Contents

This summary is available here on Github https://github.com/tintin10q/csai_ml_final_summary

0.0.1 Disclaimer

Although I have tried my best to make sure this summary is correct, I will take no responsibility for mistakes that might lead to you having a lower grade. If you do end up finding a mistake you should send me a message or make a Github issue, so I can update the summary 😀

0.0.2 Support

Do you appreciate my summaries, and you want to thank me then you can support me here:

PayPal or Tikkie

BTC: 1HjFv4NYiTwxdcnNpPEzaPY8vvYWXQnDgx ETH: 0xc0b430594A957A6A38203F45bd91f5d3568

---

1 Machine Learning CSAI Final Summary

1.1 3 types of models

All the ML techniques can be put into another 3 categories.

- Instance based classifiers
  - Uses observations directly without models
  - The only one we have that does this is KNN
- Generative
  - Build a generative statistical model by assuming data comes from a normal distribution
  - Bayes classifiers
- Discriminative
  - Directly estimate decision rule/boundary
  - Decision trees

2 Reminder about probability theory

This is how Bayes works.

2.1 Variables

The basis of Bayesian probability theory lies with **random variables** that represent events in the future that could happen. These refer to an element whose status is unknown. For example `A = "it will rain tomorrow"`

The **domain** is the set of values a random variable can take, also to be seen as the set all the possible answers to a question, or all the possible events that could happen. This can be the following types: **Binary**: Will it rain? (yes, no) **Discrete**: How much will it rain? Integers (With numbers), **Continues**: How much more did it rain today? Floating points (With change and)

Then we define the P() function which will give you the chance of a certain event happening.

P() has these rules: > 1. 0 <= P(A) <= 1
> The output is always between 0 and 1.

> 2. P(True) = 1, P(false) = 0

```
> true = 1, false = 0
```

> 3. P(A ∪ B) = P(A) + P(B) - P(A ∩ B)

```
> Set Theory. The chance for A AND B is P(A) + P(B) - P(A)   P(B)
> So A AND B is P(A) + P(B) - (What A and B have in common)
```

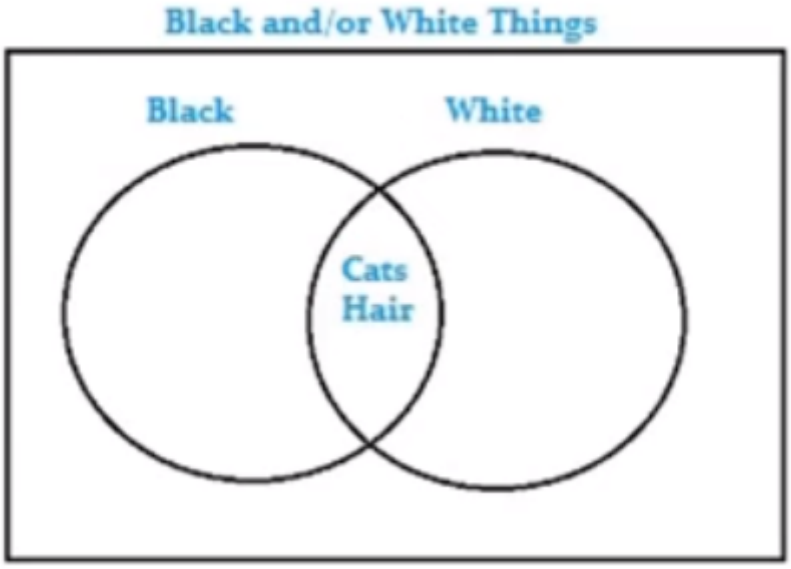


Figure 1: Set Theory explanation

How likely one certain outcome is, is called the **prior**. So the outcome of a P() is a prior. For example:

- P(Rain tomorrow) = 0.8

- P(No rain tomorrow) = 0.2 The probability of at least something happening is 1. So the sum of P() for all possible outcomes is one.

So a prior is the degree that you believe a certain outcome before it happens and one is chosen if you have no other information.

2.2 Example

Red	Blue
1	0
0	1
1	0
0	0
1	0
0	1
1	1

Here P(Red) = 0.5 or 4/8 and P(Blue) = 0.5

2.3 Joint probability

This is where you want to know the prior for A and B or P(Red) and P(Blue) or P(A, B) or P(Red, Blue). It is the chance that multiple events happen together. In this case it is = 1/8

Now it is very important if the variables are independent or not. If something is independent then the joint probability is just P(A) * P(B). If it is not independent then you need to talk about conditional probability.

2.4 Conditional probability

In some cases given knowledge of one or more random variables we can improve our prior belief of another random variable. As soon as you know that variables are not independent you need to talk about conditional probability.

This is where | comes in. (A = 1|B = 1) this asks: > What is the chance that if B happens A also happens the outcomes where A is true if B is true. You could say this as the chance of A given B.

The joint distribution can be specified in terms of conditional probability.

> You can then combine these two to get the joint probability. It looks like this: `P(x,y)= P(x|y)*P(x)`

2.5 Bayesian Probability Theorem

It turns out that the joint probability of A and B is the prior of A times the conditional probability of A and B. So `P(a,b)= P(a|b)*P(a)`

Now that is really great because we now we can make the bayes rule: From joint probability we can say `P(x,y)= P(x|y)p(y)= P(y|x)P(x)` and this gives us conditional probability.

> It looks like so: `p(y|x)= p(x|y)p(y)/p(x)`

Now if we put this back into ML terms. x = features, y = label. > P(y|x) –> What is the chance for this label y given features x

2.5.1 Example

Likelihood Table		Pass Final Exam		
		Yes	No	
Teachers	Tom	13/56	13/24	26/80
	Ben	24/56	2/24	26/80
	Larry	19/56	9/24	28/80
		56/80	24/80	

What is the probability you will pass the final exam if your teacher is Larry?

Likelihood Table		Pass Final Exam		
		Yes	No	
Teachers	Tom	P(Tom|No) = 13/56	P(Tom|No) = 13/24	P(Tom)= 26/80
	Ben	P(Ben|Yes) = 24/56	P(Ben|No) = 2/24	P(Ben) = 26/80
	Larry	P(Larry|Yes) = 19/56	P(Larry|No) = 9/24	P(Larry) = 28/80
		P(Yes) = 56/80	P(No) 24/80	

We want to know the prior of passing your exam if your teacher is Larry. We need the following. Let's also replace the P(x) and alike with semantics for this. > P(x) = P(Larry) = 28/80 = 0.35 > P(y) = P(Yes) = 56/80 = 0.7 > P(x|y) = P(Larry|Yes) = 19/56 = 0.34 (Prior of your teacher being Larry, and you're passing.)

Ok so with that we can calculate P(y|x) like P(y|x) = `P(y|x)= P(|y)p(x)/p(y)` or `P(Yes|Larry)= P(Larry|Yes)*P(Larry)/P(Yes)` So if we write it out: `P(Yes|Larry)= 0.34*0.7/0.35 = 0.68` Meaning the final probability of passing the exam if your theater is larry is 0.68

3 Naive Bayes

With the rules from above you can make a machine learning classifier. It is called Naive Bayes classifier. This is a **generative** based classifier. Meaning it builds a generative statistical model based on the data. This classifier is based on the predictions of that model. We do this by just giving the things below we saw before new names.

> - P(y|x): The posterior probability of a class (label) given the data (attribute) -> The probability of the label given the data
> - P(y): The prior probability of a label -> How likely is a certain class?
> - P(x|y): The likelihood of the data given the class -> Prior Probability of the data given a class. Opposite of (y|x)
> - P(x): The prior probability of the data

We are after P(y|x) the label given the data. We can calculate this with P(y|**x**) = P(**x**|y)*P(y)/P(**x**).

In practice, you will always have multiple features, so it looks like P(y|**X1** ... **Xn**)

This classifier is called a **Naive** Bayes because it assumes that all the features are independent. This makes it so you P(**X1**|y, **X2**, **X3**...**Xn**) = P(**X1**|y)

This gives us:

P(y|X1...Xn) =

$$\frac{p(y)\prod_{i=1}^{n} p(x_i|y)}{p(x_1,\ldots,x_n)}$$

Figure 2: Final formula naive Bayes

His means multiply P(y) with all the priors of your classes and divide by the joint prior of all the data. Then you take the class with the highest **posterior probability**. This is what you choose as your prediction.

> **posterior probability** is the prior * likelihood or the P(y) * P(**x**|y) of the equation. So we want the highest P(y|x)

3.0.1 Non discrete data

So far we assumed that the data is always discrete but usually with machine learning this is not the case. The data is often continues. For these types of data we often use a **Gaussian model** that assumes your data is normally distributed! In this model we assume that the input X is taken from a normal distribution X = N(mean, sigma).

3.1 Bayesian Regression

So far we only looked at Bayesian classification. You can also do Bayesian regression. Take linear regression and just put it in the Bayesian model. You base the predictions on probability instead of a single point. You can optimise these with gradient decent even still. There are multiple best values. You are more after the posterior distribution for the model parameters.

> **Linear regression reminder:**
>
> y(x) = **W**^T**x** (frequentst view)
>
> Cost (sum of squares): y(W) = 0/2N*Sum(y(xi)-yi)
>
> Predicted - what you found

With bayesian linear regression you formulate linear regression using probability distributions rather than point estimates. This assumes that y was drawn from a probability distribution. The sklearn linear regression can actually use Bayesian regression for regression as well as it is build in.

- Aim of Bayesian Linear Regression
  - not to find the single “best” value of the model parameters,
  - but rather to determine the posterior distribution for the model parameters.

$$p(\mathbf{w}^T|y,\mathbf{x}) = \frac{p(y|\mathbf{w}^T,\mathbf{x})p(\mathbf{w}^T|\mathbf{x})}{p(y|,\mathbf{x})}$$

Figure 3: Bayesian Regression

So this is just linear regression in Bayesian pretty cool if you ask me its like a whole another way of looking at things.

3.1.1 Different scikit learn bayes models

There are different implementations of native bayes. The one you want to use depends on the type of your data.

- Gaussian Naive Bayes classifier (GaussianNB())
  - Assumes that features follow a normal distribution
- Multinomial Naive Bayes
  - Assumes count data
  - Each feature represents how often something happens. Like how often does a word appear in a sentence.

- Bernoulli Naive Bayes
  - Assumes your feature vectors are binary (Can only take 2 values)
  - Can also be continues values which are precisely split. Like Below 10 is 0 and above 10 is 1.

3.2 Advantages and Disadvantages of Naive Bayes.

- Advantages:
  - Simple
  - Works well with a small amount of training data
  - The class with the highest probability is considered as the most likely class
  - You get a probability of all the classes
- Disadvantages:
  - You have to estimate parameters of the normal distribution
  - You data needs to be normally distributed
    You can use different ones for different types of data.

4 Decision Trees

With this technique the idea is build a large logic tree made out of questions about the input data. Each question is a node in the tree. The answer to a question decides which node you should go to next. For example: Is the feature larger than 32? If yes go left if no go right. Going left or right is called **branching** or **traversing** the tree. A node only askes **one question about one feature**. At the end of the tree there are no question anymore and just your label. This node is also called a **leaf**.

This technique is like having a lot of if/elif/else statements. You get the label by traversing the tree one node at the time by answering boolean questions about a feature in you data. A nice advantage of this is that decision trees are not at all a black box as you basically have the if statements checking your inputs after you made the model. This makes it easier to share the model.

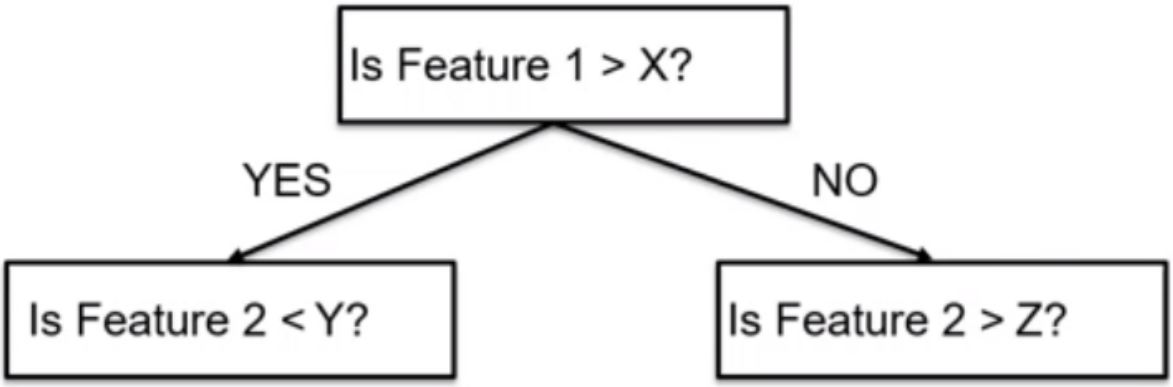


Figure 4: Left or right

> The tree is made out of binary questions about your data, so you don't have to do anything to your data to use this technique. So you don't have to convert categorical data to something numerical or anything like that as the tree can just directly ask something about a categorical feature. For instance, you can just ask: is the color red? Yes or No. The same thing for numerical values.
>
> This is really great as it reduces the preprocessing, and it is intuitive.

Decision trees are a bit like playing guess who:

Figure 5: Guess who

If you go left you might get to a different question as when you would have gone right. Everytime you branch the tree the **depth** of the tree grows. You want the least depth while separating the data as much as possible. A

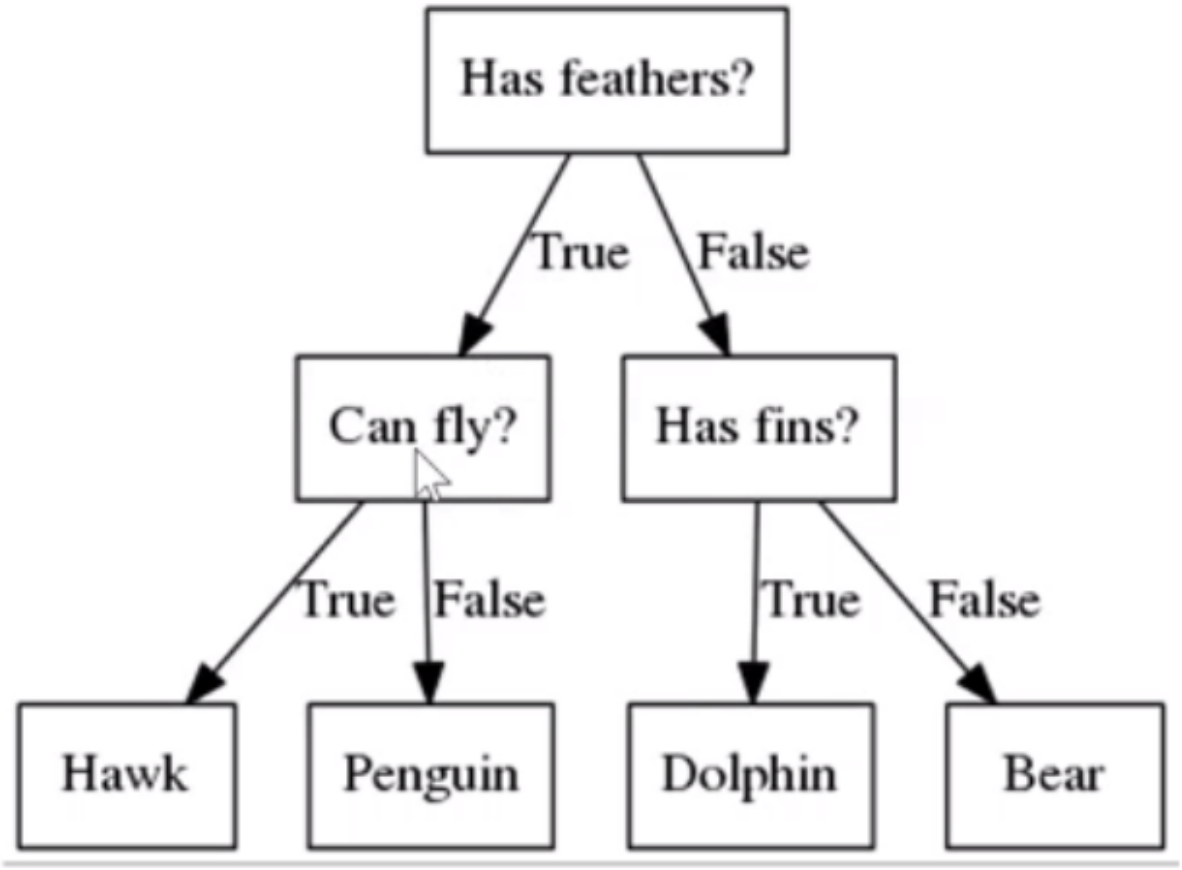


Figure 6: More complicated tree

> All decision boundaries are perpendicular to the feature axes, because at each node a decision is made about one feature only. So, if you see a decision boundary with only strait lines, it is probably a decision tree.

The goal behind decision trees is to get the best branching. You get the best branching based on the order you check the features, and the thresholds you check for.

You want to split as much “area” as possible:

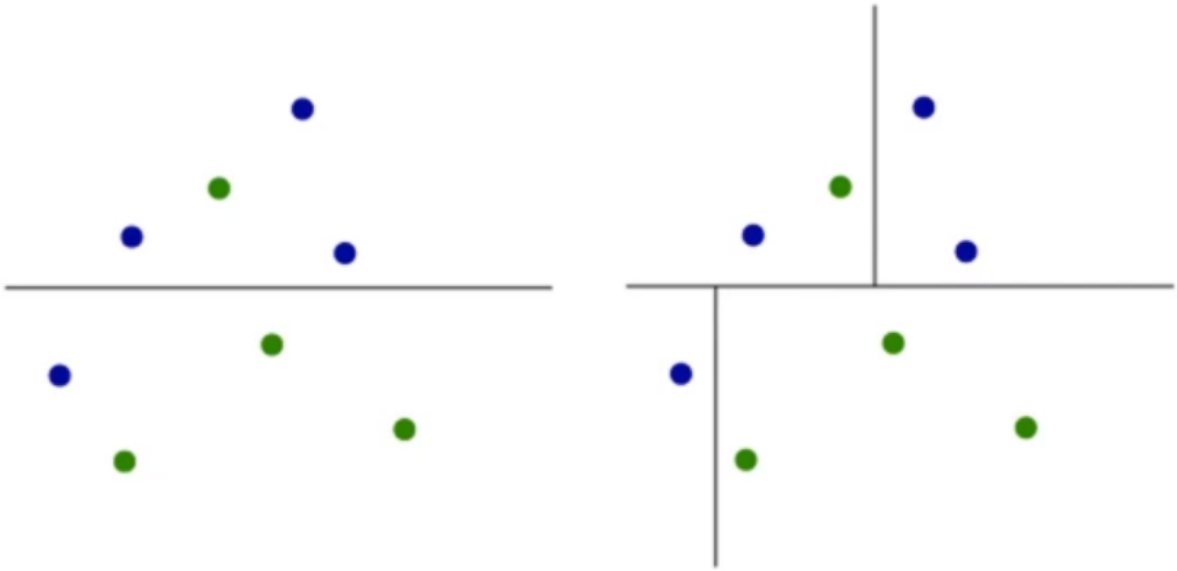
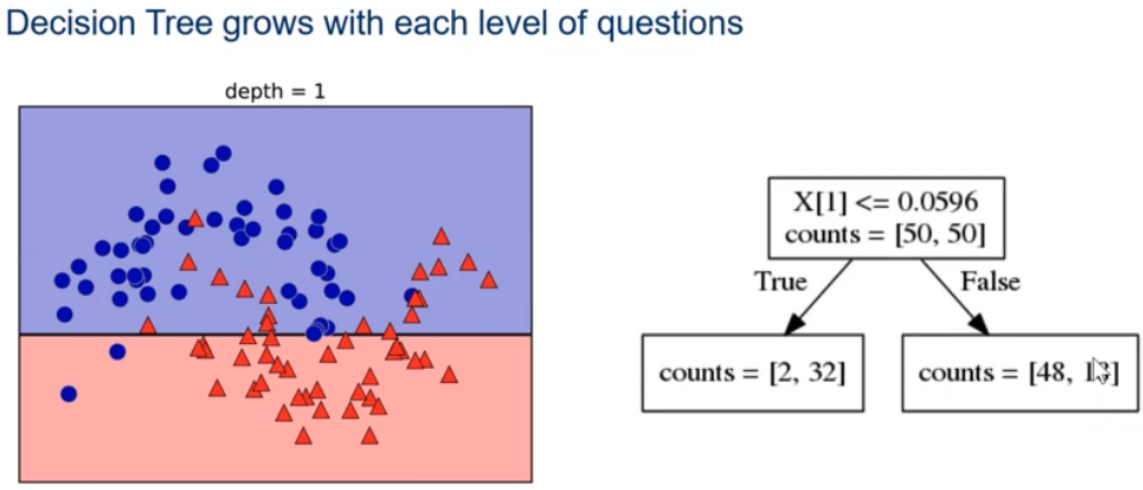
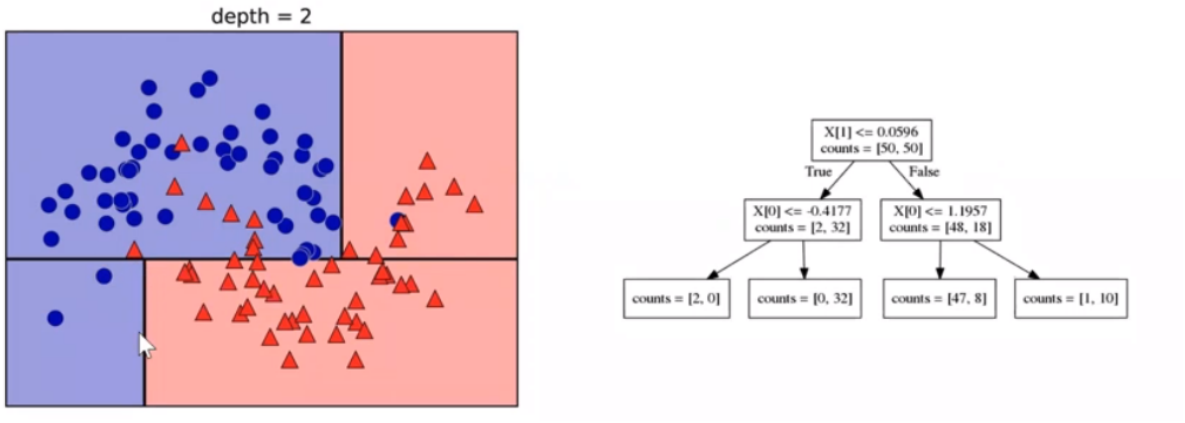
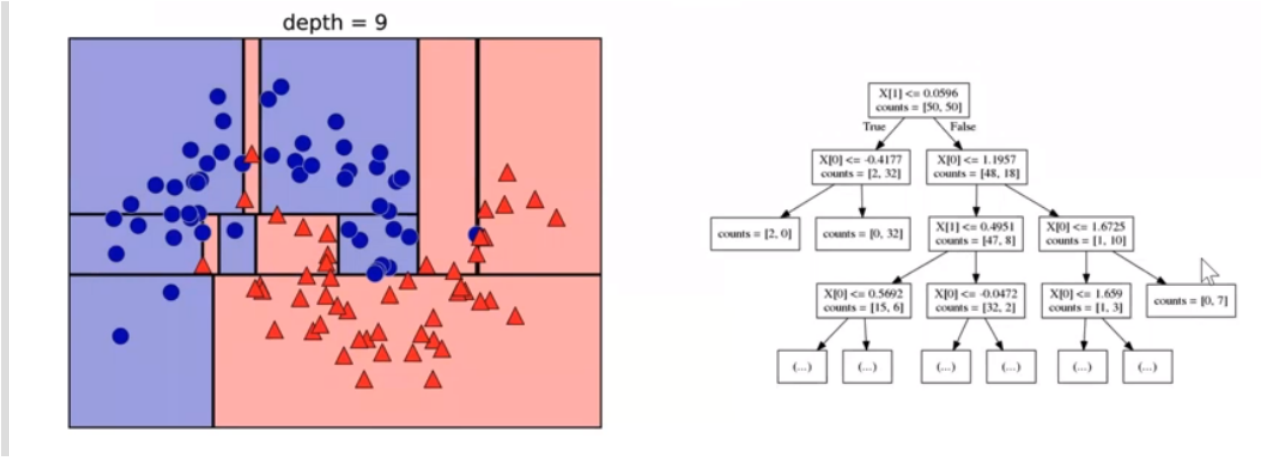


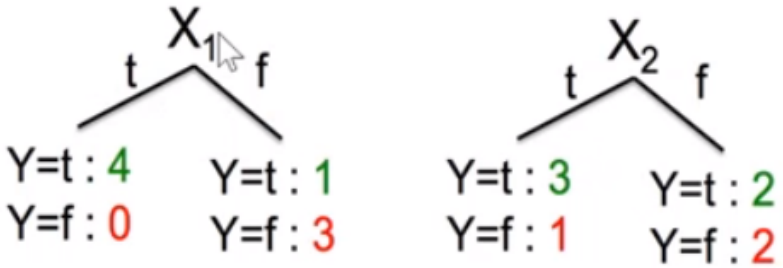
Figure 7: Each test/node layer in the tree splits your data further. Left is dept 1, Right is depth 2

> Every depth increases the amount of decision boundary lines increases with depth as well if that makes sense. This is because every depth down creates exponentially more paths. You are making your decision boundary and eliminating labels as you go
>
> 
>
>
> along
>
> 
>



4.0.0.1 Example

In this case what is better X1 or X2?



X_1	X_2	Y
T	T	T
T	F	T
T	T	T
T	F	T
F	T	T
F	F	F
F	T	F
F	F	F

Figure 8: X1 or X2

If you split by asking is X1 true of false, and it is true then you immediately get the good y. This is the most error reduction with the least depth. In that case there are still 0 remaining wrong classified outcomes anymore. Thus splitting by X1 first is better. This will then also be indicated by those three measures.

