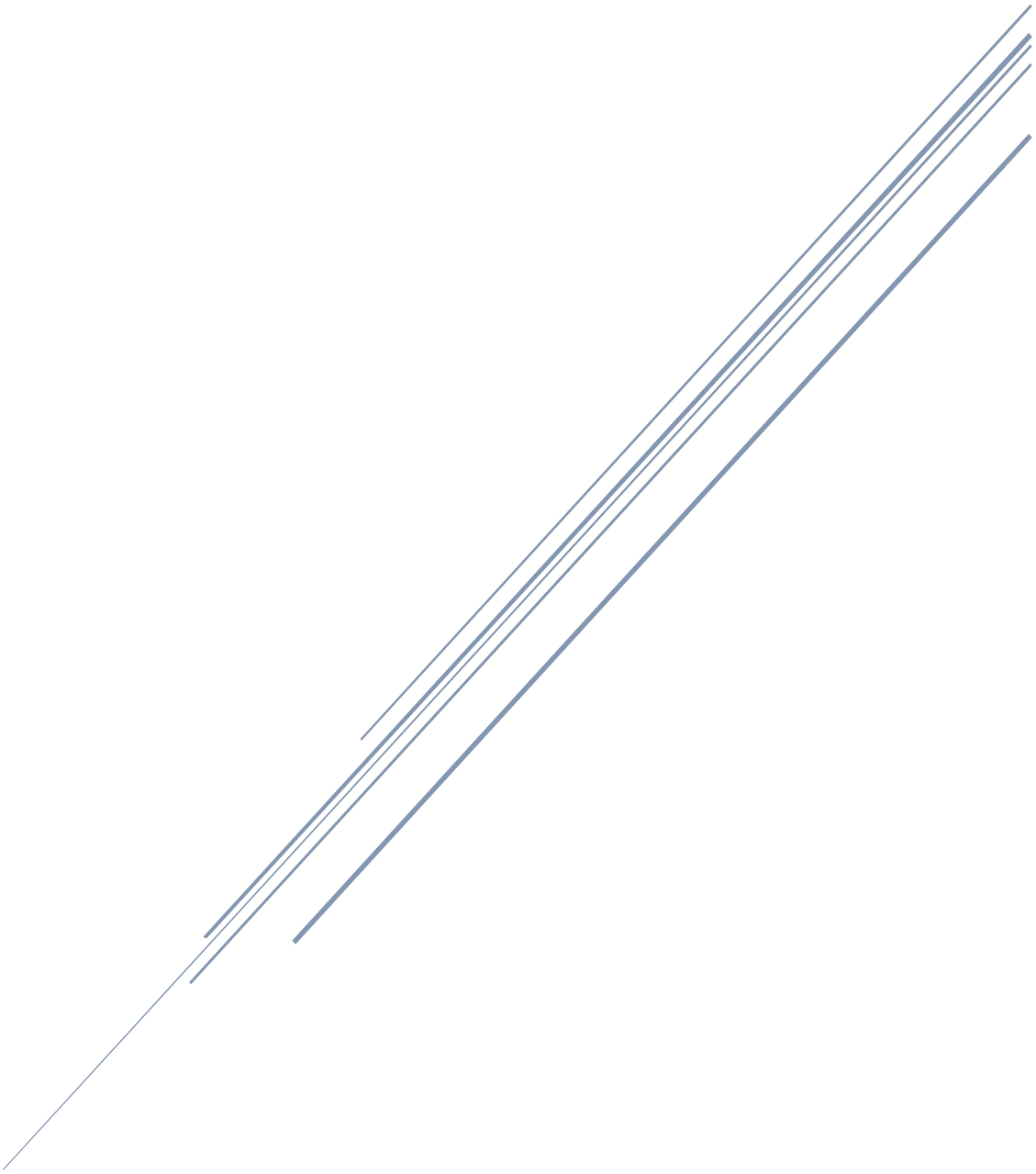
MACHINE LEARNING WITH PYTHON.

Stefan Weiß



Outline

Unleashing the Power of Machine Learning

"The only limit to our realization of tomorrow will be our doubts of today."

Franklin D. Roosevelt

Greetings, my name is Stefan Weiß, I am a 20-year-old student from Germany. I am welcoming you to a journey of innovation and discovery. In a world driven by technology, we stand at the threshold of endless possibilites. Our generation has the power to shape the future and unleash the potential of tomorrow. With curiosity as our compass, we´ll explore the uncharted territories of knowledge and creativity. The digital age offers opportunies beyond our wildest dreams.

In today´s fast-paced and data-rich world, machine learning is no longer just a buzzword. It´s a transformative force that´s reshaping industries, solving complex problems and fueling innovation.

Machine Learning is not just about algorithms and data. It´s about transforming the way we live, work and innovate. With the knowledge and skills gained in this journey, you will be able to understand the theory and code behind Machine Learning. I tried my best to explain everything the simplest way possible.
The birth of Machine Learning was back at the years of 1950s – 1960s, pioneers like Alan Turing and John von Neumann laid the theoretical foundations for artificial intelligence (AI) and machine learning.

Introduction

What is Machine Learning?

The use and development of computer systems that are able to learn and adapt without following explicit instructions, by using algorithms and statistical models to analyse and draw inferences from patterns in data.

Branches of Machine Learning:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning
- Neural Networks (Deep Learning)

The Problem:

- In ML we have a problem to solve. Normally the problem corresponds to evaluating some data and making predictions.

Tools:

- In order to solve those problems, we have a few tools. Those tools are called algorithms, for example:
  - Linear Regression
  - Classifcation Regression
  - Decision Trees

Evaluation metrics

- How do we know which tool works best for the problem? For that we have a bunch of measurement tools and we use them to evaluate all the algorithms. So once we evaluate the algorithms and their parameters and decide which ones are best for our problem, then we go ahead and solve the problem.

Supervised Learning

Im beginning with Supervised Learning as first topic in this journey. It´s the simplest way to get into machine learning and to understand the basics behind the processes.

What is Supervised Learning?

In Supervised Machine Learning, our algorithms learn from labeled data. After studying the labeled data, these techniques are able to determine which label should be given to new data based on observing patterns and associating those patterns to new unlabeled data.

Classifcation:

- Model that predict a category that an item belongs to. In some cases used for events with only two possible outcomes, like whether an email is spam or not, but also can be extended to predict any number of categories such as predicting which of many breeds a dog belongs to

- Answers questions of the form yes-no.
  Example:
  - Is this email spam or not
  - Is the patient sick or not

Regression:

- Model that predict a numeric value like home price or an individual´s height.
- Answers questions of the form how much
  Example:
  - How much does this house cost?
  - How many seconds do expect someone to watch this video?

Supervised Learning is used in a wide range of applications, including image classification, natural language processing, fraud detection, antonomous driving and healthcare diagnostics.

Algorithms:

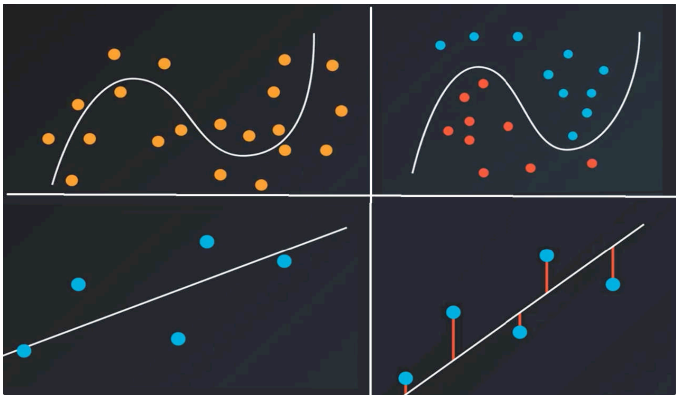
- Support-Vector machines
- Linear Regression
- Logistic Regression
- Naive Bayes
- Decision Trees
- K-nearest neighbor

Regression

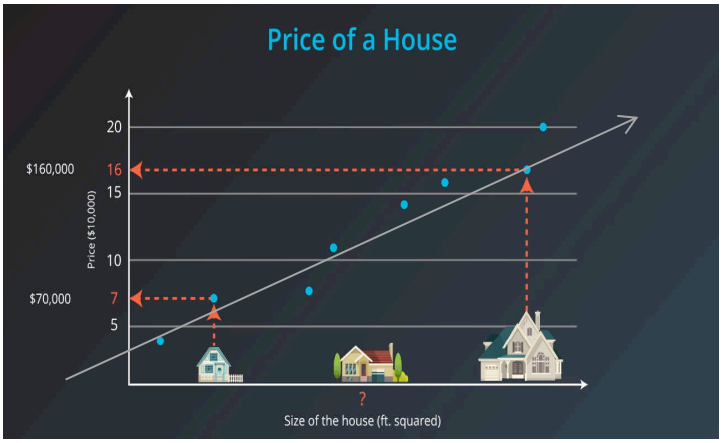
Linear Regression

Based around the idea of trying to draw a line (“fitting”) through an entire dataset of points. The algorithm uses the value of every point in the dataset to find the optimum line equation. Ultimately the equation of that line can be used to plot new data. Sometimes this is perfectly clear and other times it is quite challenging, it depends on the data in the dataset.

Examples for Linear Regression:



Example House Prices:



Question:

- What´s the best estimate for the price of the house?
  - $80.000
  - $120.000
  - $190.000

Answer:

$120.000. The point resemble a line. On this line, we can see that our best guess for the price of the house is point here over the line which corresponds to $120.000.

This method we call linear regression.

Optimization techniques

Absolute Trick

Let´s start with a point and a line:

- A point with coordinates (p,q)
- A line represented by $y = w1x + w2$

How do we move it closer to the point (p,q) with the absolute trick?

- Add to the y-intercept so that the line moves up
- Add to the slope to make the line rotate in the direction of the point

If we add 1 to the y-intercept and p to the slope, it´s giving us the equation

- y = (w1 + p)x + (w2 + 1)

That´s a too large step and we have over-corrected our line. Instead we add a small number called "learning rate" with the variable name **alpha (a),** to take smaller steps.

Equation with learning rate for above the line:

- y = (w1 + pa)x + (w2 + a)

Equation with learning rate underneath the line:

- y = (w1 – pa)x + (w2 – a)

The purpose of the learning rate is the amount by which we change the y-intercept and slope. It´s usually a small number

Square Trick

The difference to the Absolute trick is simple, both move the line, the Absolute trick moves the line by a constant value, whereas the Square Trick moves it more or less depending on the distance to the point from the line.

The vertical distance between the point and the line, the point over the line has coordinates (p,q) and the corresponding point on the line is (p,q'). The distance between the point and the line (q - q').

We take this distance and multiply it into what we add to both the y-intercept and to the slope.

- Update the y-intercept by adding a(q-q')
- Update the slope by adding pa(q-q')

This gives us the equation

- y = (w1 + p(q-q')a)x + (w2 + (q-q')a)

This trick automatically takes care of points that are under the line and we don´t need two rules as we had on the absolute trick. We just have the same rule for both.

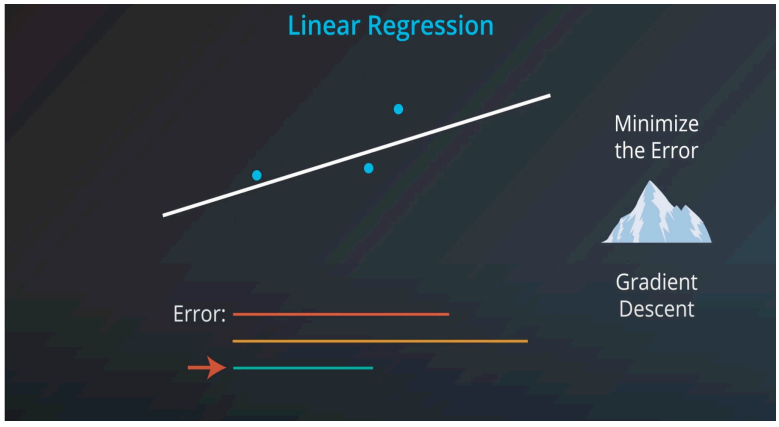
Gradient Descent

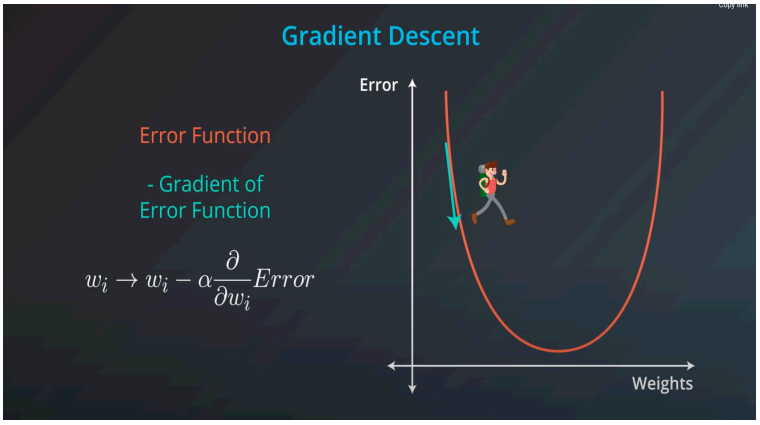
Gradient Descent

Let´s say we have our points on our plan is to develop an algorithm that will find the line that best fits this set of points. And the algorithm works like this

- First draw a random line and calculate the error. The error is some measure of how far the points are from the line, it could be any measure that tells us how far we are from the points
- Now move the line around and see if we can decrease this error.
  - We move in this direction and we see that the error kind of increases so that´s not the way to go.
  - We move in the other direction and see that the error decreased, so we pick this one and stay there
- Repeat these steps many times over and over every time descending the error a bit until we get to the perfect line.

