

Summary written by Quinten Cabo

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 so I can update the summary :)

Support

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

[PayPal](#) or [Tikkie](#)

BTC: 1HjFv4NYiTwxdcnNpPEzaPY8vvYWXQnDgx

ETH: 0xc0b430594A957A6A38203F45bd91f5d3568a39c6

ML SUMMARY

WHAT EVEN IS MACHINE LEARNING?

Well machine learning is 2 words. **Machine** + **learning**. You have the machine and you need to make it do learning. So the next logical question is what do we mean with learning? We mean methods that can extract knowledge from data. We do this with **techniques from statistics** and **optimization**. If done right the machine can use what it has learnt before to predict things.

The different strategies of extracting knowledge from data are called **algorithms**. The machine learning algorithms can be divided into 3 categories.

It seems like for the midterm we only need to know the supervised techniques so that is what this summary will talk about but you do need to know the other categories of machine learning (ML) so here they are shortly described:

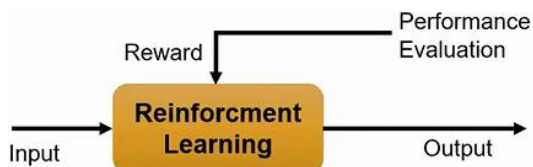
UNSUPERVISED LEARNING

With this type of learning you don't give the machine any direction only a bunch of data. The machine then has to find **groups**, clusters or patterns in the data and those become the output. So the difference with supervised learning is that there are **no examples** and you nor the machine initially really know what the output should be. You could use this for example for outlier detection.



REINFORCEMENT LEARNING

This is the one autonomous systems talked so much about. Remember with a **policy** and **utility** function and always in some sort of **environment**. Here you try to make the machine learn a policy by trying to do something in the environment. After the machine has attempted what you want to its best efforts you give the machine either a positive or negative **performance evaluation**. Based on this the machine adjusts itself for the next attempt. Repeat this a lot and the machine might start showing sensible behaviour. The evaluation can be given by either a human or another machine.



THERE ARE EVEN MORE TECHNIQUES

Semi supervised learning which is a combination of supervised and unsupervised learning and **active learning** which I think is when the model keeps learning even after training.

SUPERVISED LEARNING

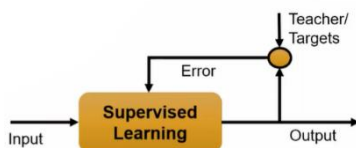
Here the idea is that you supervise the machine. This means giving the machine a **training set** that is made up out of **input and output pairs**. In practice this looks like $D = \{X_i, Y_i\}$ where X is one or more inputs or the **features** or **attributes**, and Y is the **label**, **class**, target or **output** given to this set of inputs. All these features with a label together make up 1 **datapoint** or **instance**. A lot of these datapoints together make up your **dataset**.

CONCEPT	EXAMPLE
Instance	"Nijntje (Miffy)"
Label/Class	"Rabbit" / "1955"
Features/attributes	color, # of ears, has a teddy bear
Feature values	"white", "2", "yes"
Feature vector	("white", "2", "yes")



The machine uses this dataset to build a **model**. Using this model the machine makes predictions about new never before seen data. This is usually done by comparing the new data to the training data in a clever way to make the best guess. How would you go about doing this?

The more simpler of these algorithms boil down to just comparing the X of the new data to the X s in the dataset. This will yield some group of datapoints as relevant to your new data. Then look at the Y s of these datapoints to make a prediction about the new data.



The more complex techniques take an approach involving **error**. The idea is that during training the machine makes some educated guesses after which the machine peeks at the real answer (Y) to see how wrong it was. This is measured in error. Then the machine proceeds to **minimize error by adjusting the model parameters** for example by using fancy math like the derivative and gradient decent.

When you have your model set up and the machine is able to predict the training data with a low amount of error you try it on the test set. What is the test set?

TRAINING AND TESTING DATA

Before we can apply our model to actual new data from the wild, we need to know if we should trust its predictions. Unfortunately we cannot use the data we used to build the model to evaluate this because our model can always simply remember the whole training dataset and will therefore always predict the correct output for any point in the training set.

So after training **we want to know if our model will generalize** well. In other words, whether it will also perform well on new data. To do this, we show it new data for which we have labels. This is also called **cross validation**. But how do we get this data? By splitting the labelled data we have collected already into two parts. One part of the data is used to build our machine learning model, the **training data** or **training set** usually is 75%. The rest of the data will be used to assess how well the model works called the **test data**, **test set**, or **hold-out set** (usually 25%). There is a function that does this splitting for you.



Supervised learning often requires a lot of human effort to build the training set, but afterward automates and often speeds up a boring or infeasible task.

There are 2 major types of supervised learning problems these are:

CLASSIFICATION AND REGRESSION

Classification is where you try to predict if a certain input belongs to a certain group or class. Or in other words trying to **label an input**. These types of problems have a predetermined set of labels to choose from. The prime example of these types of problems is labelling email as spam or not spam or labelling images of either cats or dogs 🐱🐶.

We can classify the classification problems in 2 classes: binary classification and multiclass classification.

Binary classification is a problem with only 2 possible labels and **multiclass classification** has more than 2 possible labels. Furthermore with binary classification we sometimes speak of 1 negative class and 1 positive class. Which is which depends on the domain.

Regression on the other hand is where you try to predict a continuous output. An example of such a problem is predicting the yield of your tomato plantation based on inputs as how many people work on your farm, what nutrients did you give your plants or what type of music did the plants listen to.

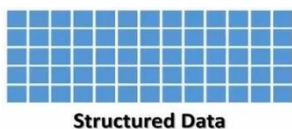
The inputs for either of these types of problems can either be categorical or continuous. **The type of problem is determined only by the output**. If you find yourself having troubles deciding what type a problem is just ask yourself: "Is the output continuous?" If the answer is yes, then you have regression otherwise it's a classification.

2 TYPES OF DATA

You have two types of data: **Structured data** and **unstructured data**.

STRUCTURED DATA

This is data that is nicely organised in a standard format by for instance using a schema. Think of xml, json, yaml, or an SQL (**structured** query language) database with a schema or any other type of database for instance a graph (those are great). Structured data is organized and quite standardized. **You know what you are going to get, and it is easy to analyse.**



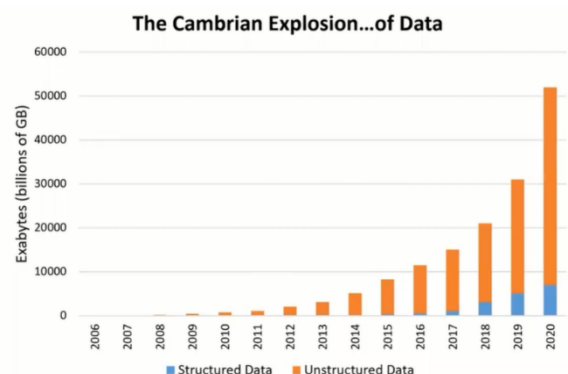
UNSTRUCTURED DATA

This is every other type of data. Think plain text, emails, pictures, videos, memes, pdfs, word documents and I can keep going. These types of files or data can really contain just about anything and you don't know what you are going to get. Most data that are (yes data is plural) out there is unstructured as can be seen in the graph below.



(Image source: Sherpa Software)

This graph shows an estimate of how much data there is in the world. What we can learn from this graph as Dr Sharon put so nicely "data is cheap and abundant, but knowledge is expensive and scarce." This is where machine learning comes in to extract knowledge from unstructured data. **Using machine learning to extract knowledge from unstructured data is necessary because of the sheer amount of data out there and its unpredictability.**



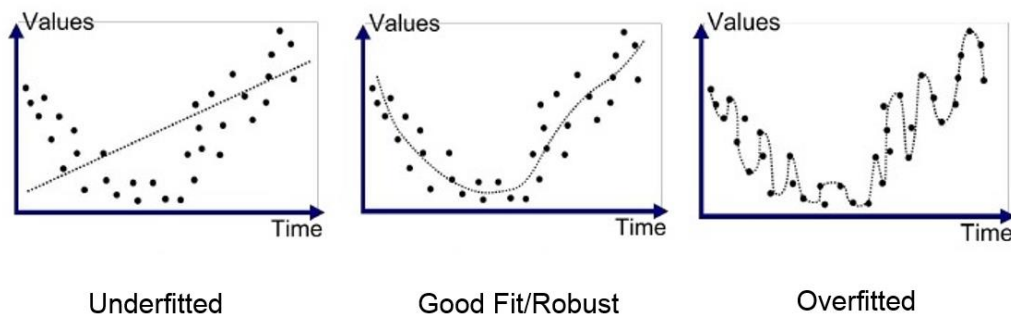
it

OVERFITTING AND UNDERFITTING

Usually we build a model in such a way that it can make accurate predictions on the training set. If the **training and test are similar, we expect the model to also be accurate on the test set**. This can go horribly wrong.

When trying to build a model you will run into overfitting and underfitting problems. **Overfitting** happens when you build a model that corresponds **too closely to the training data** to be generalizable to new data.

You are trying to take each train point into account too much. **Underfitting** is the opposite when you are not taking enough into account from your training data and ignoring a lot of points. Or basically **making too simple of a model**. The picture below should make it clearer.



You can actually see if you are overfitting from your test scores.

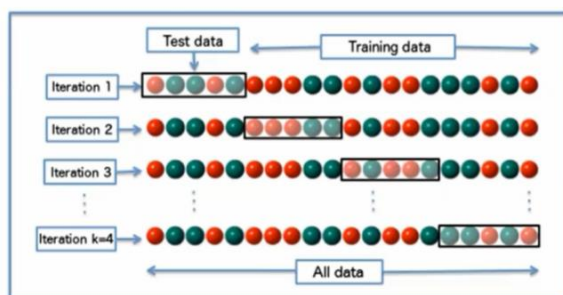
HOW TO SEE WHEN YOU ARE OVERFITTING OR UNDERFITTING

If you have a **high accuracy score on the training data** and a **low accuracy score on the test data** so something like Training set score: 0.95, Test set score: 0.61 then that is a clear sign of **overfitting**. Your model performs very well on training data but is probably too much tailored towards it as new data has quite bad performance. Also **the more complex a model the more chance of overfitting**.

If you have a **low accuracy on the training and test score but they are close together you are probably underfitting** (Or you just have a bad method). An example is Training set score: 0.67, Test set score: 0.66. This is not so good accuracy, **but the scores are close together**. This means we are probably underfitting.

To avoid underfitting and overfitting ever reaching production we use **cross validation** with a training set and test set. Based on the result of these you can adjust your model/hyper parameters. There are multiple cross validation techniques mentioned in the first lecture. **K-fold cross validation** and **Leave one out**.

- K-fold cross validation
- Leave one out (K-fold cross validation to the extreme)



The idea of **k-fold** cross validation is that you use a different part of the dataset as test and training data each time to validate and evaluate the model. You separate each part of the dataset equally to make all parts be test set once to avoid any potential accidental bias in prediction score. K is the number of iterations you want to do so in the picture K is 4. They leave $N/4$ for testing each iteration. **Leave one out** takes this to the next level and only leaves 1 out and uses the rest for training. This requires N iterations.

THE PYTHON PACKAGES

This section will describe the packages used in the course.

I have read a lot of people in the WhatsApp chat saying they are unfamiliar with the packages. I have good news for you. You are not! We have covered what are basically the R versions of these packages in statistics already.

