WELCOME

Hello, and welcome to Machine Learning with Python.
In this course, you'll learn how Machine Learning is used in many key fields and industries.
For example, in the health care industry, data scientists use Machine Learning to predict whether a human cell that is believed to be at risk of developing cancer, is either benign or malignant.
As such, Machine learning can play a key role in determining a person's health and welfare.
You'll also learn about the value of decision trees and how building a good decision tree from historical data helps doctors to prescribe the proper medicine for each of their patients.
You'll learn how bankers use machine learning to make decisions on whether to approve loan applications.
And you will learn how to use machine learning to do bank customer segmentation, where it is not usually easy to run for huge volumes of varied data.
In this course, you'll see how machine learning helps websites such as YouTube, Amazon, or Netflix develop recommendations to their customers about various products or services, such as
which movies they might be interested in going to see or which books to buy.
There is so much that you can do with Machine Learning!
Here, you'll learn how to use popular python libraries to build your model.
For example, given an automobile dataset, we use the sci-kit learn (sklearn) library to estimate the Co2 emission of cars using their Engine size or Cylinders.
We can even predict what the Co2 emissions will be for a car that hasn't even been produced yet!
And we'll see how the telecommunications industry can predict customer churn.
You can run and practice the code of all these samples using the built-in lab environment in this course.
You don't have to install anything to your computer or do anything on the cloud.
All you have to do is click a button to start the lab environment in your browser.
The code for the samples is already written using python language, in Jupyter notebooks, and you can run it to see the results, or change it to understand the algorithms better.
So, what will you be able to achieve by taking this course?
Well, by putting in just a few hours a week over the next few weeks, you'll get new skills to add to your resume, such as regression, classification, clustering, sci-kit learn and SciPy.
You'll also get new projects that you can add to your portfolio, including cancer detection, predicting economic trends, predicting customer churn, recommendation engines, and many more.
You'll also get a certificate in machine learning to prove your competency, and share it anywhere you like online or offline, such as LinkedIn profiles and social media.
So let's get started.

MODULE 1 – MACHINE LEARING

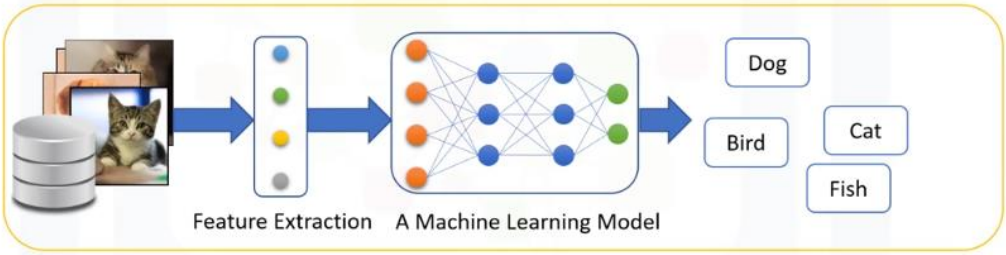
INTRO TO MACHINE LEARNING

What is machine learning?

Machine learning is the subfield of computer science that gives "**computers the ability to learn without being explicitly programmed.**"

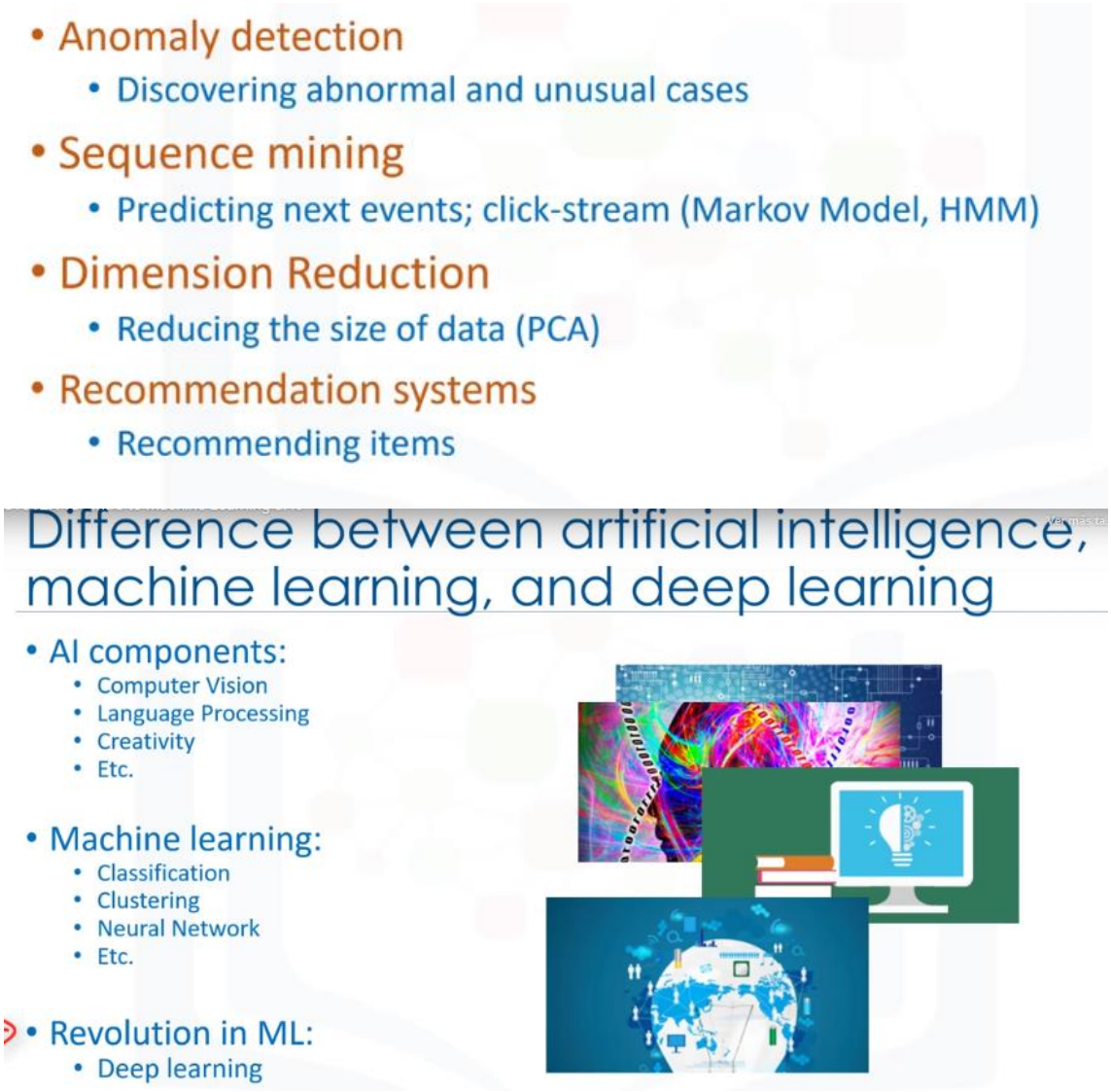
Arthur Samuel
American pioneer in the field of computer gaming and artificial intelligence, coined the term "machine learning" in 1959 while at IBM.

How machine learning works?



Major machine learning techniques

- Regression/Estimation
  - Predicting continuous values
- Classification
  - Predicting the item class/category of a case
- Clustering
  - Finding the structure of data; summarization
- Associations
  - Associating frequent co-occurring items/events

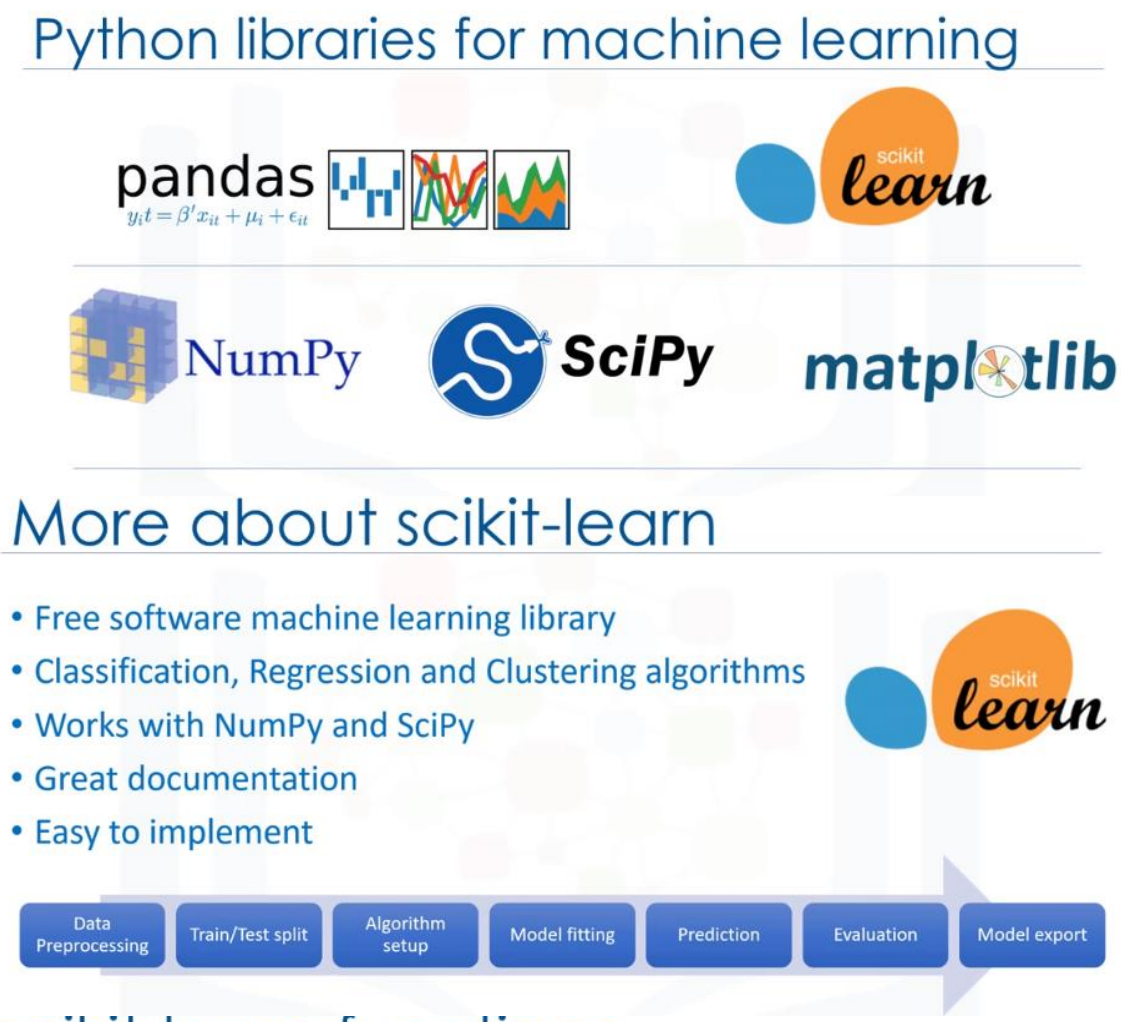


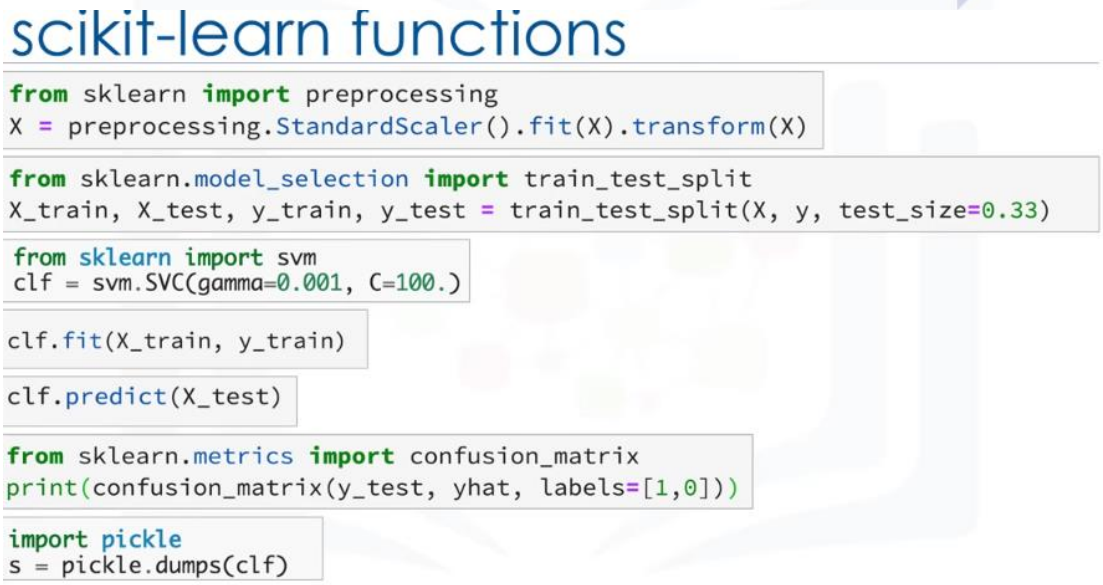
Hello, and welcome!
In this video I will give you a high level introduction to Machine Learning.
So let's get started.
This is a human cell sample extracted from a patient.
And this cell has characteristics … for example, its Clump thickness is 6, its Uniformity of cell size is 1, its Marginal adhesion is 1, and so on.
One of the interesting questions we can ask, at this point is: "Is this a Benign or Malignant cell?"
In contrast with a benign tumor, a malignant tumor is a tumor that may invade its surrounding tissue or spread around the body, and diagnosing it early might be the key to a patient's survival.
One could easily presume that only a doctor with years of experience could diagnose that tumor and say if the patient is developing cancer or not.
Right?
Well, imagine that you've obtained a dataset containing characteristics of thousands of human cell samples extracted from patients who were believed to be at risk of developing cancer.
Analysis of the original data showed that many of the characteristics differed significantly between benign and malignant samples.
You can use the values of these cell characteristics in samples from other patients to give an

early indication of whether a new sample might be benign or malignant.
You should clean your data, select a proper algorithm for building a prediction model, and train your model to understand patterns of benign or malignant cells within the data.
Once the model has been trained by going through data iteratively, it can be used to predict your new or unknown cell with a rather high accuracy.
This is machine learning!
It is the way that a machine learning model can do a doctor's task or at least help that doctor make the process faster.
Now, let me give a formal definition of machine learning.
Machine learning is the subfield of computer science that gives "computers the ability to learn without being explicitly programmed."
Let me explain what I mean when I say "without being explicitly programmed."
Assume that you have a dataset of images of animals such as cats and dogs, and you want to have software or an application that can recognize and differentiate them.
The first thing that you have to do here is interpret the images as a set of feature sets.
For example, does the image show the animal's eyes?
If so, what is their size?
Does it have ears?
What about a tail?
How many legs?
Does it have wings?
Prior to machine learning, each image would be transformed to a vector of features.
Then, traditionally, we had to write down some rules or methods in order to get computers to be intelligent and detect the animals.
But, it was a failure.
Why?
Well, as you can guess, it needed a lot of rules, highly dependent on the current dataset, and not generalized enough to detect out-of-sample cases.
This is when machine learning entered the scene.
Using machine learning allows us to build a model that looks at all the feature sets, and their corresponding type of animals, and learn it learns the pattern of each animal.
It is a model built by machine learning algorithms.
It detects without explicitly being programmed to do so.
In essence, machine learning follows the same process that a 4-year-old child uses to learn, understand, and differentiate animals.
So, machine learning algorithms, inspired by the human learning process, iteratively learn from data, and allow computers to find hidden insights.
These models help us in a variety of tasks, such as object recognition, summarization, recommendation, and so on.
Machine Learning impacts society in a very influential way.
Here are some real-life examples.
First, how do you think Netflix and Amazon recommend videos, movies, and TV shows to its users?
They use Machine Learning to produce suggestions that you might enjoy!
This is similar to how your friends might recommend a television show to you, based on their knowledge of the types of shows you like to watch.
How do you think banks make a decision when approving a loan application?
They use machine learning to predict the probability of default for each applicant, and then approve
or refuse the loan application based on that probability.
Telecommunication companies use their customers' demographic data to segment them, or predict

if they will unsubscribe from their company the next month.
There are many other applications of machine learning that we see every day in our daily life, such as chatbots, logging into our phones or even computer games using face recognition. Each of these use different machine learning techniques and algorithms.
So, let's quickly examine a few of the more popular techniques.
The Regression/Estimation technique is used for predicting a continuous value, for example, predicting things like the price of a house based on its characteristics, or to estimate the Co2 emission from a car's engine.
A Classification technique is used for Predicting the class or category of a case, for example, if a cell is benign or malignant, or whether or not a customer will churn.
Clustering groups of similar cases, for example, can find similar patients, or can be used for customer segmentation in the banking field.
Association technique is used for finding items or events that often co-occur, for example, grocery items that are usually bought together by a particular customer.
Anomaly detection is used to discover abnormal and unusual cases, for example, it is used for credit card fraud detection.
Sequence mining is used for predicting the next event, for instance, the click-stream in websites.
Dimension reduction is used to reduce the size of data.
And finally, recommendation systems; this associates people's preferences with others who have similar tastes, and recommends new items to them, such as books or movies.
We will cover some of these techniques in the next videos.
By this point, I'm quite sure this question has crossed your mind, "What is the difference between these buzzwords that we keep hearing these days, such as Artificial intelligence (or AI), Machine Learning and Deep Learning?"
Well, let me explain what is different between them.
In brief, AI tries to make computers intelligent in order to mimic the cognitive functions of humans.
So, Artificial Intelligence is a general field with a broad scope including: Computer Vision, Language Processing, Creativity, and Summarization.
Machine Learning is the branch of AI that covers the statistical part of artificial intelligence.
It teaches the computer to solve problems by looking at hundreds or thousands of examples, learning from them, and then using that experience to solve the same problem in new situations.
And Deep Learning is a very special field of Machine Learning where computers can actually learn and make intelligent decisions on their own.
Deep learning involves a deeper level of automation in comparison with most machine learning algorithms.
Now that we've completed the introduction to Machine Learning, subsequent videos will focus on reviewing two main components: First, you'll be learning about the purpose of Machine Learning and where it can be applied in the real world; and
Second, you'll get a general overview of Machine Learning topics, such as supervised vs unsupervised learning, model evaluation and various Machine Learning algorithms.
So now that you have a sense with what's in store on this journey, let's continue our exploration of Machine Learning!
Thanks for watching!

