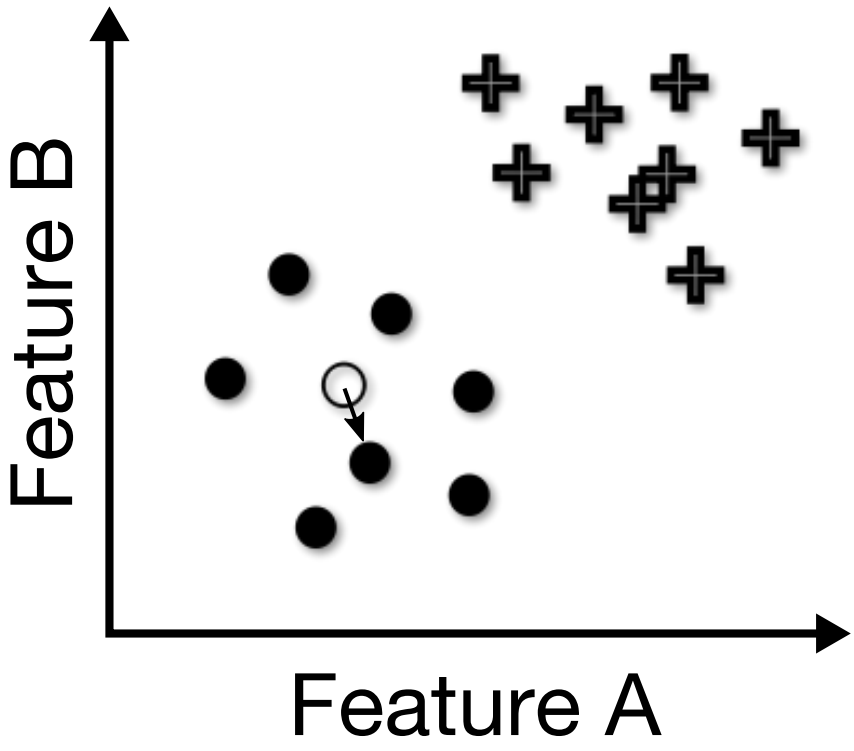
???

Remember for classification tasks, we want to predict a class for a new observation.

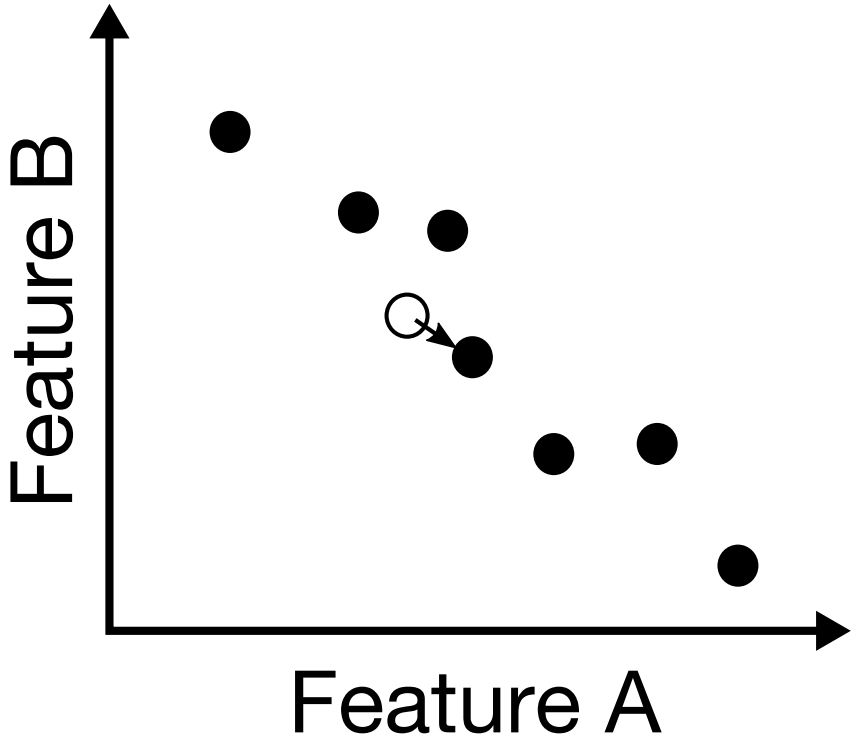
What we could do is predict a class that is the same as the nearest neighbour. Simple!

For regression tasks, we need to predict a value. Again, we could use the value of the nearest neighbour! Simple again!

Nearest neighbour classification



Nearest neighbour regression



???

We've used the nearest neighbour in these examples to demonstrate the idea.

But in reality we don't want to use a single neighbour. It could be noise.

Instead, we can use a weighting of some number of neighbours. For example in classification we would take predict the class which had the majority of nearest neighbours. For regression we might take the average value.

Equally, we could estimate the class probability by looking at the proportion of the nearest neighbours belonging to the predicted class.

(Beware of class probabilities on small numbers of observations!)

Generally, when we're using more than one neighbour the name of the algorithm is shortened to *k-NN*.

Where k refers to the number of neighbours used in the algorithm.

In general, higher values of k perform more averaging and give "smoother" results.

Let's take another look at the "iris" dataset with various values of k.

Bias and variance part duex

What we're seeing here is overfitting and over-generalisation in practice.

By choosing low values of k we are highly sensitive to outliers. We overfit.

But choosing high values of k, the model might not be complex enough to represent the data.

How do we pick a value of k?

We saw before how to use validation to pick make sure we are not over or underfitting.

We can do the same here, by varying the value of k and validating the result.

But be careful of the choice of K. It should be…

- a coprime of number of classes and K

E.g. two classes, if we used a value for K of 6, we could have ties.

- greater or equal to the number of classes plus one

To give each class opportunity to have it's say.

- High enough to avoid spurious results
- Low enough to avoid always picking the most common class

Pros/Cons

The pros of using k-NN are obvious and were stated at the beginning.

- The algorithm is very simple to understand.
- It is very flexible; you can use it for similarity matching, classification or regression.
- It is easy to tune; there's often only a simple parameter.

However, there are issues. Of which we discuss next.

Justification

- How can you justify the result?

E.g.

> *"your mortgage application was declined because of your similarity to three people that defaulted, whom were also Danish"*

- We've created a complex look-up table. It doesn't improve our knowledge of the data.

???

First, and this doesn't necessarily apply only to k-NN, how do you explain and justify the result?

Netflix justifies their k-NN recommendation by saying "you might like 'I'm Alan Partridge', because you liked both 'Brass Eye' and 'Borat'". And you're probably ok with this justification.

But if you were declined a mortgage because "your application has been declined because your circumstances (where you live) were similar to those who have defaulted". This might not be as easy to chew.

Furthermore, it is harder to learn something from the model, because we're not actually modelling anything. We've basically created a sophisticated look up table. Stakeholders may not like this result.

Dimensionality and domain knowledge

- Is it right to use the same distance measure for all features?

E.g. height and sex? CPU and Disk space?

- Some features will have more of an effect than others due to their scales.

???

In this version of the algorithm all features are used in the distance calculation.

This treats all features the same. So a measure of height has the same effect as the measure of sex.

The result is that some variables will tend to influence a result more than others.

The way around this is to make full use of feature selection and/or engineering (see later). I.e. remove or alter features that don't provide any descriptive power.

This leads to a requirement of being more of a specialist in the domain that you would need to be for other algorithms (e.g. deep learning).

Computational efficiency

- k-NN has no training period. Only prediction.
- Performance is proportional to the number of features and observations.

This means that finding the neighbours can be expensive in production.

You can find optimisations of k-NN to overcome this. E.g. `KDTree`

???

k-NN is particularly interesting because most algorithms take longer to train than they do to predict.

k-NN is different because there is no training, the prediction step of measuring the distances is all that is required.

The draw-back is that when numbers of features or numbers of observations are large, prediction times become increasingly large.

For tasks that require fast predictions (e.g. online advertisement recommendations) this might not be the best choice.

Although you could of course limit the dataset to a certain size to guarantee performance. Or use some random sampling to improve performance.

Distance Measures

- There are other distance measures (hundreds!)

???

We've already used the Euclidian distance, but there are many others (hundreds actually) which might make more sense depending on your application.

The reason is that distance measures tend to be domain specific.

E.g. there might be a good way to calculate the distance between taxi start and end points in a city (they often travel at right angles due to blocks of buildings).

To recap...

Euclidean (L2 norm)

$$(\mathbf{x}, \mathbf{y}) = ||\mathbf{x} - \mathbf{y}|| = \sqrt{(\ _1 - \ _1)^2 + (\ _2 - \ _2)^2 + \dots}$$

???

As before, the L2 norm tends to be influence more by outliers.

And again, it assumes a gaussian distribution (which is probably not true for complex multi-dimensional data).

This is the most popular distance measure, but probably not the right one.

Manhattan (L1 norm)

$$(x, y) = |x - y| = |\ _1 - \ _1| + |\ _2 - \ _2| + \dots$$

This is the L1 norm and is named *Manhatten* since it would be the total street distance you would have to travel in a city to reach a destination.