NumPy is for making fancy lists → Just a more fancy and faster version of lists (called arrays) you have seen already with `c()` from R + more math methods.

Matplotlib is for making fancy plots and graphs → `ggplot` or just `plot()` from R.

Pandas is for making organizing data in rows and columns → dataframes (or just an excel sheet) in R.

Scikit-learn just easy to use classes for ML techniques → Like the `lm()` function in R but more of them.

SciPy just more fancy math functions → You could see this as more math packages, don't be scared.

Mglearn a utility package for learning about ML → Made by the people that wrote the book

None of these packages are standard though [you need to install them with pip](#)

MATPLOTLIB

This package is for making graphs. I will show here how this package is quite like ggplot in R if you feel like you want to learn about it there is actually a secret strategy you can follow called: “follow the tutorial from the people that made the package in the first place”

<https://matplotlib.org/stable/tutorials/introductory/pyplot.html> This secret strategy usually works with anything remotely popular.

Ok so let's start by import what we use in the notebooks.

```
import matplotlib.pyplot as plt
```

This gives you the plt object this is actually a matplotlib.pyplot but it is usually imported as plt. See this as a scratch book or paper where you can plot whatever you want add things to the plot. Then if you are done adding things you can call plt.show() to flush what you currently have to the screen in a nice plot. This will in turn also clean the plt so you start a new plot. So what can this plt object do?

```
X, y = [6, 7, 8, 9, 0], [1, 2, 3, 4, 5]
plt.figure() # This makes sure your plt plot scratchpad is clean
plt.plot(X, y) # Add data to a plot, X and y should be lists or list-like
plt.show() # To show the plot call plt.show()
```

This is quite a lot like ggplot so far but matplotlib is better because you can keep adding to your plot before you show it! It is better in steps instead of creating massive functions inputs at once.

```
W, z, b, a = [10, 20, 30, 40], [40, 30, 20, 10], [3, 6, 9, 12], [5, 10, 15, 20]
plt.figure()
plt.plot(X,y, label="low numbers") # Add some data with a label
plt.plot(W,z, label="high numbers") # Add more data to the plot
# these two plots will be combined + more until you call plt.show()
plt.scatter(X,y, label="high numbers") # Add a scatter plot to the plot
plt.show() # To show the plot call plt.show()
```

Pretty nice. We can even enable things like xlabel, ylabel and legend. The legend is based on the label that you gave the plot you added to plt.

```
plt.figure()
plt.plot(X,y, label="low numbers")
plt.plot(W,z, label="high numbers")
plt.scatter(X,y, label="high numbers")
plt.xlabel("What a beautiful x label!") # add labels to the plot
plt.ylabel("What a beautiful y label!")
plt.legend() # turn on the legend by turn calling .legend
plt.show() # To show the plot call plt.show()

# Also plt is now clean we can just start to plot new data
plt.plot(X,y, label="low numbers")
plt.show()
```


Instead of calling `plt.show()` every time you are done with a plot you can also just do `plt.subplot()` to start a subplot!

```
plt.figure()      # Start a new figure
plt.subplot(211)  # Start a new subplot in the figure The number are size
plt.plot(X,y, label="low numbers")
plt.subplot(211)  # Start a new subplot in the figure
plt.plot(X,y, label="low numbers") # Add data to the new subplot
plt.show()
```

At this point it is good to say that `plt.figure` actually does not clean the `plt` it just starts a new plot. This means you can actually add to older plots! How does this work?

```
plt.figure(1)      # the first figure with id: 1
plt.subplot(211)   # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)   # the second subplot in the first figure
plt.plot([4, 5, 6])

plt.figure(2)      # a second figure
plt.plot([4, 5, 6]) # creates a subplot(111) by default

plt.figure(1)      # figure 1 current; subplot(212) still current
plt.subplot(211)   # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
```

You can do a lot more, but I hope this makes the basics clear and removes the feeling I at least had where you just copied the code from the earlier box because that one worked.

Just some extra information. We use the Jupyter notebooks. This is nice if you want to run small bits of code at the same time. These notebooks run on IPython a different version of python. Therefore you have the `%matplotlib` thing that you maybe haven't seen before in normal python or Cpython. But you don't actually have to use `%matplotlib`. The only thing it does it automatically call `.show()` on any plot that is not shown yet at the end of the notebook block. Personally I think you would want to control when to show `plt` when you want.

NUMPY

You can find the real tutorial here: https://numpy.org/devdocs/user/absolute_beginners.html NumPy is a package with fancy less-resource-costing lists called arrays. Actually, you could say these arrays are less fancy than lists because you can only store one type of data in them. But they make up for this by having lots more methods with the arrays. This is how you create arrays.

```
import numpy as np      # People usually import numpy as np idk why
a = np.array([1,2,3,4,5,6]) # array that has [1,2,3,4,5,6]
# You can say which type you want your items to be casted to
a = np.array([1,2,3,4,5,6], dtype=float) # Gives [1. 2. 3. 4. 5. 6.]

a.append(33)            # This is not how you append
a = np.append(a, [33,44,55]) # This is how you append

a = np.arange(4, dtype=float) # make arrays with a range 6 [0. 1. 2. 3.]
a = np.zeros(4, dtype=float)  # make an array with zeros [0. 0. 0. 0.]
```

Like I said the arrays might seem a bit more limiting as you can only store one type of data in there but this makes them much faster and allows to do math operations on the whole array!

```
a = np.array([1,2,3,4,5])
b = np.array([5,5,5,5,5])
c = a + b # Gives [6, 7, 8, 9, 10]
d = a * b # Gives [5, 10, 15, 20, 25]
e = a / b # Gives [0.2 0.4 0.6 0.8 1. ]
f = a + 10 # Gives [11, 12, 13, 14, 15]
g = a @ b # Dot product operator Gives 75
```

You can index arrays like normal with the normal [index] syntax.

```
a = np.array([1,2,3,4,5], dtype=float) # [1 2 3 4 5]
a[0] # Returns 1
a[2] # Returns 3
a[2:4] # Returns [3,4,5]
```

	data	data[0]	data[1]	data[0:2]	data[1:]	data[-2:]	
0	1	1		1			0
1	2		2	2	2	2	1
2	3				3	3	2
							3

You can also make a selection of your array by passing a list of Booleans.

```
a = np.array([1,2,3,4,5], dtype=float) # [1 2 3 4 5]
a[[True, True, True, False, False, False]] # Returns [1 2 3]
b = [True, True, True, False, False, False]
a[b] # also returns [1 2 3]
```

Do you have to type out or generate these lists of Booleans? No luckily not! You make comparisons on your arrays. This returns a new array of Booleans for every position with the outcome of the comparison for you! Super great 😊

```
a = np.arange(7, dtype=float) # [0 1 2 3 4 5 6]
a > 3 # Returns [False, False, False, False, True, True, True]
# This returns a new array of Booleans for every position with the outcome of
the comparison. This is operator overloading at play in the wild

# Combining all of that you can make sub selections of arrays like so
B = a[a > 3] # returns [4, 5, 6]
```

The big thing is you can have nested arrays. A nested array is an array of arrays. And you can still do all the math operations with them that's where the real power comes as they work at all levels.

```
A = np.array([[ 0, 1, 2, 3, 4],
               [ 5, 6, 7, 8, 9],
               [10, 11, 12, 13, 14]]) # This is a nested array
A += 5 # Does operation at each item in the nested arrays
''' [[ 5  6  7  8  9]
     [10 11 12 13 14]
     [15 16 17 18 19]] ''' # These are 2D arrays
```

Nested arrays also have properties with info and more about the array I assume there are more.

```
A.shape # Returns (3, 5)
A.size  # Returns amount of items 12
A.ndim  # Returns amount of dimensions 2
A.T     # A transposed version of the array
```

You can even turn 1d arrays in 2d or any dimension with the reshape function.

```
A = np.arange(9).reshape(3, 3)    # Returns array([[ 0,  1,  2],
                                   #                    [ 3,  4,  5],
                                   #                    [ 6,  7,  8]])

A @ A                             # Returns array([[ 15,  18,  21],
                                   #                    [ 42,  54,  66],
                                   #                    [ 69,  90, 111]])
```

You can take these dimensions as far as you want! This last section is about selecting from higher dimensional arrays. You still have the normal [index] but you can use a ,(comma) to go down a level.

```
H[10] # Select whatever is in index 10 of the first dimension
H[10, 3] # Go down a level. Select the third thing from the tenth thing
H[10, 3, 2, 3] # You can keep going with this
H[10, :, 2, :] # Use : to select everything at a certain level
H[10, 2:5] # You can do slicing as well
H[10, 2:5, :-2] # At multiple levels!
```

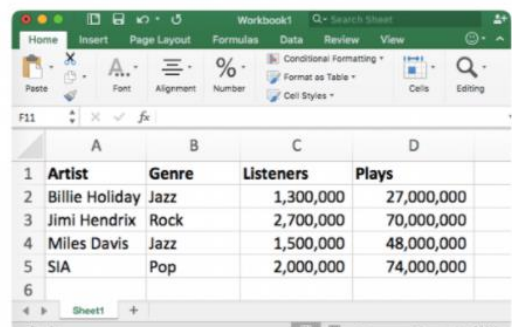
That concludes my explanation of numpy I also recommend reading

https://numpy.org/devdocs/user/absolute_beginners.html

PANDAS

This is like dataframes in R you can use it to load csv data into dataframes, we haven't used it too much so far.

music.csv



→

pandas.read_csv('music.csv')

	Artist	Genre	Listeners	Plays
0	Billie Holiday	Jazz	1,300,000	27,000,000
1	Jimi Hendrix	Rock	2,700,000	70,000,000
2	Miles Davis	Jazz	1,500,000	48,000,000
3	SIA	Pop	2,000,000	74,000,000

https://pandas.pydata.org/docs/user_guide/index.html

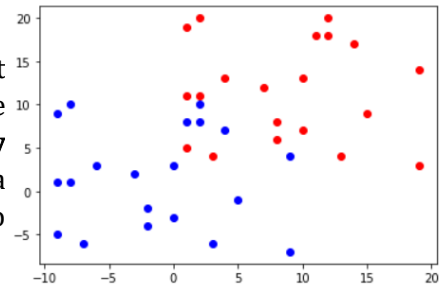
MACHINE LEARNING TECHNIQUES

Then what we all have been waiting for the supervised machine learning techniques.

K-NEAREST NEIGHBOURS

[illegible]

Then if you want to predict the label for a new unknown point. Just look at the points **closest** to your new point, **the neighbours**, if most neighbours are blue then we say the new point is also blue otherwise it is red. **But how many neighbours should we take into account?** Well **this is K**. K is a **hyperparameter**. K says how many neighbours you should take into account.



A **hyperparameter** is a parameter whose value is used to control the learning process.