PYTHON TO MACHINE LEARNING



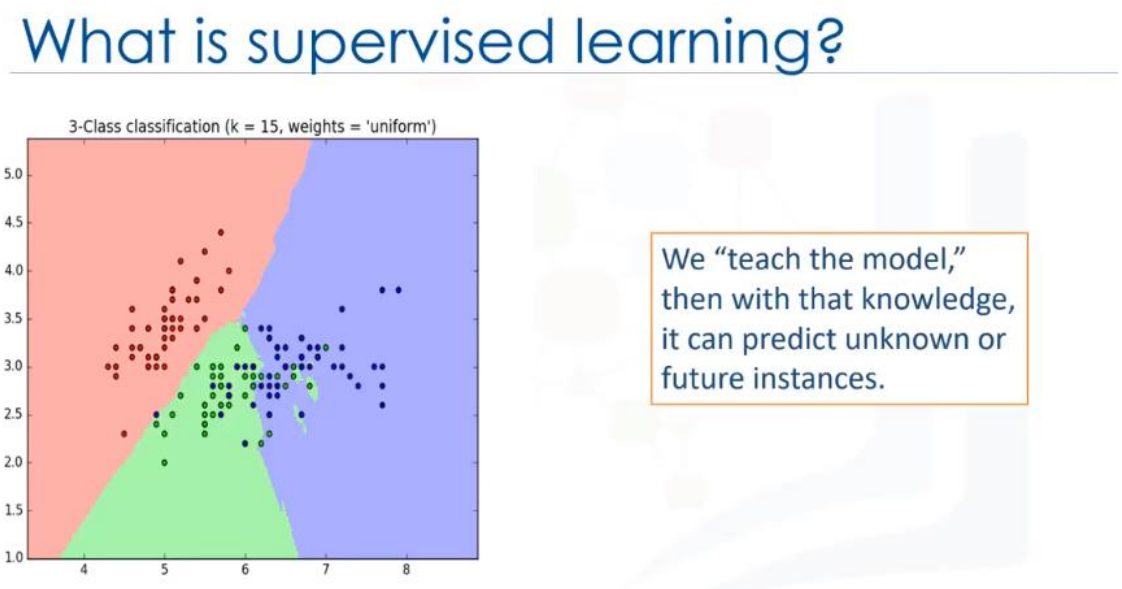


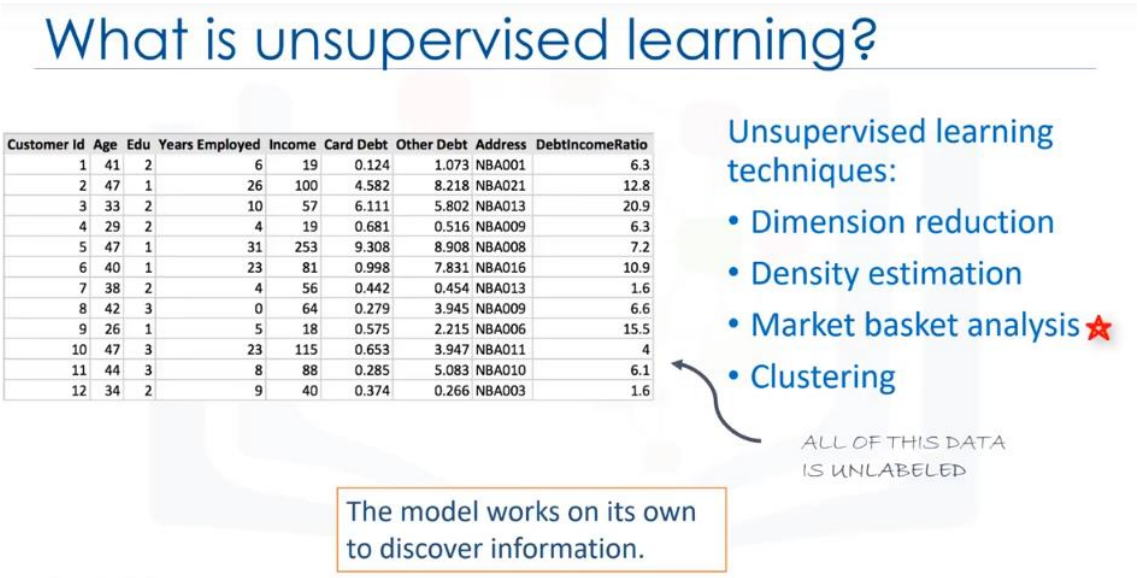
Hello, and welcome!
In this video, we'll talk about how to use python for machine learning.
So let's get started.
Python is a popular and powerful general-purpose programming language that recently emerged

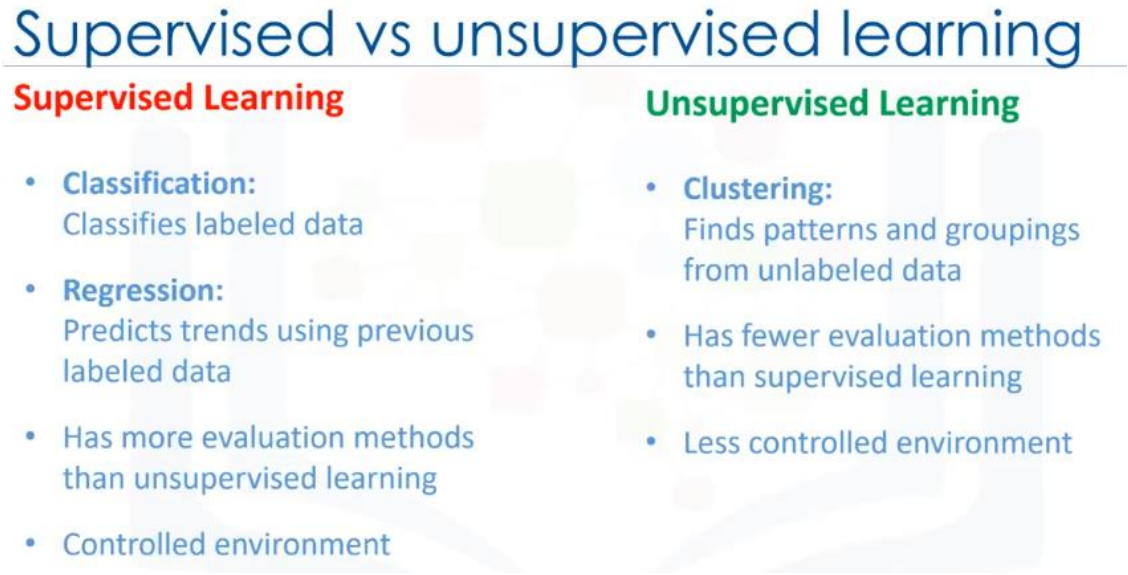
as the preferred language among data scientists.
You can write your machine learning algorithm using python, and it works very well.
However, there are a lot of modules and libraries already implemented in python that can make your life much easier.
We try to introduce the Python packages in this course and use it in the labs to give you better hands-on experience.
The first package is Numpy, which is a math library to work with n-dimensional arrays in Python.
It enables you to do computation efficiently and effectively.
It is better than regular python because of it's amazing capabilities.
For example, for working with arrays, dictionaries, functions, datatypes, and working with images,
you need to know Numpy.
SciPy is a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics and much more.
SciPy is a good library for scientific and high-performance computation.
Matplotlib is a very popular plotting package that provides 2D plotting as well as 3D plotting.
Basic Knowledge about these 3 packages, which are built on top of python, is a good asset for data scientists who want to work with real world problems.
If you are not familiar with these packages, I recommend that you take the "Data Analysis with Python" course first.
This course covers most of the useful topics in these packages.
Pandas library, is a very high-level python library that provides high-performance, easy to use data structures.
It has many functions for data importing, manipulation and analysis.
In particular, it offers data structures and operations for manipulating numerical tables and time series.
scikit-learn is a collection of algorithms and tools for machine learning, which is our focus here, and which you'll learn to use with in this course.
As we'll be using scikit-learn quite a bit, in the labs, let me explain more about it and show you why it is so popular among data scientists.
Scikit-learn is a free machine learning library for the Python programming language.
It has most of the classification, regression and clustering algorithms, and it's designed to work with the Python numerical and scientific libraries, NumPy and SciPy.
Also, it includes very good documentation.
On top of that, implementing machine learning models with scikit learn is really easy, with a few lines of python code.
Most of the tasks that need to be done in a machine learning pipeline are implemented already in scikit learn, including, pre-processing of data, feature selection, feature extraction, train/test splitting, defining the algorithms, fitting models, tuning parameters, prediction, evaluation, and exporting the model.
Let me show you an example of how scikit learn looks like when you use this library.
You don't have to understand the code for now, but just see how easily you can build a model with just a few lines of code.
Basically, Machine learning algorithms benefit from standardization of the data set.
If there are some outliers, or different scales fields in your data set, you have to fix them.
The preprocessing package of scikit learn provides several common utility functions and transformer classes to change raw feature vectors into a suitable form of vector for modeling.
You have to split your dataset into train and test sets to train your model, and then test the model's accuracy separately.
Scikit learn can split arrays or matrices into random train and test subsets for you,

in one line of code.
Then, you can setup your algorithm.
For example, you can build a classifier using a support vector classification algorithm.
We call our estimator instance clf, and initialize its parameters.
Now, you can train your model with the train set.
By passing our training set to the fit method, the clf model learns to classify unknown cases.
Then, we can use our test set to run predictions.
And, the result tells us what the class of each unknown value is.
Also, you can use different metrics to evaluate your model accuracy, for example, using a confusion matrix to show the results.
And finally, you save your model.
You may find all or some of these machine learning terms confusing, but don't worry, we will talk about all of these topics in the following videos.
The most important point to remember is that the entire process of a Machine Learning task can be done simply in a few lines of code, using scikit learn.
Please notice that, though it is possible, it would not be that easy if you want to do all of this using Numpy or Scipy packages.
And of course, it needs much more coding if you use pure python programming to implement all of these tasks.
Thanks for watching.

SUPERVISED VS UNSUPERVISED







Hello, and welcome!
In this video, we'll introduce supervised algorithms versus unsupervised algorithms.
So let's get started.
An easy way to begin grasping the concept of supervised learning is by looking directly at the words that make it up.
Supervise means to observe and direct the execution of a task, project, or activity.
Obviously, we aren't going to be supervising a person…
Instead, we'll be supervising a machine learning model that might be able to produce classification regions like we see here.
So, how do we supervise a machine learning model?
We do this by "teaching" the model.
That is, we load the model with knowledge so that we can have it predict future instances.
But … this leads to the next question, which is, "How exactly do we teach a model?"
We teach the model by training it with some data from a labeled dataset.
It's important to note that the data is labeled.
And what does a labeled dataset look like?
Well, it can look something like this.
This example is taken from the cancer dataset.

As you can see, we have some historical data for patients, and we already know the class of each row.
Let's start by introducing some components of this table.
The names up here, which are called Clump thickness, Uniformity of cell size, Uniformity of cell shape, Marginal adhesion, and so on, are called Attributes.
The columns are called Features, which include the data.
If you plot this data, and look at a single data point on a plot, it'll have all of these attributes.
That would make a row on this chart, also referred to as an observation.
Looking directly at the value of the data, you can have two kinds.
The first is numerical.
When dealing with machine learning, the most commonly used data is numeric.
The second is categorical… that is, it's non-numeric, because it contains characters rather than numbers.
In this case, it's categorical because this dataset is made for Classification.
There are two types of Supervised Learning techniques.
They are: classification and regression.
Classification is the process of predicting a discrete class label or category.
Regression is the process of predicting a continuous value as opposed to predicting a categorical value in Classification.
Look at this dataset.
It is related to Co2 emissions of different cars.
It includes Engine size, Cylinders, Fuel Consumption and Co2 emission of various models of automobiles.
Given this dataset, you can use regression to predict the Co2 emission of a new car by using other fields, such as Engine size or number of Cylinders.
Since we know the meaning of supervised learning,
what do you think unsupervised learning means?
Yes!
Unsupervised Learning is exactly as it sounds.
We do not supervise the model, but we let the model work on its own to discover information that may not be visible to the human eye.
It means, The Unsupervised algorithm trains on the dataset, and draws conclusions on UNLABELED
data.
Generally speaking, unsupervised learning has more difficult algorithms than supervised learning, since we know little to no information about the data, or the outcomes that are to be expected.
Dimension reduction, Density estimation, Market basket analysis and Clustering are the most widely used unsupervised machine learning techniques.
Dimensionality Reduction and/or feature selection play a large role in this by reducing redundant
features to make the classification easier.
Market basket analysis is a modelling technique based upon the theory that if you buy a certain
group of items, you're more likely to buy another group of items.
Density estimation is a very simple concept that is mostly used to explore the data to find some structure within it.
And finally, clustering.
Clustering is considered to be one of the most popular unsupervised machine learning techniques used for grouping data points or objects that are somehow similar.
Cluster analysis has many applications in different domains, whether it be a bank's

desire to segment its customers based on certain characteristics, or helping an individual to organize and group his/her favourite types of music!

Generally speaking, though, Clustering is used mostly for: Discovering structure, Summarization,

and Anomaly detection.

So, to recap, the biggest difference between Supervised and Unsupervised Learning is that supervised learning deals with labeled data while Unsupervised Learning deals with unlabeled data.

In supervised learning, we have machine learning algorithms for Classification and Regression.

In unsupervised learning, we have methods such as clustering.

In comparison to supervised learning, unsupervised learning has fewer models and fewer evaluation

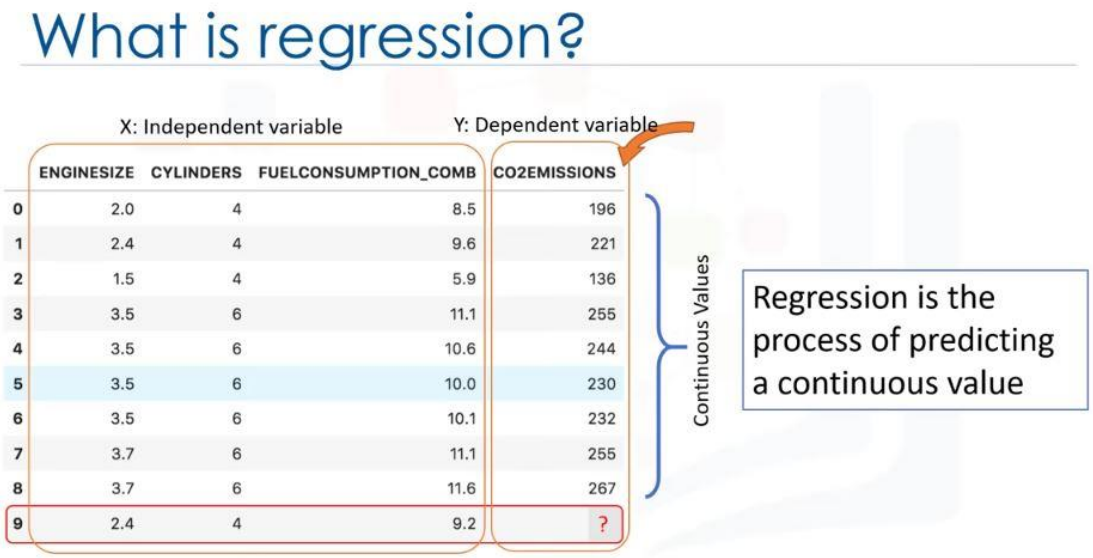
methods that can be used to ensure that the outcome of the model is accurate.

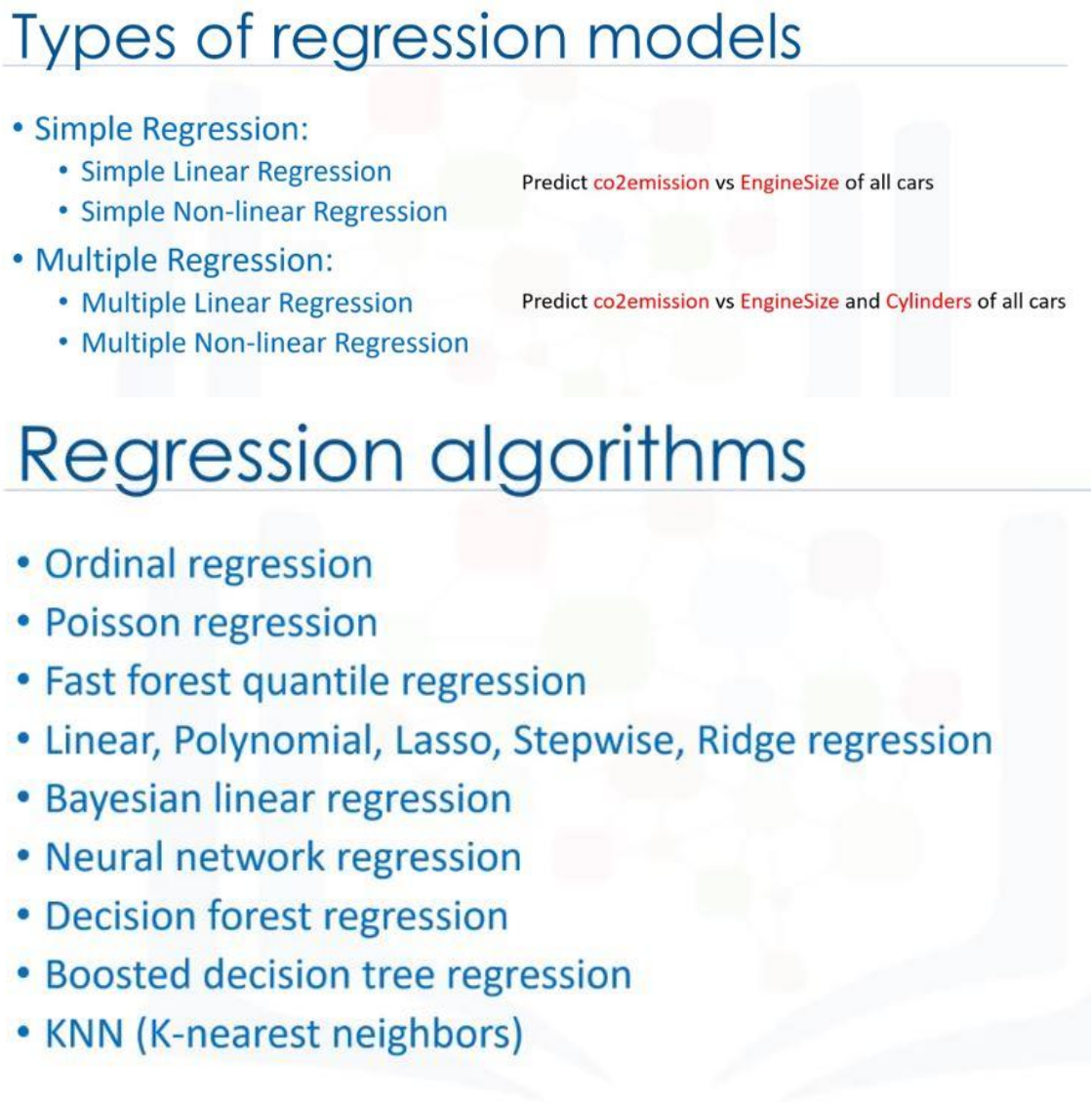
As such, unsupervised learning creates a less controllable environment, as the machine is creating outcomes for us.

Thanks for watching!

MODULE 2 – REGRESSION

INTRO





Hello, and welcome!
In this video, we'll be giving a brief introduction to regression.
So let's get started.
Look at this dataset.
It's related to Co2 emissions from different cars.
It includes Engine size, number of Cylinders, Fuel Consumption and Co2 emission from various automobile models.
The question is, "Given this dataset, can we predict the Co2 emission of a car using other fields, such as EngineSize or Cylinders?"
Let's assume we have some historical data from different cars, and assume that a car, such as in row 9, has not been manufactured yet, but we're interested in estimating its approximate Co2 emission, after production.
Is it possible?
We can use regression methods to predict a continuous value, such as CO2 Emission, using some other variables.
Indeed, regression is the process of predicting a continuous value.
In regression there are two types of variables: a dependent variable and one or more independent
variables.