4.1 Choosing how to split your tree

If you have a tree they using it is easy to use but getting the splits is the tricky part. The only way really is just to try a bunch of values/splits and look at some measures to give indication about the improvement the split gives. Then you pick the split that gives the most improvement. For this we use the decision tree algorithm. This algorithm has to decide:

- What features to ask about
- What values to use in the question (Numbers for continues values, categories for categorical values)
- What order to do the splitting in
- Decide what the outputs are if you get to the leafs

The algorithm works like this:

1. Start from an empty decision tree
2. Split on the next best **attribute**
3. Repeat

4.2 Decision tree classification

Ok but what is the next best attribute? For classification, you can determine it based on these three things:

- Entropy
- Information gain
- Gini index

4.2.1 Entropy

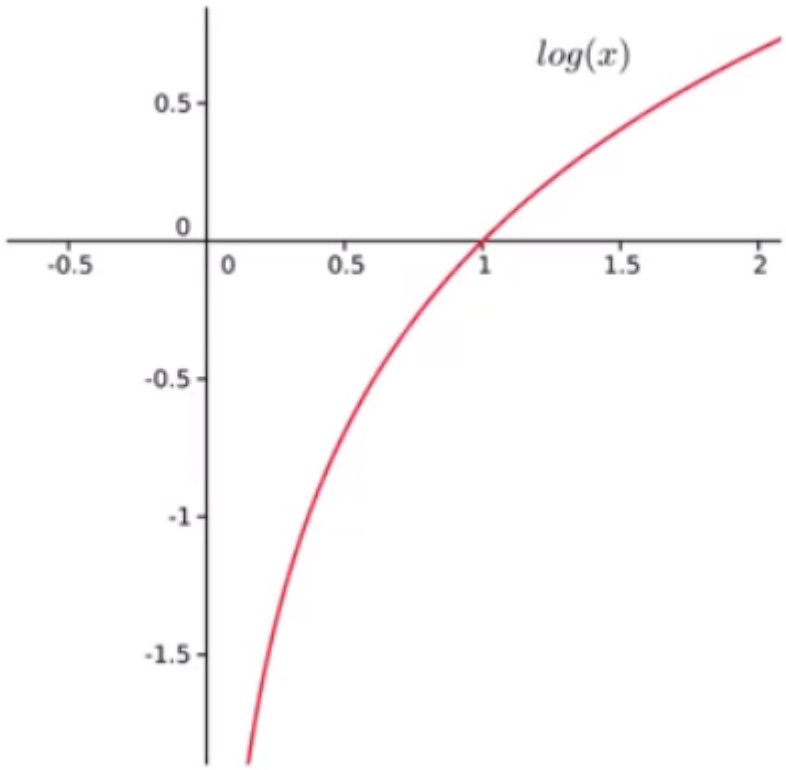
Entropy is **the level of uncertainty**. The higher the entropy the more uncertainty. The formula is:

$$H_{CE}(\mathbf{x}) = \sum_{k \in y} p_k log(p_k)$$

You saw entropy in logistic regression. Entropy is the level of uncertainty. It is the probability of your class occurring giving the log of that probability. That means the chance that any instance is that class. You get that by doing: occurrences feature/n. Entropy can never be negative. Instead of P for Bayes we have H for entropy.

We want to minimize entropy because we want to minimize uncertainty.

> Probabilities are always between 0 and 1 so the log of the p will be negative that is why this works. The lower the p the lower the log results.



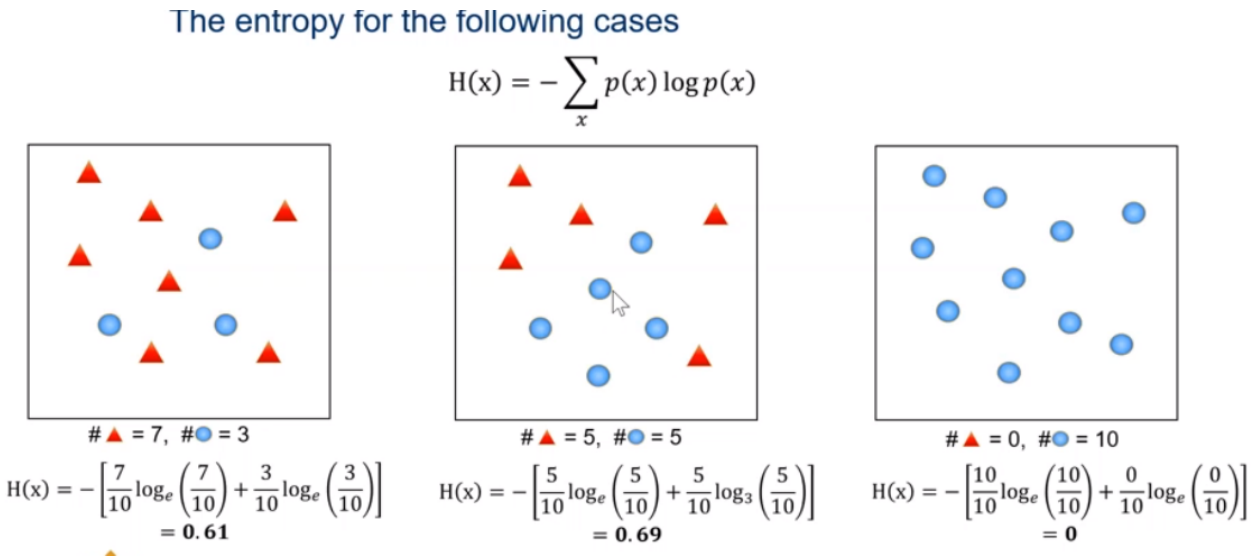


Figure 9: Examples of entropy

4.2.1.1 Examples of entropy In this case the entropy is the lowest on the right. You can also calculate entropy for sequences.

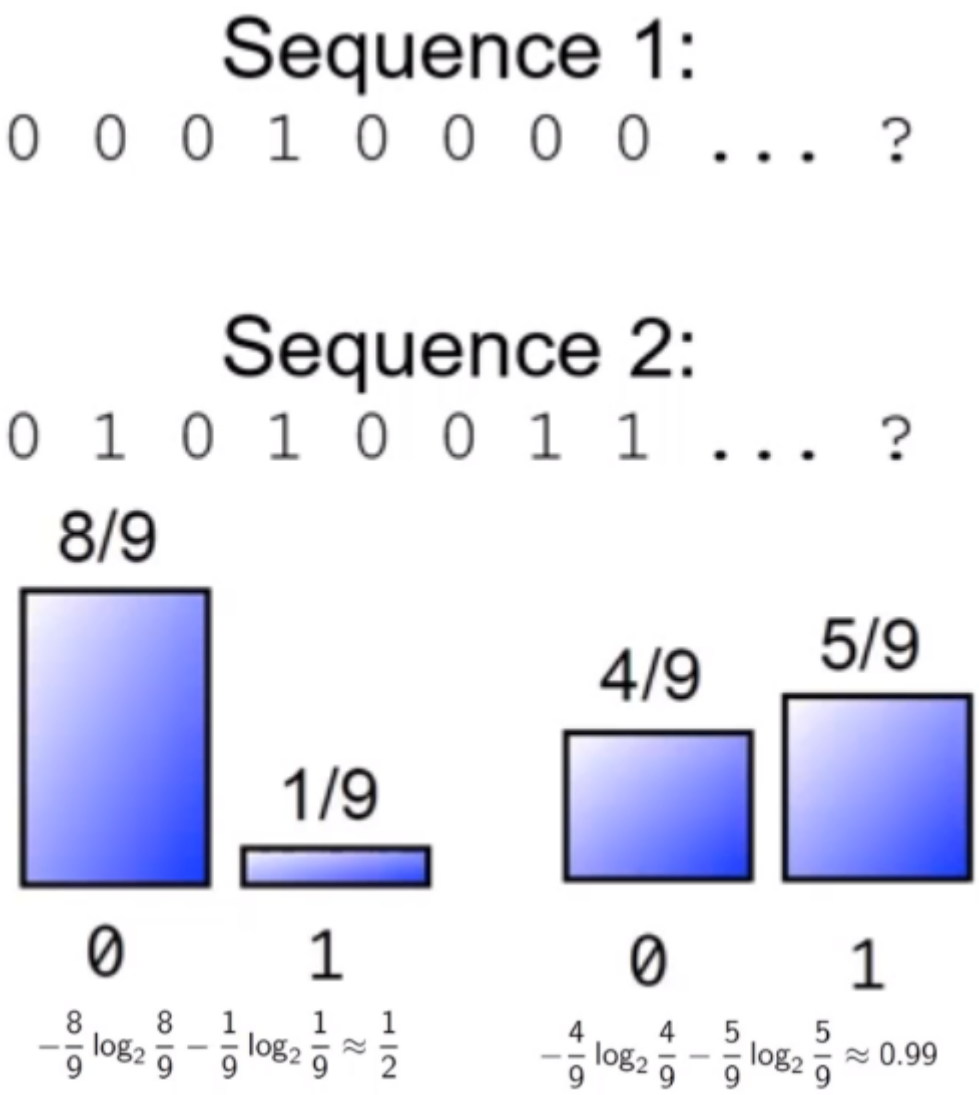


Figure 10: Sequence

Now here are some more things about entropy she discussed.

This shows that if a sequence is more equal (as in there is an equal division of p between classes), then the entropy is higher. This makes entropy the method for dealing with unbalanced data.

The idea of using entropy is to try a split and then to calculate what the entropy is after the split. Take the option that has the **lowest entropy** after the split. This option has the least uncertainty.

4.2.1.2 Joint entropy

To find the entropy of a joint distribution you can use this:

	Cloudy	Not Cloudy
Raining	24/100	1/100
Not Raining	25/100	50/100

$$
\begin{aligned}
H(X,Y) &= -\sum_{x\in X}\sum_{y\in Y} p(x,y)\log_2 p(x,y) \\
&= -\frac{24}{100}\log_2\frac{24}{100} - \frac{1}{100}\log_2\frac{1}{100} - \frac{25}{100}\log_2\frac{25}{100} - \frac{50}{100}\log_2\frac{50}{100} \\
&\approx 1.56\text{bits}
\end{aligned}
$$

Figure 11: Joint distribution

She didn't say anything more about it either. So just use this formula.

4.2.1.3 Conditional entropy

Again we use the joint thing + chain rule to get the conditional rule. Conditional entropy: `H(X, Y)= H(X|Y)+H(Y)= H(Y|X)+ H(X)` If you try to calculate the entropy of a conditional distribution you do it like this.

Example: X = {Raining, Not raining}, Y = {Cloudy, Not cloudy}

	Cloudy	Not Cloudy
Raining	24/100	1/100
Not Raining	25/100	50/100

What is the entropy of cloudiness Y, given that it is raining?

$$
\begin{aligned}
H(Y|X=x) &= -\sum_{y\in Y} p(y|x)\log_2 p(y|x) \\
&= -\frac{24}{25}\log_2\frac{24}{25} - \frac{1}{25}\log_2\frac{1}{25} \\
&\approx 0.24\text{bits}
\end{aligned}
$$

Figure 12: Joint distribution

This is asking what is the entropy of X given Y. So instead of likeliness we are after entropy.

You can just calculate it with the formula.

4.2.1.4 Dependent vs independent

If X and Y are independent, then X doesn't tell us anything about Y. So `H(Y|X)= H(Y)`. But of course Y tells us everything about Y. `H(y|y)= 0`. By knowing X we can decrease uncertainty about `Y: H(Y|X)<= H(Y)`

4.2.2 Information gain

With information gain we look for how much information we gain about the features. The formula is `IG(Y|X)= H(Y)- H(Y|X)`

`IG(Y|X)` is pronounced as information gain in Y due to X.

If X is completely informative about Y then `IG(Y|X)= 0` If X is completely uninformative about Y then `IG(Y|X)= H(Y)`

E.G., everything that is uncertain is gone This is what you use in making the tree. **You want to find the split with the highest information gain.**

> You can see information gain as taking away entropy

4.2.3 Gini Coefficient

$$H_{gini}(\mathbf{x}) = \sum_{k \in y} p_k(1 - p_k)$$

Figure 13: Gini index

The gini function is called HGini. Gini is cheaper to calculate then entropy because there is entropy has a log operation and Gini does not. This is why this is the default in sklearn. However, Gini only works for binary classification. The idea is the same. Split, fill in the formula, use the split with the lowest gini.

$$HGini_{salty} = \frac{3}{10}\left[\left(\frac{1}{3}\right)\left(1-\frac{1}{3}\right)+\left(\frac{2}{3}\right)\left(1-\frac{2}{3}\right)\right] = 0.133$$

$$HGini_{spicy} = \frac{5}{10}\left[\left(\frac{3}{5}\right)\left(1-\frac{3}{5}\right)+\left(\frac{2}{5}\right)\left(1-\frac{2}{5}\right)\right] = 0.24$$

$$HGini_{sweet} = \frac{2}{10}\left[\left(\frac{2}{2}\right)\left(1-\frac{2}{2}\right)+\left(\frac{2}{2}\right)\left(1-\frac{2}{2}\right)\right] = 0$$

$$HGini_{Taste} = 0.373$$

$$HGini_{Hot} = \frac{6}{10}\left[\left(\frac{4}{6}\right)\left(1-\frac{4}{6}\right)+\left(\frac{2}{6}\right)\left(1-\frac{2}{6}\right)\right] = 0.2666$$

$$HGini_{Cold} = \frac{4}{10}\left[\left(\frac{2}{4}\right)\left(1-\frac{2}{4}\right)+\left(\frac{2}{4}\right)\left(1-\frac{2}{4}\right)\right] = 0.2$$

$$HGini_{Temp} = 0.466$$

$$HGini_{Soft} = \frac{6}{10}\left[\left(\frac{3}{6}\right)\left(1-\frac{3}{6}\right)+\left(\frac{3}{6}\right)\left(1-\frac{3}{6}\right)\right] = 0.3$$

$$HGini_{Hard} = \frac{4}{10}\left[\left(\frac{1}{4}\right)\left(1-\frac{1}{4}\right)+\left(\frac{3}{4}\right)\left(1-\frac{3}{4}\right)\right] = 0.15$$

$$HGini_{Texture} = 0.45$$

Which attribute to split?
The attribute with the **lowest** Gini index

Figure 14: Gini example

4.2.3.1 Example:

4.3 Thresholds with continues values

For all of these methods if you have continues values you not only have to try splits but also with different thresholds. Like if X between 10 and 12 or 10 and 13 what gives the best model improvement?

4.4 Decision Tree Regression

For regression with decision trees you use the **weighted mean square error** to decide on the splits. Try a lot of splits and choose the split that reduces the weighted mean square error the most.

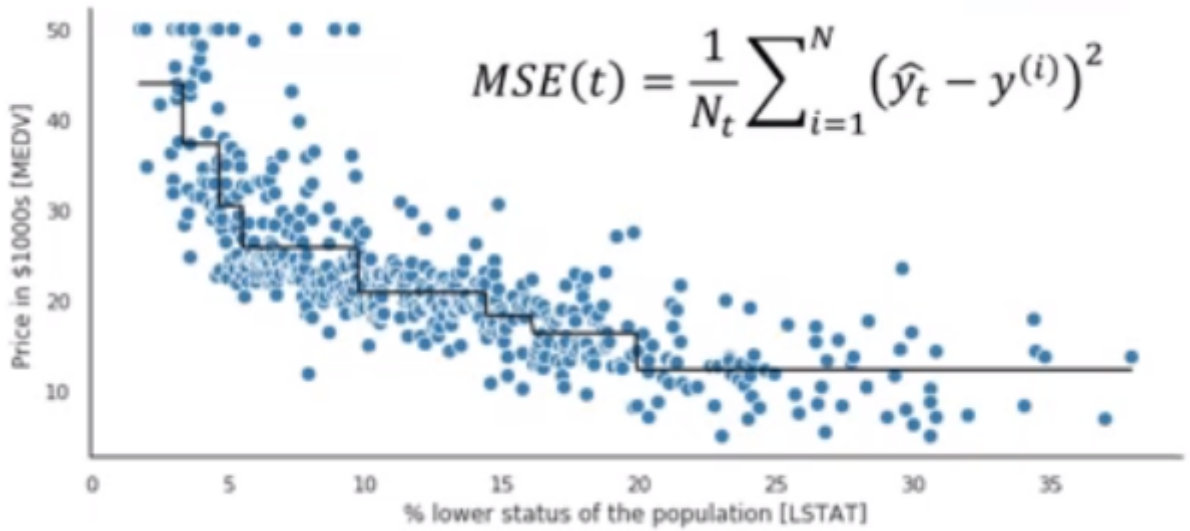


Figure 15: Decision tree regression

N is the number of training samples at a certain node. y is the true target value y^ is the predicted sample mean.

As you can see just like with decision tree classification, you again get these straight lines in the decision boundary.

4.5 Complexity of the model

The tree can get huge quickly. The complexity of a decision tree model is determined by the depth of the tree. **Increasing the depth** of the tree increases the number of decision boundaries and **may lead to overfitting**. For example all these places might have overfitting:

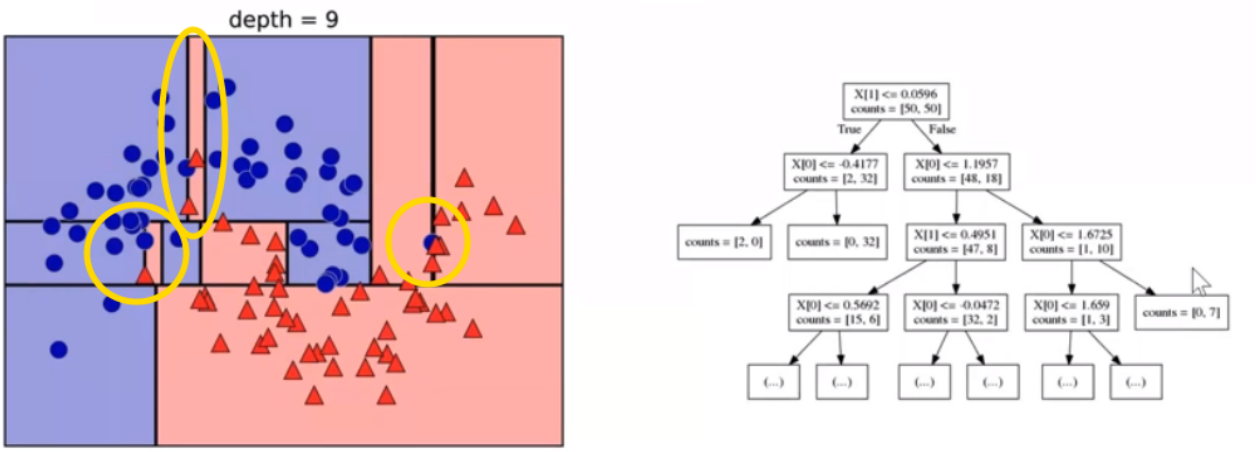


Figure 16: Overfitting

4.5.1 Ways of reducing model complexity. (hyperparameters)

There are hyper parameters that limit model complexity. You should set at least one of these as theoretically you can keep growing the tree forever.

- max_depth = The max depth the tree can grow
- max_leaf_nodes = The maximum leaf nodes that can exist
- min_samples_split = A minimum amount of samples that have to be in a split to make a split.
  There are also more parameters

4.6 Advantages and Disadvantages of decision trees

Advantages

- Easy to interpret and make for straightforward visualizations
- The internal workings are capable of being observed and thus makes it possible to easily reproduce work
- Can handle both numerical and categorical data directly without preprocessing

- Performs well on large datasets
- Performs fast in general

Disadvantages

- Building decision trees requires algorithms that can find the optimal choice at each node
- Prone to overfitting, especially when the trees' depth increases

5 Bias and variance

Before we move on to ensemble learning, lets have a reflection moment. So far these algorithms were covered:

Classification	Regression
Logistic Regression	Linear Regression
Linear SVMs	Linear SVM
KNN	KNN regression
Neural networks	Polynomial Regression
Kernel SVM	Decision Trees Regression
Naive Bayes	Kernel SVM Regression
Decision trees	Bayesian Linear Regression

Some of these are linear, and some of these are not. Linear algorithms create a straight decision boundary line. Non-linear algorithms don't.

These algorithms are known as **weak learners** because they might be sensitive to overfitting. You can overcome this with regularization as we have seen but another way is with ensemble learning.

The problem that we always have with these models is figuring out how well they will work for unseen data. This problem is called **Estimating Generalization Error**. You can calculate this with: Generalization error = bias^2 + variance + noise

Error due to Bias: Bias measures how far off in on average these models' predictions are from the correct value.

Error due to Variance: The variance is how much the predictions for a given point vary between different realizations of the model.

Noise: The irreducible error. This is error that you can't do anything about. Noise in the data collection process for example.

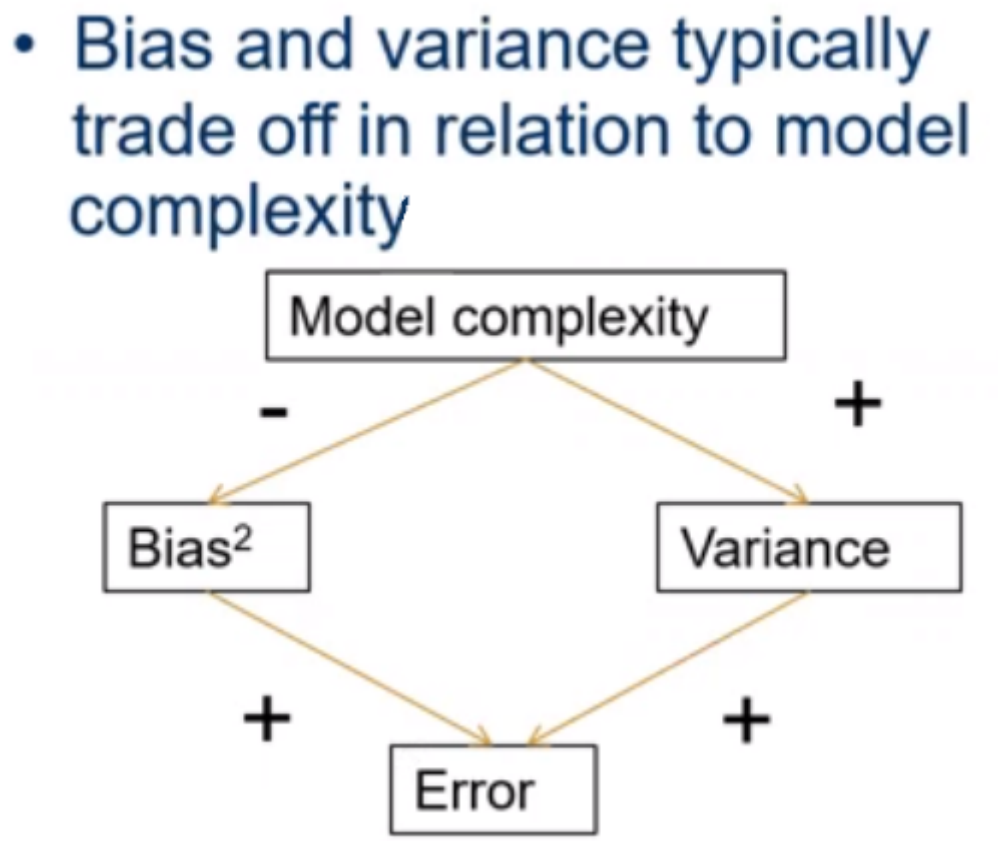


Figure 17: Bias and variance typically trade off in relation to model complexity

Both variance and bias are related to model complexity. If you make your model **less complex** typically you get **less bais but more variance**. If you make your model **more complex** you get **more bais and less variance**. They both contribute to the error, so you want both to be low as possible. Which means you want to find a model complexity that gives a low bais and low variance. It is about finding a balance.

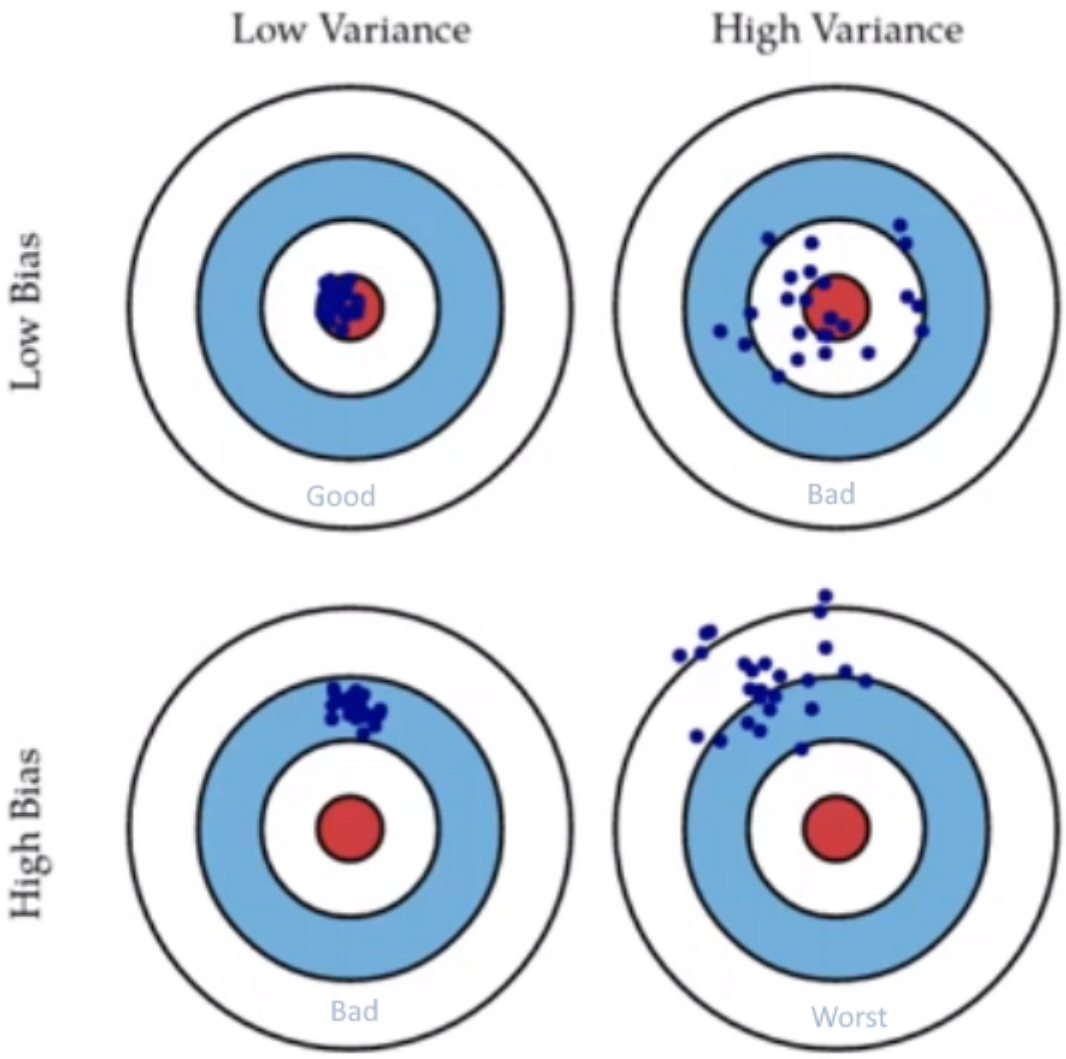


Figure 18: Depiction of low and high variance and bias in dartboards

This was picture made by scott fortmann roe. He has a nice further explanation about the bias and variance tradeoff.

A low bias, and a low variance are the two most fundamental features expected for a model.

Here are more charts that show the effect even more! Made by Duda and Hart.