To minimize this error, we are going to use something called "**gradient descent**".
Gradient Descent is a strategy that helps to minimize the error between to points of the actual data and the "best-fit line". We use gradient descent to update the parameters of our model as we train.





What is the meaning of the word "gradient descent"?

- It is the reduction of the error by taking the derivative of the error function with respect to the weights.

Error Functions

Mean Absolute Error

The two most common error functions for linear regression are:

- Mean absolute error
- Mean squared error

Mean Absolute Error

The mean absolute error is the sum of all the errors divided by m (points). It is also the average error of all points.

Let´s say we have a point with coordinates (x, y) and the line is called Y* since it is our prediction. The corresponding point on the line is (x, y*) and the vertical distance from the point to the line is (y – y*). This is the **error.**

Our total error is going to be the sum of all these distances for all the points in our dataset.

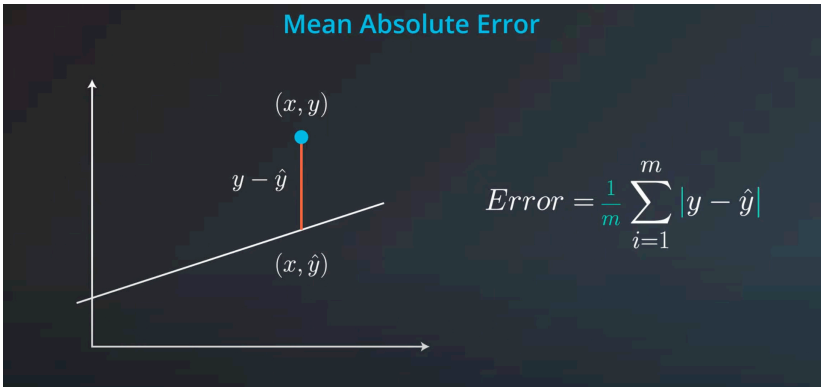
- $Error = \sum_{i=1}^{m} |y - y^{\wedge}|$

In some cases, we will use the average or the mean absolute error, which is the sum of all the errors divided by m, m is the number of points in our dataset

- $Error = m1 \sum_{i=1}^{m} |y - y^{\wedge}|$

Using the sum or the average won´t change our algorithms, since that would only scale our error by a constant, namely m.

We have an absolute value around y – y*, the reason is that if the point is on top of the line, the distance is y-y*, but if it´s under the line then it is y* - y. Error should always be positive, otherwise negative errors will cancel with positive errors.

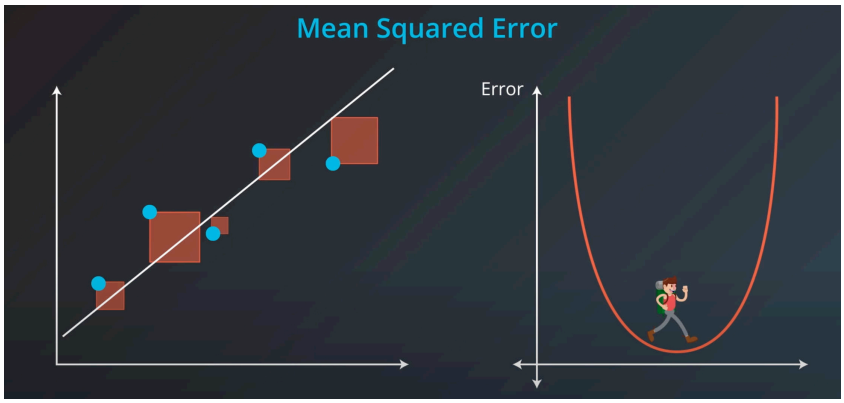


Mean Squared Error

It is very similar to the Mean Absolute Error, but instead of taking the distance between the point and the prediction, we are going to draw a square with this segment as its side. This area is $(y - y^*)^2$.
Notice that this is always non-negative, so we don´t need to worry about absolute values.

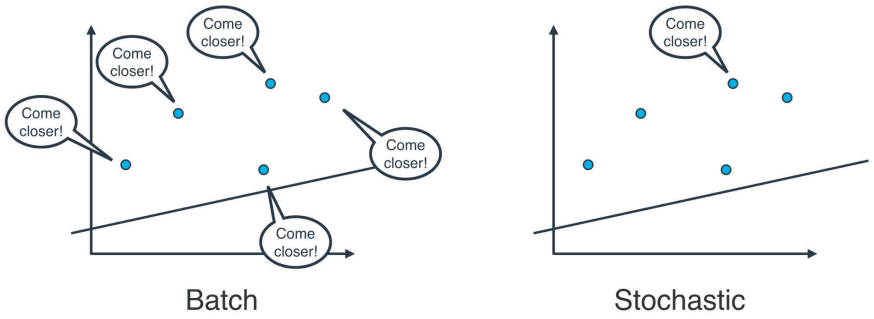
Our mean squared error is going to be the average of all these series of squares.

- $Error = 2m1i = 1\sum m(y - y\hat{})^2$

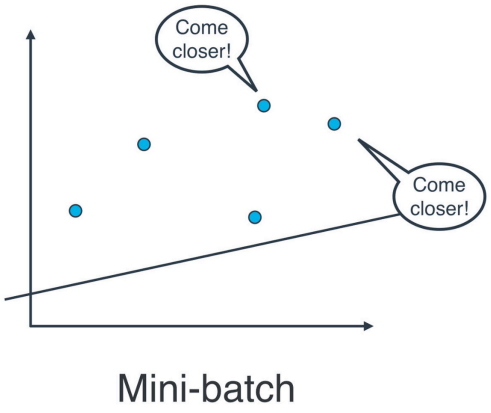


Mini-batch Gradient Descent

We don´t use those two, cause if our data is huge, both are a bit slow. The best way to linear regression is to split your data into many small batches. Each batch with roughly the same number of points. Then use each batch to update your weights. This is called “mini-batch gradient descent”.



Mini-batch gradient descent

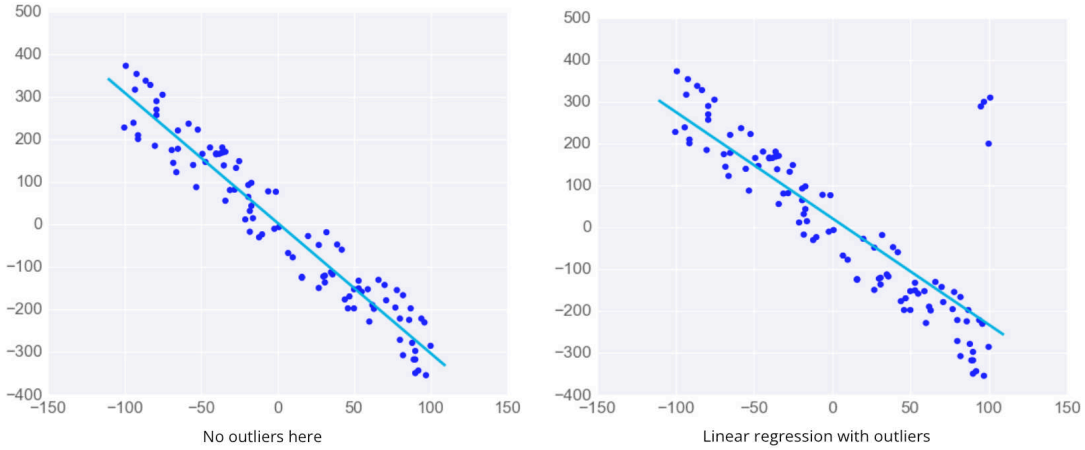


Linear Regression Warnings

Linear Regression works best when the data is linear. It produces a straight line model from the training data. If the relationship in the training data is not really linear, you´ll need to either make adjustments (like transforming training data), add features, or use another kind of model.

Linear Regression is sensitive to outliers. It tries to find a 'best fit' ling among the training data. If the dataset has some outlying extreme value that don't fit a general patter, they can have a suprisingly large effect.

Left with no outliers and right with outliers.



Examples for Linear Regression:

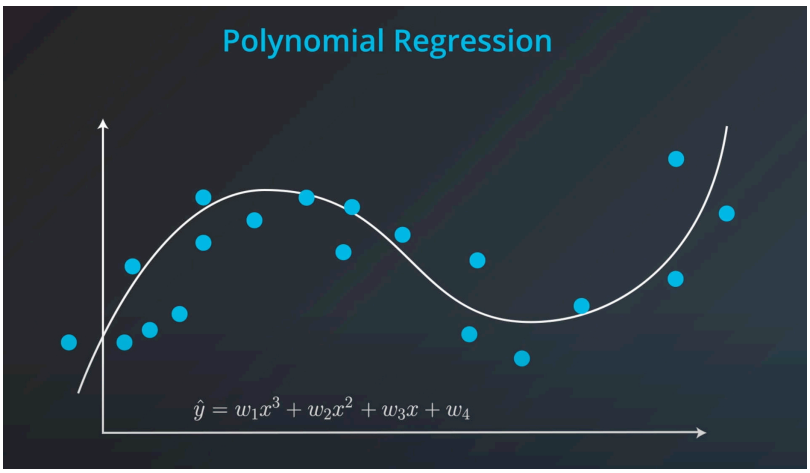
- Predicting the best stock price for an IPO
- Predicting the cost of college tuition for a new college in a new city

Examples for Not Linear Regression:

- Finding clusters of customers who reserve an Uber in the evening
- Deciding if an email is spam or not.

Polynomial Regression

We use this when a line won´t do a good job fitting in our data. This can be solved using a very similar algorithm to linear regression. Instead of considering lines, we consider "**higher degree polynomials**". This would give us more weight to solve our problem.



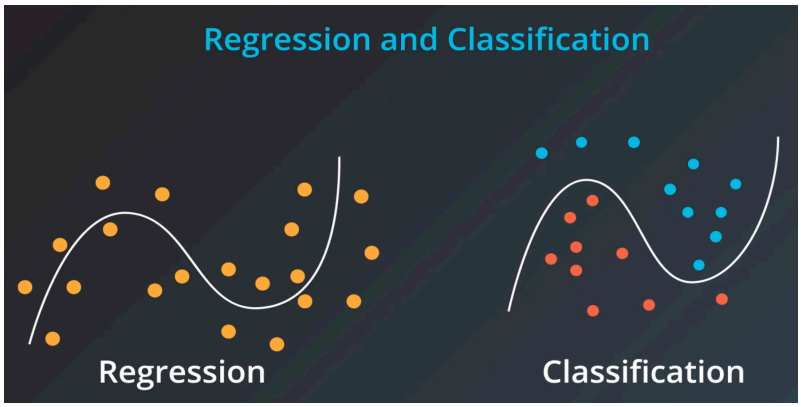
This algorithm is known as **polynomial regression**

Linear vs Polynomial Regression:

- Linear Regression assumes a linear relationship between the dependent and independent variables and models it as a straight line (y = mx + b)
- Polynomial Regression allows curved relationships between variables by using polynomial functions of different degrees to fit the data.

Regularization

Technique used to prevent overfitting, improve the generalization of the model and make them more robust. Regularization methods add a penalty term to the model´s objective function, encouraging it to have smaller or more stable coefficients, which helps control complexity and reduces the risk of overfitting.



Regularization through a classifcation problem

Left: Line -> Makes couple of mistakes
Right: Higher degree polynomial curve -> Zero mistakes but is more complicated

L1 regularization:

It takes the absolute value of the coefficients of the model as a penalty term to the objective function.

For example:

- Polynomial Model:
  - $2x_1^3-2x_1^2x_2-4x_2^3+3x_1^2+6x_1x_2+4x_2^2+5=0$ -> (2+2+4+3+6+4)=21

- Linear Model:
  - $(3x_1 + 4x_2 + 5 = 0)$ -> (3 + 4 = 7)

L2 regularization

Instead of adding the absolute values, we add the squares of the coefficients.

For example:

- Polynomial Model
  - $2x_1^3-2x_1^2x_2-4x_2^3+3x_1^2+6x_1x_2+4x_2^2+5=0$ -> $(2^2+-2^2+-4^2+3^2+6^2+4^2)=85$

- Linear Model
  - $(3x_1 + 4x_2 + 5 = 0)$ -> $(3^2 + 4^2 = 25)$

Lambda

If we punish the complicated model too little, or too much we can use **lambda** to tune or alter the amount that we want to punish the complex model.

- With a small lambda, the error that comes from the complexity of the model is not large enough to overtake the errors in the simplified model misclassifying points, so we will choose the complet model.
- With a large value for lambda, we´re multiplying the complexity part of the error by a lot. This punishes the complex model more so the simple model wins.

If we have a large lambda then we´re punishing complexity by a large amount and we´re picking a simpler model. Whereas if we have a small lambda, we´re punishing complexity by a small amount, so we are okay with having more complex models.