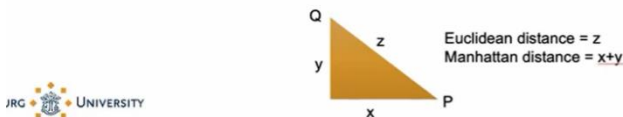
- **Euclidean distance** (straight line)

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- **Manhattan distance** (distance between projections on the axis)

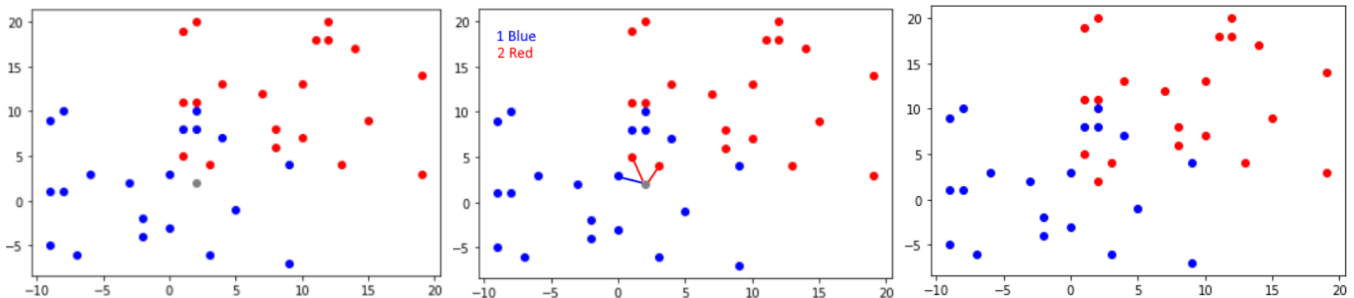
$$d = |x_2 - x_1| + |y_2 - y_1|$$

- Difference between Euclidean and Manhattan distance



So how do we define closest points? This is actually up to you. You could use different definitions of distance, but we use **Euclidean distance** if you are on a grid you use **Manhattan distance**.

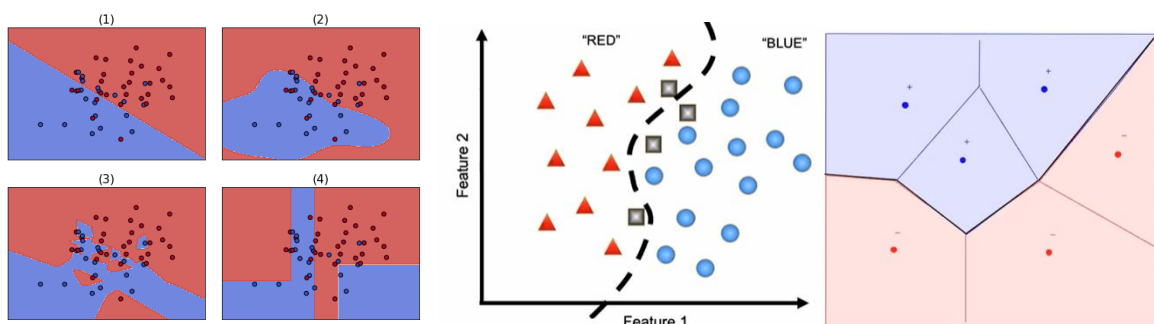
So let's say we take $K = 3$, and we want to predict the grey point below. Ok so $K = 3$ so we count the 3 closest neighbours. 1 blue and 2 red so we say the new point is red. See the example visualization ↓.



A computer can't just look at a cloud of points to see the closest ones (if only) so a computer takes the following approach.

1. Make an array of the distances from the new point to every other point.
2. Sort the list ascending.
3. Take K first elements of the array into a new array.
4. Count which label shows up the most in the new array this is the prediction.

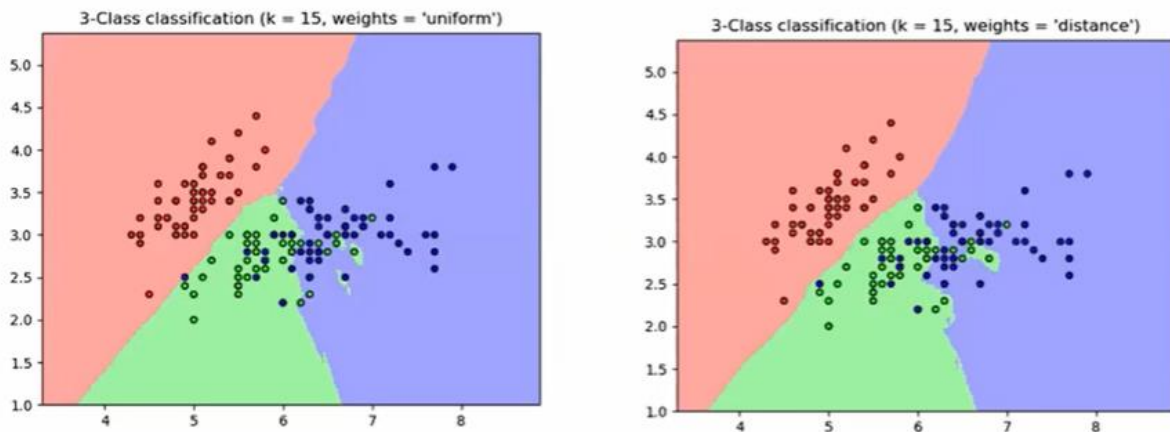
But because our training data is static, we can actually know what label will be given to a position with a K. This is called the **decision boundary**. So if a point falls in the red area it is predicted as red.



So for classification the model trained from the data defines the decision boundary. Although there is not a lot of training going on here, we just pick a K. If you increase K, then the decision boundary gets more smooth (less complex) because it is influenced by more points.

WEIGHTS

One thing you might say is what if I do K nearest neighbours with $K = 11$. **The first 4 neighbours are blue and are quite close and the next 7 points are all red but far away.** The new point would be classified as red, but it **might make more sense to classify it as blue** because those points were closer. The solution to this is use **weights**. If you have **uniform weights** all points count equal. This is what we did so far. If you have **distance weights**, then you scale how much a point matters down with distance from the new point.



There are multiple ways to do this weighting.

- **Inverse distance weighting** each point has a weight equal to the inverse of its distance to the point to be classified so weight is $1/D$
- **Inverse of the square of the distance** so weight is $1/(D)^2$ so here you punish distance more
- **Kernel functions** more complicated functions (Gaussian kernel, tricube kernel)

With uniform weights if K is the number of points you have then you just look at which points appear the most (a majority function). With distance weights this is not the case anymore.

WHAT K TO PICK?

K and complexity of the model have a negative relationship so the higher the K the less complex your model becomes. If K is too low, it will get all the points in the training set right, but overfitting happens.

K too low → Danger of overfitting, high complexity

K too high → Danger of underfitting, low complexity

You also want to pick an **odd value of K** to avoid having ties.

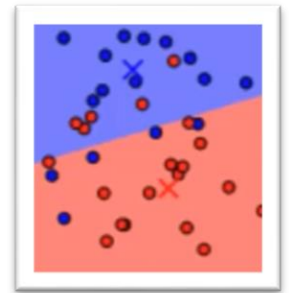
Ok but you haven't told me what K to pick yet...

A rule of thumb is $k = \sqrt{n}$. So just take the square root of your number of points. But K is a matter of trial and error.

So far these graphs have only show predictions with 2 features but if you have a datapoint with 10 features you can still calculate the distance between those 10 features. Add them up or something else. So **the k-nearest neighbours technique scales/works with more dimensions**.

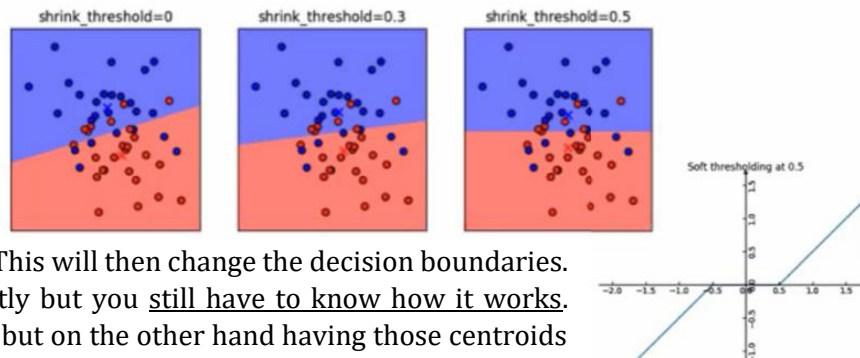
NEAREST CENTROID

For doing a **prediction you need to calculate the distance between all points and the new point**. This can take quite long if you have a lot of data with high dimensionality. So you could also calculate the centre/mean of all the points of a cluster and then you get one point, called **the centroid** that decides what the new point becomes this makes further predictions much cheaper because you have to calculate much less distances to calculate.



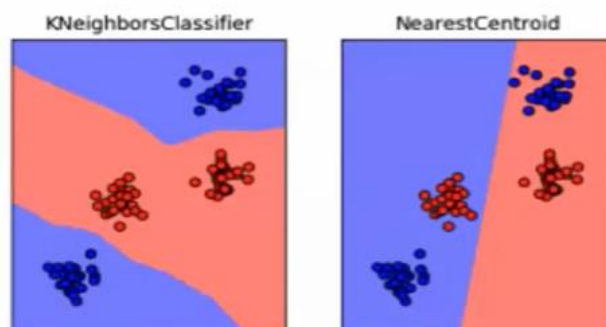
NEAREST SHRUNKEN CENTROID

This is the extension from the method above. Let's say you have **the middle at the line between the 2 centroids** this is the mean of the means $(c1+c2)/2$. Then you pick a **shrinking threshold** and that is how much you move the centroids of the clusters closer to the complete mean. This will then change the decision boundaries. This method is not widely used apparently but you still have to know how it works. Which I understand it doesn't sound sexy but on the other hand having those centroids on a draggable slider with the decision boundaries updating would look quite satisfying.



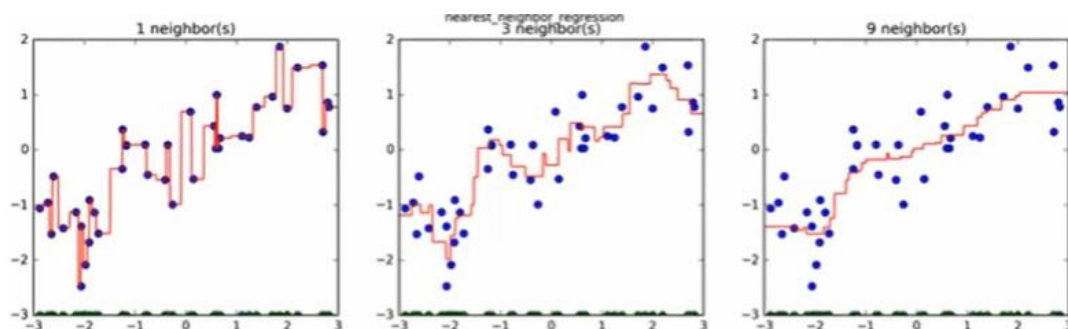
But this becomes **more useful if you have a lot of dimensions** because you could give every dimension its own shrinking threshold so you can move the centroids further or closer away from the complete mean like strings on a guitar. Then this becomes more interesting. You can also set negative shrunk centroid.

If you bring it too much to the centre, you will get this however. So be careful. Data should be linear.