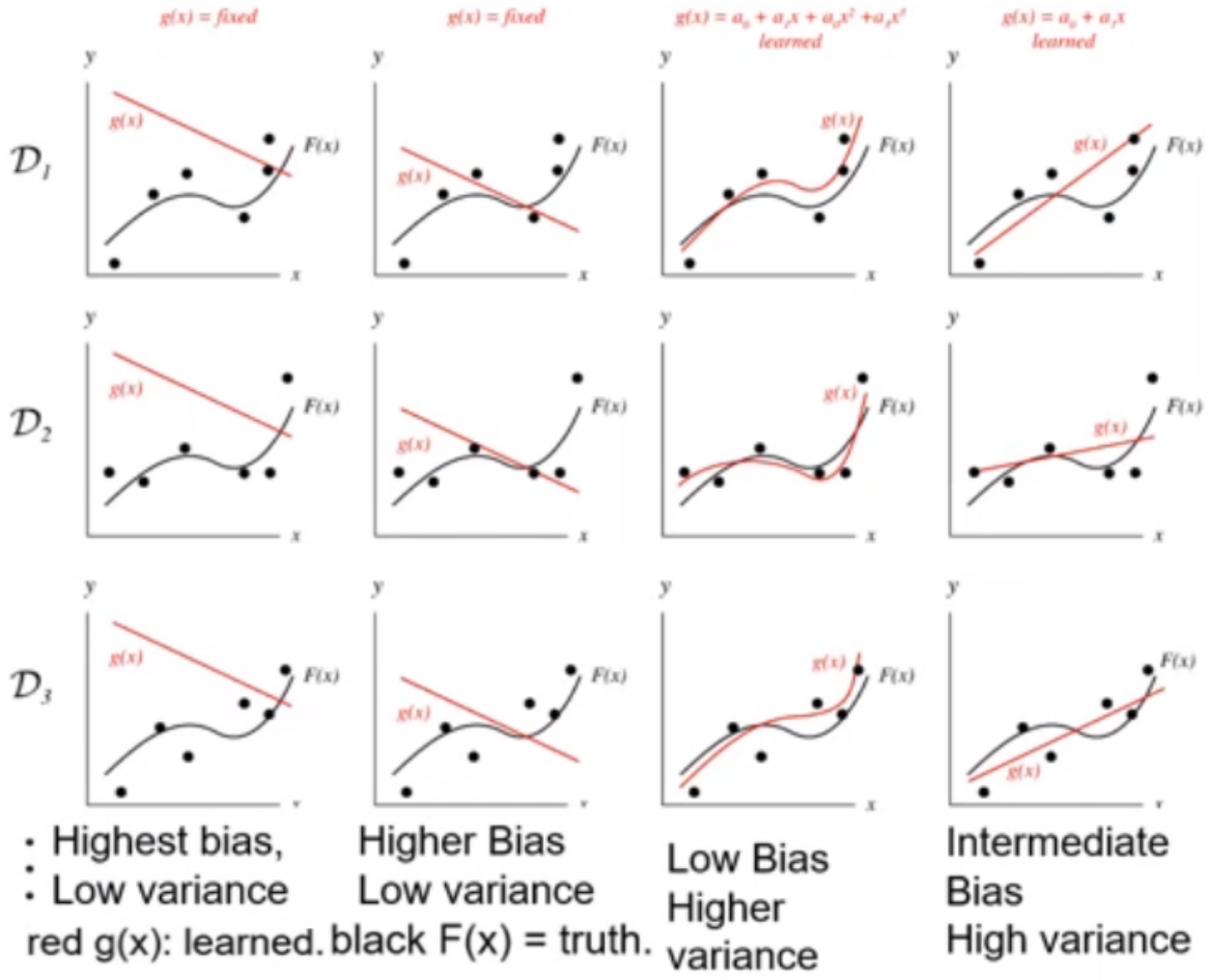


Figure 19: Variance and bias even more graphs

The bais and variance are the cause of the underfitting overfitting problem. Because of it you normally expect model performance to behave like this:

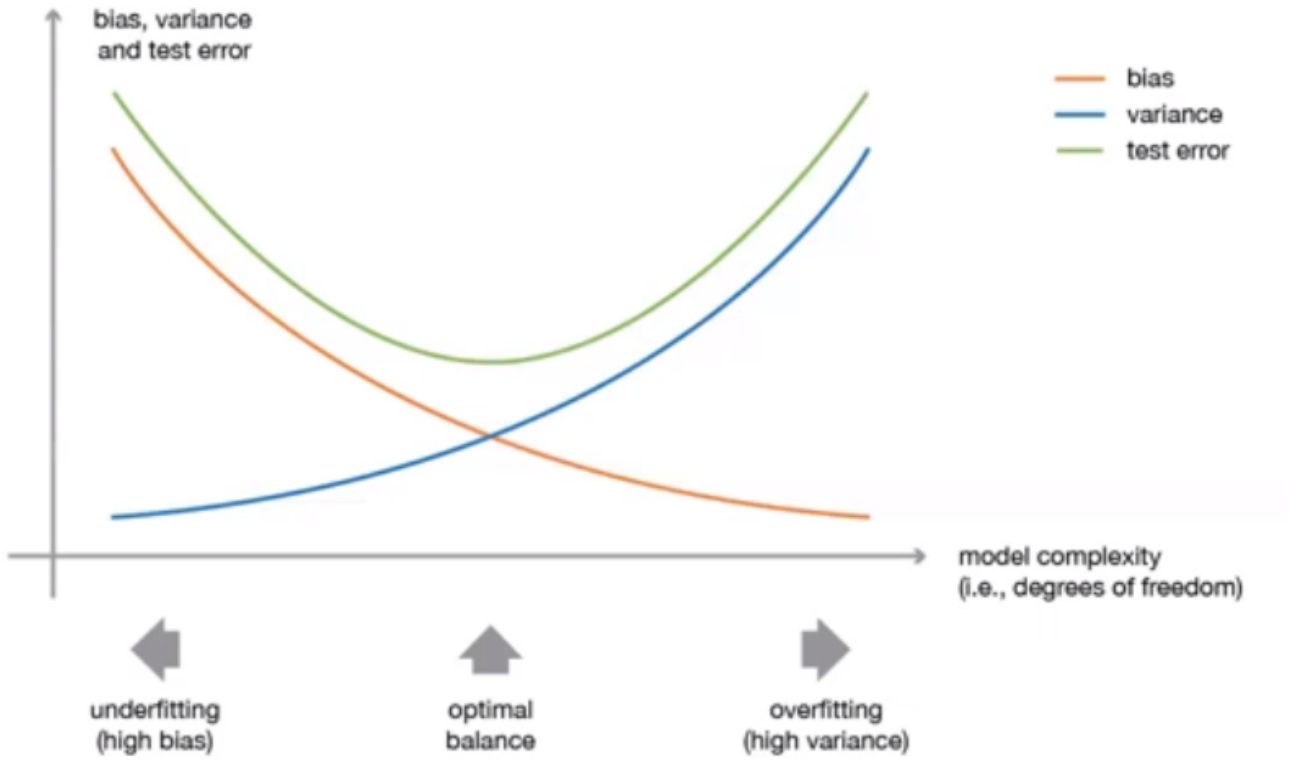


Figure 20: Bais and variance trade off plotted

It seems like the models so far have to deal with this problem. Especially logistic regression, naive bayes, knn, shallow decision trees, linear svm and kernel svm. Some of these have high bias → **low degree of freedom models**. Or they have too much variance to be robust → **high degree of freedom models**. These models do not necessarily perform well by themselves. But who says you can only have 1 model? I don't.

6 Ensemble learning

Ensemble methods try to reduce bias and or variance of weak (single) models by combining several of them together to achieve better performance. The ways of doing this are called **Voting**, **Bagging**, **Boosting** and **Stacking**.

6.1 Voting

Voting is a method were you make multiple models vote on what the output should be. This makes the most sense for classifiers. You just train multiple models and each model votes on what they predict is the correct label. You pick the label with the most votes. This is called **hard voting**. Because the idea of voting is simple, it works with any model also neural networks. But we can do better than just counting votes.

Some classifiers also know how "sure" they are of their conviction. For instance naive Bayes. You could give the votes of these models a higher weight.

Some classifiers will be the most “sure” about a particular part of the space. You could give these models vote more weight for this space.

If you use weights, you in the end you average the results instead of just counting votes. This is called **soft voting**.

More models take longer to train, but the results you will get are also based on much more. However, **only combine models if they are not correlated**. You can only do averaging instead of counting if all the models output calibrated/scaled (“good”) probabilities.

The sklearn version of this is called VotingClassifier. You give this class a list of other models.

6.2 Bagging (bootstrap aggregation)

Bagging fits several “independent” models and averages their predictions in order to lower the variance. So this technique is for low bais and high variance models.

Fitting fully independent models requires too much data. This is why we need bootstrapping. **Bootstrapping is taking random samples of the same size (B) from the dataset with repetition.** With repetition means that a datapoint can be in multiple samples. Once you have the bootstrapped samples, you can train models on them. These could be all the same or different ones.

Figure 21: Bootstrapping

So **bootstrapping is creating the samples** from the data and **bagging is fitting models on these samples and taking the average**. You basically create a lot of models based on essentially the same data, but you just leave out random data points for every sample. This way each model will have slightly different results. The average of these results will have a lower variance.

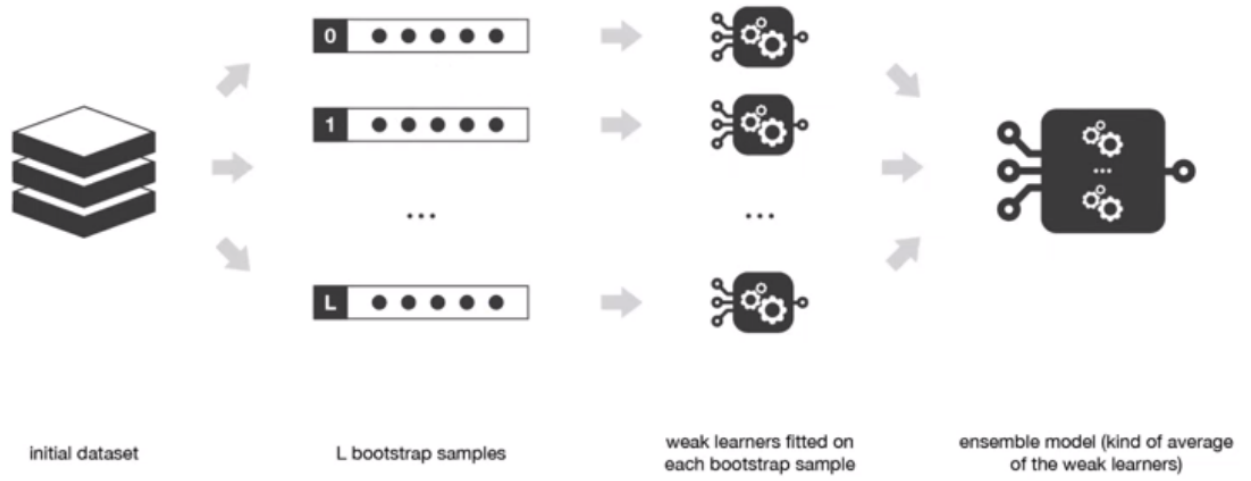
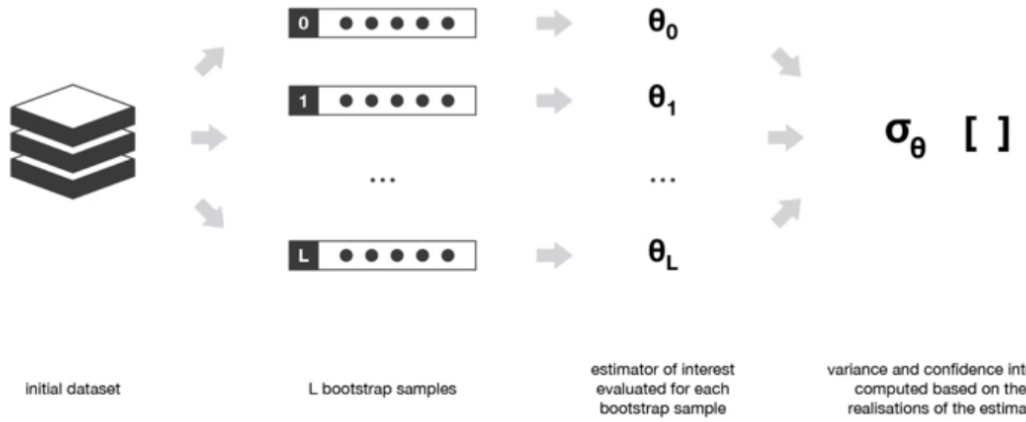


Figure 22: Nice bagging picture

With bagging, you can also **approximate the variance or confidence interval of the estimator** by evaluating the variance that all the bootstrapped samples have. For regression a simple average is used for classification you can use the voting techniques.



By using bootstrapping to generate several bootstrap samples that can be considered as being "almost-representative" and "almost-independent" (almost i.i.d. samples), These bootstrap samples will allow us to approximate the variance of the estimator, by evaluating its value for each of them.

Figure 23: Approximate the variance process

Bagging is implemented in sklearn with BaggingClassifier and BaggingRepressor.

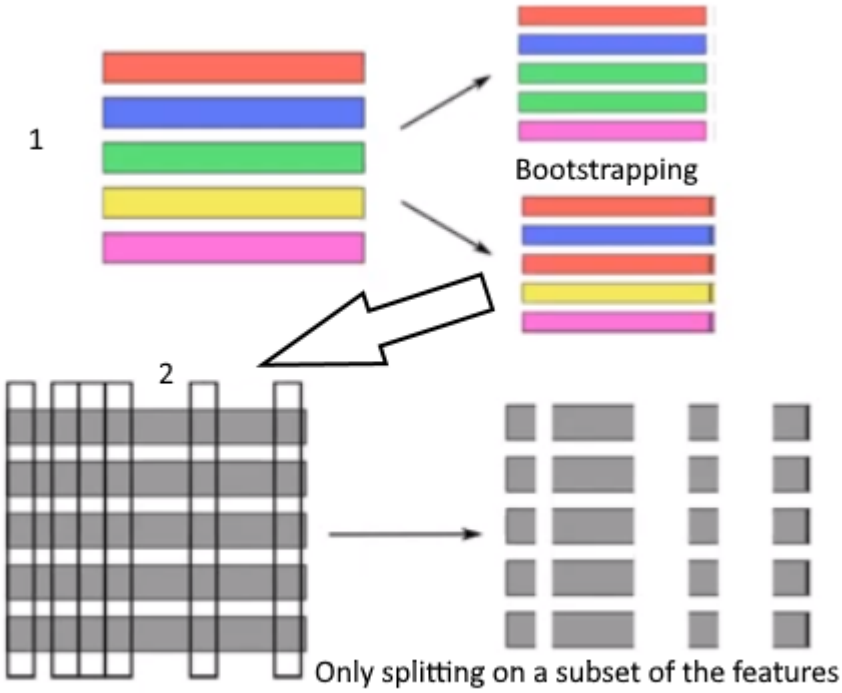
6.2.1 Random forests

Random forests is a bagging method were **deep decision trees**, fitted on bootstrap samples, are combined to produce an output with lower variance. This method is also called a random decision forest.

Decision Trees can be:

- **Shallow**. Shallow trees have low depth, these trees have less variance but higher bias, a better choice for sequential methods & boosting
- **Deep**. Deep trees have low bias and high variance. Better choice for bagging method as that is focused on reducing the variance.

With random forests you do bagging but **you only use (deep) decision trees** to train on the bootstrapped samples. In addition to that the **set of features you base the splitting decisions on are randomly selected**. So the decision trees only use a subset of the available features to make the splits. This is done to **reduce correlation**. More trees are always beter.



> So you do double randomization; each tree picks a bootstrap sample and then also only uses a random sample of the features in the picked bootstrap sample to decide on the splits of the decision tree.
> Here is the full picture of random forest/random decision forests:

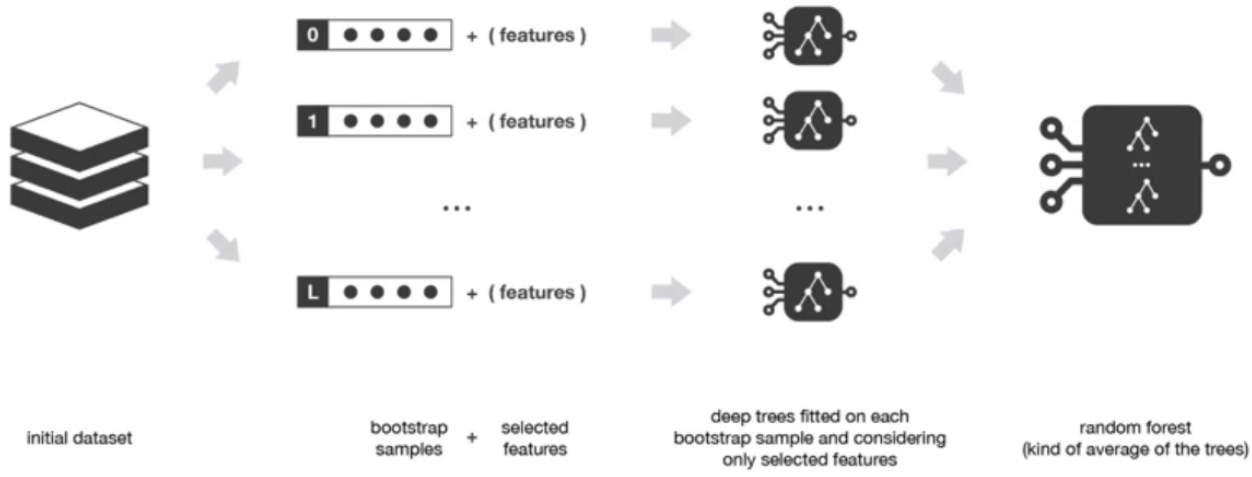


Figure 24: Random forest full picture

Random forest is called a strong learner because it is composed of multiple (weak) trees.

6.2.1.1 Getting results from the forest

How do you combine the results of the trees? You average the tree. The result of the averaging is called a random forest. Like this:

Figure 25: Random forest averaging

In this case we averaged 2 decision tree results. In the areas where the trees don't mach the average is 0 (because there are only 2 trees) which means you don't know the answer. If you have more trees you would have places were you are more and less sure instead. To decide on an output for **classification you take the mode** of the classes that were outputted and for **regression you could take the mean** of the values outputted by the trees.

With sklearn the model is called `ensemble.RandomForestClassifer`. Special hyperparameter that random forests have, are:

- **max_features**: Hopefully it is obvious what that does. The maximum amount of features. The recommendation is to pick `n_features^0.5` for classification and n_features for regression.
- **n_estimators**: This is the amount of trees you want. The more, the better. It is recommended to have at least 100. But the more, the better.

6.3 Boosting

With boosting you fit the weak models in sequence unlike bagging which fits in parallel. You do this, so that a model knows about the results of the previous model. This way a model can give more importance to the observations in the dataset that were badly handled by the previous models in the sequence. This way bias can be reduced. So this technique is for a high bais and low variance models.

This picture shows how you create models sequentially like this:

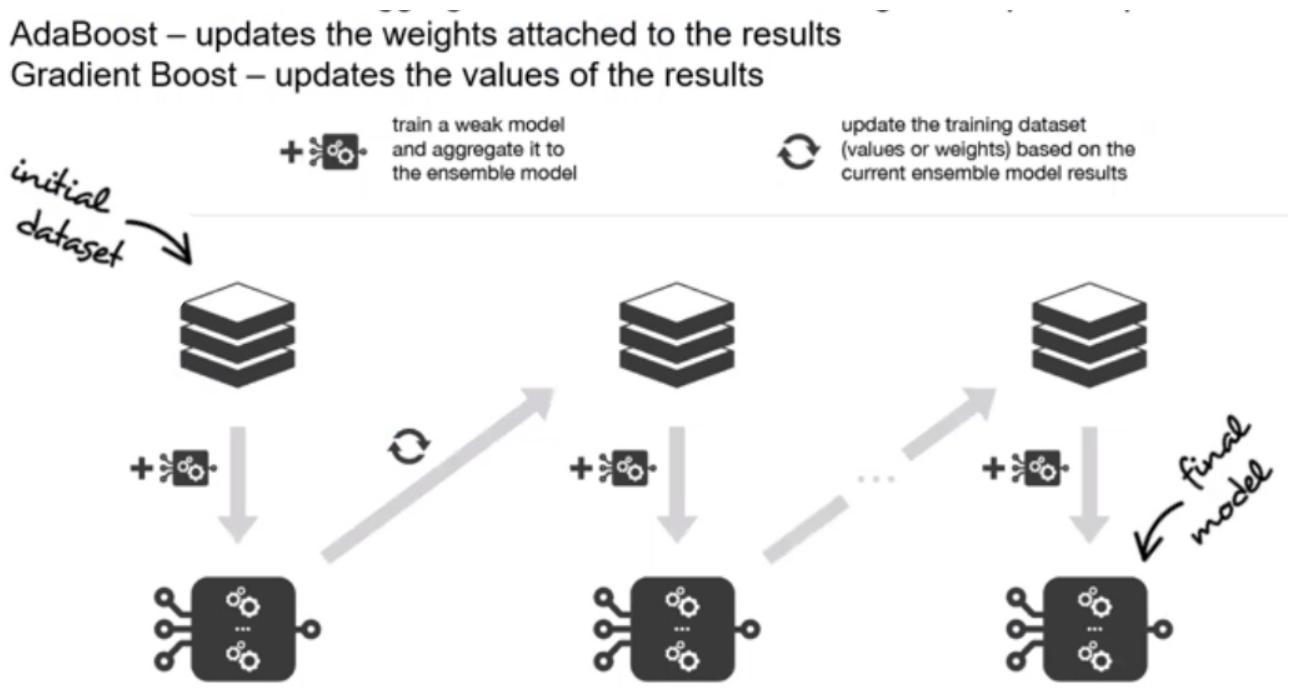


Figure 26: Boosting

The ways discussed creating models like this are **Ada boost** and **Gradient Boost**.

6.3.1 Ada boost

The idea of adaptive boosting is that you run a weak model in the chain. Then find out which points were **wrongly classified** and then give these points a **higher weight** to make them more important for the next model. This way the next model will try to focus on correctly predicting these specific points. This is called **updating the sample weights**.

The weak model itself also gets a weight based on how well it predicted the data. This is called the **update coefficient** or **the amount of say**. Keep doing this until you get through the chain. At the end you merge all the weak models based on their update coefficients and make the prediction.

You can use any weak learning model you want but often a decision tree with depth 1 is used. These are called **stumps**.

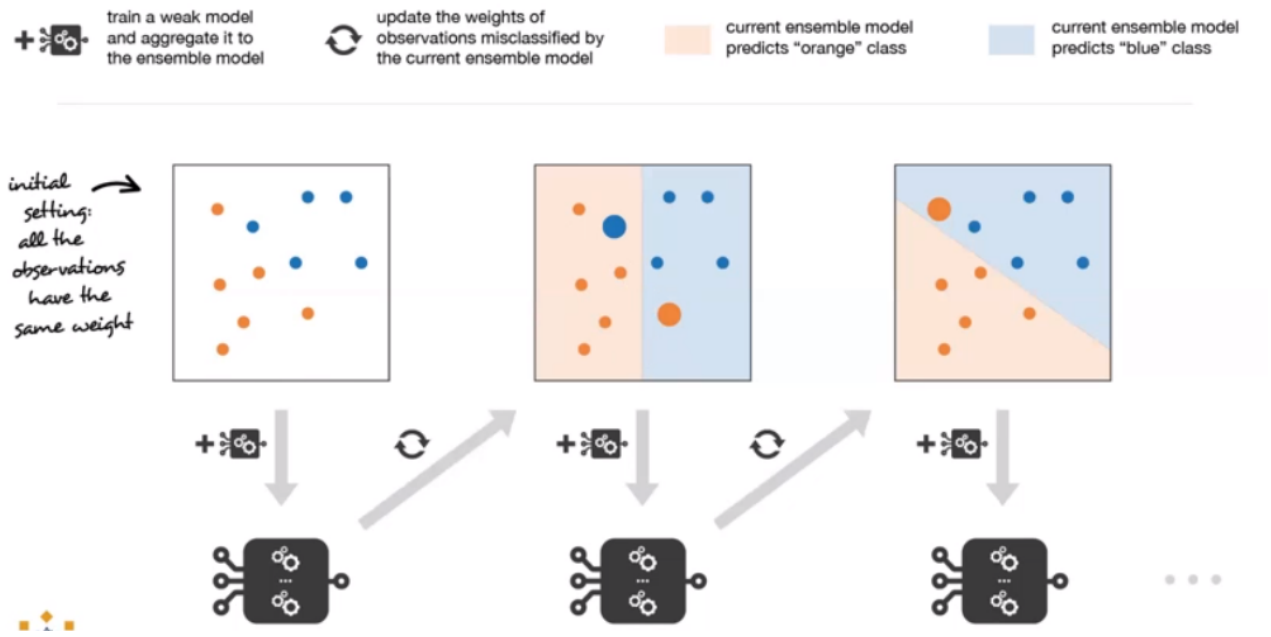


Figure 27: Ada boost

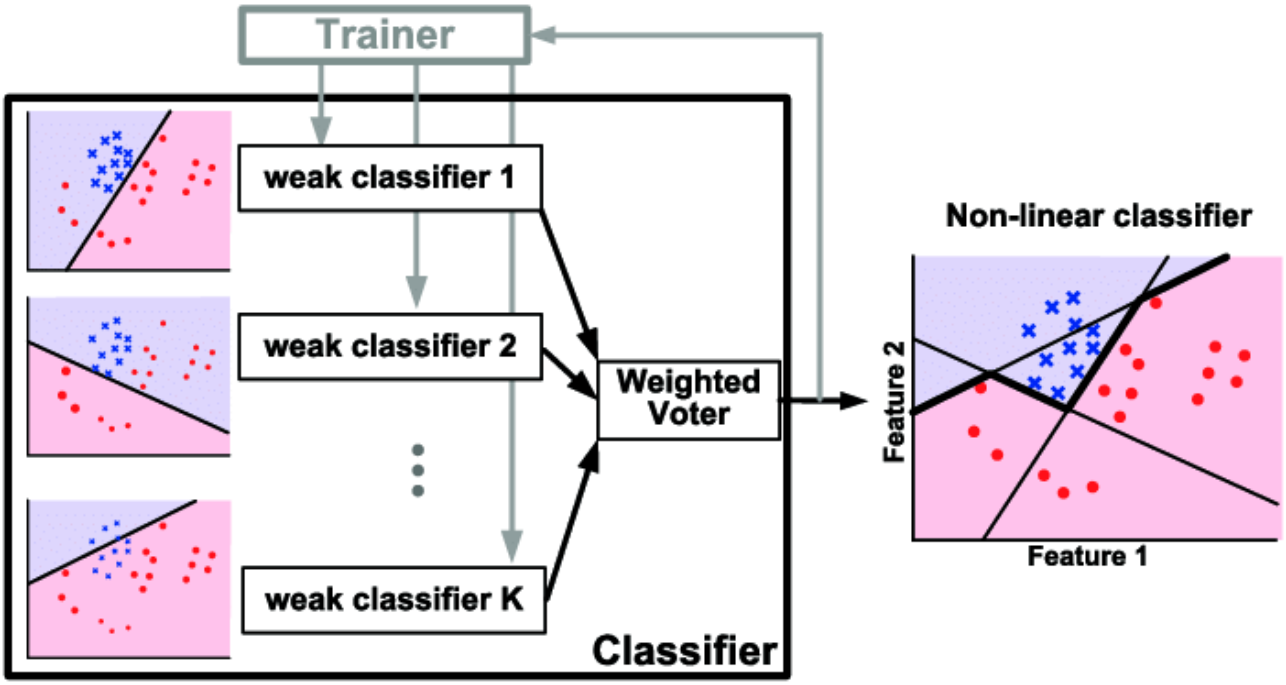


Figure 28: Ada boost

Video going into more depth about ada boosting

6.3.2 Gradient boosting

Gradient boosting starts with a simple prediction it could be the mean, but it is just a guess. This guess will have a certain error/residuals. Instead of making stumps like adaboost, gradient boost can make bigger trees, but you still set a max size. With these bigger trees it tries to predict the pseudo residuals of the data instead of the target.