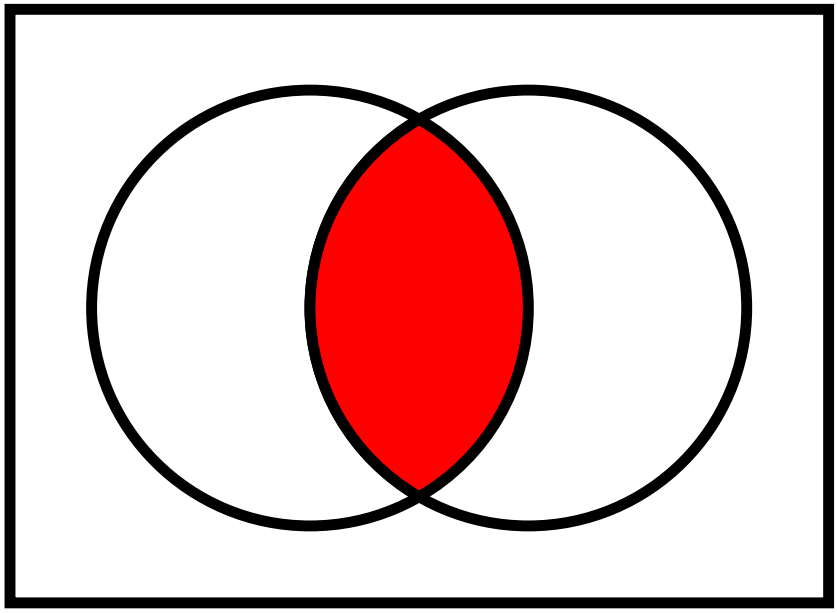
???

As before, the L1 norm also tends to be less influenced by outliers so might be a better choice for noisy datasets.

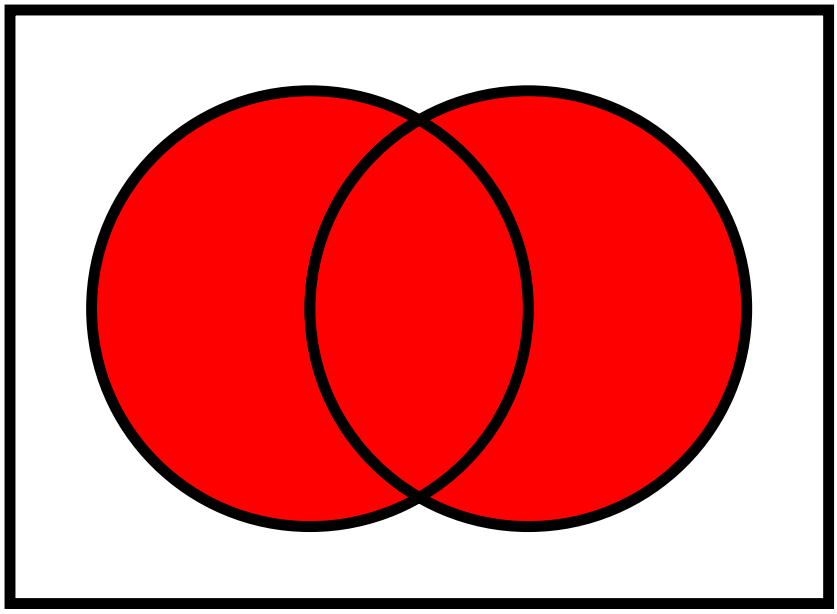
Jaccard

$$(x, y) = 1 - \frac{x \cap y}{x \cup y}$$

|Intersection|Union| |--|--| | .center[



] | .center[



] |

- Shared items / all items

E.g. if two whiskeys are peaty, that's significant. If they are not spicy, that's probably not.

???

It is the union (whole area) of two sets divided buy the intersection (shared area) of the two sets. The Jaccard distance is useful when you have highly categorical data. This is useful for situations when categories are similar, but dissimilar categories are not.

More distances measures are available here:

http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.DistanceMetric.html

Or over a hundred were collated here:

The Dictionary of Distances by Deza and Deza (Elsevier Science, 2006)

Combination functions

- Majority Vote
- Weighting the class based upon distance
- Dividing weighting by all observations to generate a probability estimate

- Using the median, rather than the average
- All of the above but for regression.

See the "weights" parameter in the sklearn k-NN documentation for example implementations: http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

???

Remember that after we've picked the k nearest neighbours, we then need to pick the best class or value or whatever.

What we need to do is combine the other observations into a result.

The normal k-NN algorithm uses a simple majority vote. But others include:

- Weighting the class based upon distance
- Dividing weighting by all observations to generate a probability estimate
- Using the median, rather than the average
- All of the above but for regression.

See the "weights" parameter in the sklearn k-NN documentation for example implementations: http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

---

class: middle, center

Workshop 6

---

name: clustering

Clustering

- *Unsupervised* learning
- Investigate the *characteristics* of our data

This is similar to segmentation, but is performed without labels

???

So far we have covered on main branch of data science: *supervised learning*.

To recap, this is the premise of using *labelled* data to train a model to make a prediction on a new observation.

In other applications we might want to investigate the *character* or our data.

For example, if we had a lot of data about our customers, then maybe we can investigate that data in order to target specific marketing strategies, or develop specific products for those segments.

Here, the key word is *segment*. This is still a segmentation task, but it is one that is done without labels.

This is an *unsupervised* task. The goal here is to *cluster* similar observations to form an abstraction about the *characteristics* of subsets of the population.

Whiskey revisited

Examples:

- You are the owner of a whiskey shop and you want to stock all major types
- You want to invest in stocks and to diversify, you want to pick a major stock from each type of business
- Your boss asks you to "see what you can do with this data"

To achieve these types of tasks we would perform clustering...

???

Remember our whiskey example from before?

Imagine that we were interested in clustering whiskeys, rather than finding similar ones.

This might happen if you were a shop and you wanted to stock a range of whiskeys that suited a range of tastes.

To do this we must first segment the whiskeys into categories, then we can pick one from each category.

Finance example: imagine we wanted to pick stocks. We wanted to pick a limited number of stocks because trading is expensive. But we also want to diversify our

portfolio.

What we could do is cluster all businesses in the FTSE 100 into categories, then pick the best stock from each category.

Given the business reasons to perform clustering, how do we do it? Well, there's two main ways...

Hierarchical clustering

- Create a taxonomy of observations, based upon distance measures

We can create the taxonomy in two ways:

- Bottom up (*agglomerative* clustering)
- Top down (*divisive* clustering)

???

The first is an attempt to create a taxonomy of observations, based upon their distance measures.

At the top level, there is a single cluster representing the entire dataset.

At the bottom level, there are as many clusters as observations; one cluster for each observation.

Hierarchical algorithms generally start at the bottom and iteratively including neighbours to the cluster, until there is only one cluster left. (*Agglomerative* clustering, as opposed to *Divisive* clsutering, which starts at the top and splits)

How and when samples are added to the cluster is at the heart of the algorithm...

Linkage

- We need a distance measure. Often called the *metric* in implementations.

Slightly different to nearest neighbours, as we are trying to form clusters.

- We need a measure that includes how clustered an observation is

This is called *linkage*.

???

Like the similarity measures, the hierarchical clustering algorithms require a measure of distance between clusters. This is often called the *metric* in the implementations.

Also, we need a function to decide which observations get added to which cluster. This is called the *linkage* function:

- *Ward* linkage minimises the variance of the clusters being merged.

  i.e. two clusters are merged that have the smallest variance (most similar)

- *Average* linkage minimises the average distance between clusters

  i.e. two clusters are merged that have the smallest average distance

- *Complete* (or *maximum*) minimises the maximum distance between clusters

  i.e. two clusters are merged that have the minimum maximum distance between clusters

As you might expect, there are more. Base the decision upon the business problem.

Colours = original class in iris dataset

Numbers = final agglomerative clusters

Dendrogram

A visualisation of hierarchy is called a *dendrogram*.

???