The dependent variable can be seen as the "state", "target" or "final goal" we study and try to predict, and the independent variables, also known as explanatory variables, can be seen as the "causes" of those "states".
The independent variables are shown conventionally by x; and the dependent variable is notated by y.
A regression model relates y, or the dependent variable, to a function of x, i.e., the independent variables.
The key point in the regression is that our dependent value should be continuous, and cannot be a discreet value.
However, the independent variable or variables can be measured on either a categorical or continuous measurement scale.
So, what we want to do here is to use the historical data of some cars, using one or more of their features, and from that data, make a model.
We use regression to build such a regression/estimation model.
Then the model is used to predict the expected Co2 emission for a new or unknown car.
Basically there are 2 types of regression models: simple regression and multiple regression.
Simple regression is when one independent variable is used to estimate a dependent variable.
It can be either linear on non-linear.
For example, predicting Co2emission using the variable of EngineSize.
Linearity of regression is based on the nature of relationship between independent and dependent
variables.
When more than one independent variable is present, the process is called multiple linear regression.
For example, predicting Co2emission using EngineSize and the number of Cylinders in any given car.
Again, depending on the relation between dependent and independent variables, it can be either
linear or non-linear regression.
Let’s examine some sample applications of regression.
Essentially, we use regression when we want to estimate a continuous value.
For instance, one of the applications of regression analysis could be in the area of sales forecasting.
You can try to predict a salesperson's total yearly sales from independent variables such as age, education, and years of experience.
It can also be used in the field of psychology, for example, to determine individual satisfaction based on demographic and psychological factors.
We can use regression analysis to predict the price of a house in an area, based on its size, number of bedrooms, and so on.
We can even use it to predict employment income for independent variables, such as hours of work, education, occupation, sex, age, years of experience, and so on.
Indeed, you can find many examples of the usefulness of regression analysis in these and many other fields or domains, such as finance, healthcare, retail, and more.
We have many regression algorithms.
Each of them has its own importance and a specific condition to which their application is best suited.
And while we've covered just a few of them in this course, it gives you enough base knowledge for you to explore different regression techniques.
Thanks for watching!

SIMPLE LIENAR REGRESSION

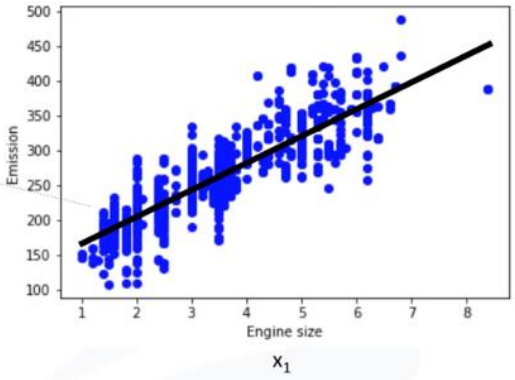
Linear regression model representation

$\hat{y} = \theta_0 + \theta_1 x_1$

response variable

a single predictor

y



How to find the best fit?

$x_1 = 2.4$ independent variable

$y = 250$ actual Co2 emission of x1

$\hat{y} = \theta_0 + \theta_1 x_1$

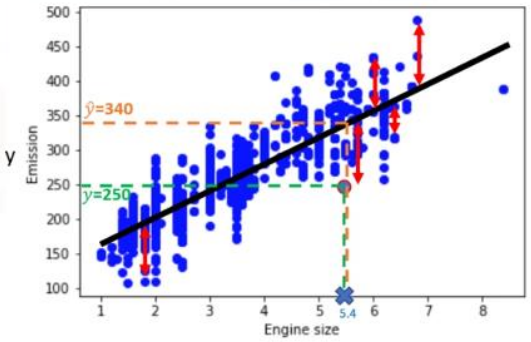
$\hat{y} = 340$ the predicted emission of x1

Error = $y - \hat{y}$

= 250 – 340

= -90

$$MSE = \frac{1}{n}\sum^{n}(y_i - \hat{y}_i)^2$$



Estimating the parameters

	ENGINESIZE	CYLINDERS	FUELCONSUMPTION_COMB	CO2EMISSIONS
0	2.0	4	8.5	196
1	2.4	4	9.6	221
2	1.5	4	5.9	136
3	3.5	6	11.1	255
4	3.5	6	10.6	244
5	3.5	6	10.0	230
6	3.5	6	10.1	232
7	3.7	6	11.1	255
8	3.7	6	11.6	267

x_1 y

$$\hat{y} = \theta_0 + \theta_1 x_1$$

$$\theta_1 = \frac{\sum_{i=1}^{s}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{s}(x_i - \bar{x})^2}$$

$\bar{x}$ = (2.0 + 2.4 + 1.5 + …)/9 = 3.34

$\bar{y}$ = (196 + 221 + 136 + …)/9 = 256

$$\theta_1 = \frac{(2.0 - 3.34)(196 - 256) + (2.4 - 3.34)(221 - 256) + \ldots}{(2.0 - 3.34)^2 + (2.4 - 3.34)^2 + \ldots}$$

$\theta_1 = 39$

$$\theta_0 = \bar{y} - \theta_1 \bar{x}$$

$\theta_0 = 256 - 39 * 3.34$

$\theta_0 = 125.74$

$$\hat{y} = 125.74 + 39 x_1$$

COGNITIVE CLASS

7

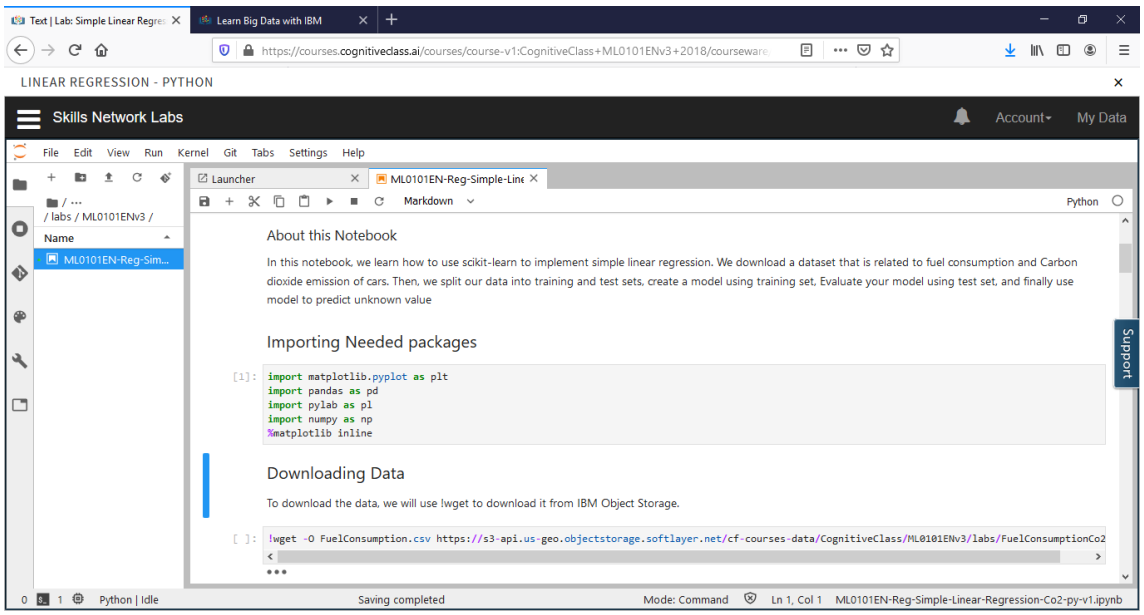
Hello, and welcome! In this video, we'll be covering linear regression.

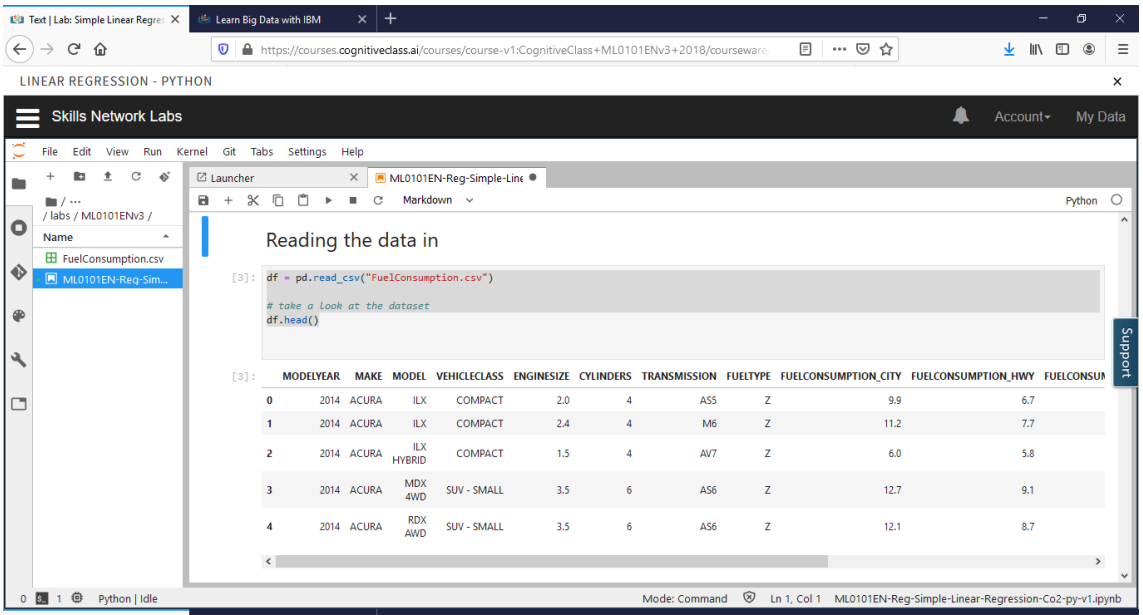
You don't need to know any linear algebra to understand topics in linear regression.
This high-level introduction will give you enough background information on linear regression to be able to use it effectively on your own problems.
So, let's get started.
Let's take a look at this dataset. It's related to the Co2 emission of different cars. It includes Engine size, Cylinders, Fuel Consumption and Co2 emissions for various car models. The question is: Given this dataset, can we predict the Co2 emission of a car, using another field, such as Engine size?
Quite simply, yes! We can use linear regression to predict a continuous value such as Co2 Emission, by using other variables.
Linear regression is the approximation of a linear model used to describe the relationship between two or more variables. In simple linear regression, there are two variables: a dependent variable and an independent variable.
The key point in the linear regression is that our dependent value should be continuous and cannot be a discreet value. However, the independent variable(s) can be measured on either a categorical or continuous measurement scale.
There are two types of linear regression models. They are: simple regression and multiple regression.
Simple linear regression is when one independent variable is used to estimate a dependent variable. For example, predicting Co2 emission using the EngineSize variable. When more than one independent variable is present, the process is called multiple linear regression.
For example, predicting Co2 emission using EngineSize and Cylinders of cars.
Our focus in this video is on simple linear regression.
Now, let's see how linear regression works. OK, so let's look at our dataset again.
To understand linear regression, we can plot our variables here.
We show Engine size as an independent variable, and Emission as the target value that we would like to predict. A scatterplot clearly shows the relation between variables where changes in one variable "explain" or possibly "cause" changes in the other variable.
Also, it indicates that these variables are linearly related.
With linear regression you can fit a line through the data.
For instance, as the EngineSize increases, so do the emissions.
With linear regression, you can model the relationship of these variables.
A good model can be used to predict what the approximate emission of each car is.
How do we use this line for prediction now? Let us assume, for a moment, that the line is a good fit of data. We can use it to predict the emission of an unknown car. For example, for a sample car, with engine size 2.4, you can find the emission is 214.
Now, let's talk about what this fitting line actually is.
We're going to predict the target value, y.
In our case, using the independent variable, "Engine Size," represented by x1.
The fit line is shown traditionally as a polynomial. In a simple regression problem (a single x), the form of the model would be $\theta 0 + \theta 1\, x1$. In this equation, $\hat{y}$ is the dependent variable or the predicted value, and x1 is the independent variable; $\theta 0$ and $\theta 1$ are the parameters of the line that we must adjust. $\theta 1$ is known as the "slope" or "gradient" of the fitting line and $\theta 0$ is known as the "intercept."
$\theta 0$ and $\theta 1$ are also called the coefficients of the linear equation.
You can interpret this equation as $\hat{y}$ being a function of x1, or $\hat{y}$ being dependent of x1.
Now the questions are: "How would you draw a line through the points?" And, "How do you determine which line 'fits best'?"

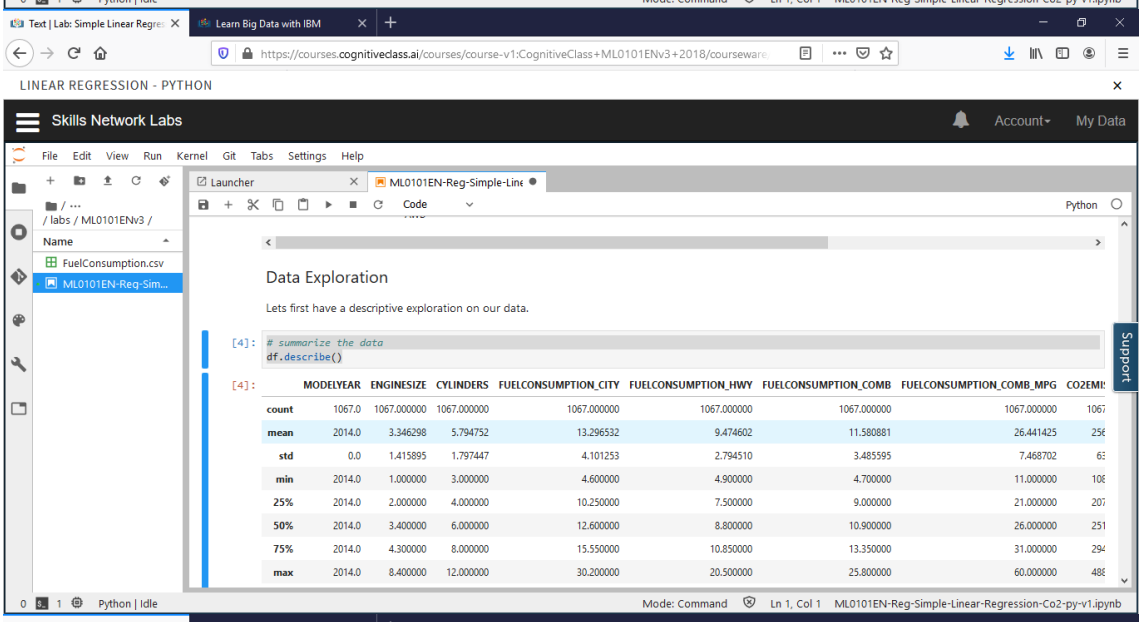
Linear regression estimates the coefficients of the line.
This means we must calculate θ0 and θ1 to find the best line to 'fit' the data.
This line would best estimate the emission of the unknown data points.
Let's see how we can find this line, or to be more precise, how we can adjust the parameters to make the line the best fit for the data.
For a moment, let's assume we've already found the best fit line for our data.
Now, let's go through all the points and check how well they align with this line.
Best fit, here, means that if we have, for instance, a car with engine size x1=5.4, and actual Co2=250, its Co2 should be predicted very close to the actual value, which is y=250, based on historical data.
But, if we use the fit line, or better to say, using our polynomial with known parameters to predict the Co2 emission, it will return $\hat{y}$ =340.
Now, if you compare the actual value of the emission of the car with what we predicted using our model, you will find out that we have a 90-unit error.
This means our prediction line is not accurate. This error is also called the residual error.
So, we can say the error is the distance from the data point to the fitted regression line.
The mean of all residual errors shows how poorly the line fits with the whole dataset.
Mathematically, it can be shown by the equation, mean squared error, shown as (MSE).
Our objective is to find a line where the mean of all these errors is minimized.
In other words, the mean error of the prediction using the fit line should be minimized.
Let's re-word it more technically. The objective of linear regression is to minimize this MSE equation, and to minimize it, we should find the best parameters, θ0 and θ1.
Now, the question is, how to find θ0 and θ1 in such a way that it minimizes this error?
How can we find such a perfect line? Or, said another way, how should we find the best parameters for our line? Should we move the line a lot randomly and calculate the MSE value every time, and choose the minimum one?
Not really! Actually, we have two options here:
Option 1 - We can use a mathematic approach. Or, Option 2 - We can use an optimization approach.
Let's see how we can easily use a mathematic formula to find the θ0 and θ1.
As mentioned before, θ0 and θ1, in the simple linear regression, are the coefficients of the fit line. We can use a simple equation to estimate these coefficients. That is, given that it's a simple linear regression, with only 2 parameters, and knowing that θ0 and θ1 are the intercept and slope of the line, we can estimate them directly from our data.
It requires that we calculate the mean of the independent and dependent or target columns, from the dataset. Notice that all of the data must be available to traverse and calculate the parameters. It can be shown that the intercept and slope can be calculated using these equations. We can start off by estimating the value for θ1.
This is how you can find the slope of a line based on the data. $\bar{x}$ is the average value for the engine size in our dataset. Please consider that we have 9 rows here, row 0 to 8. First, we calculate the average of x1 and average of y. Then we plug it into the slope equation, to find θ1. The xi and yi in the equation refer to the fact that we need to repeat these calculations across all values in our dataset and i refers to the i'th value of x or y.
Applying all values, we find θ1=39; it is our second parameter.
It is used to calculate the first parameter, which is the intercept of the line.
Now, we can plug θ1 into the line equation to find θ0.
It is easily calculated that θ0=125.74. So, these are the two parameters for the line, where θ0 is also called the bias coefficient and θ1 is the coefficient for the Co2 Emission

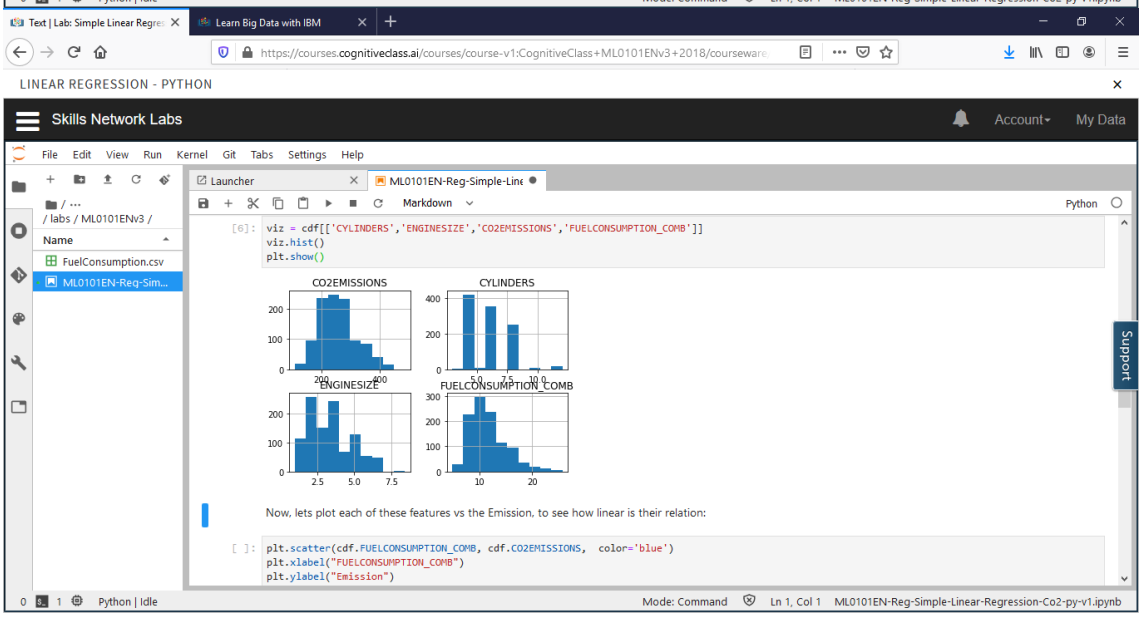
column. As a side note, you really don't need to
remember the formula for calculating these parameters, as most of the libraries used for machine learning in Python, R, and Scala can easily find these parameters for you. But it's always good to understand how it works.
Now, we can write down the polynomial of the line.
So, we know how to find the best fit for our data, and its equation.
Now the question is: "How can we use it to predict the emission of a new car based on its engine size?"
After we found the parameters of the linear equation, making predictions is as simple as solving the equation for a specific set of inputs.
Imagine we are predicting Co2 Emission(y) from EngineSize(x) for the Automobile in record number 9. Our linear regression model representation
for this problem would be: $\hat{y} = \theta 0 + \theta 1\ x1$.
Or if we map it to our dataset, it would be Co2Emission = θ0 + θ1 EngineSize.
As we saw, we can find θ0, θ1 using the equations that we just talked about.
Once found, we can plug in the equation of the linear model.
For example, let's use θ0=125 and θ1=39. So, we can rewrite the linear model as $Co2Emission$=125+39$EngineSize$.
Now, let's plug in the 9th row of our dataset and calculate the Co2 Emission for a car with an EngineSize of 2.4. So Co2Emission = 125 + 39 × 2.4.
Therefore, we can predict that the Co2 Emission for this specific car would be 218.6.
Let's talk a bit about why Linear Regression is so useful.
Quite simply, it is the most basic regression to use and understand.
In fact, one reason why Linear Regression is so useful is that it's fast!
It also doesn't require tuning of parameters. So, something like tuning the K parameter in K-Nearest Neighbors or the learning rate in Neural Networks isn't something to worry about. Linear Regression is also easy to understand
and highly interpretable.
Thanks for watching this video.