Then when this tree is made you have to scale the prediction down with a learning rate, so the model has less impact on the final result. Then combine this tree with the original prediction, and train a new model based on the new residuals. You should have moved a bit into the right direction from the original prediction. This will give you new pseudo residuals the next tree can try to predict. This tree is then also added to the chain, and the residuals should keep getting smaller with every tree you add.

The idea is to find out what the best next tree is every time. I don't really get this one :'(

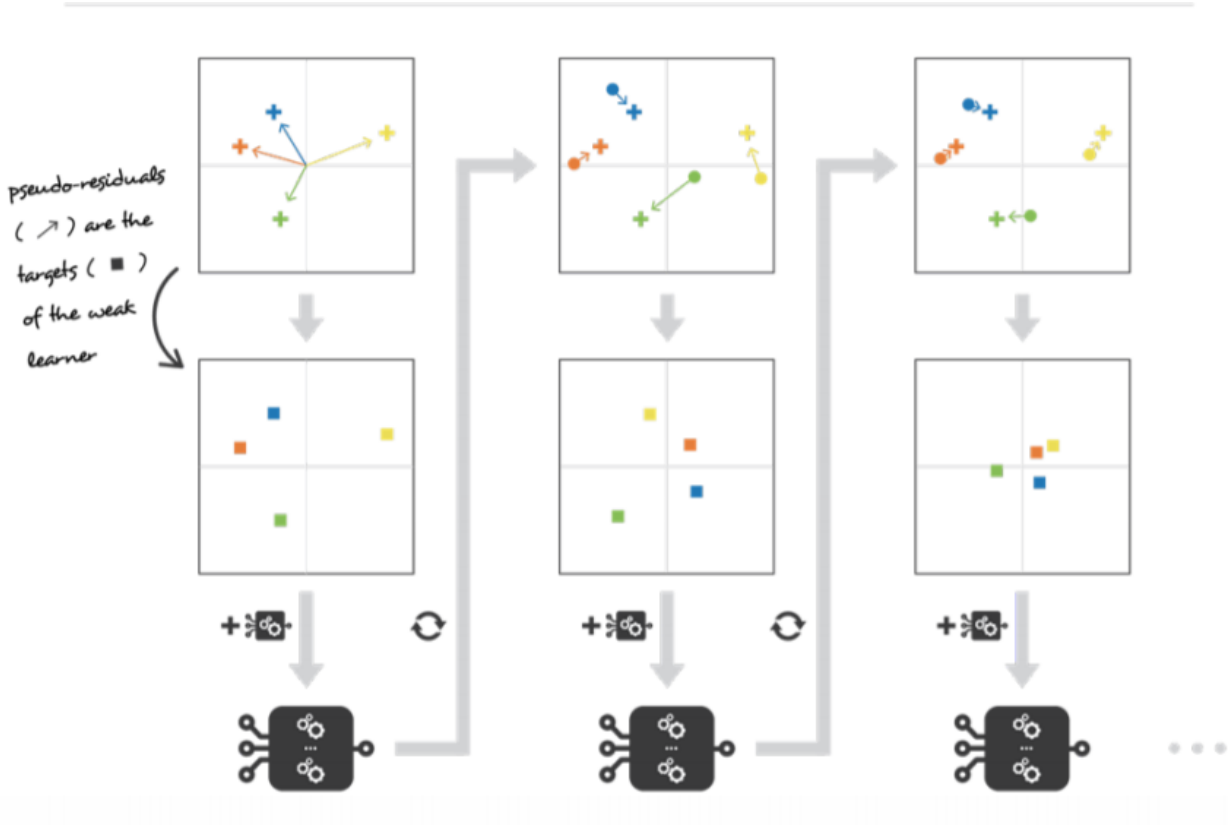


Figure 29: img_5.png

For classification log loss is used and for regression square loss is used.

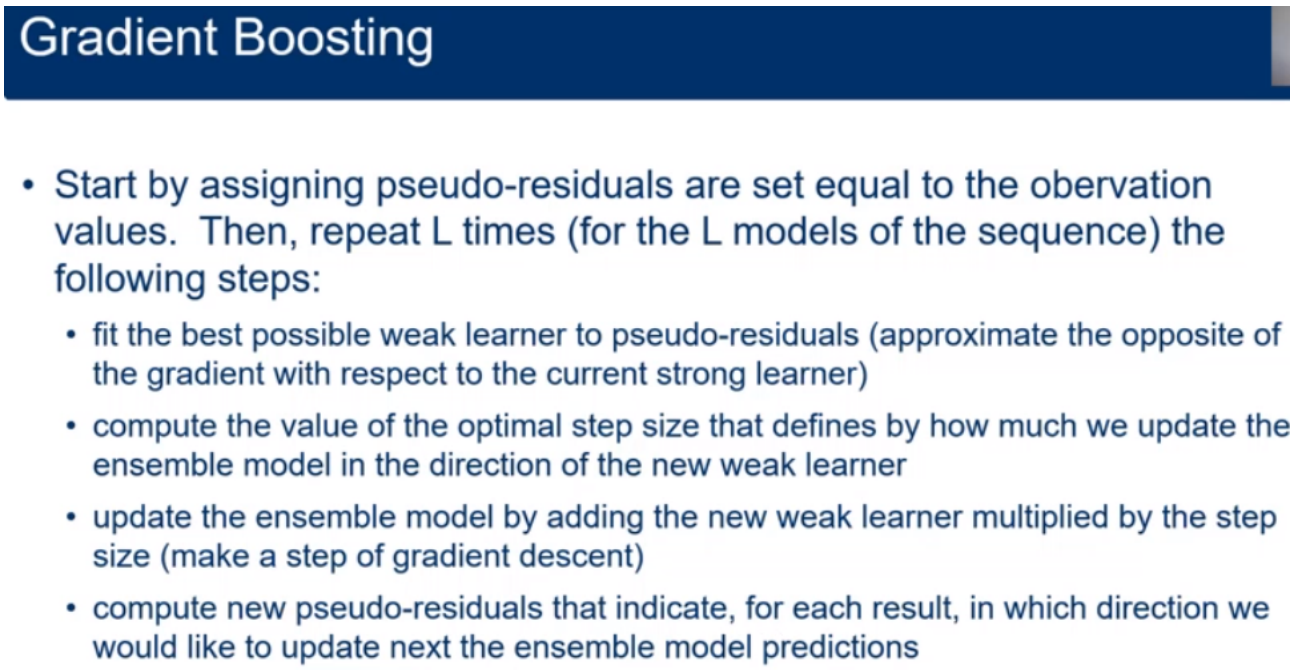


Figure 30: Gradient boosting slide

Video going into more depth about gradient decent its a series

There is also extreme boosting.

6.4 Stacking

Trains many models in parallel and combines them by training on a meta model to output a prediction based on different weak models predictions.

So you choose some weak learners for instance KNN or SVM or a decision tree, and then you also choose something as a meta model. Usually a neural network. Then you use the combined output of the weak learners that were trained in parallel as input to the strong learner.

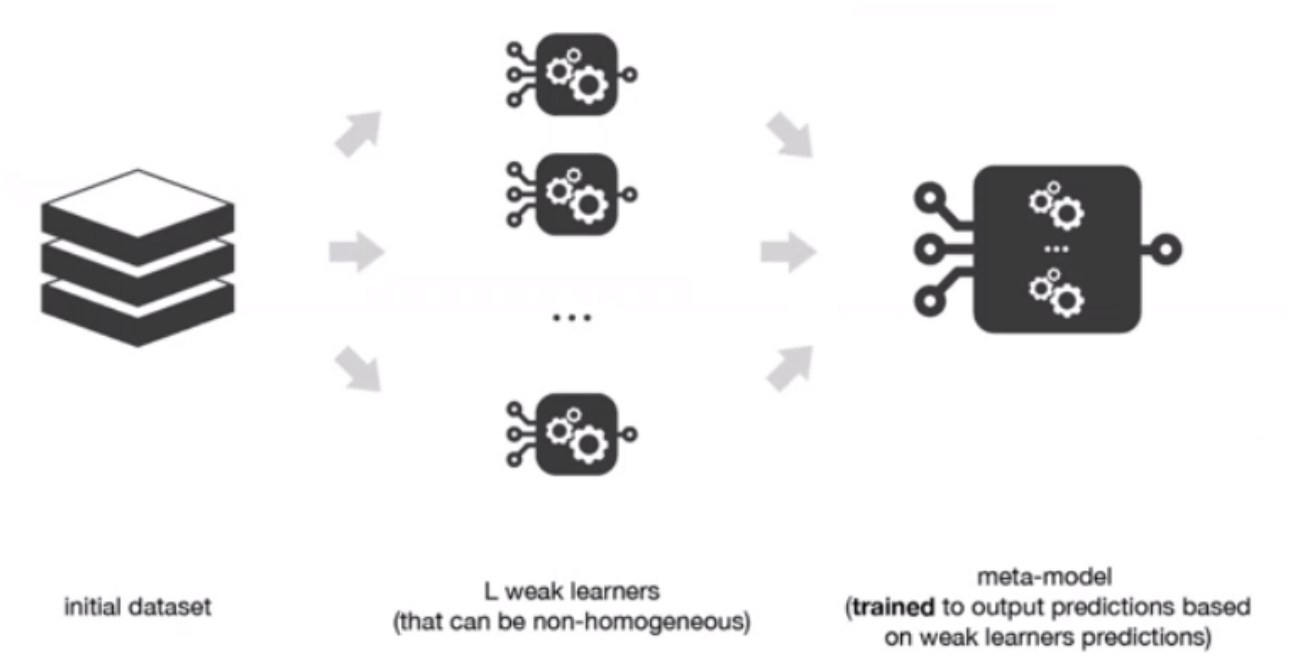
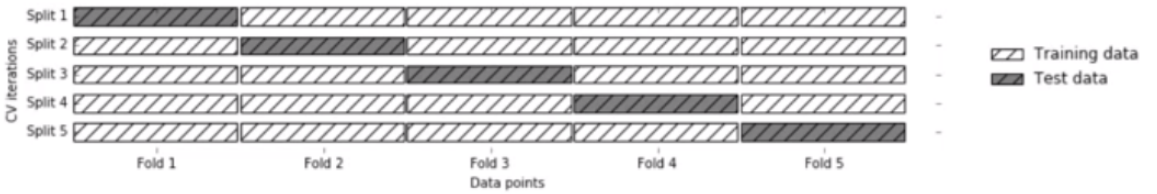


Figure 31: Stacking

You train the weak learners, and the meta model on different parts of the data. So you split your data in **two folds**. The first fold is to fit the weak learners, and the second fold is for fitting the meta model. Each of the weak learners then tries to make predictions for the observations in the second fold. This gives you the ability to fit the meta model with the predictions made by the weak learners as input. **Doing this does mean you only have half of the data**. But you can use **k-fold cross-training** (similar to k-fold-cross validation) to get around by making sure that all the observations can be used to train the meta model.



- Split 1 produces probabilities for Fold 1, split2 for Fold 2 etc.
- Get a probability estimate for each data point!
- Unbiased estimates (like on the test set) for the whole training set!

Figure 32: k-fold-training

In sklearn there is poor man staking, and you just use a Voting classifier.

There is also hold out stacking and naive stacking. The idea of this is that with naive you assume that each model in the stack is equally skillful. With hold out you give every model a weight based on their performance in a cross validation.

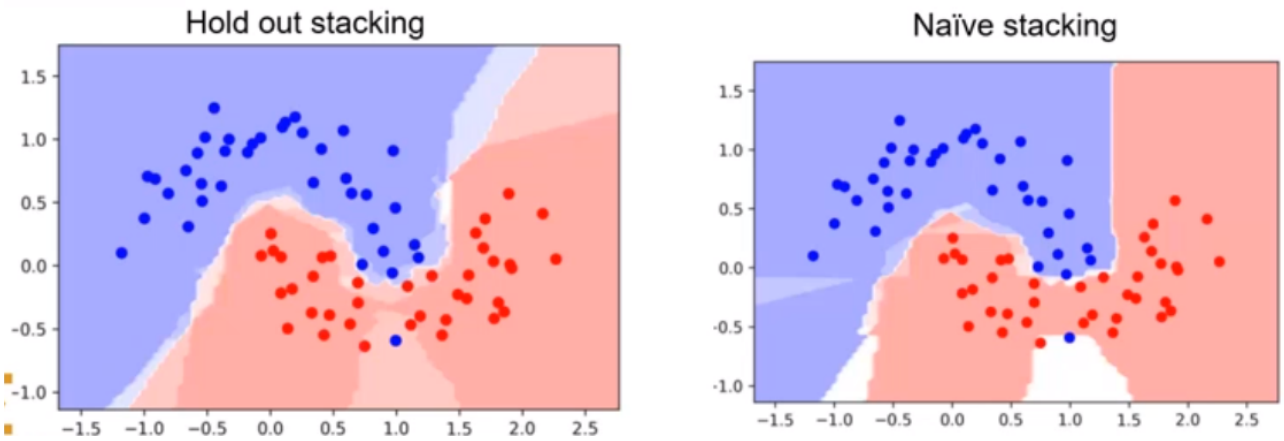


Figure 33: Stacking

6.4.1 Tree based models

You often use tree based models as the weak learners in the ensemble learning because the trees can model non-linear relationships. Trees are very interpretable (if they are not too big). Random forest are very robust. Gradient boosting often has the best performance with good tuning. Trees don't care about scaling they can work with any data so no feature engineering.

We now discussed all the models. There is still more to the story however.

7 Model evaluation

How do you beter evaluate your models? Before you answer that question you should ask what do we actually want to know when we evaluate our model? The answer is the **generalization performance** of the model. We actually don't care really how well the model does on the training data we care about if the training data will be enough to do well on unseen data. We can just get generalization performance of the model by testing it with unseen data. This is only really true for supervised learning because unsupervised is more qualitative.

So far we have just done train/test data. There are few problems with this approach:

- What is the % of test and spit?
- How do we know if the test data is exceptionally different from the training data?
- How do we know whenever the model is overfitting the data?

The answer to these questions is: We don't know. You have to determine on a case by case basis.

So can we do better?

7.1 Cross validation

7.1.1 K-fold Cross Validation (again)

You can also do k-fold cross validation we have seen this already in the other summary. The idea is making multiple models of your data with different parts being part of training and testing every time. k is the amount of models that you want to make.

7.1.1.1 Benefits of k-fold cv

- Leaves less to luck, if you get really good or bad performance by luck then this will show in the results. One of the results will be an outlier. This can happen a lot with imbalanced data.
- Shows how sensitive the model is to the training data set. High variance between fold scores means high sensitivity to the training data.

7.1.1.2 Disadvantages of k-fold cv

- Increases computational time because you train more models
- Simple cross validation can result in class imbalance between training and test. This would lead to lower scores than you could really get. To get around this we can do **stratified cross validation**. This is where you make sure that there is no class imbalance in the different folds.

You can also set k to N, and then you make the whole model with all the data and only test with one point. This as you know is called **leave one out validation**. This is very time-consuming but this can be the **best method for small datasets** as it generates predictions on the maximum available data. Small datasets also decrease the training time again. This can also be useful to find out which items are regular and irregular from the point of view of the dataset.

7.1.2 Shuffle split

The idea of this is that you do k-fold cross validation but which fold you use is picked at random. This:

- Controls test-size, training-size and number of iterations. You can again also do **stratified shuffle split cross validation** what a mouthful.

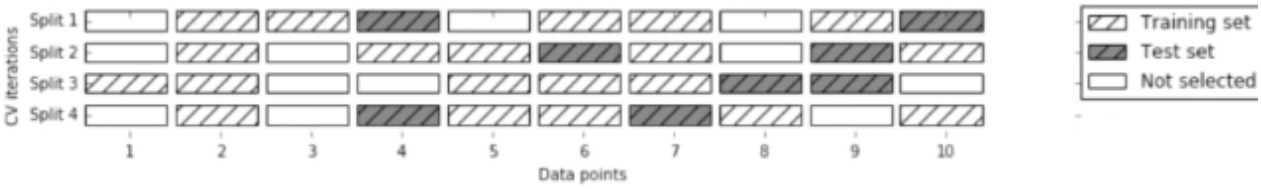


Figure 34: Shuffle Split

7.1.3 Cross validation with groups

In cases that groups in the data are relevant to the learning problem you have to make sure that you **keep the whole group either in the test set or training set**. For instance with the machine learning take home we saw that some faces were in the dataset multiple times but with different expressions. In these cases it is important to keep the whole group of faces in either the test set or training set otherwise you get better or worse performance then is real. Another example is with **time data** this is important to keep in order.

7.2 Tuning

Tuning is where you improve the models' generalization performance by **tuning** adjusting the hyperparameters. You can do this with grid search. Grid search is just a fancy word for training and testing with a lot of combinations of hyperparameters and using the best. You however should make clear before you do grid search which values you want to try. **Simple grid search** is when you just try all the possible combinations of the hyperparameters you specified. **Grid search with cross validation** use cross validation to evaluate the performance of each combination of parameter values.

The **problem with grid search** is that if you optimize the model based on the test data that the **test data is not independent anymore!** To fix this requires another final test set that you use with the best model you found with grid search. This is where the **validation set** comes in. You use the validation set to get a final score on the best model.

When doing grid search you can save all the results you got for the different models and create a grid. This grid can be very useful as usually things that are close on the grid have similar performance.

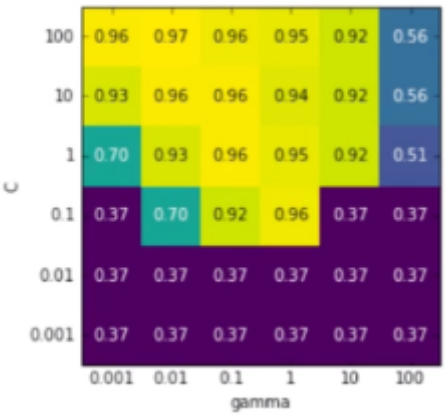


Figure 35: Grid search grid

7.3 Metrics for classification

For classification, you can get a couple of extra metrics besides training testing and validation score. These are:

- **Accuracy:** How many data points were correctly classified?
- **Precision:** When a classification for a class, is it a true positive?
- **Recall:** Of a certain class how many are correctly classified. Also called sensitivity.
- **F1 score:** A way of combining precision and recall. Defined as the harmonic mean of precision and recall.

How to calculate them:

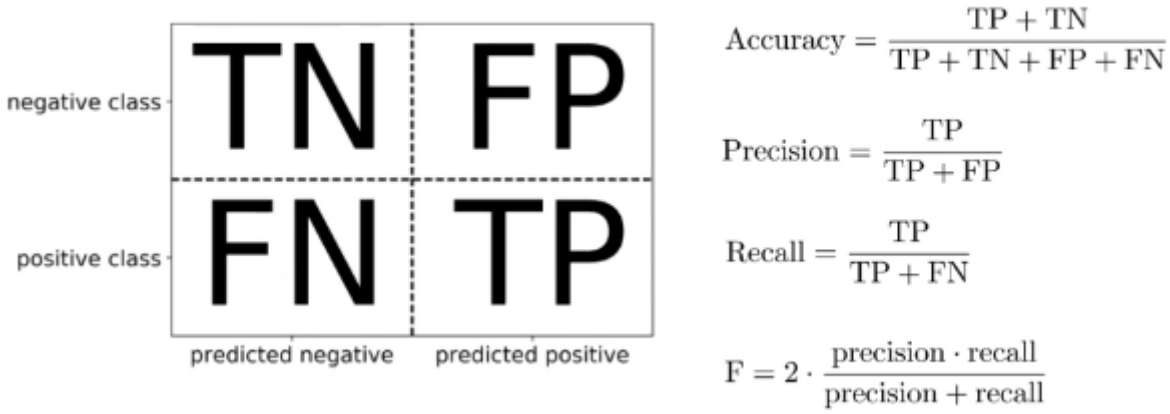


Figure 36: Metrics for classification models

7.3.1 Binary classification

If you only use accuracy you can get some problems if you have imbalanced data. This is because you might get class A right all the time but class B only sometimes but if most of the data is class A you will still have a high accuracy.

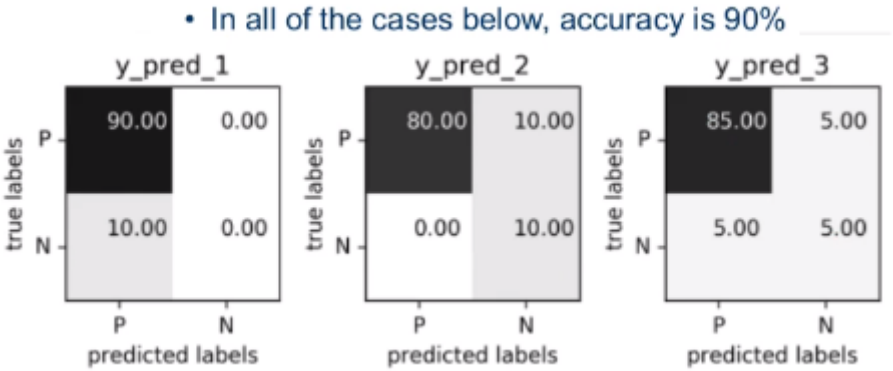


Figure 37: Accuracy can be a problem

What scores you attach to most value to depend on the **goal setting**. Let's say you have

a pacemaker factory then you have to be sure that every pacemaker is 100% save, and you are ok with throwing away some pacemakers. Even some false negatives. Because economically speaking a false positive (working but looks broken) will cost you 10 euros while a false negative (broken but looks ok) will cost you a potential human live (wich will cost you more than 10 euros).

Changing the threshold that is used to make a classification decision in a model is a way to adjust the trade-off of precision and recall for a given classifier. This gives you the precision recall curve. This is useful when you make a new model.

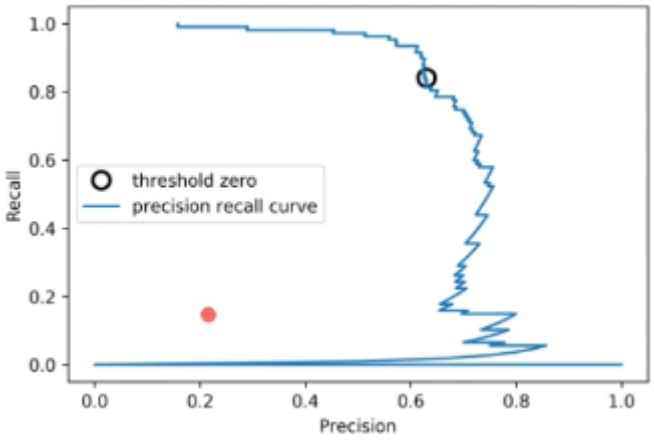


Figure 38: Precision and recall curve

So lowering recall gives you more precision but at some point recall drops off really quick. In this case a **precision zero** gives a good balance between precision and recall. These curves are better for imbalanced data then Roc curves.

Another way to look at this is the Receiver operating characteristics curve (Roc). Instead of reporting precision and recall it shows the false positive rate (FPR) against the tue positive rate (TPR). Here is how you calculate those:

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{= recall}$$

Figure 39: FPR and TPR formulas

Then using that you can make the roc curve it looks like this:

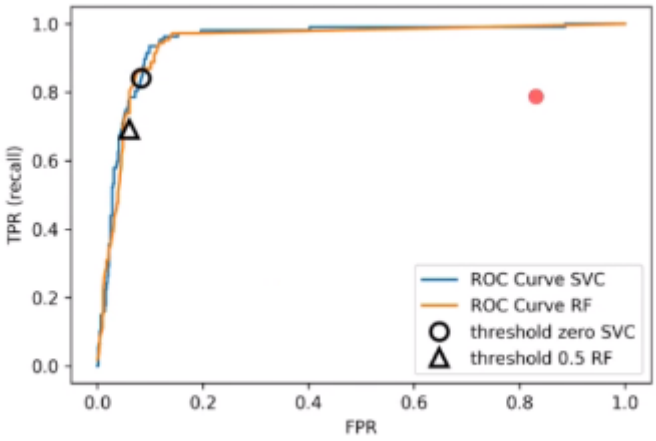


Figure 40: Roc-curve

The more area under the curve the better. If you had complete random guesses the lines would go directly through the middle. This model is better and there is a lot of area under the curve. Better for balanced data.

7.3.2 Multiclass classification metrics

How do you use these extra metrics for classification were you have multiple classes? You make a **confusion matrix** where you calculate the metrics for every class.

They look like this:

```
Precision: [0.736 0.891 0.5  ]

Recal:     [0.869 0.766 0.341]

fscore:    [0.797 0.824 0.406]

Support:   [122.  64.  41.]
```

Figure 41: Confusion matrix

This is the take home assigment where the first collum is for image type 1 second collum for type 2 and third model is type 3.

This also gives you different types of F1 score options: **Macro-averaged F1:** Average F1 scores over classes. So here you assume that all classes are equally important. **Weighted F1** Mean of the per class f-scores weighted by their support **Micro-averaged F1** Make one binary confusion matrix over all classes then compute recall, precision once (all samples are equally important)

You can calculate all the metrics it's about what metric you attribute the most meaning towards.

7.4 Metrics for regression models.

- **R2**: Easy to understand scale
- **MSE**: Mean square error, easy to relate to input
- **Mean or median absolute error**: More robust metrics meaning they are less influenced by the distribution characteristics of the data.
- **Negative version**: Negative versions of the other metrics. Something that decreases if the model gets better.

You can also plot prediction or residual plots with regression models.