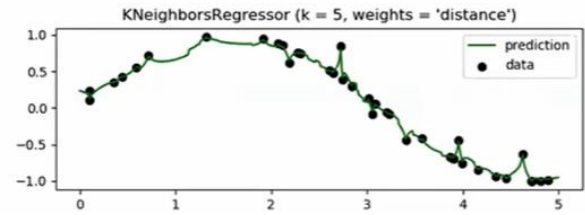
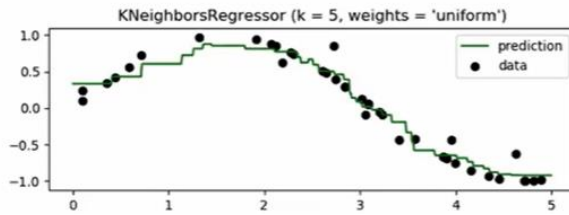
KNN REGRESSION

This technique will **find the best line between all the points/neighbours**.



The idea is to take the average of K nearest neighbours every time and that becomes a point on the new line. The higher K the smoother your line, again the negative relationship between K and complexity.

You can also do distance weighting which looks like a good idea.



k-NN – Advantages

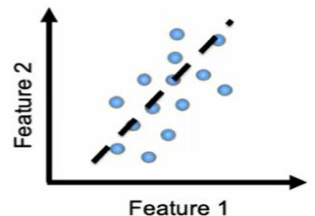
- The cost of the learning process is zero
- No assumptions about the characteristics of the concepts to learn have to be done
- Complex concepts can be learned by local approximation using simple procedures

k-NN – Disadvantages

- The model can not be interpreted (there is no description of the learned concepts)
- It is computationally expensive to find the k nearest neighbours when the dataset is very large
- Performance depends on the number of dimensions that we have (*curse of dimensionality*)

LINEAR REGRESSION

With linear regression you have a bunch of data points and you try and find the best linear line to describe the relation between these datapoints. If you do this well then you can use this line to predict new points. We have seen this in the statistic course. Because the line is linear, we get a model like $f(x) = y$ where f is $y = mx + b$.



To get a good model we need a lot of x 's for which y is known. We also need a **loss/cost function** which tells us how well our model approximates the known (training) examples. If we have this loss function, then we can **optimize/minimize** it to find the best m and b . So what cost function do we use? The least squared error. How does that look again?

$$\min \sum_i (y_i - mx_i)^2$$

Least squared error explanation

This means **take what your model predicted (mx)** and then **subtract that from the real value (y)**. That gives you the **difference between your prediction and what it should be**. Then **square this value**. Why do we square again? To make the value positive, ok but we could have used $\text{abs}()$ for this? Yes, that is also a valid technique, but we use square because this has some alleged nice math properties that we are of course not going to go into. Then you want to **do this for every instance in the dataset and add all the results together**. This gives you the **error of the model**. Nice. So now we find the parameters that give the lowest possible value here and that's our final model. So how do we find the lowest possible values?

You could try a lot of lines based on your human capacity for pattern recognition, but we can do better by taking the derivative of the $\sum (y - mx)^2$ and equalling it to 0. Doing that you end up with the magical equation:

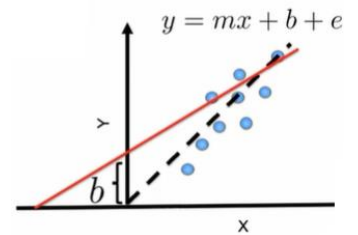
$$m = \frac{\sum(xy)}{\sum(x^2)}$$

FINDING B

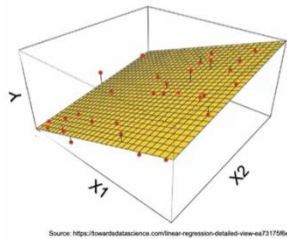
Now we have m you are done if your line goes through 0.

If it does not go through 0, we need to add the bias parameter (b). This you can find with $b = y_{\text{mean}} - mx_{\text{mean}}$.

Now on the right we also see an e parameter this is the error you want to minimize. This is always shown (however not in the book) but if you actually try to calculate a line you don't need it, I think it is more for the idea.



SCALING UP WITH MORE FEATURES



If we could only use this for comparing 2 features, then this would not be that useful as with machine learning there are usually **a lot of features** so how do we go beyond 2 features? Just add more mx pairs.

So you would get $y = m_1x_1 + m_2x_2 + m_3x_3 \dots + b$

Two input variables make your line into a plane. If we do this with more variables it makes sense to show the model in **matrix notation**. See formula below:

$$y = m^T \mathbf{x} + b = \sum_{i=1}^k m_i x_i + b$$

So here instead of saying you have a couple features x_i that each have their own m_i you have **a bunch of features in matrix X** and these are your input features. Then you **multiply this matrix with a vector of corresponding coefficients**. This way you can give another matrix that has values for all the features and you can just **multiply it with the same coefficient vector**.

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix} * \begin{bmatrix} 0 & 6 & 12 & 18 & 24 & 30 \\ 1 & 7 & 13 & 19 & 25 & 31 \\ 2 & 8 & 14 & 20 & 26 & 32 \\ 3 & 9 & 15 & 21 & 27 & 33 \\ 4 & 10 & 16 & 22 & 28 & 34 \\ 5 & 11 & 17 & 23 & 29 & 35 \end{bmatrix}$$

You transpose the vector to make it from - (row) to | (column) because so you multiply each row. This is nice because you only need the vector and you can then give it any matrix.

This matrix way is just a nicer way to work.

MEASURING FIT

This whole process is called **fitting a line**. We can measure how well we fit a line (how well does your model predict) by calculating our old friend R^2 ; which is the % of variance explained by the model.

If you can't remember: $R^2 = 1 - \frac{SS_{\text{residual}}}{SS_{\text{total}}}$

where $SS_{\text{total}} = \sum (y_i - y_{\text{mean}})^2$ and $SS_{\text{residual}} = \sum (y_i - f_i)^2$ where f_i is the predicted value for y_i .

This only works if the input variables are on the same scale. You could do centring (from statistics) or regularization (described below) if they are not on the same scale yet.

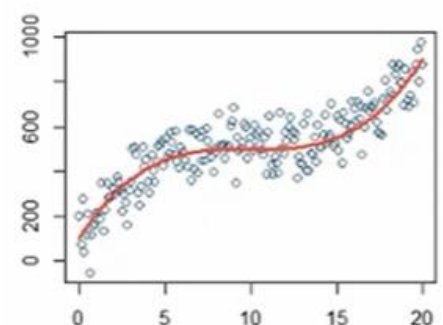
HOW TO DO THIS IN PYTHON?

Numpy actually has a function to do this called polyfit.

```
a, b = np.polyfit(x, y, 1)
print(a, b)
plot_line_and_ss(x, y, a, b)
```

NON-LINEAR REGRESSION

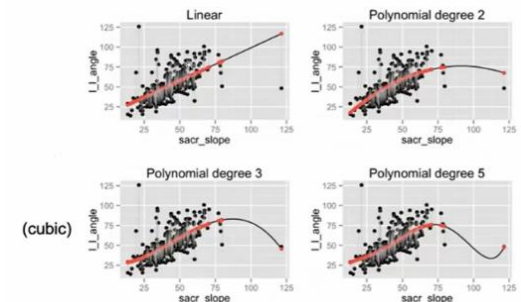
So what if your data can not be well described with a linear line? Then we just make polynomial functions like $y = mx + mx^2 + b$ or $y = mx + mx^2 + mx^3 + b$ you can just keep going with this. You can use plots to find if you're fitting well but this gets harder if you add more input variables.



HOW TO COMBAT OVERFITTING IN REGRESSION

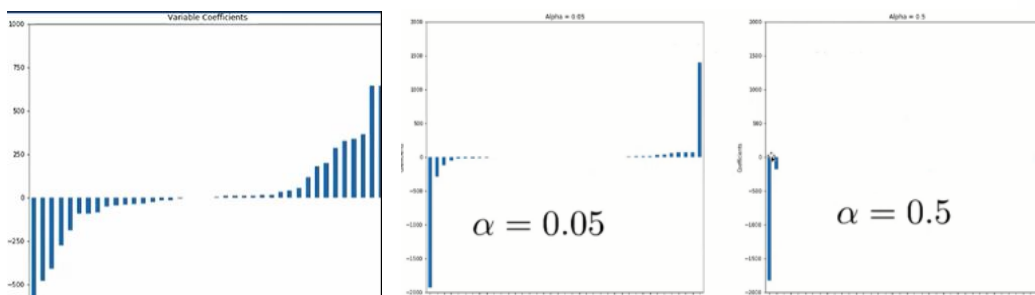
Using non-linear regression, you could make a model that describes all the points perfectly by just adding more and more powers however this model won't be any good as it won't generalize well at all with new data. There are 2 ways to do this. The first possibility is just to **reduce the model complexity** the other way is **regularization**.

Linear vs polynomial fit?



REGULARIZATION

The idea of regularization is to reduce the **magnitude** (effect on output) of the **coefficients** (by what you multiply the input variables). There are 3 regularization techniques.



But first why would you want to do this? Because **we want to prevent that only a couple coefficients determine the whole output** while a lot of other inputs only have a minor impact. These smaller variables will be overshadowed in a sense by the larger ones. If you keep adding more

dimensions the chance of this happening gets larger. This is called **the curse of dimensionality**. We can do this by, in addition to minimizing the squared error, also taking additional steps.

The three discussed techniques to do this are:

RIDGE REGRESSION

Here you take the original formula and turn it into $\min \sum_i (y_i - m^T x_i)^2 + \alpha \|m\|^2$. The first part is the same but then you add a penalty term **alpha times the coefficients vector m squared and then normed**.

$\|m\|$ is $\|u\| = \sqrt{\sum (u^2)}$. Norm of the vector is like the absolute value; we saw this in linear algebra.

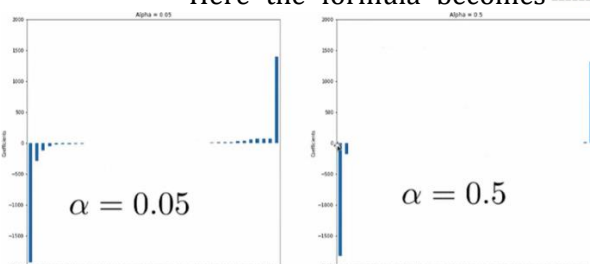
The point of squaring is to have a bigger effect on large coefficients. $2^2 = 4$, 100% increase but $3^2 = 9$ 150% increase. And **we remove this from the original sum part we have**. So ridge regression allows you to **dampen the impact of large coefficients** based on the penalty term **alpha**. If you have an alpha between 0 and 1 you also actually increase the effect of the large coefficients.

Doing ridge regression gives a better model because your coefficients become more uniform.

This type of regression is **called L2 regression**.

LASSO REGRESSION

Here the formula becomes $\min \sum_i (y_i - m^T x_i)^2 + \alpha \|m\|$ the **difference with ridge regression** being that you **don't square**. This has the result that you can actually get that $\alpha \|m\| = 0$ the first part of the formula and brings a coefficient to 0. This causes that **coefficient to be eliminated from the model**. That can be great if you want to **reduce complexity**. So this is more like playing into that some coefficients have much more effect than others and embracing it. This type of regression is **called L1 regression**.



ELASTIC NET REGRESSION

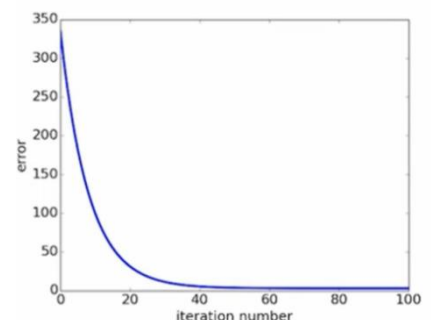
Here the formula becomes $\min \sum_i (y_i - m^T x_i)^2 + \alpha_1 \|m\| + \alpha_2 \|m\|^2$ so just both ridge and lasso.