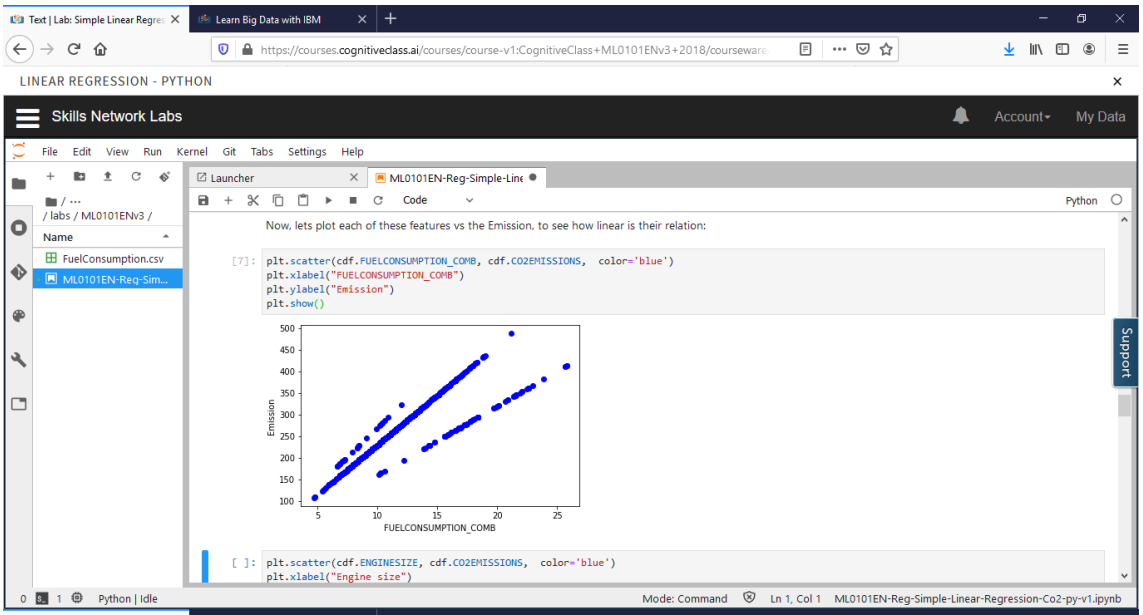
>>Lab:

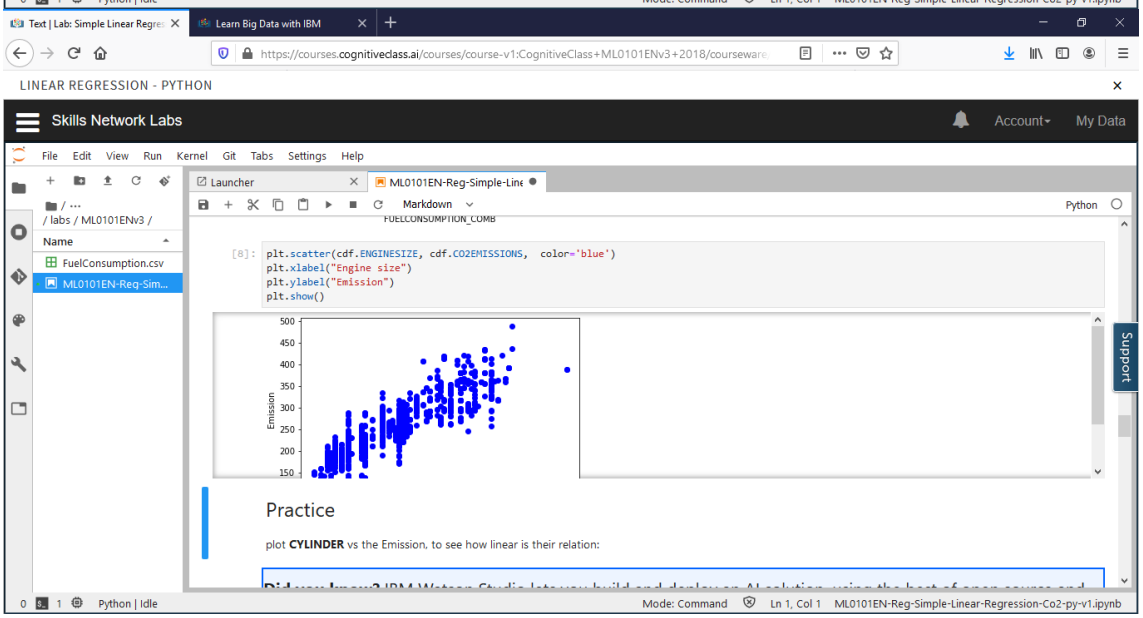


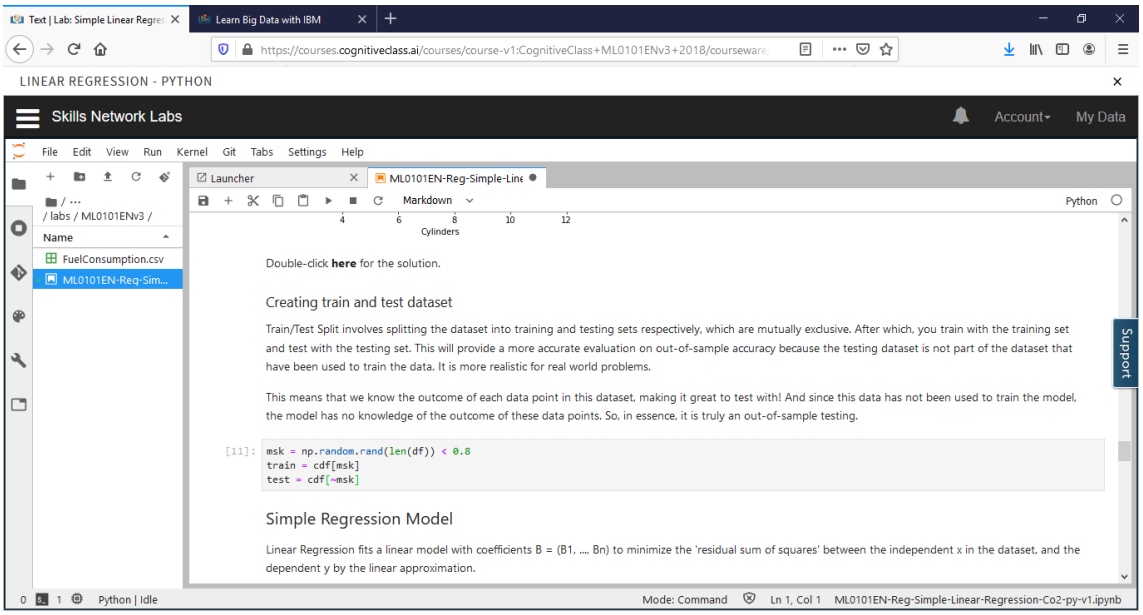

LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
Reading the data in
[3]:
df = pd.read_csv("FuelConsumption.csv")
# take a look at the dataset
df.head()
[3]:
MODELYEAR MAKE MODEL VEHICLECLASS ENGINESIZE CYLINDERS TRANSMISSION FUELTYPE FUELCONSUMPTION_CITY FUELCONSUMPTION_HWY FUELCONSUM
0 2014 ACURA ILX COMPACT 2.0 4 AS5 Z 9.9 6.7
1 2014 ACURA ILX COMPACT 2.4 4 M6 Z 11.2 7.7
2 2014 ACURA ILX HYBRID COMPACT 1.5 4 AV7 Z 6.0 5.8
3 2014 ACURA MDX 4WD SUV - SMALL 3.5 6 AS6 Z 12.7 9.1
4 2014 ACURA RDX AWD SUV - SMALL 3.5 6 AS6 Z 12.1 8.7
Support
Python | Idle
Mode: Command
Ln 1, Col 1
ML0101EN-Reg-Simple-Linear-Regression-Co2-py-v1.ipynb

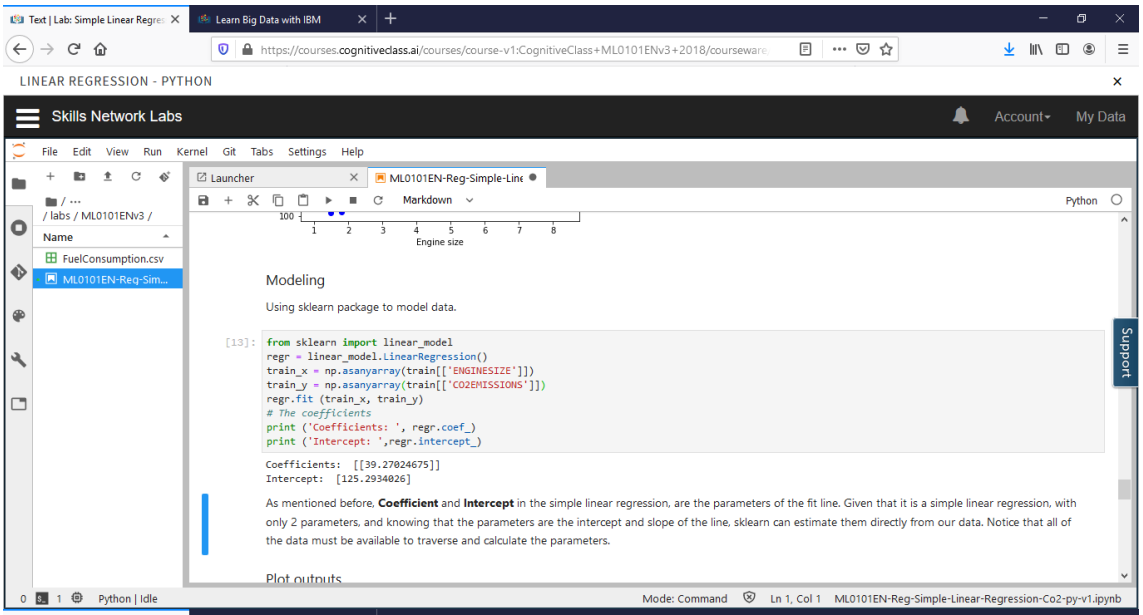

LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
Data Exploration
Lets first have a descriptive exploration on our data.
[4]:
# summarize the data
df.describe()
[4]:
MODELYEAR ENGINESIZE CYLINDERS FUELCONSUMPTION_CITY FUELCONSUMPTION_HWY FUELCONSUMPTION_COMB FUELCONSUMPTION_COMB_MPG CO2EMI
count 1067.0 1067.000000 1067.000000 1067.000000 1067.000000 1067.000000 1067.000000 1067
mean 2014.0 3.346298 5.794752 13.296532 9.474602 11.580881 26.441425 256
std 0.0 1.415895 1.797447 4.101253 2.794510 3.485595 7.468702 63
min 2014.0 1.000000 3.000000 4.600000 4.900000 4.700000 11.000000 108
25% 2014.0 2.000000 4.000000 10.250000 7.500000 9.000000 21.000000 207
50% 2014.0 3.400000 6.000000 12.600000 8.800000 10.900000 26.000000 251
75% 2014.0 4.300000 8.000000 15.550000 10.850000 13.350000 31.000000 294
max 2014.0 8.400000 12.000000 30.200000 20.500000 25.800000 60.000000 488
Support
Python | Idle
Mode: Command
Ln 1, Col 1
ML0101EN-Reg-Simple-Linear-Regression-Co2-py-v1.ipynb

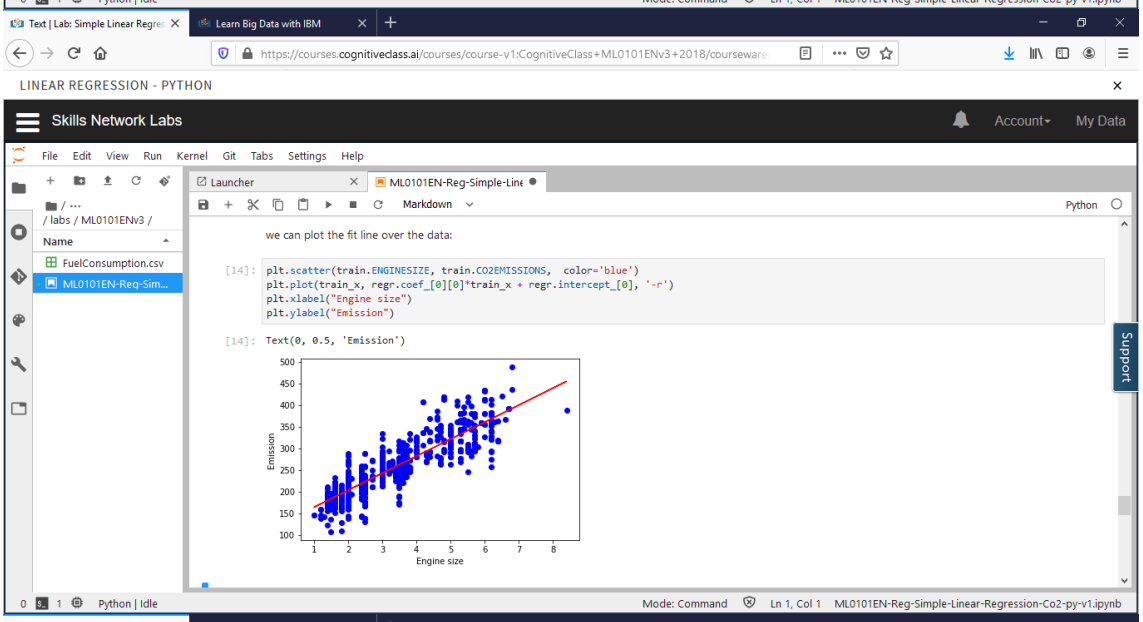

LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
[6]:
viz = cdf[['CYLINDERS','ENGINESIZE','CO2EMISSIONS','FUELCONSUMPTION_COMB']]
viz.hist()
plt.show()
CO2EMISSIONS
CYLINDERS
ENGINESIZE
FUELCONSUMPTION_COMB
Now, lets plot each of these features vs the Emission, to see how linear is their relation:
[ ]:
plt.scatter(cdf.FUELCONSUMPTION_COMB, cdf.CO2EMISSIONS, color='blue')
plt.xlabel("FUELCONSUMPTION_COMB")
plt.ylabel("Emission")
Support
Python | Idle
Mode: Command
Ln 1, Col 1
ML0101EN-Reg-Simple-Linear-Regression-Co2-py-v1.ipynb

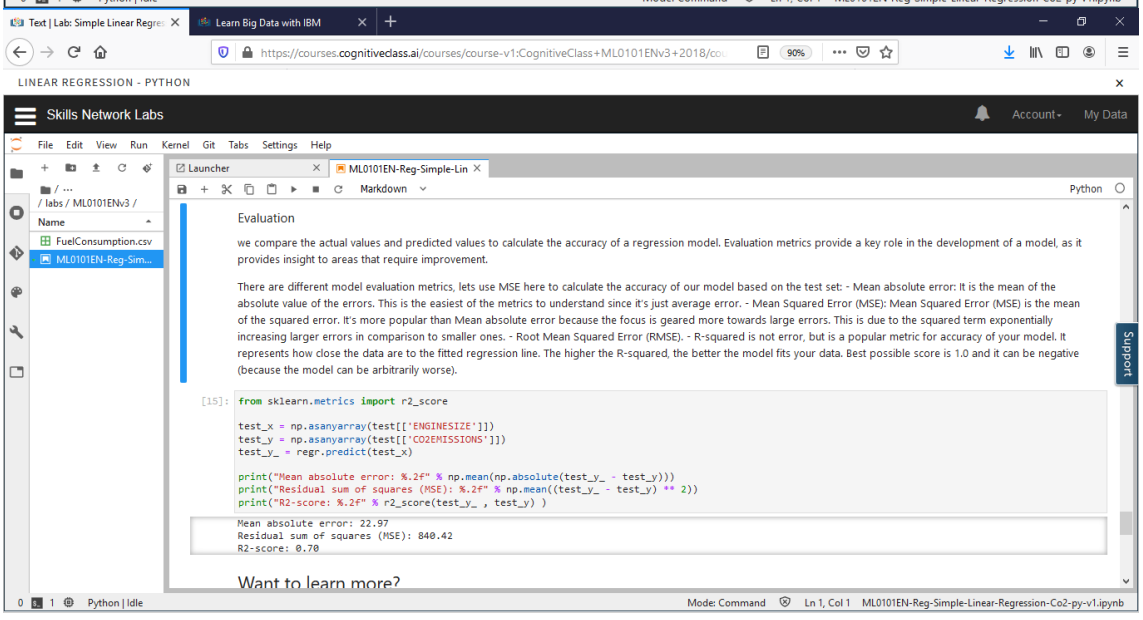

LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
Now, lets plot each of these features vs the Emission, to see how linear is their relation:
plt.scatter(cdf.FUELCONSUMPTION_COMB, cdf.CO2EMISSIONS, color='blue')
plt.xlabel("FUELCONSUMPTION_COMB")
plt.ylabel("Emission")
plt.show()
Emission
FUELCONSUMPTION_COMB
plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS, color='blue')
plt.xlabel("Engine size")


LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
plt.scatter(cdf.ENGINESIZE, cdf.CO2EMISSIONS, color='blue')
plt.xlabel("Engine size")
plt.ylabel("Emission")
plt.show()
Emission
Practice
plot CYLINDER vs the Emission, to see how linear is their relation:


LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
Cylinders
Double-click here for the solution.
Creating train and test dataset
Train/Test Split involves splitting the dataset into training and testing sets respectively, which are mutually exclusive. After which, you train with the training set and test with the testing set. This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is not part of the dataset that have been used to train the data. It is more realistic for real world problems.
This means that we know the outcome of each data point in this dataset, making it great to test with! And since this data has not been used to train the model, the model has no knowledge of the outcome of these data points. So, in essence, it is truly an out-of-sample testing.
msk = np.random.rand(len(df)) < 0.8
train = cdf[msk]
test = cdf[~msk]
Simple Regression Model
Linear Regression fits a linear model with coefficients B = (B1, ... Bn) to minimize the 'residual sum of squares' between the independent x in the dataset, and the dependent y by the linear approximation.


LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
Modeling
Using sklearn package to model data.
[13]:
from sklearn import linear_model
regr = linear_model.LinearRegression()
train_x = np.asanyarray(train[['ENGINESIZE']])
train_y = np.asanyarray(train[['CO2EMISSIONS']])
regr.fit (train_x, train_y)
The coefficients
print ('Coefficients: ', regr.coef_)
print ('Intercept: ',regr.intercept_)
Coefficients: [[39.27024675]]
Intercept: [125.2934026]
As mentioned before, Coefficient and Intercept in the simple linear regression, are the parameters of the fit line. Given that it is a simple linear regression, with only 2 parameters, and knowing that the parameters are the intercept and slope of the line, sklearn can estimate them directly from our data. Notice that all of the data must be available to traverse and calculate the parameters.
Plot outputs
Engine size
Support


LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
we can plot the fit line over the data:
[14]:
plt.scatter(train.ENGINESIZE, train.CO2EMISSIONS, color='blue')
plt.plot(train_x, regr.coef_[0][0]*train_x + regr.intercept_[0], '-r')
plt.xlabel("Engine size")
plt.ylabel("Emission")
[14]: Text(0, 0.5, 'Emission')
Emission
Engine size
Support


LINEAR REGRESSION - PYTHON
Skills Network Labs
Account
My Data
Evaluation
we compare the actual values and predicted values to calculate the accuracy of a regression model. Evaluation metrics provide a key role in the development of a model, as it provides insight to areas that require improvement.
There are different model evaluation metrics, lets use MSE here to calculate the accuracy of our model based on the test set: - Mean absolute error: It is the mean of the absolute value of the errors. This is the easiest of the metrics to understand since it's just average error. - Mean Squared Error (MSE): Mean Squared Error (MSE) is the mean of the squared error. It's more popular than Mean absolute error because the focus is geared more towards large errors. This is due to the squared term exponentially increasing larger errors in comparison to smaller ones. - Root Mean Squared Error (RMSE). - R-squared is not error, but is a popular metric for accuracy of your model. It represents how close the data are to the fitted regression line. The higher the R-squared, the better the model fits your data. Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse).
[15]:
from sklearn.metrics import r2_score
test_x = np.asanyarray(test[['ENGINESIZE']])
test_y = np.asanyarray(test[['CO2EMISSIONS']])
test_y_ = regr.predict(test_x)
print("Mean absolute error: %.2f" % np.mean(np.absolute(test_y_ - test_y)))
print("Residual sum of squares (MSE): %.2f" % np.mean((test_y_ - test_y) ** 2))
print("R2-score: %.2f" % r2_score(test_y_ , test_y))
Mean absolute error: 22.97
Residual sum of squares (MSE): 840.42
R2-score: 0.70
Want to learn more?
Support

MULTIPLE LINEAR REGRESSION

Hello, and welcome! In this video, we'll be covering multiple linear regression.
As you know there are two types of linear regression models: simple regression and multiple regression. Simple linear regression is when one independent variable is used to estimate a dependent variable. For example, predicting Co2 emission using the variable of EngineSize. In reality, there are multiple variables that predict the Co2 emission. When multiple independent variables are present, the process is called "multiple linear regression." For example, predicting Co2 emission using EngineSize and the number of Cylinders in the car's engine.
Our focus in this video is on multiple linear regression.
The good thing is that multiple linear regression is the extension of the simple linear regression model. So, I suggest you go through the Simple Linear Regression video first, if you haven't watched it already.
Before we dive into a sample dataset and see how multiple linear regression works, I want to tell you what kind of problems it can solve; when we should use it; and, specifically, what kind of questions we can answer using it.
Basically, there are two applications for multiple linear regression.
First, it can be used when we would like to identify the strength of the effect that the independent variables have on a dependent variable.
For example, does revision time, test anxiety, lecture attendance, and gender, have any effect on exam performance of students? Second, it can be used to predict the impact of changes. That is, to understand how the dependent variable changes when we change the independent variables. For example, if we were reviewing a person's health data, a multiple linear regression can tell you how much that person's blood pressure goes up (or down) for every unit increase (or decrease) in a patient's body mass index (BMI), holding other factors constant.
As is the case with simple linear regression, multiple linear regression is a method of predicting a continuous variable. It uses multiple variables, called independent variables, or predictors, that best predict the value of the target variable, which is also called the dependent variable. In multiple linear regression, the target value, y, is a linear combination of independent variables, x.
For example, you can predict how much Co2 a car might emit due to independent variables, such as the car's Engine Size, Number of Cylinders and Fuel Consumption.
Multiple linear regression is very useful because you can examine which variables are significant predictors of the outcome variable. Also, you can find out how each feature impacts the outcome variable. And again, as is the case in simple linear regression, if you manage to build such a regression model, you can use it to predict the emission amount of an unknown case, such as record number 9.
Generally, the model is of the form: $\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ and so on, up to ... $+\theta_n x_n$.
Mathematically, we can show it as a vector form as well.
This means, it can be shown as a dot product of 2 vectors: the parameters vector and the feature set vector.
Generally, we can show the equation for a multi-dimensional space as $\theta^T x$, where θ is an n-by-one vector of unknown parameters in a multi-dimensional space, and x is the vector of the feature sets, as θ is a vector of coefficients, and is supposed to be multiplied by x. Conventionally, it is shown as transpose θ.
θ is also called the parameters, or, weight vector of the regression equation … both these terms can be used interchangeably. And x is the feature set, which represents

a car. For example x1 for engine size, or x2 for cylinders, and so on.
The first element of the feature set would be set to 1, because it turns the θ_0 into the intercept or bias parameter when the vector is multiplied by the parameter vector.
Please notice that θ^T x in a one dimensional space, is the equation of a line.
It is what we use in simple linear regression. In higher dimensions, when we have more than one input (or x), the line is called a plane or a hyper-plane.
And this is what we use for multiple linear regression.
So, the whole idea is to find the best fit hyper-plane for our data.
To this end, and as is the case in linear regression, we should estimate the values for θ vector that best predict the value of the target field in each row.
To achieve this goal, we have to minimize the error of the prediction.
Now, the question is, "How do we find the optimized parameters?"
To find the optimized parameters for our model, we should first understand what the optimized parameters are. Then we will find a way to optimize the parameters.
In short, optimized parameters are the ones which lead to a model with the fewest errors.
Let's assume, for a moment, that we have already found the parameter vector of our model. It means we already know the values of θ vector. Now, we can use the model, and the feature set of the first row of our dataset to predict the Co2 emission for the first car, correct?
If we plug the feature set values into the model equation, we find $\hat{y}$.
Let's say, for example, it returns 140 as the predicted value for this specific row.
What is the actual value? y=196. How different is the predicted value from the actual value of 196? Well, we can calculate it quite simply, as 196-140, which of course = 56. This is the error of our model, only for one row, or one car, in our case. As is the case in linear regression, we can say the error here is the distance from the data point to the fitted regression model.
The mean of all residual errors shows how bad the model is representing the dataset.
It is called the mean squared error, or MSE.
Mathematically, MSE can be shown by an equation. While this is not the only way to expose the error of a multiple linear regression model, it is one the most popular ways to do so.
The best model for our dataset is the one with minimum error for all prediction values.
So, the objective of multiple linear regression is to minimize the MSE equation.
To minimize it, we should find the best parameters θ, but how?
Okay, "How do we find the parameter or coefficients for multiple linear regression?"
There are many ways to estimate the value of these coefficients.
However, the most common methods are the ordinary least squares and optimization approach.
Ordinary least squares tries to estimate the values of the coefficients by minimizing the "Mean Square Error." This approach uses the data as a matrix and uses linear algebra operations to estimate the optimal values for the theta.
The problem with this technique is the time complexity of calculating matrix operations, as it can take a very long time to finish. When the number of rows in your dataset is less 10,000 you can think of this technique as an option, however, for greater values, you should try other faster approaches.
The second option is to use an optimization algorithm to find the best parameters.
That is, you can use a process of optimizing the values of the coefficients by iteratively minimizing the error of the model on your training data.
For example, you can use Gradient Descent, which starts optimization with random values for each coefficient. Then, calculates the errors, and tries to minimize it through wise changing of the coefficients in multiple iterations.
Gradient descent is a proper approach if you have a large dataset.
Please understand, however, that there are other approaches to estimate the parameters of the multiple linear regression that you can explore on your own.

After you find the best parameters for your model, you can go to the prediction phase.
After we found the parameters of the linear equation, making predictions is as simple as solving the equation for a specific set of inputs.
Imagine we are predicting Co2 emission (or y) from other variables for the automobile in record number 9. Our linear regression model representation for this problem would be: $\hat{y} = \theta^T x$. Once we find the parameters, we can plug them into the equation of the linear model. For example, let's use $\theta 0 = 125$, $\theta 1 = 6.2$, $\theta 2 = 14$, and so on. If we map it to our dataset, we can rewrite the linear model as "Co2Emission=125 plus 6.2 multiplied by EngineSize plus 14 multiplied by Cylinder," and so on. As you can see, multiple linear regression estimates the relative importance of predictors. For example, it shows Cylinder has higher impact on Co2 emission amounts in comparison with EngineSize.
Now, let's plug in the 9th row of our dataset and calculate the Co2 emission for a car with the EngineSize of 2.4. So $\text{Co2Emission} = 125 + 6.2 \times 2.4 + 14 \times 4$ … and so on. We can predict the Co2 emission for this specific car would be 214.1.
Now let me address some concerns that you might already be having regarding multiple linear regression. As you saw, you can use multiple independent variables to predict a target value in multiple linear regression.
It sometimes results in a better model compared to using a simple linear regression, which uses only one independent variable to predict the dependent variable.
Now, the question is, "How many independent variables should we use for the prediction?" Should we use all the fields in our dataset? Does adding independent variables to a multiple linear regression model always increase the accuracy of the model?
Basically, adding too many independent variables without any theoretical justification may result in an over-fit model. An over-fit model is a real problem because it is too complicated for your data set and not general enough to be used for prediction.
So, it is recommended to avoid using many variables for prediction.
There are different ways to avoid overfitting a model in regression, however, that is outside the scope of this video.
The next question is, "Should independent variables be continuous?"
Basically, categorical independent variables can be incorporated into a regression model by converting them into numerical variables. For example, given a binary variable such as car type, the code dummies "0" for "Manual" and 1 for "automatic" cars.
As a last point, remember that "multiple linear regression" is a specific type of linear regression. So, there needs to be a linear relationship between the dependent variable and each of your independent variables.
There are a number of ways to check for linear relationship.
For example, you can use scatterplots, and then visually check for linearity.
If the relationship displayed in your scatterplot is not linear, then, you need to use non-linear regression.
This concludes our video. Thanks for watching.

MODEL EVALUATION

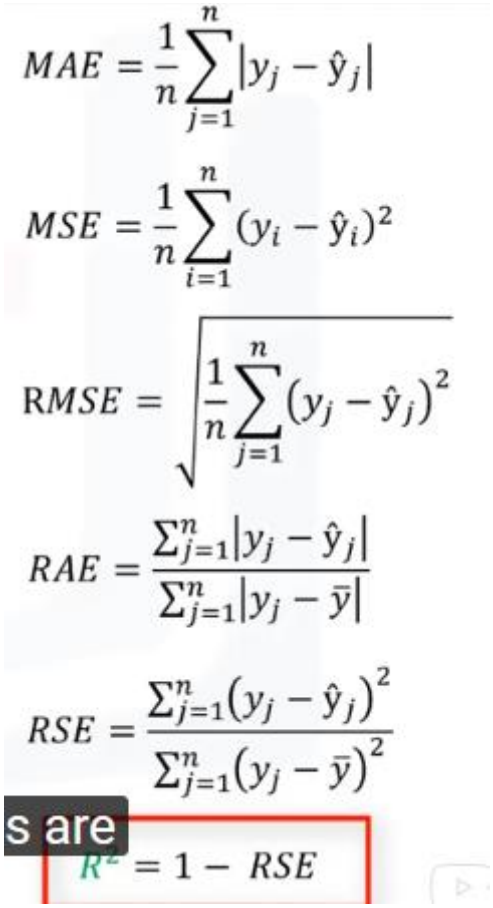
Hello, and welcome!
In this video, we'll be covering model evaluation. So, let's get started.
The goal of regression is to build a model to accurately predict an unknown case.
To this end, we have to perform regression evaluation after building the model.
In this video, we'll introduce and discuss two types of evaluation approaches that can be used to achieve this goal.

These approaches are: train and test on the same dataset, and train/test split.
We'll talk about what each of these are, as well as the pros and cons of using each of these models.
Also, we'll introduce some metrics for accuracy of regression models.
Let's look at the first approach.
When considering evaluation models, we clearly want to choose the one that will give us the most accurate results.
So, the question is, how we can calculate the accuracy of our model?
In other words, how much can we trust this model for prediction of an unknown sample, using a given a dataset and having built a model such as linear regression.
One of the solutions is to select a portion of our dataset for testing.
For instance, assume that we have 10 records in our dataset.
We use the entire dataset for training, and we build a model using this training set.
Now, we select a small portion of the dataset, such as row numbers 6 to 9, but without the labels.
This set, is called a test set, which has the labels, but the labels are not used for prediction, and is used only as ground truth.
The labels are called "Actual values" of the test set.
Now, we pass the feature set of the testing portion to our built model, and predict the target values.
Finally, we compare the predicted values by our model with the actual values in the test set.
This indicates how accurate our model actually is.
There are different metrics to report the accuracy of the model, but most of them work generally, based on the similarity of the predicted and actual values.
Let's look at one of the simplest metrics to calculate the accuracy of our regression model.
As mentioned, we just compare the actual values, y, with the predicted values, which is noted as $\hat{y}$ for the testing set.
The error of the model is calculated as the average difference between the predicted and actual values for all the rows.
We can write this error as an equation.
So, the first evaluation approach we just talked about is the simplest one: train and test on the SAME dataset.
Essentially, the name of this approach says it all … you train the model on the entire dataset, then you test it using a portion of the same dataset.
In a general sense, when you test with a dataset in which you know the target value for each data point, you're able to obtain a percentage of accurate predictions for the model.
This evaluation approach would most likely have a high "training accuracy" and a low "out-of-sample accuracy", since the model knows all of the testing data points from the training.
What is training accuracy and out-of-sample accuracy?
We said that training and testing on the same dataset produces a high training accuracy, but what exactly is "training accuracy?"
Training accuracy is the percentage of correct predictions that the model makes when using the test dataset.
However, a high training accuracy isn't necessarily a good thing.
For instance, having a high training accuracy may result in an 'over-fit' of the data.
This means that the model is overly trained to the dataset, which may capture noise and produce a non-generalized model.
Out-of-sample accuracy is the percentage of correct predictions that the model makes on data that the model has NOT been trained on.

Doing a “train and test” on the same dataset will most likely have low out-of-sample accuracy due to the likelihood of being over-fit.
It’s important that our models have high, out-of-sample accuracy, because the purpose of our model is, of course, to make correct predictions on unknown data.
So, how can we improve out-of-sample accuracy?
One way is to use another evaluation approach called "Train/Test Split."
In this approach, we select a portion of our dataset for training, for example, rows 0 to 5.
And the rest is used for testing, for example, rows 6 to 9.
The model is built on the training set.
Then, the test feature set is passed to the model for prediction.
And finally, the predicted values for the test set are compared with the actual values of the testing set.
This second evaluation approach, is called "Train/Test Split."
Train/Test Split involves splitting the dataset into training and testing sets, respectively, which are mutually exclusive, after which, you train with the training set and test with the testing set.
This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is NOT part of the dataset that has been used to train the data.
It is more realistic for real world problems.
This means that we know the outcome of each data point in this dataset, making it great to test with!
And since this data has not been used to train the model, the model has no knowledge of the outcome of these data points.
So, in essence, it’s truly out-of-sample testing.
However, please ensure that you train your model with the testing set afterwards, as you don’t want to lose potentially valuable data.
The issue with train/test split is that it’s highly dependent on the datasets on which the data was trained and tested.
The variation of this causes train/test split to have a better out-of-sample prediction than training and testing on the same dataset, but it still has some problems due to this dependency.
Another evaluation model, called "K-Fold Cross-validation," resolves most of these issues.
How do you fix a high variation that results from a dependency?
Well, you average it.
Let me explain the basic concept of “k-fold cross-validation” to see how we can solve this problem.
The entire dataset is represented by the points in the image at the top left.
If we have k=4 folds, then we split up this dataset as shown here.
In the first fold, for example, we use the first 25 percent of the dataset for testing, and the rest for training.
The model is built using the training set, and is evaluated using the test set.
Then, in the next round (or in the second fold), the second 25 percent of the dataset is used for testing and the rest for training the model.
Again the accuracy of the model is calculated.
We continue for all folds.
Finally the result of all 4 evaluations are averaged.
That is, the accuracy of each fold is then averaged, keeping in mind that each fold is distinct, where no training data in one fold is used in another.
K-fold cross-validation, in its simplest form, performs multiple train/test splits using the same dataset where each split is different.
Then, the result is averaged to produce a more consistent out-of-sample accuracy.

We wanted to show you an evaluation model that addressed some of the issues we've described in the previous approaches.
However, going in-depth with the K-fold cross-validation model is out of the scope for this course.
Thanks for watching!

EVALUATION METRICS



Hello, and welcome!
In this video, we'll be covering accuracy metrics for model evaluation.
So, let's get started.
Evaluation metrics are used to explain the performance of a model.
Let's talk more about the model evaluation metrics that are used for regression.
As mentioned, basically, we can compare the actual values and predicted values to calculate the accuracy of a regression model.

Evaluation metrics provide a key role in the development of a model, as it provides insight to areas that require improvement.

We'll be reviewing a number of model evaluation metrics, including:

Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE).

But, before we get into defining these, we need to define what an error actually is.

In the context of regression, the error of the model is the difference between the data points and the trend line generated by the algorithm.

Since there are multiple data points, an error can be determined in multiple ways.

Mean absolute error is the mean of the absolute value of the errors.

This is the easiest of the metrics to understand, since it's just the average error.

Mean Squared Error (MSE) is the mean of the squared error.

It's more popular than Mean absolute error because the focus is geared more towards large errors.

This is due to the squared term exponentially increasing larger errors in comparison to smaller ones.

Root Mean Squared Error (RMSE) is the square
root of the mean squared error.

This is one of the most popular of the evaluation metrics because Root Mean Squared Error is interpretable in the same units as the response vector (or 'y' units) making it easy to relate its information.

Relative Absolute Error (RAE), also known as Residual sum of square, where y-bar is a mean value

of y, takes the total absolute error and normalizes it by dividing by the total absolute error of the simple predictor.

Relative Squared Error (RSE) is very similar to "Relative absolute error ", but is widely adopted by the data science community, as it is used for calculating R-squared.

R-squared is not error, per se, but is a popular metric for the accuracy of your model.

It represents how close the data values are to the fitted regression line.

The higher the R-squared, the better the model fits your data.

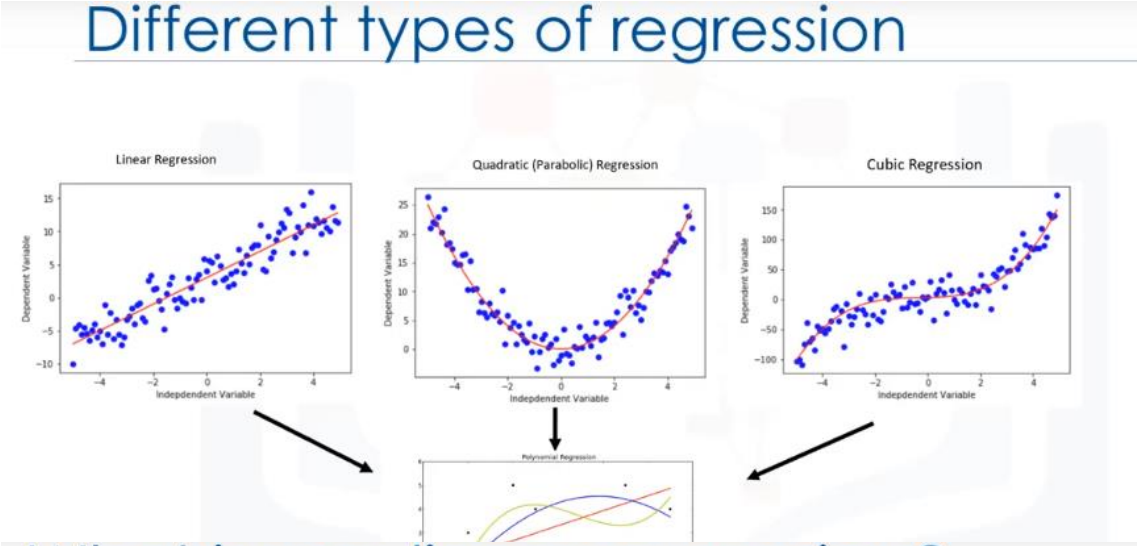
Each of these metrics can be used for quantifying of your prediction.