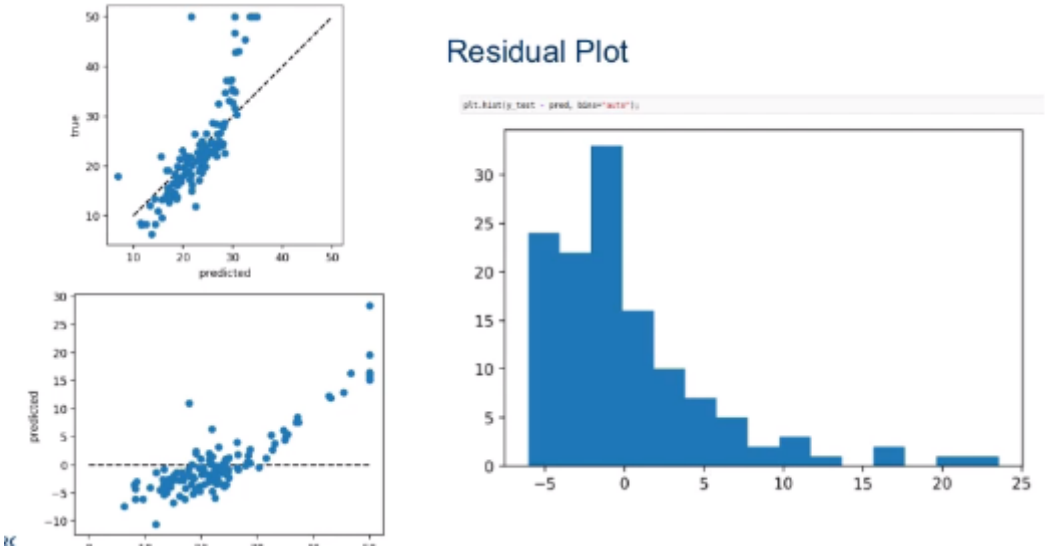


Figure 42: Regression metrics plots

7.5 Dealing with unbalanced data

Imbalanced data is when you don't have the same amount of data per class/outcome. This can happen due to:

- **Asymmetric cost** some labels being more important that others.
- **Asymmetric data** the distribution of your labels in the dataset is unbalanced.

Most datasets follow a Zipfian distribution. The amount of the next class is about half of the current class. How to deal with it:

- Change the data by adding or removing samples. This is called under and over sampling.
- Change the training procedure.
- Don't deal with it.

With **random undersampeling** you lose data but this also makes training faster. With **random oversampling** you get more data so more chance at overfitting and slower training but at least you don't lose data.

You can also do an **edited nearest neighbors method**. The idea is to remove all the samples that are misclassified by a KNN from the training data. Cleans up the training

data by removing outliers. You can also do **condensed nearest neighbors method**. This iteratively adds points to the data that are misclassified by KNN and tries to do the opposite of edited. This way you get a dataset that focuses on the boundaries and so usually removes a lot of points. This is not really used a lot.

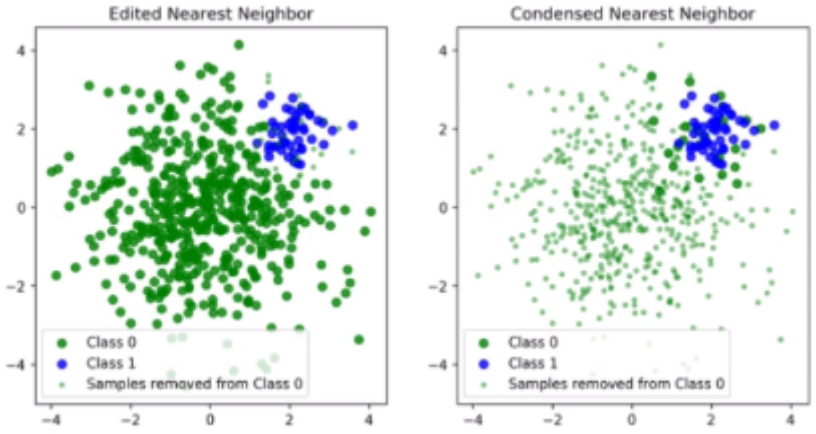


Figure 43: Edited vs Condensed KNN

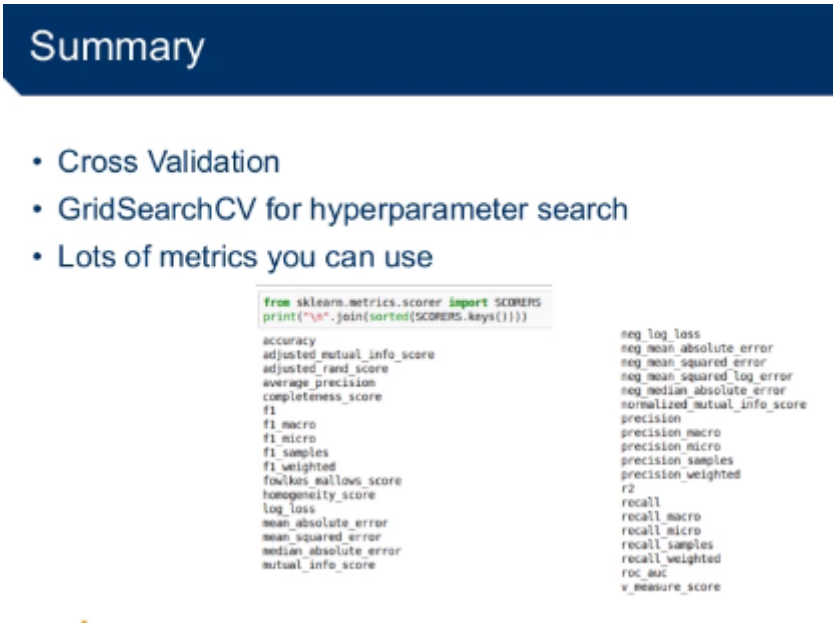


Figure 44: Summary of video 8

Again the idea is to calculate multiple metrics and focus more on the ones that are important to your goals and needs.

8 Preprocessing and Feature Engineering

When you throw your data at a machine learning model as is, you might not get optimal performance. You can do things to your data that do not change the semantics of your data while improving the generalization ability of your model.

8.1 Scaling

We have seen this already a lot. The idea is that you scale the data to be on the scale. This makes it easier to compare data that is on scales with small and big numbers that is not so easy to convert. For instance km and hertz.

8.1.1 Standard Scaler

With this method you calculate the z score for every data point. Effectively you calculate how far something is from the mean.

$$z = \frac{x - \mu}{\sigma}$$

μ = Mean
σ = Standard Deviation

Figure 45: z-score

This works well for non skewed data.

8.1.2 Robust Scaler

The same formula but instead you assume a normal distribution, and you use the median instead of the mean, and you use the interquartile range instead of the standard deviation. This is better for skewed data and deals better with outliers.

8.1.3 Min Max Scaler

Shifts data to the 0-1 interval. Take the maximum value of your dataset and minimum value of your dataset and just calculate where the other data points lie in between those, and then you can get a 0-1 scale. Nice.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Figure 46: Min max scaling

8.1.4 Normalizer

This method does not work for one feature but for the features of a point. It does this by seeing all the features of a datapoint as a row or vector. The idea is then that you scale all the data of a point (one row) so that the norm (of the vector) becomes 1. Then you divide each point of the row by the norm. This is method is not used that often. Mostly used when the direction of the data matters, and it could be helpful for histograms.

> The norm of a vector is the square root of the squared elements of the vector.

8.1.5 All techniques in one graph:

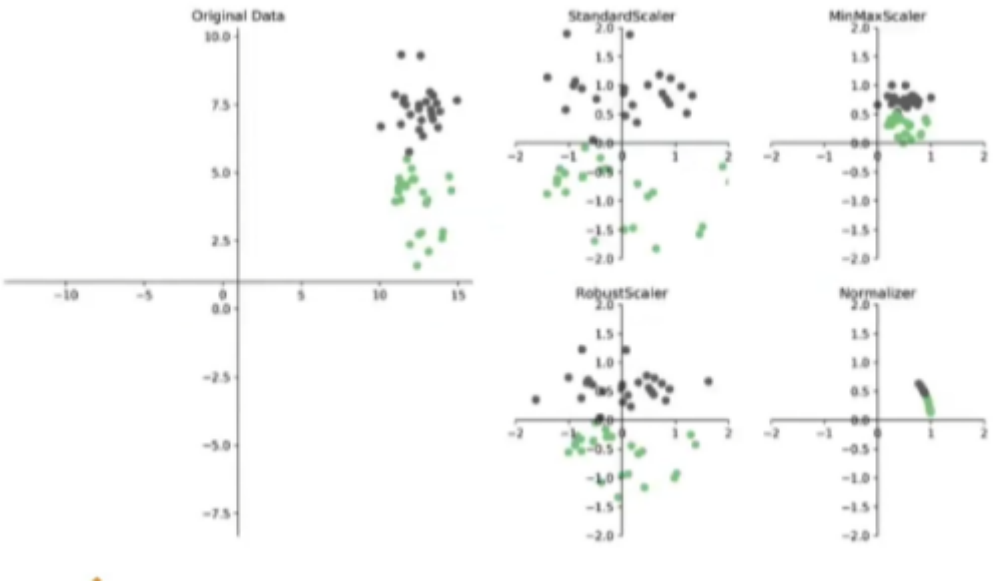


Figure 47: Scaling techniques

8.2 Transforming the data

Instead of just scaling the data you can also **transform** it. There are many ways of doing this, not only for machine learning but, also for stats in general. However most models perform best with normally distributed Gaussian data. Not every model will benifit from normal data. For instance KNN does not. But most models benefit from it.

There are even methods to find the best method to transform your data to a Gaussian distribution. For instance Box-Cox and Yeo-Johnson transform. Both are power transform methods. The difference between these 2 methods is that Yeo-Johnson can also do negative numbers but therefore is also less interpretable. **With these techniques you can automatically estimate the parameters so that skewness is minimized and variance is stabilized.** Here is a visualization of the different techniques:

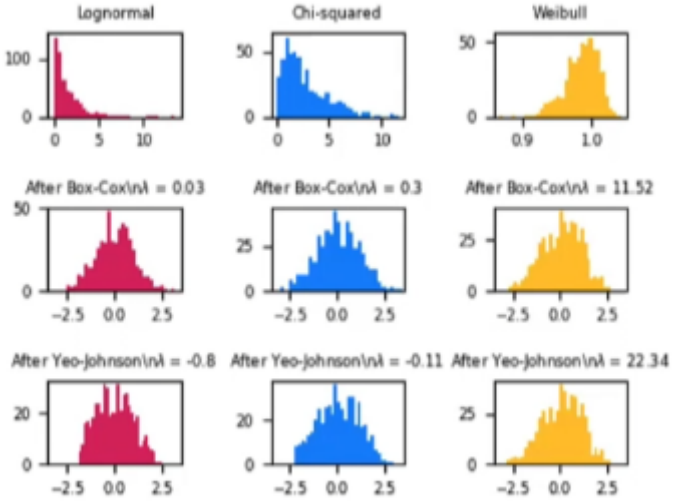


Figure 48: Univariate transformations

These methods are really great because you can just almost blindly transform your data to a normal distribution. You could choose your own parameters to make your data more interpretable. For instance in this case a transformation to the log scale is apparently better:

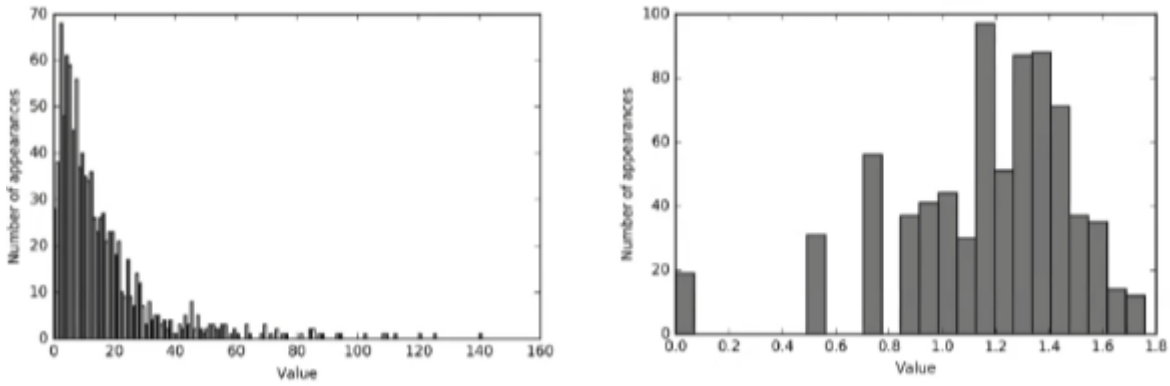


Figure 49: Log transformation

8.3 Binning

Binning (also known as discretization) is a preprocessing method to make linear models more powerful on continuous data. The idea is to separate feature values into n categories that are equally spaced over the range of values. You do this by separating the features into n categories by a single value (usually the mean) and then you **replace all values within a category by a single value**. This is effective for models with only a few parameters such as regression, but it is not effective for models with a lot of parameters like decision trees as can be seen in the picture below.

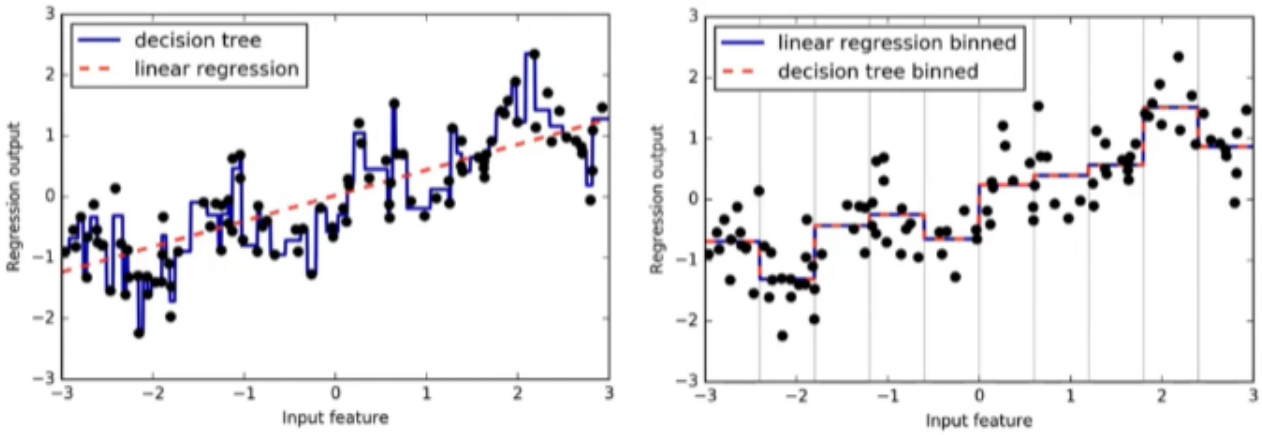


Figure 50: Binning in action with linear regression and a decision tree

8.4 Missing values imputation

Missing values imputation is about dealing with missing values in the dataset. With real data there are various reasons why data would be missing. Missing values might be represented in different ways common missing value representations are **None, blank, ~~0~~, NA, Na, NaN, Null, undefined ……** You should **NOT** use numbers to indicate missing data as this might not be picked up on. Always use strings or something else inconvenient. You could just throw away the data that contains missing values, but we can do better than that. You want to do this because it is sad to throw away a whole vector of data only because one scalar is missing.

This is how it would look like:

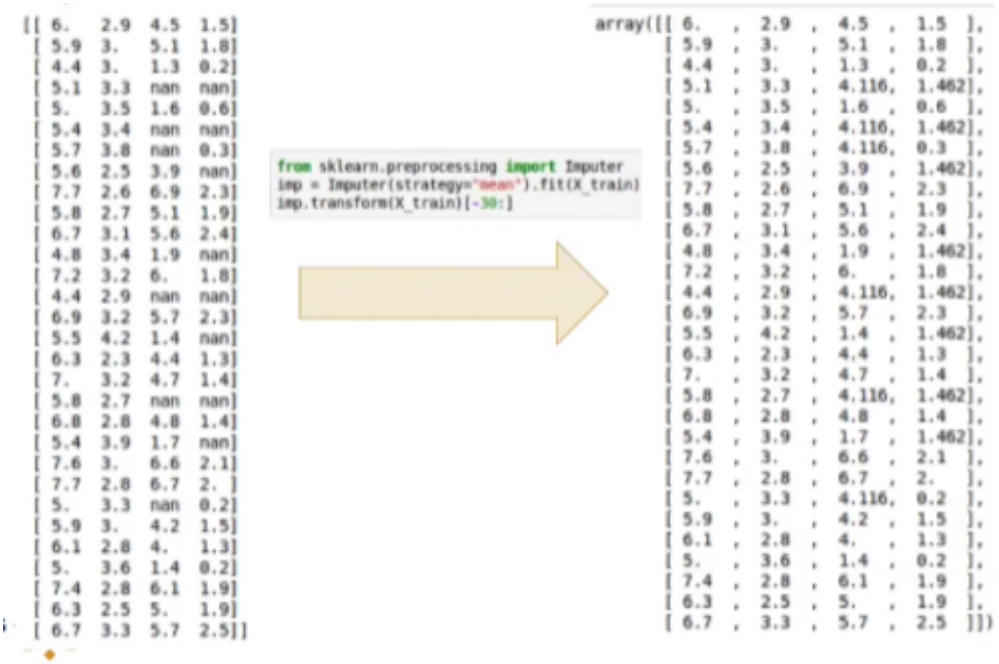


Figure 51: Missing values

We can make a model to impute (predict) the missing values. Different techniques for this include:

- **Mean / median** replace missing values with median or mean of the data. Not a great technique not very flexible.
- **KNN** use the average of k neighbors feature values. This is much more flexible. The mean is basically k = n.
- **Model driven** using another model to impute the values like random forest.
- **Iterative** using a regression model with all features expect one to predict the missing features and then do this for all the missing features. You can use the recited values for the next values.

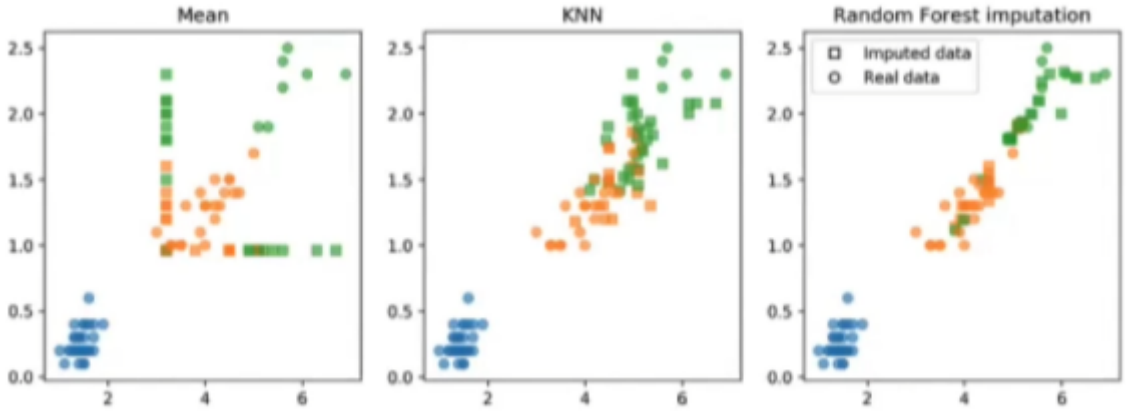


Figure 52: Data imputation

8.5 Dealing with categorical data

Data is often not nice and numeric but categorical. Child/adult boy/girl/other True/False. Data can be categorical - ordinal - interval - ratio. We have seen this before there are ways to deal with this. Usually by converting the categorical data to a number. Remember you don't have to do this for decision trees. To change the categorical data into numbers you can use **one hot encoding** (making multiple boolean features for each possible category), or **a count based encoding** using an aggregation (how much something occurs). Good for high cardinality (a lot of possible classes). We saw this in other courses already.

8.6 Feature selection

Feature selection is choosing what features you want to use when training your models. Why not just use all the features you have? There are good reasons not to do this.

8.6.0.1 Benefits of feature selection/dimension reduction:

- Avoid overfitting
- Lower compute time

- Avoid curse of dimensionality
- Less storage for model and dataset
- Reduced space required to store the data
- Less dimensions makes models faster to compute
- Some algorithms like KNN don’t perform well with a high dimension
- Takes care of correlations by removing redundant features. For instance time run and calories burnt both sort of say the same thing

8.6.1 The curse of dimensionality

The curse of dimensionality is the fact that adding more and more features will give diminishing returns on the model performance. You will eventually even get negative performance by adding more features.

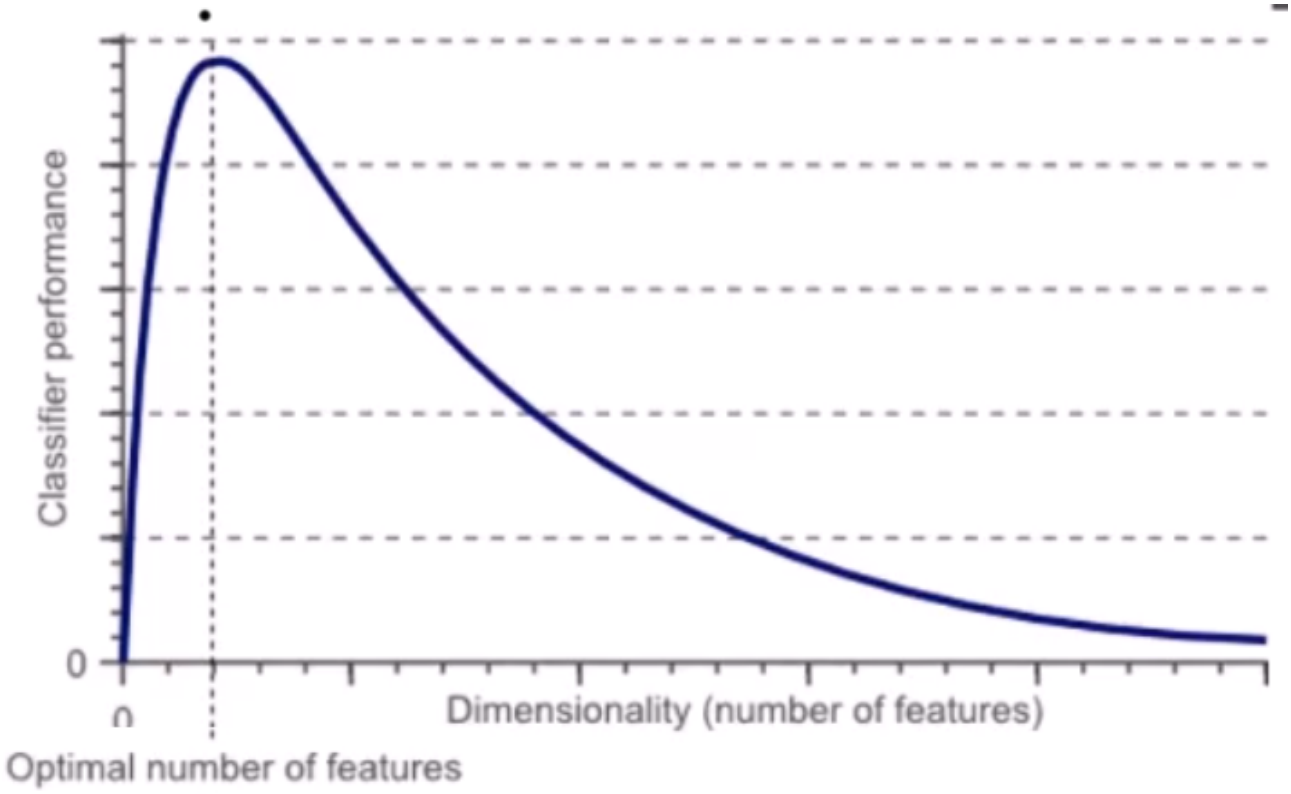


Figure 53: Curse of dimensionality

After you have reached the optimal number of features, your performance decreases by adding more features. This is because of overfitting because adding more features exponentially increases the volume of the feature space.

This is a trade of because we want informative features but having too much will cause overfitting. As such, the features you choose should be as informative as possible.

There are 3 different strategies discussed.

8.6.2 Univariate statistics

With this feature selection technique you look at each feature individually and **remove features** that do not have a **significant relationship** to the target. You can either say keep 20 features with the highest confidence, or you could say keep the features with a confidence value higher than a threshold. How to get confidence is related to the ANOVA method. The idea of this technique is to keep features that have a high statistical significance to the target. You can use f-value or **mutual information** (not linear). Or you can calculate both.

8.6.3 Model based selection

This is for getting the best fit for a particular model. Ideally you do an exhaustive search over all possible combinations but this is not feasible as it would take too long. That's why you use heuristics. But this is lasso regression or linear models, tree based models. You can also do **multivariate models** those are linear models that assume linear relation. The idea of this technique is to just try a lot of combinations of features and find out based on the results and heuristics which combination works the best.

8.6.4 Iterative approach

Start with the most important feature and keep adding until you have the amount of features that you want. This is used more for statistical models. You can also do this backwards were you drop features the least important features until you have the amount you need. This method is quite computationally expensive as every time you try leaving out a feature you have to train again.

You can rank the features using RFE -> Recursive feature elimination and selection. This is doing the iterative approach backwards and forwards to come to the best combination of features.

8.7 Dealing with Text data

Most machine learning models really like their data to be numerical but when you get text data this gets more difficult and preprocessing steps get more involved. How do you represent Text data in a matrix format? Text data has no predefined features. The most common way to deal with this problem is to use **Bag of words**.

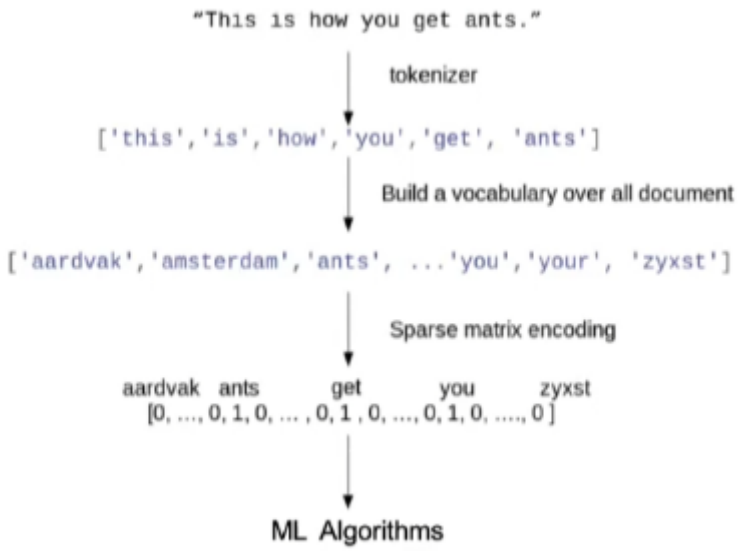


Figure 54: The bag of words technique steps

Before you do anything, you need to **tokenize** your raw text. This means turning the raw text into a list of the raw elements of the text. Tokenizing is a whole subject in its own right. Are . , ; ” < > ’ tokens for instance? You could do **stemming** where you reduce words to their root like walking to walk. You could do **lemmatization** were you replace words with words from language databases. You could **restrict the dataset** by removing words that only occur once to reduce matrix size. You could remove words with low semantic content like the, is, a. Or you could not do any of these things. Do you split the words as 1 word or do you try parts of sentences? What about misspelled words? All dials you can tune in the tokenizing step. Tokenizing is easy to do but hard to do well.

Once you have tokenized your all your data, you have built a vocabulary with the tokens. Then the next step is to create a sparse matrix encoding out of this vocabulary. Where for every token you say if int appears in a certain document or not. You could also use count instead of boolean to represent how often a word appears in a document.

> A document is a very general term that can mean many things. It could mean a paragraph, different chapters, different files, different sentences etc.