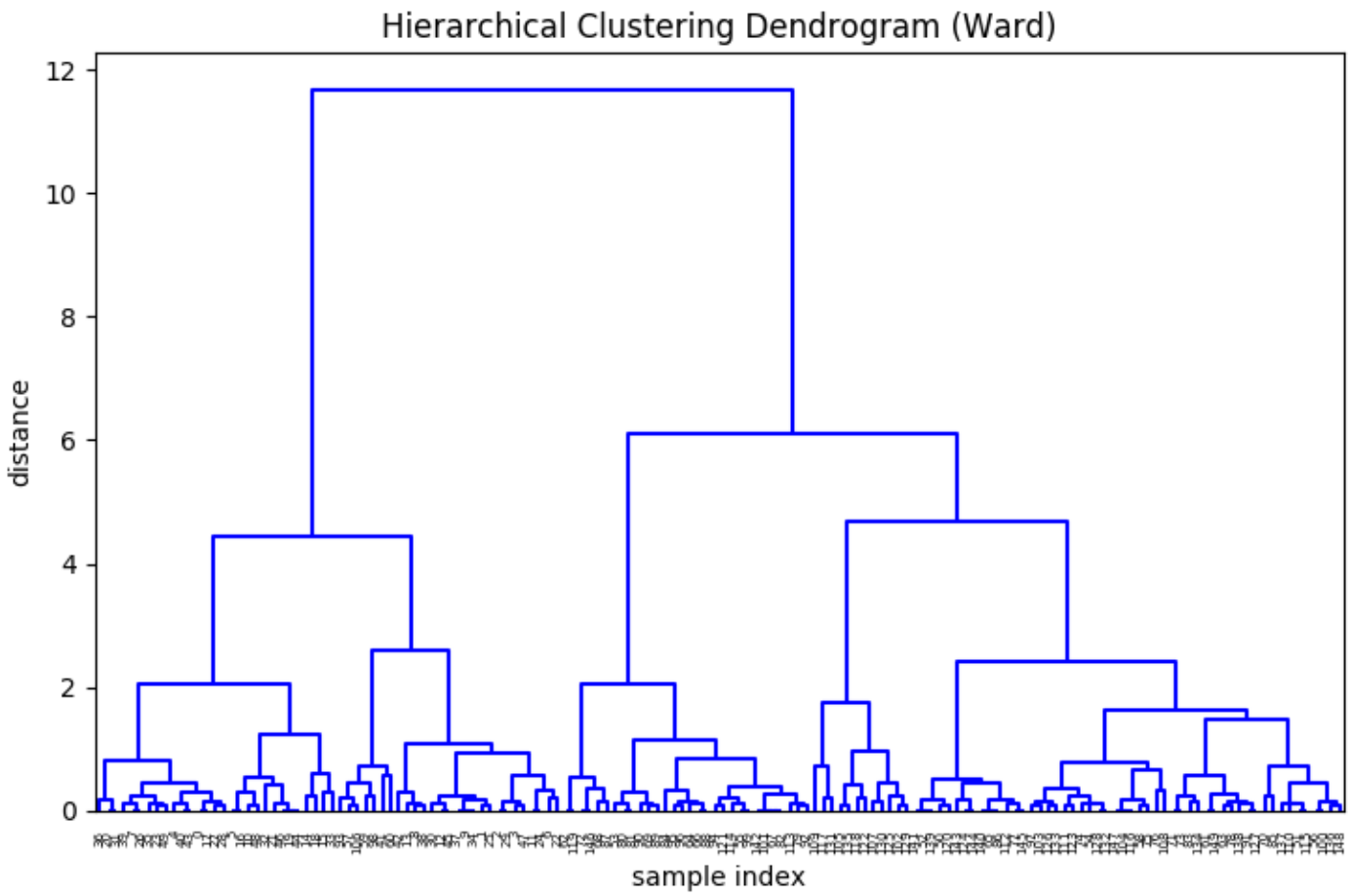
This is quite difficult to parse, lets spend a minute discussing it. For the benefit of readers:

- Vertical lines are observations/clusters
- Horizontal lines are merges of observations/clusters
- heights between horizontal lines are the distance between observations/clusters

Hence, larger heights are of more interest, as they signify large distances between clusters. I.e. these are probably separate clusters.

Be careful of the colouring. I have intentionally disabled it in the next plot, because it is confusing. It DOES NOT represent classes.

By default, it attempts to colour clusters that are close together. I.e. when it makes a big jump it starts another colour.



???

- Vertical lines are observations/clusters
- Horizontal lines are merges of observations/clusters
- heights between horizontal lines are the distance between observations/clusters

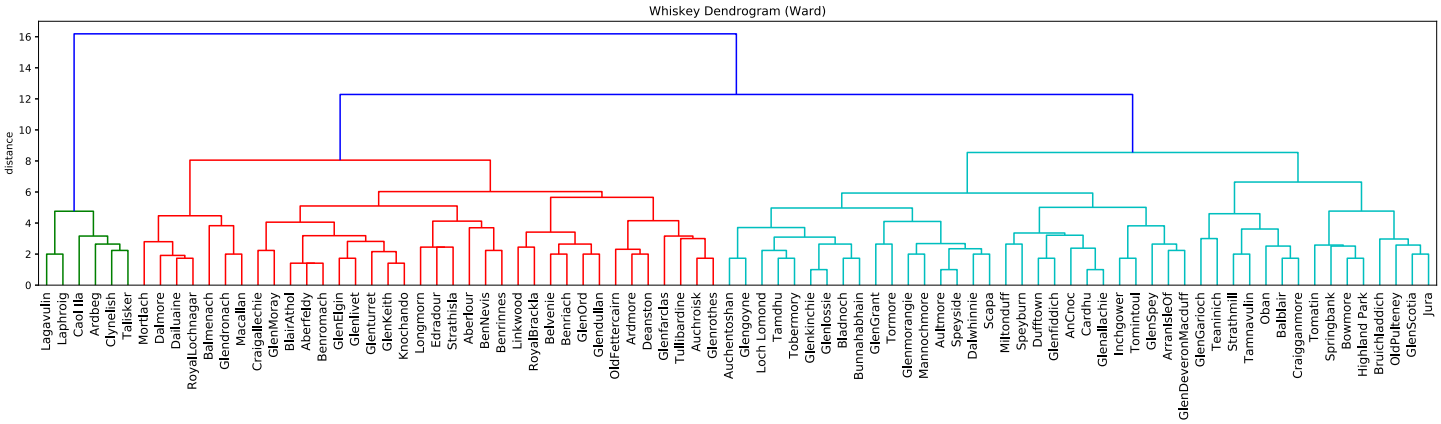
---

More typically, we would aggogate the observations into bins by truncating the tree:

Good tutorial here.

Let's take another look at a (zoomed in) dendrogram of the whiskey dataset.

Re: Previous requirement to hierarchically cluster data. the largest separators are: Island, Highland and Speyside. Let's pick them!



Clustering around centroids

- Concentrates on centering on classes, rather than merging/slitting observations.

The centre of a class is called a *centroid*.

This represents the typical features for that cluster.

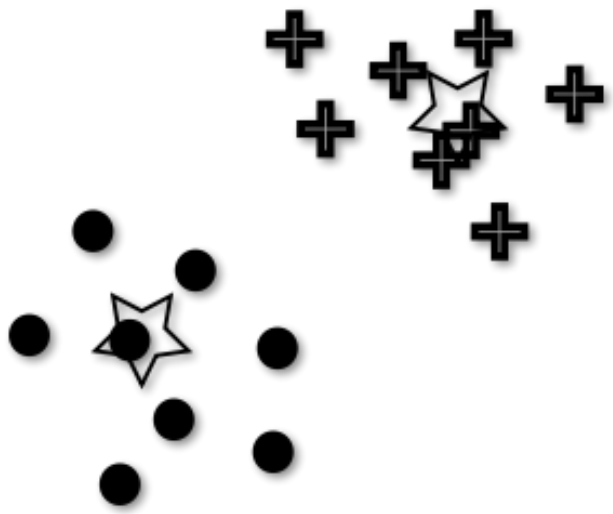
The most popular centroid-based algorithm is called *k-means*.

???

Another way of thinking about the clustering problem is to concentrate more on the classes themselves, as opposed to inferring classes from agglomerative similarity.

One common technique is to attempt to pick the centres of each cluster; the *centroid*.

Centroids



- K-Means defines the centroid as the geometrical mean of the distances of the current cluster

Assumes normally distributed. Other measures of "average" are available.

???

The definition of a centroid can have a whole range of meanings.

But K-Means defines it as the geometrical mean of the distances within the current cluster; hence *means*.

But always remember that there are other measures of "average". Again, this assumes gaussian noise and doesn't work well with outliers.

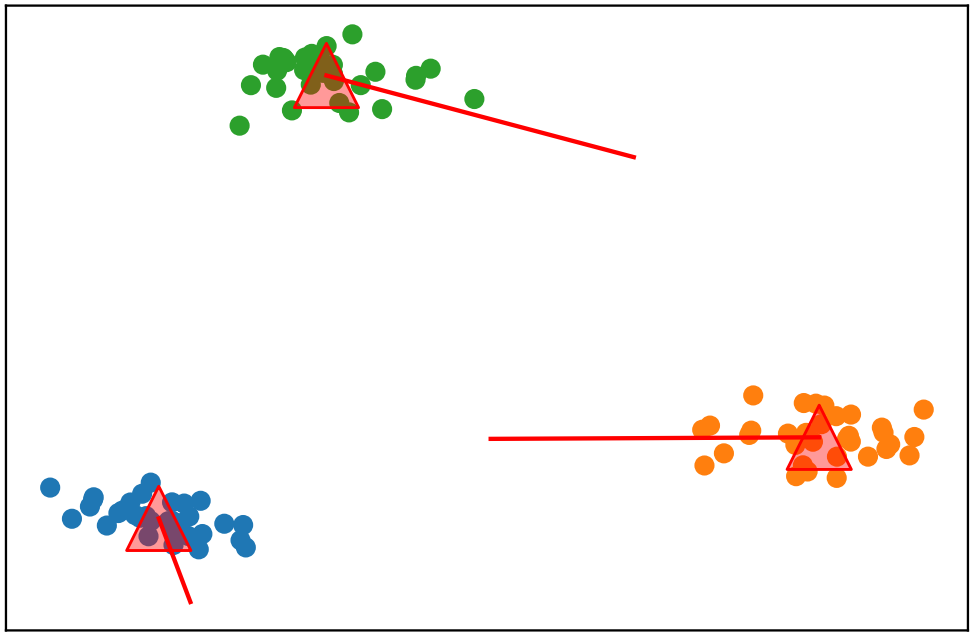
K-Means

The *K* in K-Means refers to the number of clusters. You must specify k.

Here in lies the most difficult aspect of the algorithm, picking a value for k.