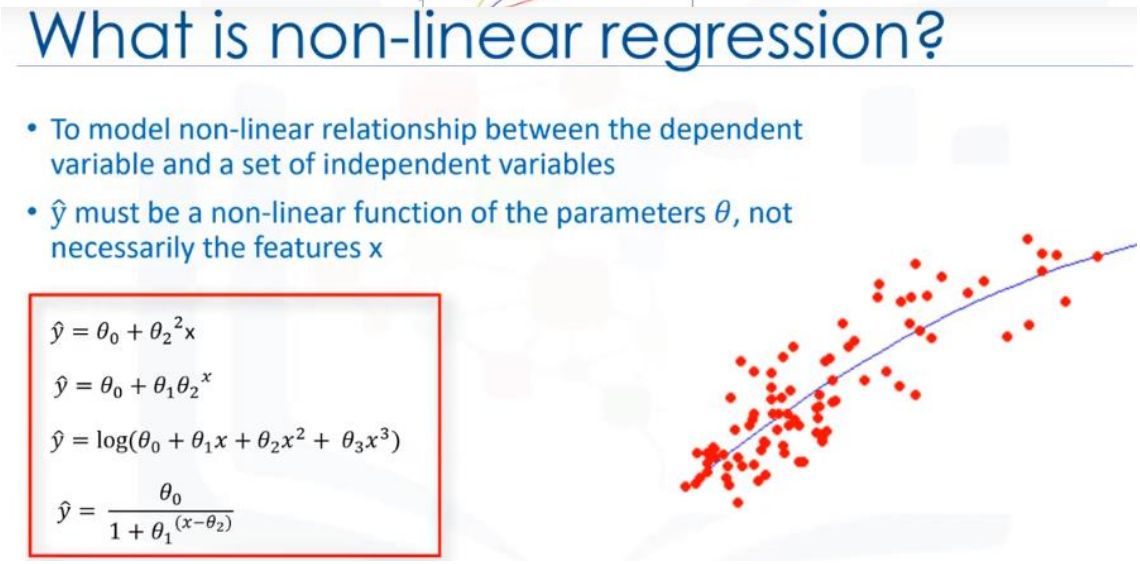
The choice of metric completely depends on the type of model, your data type, and domain of knowledge.

Unfortunately, further review is out of scope of this course.

Thanks for watching!

NON-LIEAR REGRESSION





Hello, and welcome!
In this video, we'll be covering non-linear regression basics.
So let's get started!
These data points correspond to China's Gross Domestic Product (or GDP) from 1960 to 2014.
The first column, is the years, and the second, is China's corresponding annual gross domestic income in US dollars for that year.
This is what the data points look like.
Now, we have a couple of interesting questions.
First, "Can GDP be predicted based on time?"
And second, "Can we use a simple linear regression to model it?"
Indeed, if the data shows a curvy trend, then linear regression will not produce very accurate results when compared to a non-linear regression -- simply because, as the name implies, linear
regression presumes that the data is linear.
The scatterplot shows that there seems to be a strong relationship between GDP and time, but the relationship is not linear.

As you can see, the growth starts off slowly, then from 2005 onward, the growth is very significant.
And finally, it decelerates slightly in the 2010s.
It kind of looks like either a logistical or exponential function.
So, it requires a special estimation method of the non-linear regression procedure.
For example, if we assume that the model for these data points are exponential functions, such as ŷ = θ_0 + θ_1 〖θ_2〗^x, our job is to estimate the parameters of the model, i.e. θs, and use the fitted model to predict GDP for unknown or future cases.
In fact, many different regressions exist that can be used to fit whatever the dataset looks like.
You can see a quadratic and cubic regression lines here, and it can go on and on to infinite degrees.
In essence, we can call all of these "polynomial regression," where the relationship between the independent variable x and the dependent variable y is modelled as an nth degree polynomial in x.
With many types of regression to choose from, there's a good chance that one will fit your dataset well.
Remember, it's important to pick a regression that fits the data the best.
So, what is polynomial Regression?
Polynomial regression fits a curved line to your data.
A simple example of polynomial, with degree 3, is shown as ŷ = θ_0 + θ_1x + θ_2x^2 + θ_3x^3 or to the power of 3, where θs are parameters to be estimated that makes the model fit perfectly to the underlying data.
Though the relationship between x and y is non-linear here, and polynomial regression can fit them, a polynomial regression model can still be expressed as linear regression.
I know it's a bit confusing, but let's look at an example.
Given the 3rd degree polynomial equation, by defining x_1 = x and x_2 = x^2 or x to the power of 2 and so on, the model is converted to a simple linear regression with new variables, as ŷ = θ_0+ θ_1x_1 + θ_2x_2 + θ_3x_3. This model is linear in the parameters to be estimated, right?
Therefore, this polynomial regression is considered to be a special case of traditional multiple linear regression.
So, you can use the same mechanism as linear regression to solve such a problem.
Therefore, polynomial regression models CAN fit using the model of least squares.
Least squares is a method for estimating the unknown parameters in a linear regression model, by minimizing the sum of the squares of the differences between the observed dependent variable in the given dataset and those predicted by the linear function.
So, what is "non-linear regression" exactly?
First, non-linear regression is a method to model a non-linear relationship between the dependent variable and a set of independent variables.
Second, for a model to be considered non-linear, ŷ must be a non-linear function of the parameters θ, not necessarily the features x.
When it comes to non-linear equation, it can be the shape of exponential, logarithmic, and logistic, or many other types.
As you can see, in all of these equations, the change of ŷ depends on changes in the parameters θ, not necessarily on x only.
That is, in non-linear regression, a model is non-linear by parameters.
In contrast to linear regression, we cannot use the ordinary "least squares" method to fit

the data in non-linear regression, and in general, estimation of the parameters is not easy.

Let me answer two important questions here: First, “How can I know if a problem is linear or non-linear in an easy way?”

To answer this question, we have to do two things:

The first is to visually figure out if the relation is linear or non-linear.

It’s best to plot bivariate plots of output variables with each input variable.

Also, you can calculate the correlation coefficient between independent and dependent variables,

and if for all variables it is 0.7 or higher there is a linear tendency, and, thus, it’s not appropriate to fit a non-linear regression.

The second thing we have to do is to use non-linear regression instead of linear regression when

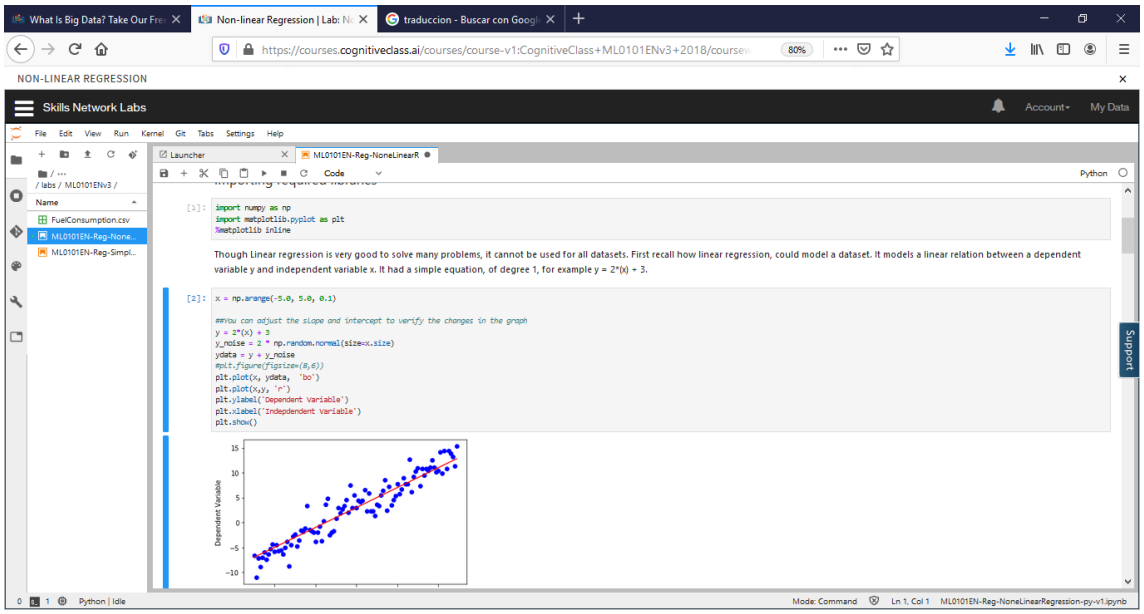
we cannot accurately model the relationship with linear parameters.

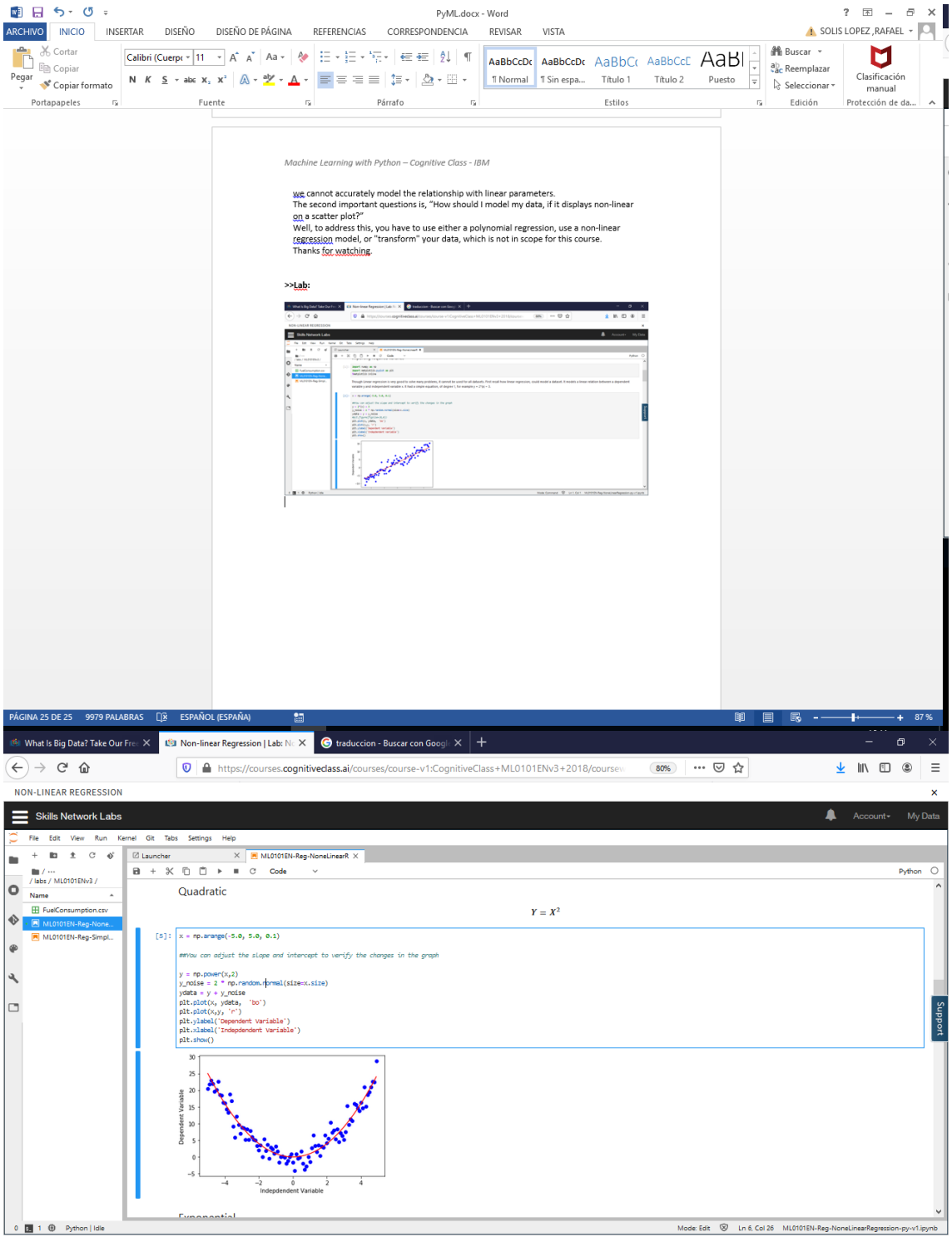
The second important questions is, “How should I model my data, if it displays non-linear on a scatter plot?”

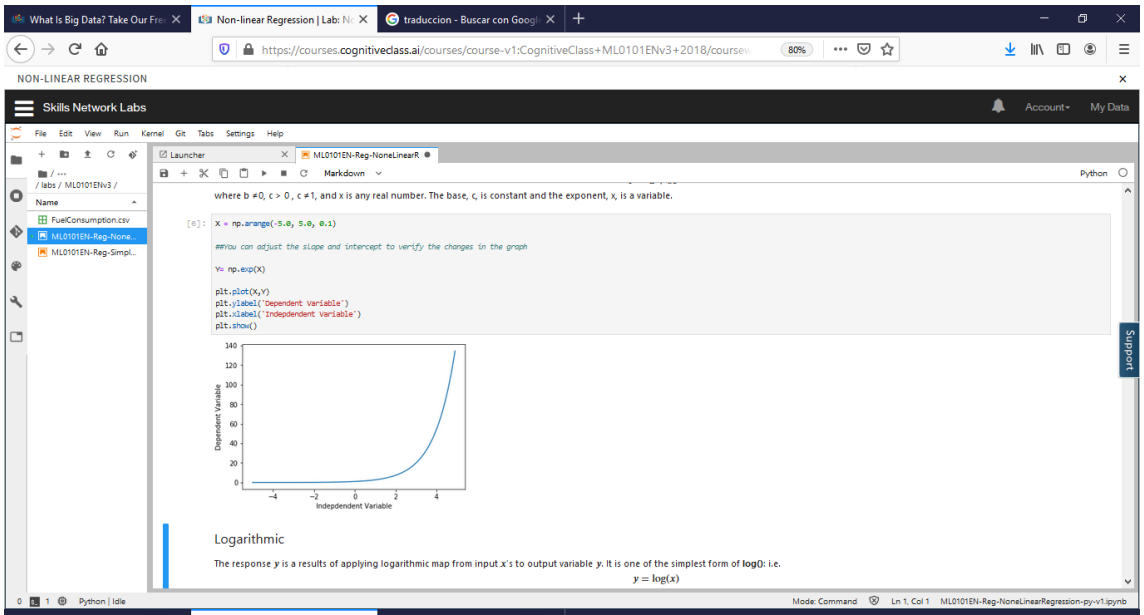
Well, to address this, you have to use either a polynomial regression, use a non-linear regression model, or "transform" your data, which is not in scope for this course.

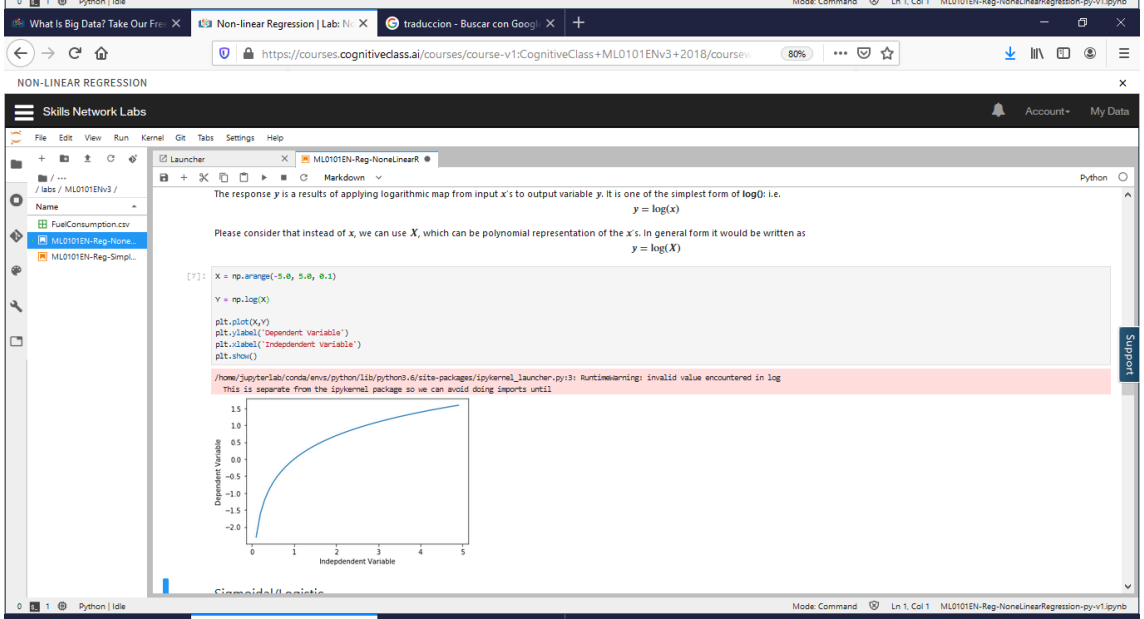
Thanks for watching.