L1 Regularization	L2 Regularization
Computationally Inefficient (unless data is sparse)	Computationally Efficient
Sparse Outputs	Non-Sparse Outputs
Feature Selection	No Feature Selection

Cheat Sheet

- Efficiency
  - **Contra:** L1 regularization is actually computationally inefficient even though it seems simpler because it has no squares, but actually, those absolute values are hard to differentiate.
  - **Pro:** L2 regularization squares have very nice derivatives. These are easy to deal with computation

- Spare Data
  - **Pro:** L1 regularization is faster than L2 regularization. If you have a thousand columns of data but only 10 are relevant and the rest are mostly zeros, then L1 is faster.
  - **Pro:** L2 is better for non-sparse outputs which are when the data is more equally distributed among the columns

- Feature selection
  - **Pro:** L1 has one huge benefit which is that it gives us feature selection. So let´s say, we have again, data in a thousand columns but really only 10 of the matters, and the rest are mostly noise. So, L1 will detect this and will make the relevant columns into zeroes.
  - **Con:** L2 on the other hand won´t do this and it just takes the columns and treat them similarly.

- **Gradient descent** is a method to optimize your linear models.
- **Multiple Linear Regression** is a technique for when you are comparing more than two variables.
- **Polynomial Regression** for relationships between variables that aren´t linear
- **Regularization** is a technique to assure that your models will not only fit the data available but also extend to new situations.

Key Term	Definition
Batch gradient descent	The process of repeatedly calculating errors for all points at the same time and updating weights accordingly.
Error	The vertical distance from a given point to the predictive line.
Feature scaling	Transforming data into a common range of values using standardizing or normalizing.
Gradient descent	The reduction of the error by taking the derivative of the error function with respect to the weights.
L1 Regularization	Absolute values of the coefficients of the model are used for regularization.
L2 Regularization	Squares of the values of the coefficients of the model are used for regularization.
Lambda	The amount by which we punish complex models during the process of regularization.
Learning rate	The amount by which we adjust the weights of our equation. The larger the learning rate, the larger our adjustments.
Mean absolute error	The sum of the absolute value of all errors divided by the total number of points.
Mean squared error	The sum of the square of all errors divided by the total number of points.
Regularization	Taking into consideration the complexity of the model when evaluating regression models.
Stochastic gradient descent	The process of repeatedly calculating errors one point at a time and updating weights accordingly.

Perceptron Algorithm

Classification Problems

A predictive modeling problem where a predefined class label is predicted based on training for a given dataset.

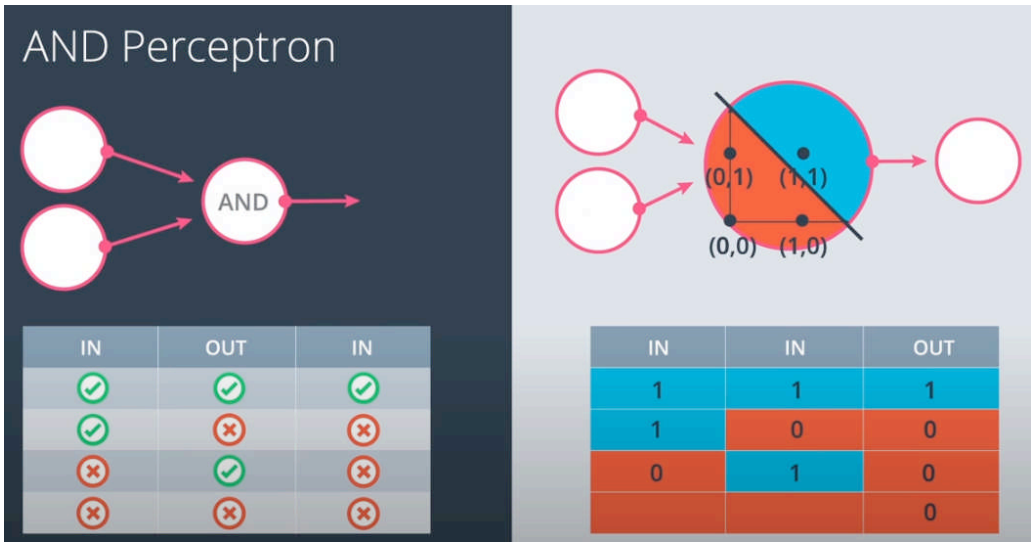
Perceptrons

It is the building block of nerual networks, and it´s an encoding of our equation into a small graph.

Logical operators as perceptrons

AND operator:

- It takes two inputs and returns an output



What is the purpose of the learning rate?

It´s a hyperparameter value that is used in the calculation that controls how fast/far the model adjusts during training. It is used to move the line closer towards the points.
One important aspect of the Perceptron Trick is, that it uses the coordinates of misclassified points to adjust the parameters in the line equation. To keep the adjustments small, the adjustment calculations are multiplied by the Learning Rate.

Key Term	Definition
Classification problem	A predictive modeling problem where a predefined class label is predicted based on training for a given dataset.
Linear boundary	A boundary line that separates data into two groups.
Perceptron	Building block of neural networks; an encoding of an equation into a small graph.
Step function	Function that takes a value and returns a 1 if the input is positive or 0, and returns a 0 if the input is negative.

Decision Trees

Decision Tree is a supervised machine learning algorithm, used for both classification and regression tasks.

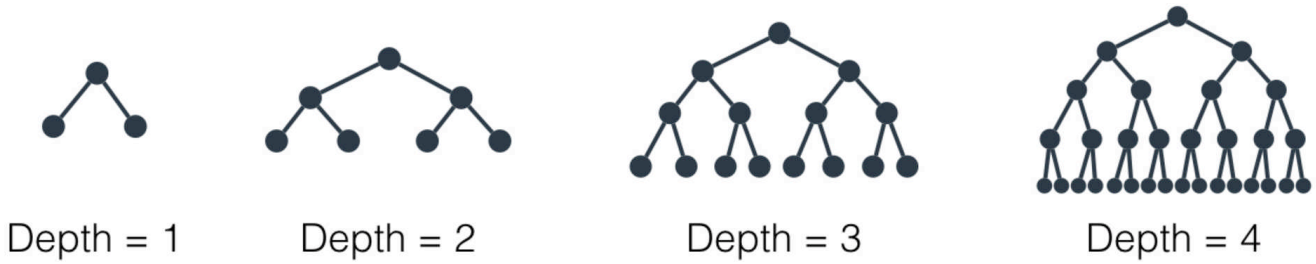
- **Root Node:**
  - The top node is called the root node and contains all the data in the dataset

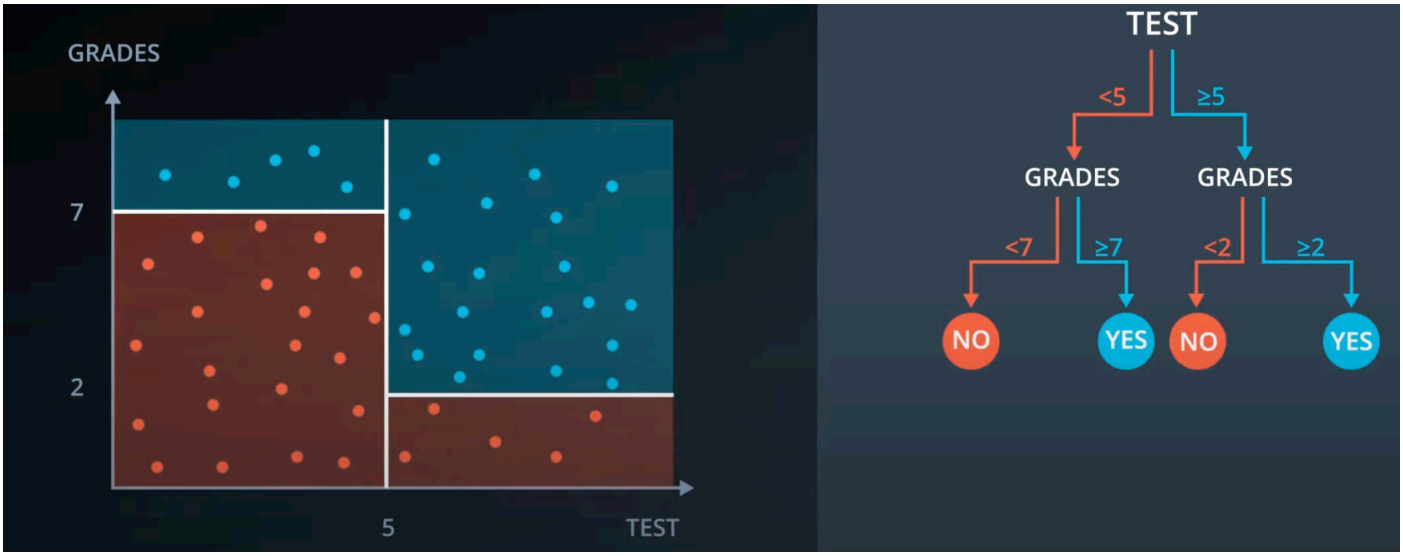
- **Splitting:**
  - The tree splits the data into subsets based on feature´s value. The feature and value that provides the best split (maximizing information gain or minimizing impurity) are chosen at each node
  - For classification commonly used measures include Gini impurity and entropy
  - For regression commonly used measure is the **Mean Squared Error (MSE)**

- **Internal Nodes:**
  - Internal Nodes represent decisions based on features. For example, in a decision tree for classifying animals, an internal node might ask whether the animal has fur or not

- **Leaf Nodes:**
  - Leaf Nodes represent the final class label (classification) or regression value (regression) assigned to a sample.
  - For classification each leafe node corresponds to a class label. For regression, it contains a numeric prediction.

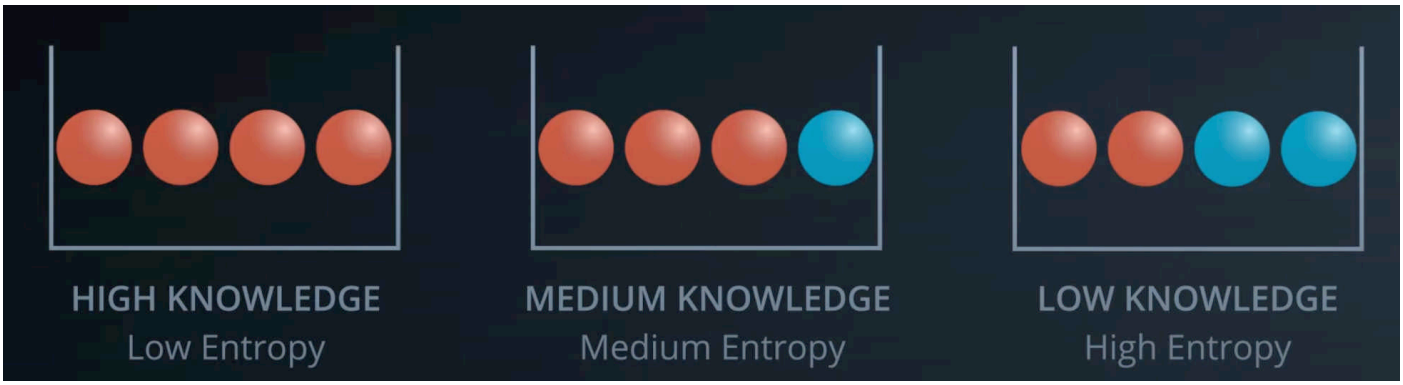
- **Decision Making:**
  - To make predictions, start at the root node and follow the path down the tree based on feature value.
  - At each internal node, evaluate the feature value for the sample and choose the branch that matches the condition.
  - Continue until a leaf node is reached and assign the class label or regression value of that leaf to the sample

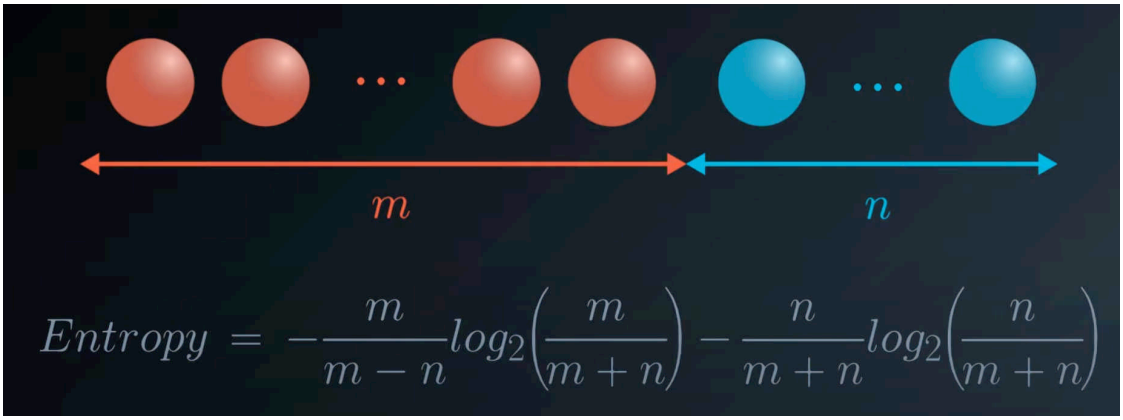




Entropy

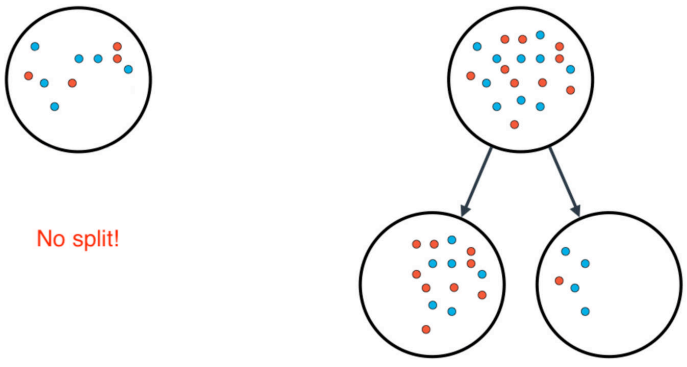
It´s like a measure of surprise or disorder. In the context of a dataset, it tells you how mixed or uncertain the data is. Low entropy means the data is predictable and well organized, while high entropy indicated unpredictability and randomness.