Ignoring this for a second, let's look at the algorithm. The simplest implementation consists of:

```
Initialise centroid locations randomly
foreach centroid:
    foreach observation:
        assign observation to closest centroid
    calculate new centroid location
    if converged:
        break
```

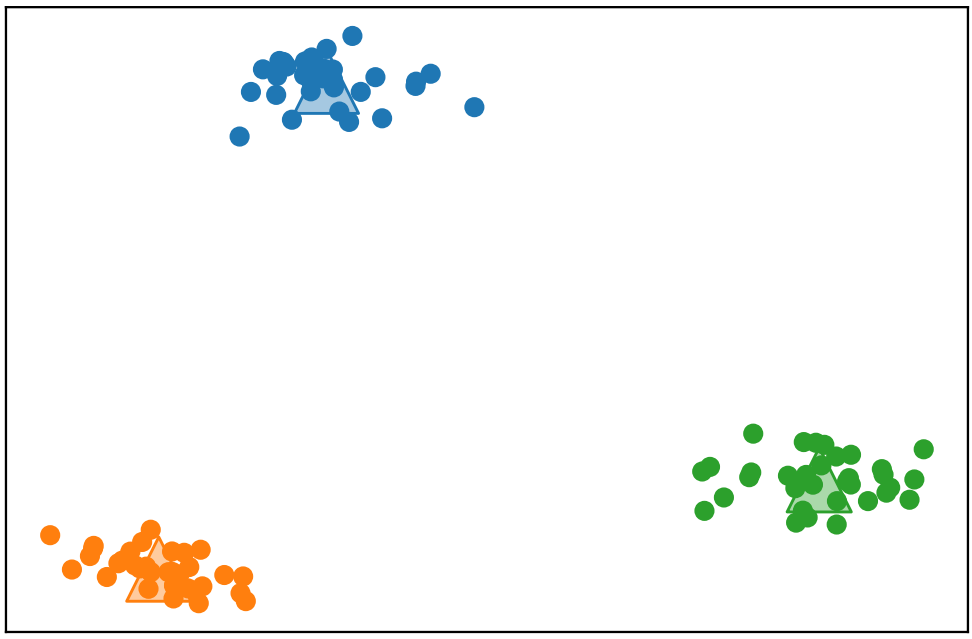


This is another great, simple algorithm. When we implement it ourselves, we will see a few issues. Namely:

- Sometimes two (or more) centroids cluster on the same location
- Centroids can get stuck in local minima
- Don't work well with noisy, mixed data.

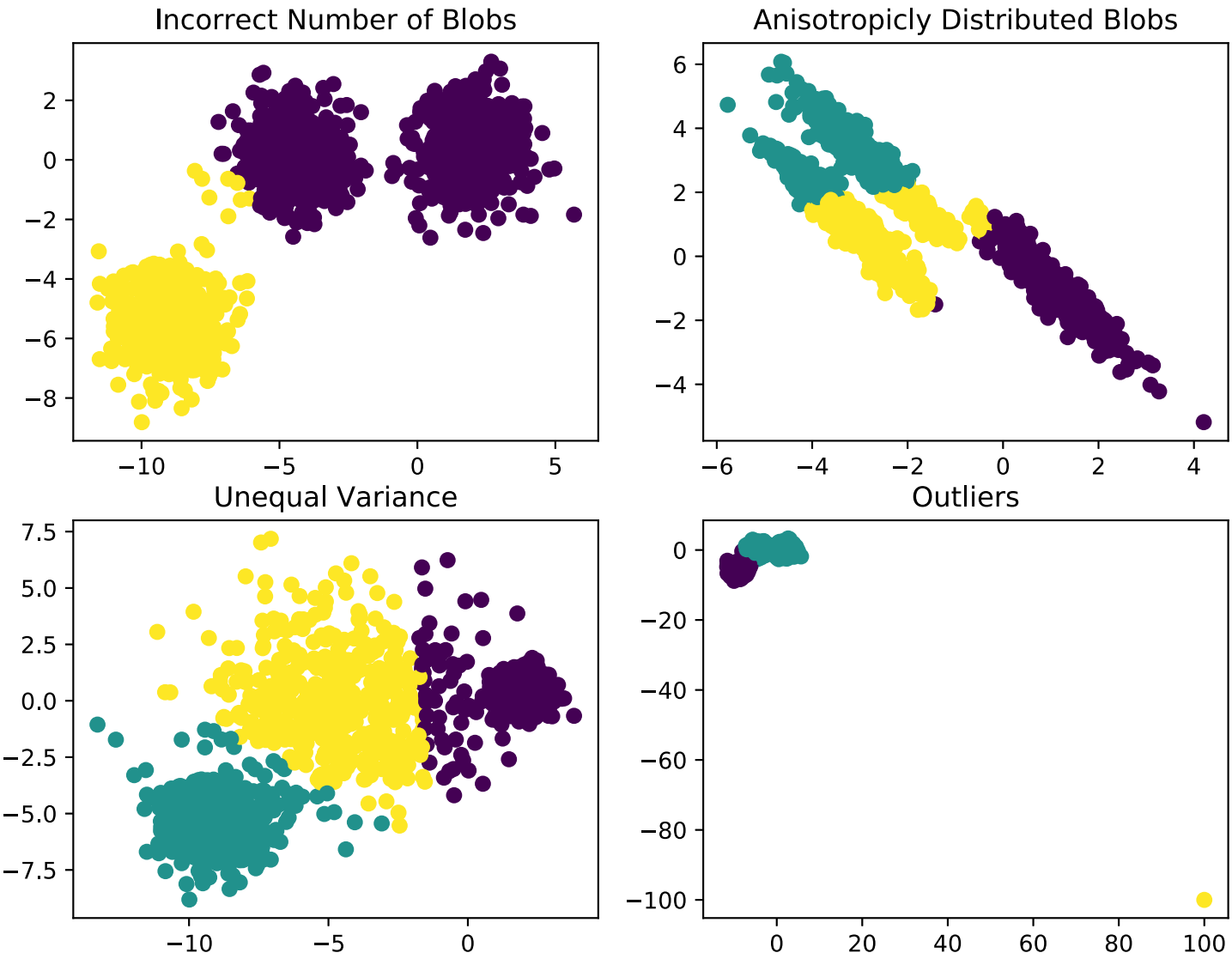
Luckily, the `sklearn` implementation tackles most of these problems and leaves us with the task of picking `k`.

For example, it sklearn is fully capable of handling this example out of the box (due to it using the K-Means++ algorithm):



But if we start to violate some of the assumptions about the underlying metric (L2

norm) then we start to get in trouble...