Summary of linear regression

- Estimates best fitting linear model using Least Squares Estimation as a cost function (= loss function = objective function).
- Parameters for coefficient and intercept can be mathematically derived.
- Model fit is expressed by R^2
- Can be generalized to multiple inputs (estimates hyperplanes instead of lines)
- Polynomial functions allow more complex non-linear fits
- Ridge, Lasso, and Elastic net regression can be used to reduce the magnitude of large coefficients.

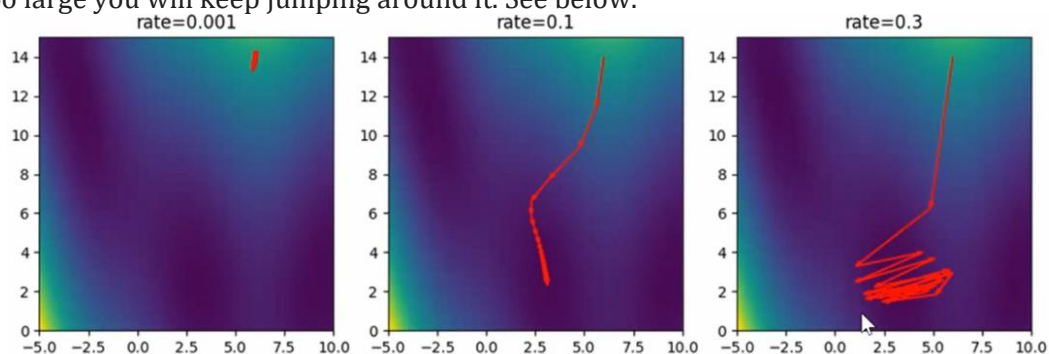
GRADIENT DESCENT

For the following ML techniques, you can't use a direct formula to get the hyperparameters and the weights. This is where **gradient descent** comes in. Gradient descent means **taking a derivative** of the **cost function** (whatever it may be) and **then adjusting parameters to get closer to a minimum cost**. How much you adjust the parameters each time is based on a **learning rate** (λ , lambda) you choose beforehand. After the adjustment in the direction, you got from the derivative, you repeat until you get to a local minimum.

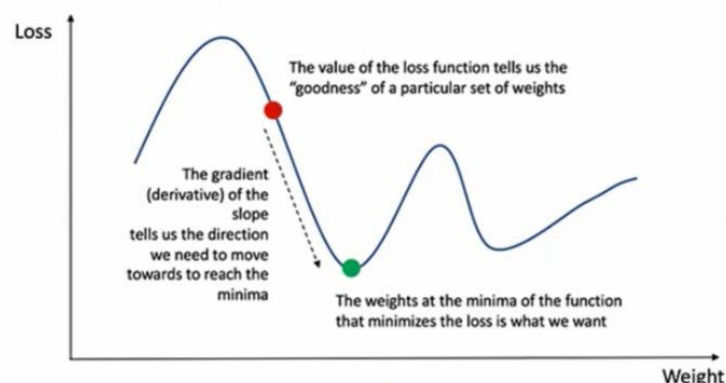


This works well but of course you do have the **local minima problem** if you are working with **non-linear data** so when you are, try this technique with multiple **initial values**.

You also have to set a good learning rate because if it's too small it takes too long to get to the minimum and if it is too large you will keep jumping around it. See below.



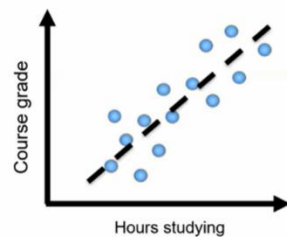
$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \lambda \frac{\partial l(w)}{\partial w_j^{(t)}}$$



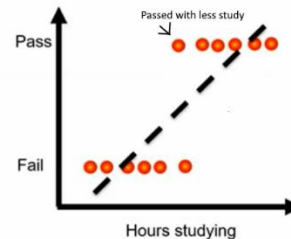
LOGISTIC REGRESSION

This is regression but for **classification**. The output is a class and not a function to transform.

Given the number of hours a student spent learning, what grade will the student achieve in the exam?



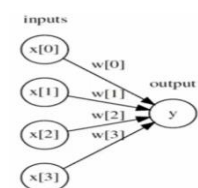
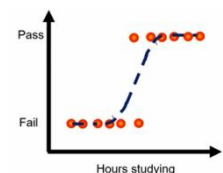
Given the number of hours a student spent learning, what grade will the student achieve in the exam?



We get from a continuous output to a class by adding **thresholds** so if we have a function like for instance $\hat{y} = m_1x_1 + w_2m_2 + \dots$ (Remember, we can also write this as $\hat{y} = \mathbf{w}^T\mathbf{X} + b$). Then you convert it to $\hat{y} = m_1x_1 + w_2m_2 + \dots > 5.5$ or $\hat{y} = \mathbf{w}^T\mathbf{X} + b > 5.5$. So if $\hat{y} > 5.5$ then the student passed.

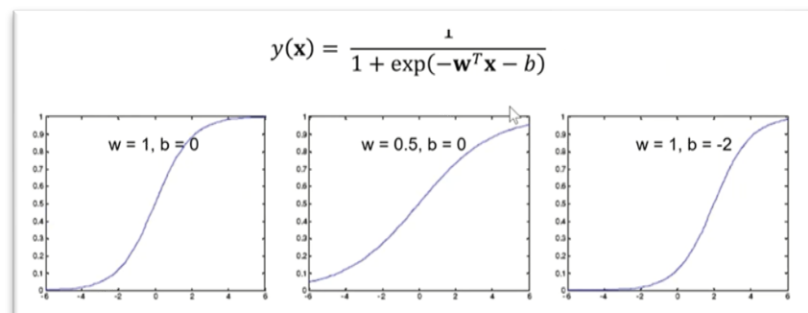
THE SIGMOID FUNCTION

We can also do this automatically by putting our linear function in a **sigmoid** or **logistic** function. These functions take functions to transform them woah. The sigmoid function looks like this: $\sigma(z) = \frac{1}{1 + \exp(-z)}$ where z is the original linear function. The weird σ is called sigma. So that gives you $y(x) = \sigma(\mathbf{w}^T\mathbf{X} + b)$ and that gives you $y(x) = \frac{1}{1 + \exp(-\mathbf{w}^T\mathbf{X} - b)}$

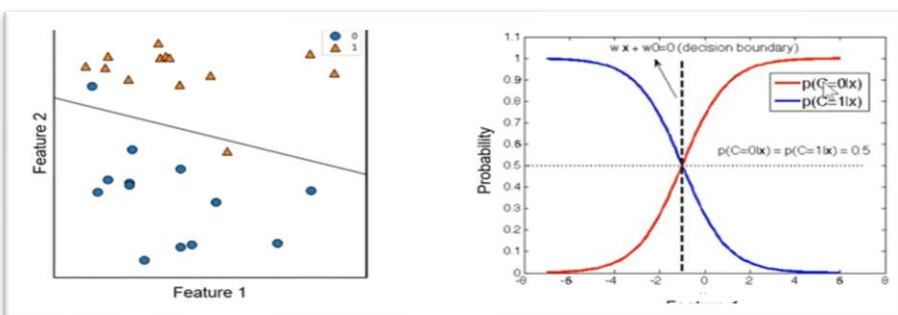


The output is a smooth function of the inputs and the coefficients/weights. It can be seen as a smoothed and differentiable alternative to `sign()`. This is a function with a steep increase at some point you can see this as a sort of continuous threshold. These usually fit better with classification. It also always outputs a value between 0 and 1 which means you can interpret the output as a percentage.

Below are some pictures to see how the sigmoid function reacts to changes to its parameters.



So w controls the curve and b shifts the so called s-curve.



Using this technique we get a decision boundary again for the classification like seen in the left plot. The idea of this second plot is that we can plot the probability for both classes because the total probability is always 1. So if you have probability for the red line the blue line would be $1 - \text{the red \%}$. This does become more complicated with more features than you have to calculate it individually. But we can actually interpret

the output of these models as a % of confidence so that is nice.

CROSS ENTROPY

For regression the cost function is what we have seen, least errors $\sum(\hat{y}_i - y_i)^2$. For classification we use something called the **cross-entropy loss function**. Entropy? **Entropy is a level of uncertainty**. So let's say we have a function $y(x)$ then you can calculate the **entropy** $H(x)$ with $-\sum(y(x) \log y(x))$. The - is there to make sure this function always outputs a positive number in the range of 0-1. Notice that this is another function that takes another function as input so now we have $H(\sigma(y(x)))$. So the full function would look like this:

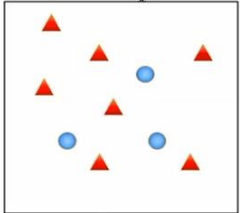
$$-\sum_{k=1}^K y_k \log a_k \quad a_k = \frac{\exp(\hat{y}_k)}{\sum_{j=1}^K \exp(\hat{y}_j)}$$

And then with all the layers

$$l(\mathbf{w}) = -\mathbf{y}^T \log(\sigma(\mathbf{z})) - (1 - \mathbf{y})^T \log(1 - \sigma(\mathbf{z}))$$

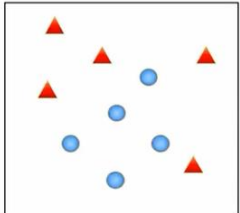
Let's compute the entropy for the following cases

$$H(x) = -\sum_x p(x) \log p(x)$$



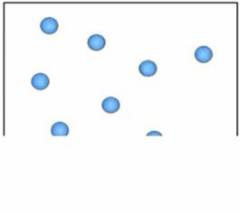
#▲ = 7, #● = 3

$$H(x) = -\left[\frac{7}{10} \log_e \left(\frac{7}{10}\right) + \frac{3}{10} \log_e \left(\frac{3}{10}\right)\right] = 0.61$$



#▲ = 5, #● = 5

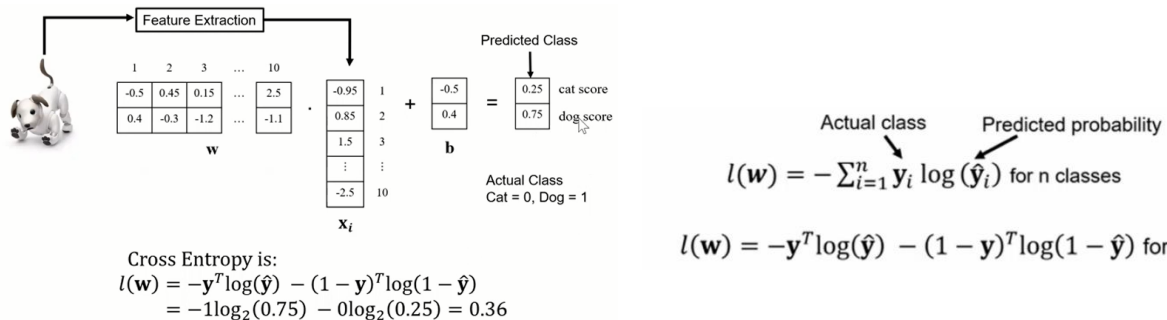
$$H(x) = -\left[\frac{5}{10} \log_e \left(\frac{5}{10}\right) + \frac{5}{10} \log_e \left(\frac{5}{10}\right)\right] = 0.69$$



#▲ = 0, #● = 10

$$H(x) = -\left[\frac{10}{10} \log_e \left(\frac{10}{10}\right) + \frac{0}{10} \log_e \left(\frac{0}{10}\right)\right] = 0$$

But we are actually after **cross entropy** this just means that you are comparing 1 class to the other classes. We want to calculate how right our model is based on **how certain it is that your input is one class**. This makes more sense if you remember that the classification model gives percentages for each class. What we want our model to say is "ok I think there is 99% chance this is a dog" instead of "hmm I think there is 70% chance that this is a dog and 30% that this is a rattle's sake". To get this we **punish predictions where multiple classes have a high output**. That looks like this:



To get the best coefficients, use gradient decent. You have to use the chain derivative rule. I don't think you actually need to know how to use it.