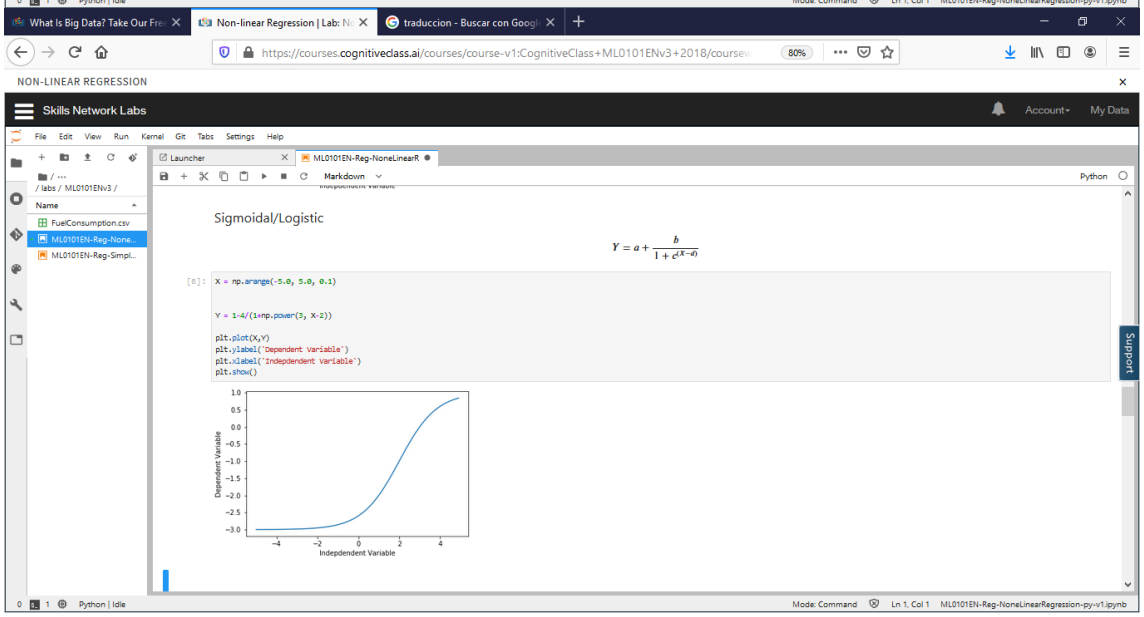
>>Lab:




PyML.docx - Word
Machine Learning with Python – Cognitive Class - IBM
we cannot accurately model the relationship with linear parameters.
The second important questions is, "How should I model my data, if it displays non-linear on a scatter plot?"
Well, to address this, you have to use either a polynomial regression, use a non-linear regression model, or "transform" your data, which is not in scope for this course.
Thanks for watching.
>>Lab:
PÁGINA 25 DE 25 9979 PALABRAS ESPAÑOL (ESPAÑA)
NON-LINEAR REGRESSION
Skills Network Labs
Quadratic
$Y = X^2$
x = np.arange(-5.0, 5.0, 0.1)
##You can adjust the slope and intercept to verify the changes in the graph
y = np.power(x,2)
y_noise = 2 * np.random.normal(size=x.size)
ydata = y + y_noise
plt.plot(x, ydata, 'bo')
plt.plot(x,y, 'r')
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()
Dependent Variable
Independent Variable


NON-LINEAR REGRESSION
Skills Network Labs
Account
My Data
where b ≠0, c > 0 , c ≠1, and x is any real number. The base, c, is constant and the exponent, x, is a variable.
X = np.arange(-5.0, 5.0, 0.1)
##You can adjust the slope and intercept to verify the changes in the graph
Y= np.exp(X)
plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()
Dependent Variable
Indepdendent Variable
Logarithmic
The response y is a results of applying logarithmic map from input x's to output variable y. It is one of the simplest form of log(): i.e.
$y = \log(x)$


NON-LINEAR REGRESSION
Skills Network Labs
Account
My Data
The response y is a results of applying logarithmic map from input x's to output variable y. It is one of the simplest form of log(): i.e.
$y = \log(x)$
Please consider that instead of x, we can use X, which can be polynomial representation of the x's. In general form it would be written as
$y = \log(X)$
X = np.arange(-5.0, 5.0, 0.1)
Y = np.log(X)
plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()
/home/jupyterlab/conda/envs/python/lib/python3.6/site-packages/ipykernel_launcher.py:3: RuntimeWarning: invalid value encountered in log
This is separate from the ipykernel package so we can avoid doing imports until
Dependent Variable
Independent Variable

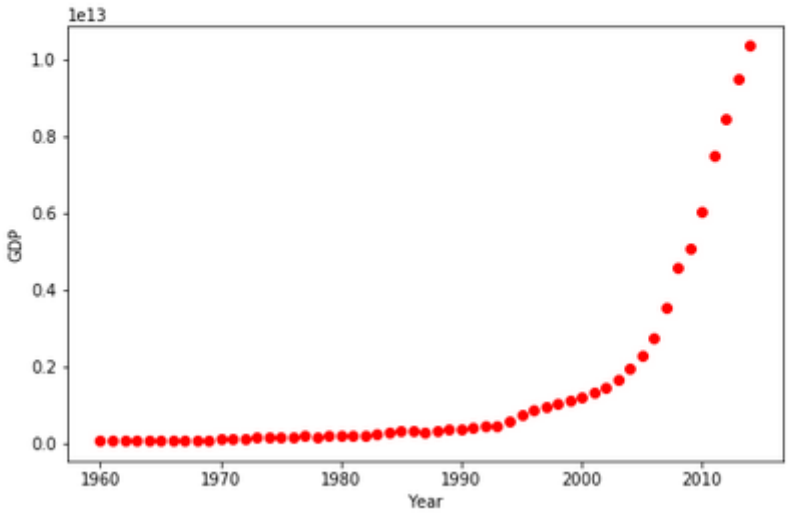

NON-LINEAR REGRESSION
Skills Network Labs
Account
My Data
Sigmoidal/Logistic
$Y = a + \frac{b}{1 + c^{(X-d)}}$
X = np.arange(-5.0, 5.0, 0.1)
Y = 1-4/(1+np.power(3, X-2))
plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Independent Variable')
plt.show()
Dependent Variable
Independent Variable

Non-Linear Regression example

For an example, we're going to try and fit a non-linear model to the datapoints corrensponding to China's GDP from 1960 to 2014. We downl
and 2014, the second, China's corresponding annual gross domestic income in US dollars for that year.

```
import numpy as np
import pandas as pd

#downloading dataset
!wget -nv -O china_gdp.csv https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/china_gdp.csv

df = pd.read_csv("china_gdp.csv")
df.head(10)
```

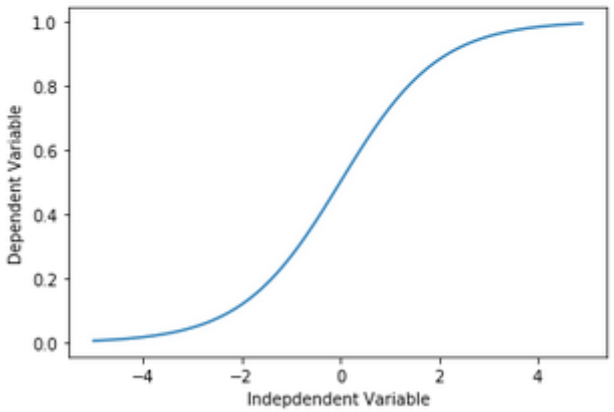
```
[10]: plt.figure(figsize=(8,5))
      x_data, y_data = (df["Year"].values, df["Value"].values)
      plt.plot(x_data, y_data, 'ro')
      plt.ylabel('GDP')
      plt.xlabel('Year')
      plt.show()
```



Choosing a model

From an initial look at the plot, we determine that the logistic
decreasing again at the end; as illustrated below:

```
X = np.arange(-5.0, 5.0, 0.1)
Y = 1.0 / (1.0 + np.exp(-X))

plt.plot(X,Y)
plt.ylabel('Dependent Variable')
plt.xlabel('Indepdendent Variable')
plt.show()
```



The formula for the logistic function is the following:

Building The Model

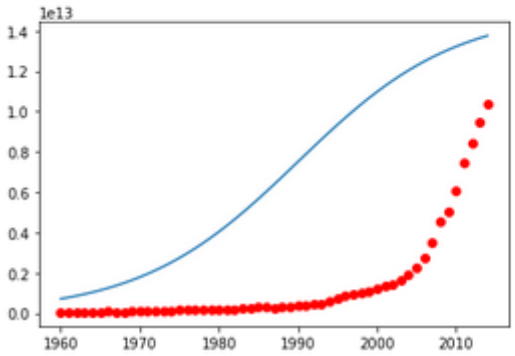
Now, let's build our regression model and initialize its parameters.

```
def sigmoid(x, Beta_1, Beta_2):
     y = 1 / (1 + np.exp(-Beta_1*(x-Beta_2)))
     return y
```

Lets look at a sample sigmoid line that might fit with the data:

```
beta_1 = 0.10
beta_2 = 1990.0

#Logistic function
Y_pred = sigmoid(x_data, beta_1 , beta_2)

#plot initial prediction against datapoints
plt.plot(x_data, Y_pred*15000000000000.)
plt.plot(x_data, y_data, 'ro')
```

```
[<matplotlib.lines.Line2D at 0x7f5aac62ccc0>]
```

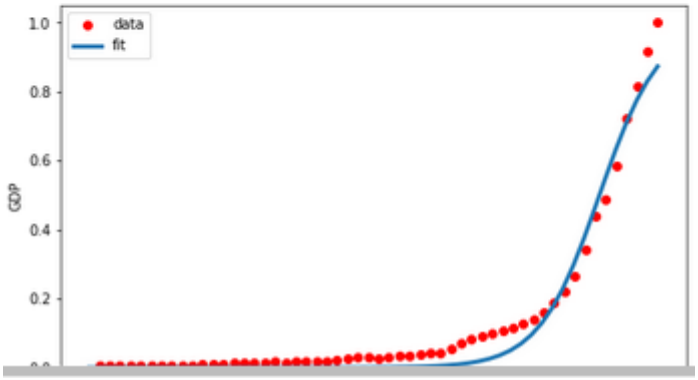


```
from scipy.optimize import curve_fit
popt, pcov = curve_fit(sigmoid, xdata, ydata)
#print the final parameters
print(" beta_1 = %f, beta_2 = %f" % (popt[0], popt[1]))
```

```
beta_1 = 690.447527, beta_2 = 0.997207
```

Now we plot our resulting regresssion model.

```
x = np.linspace(1960, 2015, 55)
x = x/max(x)
plt.figure(figsize=(8,5))
y = sigmoid(x, *popt)
plt.plot(xdata, ydata, 'ro', label='data')
plt.plot(x,y, linewidth=3.0, label='fit')
plt.legend(loc='best')
plt.ylabel('GDP')
plt.xlabel('Year')
plt.show()
```




```
# write your code here

# split data into train/test
msk = np.random.rand(len(df)) < 0.8
train_x = xdata[msk]
test_x = xdata[~msk]
train_y = ydata[msk]
test_y = ydata[~msk]

# build the model using train set
popt, pcov = curve_fit(sigmoid, train_x, train_y)

# predict using test set
y_hat = sigmoid(test_x, *popt)

# evaluation
print("Mean absolute error: %.2f" % np.mean(np.absolute(y_hat - test_y)))
print("Residual sum of squares (MSE): %.2f" % np.mean((y_hat - test_y) ** 2))
from sklearn.metrics import r2_score
print("R2-score: %.2f" % r2_score(y_hat , test_y) )
```

```
Mean absolute error: 0.03
Residual sum of squares (MSE): 0.00
R2-score: 0.98
```

MODULE 3 – CLASSIFICATION

INTRO

Hello!

In this video, we'll give you an introduction to Classification.

So let's get started.

In Machine Learning, classification is a supervised learning approach, which can be thought of as a means of categorizing or "classifying" some unknown items into a discrete set of "classes."

Classification attempts to learn the relationship between a set of feature variables and a target variable of interest.

The target attribute in classification is a categorical variable with discrete values.

So, how does classification and classifiers work?

Given a set of training data points, along with the target labels, classification determines the class label for an unlabeled test case.

Let's explain this with an example.

A good sample of classification is the loan default prediction.

Suppose a bank is concerned about the potential for loans not to be repaid.

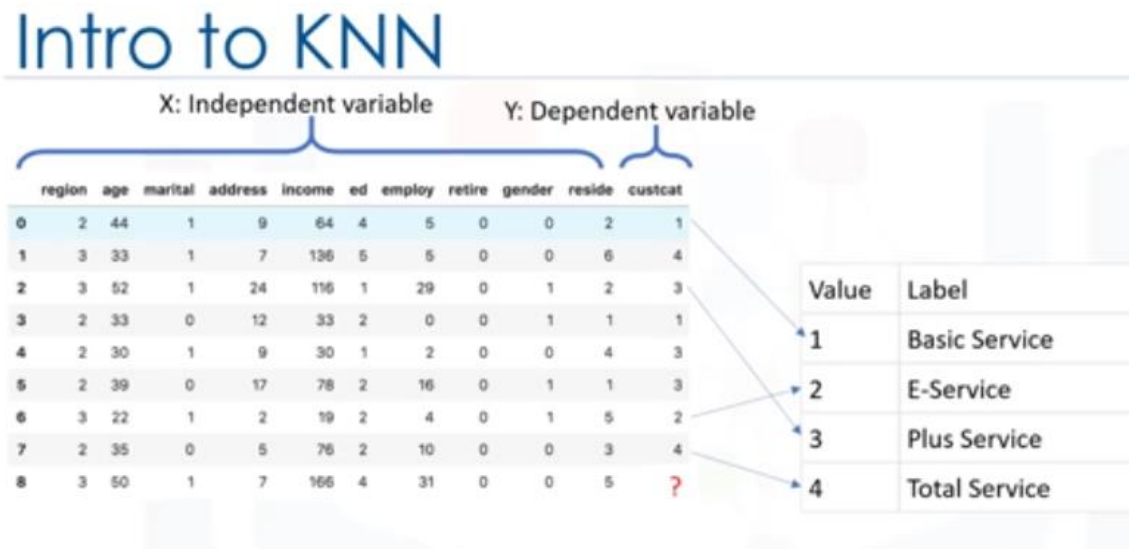
If previous loan default data can be used to predict which customers are likely to have problems repaying loans, these "bad risk" customers can either have their loan application declined or offered alternative products.

The goal of a loan default predictor is to use existing loan default data, which is information about the customers (such as age, income, education, etc.), to build a classifier, pass a new customer or potential future defaulter to the model, and then label it (i.e. the data points) as "Defaulter" or "Not Defaulter", or for example, 0 or 1.

This is how a classifier predicts an unlabeled test case.

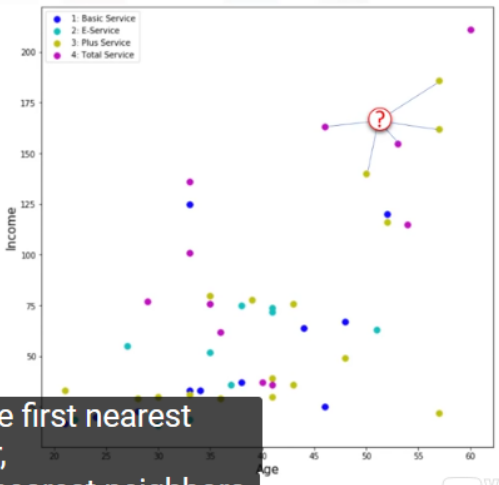
Please notice that this specific example was about a binary classifier with two values.
We can also build classifier models for both binary classification and multi-class classification.
For example, imagine that you collected data about a set of patients, all of whom suffered from the same illness.
During their course of treatment, each patient responded to one of three medications.
You can use this labeled dataset, with a classification algorithm, to build a classification model.
Then you can use it to find out which drug might be appropriate for a future patient with the same illness.
As you can see, it is a sample of multi-class classification.
Classification has different business use cases as well, for example:
To predict the category to which a customer belongs;
For Churn detection, where we predict whether a customer switches to another provider or brand; Or to predict whether or not a customer responds to a particular advertising campaign.
Data classification has several applications in a wide variety of industries.
Essentially, many problems can be expressed as associations between feature and target variables, especially when labeled data is available.
This provides a broad range of applicability for classification.
For example, classification can be used for email filtering, speech recognition, handwriting recognition, bio-metric identification, document classification, and much more.
Here we have the types of classification algorithms in machine learning.
They include: Decision Trees, Naïve Bayes, Linear Discriminant Analysis, K-nearest neighbor, Logistic regression, Neural Networks, and Support Vector Machines.
There are many types of classification algorithms.
We will only cover a few in this course.
Thanks for watching.

K-NEAREST NEIGHBORS



Determining the class using the 5 KNNs

	region	age	marital	address	income	ed	employ	retire	gender	reside	custcat
0	2	44	1	9	64	4	5	0	0	2	1
1	3	33	1	7	136	5	5	0	0	6	4
2	3	52	1	24	116	1	29	0	1	2	3
3	2	33	0	12	33	2	0	0	1	1	1
4	2	30	1	9	30	1	2	0	0	4	3
5	2	39	0	17	78	2	16	0	1	1	3
6	3	22	1	2	19	2	4	0	1	5	2
7	2	35	0	5	76	2	10	0	0	3	4
8	3	50	1	7	166	4	31	0	0	5	?



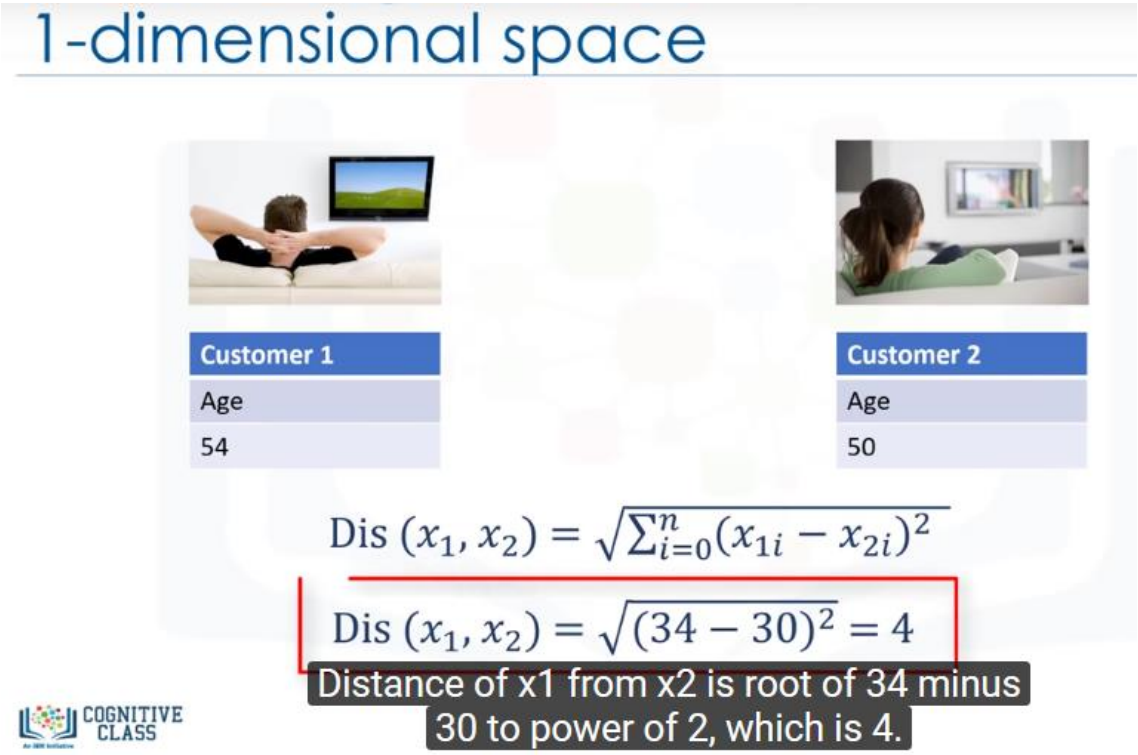
Rather than choose the first nearest neighbor,
what if we chose the five nearest neighbors,

COGNITIVE CLASS

- A method for classifying cases based on their similarity to other cases
- Cases that are near each other are said to be "neighbors"
- Based on similar cases with same class labels are near each other

The K-Nearest Neighbors algorithm

1. Pick a value for K.
2. Calculate the distance of unknown case from all cases.
3. Select the K-observations in the training data that are "nearest" to the unknown data point.
4. Predict the response of the unknown data point using the most popular response value from the K-nearest neighbors.



Hello and welcome!

In this video, we'll be covering the k-nearest neighbors algorithm.

So let's get started.

Imagine that a telecommunications provider has segmented its customer base by service usage patterns, categorizing the customers into four groups.

If demographic data can be used to predict group membership, the company can customize offers for individual prospective customers.

This is a classification problem.

That is, given the dataset, with predefined labels, we need to build a model to be used to predict the class of a new or unknown case.

The example focuses on using demographic data, such as region, age, and marital status, to predict usage patterns.

The target field, called custcat, has four possible values that correspond to the four customer groups, as follows: Basic Service, E-Service, Plus Service, and Total Service.

Our objective is to build a classifier, for example using the rows 0 to 7, to predict the class of row 8.

We will use a specific type of classification called K-nearest neighbor.

Just for sake of demonstration, let's use only two fields as predictors - specifically, Age and Income, and then plot the customers based on their group membership.

Now, let's say that we have a new customer, for example, record number 8 with a known age and income.

How can we find the class of this customer?

Can we find one of the closest cases and assign the same class label to our new customer?

Can we also say that the class of our new customer is most probably group 4 (i.e. total service) because its nearest neighbor is also of class 4?

Yes, we can.

In fact, it is the first-nearest neighbor.

Now, the question is, "To what extent can we trust our judgment, which is based on the first nearest neighbor?"