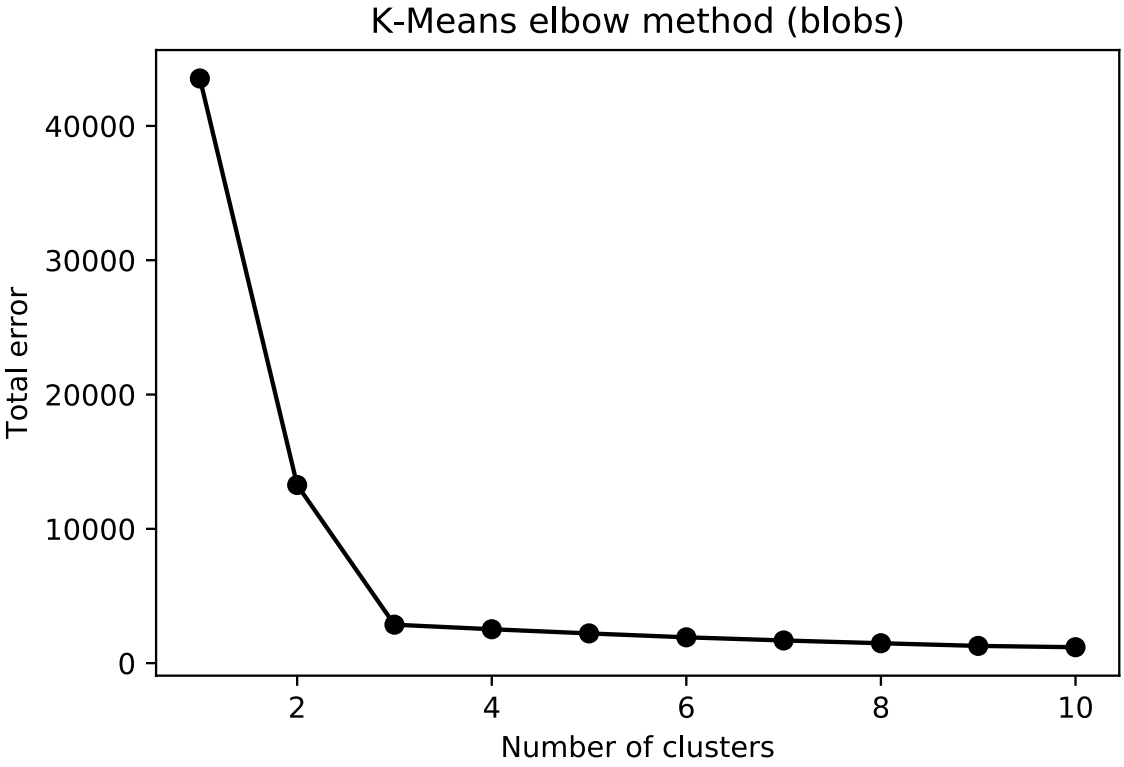


How to pick the number of clusters

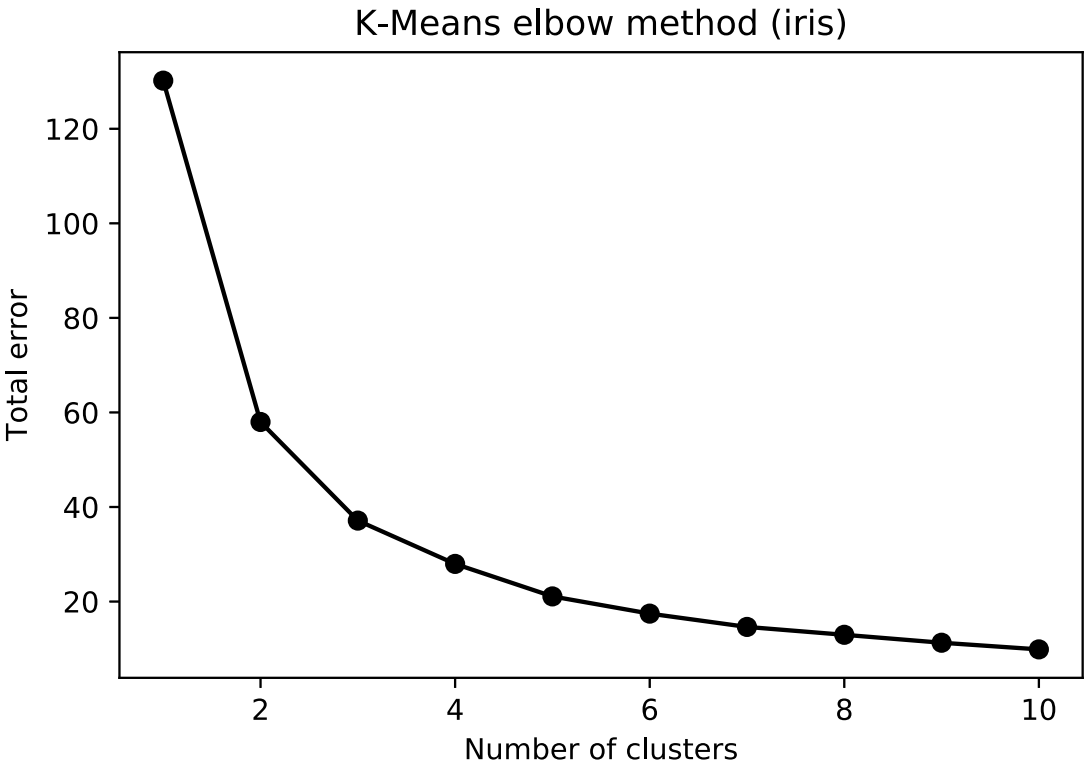
The first and most common way of picking the number of clusters is by the *elbow method*.

In this method, we plot the sum of the in-cluster error. I.e. the sum of the variance for each cluster.

By doing this, there will be an elbow in the data (depending on how well separate the classes are) and we can pick this elbow point as the "most descriptive" number of clusters.



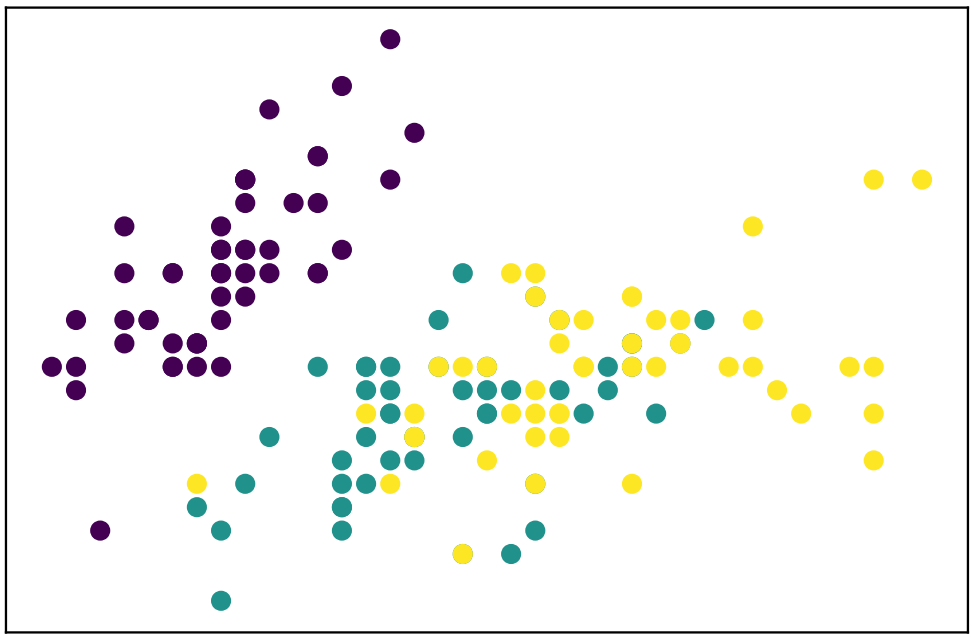
This is the error plot for the previous `blobs` data. We can see that the elbow is obvious.



For the 3-class iris data from earlier, the elbow is a little harder to spot. It is either 2 or 3, depending on your knowledge of the domain.

But this is usually indicative of your data being not well separated.

Indeed the third class in the iris dataset is not well separated in 2D...



Silhouette score

Another method (and score) of measuring how well a choice of `k` fits the data is with a plot of the silhouette score.

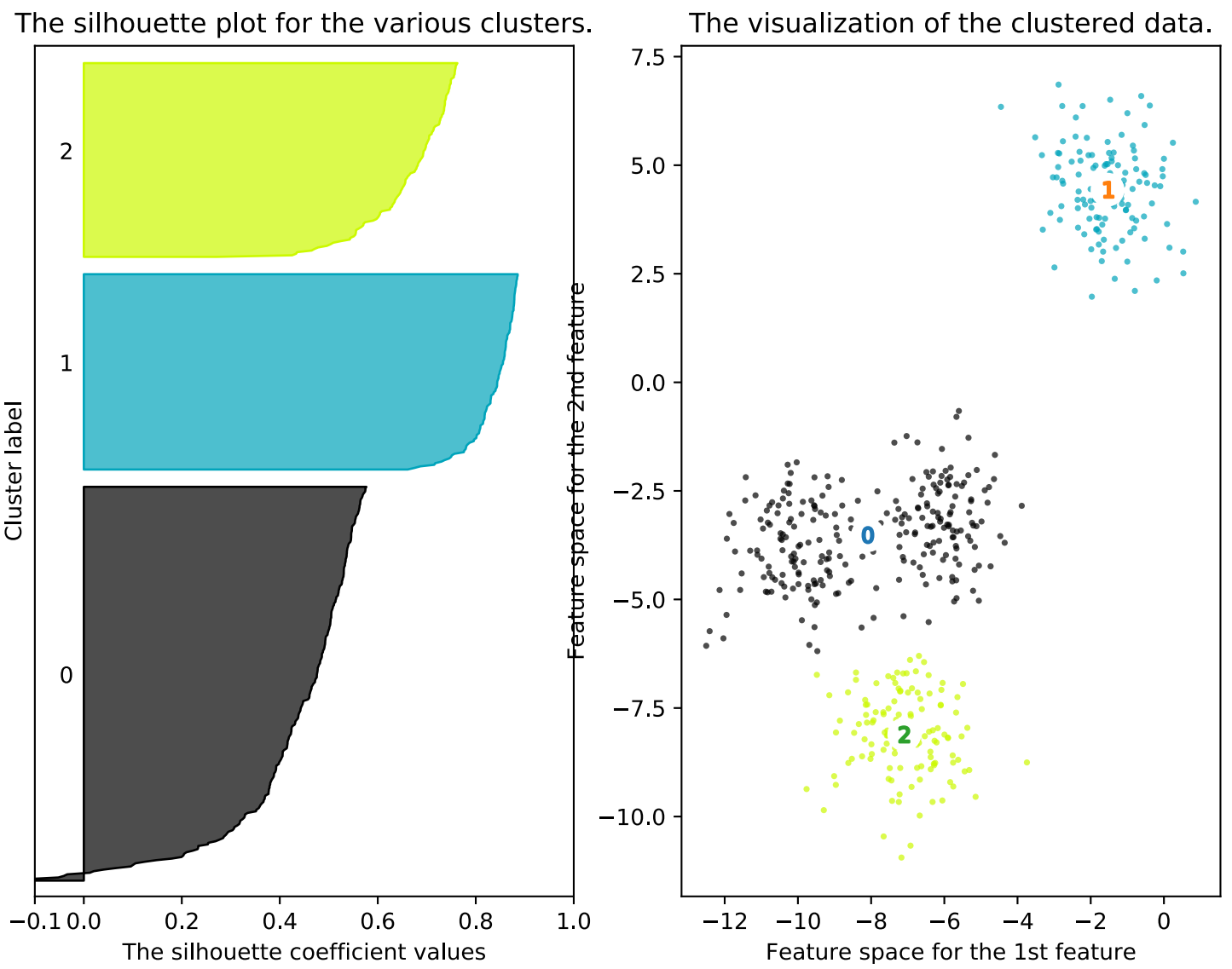
It is calculated by:

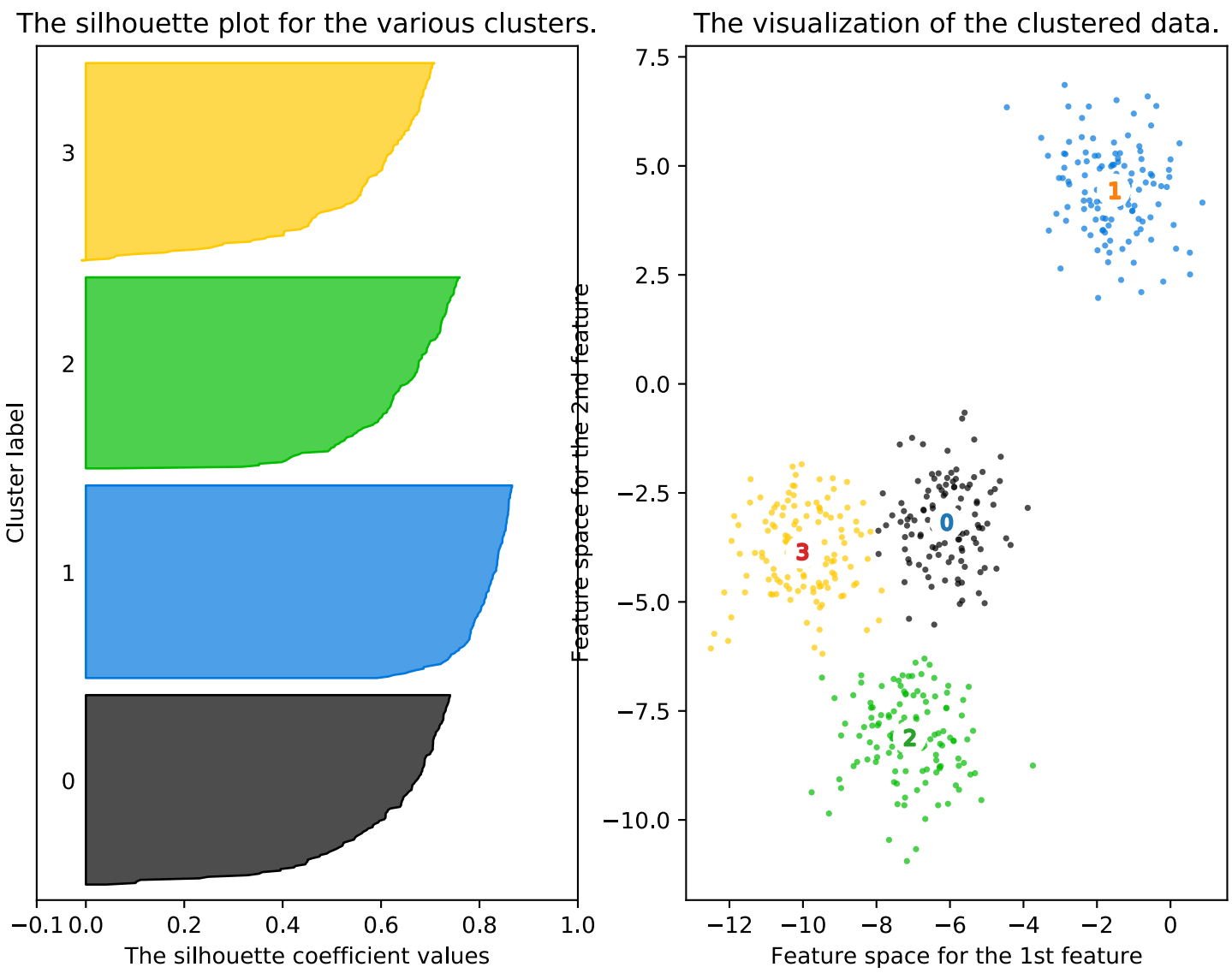
1. Calculate the average distance between each observation and all other observations in the same cluster: $(\ )$. (the *cohesion*)
2. Calculate the average distance between the observation and all other observations in the next nearest cluster: $(\ )$. (the *separation*)
3. Then the silhouette score is calculated by:

$$(\ ) = \frac{(\ ) - (\ )}{\max\{(\ ), (\ )\}}$$

i.e. the difference between the separation and the cohesion, divided by the greater of the two.

Zero when the separation and the cohesion are equal (i.e. two centroids in the same location). One when clusters are cohesive and well separated.


Silhouette analysis with n_clusters = 3
The silhouette plot for the various clusters.
The visualization of the clustered data.
Cluster label
2
1
0
−0.1 0.0 0.2 0.4 0.6 0.8 1.0
The silhouette coefficient values
Feature space for the 2nd feature
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12 −10 −8 −6 −4 −2 0
Feature space for the 1st feature
0
1
2



class: middle, center

Workshop 7

Example: Clustering sales information

Let's get deep into a realistic example.

Imagine we work for a bike manufacturer. Our boss has asked us to investigate some sales data. They have been fairly vague about the business requirements, but they said they want to "understand their wholesale customers better".

This is clearly an unsupervised learning problem. We want to use our super-powers to investigate the structure within the sales data.

The data we have today is some simulated bike sale data from: https://github.com/mdancho84/orderSimulatoR

class: pure-table, pure-table-striped

Orders

order.id	order.line	order.date	customer.id	product.id	quantity
1	1	2011-01-07	2	48	1
1	2	2011-01-07	2	52	1
2	1	2011-01-10	10	76	1
2	2	2011-01-10	10	52	1
3	1	2011-01-10	6	2	1

class: pure-table, pure-table-striped

Models

model	category1	category2	frame	price
Supersix Evo Black Inc.	Road	Elite Road	Carbon	12790
Supersix Evo Hi-Mod Team	Road	Elite Road	Carbon	10660
Supersix Evo Hi-Mod Dura Ace 1	Road	Elite Road	Carbon	7990

model	category1	category2	frame	price
Supersix Evo Hi-Mod Dura Ace 2	Road	Elite Road	Carbon	5330
Supersix Evo Hi-Mod Utegra	Road	Elite Road	Carbon	4260

class: pure-table, pure-table-striped

Shops

bikeshop.name	bikeshop.city	bikeshop.state	latitude	longitude
Pittsburgh Mountain Machines	Pittsburgh	PA	40	-80
Ithaca Mountain Climbers	Ithaca	NY	42	-77
Columbus Race Equipment	Columbus	OH	40	-83
Detroit Cycles	Detroit	MI	42	-83
Cincinnati Speed	Cincinnati	OH	39	-85

class: pure-table, pure-table-striped

Aggregate data

model	kind	type	price	1	2	3	4
Bad Habit 1	Mountain	Trail	3200	6.0	23.0	4.0	5.0

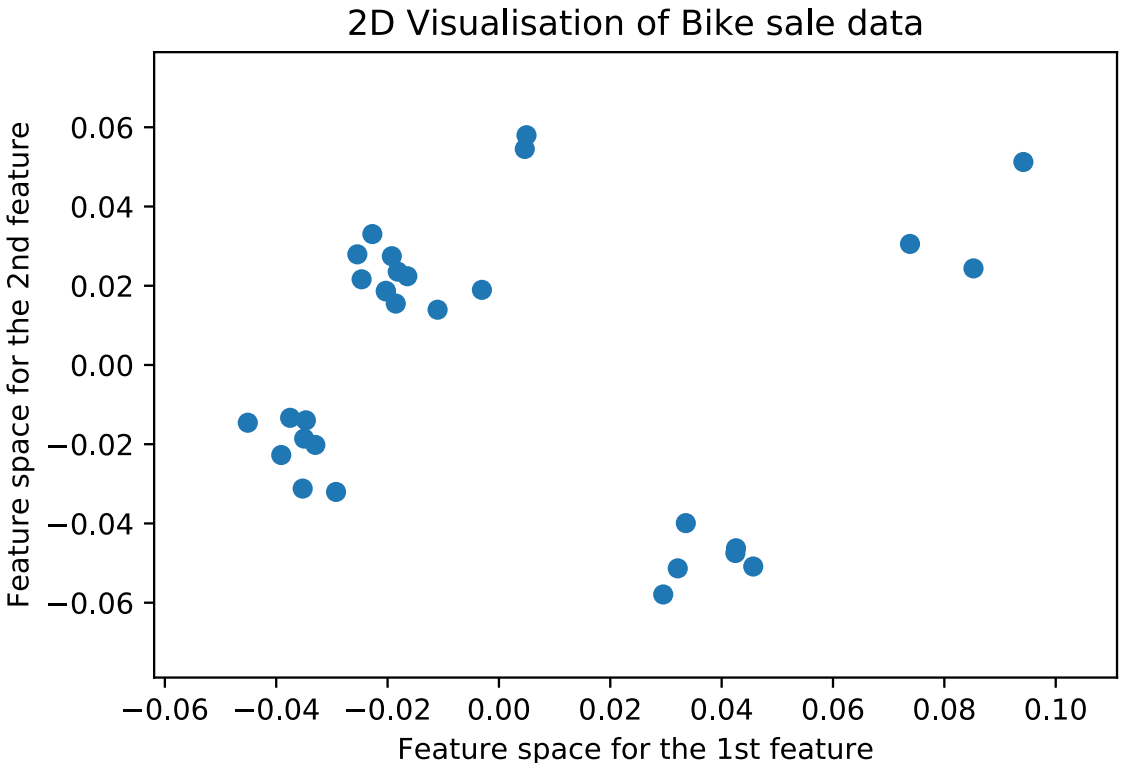
model	kind	type	price	1.0	24.0	3.0	8.0
Bad Habit 2	Mountain	Trail	2660	1.0	24.0	3.0	8.0
Beast of the East 1	Mountain	Trail	2770	2.0	27.0	0.0	6.0
Beast of the East 2	Mountain	Trail	2130	4.0	23.0	2.0	3.0
Beast of the East 3	Mountain	Trail	1620	1.0	13.0	1.0	6.0
…							