Instead of boolean or count you can also use the **TF-IDF** value of a token. This will tell you how informative a certain token is for a particular document.

Term Frequency (TF) = Number of times a token appears in a document Inverse Document Frequency (IDF) = log(N/n). Where, N is the number of documents and n is the number of documents the token has appeared in. Rare tokens have high IDF and frequent words have low IDF. Thus, this highlight words that are distinct. This kind of leans into information theory where the things that are less common actually give you the most information. You calculate TF-IDF as TF*IDF.

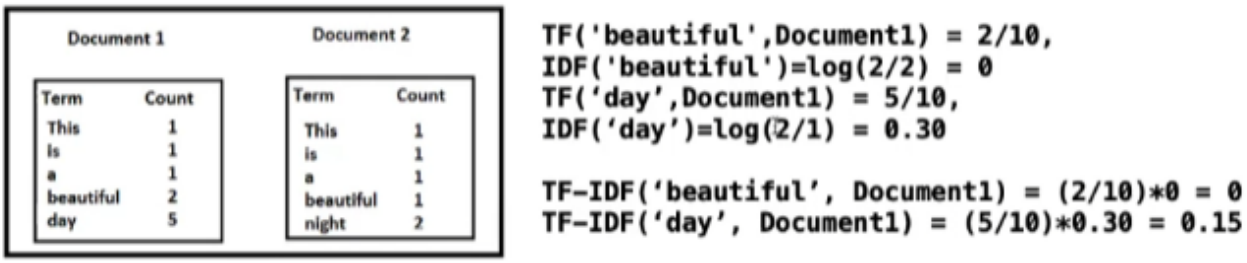


Figure 55: TF-IDF example

As you see, the word day has a higher score than beautiful.

9 Dimensionality reduction

Dimensionality reduction is the first **unsupervised technique** that we got a look at. This means this technique does not have targets to optimize the models. This is more of a data preprocessing method. The idea of dimensionality reduction is to represent high dimensional data with lower dimensional data with fewer features. But the features that are there will summarize all the other data. A good example were this is useful is images. Only if you have an 8 x 8 pixel image this is already 64 features. If you have a 1280x720 image, you have 77600 features. The possible **combinations** increase exponentially. This becomes hard to deal with and besides that the **curse of dimensionality** comes knocking on your door. Do you really need to take all the individual pixels into account anyways? Reducing the dimensionality of your data also makes it much easier to visualize.

There are 2 discussed ways of doing this:

9.1 Principal components analysis (PCA)

PCA operates on the features without labels and thus is an unsupervised learning method. PCA threats each feature as an axis of an N-dimensional coordinate system. Then it rotates the XY coordinate system to turn the 2 dimensional data into 1 dimensional data. This only works if the features are highly correlated and form a nice vector together.

In short this is what happens:

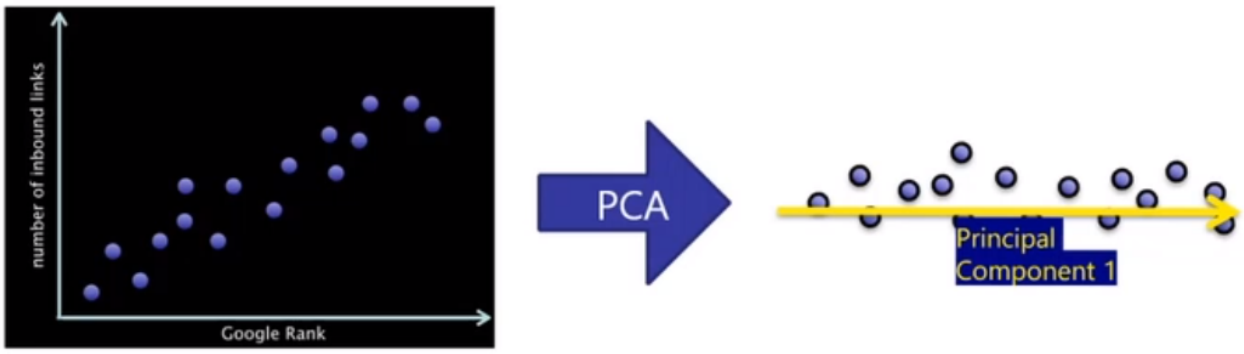


Figure 56: Pca summary

9.1.1 Longer Example:

If you have 2 features that are highly correlated, we can reduce them to 1!

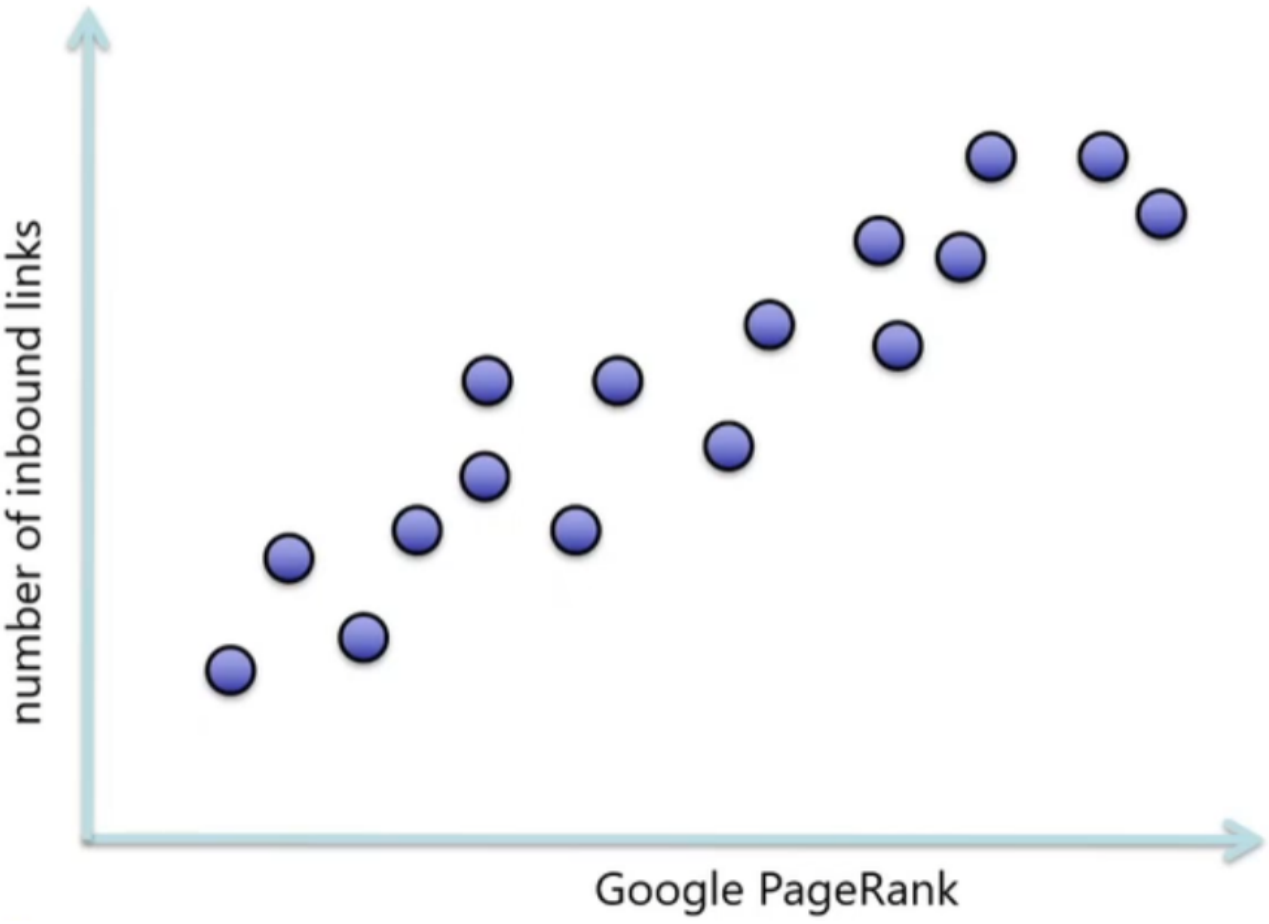


Figure 57: Correlated features

These features are nicely correlated. Which means you could make a 2 vectors through the points with a 90-degree angle like this:

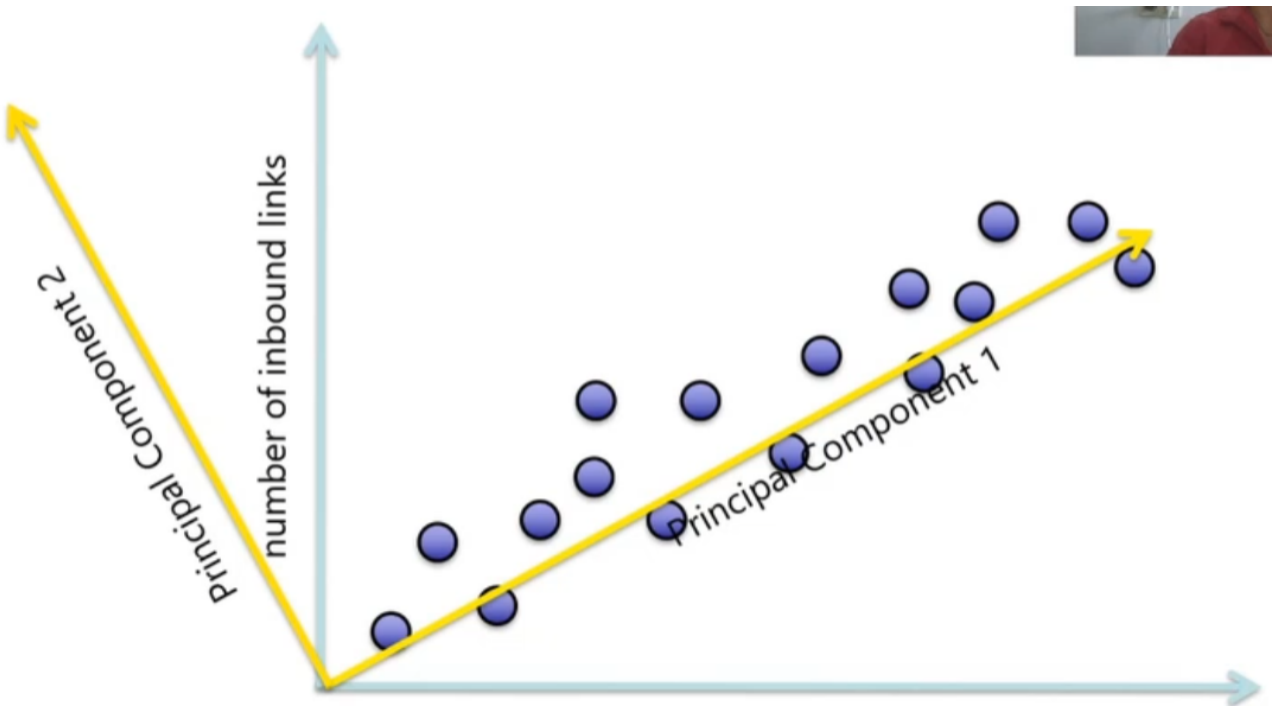


Figure 58: Pca components

Then you rotate the vectors to be in line with the x and y-axis. This way you make the line almost 1d instead of 2d.

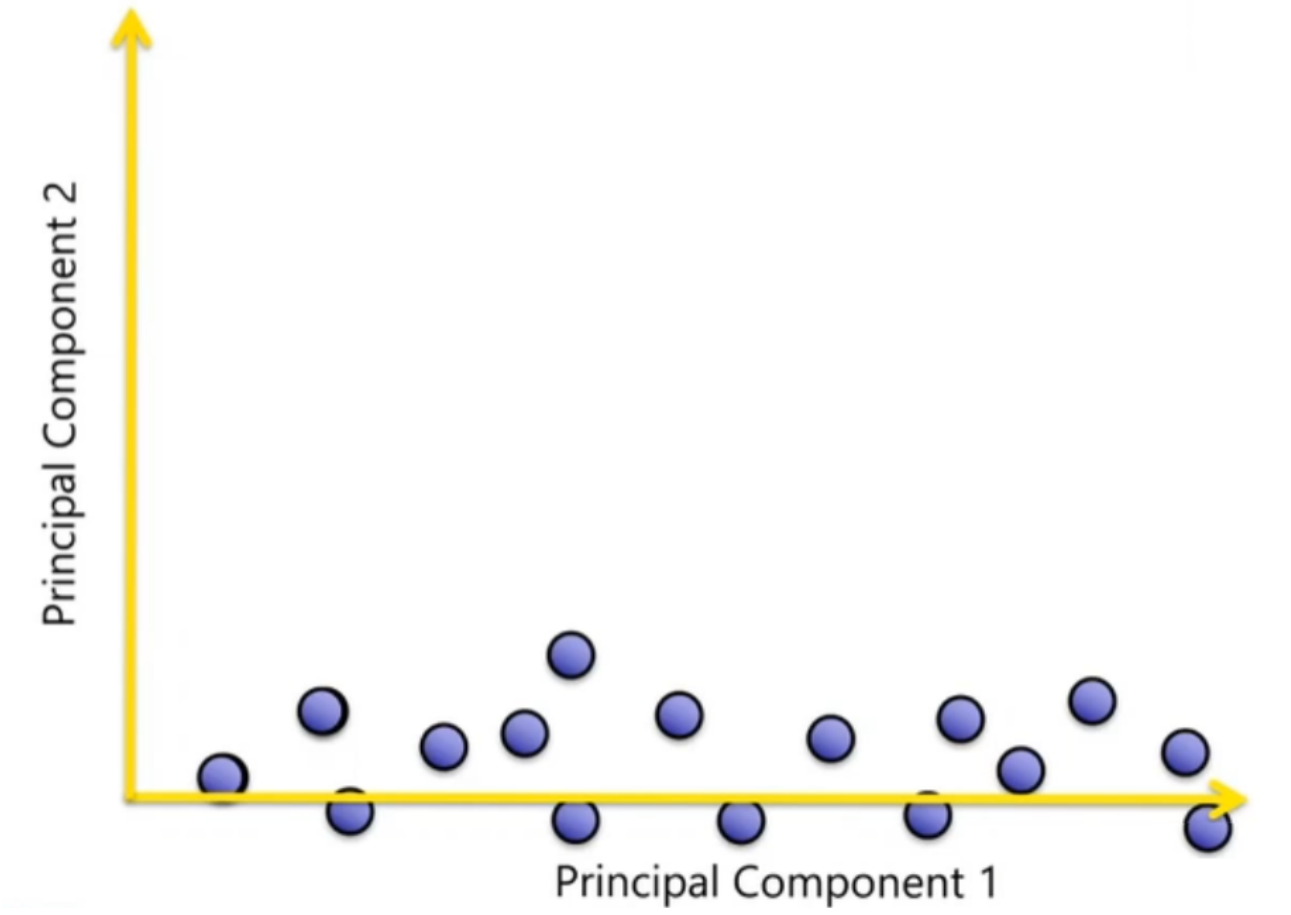


Figure 59: Pca components rotated

So now you have 2 components. We put in 2 features and got 2 other features back... But now these 2 features are a combination of the 2 original features:

- Principle component 1 = a * feature1 + b * feature2
- Principle component 2 = c * feature1 + d * feature2

a, b, c and d are related to the **Pearson Coefficient** of the lines and the correlation of the points. The first component explains as much variance as possible of the dataset. The second component tries to explain remaining variance in the dataset and is uncorrelated to the first component. You start the line in the middle of your cloud of points.

If you look at the last plot you see that now the data is almost completely described by principal component 1. Principal component 2 describes the noise. **Therefore we can discard principale component 2.**

This leaves us with one feature!

If you remove the second component you get a line like this:

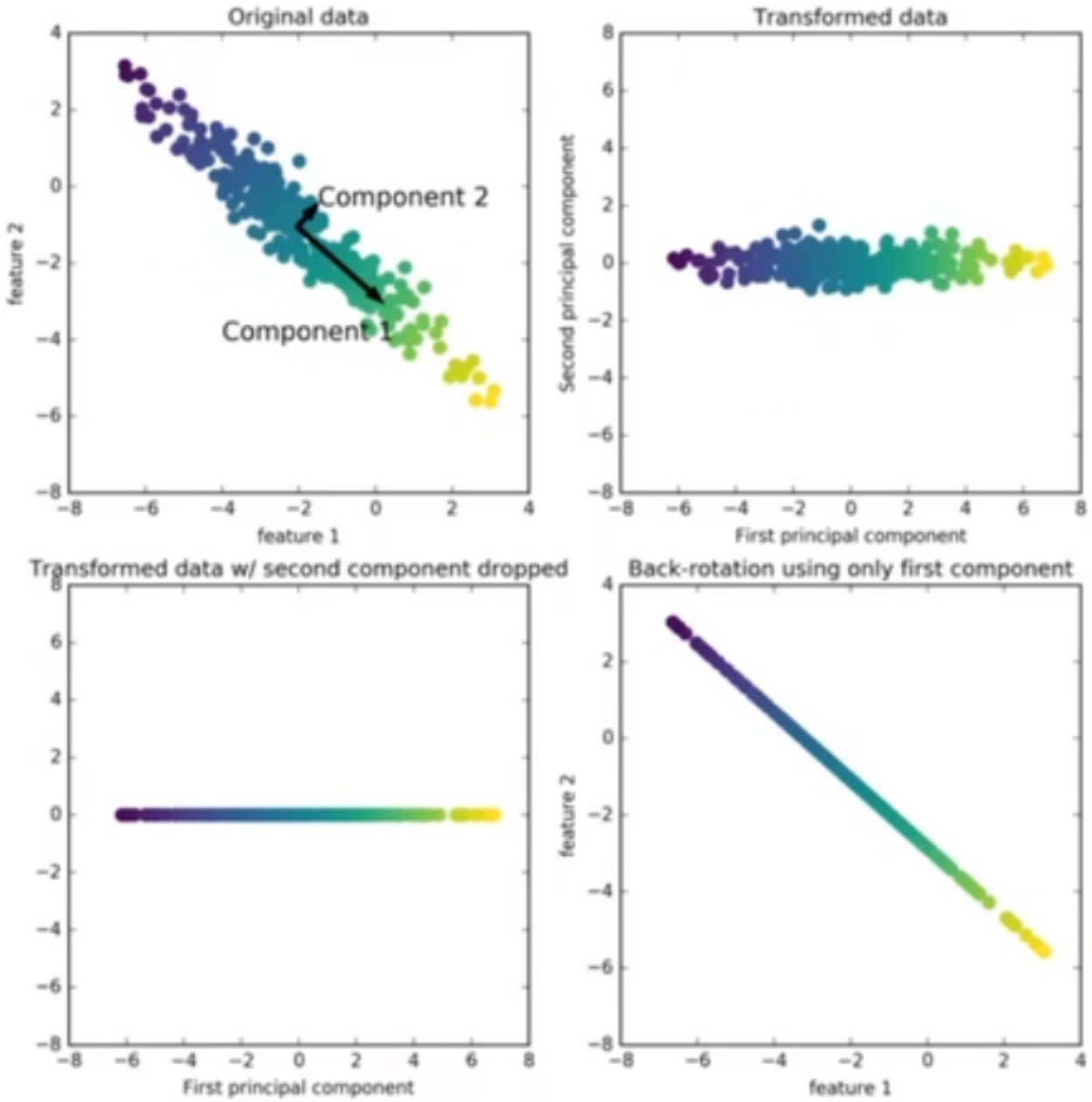


Figure 60: PCA overview

So now we have a machine that can combine 2 correlated features into 1.

9.1.2 How do you compute PCA

1. Take the whole dataset and ignore the labels
2. Compute the mean and covariance
3. Center and scale the data (PCA is sensitive for scaling)
4. Get the eigen vectors and eigen values from the covariance matrix or correlation matrix, or perform a singular vector decomposition
5. Sort eigenvalues in descending order
6. Choose the k eigenvectors that correspond to the eigenvalues where k is the number of dimensions of the new feature space. (So k <= d where d is the original number

of dimensions)

7. Construct the projection matrix W that will be used to transform the data.

> A projection matrix is a matrix out of the top k eigenvectors

If you want to do more than 2 features you just rotate the coordinate system in such a way that the projection of the data of the first principal component has the largest variance, the second principal component has the one-but the largest variance and so on.....

The first couple principal components will be the most relevant for classification.

This is how pca looks when you do it to an image:

Figure 61: Pca when applied to images

9.1.2.1 How many components to keep?

This is one of those case by case questions, it depends on the amount of training data you have, the complexity of the decision boundaries and what classifier you are using.

A good measure is the **total explained variance** by the new data. The more components you keep the more variance is explained by the data. However, each component you add will give diminishing returns on how much extra variance you explain. Try to

choose a k that explains an amount of the variance that you are happy with. My advice is at least 90 to 95 percent of total variance explained.

9.2 Non-negative Matrix factorization (NMF)

The idea of this is to take the data and find the **latent space**. Latent means hidden. This will give you smaller data. **The latent features have to be non-negative**. You find the space with Matrix factorization.

9.2.1 Matrix factorization is:

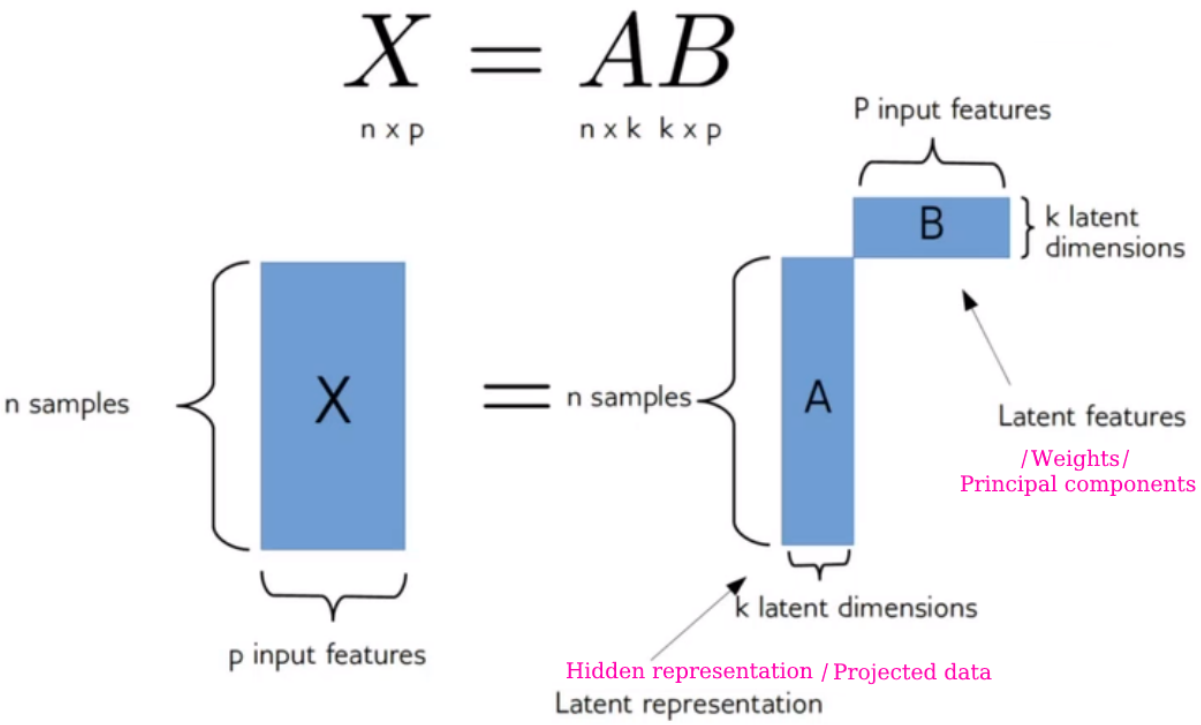


Figure 62: Matrix factorization

The idea is to represent the original **X** matrix in a latent space. The latent space contains a hidden simpler compressed representation of the data. In the latent space similar features are closer together. Here is an example with numbers in matrix and latent space:

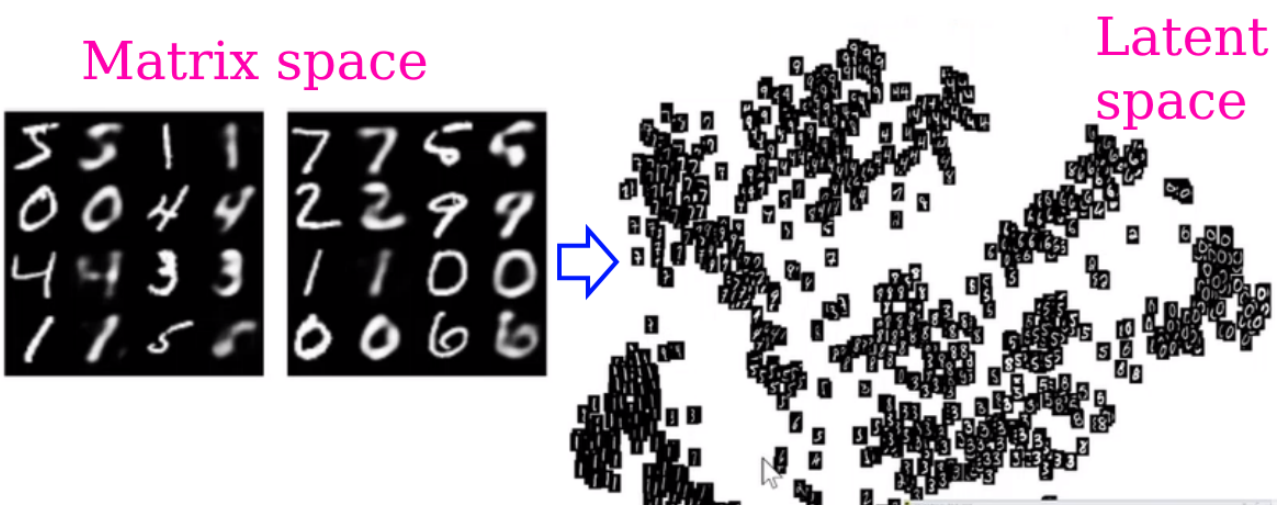


Figure 63: Latent space example

You can then multiply the latent space (A) with weights (B) to get your original data back (X).

9.2.1.1 Why use NMF?

- Positive weights can be easier to interpret
- No cancellation like PCA
- No sign ambiguity like PCA (what does the - mean?)

NMF can be viewed as soft clustering each point is positive linear combination of the weights.