REGULARIZATION FOR Logistic Regression

Like for linear regression you can also have overfitting or underfitting with logistic regression. You can use the same regression ideas as mentioned for linear regression. This is how it looks for L2 (ridge) for example:

$$l(\mathbf{w}) = -\sum_{i=1}^N (y^{(i)} \log(\hat{y}^{(i)}) - (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})) - \sum_{i=1}^N \frac{(w_i)^2}{2\sigma}$$

In the lecture they did not really go in depth about how this would look for L1. It seems like the ideas are the same. But with instead of alpha we have C.

MULTICLASS CLASSIFICATION

This is when you are trying to predict more than 2 different classes. There are 2 ways to do this. **1 vs rest** and **1 vs 1**.

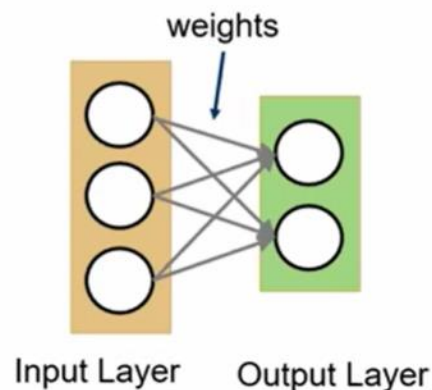
Let's say you have 4 different classes.

The idea with 1 vs rest is that you build 4 binary classifiers one for each group compared with the other group so that would look like 1 vs {2,3,4}, 2 vs {1,3,4}, 3 vs {1,2,4} and 4 vs {1,2,3}. So 4 classifiers on all the data.

With 1 vs 1 you compare each class to each other class so you would have 1v1 1v2 1v3 1v4 2v3 2v4, 3v4. So you would always have $n * (n-1)/2$ binary classifiers on a fraction of the data.

Logistic Regression Classifier

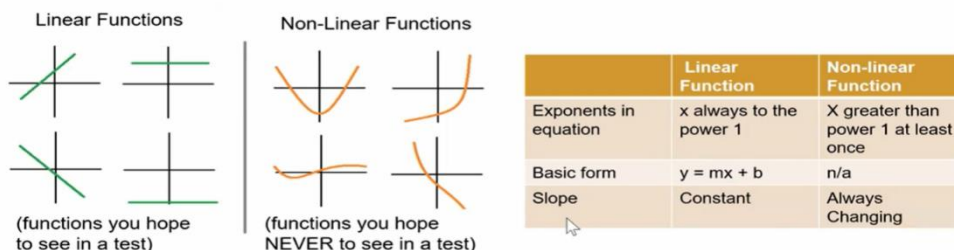
- Advantages:
 - Easily extended to multiple classes
 - Probability distributions available
 - Quick to train
 - Good accuracy for many simple datasets
 - Resistant to overfitting
 - Can interpret model coefficients
- Disadvantages
 - Linear decision boundary



NEURAL NETWORKS

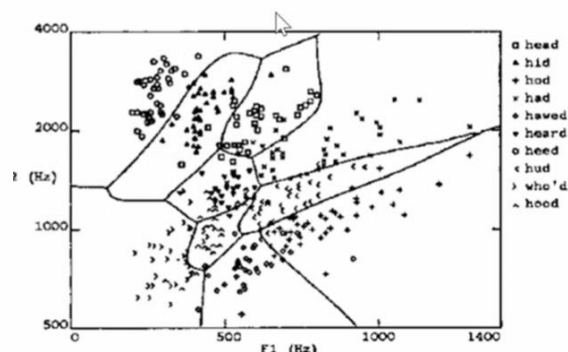
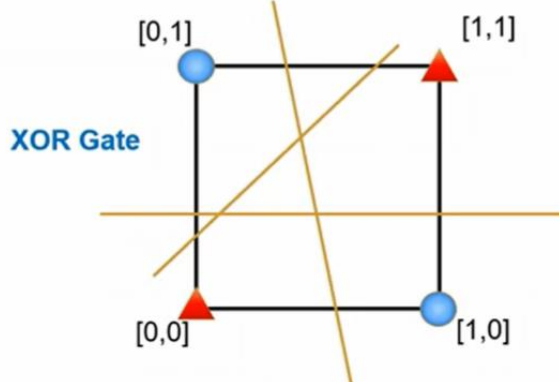
This technique is used for **highly non-linear functions**. So what makes functions non-linear again?

What Makes a Function Linear?



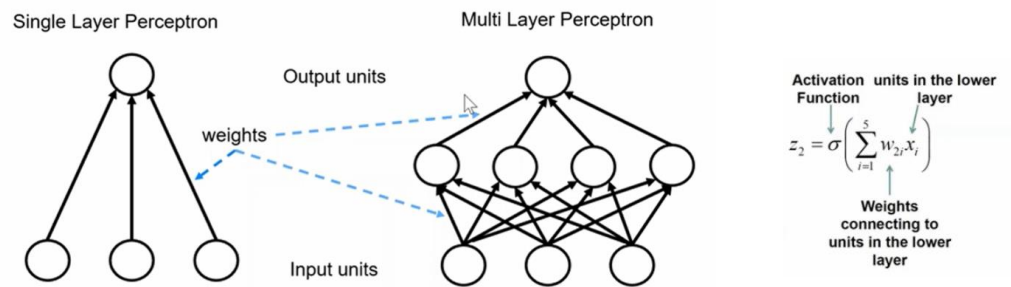
Real life has a lot of non-linear relationships like speech or vision. These are interesting ML problems.

Sometimes you **can't split a decision boundary with just one line** for instance with these:



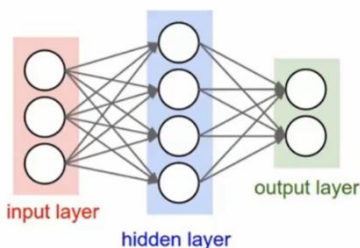
So it is useful to make nonlinear classifiers. How do we do this? One way is to use a neural network. A neural network is made out of a **large number of simple linear functions** called **activation functions** of which the format never changes (gaussian, sigmoid but also maybe something called Relu or SoftMax). Then you want to **optimize the neural network by combining the activation functions in linear ways**.

The inspiration of neural networks is the human brain with its neurons what a shocker. **Another name** for neural networks is **multi layer perceptron**. Logistic regression is actually a single layer perceptron. So the left is **logistic regression**, and the right is a **neural network**.



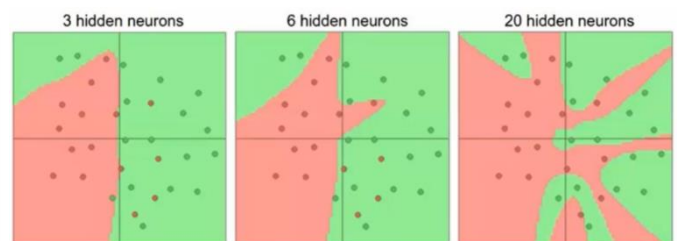
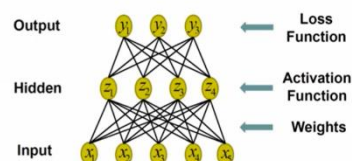
Again we optimize by minimizing error. Real value – what you got.

A neural network has **layers** and **units**. The columns with circles are called a layer and every circle is a unit in a layer. The network on the left has 2 layers **as the lines between the columns actually count the number of layers**. **The more layers you add the more non-linear things you can predict** as you can see in the picture below. Adding more layers is better than adding more units. Which activation function do you choose? It depends on the size of your network. If it is big/deeper, then Relu apparently because its easier to compute, otherwise SoftMax because it is smoother.



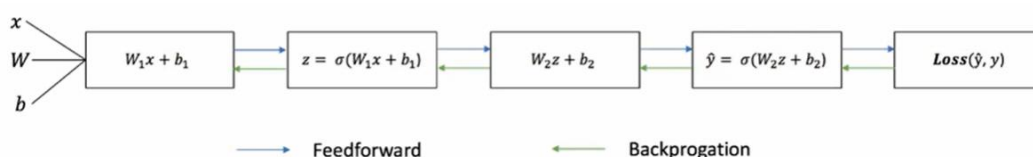
If this is still a bit vague then look at the structure of a neural network below:

- Input layer: \mathbf{x} , Independent variable
- An arbitrary amount of hidden layers
- Output layer: \mathbf{y} , dependent variable
- A set of weights (coefficients) and biases **at each layer** (\mathbf{W} and \mathbf{b})
- A choice of **activation functions** for each layer σ .

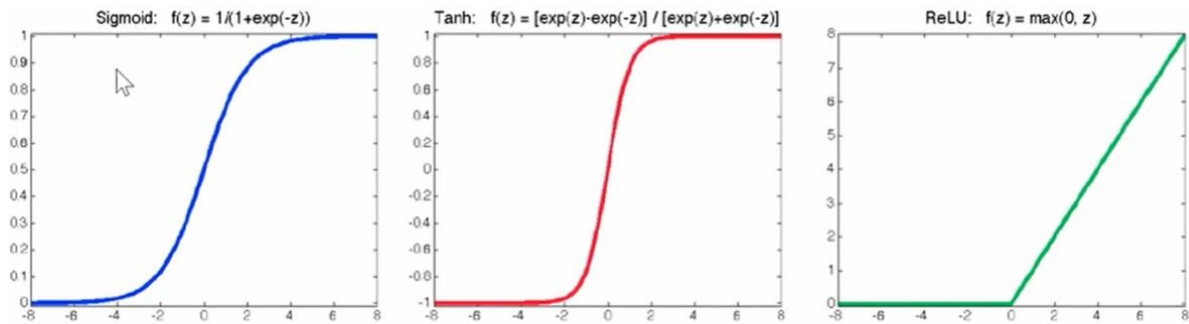


So I think the idea is that you start with a vector \mathbf{x} which has the inputs and then you **multiply each of the inputs by different weights to get the value for a unit in a hidden layer**. So in practice you would multiply by a vector of weights. Each line represents a weight in this vector. So for instance z_2 would be calculated by the result of all the different x multiplied by different weights (all lines connected to z_2 from below). Then if you calculated all the units in a hidden layer proceed to do this again for the next layer. If you just take the math formulas and forget about the graph for a second, a formula for a single output of neural network with 2 layers would like so $\hat{y} = \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2))$. Start in the middle. **The outputs of the linear functions are given as the x /inputs to the next layer**. This layer then puts new weights on these inputs and then gives the result to the next layer/function. Keep going till you reach y . This makes it that your output is only affected by the bias and the weights at each layer. I feel after seeing that formula that layer has a new meaning: each layer of the functions.