class: pure-table, pure-table-striped

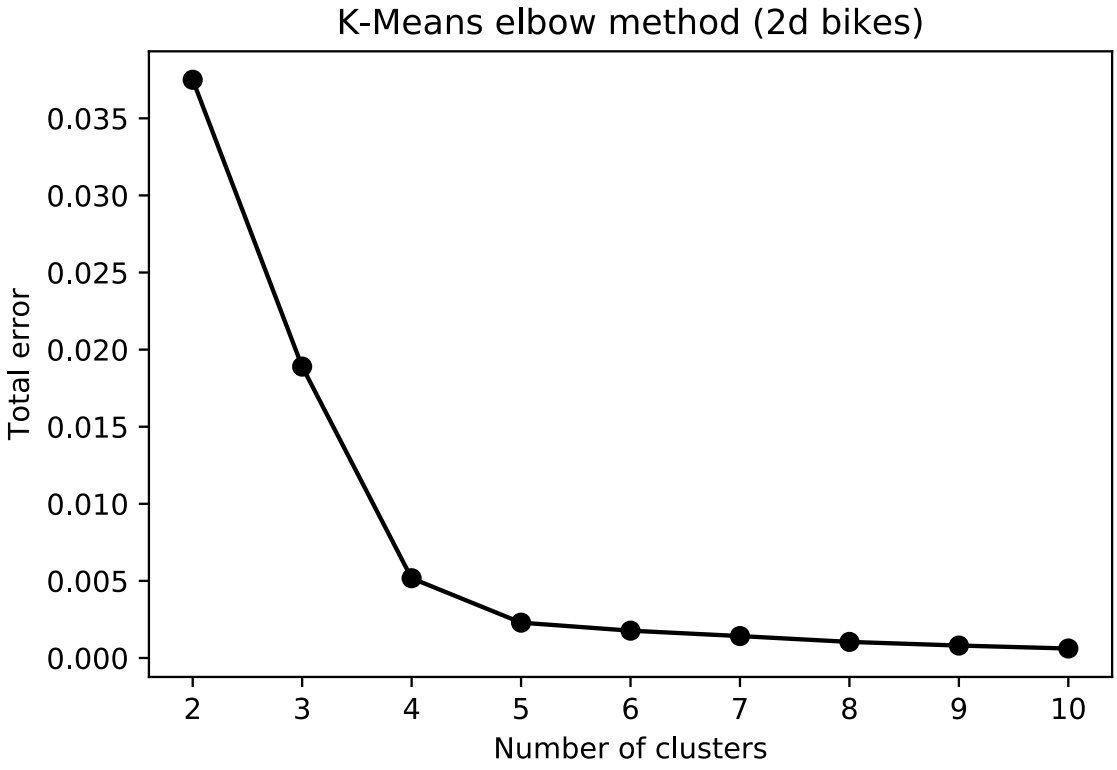
Scaled proportionate data

0	1	2	3	4	5	6	7	8	9
0.02	0.00	0.01	0.01	0.00	0.00	0.00	0.01	0.00	0.03
0.02	0.01	0.02	0.02	0.01	0.00	0.00	0.01	0.00	0.01
0.01	0.00	0.00	0.01	0.00	0.01	0.03	0.03	0.03	0.01
0.01	0.02	0.01	0.01	0.01	0.01	0.03	0.01	0.03	0.01
0.01	0.00	0.00	0.00	0.00	0.02	0.01	0.02	0.01	0.02
…									

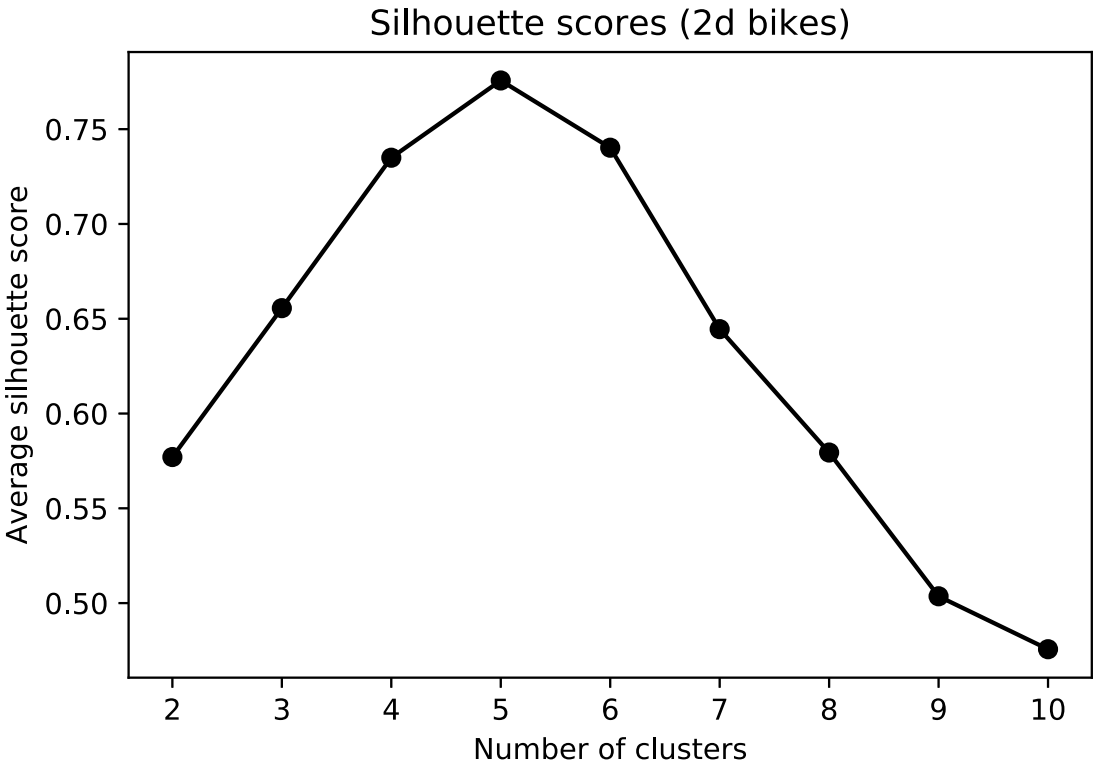
Multidimensional data mapped onto two dimensions



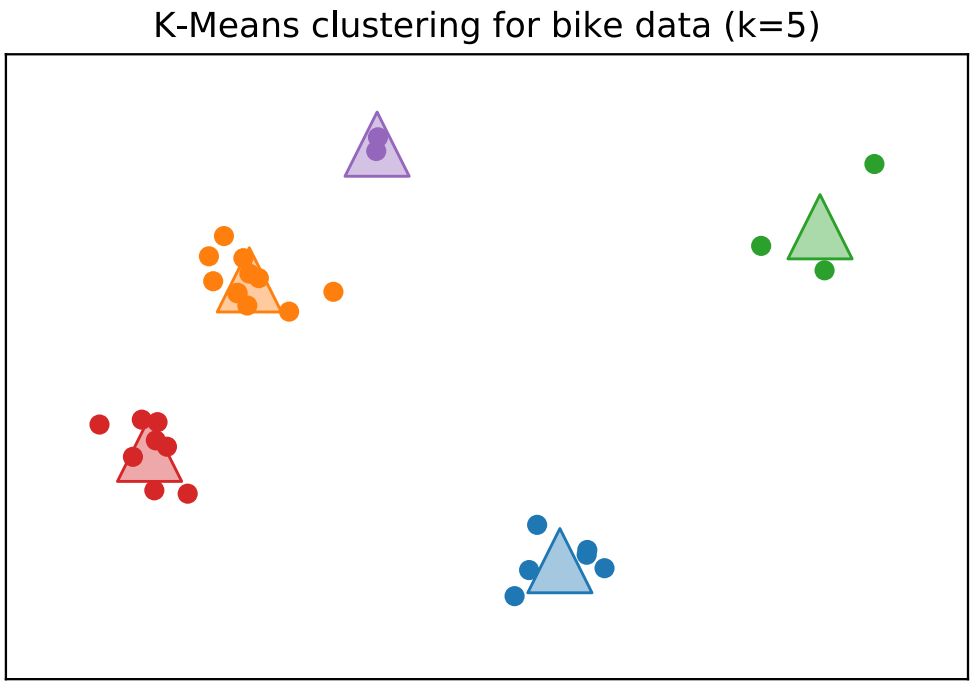
K-Means elbow method for bike data



Silhouette scores for bike data



Clusters



Results!

Now lets spend some time looking at the results. Based upon the clustering we have

performed, we could then delve into the clusters to figure out exactly what it is that sets these clusters apart.

We could go back and do this numerically from the data, but for now let's just investigate the main differences manually.

This is always a good idea to do anyway, because it instantly tells you whether you have some worthwhile results to show.