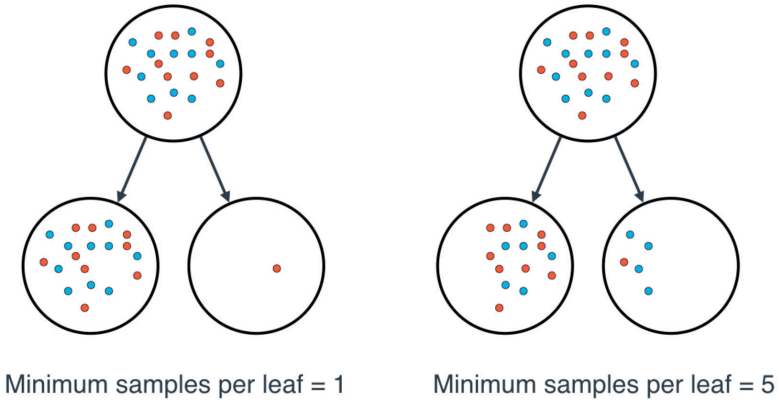
Hyperparameters

- Minimum number of samples to split.
  - A node must have at least **'min_samples_split'** samples in order to be large enough to split. If a node has fewer samples than **'min_samples_split' samples, it will not be split.**



Minimum number of samples to split = 11 Minimum number of samples to split = 11

- Minimum number of samples per leaf.
  - When splitting a node, one could run into the problem of having 99 samples in one of them, and 1 on the other. This will not take us too far in our process and would be waste of resources and time. If we want to avoid this, we can set a minimum for the number of samples we allow on each leaf. The function is **'min_samples_leaf'**



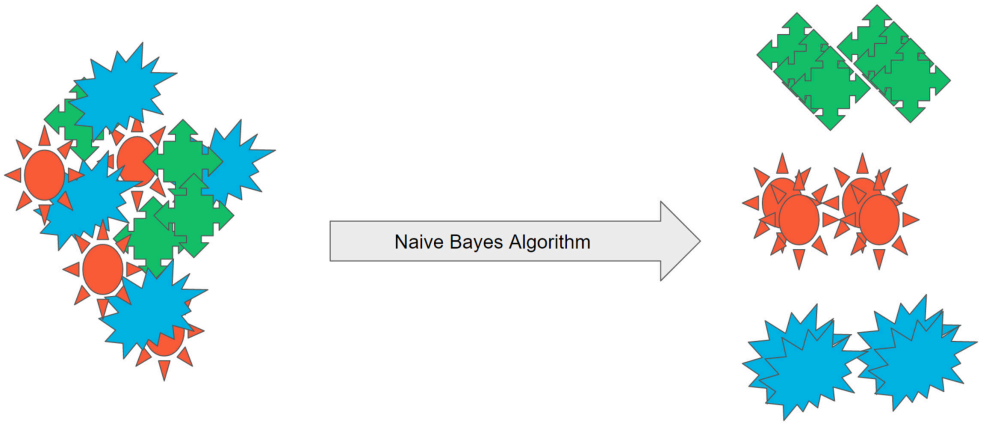
- Small maximum depth -> Underfitting
- Large maximum depth -> Overfitting
- Small minimum samples per split -> Overfitting
- Large minimum samples per split -> Underfitting

Key Term	Definition
Entropy	The amount of freedom that a data point has to move around.
Hyperparameter	Different aspects of a decision tree, such as depth, that can be tuned to create decision trees that generalize well to new problems.
Information gain	Difference in entropy between the parent node and the average entropy in the children in a decision tree.

Naive Bayes Algorithm

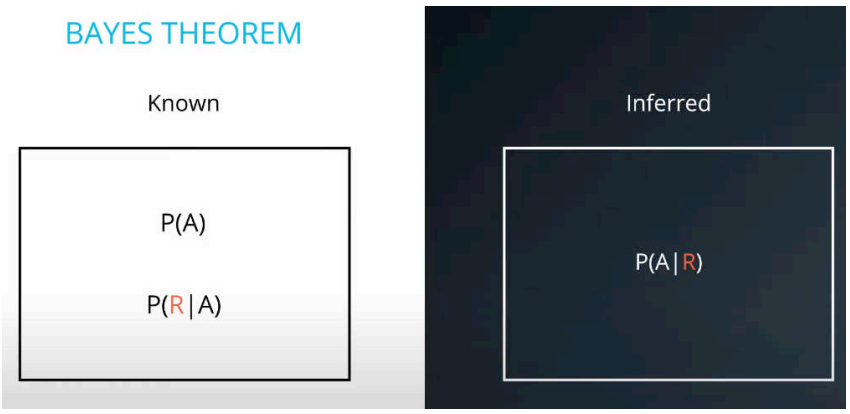
Naive Bayes

It´s a supervised machine learning algorithm that can be trained to classify data into multi-class categories. In the heart of Naive Bayes algorithm is the probabilistic model that computes the conditional probabilities of the input features and assigns the probability distributions to each of the possible classes. This algorithm has great benefits such as **being easy to implement** and **very fast to train**.



- **Prior:**
  - Refers to guesses we make before having complete information
- **Posterior:**
  - Refers to guesses we´ve inferred after the new information has arrived.

Bayes theorem



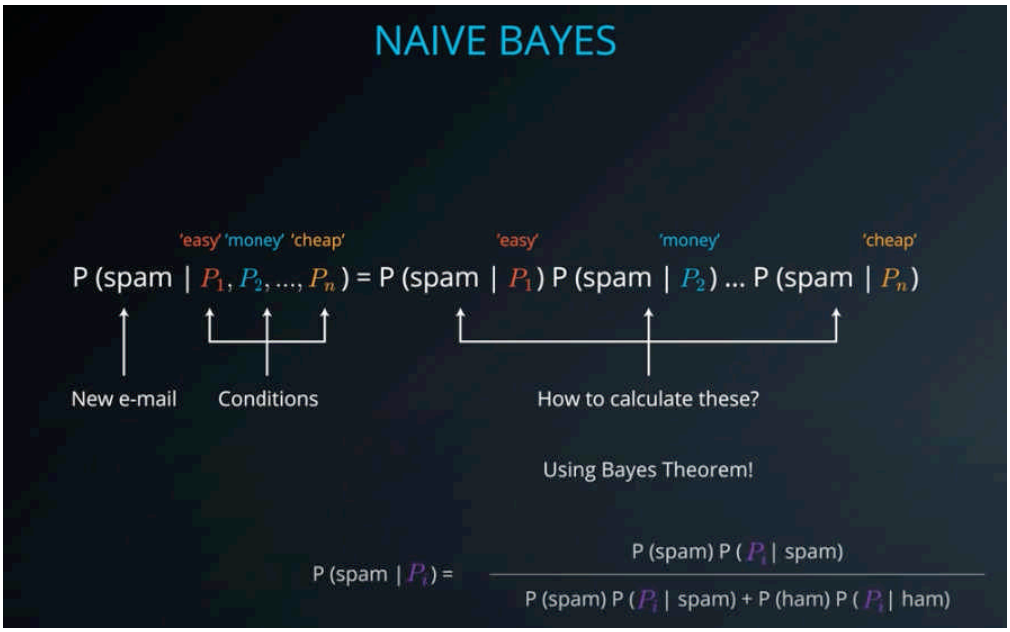
What Bayes theorem does is from these two, it infers the probability of A given R
Based on information that is known, it can infer other information.

Conditional Probability

In Naive Bayes we assume that our probabilities are independent.

Conditional probability

We have $P(A|B)$ to be proportional to P(B|A)P(A)$P(B|A)P(A)$.



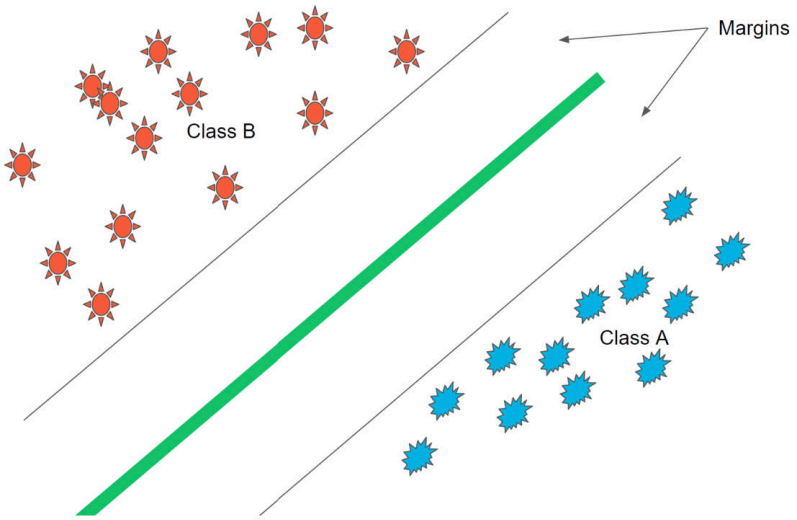
Advantages of using Naive Bayes:

- It has over other classifcation algorithms is its ability to handle an extremely large number of features
- It performs well even with presence of irrelevant features and is relatively unaffected by them. It has its relative simplicity
- It works well right out of the box and tuning its parameters is rarely ever necessary, excpet usually in cases where the distribution of the data is known
- It rarely ever overfits the data
- Its model training and prediction times are very fast for the amount of data it can handle

Key Term	Definition
Conditional probability	In probability theory, conditional probability is a measure of the probability of an event occurring given that another event has (by assumption, presumption, assertion
Naive assumptions	The assumpiot that assume probabilities are independent.
Posterior probabilities	Posterior probabilities are what we inferred after we knew that R occurred
Prior probabilities	Prior probabilities are what we knew before we knew that R occurred.
Sensitivity	How often a test correctly gets a positive result for the condition that's being tested for (also known as the "true positive" rate).The true-positive recognition rate
Specificity	The proportion of truly negative cases that were classified as negative. The true-negative recognition rate

Support Vector Machines (SVM)

SVM is a powerful algorithm for classification which also finds the best boundary.
We want the boundary to be as far away from the points as possible.
The error for the SVM algorithm is Classification Error + Margin Error



Calculating errors for misclassified points

Adding the errors of each individual misclassified point gives us 2.5 + 0.5 + 1 + 2 = 6. So the error is 6

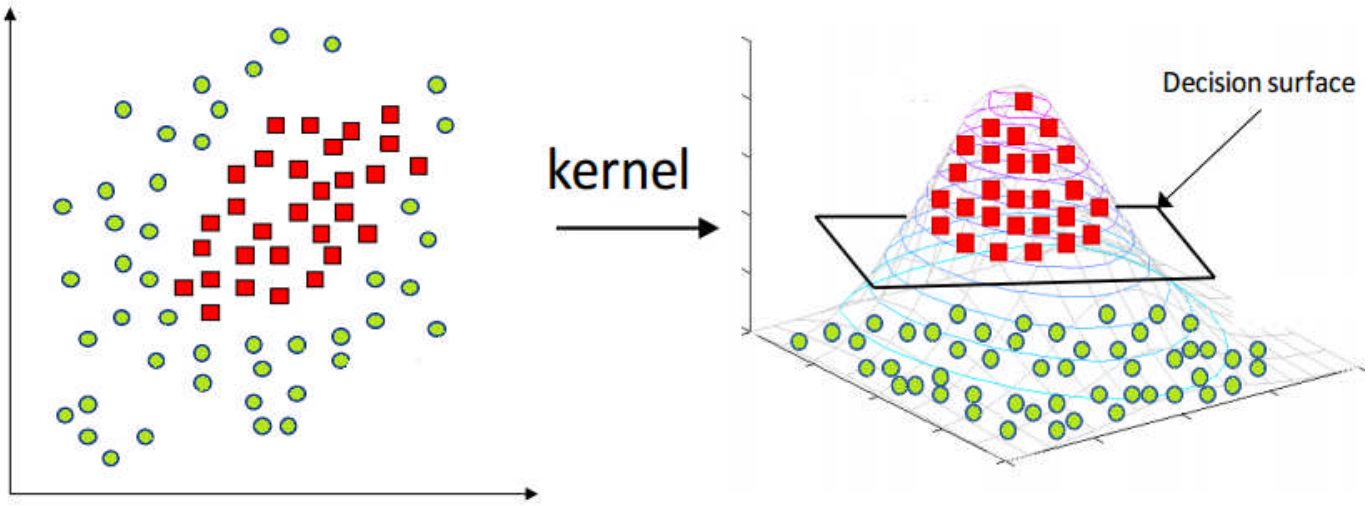
Margin Error

- Large margin = small error
- Small margin = large error

It can be minimized by using gradient descent.

Kernel Trick

Powerful technique for handling non-linear classification problems by implicitly transforming data into high-dimensional feature spaces using kernel functions.