So how do you find the best weights for each layer? First **feedforward**. That means calculate the predicted output (y) and then do **feedback** more often called **backpropagation**. **Update the weights and biases based on the output you got**. We are not going more into backpropagation until the deep learning course.



Here are some of the activation functions I mentioned plotted:



name	function	derivative
Sigmoid	$\sigma(z) = \frac{1}{1+\exp(-z)}$	$\sigma(z) \cdot (1 - \sigma(z))$
Tanh	$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$	$1 / \cosh^2(z)$
ReLU	$\text{ReLU}(z) = \max(0, z)$	$\begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases}$

You can use neural networks for both classification and regression by using different kind of loss functions, remember: sum of squared errors for regression and cross entropy for classification. We change the weights and biases to reduce error. We use the chain rule to get the derivative, makes sense. With the derivative you can do gradient descent and with that you can update your weights.

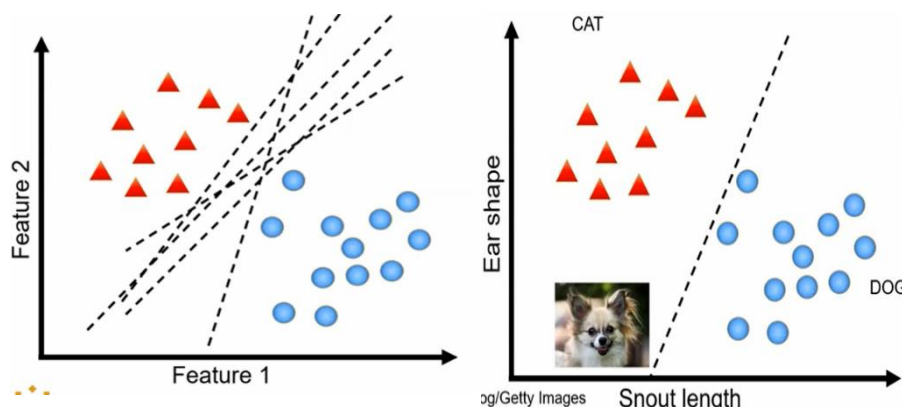
Artificial Neural Networks

- Highly expressive non-linear functions
- Highly parallel network of logistic function units
- Goal : $\sum_{i=1}^n (\hat{y} - y)^2$
 - minimize sum of square training errors
 - minimize sum of square training errors plus weight squared (regularization)
- Gradient descent as training procedure
- Problems
 - Local minima
 - Overfitting
 - Early stopping

If you still feel unsure about neural networks then watch [this great video about them](#).

SUPPORT VECTOR MACHINES

This is a **classification technique**. The idea of this technique is to find the best decision boundary out of many decision boundaries that are possible. **We do this by looking at the most difficult to predict points.** As you can see on the left there are **many decision boundaries possible that separate the dataset, but which is the best one?** Which generalizes the best? Because as you can see in the second picture that poor dog was classified as a cat even though he was closer to the other blue points. This happened because of the decision boundary we choose.



So how do we know what the best decision boundary is? We take a journey to get there. The idea is to **take the decision boundary with the largest margin**. To get the margin for each possible decision boundary we draw 2 **support vectors** on each side of the decision boundary with the same slope as the decision boundary. To find the slope you just take the norm of the weights of that decision boundary. But what intercept do we use? **You get the intercept by drawing your support vectors so the points that are closest to the decision boundary are on it**. Then **the margin is the distance between the support vectors and the decision boundary** (the black arrows). A way to formalize this is to say we want to draw the biggest margin that makes sure all the points are classified. This is hard margin SVM. **With SVM we say that the best decision boundary is the one with the highest margin! Why?**

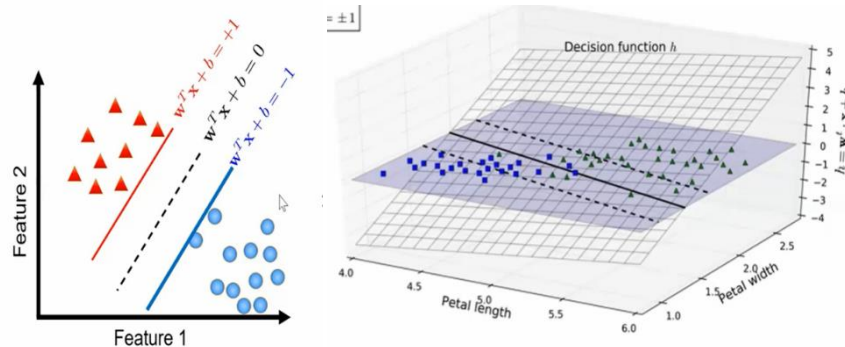
Because now points that fall between the support vectors and the decision boundaries are classed as **undefined**. With the biggest margin we get the biggest undefined area, so we are the most certain that we don't misclassify. So we predict the class of a new instance by computing the decision function:

$$\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + \dots + w_k x_k + b$$

If the result is positive, then the predicted class is 1 (positive class) if it is negative then its negative. Basically like this:

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} + b \geq 1 \\ -1 & \text{if } \mathbf{w}^T \mathbf{x} + b \leq -1 \\ \text{Undefined} & \text{if } -1 \leq \mathbf{w}^T \mathbf{x} \leq 1 \end{cases}$$

If the result is between 1 and -1 (so between the support vectors) then the result is undefined.

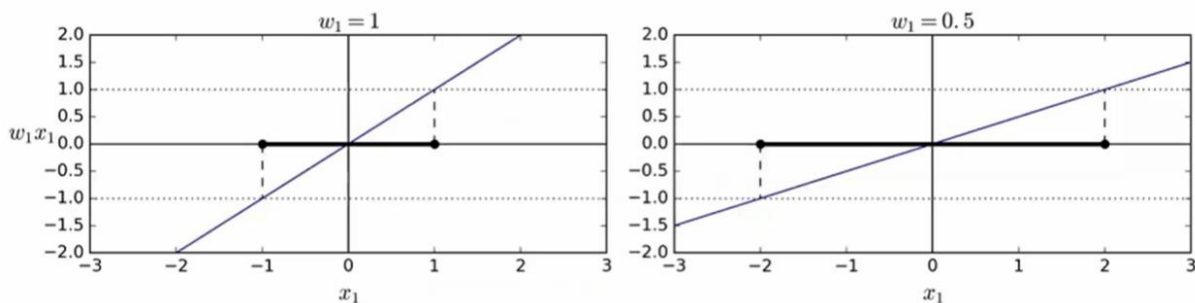


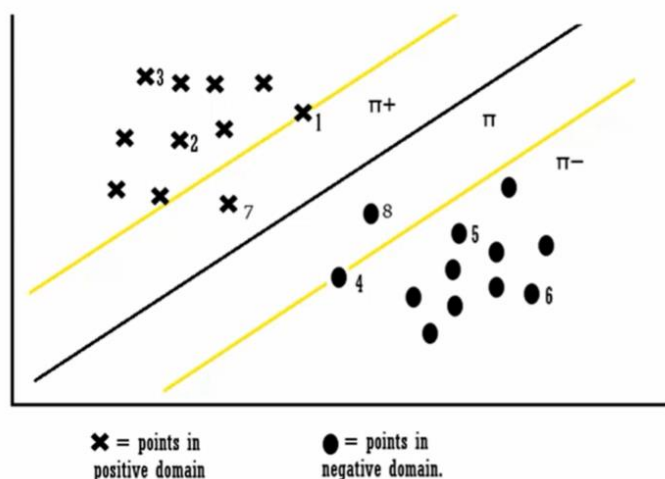
How do we find the best support vectors quick? With a **loss function** and computing a **gradient descent with the change in margin**. We want to maximize the margin between the data points and the decision boundary.

Ok so here are some examples:

First see that here the slope calculated by the weights:

Here is an example of drawing the SVM.





Correctly Classified

Point 3:
 $y_i = 1$
 $y_i(w^T x_i + b) > 1$

Point 4:
 $y_i = -1$
 $y_i(w^T x_i + b) = 1$

Point 5 and 6
 $y_i = -1$
 $y_i(w^T x_i + b) > 1$

Incorrectly Classified

Point 7:
 $y_i = 1$
 $y_i(w^T x_i + b) < 1$

Point 8:
 $y_i = -1$
 $y_i(w^T x_i + b) < 1$

SUMMARY SO FAR OF SVM

So again we want to use the **decision boundary with the largest margin**. The margin is based on the distance between the support vectors. We draw the support vectors, so we get the largest margin but also based on the constraint of that we make sure each point in the training set is classified. This is called hard margin SVM. If you don't do this constrained, then the largest margin is infinity. To find the best support vectors we use gradient descent to maximize the margin (while keeping to the constraint).

To maximize your margin you want to have smaller weights.

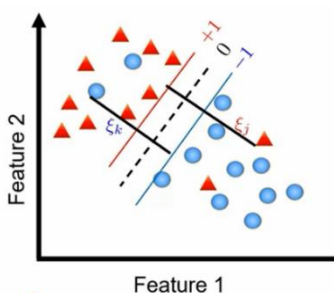
$$\text{minimize } \frac{1}{2} \|w\|^2 \text{ instead of } \|w\|$$

$\|w\|$ if not differentiable at $w = 0$

MISCLASSIFY OR UNDEFINED?

Is it better to say undefined instead of making a wrong prediction? I would say yes, it is better to admit that you don't know the answer or are unsure then to guess. Also because you might have things that are in a grey area and this is literally adding a grey area to your model. But if this is good use your judgement and you can also use other classifiers.

WHAT IF YOUR DATASET HAS OVERLAP?



So what do we do if you can't linearly separate your dataset like below? Then we use soft margin SVM. We just say: "ok instead of saying we have to classify each label we allow for x number of wrong classifications". The number of wrong classifications allowed are called **slack variables**. We express them with ξ (xi). So we allow a certain amount of ξ in the classification when finding the biggest margin. We have to be ok with some errors because if we would use hard margin SVM (so with $\xi = 0$) then we couldn't make any support vectors for this dataset.

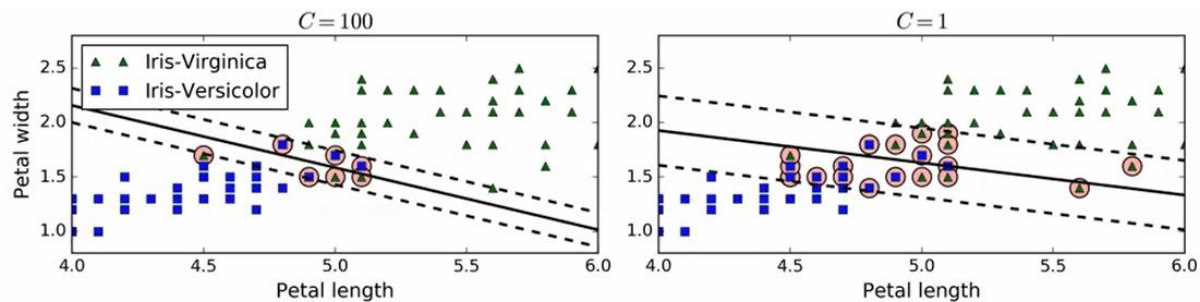
So we get 2 objectives:

- Make the number of slack variables as small as possible to reduce the margin violations.
- Make the weights as small as possible to increase the margin.

$$\text{minimize } \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \xi^{(i)}$$

$$\text{subject to } t^{(i)}(\mathbf{w}^T x^{(i)} + b) \geq 1 - \xi^{(i)} \text{ and } \xi^{(i)} \geq 0 \quad i = 1, 2, \dots, n$$

The trade-off between these 2 objectives can be set with **the C hyperparameter**.