9.2.1.2 Downsides of NMF

- Can only be applied to non-negative data (images, text)
- Sometimes components are not interpretable
- Non-convex optimization, requires initialization
- Can be very slow
- Not orthogonal (You don’t really know where the components are)

9.3 PCA VS NMF

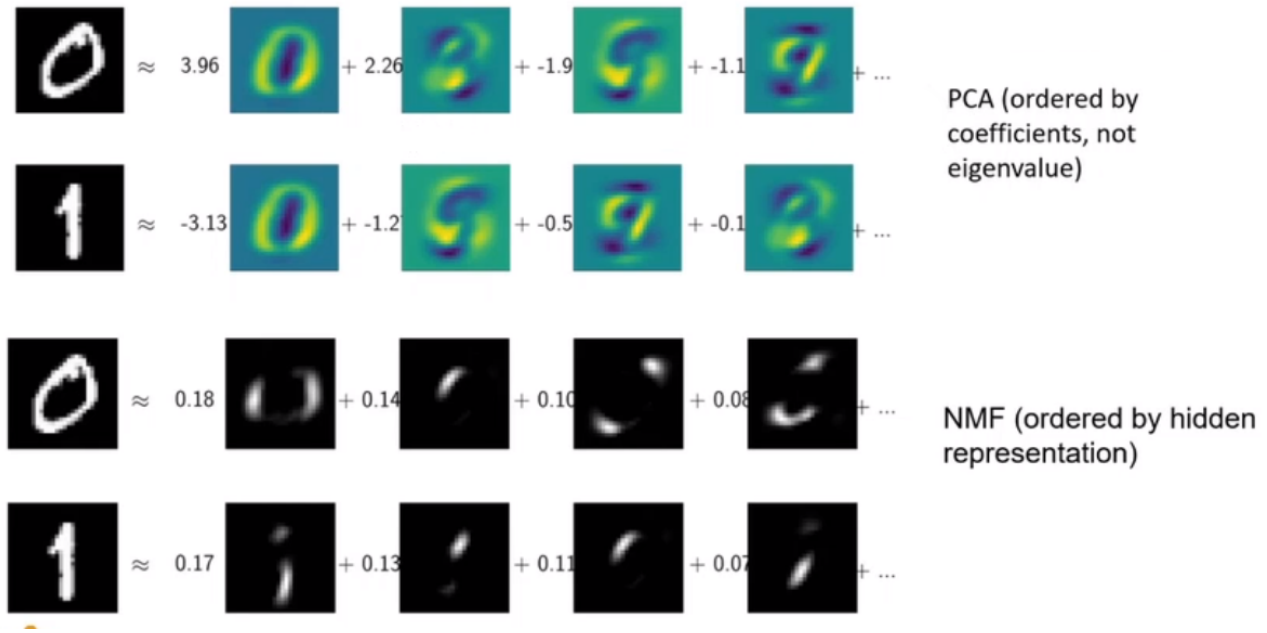


Figure 64: PCA VS NMF

9.4 Manifold learning

Manifold learning is **dimensionality reduction for data visualization**. The idea behind manifold learning is to make a complex mapping of the data also called the **underlying manifold structure** that can then be brought down to a lower dimension using dimensionality reduction to be visualized. This seems to work really well. **This technique is only for visualizations** as the axes of the plots don't correspond to anything in the input space, and these mappings can often not transform new data. But they give pretty pictures!

There are multiple ways of doing this some of which can also be used for other things then visualization, but the one talked about in the lecture was T-distributed Stochastic Neighbor Embedding. Actually a technique made by someone from tilburg University so that's cool.

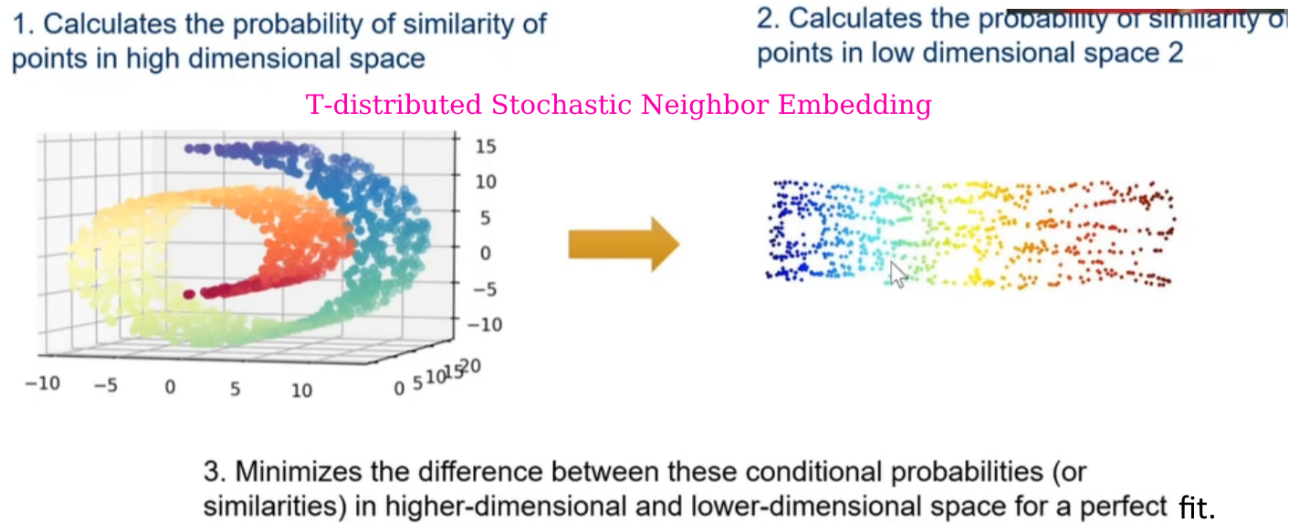


Figure 65: T-distributed Stochastic Neighbor Embedding

By minimizing the probabilities of difference you make sure that points that are close together in the high dimensional space are also close together in the low dimensional space.

You start with a random embedding and then iteratively update points to make “close” points close. You do this with a learning rate. The global distances are less important it is more about neighbours. This technique is good for getting a coarse view of the topology of your data and can be good for finding interesting datapoints.

9.4.1 t-SNE VS PCA

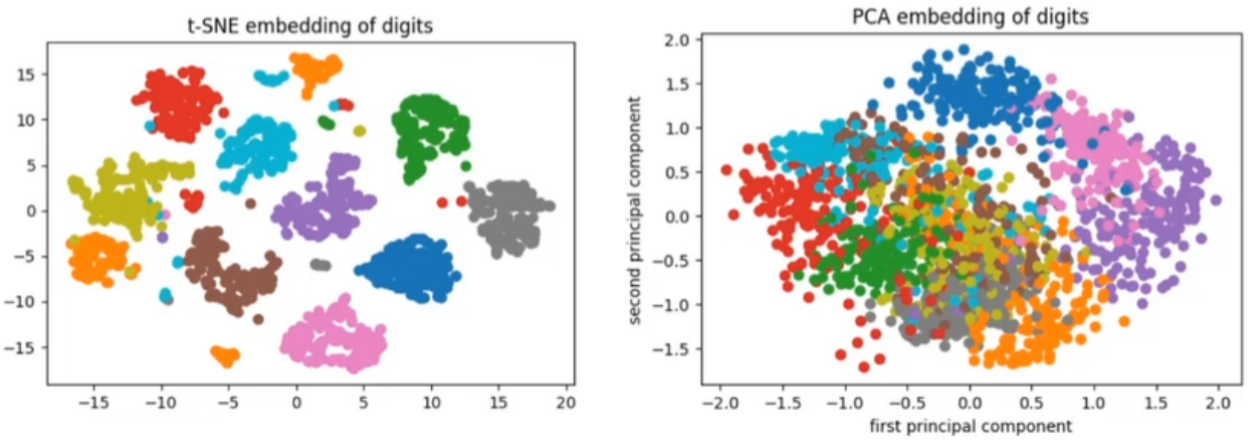


Figure 66: t-SNE vs PCA

The values on the t-sne axis have no meaning

9.4.2 Tuning t-SNE

There are a lot of hyperparameters for t-SNE

- **n_components** (default: 2): Dimension of the embedded space.
- **perplexity** (default: 30): The perplexity is related to the number of nearest neighbors that are used in other manifold learning algorithms. Consider selecting a value between 5 and 50.
- early_exaggeration** (default: 12.0): Controls how tight natural clusters in the original space are in the embedded space and how much space will be between them.
- **learning_rate** (default: 200.0): The learning rate for t-SNE is usually in the range (10.0, 1000.0).
- **n_iter** (default: 1000): Maximum number of iterations for the optimization. Should be at least 250.

Here is a website to play with t-sne its pretty fun https://distill.pub/2016/misread-tsne/

10 Clustering

Clustering is finding natural groups in data. You don't know the groups beforehand so this is an **unsupervised learning method**. This can be a hard problem because you can cluster in many ways. For instance by choosing a different amount of groups/clusters that you want:

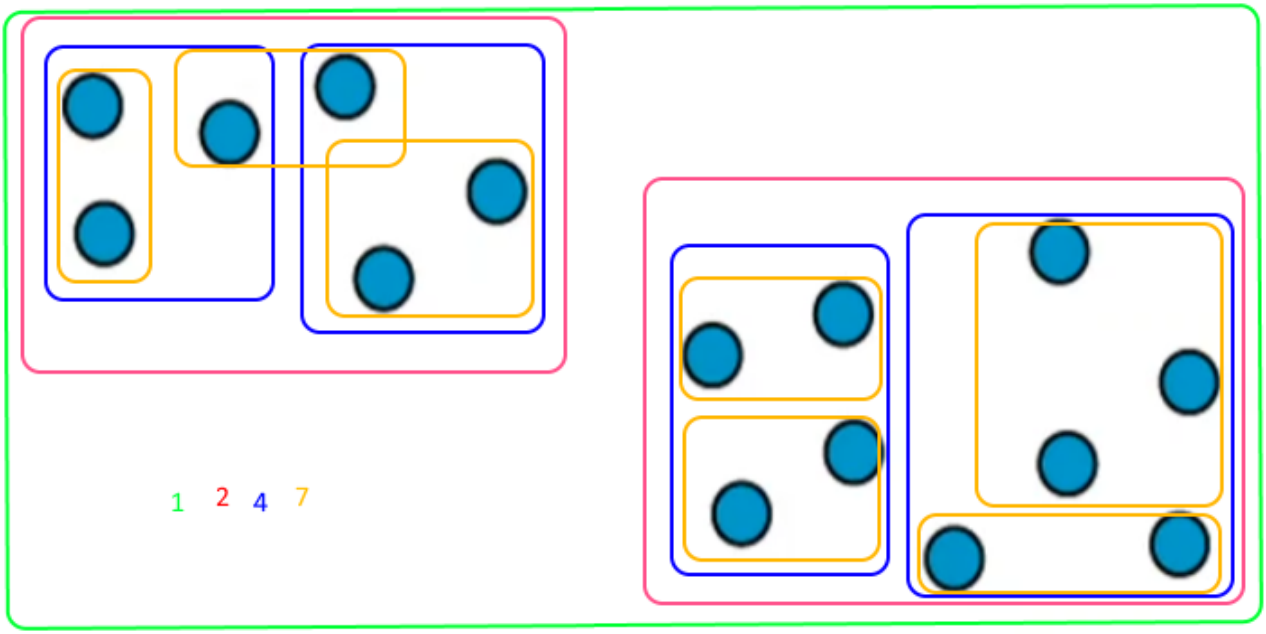


Figure 67: Clustering with different amount of clusters

Humans are actually really good at these types of problems but it is really hard to tell a computer how to do this.

10.1 Goals of clustering

- Data exploration

  - Are there coherent groups?
  - How many groups are there?
- Data partitioning
  - Divide data by group before further processing
- Unsupervised feature extraction
  - Derive features from clusters or cluster distances

We can achieve these goals with these techniques:

- K-means
- Hierarchical clustering
- Density based Techniques
- Gaussian Mixture models

10.2 K means

The idea of k means clustering is to separate samples into **k groups of equal variance**. So to do K-means clustering you need to know how many clusters you want. There are techniques to make good decisions about this.

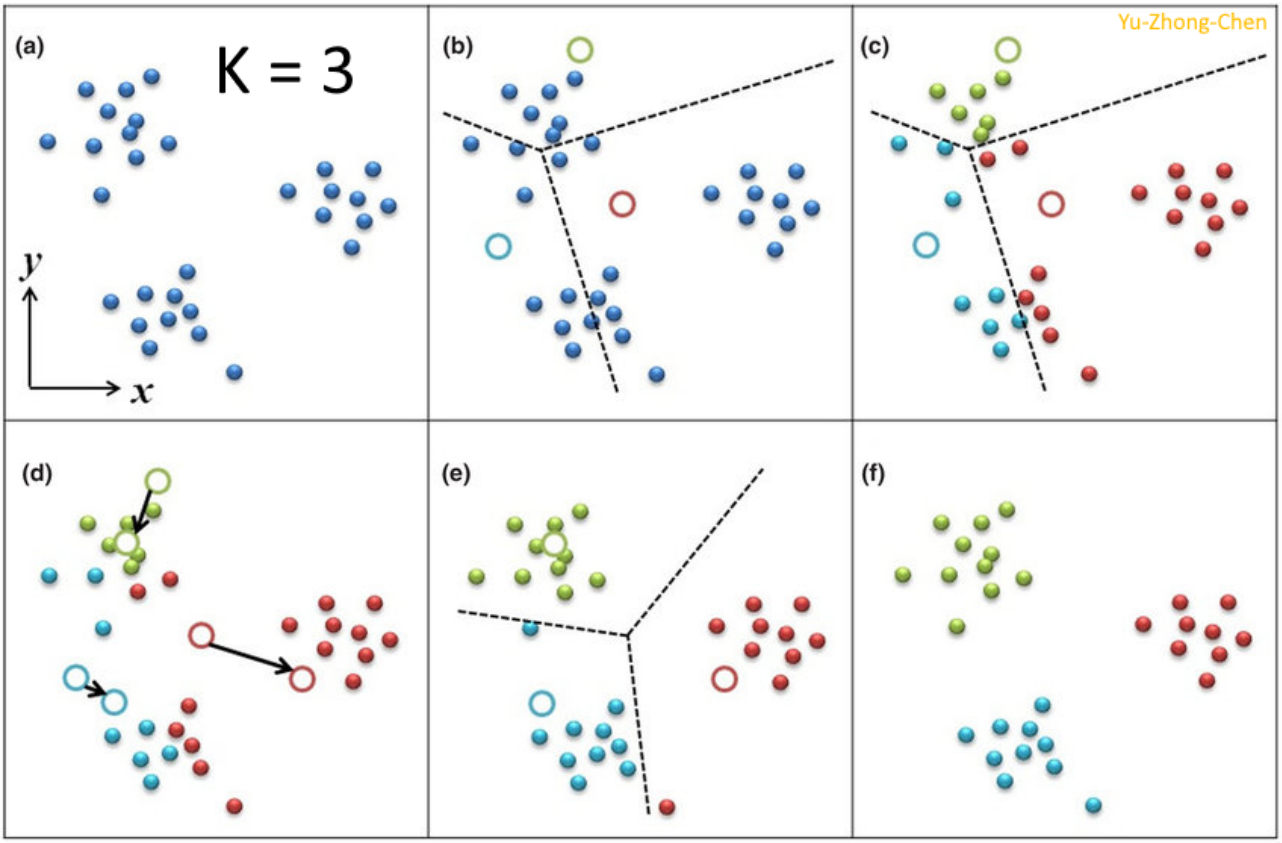


Figure 68: K means clustering

The algorithm goes like this:

1. Choose the number of clusters k (a)
2. Randomly choose initial positions of K centroids (b)
3. Assign each of the points to the “nearest centroid” (c)
    - This depends on the distance measure. Manhattan or Euclidean etc.
4. Recompute the centroid positions (d)
5. If solution converges stop otherwise go back to step 3. (e)
6. Clustering is done and data is divided into clustering (f)

You recompute the centroids (step 4) by finding the local minimum of minimizing squared distances.

$$\min_{\mathbf{c}_j \in \mathbf{R}^p, j=1,\ldots,k} \sum_i \left\| \mathbf{x}_i - \mathbf{c}_{x_i} \right\|^2$$

$$\mathbf{c}_{x_i} \text{ is the cluster center } \mathbf{c}_j \text{ closest to } \mathbf{x}_i$$

Figure 69: Formula for minimizing squared distances

In English this means moving the centroid into the direction that will make the distances between the points in the cluster, and the centroid the smallest.

When does a point belong to a certain cluster? **A point belongs to the cluster of the centroid that’s the closest to the point**. Based on this rule you can draw decision boundaries around clusters. This is called a **Voronoi-diagram**. Clusters/decision boundaries are always convex in space meaning a shape where all of its parts point outwards.

Cluster decision boundaries

Things can go wrong like this:

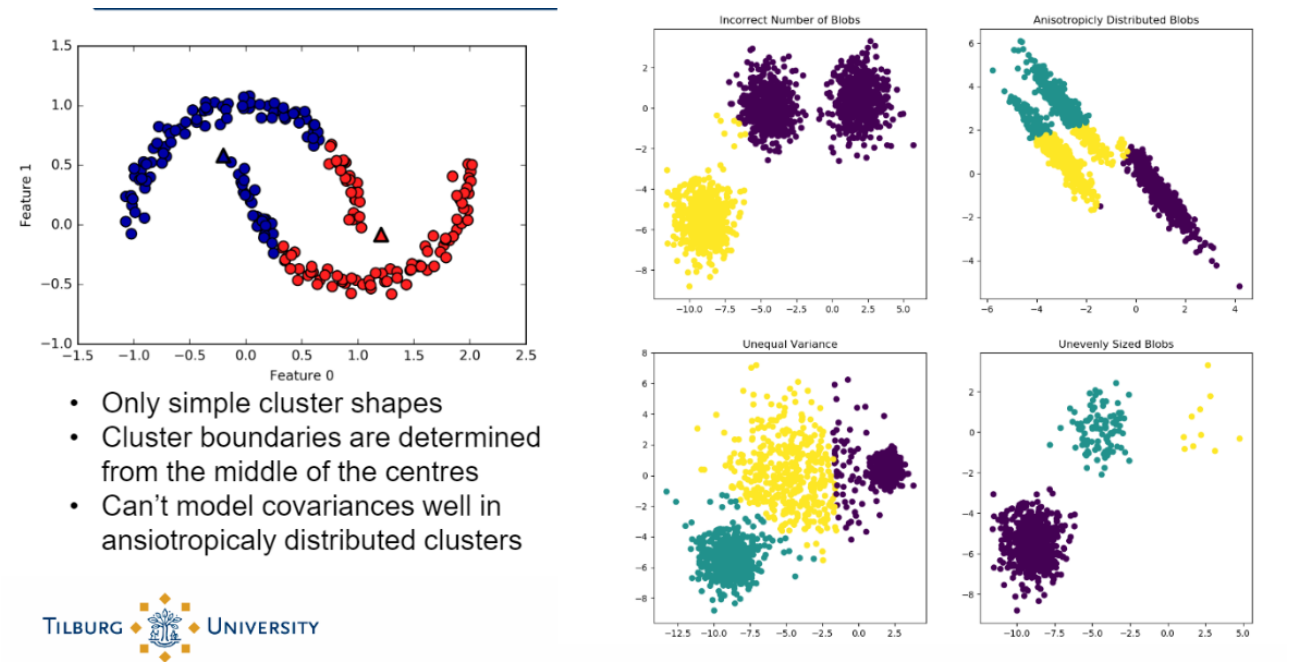


Figure 70: K-means clustering going wrong

Because K-mean clustering is very dependent on were you start with the centroids, it's a good idea to try cluster multiple times (default) with different starting positions. For large datasets K-means initialization (calculating the distances) can take longer than the actual clustering (Minimizing the distance). For this there is MiniBatchKmeans.

10.2.1 MiniBatchKMeans

MiniBatchKMeans uses mini batches to reduce the computation time while still attempting to optimising the same objective. It works like this:

1. Draw samples randomly from the dataset to form a minibatch
2. Assign the nearest centroids
3. Update the centroids by using a convex combination of the average of the samples, and the previous samples assigned to that centroid.
4. Perform above till convergence or iteration limit.

Turns out that this is much faster and almost the same results.

10.3 Hierarchical clustering

The idea behind hierarchical clustering is to have a series of partitions from a single cluster containing all the data points to N clusters containing 1 data point each.

In simpler words you start with all points being their own independent cluster and each iteration you **merge the 2 closest points together** into a new cluster. Keep going until you have the amount of clusters you want or until each point belongs to the same cluster.

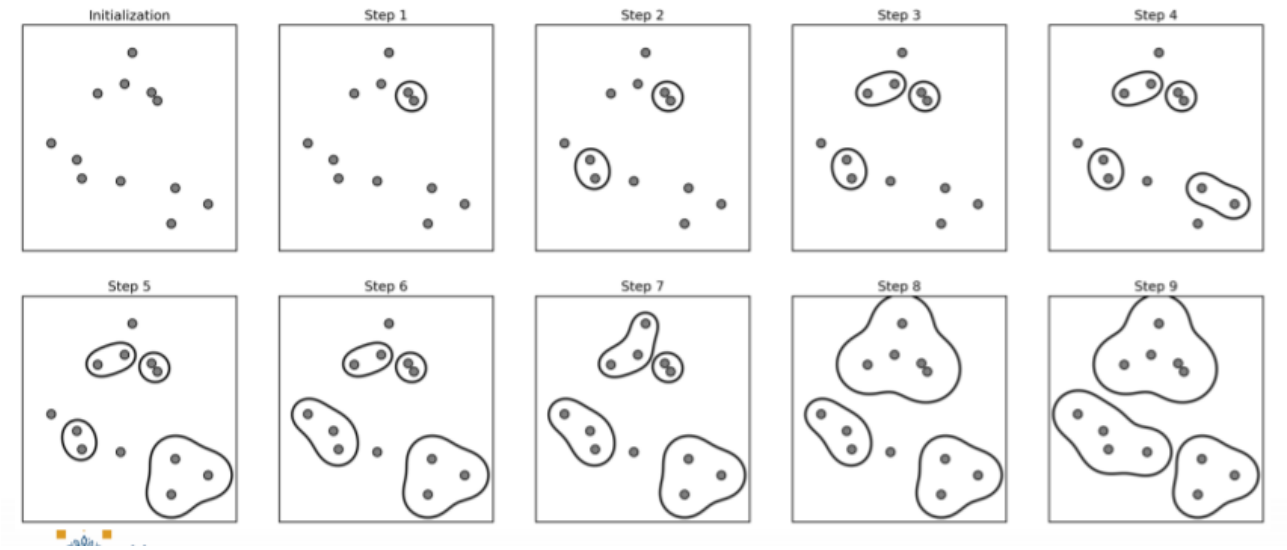


Figure 71: Hierarchical clustering in action

You can also visualize this as a tree were each node of the tree is a merger of the nodes up to that point forming a new cluster.

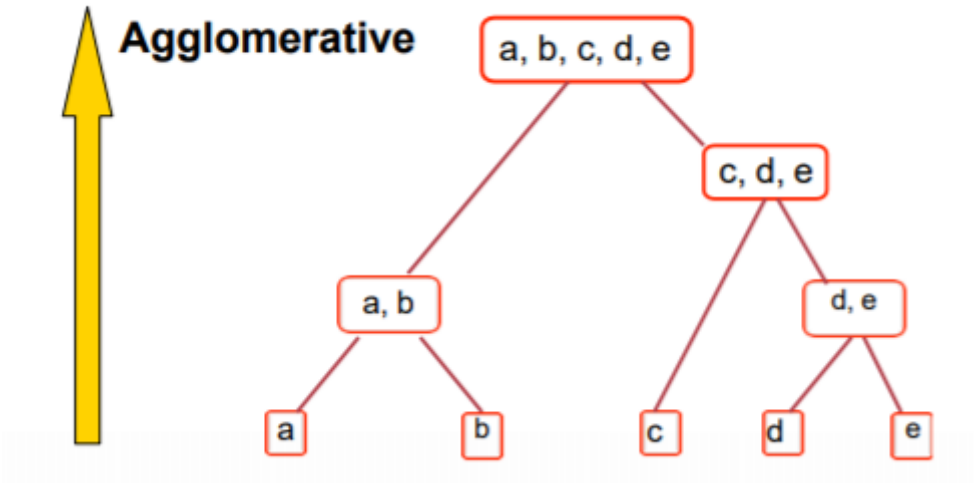


Figure 72: Hierarchical clustering

You can do even better by making special plots called **dendrogram** that visualize the arrangement of the clusters produced.

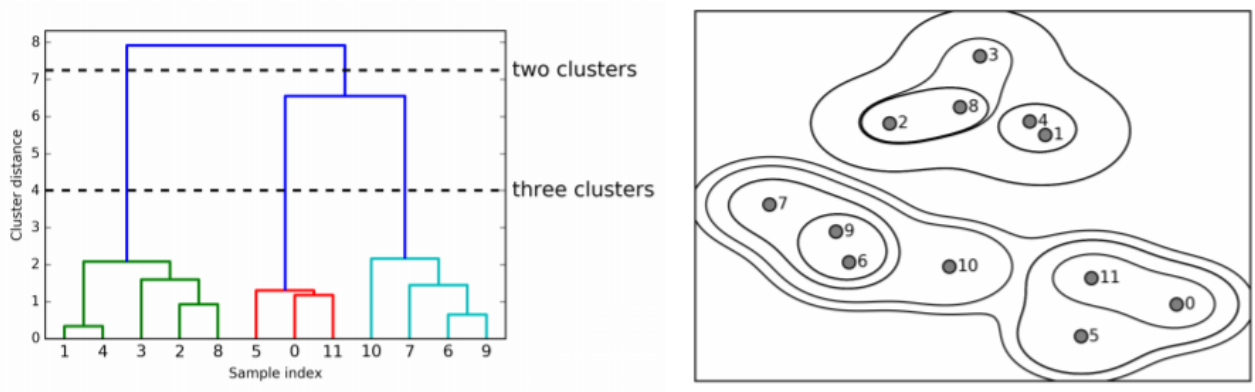


Figure 73: A dendrogram

This helps to decide where to stop merging and also keeps a nice history of merges.