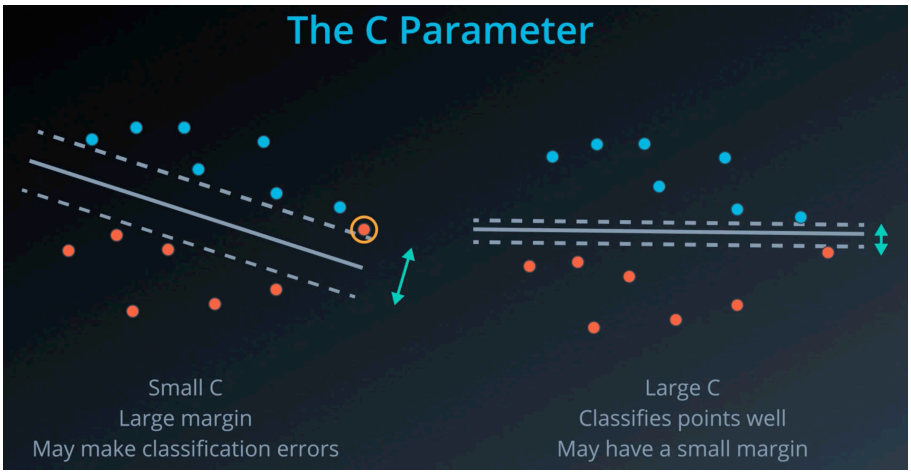


C Parameter & Gamma

The C Parameter

It´s just a constant that attaches itself to the classification error by multiplying the classification error by the constant.

- It is used to modify the classification error
- It is a hyperparameter that provides some flexibility during training
- A large value for C will usually result in a small margin
- A small value for C will usually result in a large margin



Gamma

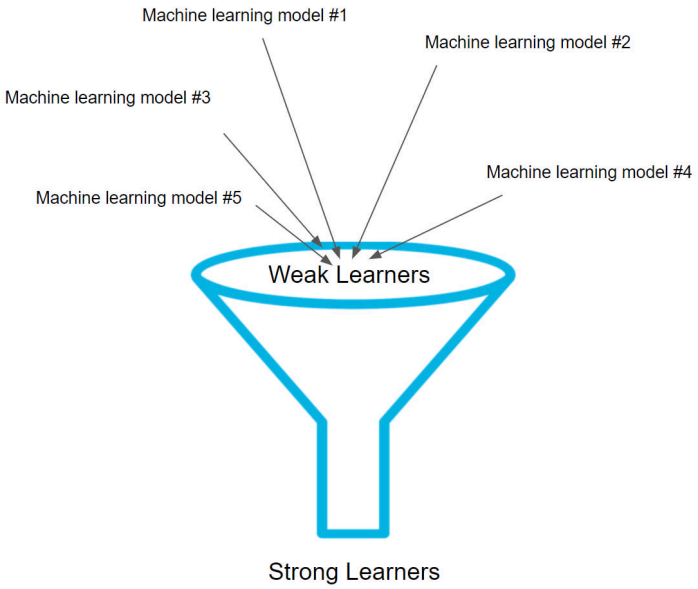
- If gamma is small, then sigma is large and the curve is wide
- If gamme is large, then sigma is small and the curve is narrow

Key Term	Definition
C	Also referred to as the C hyper-parameter - determines how flexible we are willing to be with the points that fall on the wrong side of our dividing boundary.
Classification error	An error in prediction caused by a false negative response or a false positive response.
Error function	Classification error + margin error. Used to minimize SVG
Gamma	A hyperparameter that we tune during training to alter the type of curve from wide ones to very narrow.
Kernal Trick	A more efficient and less expensive way to transform data into higher dimensions, especially in the SVM algorithm
Marthin error	The distance between the 2 boundary lines and prediction line. Use to minimize gradient descent
Ploynomial Kernel	A kernel function commonly used with SVM that allows learning of non-linear models.
RBF	Radial Basis Functions are used in machine learning to find a non-linear classifier or regression lines, especially in SVM.

Ensemble Methods

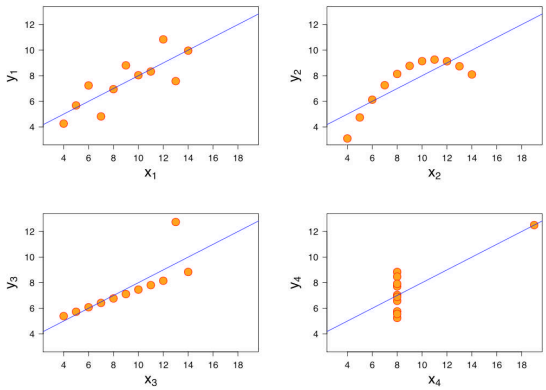
What is Ensemble Methods?

It´s about bringing together multiple models (**Weak Learners**) so that the result is an incredibly powerful and more accurate model (**Strong Learner**). The idea behind ensemble methods is that by combining the strengths of multiple models, they can compensate for each other´s weaknesses and produce more accurate and stable predictions. They are particularly effective in classification and regression tasks.



To find a well-fitting machine learning model, there are two competing variables.

- **Bias**
  - When a model has a high bias, this means that it doesn´t do a good job of bending to the data. Linear regression is an example of an algorithm that usually has a high bias. When a model have high bias, it´s bad.



- **Variance**
  - When a model has high variance, it means that it changes drastically to meet the needs of every point in our dataset. Linear models like the one above has low variance, but high bias. An example of an algorithm that tends to have high variance and low bias is a decision tree. A decision tree, as a high variance algorithm, will attempt to split every point into its own branch if possible. This is a trait of high variance, low bias algorithms - they are extremely flexible to fit exactly whatever data they see.

Examples Ensemble Methods

Randomness into Ensembles

Another method to improve ensemble methods is to introduce randomness into high variance algorithms before they are ensembled together. There are two main ways that randomness is introduced:

- Bootstrap the data
  - Sampling the data with replacement and fitting your algorithm to the sampled data.

- Subset the features
  - In each split of a decision tree or with each algorithm used in an ensemble, only a subset of the total possible features are used.

Decision Trees

They tend to overfit a lot, but how we solve this? Don´t create a decision tree from all columns, pick some columns randomly of the data and build a decision tree in those columns, and so on.
This is called **random forest.**

Bagging

- Voting:
  - It´s the last step after training and is used to combine the weak learne results. We over impose each learner on the data.
    - If we have 3 examples:
      - 2 or more blue -> region is blue
      - 2 or more red -> region is red

ADABoost

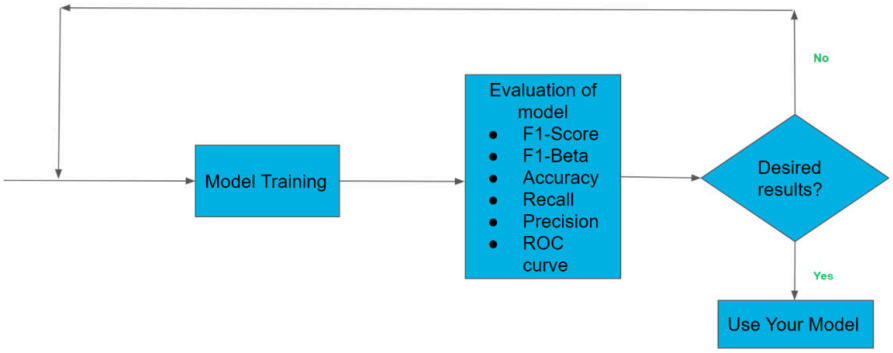
3 Steps of the AdaBoost:

- 1. Maximize accuracy, minimize errors
- 2. Identify misclassified points from previous step and fix the mistakes.
- 3. Try to classify points identified in the previous step

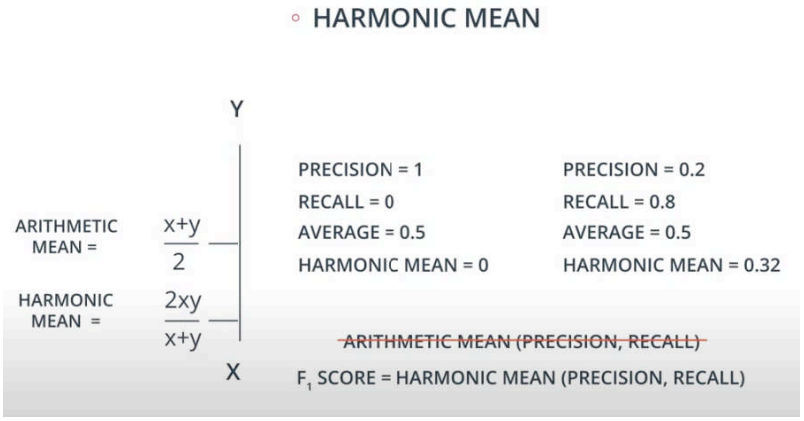
Key Term	Definition
AdaBoost	(Ada)ptive (Boost)ing, is an ensemble ethod technique that re-assigns weights to each instance, with higher weights to incorrectly classified instances.
Bagging	(B)ootstrap (agg)regating is an ensemble algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression.
Ensembles	You can combine (or ensemble) models in a way that makes the combination of these models better at predicting than the individual models.
Random forest	Using 2+ decision trees on randomly picked columns.

```python
# Import necessary libraries
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Create a synthetic dataset for classification
X, y = make_classification(n_samples=1000, n_features=20, random_state=42)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create a weak learner (decision tree) as the base estimator
base_estimator = DecisionTreeClassifier(max_depth=1)

# Create an AdaBoost classifier with the base estimator
# You can specify the number of weak learners (n_estimators) and other hyperparameters
adaboost_classifier = AdaBoostClassifier(base_estimator=base_estimator, n_estimators=50, random_state=42)

# Train the AdaBoost classifier on the training data
adaboost_classifier.fit(X_train, y_train)

# Make predictions on the test data
y_pred = adaboost_classifier.predict(X_test)

# Calculate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

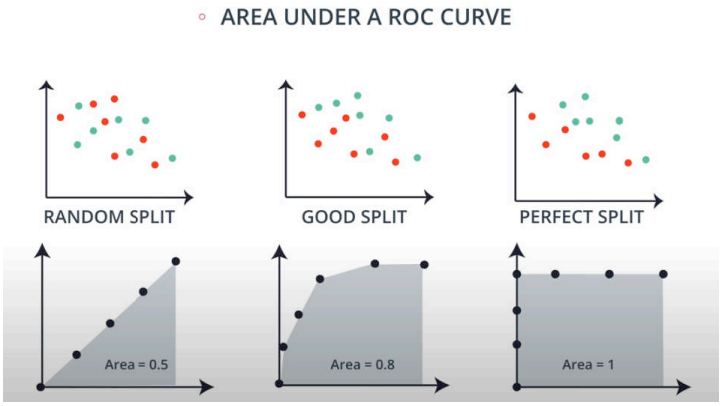
Model Evaluation Metrics



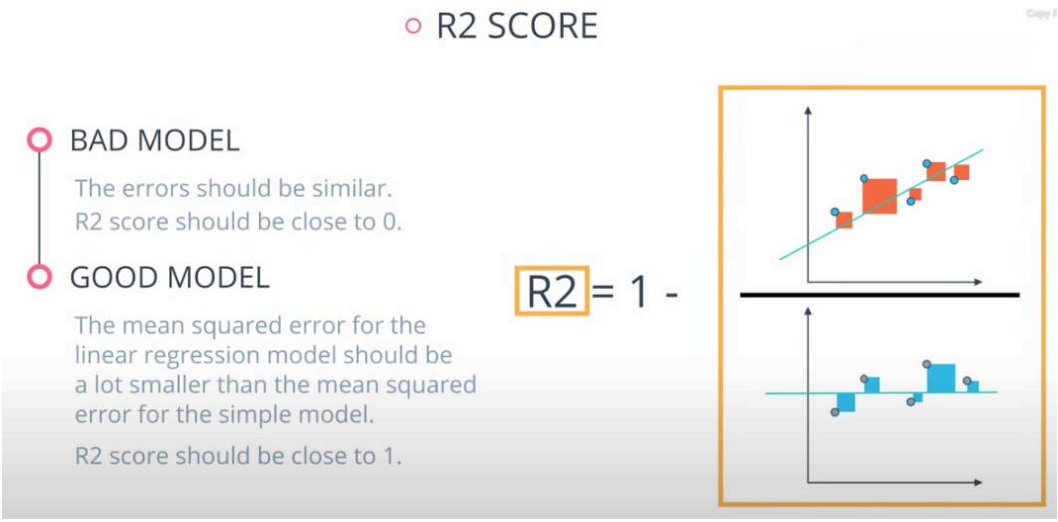
- Prediction:
  - Out of all the points predicted to be positive, how many of them were actually positive?
- Recall:
  - Out of the points that are labeled positive, how many of them were correctly predicted as positive?
- F1 Score:
  - It will always be closer to the larger result of precision and recall values.
  - F1 Score = 2 * (Precision * Recall) / (Precision + Recall)



ROC Curve

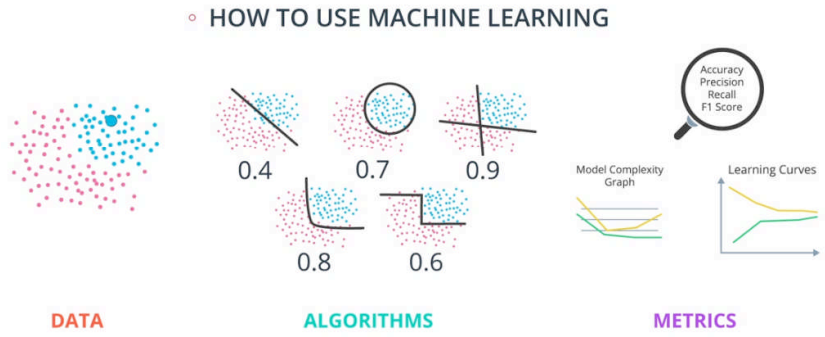


Regression Metrics



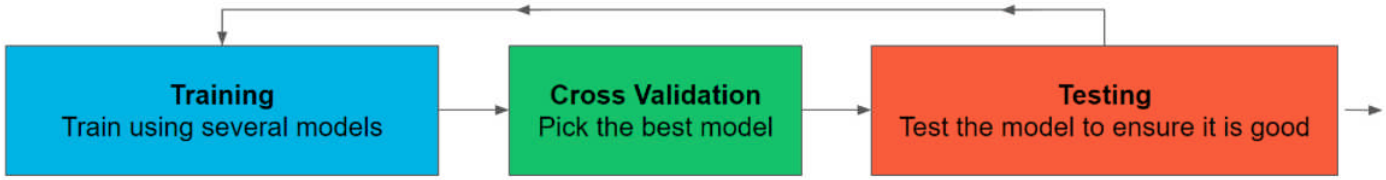
Calculating R2 Score in Python:

```
from sklearn.metrics import r2_score
y_true = [1, 2, 4]
y_pred = [1.3, 2.5, 3.7]

r2_score(y_true, y_pred)
```



Key Term	Definition
Accuracy	Accuracy is the answer to the question, Out of all the patients, how many did we classify correctly?
F1-score	Metric that conveys the balance between the precision and the recall.
Mean Absolute Error(MAE)	Regression metric that adds the absolute values of the distances from the points to the line.
Mean-Squared Error (MSE)	The most used metric for optimization in regression problems that adds the squares of the distances between the points and the line.
Precision	precision will be the answer to the question, Out of all the points predicted to be positive, how many of them were actually positive?
R2	Regression metric that represents the 'amount of variability captured by a model, or the average amount you miss across all the points and the R2 value as the amount of t

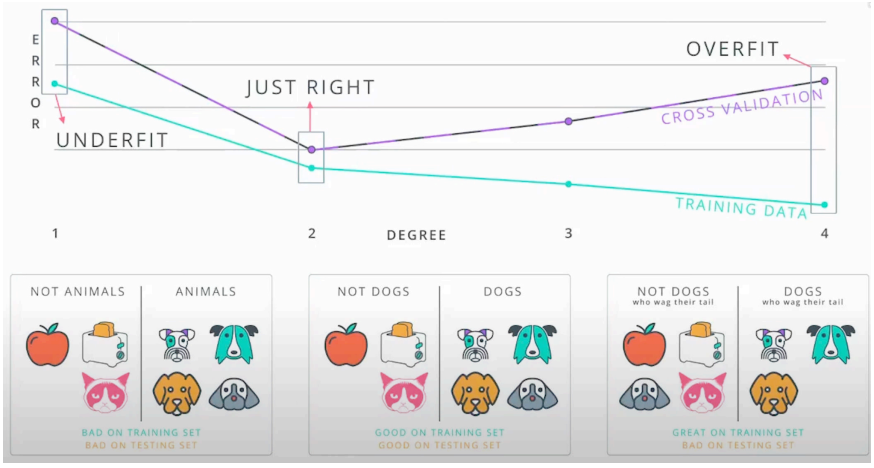
Training and Tuning



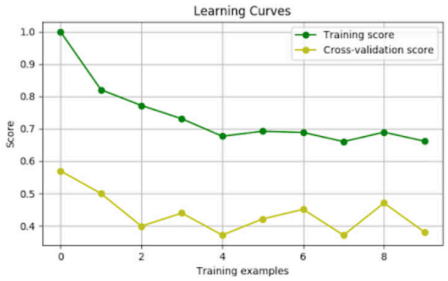
- Underfitting:
  - We **oversimplify** the problem
  - Does not well in the training set
  - Error due to bias

- Overfitting:
  - We **overcomplicate** the problem
  - Does well in the training set, but tends to memorize it itnstead of learning the charasteristics of it
  - Error due to variance

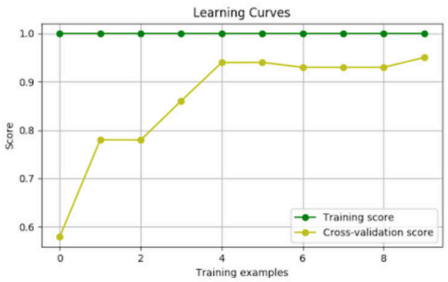
MARK: NEVER USE TESTING DATA FOR TRAINING



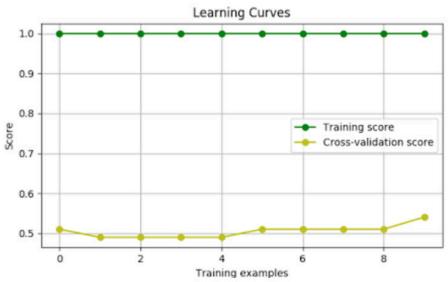
Training and Tuning



Logistic Regression

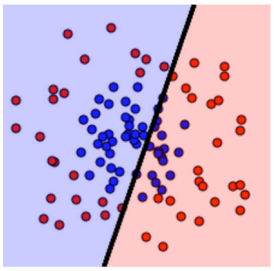


Decision Tree

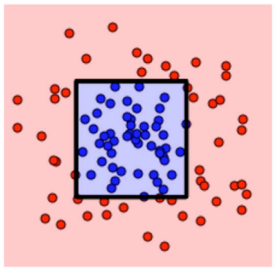


Support Vector Machine

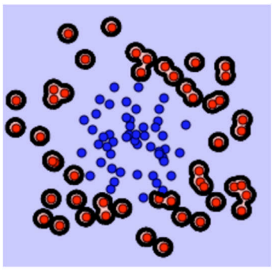
- **Logistic Regression** model has a low training and testing score
- **Decision Tree** model has a high training and testing score
- **Support Vector Machine** model has a high training score and a low testing score



Logistic Regression
(Underfitting)



Decision Tree
Just Right



Support Vector Machine
(Overfitting)

- **Logistic Regression** model uses a line, which is too simple. It doesn't do very well on the training set -> It **underfits**.
- **Decision Tree** model uses a square, which is a pretty good fit and generalizes well -> **Good** model.
- **Support Vector Machine** model actually draws a tiny circle around each point. This is clearly just memorizing the training set and won´t generalize well -> It **overfits**

Key Term	Definition
Bias	The bias is known as the difference between the prediction of the values by the ML model and the correct value. High bias reslts in a large error in training as well as testing data.
Grid search	A table with results of all probabilities and the best result is used.
K-fold cross validation	A parameter called 'k' that represents the number of groups that a given data sample is to be split into. Then the results for all subsets are averaged.
Learning curves	A tool used in machine learning that learns from a training dataset incrementally, and the plot shows changes in learning performance over time.
Overfitting	When a model uses a lines which is too complex, as if memorizing the training set, and won't generalize well. Over-complication of the problem
Underfitting	When a model uses a line, which is too simplistic. It doesn't do very well on the training set. Oversimplification of the problem
Variance	iA type of error due to a model's sensitivity to small fluctuations in the training set. High variance would cause an algorithm to model the noise in the training set. This is known as

Neural Networks

A neural network is a computational model inspired by the way biological neural networks in our brain work. It´s designed to learn and recognize patterns, correlations and relationships within data. They consist of interconnected nodes, called 'neurons', organized into layers. Each neuron processes information and passes its output to other neurons, mimicking the information processing that occurs in the brain.

A network typically consists of three main types of layers:

- Input Layers
  - They receive raw data
- Hidden Layers
  - Computations take place
- Output Layer
  - Produces the final result

Neurons in one layer are connected to neurons in the next layer, each connection has a **weight** that adjusts during training to fine-tune the network´s behavior.

Cross Entropy

Cross Entropy is a statistical measure that quantifies the difference between two probability distributions. It is often used to compare the predicted probability distribution (output by a machine learning model) with the actual distribution (ground truth).

A higher cross-entropy implies a lower probability for an event.

Formular:

$CE(P, Q) = -\Sigma(P(x) * \log(Q(x)))$

Logistic Regression

- Take data
- Pick a random model
- Calculate the error
- Minimize the error, obtain a better model

Functions in Formulas

- Sigmoid activation function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Output (prediction) formula

$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + b)$$

- Error function

$$Error(y, \hat{y}) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

- The function that updates the weights

$$w_i \longrightarrow w_i + \alpha(y - \hat{y})x_i$$

$$b \longrightarrow b + \alpha(y - \hat{y})$$

Functions in Python

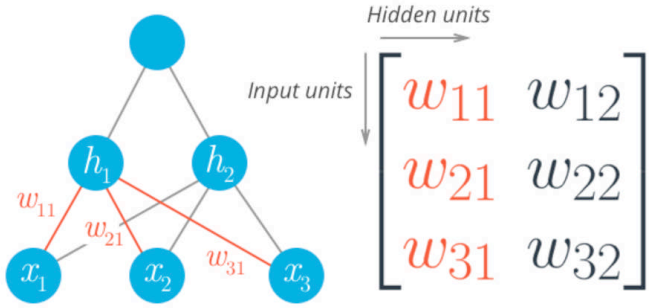
```
def sigmoid(x):
          return 1 / (1 + np.exp(-x))

def output_formula(features, weights, bias):
          return sigmoid(np.dot(features, weights) + bias)

def error_formula(y, output):
          return -y * np.log(output) - (1 - y) * np.log(1 - output)

def update_weights(x, y, weights, bias, learnrate):
          output = output_formula(x, weights, bias)
          d_error = -(y - output)
          weights -= learnrate * d_error * x
          bias -= learnrate * d_error
          return weights, bias
```

Backpropagation

- Doing a feedforward operation
- Comparing the output of the model with the desired output
- Calculating the error
- Running the feedforward operation backwards (backpropagation) to spread the error to each of the weights
- Use this to update the weights and get a better model
- Continue this until we have a model that is good

Matrix

In mathematics and linear algebra a matrix is a two-dimensional data structure consisting of a rectangular arrangement of numbers, symbols or expressions. Matrices have rows and columns and each entry in the matrix is called an **element** or a **coefficient.**



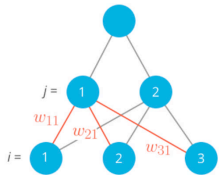
Weights matrix for 3 input units and 2 hidden units

Weights

Initalizing weights in NumPy

The input to a hidden unit is the sum of all the inputs multiplied by the hidden unit´s weights.

```
import numpy as np

# Define the size of the input, hidden, and output layers
input_size = 10
hidden_size = 20
output_size = 5

# Initialize weights and biases for the hidden layer and output layer
# You can choose from various initialization methods; here are a few examples:

# Method 1: Random Initialization
np.random.seed(0)  # For reproducibility
W1 = np.random.randn(input_size, hidden_size)  # Weight matrix for the hidden layer
b1 = np.zeros((1, hidden_size))  # Bias for the hidden layer
W2 = np.random.randn(hidden_size, output_size)  # Weight matrix for the output layer
b2 = np.zeros((1, output_size))
```



$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Calculating the input to the first hidden unit with the first column of the weights matrix.

$$h_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31}$$

Training: ---------- Testing:

```
from tqdm import tqdm
epochs = 50

train_losses, test_losses = [], []

for epoch in range(epochs):
    train_loss = 0
    test_loss = 0
    correct = 0
    total = 0
    accuracy = 0
    running_loss = 0.0

    model.train()


    for i, (images, labels) in tqdm(enumerate(training_dataloader, 0), unit='training'):
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()

        output = model(images)

        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        _, predicted = output.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    # Calculate and print accuracy after each epoch
    train_accuracy = 100 * correct / total
    print(f'Epoch [{epoch+1}/{epochs}], '
          f'Training Loss: {train_loss / len(training_dataloader):.3f}, '
          f'Training Accuracy: {train_accuracy:.2f}%')
```