So with all this the loss function looks like this:

$\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^i + b))$ This is the loss function for multiclass SVM function:

$$SVM Loss = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

DIFFERENCE BETWEEN HARD AND SOFT SVM

Hard margin SVM minimize $\frac{1}{2} \mathbf{w}^T \mathbf{w}$
subject to $t^{(i)}(\mathbf{w}^T x^{(i)} + b) \geq 1, \quad i = 1, 2, \dots, n$

Soft margin SVM minimize $\frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \xi^{(i)}$
subject to $t^{(i)}(\mathbf{w}^T x^{(i)} + b) \geq 1 - \xi^{(i)}$ and $\xi^{(i)} \geq 0 \quad i = 1, 2, \dots, n$

- Are both convex quadratic optimization problems with linear constraints.
- Such problems are Quadratic Programming (QP) problems
- Can be solved with QP solvers

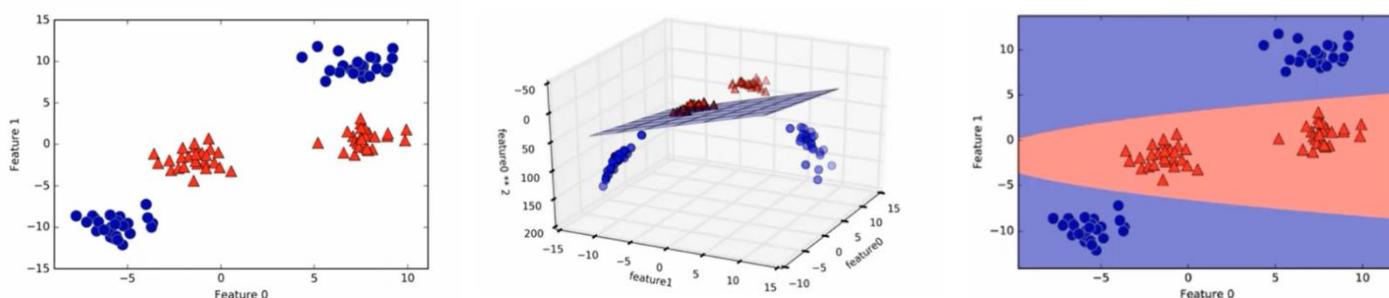
WHAT IF YOU CAN'T SEPARATE THE DATASET

If you have something more complicated like the first graph below (on the left) then you can't use linear SVM because you can't have 1 line that separates the dataset without having a kind of unacceptable ξ so what do we do? We go to kernel SVM.

KERNEL SVM

The idea here is to **add a dimension** to your dataset. You do this by adding something **based on the original features**. For instance each feature to the power of 2. Then you **separate it linearly in the higher dimension** with a higher dimensional object like a plane and then bring it all back down to first dimension. For 2D this leaves you with an ellipse-like decision boundary. The functions used to apply this transformations are called **kernels**. Applying a kernel makes the model non-linear.

This works better for smaller datasets because you do calculation for every instance in the dataset. This makes Kernel SVM slower than linear svm. See an example below:



DIFFERENT KERNELS

See below for examples of different kernels available. Again the idea is to add another dimension to your dataset and these kernels will give you that. RBF stands for radio basis function.

$$\text{Linear : } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polynomial : } K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$$

$$\text{Gaussian RBF : } K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$$

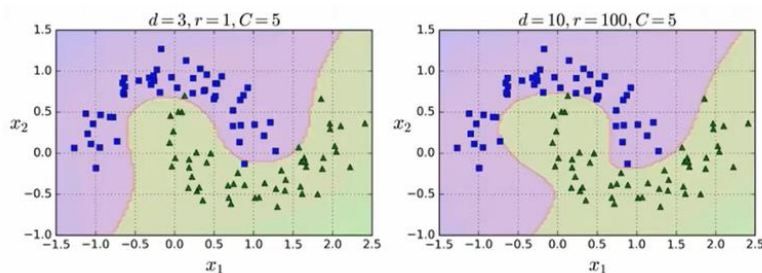
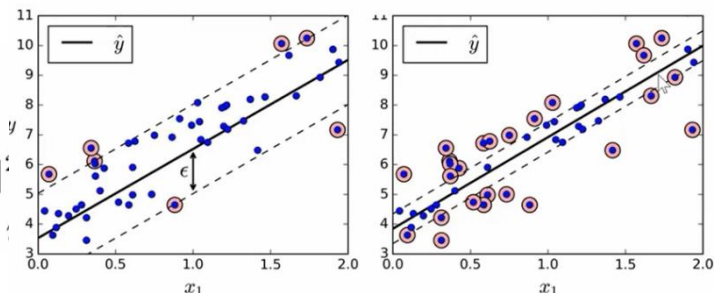
$$\text{Sigmoid : } K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \cdot \mathbf{b} + r)$$

We also have 2 hyperparameters (the r is just for the kernel formula). C is for regularization like you read before it is also how many slack variables you may have. D is what you multiply your polynomial with. The larger the d the more you are prone to overfitting lol.

```
from sklearn.svm import LinearSVR
```

```
svm_reg = LinearSVR(epsilon=1.5)
```

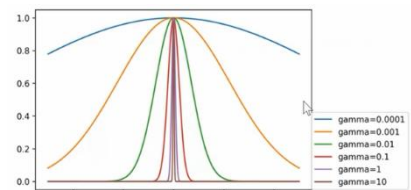
```
svm_reg.fit(X, y)
```



SIMILARITY FUNCTION

Another technique to tackle non-linear problems is to **add features computed using a similarity function** that measures how much each instance resembles a particular landmark.

The landmarks here are the two functions that give those hills x_2 and x_3 . The technique looks **how similar each feature is to the other features of an instance and then plots the similarity**. So if something is high in x_2 and x_3 then it is more in the middle of the similarity plot (I think, I found this lecture hard to follow). **If 2 points lie on the same gaussian function they are more similar**. The idea is to see how similar they are. This is an example of RBF. The hyperparameter Gamma represents the bandwidth which represents how wide the hill is.



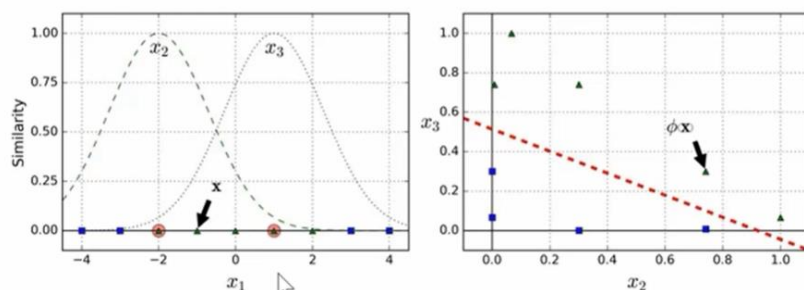
REGRESSION WITH SVM LOOKS LIKE

This technique is more for classification but you can also do regression.

$$\begin{aligned} &\text{minimize } \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{c}{2} \sum_{i=1}^n \epsilon_i^2 \\ &\text{subject to } t_i(\mathbf{w}^T \mathbf{x}_i + b) \geq 1 - \epsilon_i \quad i = 1, 2, \dots, n \end{aligned}$$

number of errors (points in the margin). You also try to go for functions as flat as possible. We don't care about points that are in the margin. So the higher ϵ the bigger the margin. The slope of the line is calculated by the points that do not fall in the margin. Again the margin is the space between the support vectors.

With regression with SVM you set an ϵ (epsilon). **This is the distance between the line and the decision boundary. So it is the margin**. Then you use gradient decent to see which line fits best with that margin with the lowest



You can also have slack variables to allow some error.

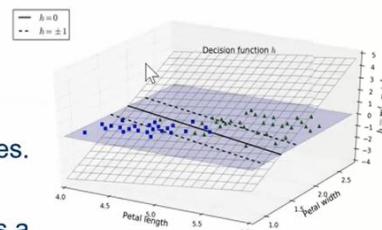
$$\text{minimize} \quad \frac{1}{2} \|w\|^2 + C \sum_i^l (\xi_i + \xi_i^*)$$

$$\text{subject to} \quad \begin{cases} y_i - \langle w, x_i \rangle - b \leq \epsilon + \xi_i \\ \langle w, x_i \rangle + b - y_i \leq \epsilon + \xi_i^* \\ \xi_i, \xi_i^* \geq 0 \end{cases}$$

Again you can also use a kernel with this. Higher c allows for more error.

Support Vector Machines

- A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane.
- Given labelled training data (supervised learning), the algorithm outputs an optimal hyperplane which categorizes new examples.
- In two dimensional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side.



Kernel SVMs

- Advantages
 - Allow for complex decision boundaries, even if the data has only a few features.
 - Work well on low-dimensional and high-dimensional data (i.e., few and many features)
- Disadvantages
 - Do not scale very well with the number of samples. Running an SVM on data with up to 10,000 samples might work well, but working with datasets of size 100,000 or more can become challenging in terms of runtime and memory usage.
 - Require careful preprocessing of the data and tuning of the parameters..
 - SVM models are hard to inspect; it can be difficult to understand why a particular prediction was made, and it might be tricky to explain the model to a nonexpert.
- Still, it might be worth trying SVMs, particularly if all of your features represent measurements in similar units (e.g., all are pixel intensities) and they are on similar scales.

~~NATIVE BAYES~~

~~DECISION TREES~~

~~ENSEMBLE LEARNING~~

USING THESE MODELS WITH PYTHON

1. Import the model you want to use from sklearn.
2. Turn your data in a numpy array of numpy arrays
3. Split your data with the `train_test_split` function from sklearn
4. Make an instance of the model with the hyperparameters for that model
5. Call the `.fit` method with your train data to fit the model
6. Call the `.score` method with your test data see how good your model fit is
7. If it is not good enough keep changing hyper parameters or switch technique
8. Call the `.predict` method to use the model to predict data.

```
from sklearn import MyAwesomeLearningModelElonPlsInvest as MALMEPI      # 1
from sklearn.model_selection import train_test_split
from numpy import genfromtxt

my_data = genfromtxt('my_data.csv', delimiter = ',')                      # 2
my_labels = genfromtxt('my_labels.csv', delimiter = ',')                  # 2
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1) # 3

clf = MALMEPI(hyper_paramater1 = 42, cool_greek_letter = 'ΦηQ')          # 4
clf.fit(X_train, y_train)                                                  # 5
print(f"Test set accuracy: {clf.score(X_test, y_test)}")                  # 6
print(f"Test set predictions: {clf.score(X_test, y_test)}")              # 8
# Insert code to make a nice plot here
```

But you should still probably look at the notebooks a before taking the exam.

Now if you found this summary to wordy then I recommend looking at the [cheat sheet](#) found by Giedrius (Geed-ri-uhs) on LinkedIn

Thanks for using my summary and thanks to Marieke and Esteban for going through it first.
I hope it was useful to you!

This summary was lovingly made by Quinten Cabo :)

Good luck!