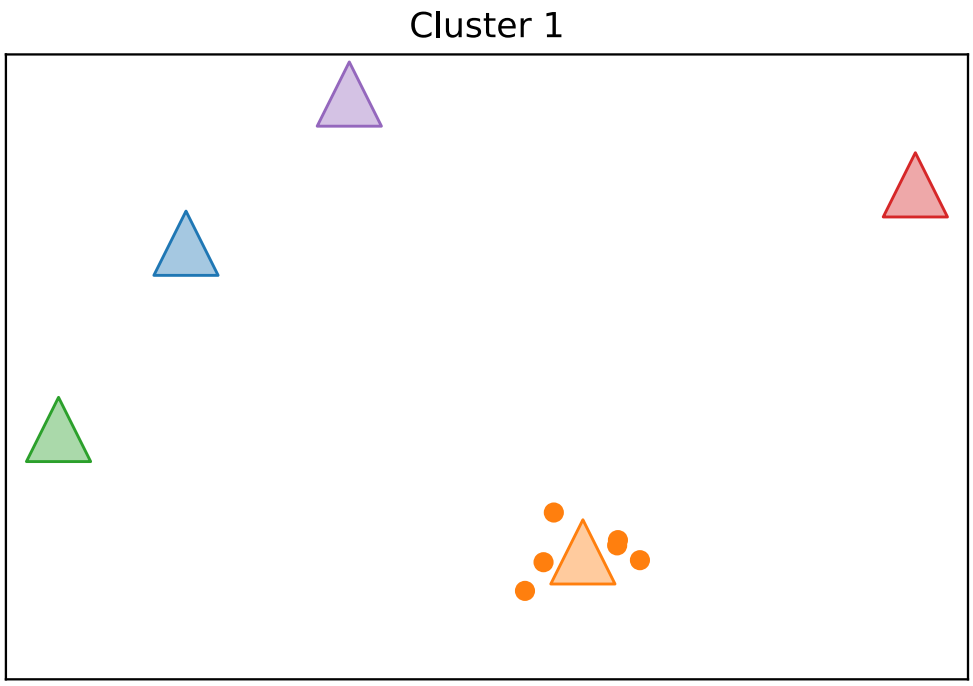
Some questions we could as:

- What are the best selling bikes for each cluster?
- Which cluster makes up the majority of our revenue?
- Is there a particular bike that sells well across all clusters?
- Is there a particular bike that only sells in one cluster? Do the other clusters know about this bike?

The questions at this stage are often endless, so we would now go back and ask the business guys what the goal of this analysis is.

Let's look at a couple of the 5 clusters...

Cluster 1 (Sales $12m)



---

class: pure-table, pure-table-striped

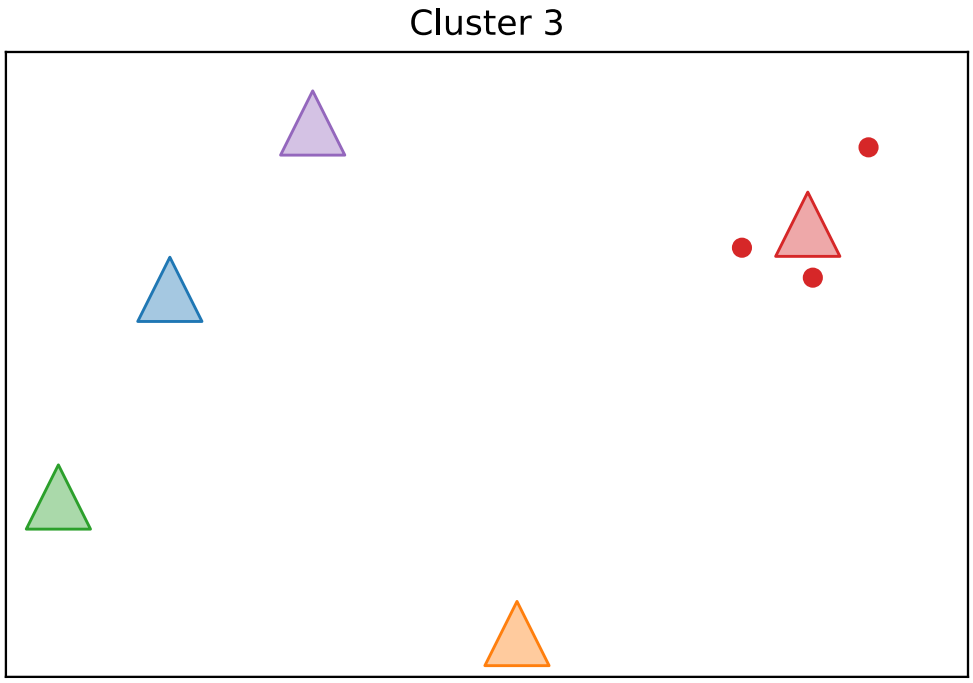
Cluster 1 (Sales $12m)

7	Nashville Cruisers
13	Oklahoma City Race Equipment
15	Austin Cruisers
21	Seattle Race Equipment
22	Ann Arbor Speed
25	New Orleans Velocipedes
26	Miami Race Equipment
29	Indianapolis Velocipedes

Best selling models:

model	kind	type	price	quantity
CAAD8 105	Road	Elite Road	1410	67
CAAD8 Sora	Road	Elite Road	1030	65
Synapse Sora	Road	Endurance Road	1030	61
Supersix Evo 105	Road	Elite Road	2240	61
Synapse Carbon Ultegra 3	Road	Endurance Road	3200	60

Cluster 3 (Sales $2m)



class: pure-table, pure-table-striped

Cluster 3 (Sales $2m)

```
1     Pittsburgh Mountain Machines
2       Ithaca Mountain Climbers
30                Tampa 29ers
```

Best selling models:

model	kind	type	price	quantity
CAAD12 Red	Road	Elite Road	3200	20
Slice Hi-Mod Black Inc.	Road	Triathalon	7000	19
Scalpel-Si Carbon 3	Mountain	Cross Country Race	5330	17
Scalpel 29 Carbon Race	Mountain	Cross Country Race	6390	16
F-Si 1	Mountain	Cross Country Race	2340	14

model	kind	type	price	quantity

Interpreting the results of clustering

As we have seen, the results are highly domain and business specific.

From a single set of data we can draw many conclusions.

This places an additional burden on the Data Scientist; they must also be domain experts.

But it also highlights how much value can be delivered from one small, inconsequential dataset.

One can imagine sending out a nice email to the sales team to collect this information. It wouldn't be so hard to do?

One really cool features of using K-Means is that it gives us a centroid. I.e the best description of an entire range of data.

We can use this to our advantage to promote "best of kind" types, or direct production efforts to make the "best" product.

The benefits are only limited by your imagination.

Automatic cluster descriptions!

This is where we start to get fancy!

I mentioned earlier that we could "back-propagate" (the technical term) the result of the classification back in to the original data domain. That is possible, but is often quite hard to do.

Another method is to use the clustering labels as the target for a supervised classification algorithm.

Thus we can build an algorithm to distinguish particular features that belong to the resultant class.

This works best when we have categorical data, like in the whiskey dataset, describing the taste in human terms.

---

class: middle, center

Workshop 8 - Challenge!

Review

What we have just gone seen is the CRISP-DM process in action.

.center[ ]

Clustering vs prediction/classification

- We've come across both types of modelling now: supervised and unsupervised
- Unsupervised questions often come in an open-ended form: "we'd like to know more about...", "investigate this data..." and so on.
- Clustering is an unsupervised task that is capable of interrogating the structure of the data to glean some very important business insights
- In contrast to prediction/classification, the results of the clustering task are limited only to your imagination (supervised tasks often have a decisive metric, like accuracy, or cost)

Vague business requirements

- And we've also seen the beginning of vague business requirements
- Other than data wrangling, this is the most frustrating part of data science
- Communication, education and training helps dramatically

End of Developer Data Science (Beginner)

Now on to the intermediate and advanced courses!

I have two other courses that cover more advanced topics.

The intermediate course covers: - Evaluating performance - Statistics, bayes rule, classifiers - Model management - More unsupervised techniques including all sorts of Principal Component Analysis and Self organising maps - A complete picture of all "standard" algorithms - Further excellent examples

---

The advanced course covers: - Text - Deep learning - Semi-supervised methods - Ensemble methods

---

Please visit

http://WinderResearch.com/training For more details, and public dates.

Online Training: visit https://TrainingDataScience.com

If you would like private, bespoke training for the rest of your company, please talk to me direct at:

training@WinderResearch.com

Thanks!

.center[Winder Research]

(1-day) Data Science and Analytics for Developers

(Beginner)

Beginners-level course for developers introducing Data Science, Analytics and Machine Learning. Learn how to utilise data to build products through a range of realistic theory and practice.

(1-day) Data Science and Analytics for Developers (Beginner)

In this one-day beginners-level course, you will be introduced to a range of

fundamental data science concepts. You will discover how to interrogate data, choose a which learning methods suit your problems and how to achieve results quickly.

Throughout the day theory will be complemented by "peer-instruction"; a teaching method that improves your learning experience by asking you to solve examples. This will provide you with valuable experience that you can apply to your own problems.

Who will benefit

This course is aimed towards developers, in which we will delve into the mathematics behind the code as well as developing real life algorithms in Python. One-to-one help will be provided for developers new to Python and all algorithms, frameworks and libraries used will be demonstrated by the instructor.

This is an introductory course, which is suitible for most users with limited development experience. Some experience of Python is helpful, but not necessary. No data science experience is expected.

What you will achieve

The day will comprise of a series of sub-hour theoretical sessions separated by practical exercises. It will cover a range of topics, but it is expected that you will be able to:

- Discuss the differences between types of learning
- Describe problems in a way which can be solved with Data Science
- Understand the difference between regression and classification
- Solve problems using regression algorithms
- Solve problems using classification algorithms
- Learn how to avoid overfitting and appreciate generalisation
- Develop features within data
- Describe how and where to obtain data

Topics covered in this training

- How data science fits within a business context
- Data science processes and language
- Information and uncertainty
- Types of learning
- Segmentation
- Modelling
- Overfitting and generalisation
- Holdout and validation techniques
- Optimisation and simple data processing
- Linear regression
- Classification and clustering
- Feature engineering
- An in-depth practical example demonstrating the day's concepts