```
    # Validation/testing
    model.eval()

    with torch.no_grad():

        for i, (images, labels) in tqdm(enumerate(testing_dataloader, 0), unit='testing'):
            images, labels = images.to(device), labels.to(device)
            log_ps = model(images)
            loss = criterion(log_ps, labels)
            test_loss += loss.item()

            ps = torch.exp(log_ps)
            top_probability, top_class = ps.topk(1, dim=1)
            equals = top_class == labels.view(*top_class.shape)
            accuracy += torch.mean(equals.type(torch.FloatTensor))
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

            print(accuracy)

    train_losses.append(train_loss/len(training_dataloader))
    test_losses.append(test_loss/len(testing_dataloader))

    model.train()

    print("Epoch: {}/{}.. ".format(epoch+1, epochs),
          "Training Loss: {:.3f}.. ".format(train_loss/len(training_dataloader)),
          "Test Loss: {:.3f}.. ".format(test_loss/len(testing_dataloader)),
          "Test Accuracy: {:.3f}".format(accuracy/len(testing_dataloader)))
```

Layers

Input Layer - Eyes

It´s the starting point of the machine learning system. It´s like the '**eyes'** of the model. This layer takes in the initial information about what you want the model to work with.
Let´s say we are trying to teach a model to recognize animals from pictures. Each pixel in the picture could be an input to the input layer. So if we are showing the model a picture of a cat, the input layer is what receives all the pixel values from that image.

Hidden Layer - Brain

The hidden layer is where the magic happens in a NN. It´s like the '**brain'** of the model. This layer processes the information from the input layer and tries to figure out patterns and relationships in the data. Imagine we are teaching the model to recognize whether a person is happy or sad from a picture of their face. The hidden layer would analyze the various features of the face, like the position of the mouth and the shape of the eyes and use that information to make a guess about the person´s emotion.

Output Layer - Mouth

The output layer is where the model gives you its final answers. It´s like the '**mouth'** of the model. This layer takes the processed information from the hidden layer and presents the model´s prediction or decision. If we are going back to the example of recognizing animal pictures, the output layer might tell you whether the model thinks the picture contains a cat, a dog, or something else

Simple Terms:

- Input layer: Receives initial data
- Hidden layer: Processes and understands the data
- Output layer: Gives you the final result or prediction

Weights

Think of it as the 'importance knobs' that the model uses to adjust how much it pays attention to different things in the input data. Imagine we are teaching a model to predict the price of a house based on its size and number of rooms. The weights are like the factors that the model learns to multiply with the size and number of rooms to get a reasonable prediction. If the model figures out that size is really important in determining the price, it will give a larger weight to the size feature.

Biases

Biases are like the 'base values' that the model adds to the weighted input before making a prediction. They allow the model to make predictions even when all the input values are zero. Sticking with the house price prediction example, let´s say the model knows that even if a house has zero size and zero rooms, it should still have some base price (maybe due to other factors like location). The bias helps the model account for this base value.

Simple Terms:

- Weights: Adjust how much the model cares about different aspects of the input data
- Biases: Add a base value to the model´s prediction, even when input values are zero.

Cross Entropy

It´s like a yardstick that measures how well your guesses match reality. Imagine we are playing a guessing game where you try to predict things like whether it will rain tomorrow or not. Cross entropy is a way to see how close your guesses (your predicted probabilites) are to what really happens (the actual probabilities). If the guesses match perfectly, cross entropy is low, if the guesses are far off from reality, cross entropy is higher.

Forward Pass

It´s like following a recipe. You start with the raw ingredients (input data), mix them together using a set of rules (weights and biases) and end up with a delicious dish (output prediction).
It´s the process of taking input data, processing it through the network layers with weights and biases and getting an output prediction.

1. **Input Data:**
    a. Imagine we are **baking a cake**. We gather all the ingredients like flour, eggs and sugar. These ingredients are like the data you want to process.

2. **Weights and Biases:**
    a. They are the **recipe to mix** the ingredients. The weights and biases are like the secret proportions we use in our recipe. They tell the model how much attention to give to each ingredient.

3. **Mixing:**
    a. We **follow** the recipe step by step, combining the ingredients and applying the proportions. This mixing process is like the calculations that happen in the neural network layers.

4. **Output Prediction:**
    a. When we are done, we have a cake. **This cake is our output prediction**. It´s what the model thinks the result should be based on the input data and the 'recipe'

Calculating Bias and Weights in Python

```
import numpy as np

# Define the input size and the number of neurons in the layer
input_size = 5
output_size = 3

# Initialize weights and biases
# Random initialization is common, but you can choose other methods as well
weights = np.random.randn(input_size, output_size)
biases = np.zeros((1, output_size))

print("Weights:")
print(weights)
print("\nBiases:")
print(biases)
```

Calculating in Python

Calculating the input, output and hidden layer in Python

```
import numpy as np

# Define input features (example data with 4 features)
input_features = np.array([0.2, 0.5, 0.7, 0.9])

# Define the architecture of the neural network
input_size = 4
hidden_size = 3
output_size = 2

# Initialize weights and biases
# Weights are initialized with small random values and biases with zeros
input_to_hidden_weights = np.random.randn(input_size, hidden_size)
hidden_layer_biases = np.zeros((1, hidden_size))
hidden_to_output_weights = np.random.randn(hidden_size, output_size)
output_layer_biases = np.zeros((1, output_size))

# Perform the forward pass
hidden_layer_input = np.dot(input_features, input_to_hidden_weights) + hidden_layer_biases
hidden_layer_output = 1 / (1 + np.exp(-hidden_layer_input))  # Sigmoid activation for the hidden layer
output_layer_input = np.dot(hidden_layer_output, hidden_to_output_weights) + output_layer_biases
output_layer_output = 1 / (1 + np.exp(-output_layer_input))  # Sigmoid activation for the output layer
```

Calculating Cross Entropy

```
# True labels (ground truth)
y_true = np.array([1, 0, 1, 1, 0])

# Predicted probabilities (predicted by your model)
y_pred_prob = np.array([0.9, 0.2, 0.8, 0.75, 0.3])

# Calculate binary cross-entropy loss
epsilon = 1e-15  # Small constant to prevent log(0)
y_pred_prob = np.clip(y_pred_prob, epsilon, 1 - epsilon)  # Clip values to avoid taking the log of 0 or 1
cross_entropy_loss = -np.mean(y_true * np.log(y_pred_prob) + (1 - y_true) * np.log(1 - y_pred_prob))
```

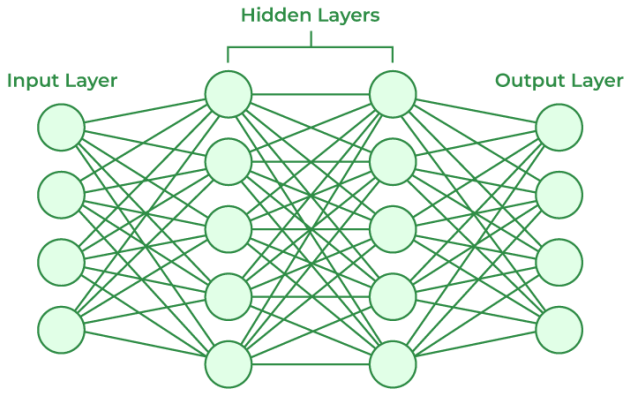
How to train the network?

1. Make a forward pass through the network
2. Use the network output to calculate the loss
3. Perform a backward pass through the network with loss.backward() to calculate the gradients
4. Take a step with the optimizer to update the weights

Neural Network Example

```
class CIFARNeuralNetwork(nn.Module):
  def __init__(self):
    super().__init__()
    ## Conv
    self.conv1 = nn.Conv2d(3, 128, 5, padding=2)
    self.conv2 = nn.Conv2d(128, 128, 5, padding=2)
    self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
    self.conv4 = nn.Conv2d(256, 256, 3, padding=1)
    # Pool
    self.pool = nn.MaxPool2d(2, 2)
    # Dropout
    self.dropout_conv = nn.Dropout2d(p=0.25)
    self.dropout = nn.Dropout(0.5)
    # Batch Norm 2d
    self.bn_conv1 = nn.BatchNorm2d(128)
    self.bn_conv2 = nn.BatchNorm2d(128)
    self.bn_conv3 = nn.BatchNorm2d(256)
    self.bn_conv4 = nn.BatchNorm2d(256)
    #Batch Norm 1d
    self.bn_dense1 = nn.BatchNorm1d(1024)
    self.bn_dense2 = nn.BatchNorm1d(512)
    # Linear
    self.fc1 = nn.Linear(256 * 8 * 8, 1024)
    self.fc2 = nn.Linear(1024, 512)
    self.fc3 = nn.Linear(512, 10) # 10 Classes

  def conv_layers(self, x):
    x = F.relu(self.bn_conv1(self.conv1(x)))
    x = F.relu(self.bn_conv2(self.conv2(x)))
    x = self.pool(x)
    x = self.dropout_conv(x)
    x = F.relu(self.bn_conv3(self.conv3(x)))
    x = F.relu(self.bn_conv4(self.conv4(x)))
    x = self.pool(x)
    x = self.dropout_conv(x)
    return x


  def forward(self, x):
    x = self.conv_layers(x)
    x = x.view(-1, 256 * 8 * 8)
    x = F.relu(self.bn_dense1(self.fc1(x)))
    x = self.dropout(x)
    x = F.relu(self.bn_dense2(self.fc2(x)))
    x = self.dropout(x)
    x = self.fc3(x)

    return x
```

Simple Image of a Neural Network



Unsupervised Learning

Unsupervised Learning is a category of machine learning where the algorithm learns patterns and structures in data without explicit supervision or labeled target values. Instead, it focuses on finding hidden patterns, grouping similar data points, or reducing the dimensionality of data.
Some real world examples for unsupervised learning and when to choose it are **Recommendation Systems**, **Customer Segmentation or Autnomous Vehicles.**

- **Supervised Learning:**
  - Starting with a label or a value that you are trying to predict
- **Unsupervised Learning:**
  - Starting with grouping data together that does not have labels

Clustering:

➔ Groups data together based on similarities

Dimensionality Reduction:

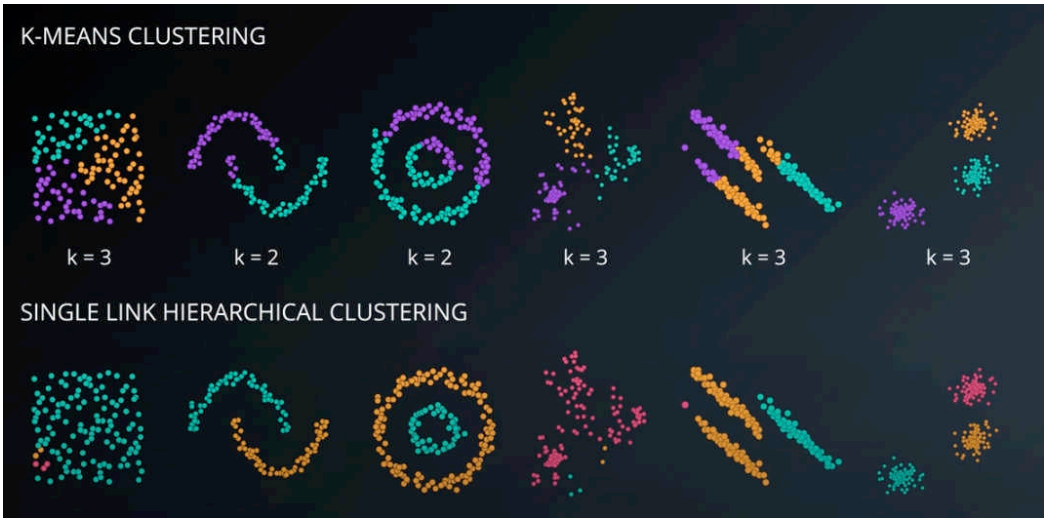
➔ Condenses a large number of features into a (usually much) smaller set of features

K-Means Algo:

1. Books of similar genres or written by the same authors.
2. Similar movies
3. Similar music
4. Similar groups of customers.

```
from sklearn.cluster import KMeans

# Generate some example data
from sklearn.datasets import make_blobs
X, _ = make_blobs(n_samples=300, centers=4, random_state=42)

# Instantiate K-Means with the number of clusters
kmeans = KMeans(n_clusters=4)

# Fit the model to the data
kmeans.fit(X)

# Get cluster labels and cluster centers
labels = kmeans.labels_
centers = kmeans.cluster_centers_
```

'k' represents the number of clusters you have in your dataset.



Hierarchical Clustering

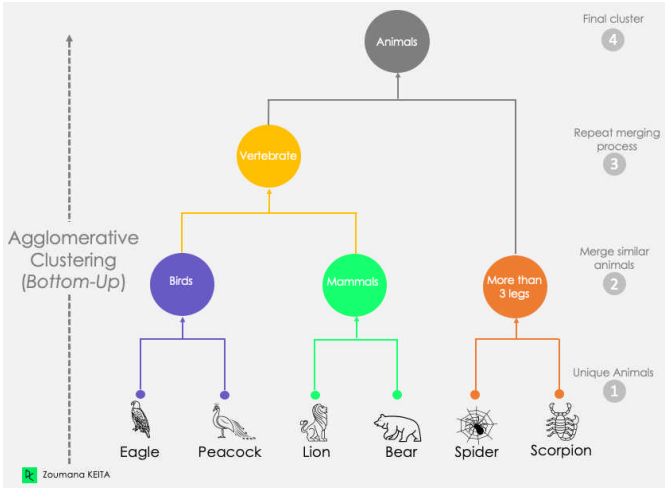
It is used to group similar data points into clusters or groups. It creates a hierarchical representation of data by successively merging or splitting clusters based on their similarity.

Agglomerative (Bottom-Up) Hierarchical Clustering:

In agglomerative hierarchical clustering, each data point initially represents a separate cluster, and the algorithm proceeds by iteratively merging clusters based on their similarity. It is often used when **the number of clusters is unknown.** Agglomerative clustering is usually the right choice to make if you have large datasets, because it tends to be **computationally less expensive** than Divisive Clustering.

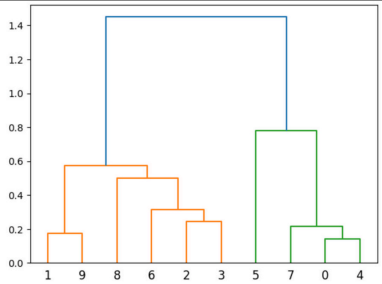
Divisive (Top-Down) Hierarchical Clustering:

It takes the opposite approach by starting with all data points in a single cluster and then successively splitting the clusters into smaller ones based on dissimilarity. Divisive Clustering can be more effective when you want to perform a detailed analysis of the data structure by successively splitting clusters into smaller ones.



https://images.datacamp.com/image/upload/v1674149819/Dendrogram_of_Agglomerative_Clustering_Approach_4eba3586ec.png