There are 4 ways to decide on how to cluster:

- **Single linkage clustering**: Distance between pairs of points
- **Complete linkage clustering**: Distance between the farthest pairs of points
- **Average linkage clustering**: Mean distance of all mixed pairs of points
- **Ward clustering**: Minimises sum of squared differences within all clusters, this leads to more equal clusters (default in sklearn)

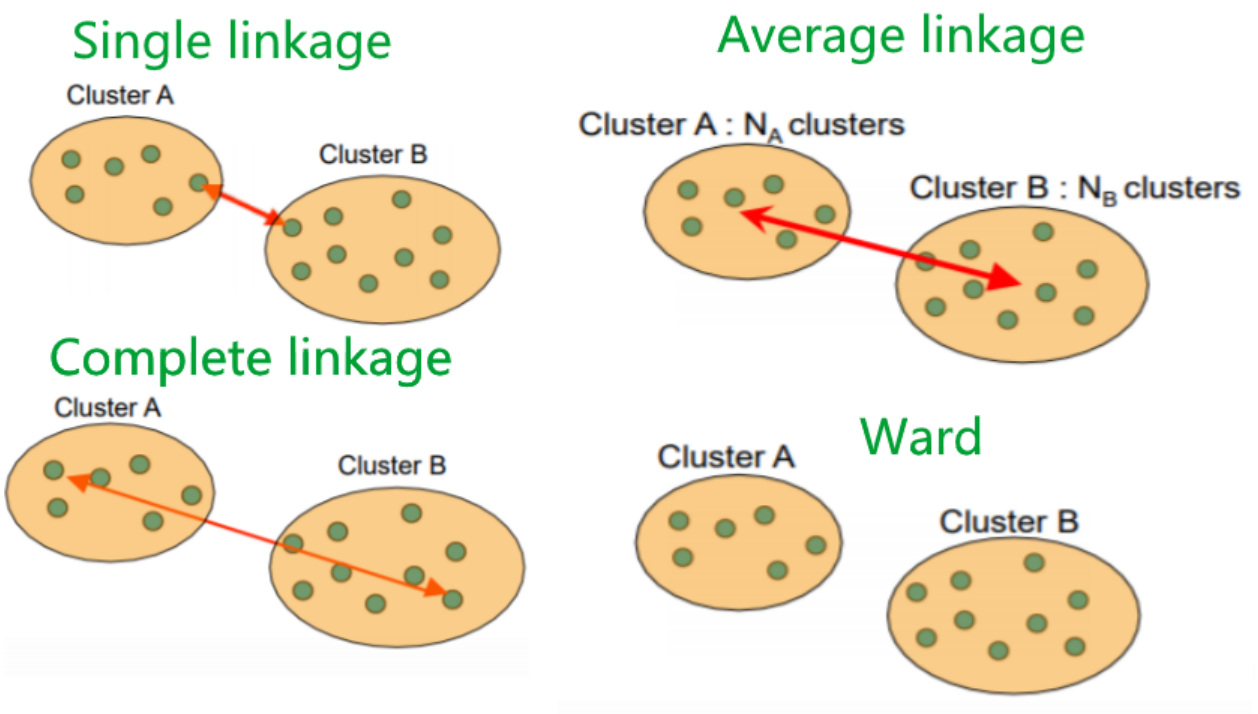


Figure 74: Different agglomerative cluster techniques

Here are different results when you use the different clustering techniques:

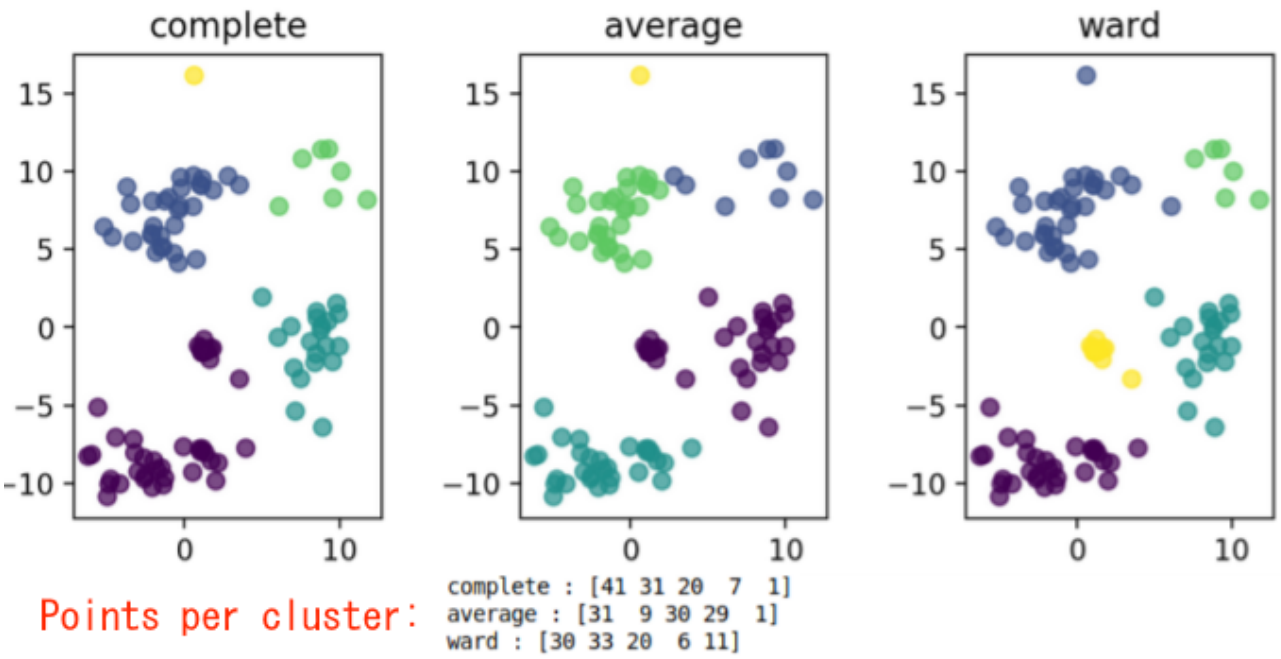


Figure 75: Results of different hierarchical clustering techniques

10.3.1 Pros and cons

- Can restrict to input topology given by any graph for example neighborhood graph
- Fast with sparse connectivity
- Some linkage criteria can lead to very imbalanced cluster sized (can be pro or con)
- Hierarchical clustering gives more holistic view, can help with picking the number of clusters for other techniques

10.4 Density based clustering methods (DBSCAN)

Density-Based Spatial Clustering of applications with noise (DBSCAN) is a clustering technique were you divide the datapoints into 3 groups based on hyperparameters called **Density** this is the number of points within a specified radius r. r is usually called **epsilon**. The 3 groups are:

- **Core points**: Points with more than a specified number of points (min_samples) within epsilon. This includes samples inside the cluster
- **Border points** Points with fewer than min_samples within epsilon, but is in the neighborhood of a core point. So one of the points that makes another point a core point
- **Noise points**: Any point that is not a core or border point.

So if a point is close to a lot of other points, it is a core point. If a point contributed to making a point a core point, it becomes a border point (if it isn't a core point already). All the other points are noise points.

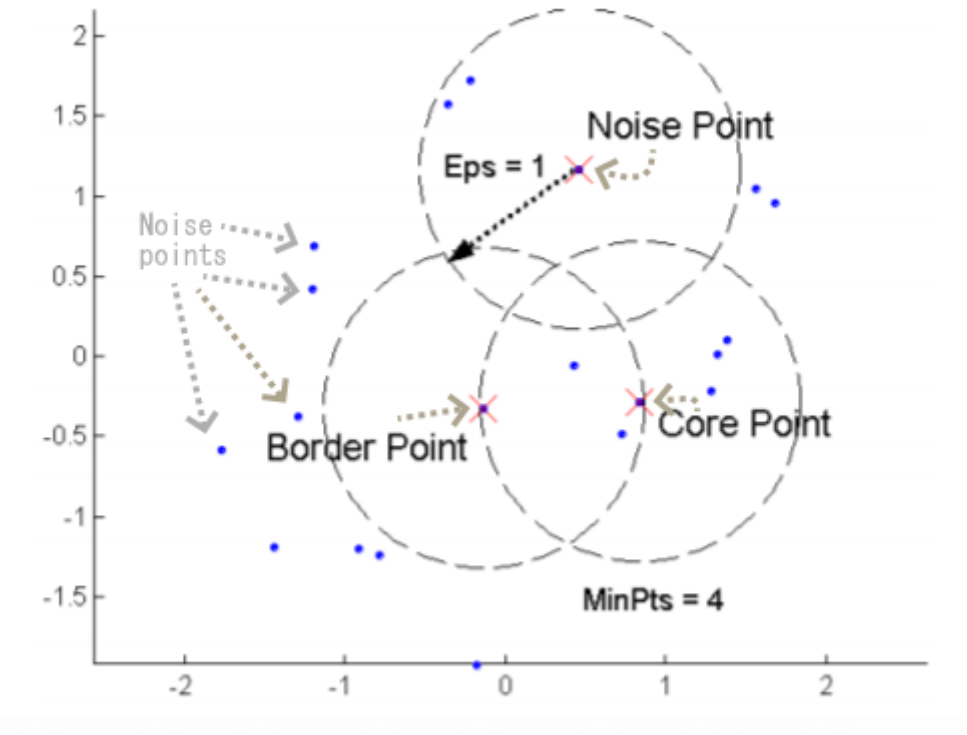


Figure 76: DBSCAN

10.4.1 DBSCAN Algorithm

1. Start with a core sample/point
2. Recursively find neighbors that are also core points and add them to the cluster
3. Also add samples within epsilon that are not core points (but don't recurse) this means adding the border points but don't search further with those points
4. If no other points are "reachable", pick another core point to start a new cluster.
5. Repeat till you went through all core points
6. Remaining points are labeled outliers/noise points

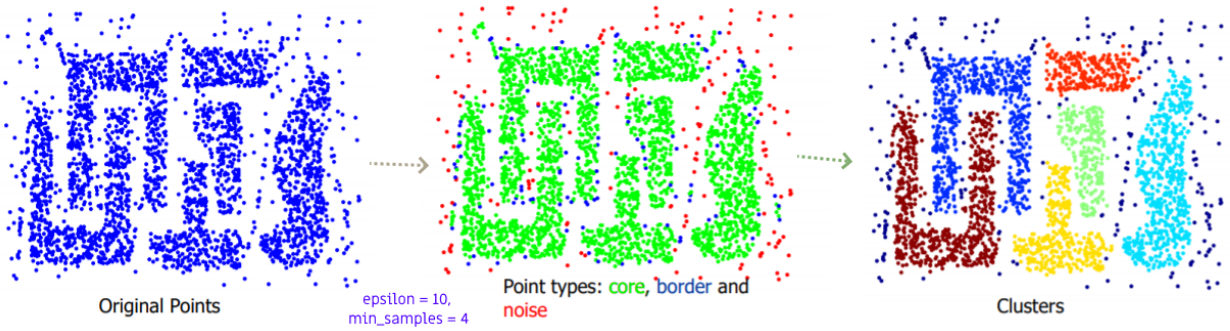


Figure 77: DBSCAN in action

10.4.1.1 Pros:

- Allows complex cluster shapes
- Can detect outliers

  - Good for spam, Maybe you are more interested in the noise points

- You don’t need to say how many clusters you want!
- Needs only two parameters to adjust,
- Can learn arbitrary cluster shapes

10.4.1.2 Limitations:

- Varying densities because your cluster size varies
- Is slower with High-dimensional data
- Epsilon is hard to pick (can be done based on number of clusters though)

All with all this is a wonderful technique and can even predict hard clusters like this no problem.

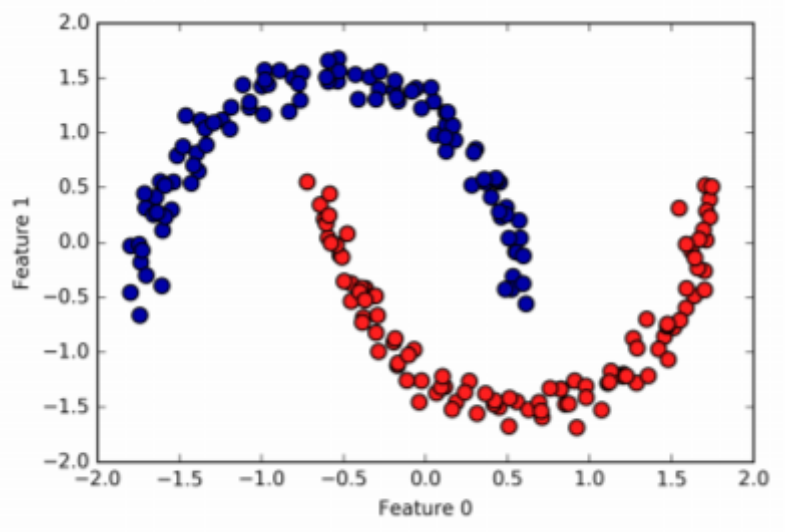


Figure 78: Hard to predict cluster

10.5 Mixture methods

The idea of a mixture model is that you want to find a generative model. This means finding the parameters for a function p(x).

This method makes a couple assumptions

- Data is mixture of small number of known distributions
- Each mixture component distribution can be learned “simply”
- Each point comes from one particular component

The goal is to learn the component parameters and weights of the components.

$$p(\mathbf{x}) = \sum_{j=1}^{k} \pi_k p_k(\mathbf{x}|\theta)$$

Figure 79: P formula mix models

10.5.1 Gaussian Mixture models

Here you assume that each component is created by a Gaussian distribution and there is a multinomial distribution over the components.

Here is the big scary formula:

$$p(\mathbf{x}) = \sum_{j=1}^{k} \pi_k \mathcal{N}(\mathbf{x}, \mu_k, \Sigma_k)$$

Figure 80: Gaussian mixture model formula for p(x)

The parameters are learned by a process called **non-convex optimization**. You make a lot of distributions for each potential cluster and then merge them together based on how good they are. Each distribution you generate is called a **component**. This is done with an algorithm called the **EM algorithm**. The EM algorithm assigns points to the components and then calculates the mean and the variance. You initialize this method with K-means, and then you do random restarts.

This gives you different distributions each time. The idea is that you merge the distributions you got into a final distribution.

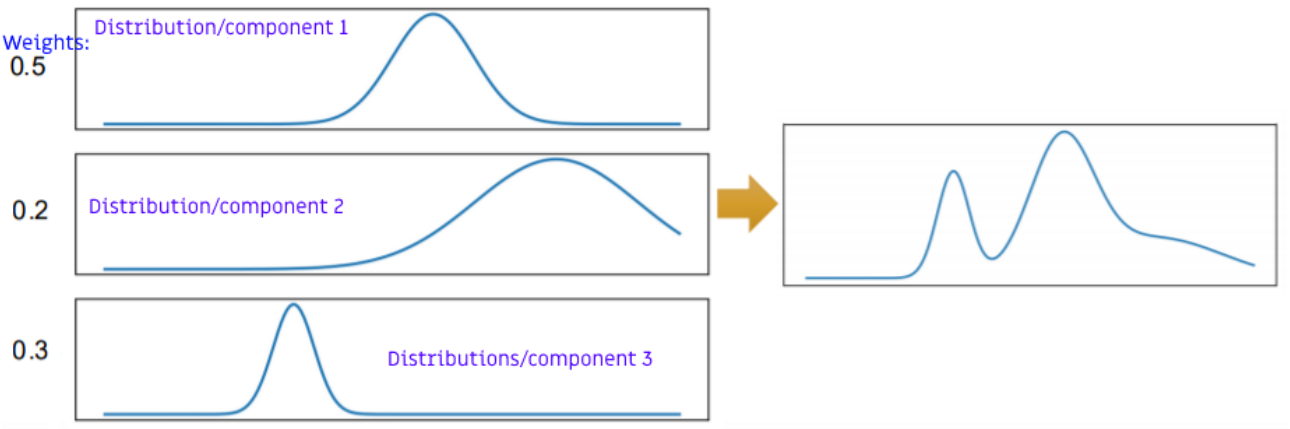


Figure 81: Merging found distributions

This model needs to know the amount of clusters that you want. The nice thing with mixture models is that you can get a variance for each cluster.

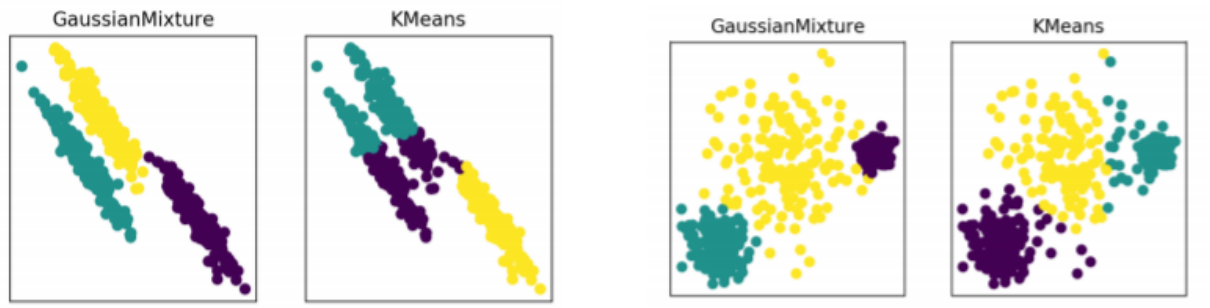


Figure 82: K-means vs GMM

10.6 How to evaluate unsupervised learning models

This is a problem because unlike with supervised learning there is no source of truth. There are a lot of ways to this, but the 2 ways highlighted in the course are:

10.6.1 Elbow plot

This is a way to find out how many clusters you should ask for. This computes the sum of squared distances (SSE) between data points, and their assigned clusters centroids. If you do this you will get a graph like this:

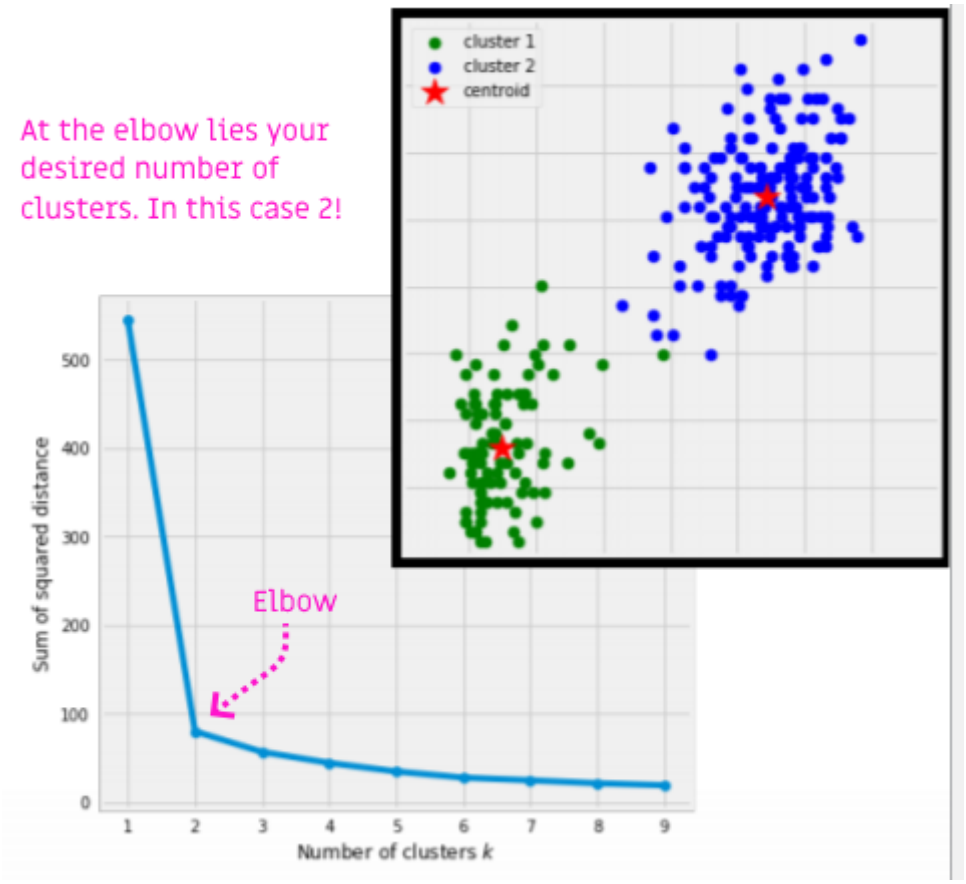


Figure 83: Elbow plot

Pick the desired number of clusters at the spot where SSE starts to flatten out and forming an elbow.

10.6.2 Silhouette Coefficient

Another way to find out how many clusters to ask for is silhouette coefficient. For this technique the idea is that for each sample:

1. You compute the average distance from all the data points in the same cluster (ai)
2. Compute the average distance from all data points in the closest cluster
3. Then you compute the silhouette coefficient:

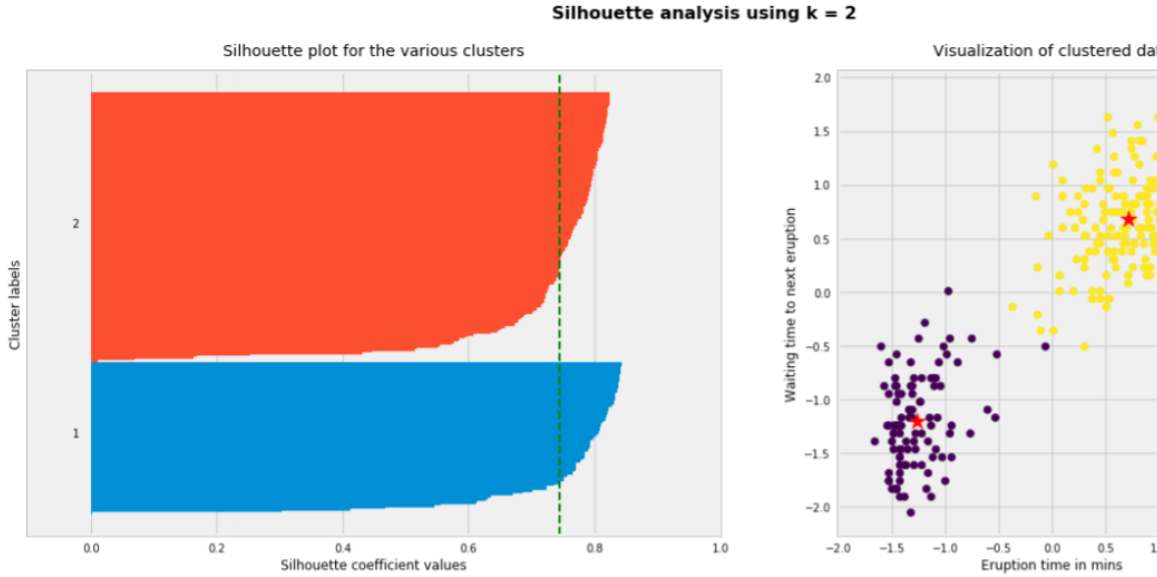
$$S(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}$$

Figure 84: Formula for silhouette Coefficient

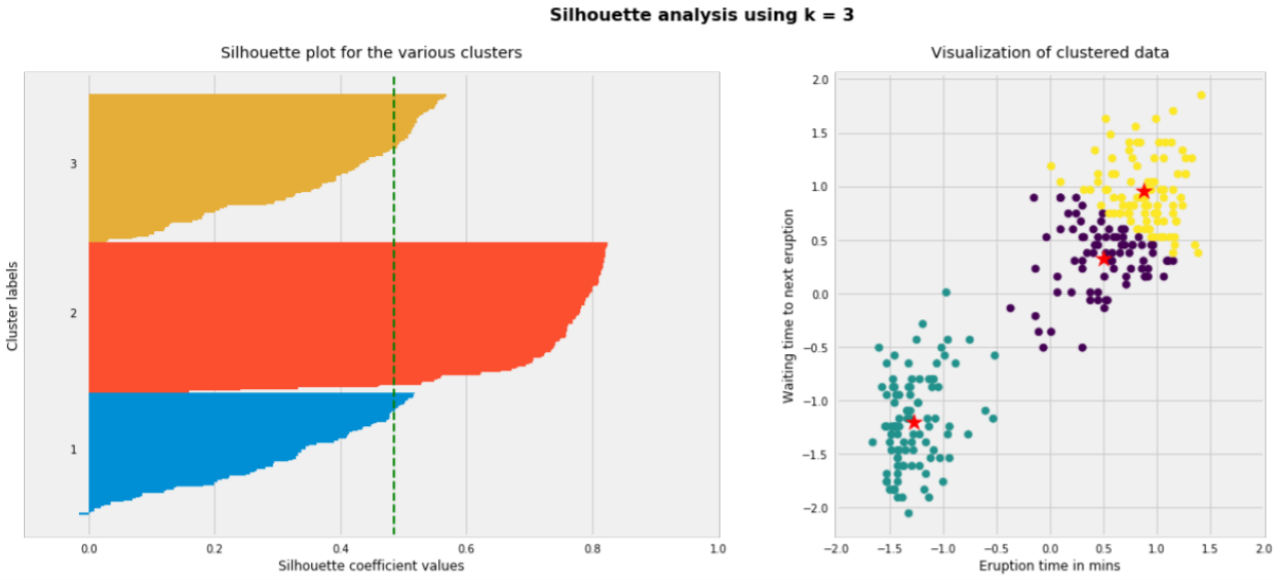
You do this for every datapoint. So every datapoint gets a silhouette coefficient score.

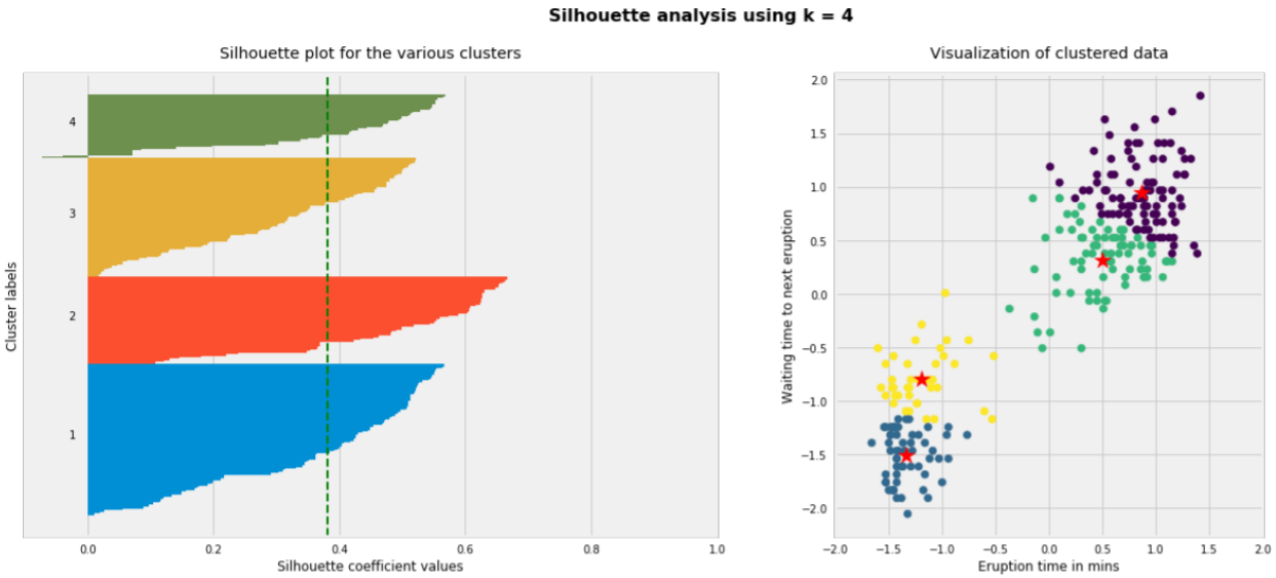
The result will be in the range of [-1, 1]. This is what the results mean:

- If 0: The sample is very close to the neighboring clusters
- If 1: the sample is far away from the neighboring clusters
- if -1: The sample is assigned to the wrong clusters



10.6.2.1 Examples:





The thickness of the plot indicates how many points are in that cluster. Every horizontal line is the silhouette coefficient of one point in the cluster.

The mean silhouette coefficient is highest for n_clusters = 2 so this is the amount of clusters you should go for with this example. You can also see this because this is the only version where all clusters are above average.

This summary was completely written in markdown it is easy to write, and you can make beautiful pdfs out of it with a tool named Pandoc. A great tool to convert documents to other types of documents.

This summary was lovingly writen by Quinten Cabo Good luck!