```
from sklearn import datasets, cluster
from scipy.cluster.hierarchy import dendrogram, ward, single
import matplotlib.pyplot as plt

X = datasets.load_iris().data[:10]

# Set Clusters
clust = cluster.AgglomerativeClustering(n_clusters=3, linkage='ward')
labels = clust.fit_predict(X)

# Plotting Dendrogram
linkage_matrix = ward(X)

dendrogram(linkage_matrix)
plt.show()
```



DBSCAN

DBSCAN (Density-based spatial clustering of application with noise) is a clustering method that groups points closely packed together. It is expecting the inputs Epsilon (the distance that defines each point´s neighborhood) and MinPts/min_samples (the number of points in a neighborhood for a point to be regarded as a core point). Usually good reasons to use DBSCAN are clustering a dataset based on point density or identifying noise in a dataset.

Steps for DBSCAN:

1. Select a point arbitrarily

2. Define a search area around the selected point using a predefined search distance

3. Search if there is any point in the search area
   a. If there is **NOT** a single point in the area, the selected point is labeled as **noise**
   b. If there are points in the area, compare the number of points in **the search area (including the selected point) to a predefined minimum number of points parameter MinPts.**

4. Procced to each point in the cluster found above and search for point in its search area
   a. If the number of points is **greater** than MinPts, the point is also a **core point** in the **same** cluster.
   b. If the number of points is **smaller** than MinPts, the point is a **border point** in the **same** cluster.

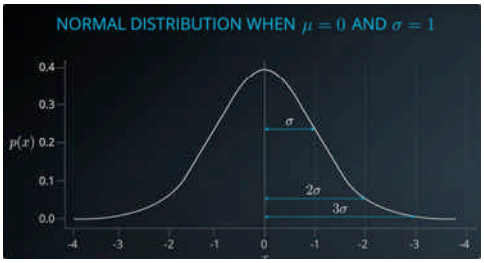
```
# Import Libraries
from sklearn import datasets, cluster

# Load Data
X = datasets.load_iris().data

# Define and Fit DBSCAN
db = cluster.DBSCAN(eps=0.5, min_samples=5)
db.fit(X)

OUTPUT:
```

```
db.labels_
```

```
array([ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0, -1,  0,  0,  0,  0,  0,  0,  0,  0,  1,
        1,  1,  1,  1,  1,  1, -1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,
       -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
        1,  1, -1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1, -1,  1,  1,  1,
        1,  1,  1, -1, -1,  1, -1, -1,  1,  1,  1,  1,  1,  1,  1, -1, -1,
        1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1, -1, -1,
        1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

Gaussian Mixture Model (GMM) Clustering

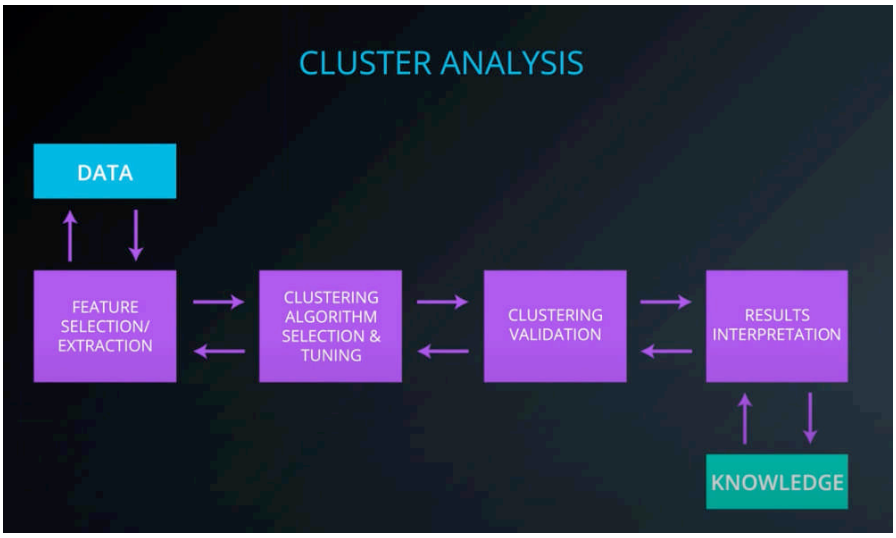
It assumes that each cluster follows a particular statistical distribution. Its clustering methodology relies heavily on probability and statistics. The **Gaussian distribution** is also known as a normal distribution.



Advantages	Disadvantages
☐ Soft-clustering (sample membership of multiple clusters) ☐ Cluster shape flexibility	☐ Sensitive to initialization values ☐ Possible to converge to a local optimum ☐ Slow convergence rate

```
# Import Libraries
from sklearn import datasets, mixture

# Load Dataset
X = datasets.load_iris().data

# Set Gaussian Mixture
gmm = mixture.GaussianMixture(n_components=3)
gmm.fit(X)
clustering = gmm.predict(X)

OUTPUT:
clustering
```

```
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 2, 0, 2, 0, 2, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```



1. Feature selection
    a. Choose best features in the dataset needed for analysis
    b. Transform data to generate additional new and useful features, using principal component analysis (PCA)
2. Choose a clustering algorithm
    a. Make the selection using best results based on your use case
3. Clustering validation
    a. Evaluating how well a clustering performed based on visualizations and using a clustering validation index
4. Results and Interpretation
    a. What can be understood from the clustering structure, using background knowledge and domain expertise

Cluster Validation

Three categories of cluster validation are:

- External indices - used to score clustering model performance when a dataset is labeled
- Internal indices - used to measure the fit between data and structure using only the data, when the dataset is unlabeled
- Relative Indices - used to indicated which of two clustering structures is 'better'

Dimensionality Reduction and PCA

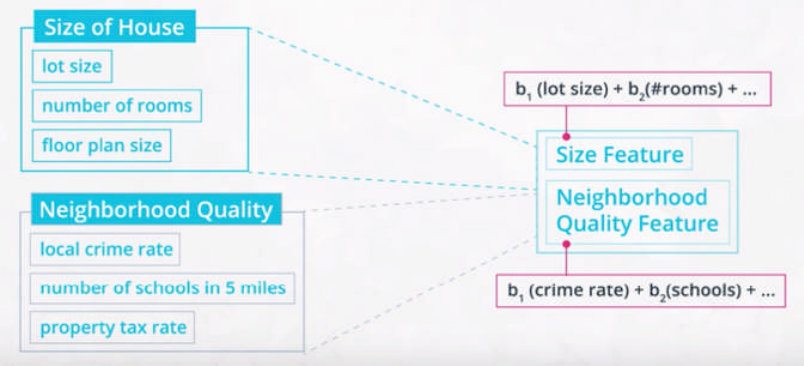
Feature Selection

It involves finding a subset of the original features of the data that you determine are most relevant and useful.

- Filter methods – Filtering approaches use a ranking or sorting algorithm to filter out those features that have less usefulness. Filter methods are based on discerning some inherent correlations among the feature data in unsupervised learning or on correlations with the output variable in supervised settings. Common tools are **Pearon´s Correlation, Linear Discriminant Analysis (LDA)** and **Analysis of Variance (ANOVA**)

Feature Extraction

Feature Extraction involves extracting or constructing new features called **latent features.**
Some other words for **principal components** are linear combination of the original features in a dataset, latent variable or a new feature that can be used in a future analysis



PCA in Python

```
import numpy as np
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Create some sample data
np.random.seed(0)
n_samples = 100
n_features = 2
X = np.random.randn(n_samples, n_features)

# Instantiate PCA with the number of components you want to retain
pca = PCA(n_components=1)

# Fit PCA on your data
pca.fit(X)

# Transform the data into the reduced feature space
X_reduced = pca.transform(X)

# Transform the reduced data back to the original space (for illustration)
X_original = pca.inverse_transform(X_reduced)

# Plot the original data and the reduced data
plt.figure(figsize=(8, 4))
plt.subplot(1, 2, 1)
plt.scatter(X[:, 0], X[:, 1], alpha=0.6)
plt.title("Original Data")

plt.subplot(1, 2, 2)
plt.scatter(X_reduced, np.zeros_like(X_reduced), alpha=0.6)
plt.title("Reduced Data")
plt.tight_layout()
plt.show()
```

Definitions and Theory

Cross Entropy Loss (Loss Function)

Cross Entropy Loss is like a scorecard for how well your machine learning model is performing in classification tasks. In classifications, you want the model to predict which category an input belongs (classifying emails as spam or not). Cross Entropy Loss measures how different the model´s predictions are from the actual labels. If the prediction is perfect, the loss is low (close to 0), otherwise if it´s way off, the loss will be high. The goal is to minimize this loss during training, making the model better at making accurate predictions.
It should be used for classification tasks, where you have to predict categories or classes. You might ask why, because the Cross Entropy Loss is well-suited for classification because it measures the difference between predicted probabilities and actual labels. It´s particularly effective when you have multiple classes to predict. One real world example would be Sentiment Analysis in Natural Language Processing (NLP) where you classify text as positive, negative or neutral.

Optimizer

An optimizer is like a GPS for teaching the model to find the best parameters for making predictions.
During training, the model´s goal is to find the best set of parameters (weights and biases) to minimize the loss function. Optimizers help adjust these parameters step by step to reach that goal. Think of it like fine-tuning a musical instrument -> the optimizer guides the process of getting the best tune out of the model. It is used in every machine learning task that involves parameter tuning. It is crucial for finding the best set of model parameters that minimize the loss function. Without them, training would be slow or inefficient + models wouldn´t improve over time.

Criterion

Criterion is like a teacher´s feedback, telling the model how well it´s doing and where it needs improvement.
Criterion and loss functions are often used interchangeably. It´s the mathematical formula that calculates the loss based on model predictions and actual labels. Different tasks (classification, regression) require different loss functions. For example, the Mean Squared Error is a common loss function for regression tasks, while Cross Entropy Loss is often used for classification. One real world example would be predicting house prices.

Scheduler (Learning Rate Scheduler):

A scheduler is like adjusting the volume of the training process, making it louder (faster learning) or quieter (slower learning) as needed.
The learning rate is a crucial hyperparameter that controls the size of steps the optimizer takes during training. If it´s too big it might overshoot the best solution and if it´s too small it will take forever. A scheduler helps adjust the learning rate during training. For example, we might start with a high learning rate to make quick progress and then reduce it gradually to fine-tune and converge to the best model. Schedulers are handy for avoiding overshooting and getting stuck in local minima. They are primarily used in deep learning for controlling the learning rate.

Features Definitions

Features are also known as attributes, variables or columns in a dataset. They are the specific data elements or attributes that provide information about each data point in a dataset. They are building blocks for creating models and making predictions.

- Numerical Features
  - These are features with numeric values, for example age, price, temperature or income
- Categorical Features
  - These are features that represent categories or labels, for example color, gender or country
- Text Features
  - In natural language processing, text data is often converted into features using techniques like TF-IDF or word embeddings
- Image Features
  - In computer vision, image data is represented by pixel values and extracted features like edge, textures, or shape
- Time Series Features
  - In time series analysis time-related data points are treated as features

Feature Scaling:

Feature scaling is like making sure all ingredients in a recipe are in the same units (e.g. all measurements in tablespoons) to create a blanaced and tasty dish. In machine learning, it ensures all input features have the same units. Imagine the task is to create a model to predict house prices, and the features include the number of bedrooms and the total square footage of the house. The number of bedrooms might range form 1 to 5, while the square footage could range from 800 to 4000. These two features are on very different scales. If we use an algorithm that calculates distances between data points, like k-nearest neighbors or gradient descent, the algorithm will be heavily influenced by the feature with the larger scale (square footage).
Feature scaling techniques standardize these features, typically to have a mean of 0 and a standard deviation of 1 (known as **z-score scaling**), or to a range between 0 and 1 (known as **min-max scaling).** This ensures that all features contribute equally to the model´s learning process and makes the model less sensitive to the scale of the data.

Ending

Congratulations on completing 'Intro to Machine Learning with PyTorch'! We hope this journey through the fascinating world of machine learning has equipped you with valuable insights and practical skills. By grasping the fundamentals of PyTorch and delving into the essentials of machine learning, you've taken a significant step toward mastering this dynamic field.

As you conclude this guide, remember that learning is a continuous process. We encourage you to apply the knowledge you've gained to real-world projects and explore advanced concepts to further enhance your understanding.

We hope you've enjoyed the hands-on examples, practical exercises, and the journey through constructing and training neural networks. The skills you've acquired are invaluable, and we trust they will serve as a solid foundation for your future endeavors in the realms of data science, predictive modeling, and artificial intelligence.

Keep exploring, experimenting, and most importantly, enjoy the process. Your curiosity and dedication to learning are the keys to unlocking new opportunities and innovations in the realm of machine learning.

Thank you for joining us on this learning adventure. Your enthusiasm and commitment are truly appreciated. We wish you the very best in all your future machine learning endeavors. Keep innovating, keep learning, and keep pushing the boundaries of what's possible in the world of PyTorch and machine learning!