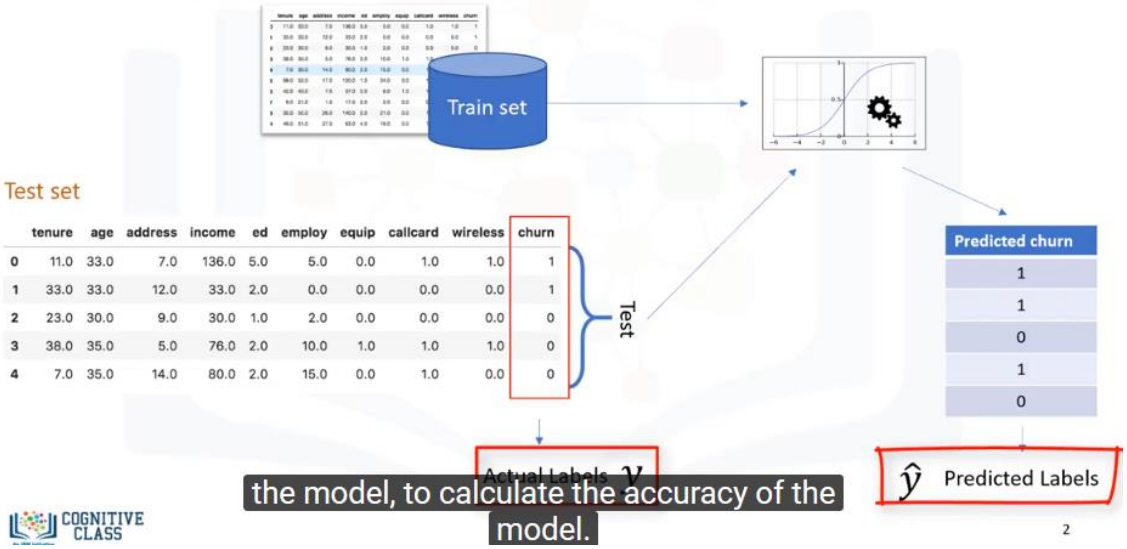
It might be a poor judgment, especially if the first nearest neighbor is a very specific

case, or an outlier, correct?
Now, let's look at our scatter plot again.
Rather than choose the first nearest neighbor, what if we chose the five nearest neighbors, and did a majority vote among them to define the class of our new customer?
In this case, we'd see that three out of five nearest neighbors tell us to go for class 3, which is "Plus service."
Doesn't this make more sense?
Yes, in fact, it does!
In this case, the value of K in the k-nearest neighbors algorithm is 5.
This example highlights the intuition behind the k-nearest neighbors algorithm.
Now, let's define the k-nearest neighbors.
The k-nearest-neighbors algorithm is a classification algorithm that takes a bunch of labelled points and uses them to learn how to label other points.
This algorithm classifies cases based on their similarity to other cases.
In k-nearest neighbors, data points that are near each other are said to be "neighbors."
K-nearest neighbors is based on this paradigm: "Similar cases with the same class labels are near each other."
Thus, the distance between two cases is a measure of their dissimilarity.
There are different ways to calculate the similarity, or conversely, the distance or dissimilarity of two data points.
For example, this can be done using Euclidian distance.
Now, let's see how the k-nearest neighbors algorithm actually works.
In a classification problem, the k-nearest neighbors algorithm works as follows:
1. Pick a value for K.
2. Calculate the distance from the new case (holdout from each of the cases in the dataset).
3. Search for the K observations in the training data that are 'nearest' to the measurements of the unknown data point.
And 4, predict the response of the unknown data point using the most popular response value from the K nearest neighbors.
There are two parts in this algorithm that might be a bit confusing.
First, how to select the correct K; and second, how to compute the similarity between cases, for example, among customers?
Let's first start with second concern, that is, how can we calculate the similarity between two data points?
Assume that we have two customers, customer 1 and customer 2.
And, for a moment, assume that these 2 customers have only one feature, Age.
We can easily use a specific type of Minkowski distance to calculate the distance of these 2 customers.
It is indeed, the Euclidian distance.
Distance of x1 from x2 is root of 34 minus 30 to power of 2, which is 4.
What about if we have more than one feature, for example Age and Income?
If we have income and age for each customer, we can still use the same formula, but this time, we're using it in a 2-dimensional space.
We can also use the same distance matrix for multi-dimensional vectors.
Of course, we have to normalize our feature set to get the accurate dissimilarity measure.
There are other dissimilarity measures as well that can be used for this purpose but, as mentioned, it is highly dependent on data type and also the domain that classification is done for it.
As mentioned, K in k-nearest neighbors, is the number of nearest neighbors to examine.
It is supposed to be specified by the user.

So, how do we choose the right K?
Assume that we want to find the class of the customer noted as question mark on the chart.
What happens if we choose a very low value of K, let's say, k=1?
The first nearest point would be Blue, which is class 1.
This would be a bad prediction, since more of the points around it are Magenta, or class 4.
In fact, since its nearest neighbor is Blue, we can say that we captured the noise in the data, or we chose one of the points that was an anomaly in the data.
A low value of K causes a highly complex model as well, which might result in over-fitting of the model.
It means the prediction process is not generalized enough to be used for out-of-sample cases.
Out-of-sample data is data that is outside of the dataset used to train the model.
In other words, it cannot be trusted to be used for prediction of unknown samples.
It's important to remember that over-fitting is bad, as we want a general model that works for any data, not just the data used for training.
Now, on the opposite side of the spectrum, if we choose a very high value of K, such as K=20, then the model becomes overly generalized.
So, how we can find the best value for K?
The general solution is to reserve a part of your data for testing the accuracy of the model.
Once you've done so, choose k =1, and then use the training part for modeling, and calculate the accuracy of prediction using all samples in your test set.
Repeat this process, increasing the k, and see which k is best for your model.
For example, in our case, k=4 will give us the best accuracy.
Nearest neighbors analysis can also be used to compute values for a continuous target.
In this situation, the average or median target value of the nearest neighbors is used to obtain the predicted value for the new case.
For example, assume that you are predicting the price of a home based on its feature set, such as number of rooms, square footage, the year it was built, and so on.
You can easily find the three nearest neighbor houses, of course -- not only based on distance, but also based on all the attributes, and then predict the price of the house, as the median of neighbors.
This concludes this video. Thanks for watching!

EVALUATION METRICS

Classification accuracy



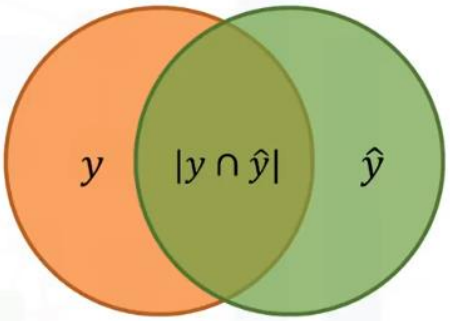
the model, to calculate the accuracy of the model.

Jaccard index

y: Actual labels

$\hat{y}$: Predicted labels

$$J(y,\hat{y}) = \frac{|y \cap \hat{y}|}{|y \cup \hat{y}|} = \frac{|y \cap \hat{y}|}{|y| + |\hat{y}| - |y \cap \hat{y}|}$$



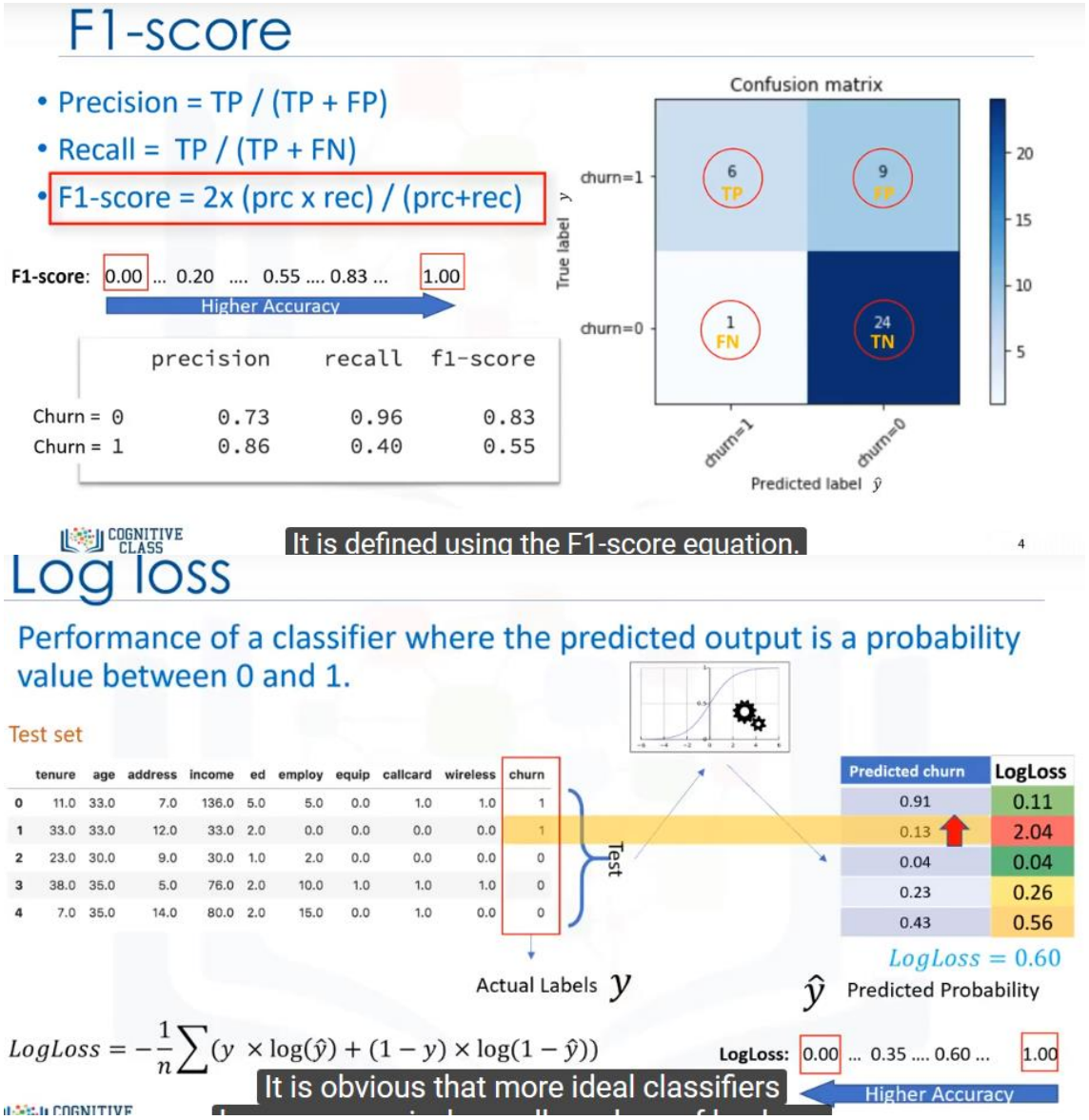
y: [0, 0, 0, 0, 0, 1, 1, 1, ,1 ,1]

$\hat{y}$: [1, 1, 0, 0, 0, 1, 1, 1, ,1 ,1]

$$J(y,\hat{y}) = \frac{8}{10+10-8} = 0.66$$

$J(y,\hat{y}) = 1.0$

If the entire set of predicted labels for a sample strictly matches with the true set



Hello, and welcome!

In this video, we'll be covering evaluation metrics for classifiers.

So let's get started.

Evaluation metrics explain the performance of a model.

Let's talk more about the model evaluation metrics that are used for classification.

Imagine that we have an historical dataset which shows the customer churn for a telecommunication company.

We have trained the model, and now we want to calculate its accuracy using the test set.

We pass the test set to our model, and we find the predicted labels.

Now the question is, "How accurate is this model?"

Basically, we compare the actual values in the test set with the values predicted by the model, to calculate the accuracy of the model.

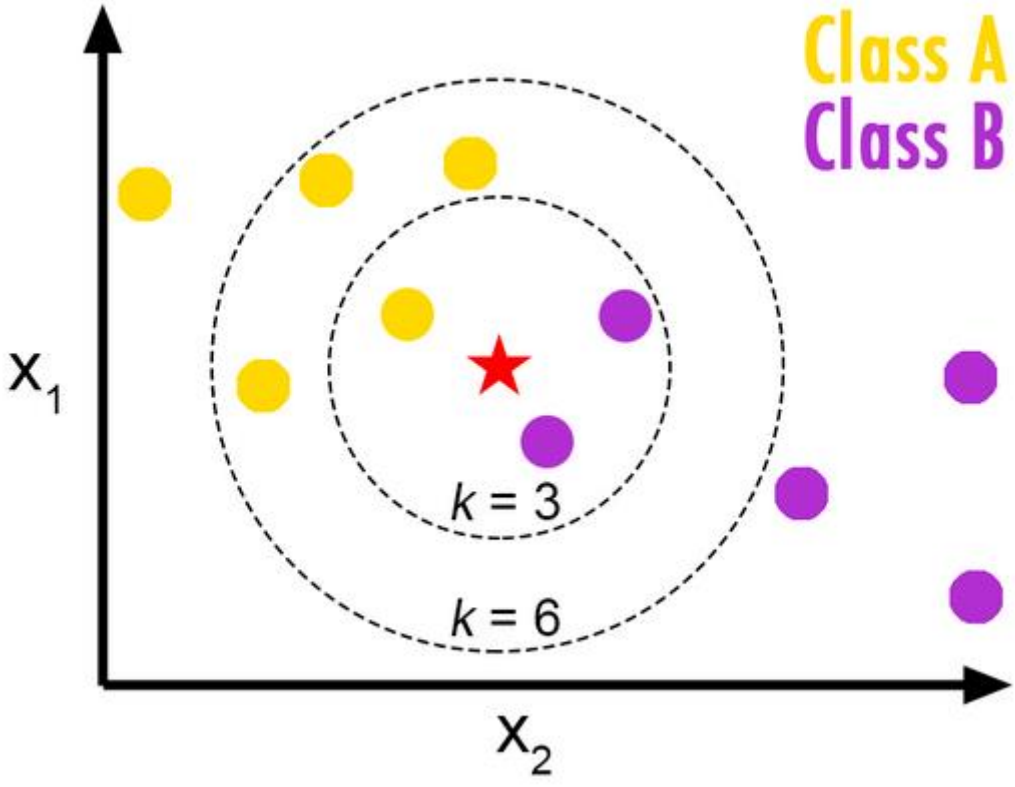
Evaluation metrics provide a key role in the development of a model, as they provide insight to areas that might require improvement.

There are different model evaluation metrics but we just talk about three of them here, specifically: Jaccard index, F1-score, and Log Loss.

Let's first look at one of the simplest accuracy measurements, the Jaccard index -- also known as the Jaccard similarity coefficient.
Let's say y shows the true labels of the churn dataset.
And $\hat{y}$ shows the predicted values by our classifier.
Then we can define Jaccard as the size of the intersection divided by the size of the union of two label sets.
For example, for a test set of size 10, with 8 correct predictions, or 8 intersections, the accuracy by the Jaccard index would be 0.66.
If the entire set of predicted labels for a sample strictly matches with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.
Another way of looking at accuracy of classifiers is to look at a confusion matrix.
For example, let's assume that our test set has only 40 rows.
This matrix shows the corrected and wrong predictions, in comparison with the actual labels.
Each confusion matrix row shows the Actual/True labels in the test set, and the columns show the predicted labels by classifier.
Look at the first row.
The first row is for customers whose actual churn value in the test set is 1.
As you can calculate, out of 40 customers, the churn value of 15 of them is 1.
And out of these 15, the classifier correctly predicted 6 of them as 1, and 9 of them as 0.
This means that for 6 customers, the actual churn value was 1, in the test set, and the classifier also correctly predicted those as 1.
However, while the actual label of 9 customers was 1, the classifier predicted those as 0, which is not very good.
We can consider this as an error of the model for the first row.
What about the customers with a churn value 0?
Let's look at the second row.
It looks like there were 25 customers whose churn value was 0.
The classifier correctly predicted 24 of them as 0, and one of them wrongly predicted as 1.
So, it has done a good job in predicting the customers with a churn value of 0.
A good thing about the confusion matrix is that it shows the model's ability to correctly predict or separate the classes.
In the specific case of a binary classifier, such as this example, we can interpret these numbers as the count of true positives, false positives, true negatives, and false negatives.
Based on the count of each section, we can calculate the precision and recall of each label.
Precision is a measure of the accuracy, provided that a class label has been predicted.
It is defined by: precision = True Positive / (True Positive + False Positive).
And Recall is the true positive rate.
It is defined as: Recall = True Positive / (True Positive + False Negative).
So, we can calculate the precision and recall of each class.
Now we're in the position to calculate the F1 scores for each label, based on the precision and recall of that label.
The F1 score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (which represents perfect precision and recall) and its worst at 0.
It is a good way to show that a classifier has a good value for both recall and precision.
It is defined using the F1-score equation.
For example, the F1-score for class 0 (i.e. churn=0), is 0.83, and the F1-score for class 1 (i.e. churn=1), is 0.55.
And finally, we can tell the average accuracy for this classifier is the average of the F1-score for both labels, which is 0.72 in our case.
Please notice that both Jaccard and F1-score can be used for multi-class classifiers as

well, which is out of scope for this course.
Now let's look at another accuracy metric for classifiers.
Sometimes, the output of a classifier is the probability of a class label, instead of the label.
For example, in logistic regression, the output can be the probability of customer churn, i.e., yes (or equals to 1).
This probability is a value between 0 and 1.
Logarithmic loss (also known as Log loss) measures the performance of a classifier where the predicted output is a probability value between 0 and 1.
So, for example, predicting a probability of 0.13 when the actual label is 1, would be bad and would result in a high log loss.
We can calculate the log loss for each row using the log loss equation, which measures how far each prediction is, from the actual label.
Then, we calculate the average log loss across all rows of the test set.
It is obvious that more ideal classifiers have progressively smaller values of log loss.
So, the classifier with lower log loss has better accuracy.
Thanks for watching!

>>Lab:




```
import itertools
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import NullFormatter
import pandas as pd
import numpy as np
import matplotlib.ticker as ticker
from sklearn import preprocessing
%matplotlib inline
```

About dataset

Imagine a telecommunications provider has segmented its customer base by service usage patterns, categorizing the customers into four groups. If demographic data can be used to predict group membership, the company can customize offers for individual prospective customers. It is a classification problem. That is, given the dataset, with predefined labels, we need to build a model to be used to predict class of a new or unknown case.

The example focuses on using demographic data, such as region, age, and marital, to predict usage patterns.

The target field, called **custcat**, has four possible values that correspond to the four customer groups, as follows: 1- Basic Service 2- E-Service 3- Plus Service 4- Total Service

Our objective is to build a classifier, to predict the class of unknown cases. We will use a specific type of classification called K nearest neighbour.

Load Data From CSV File

```
df = pd.read_csv('teleCust1000t.csv')
df.head()
```

	region	tenure	age	marital	address	income	ed	employ	retire	gender	reside	custcat
0	2	13	44	1	9	64.0	4	5	0.0	0	2	1
1	3	11	33	1	7	136.0	5	5	0.0	0	6	4
2	3	68	52	1	24	116.0	1	29	0.0	1	2	3
3	2	33	33	0	12	33.0	2	0	0.0	1	1	1
4	2	23	30	1	9	30.0	1	2	0.0	0	4	3

Data Visualization and Anylisis

Let's see how many of each class is in our data set

```
df['custcat'].value_counts()
```

```
3    281
1    266
4    236
```

Data Visualization and Anylisis

Let's see how many of each class is in our data set

```
df['custcat'].value_counts()
```

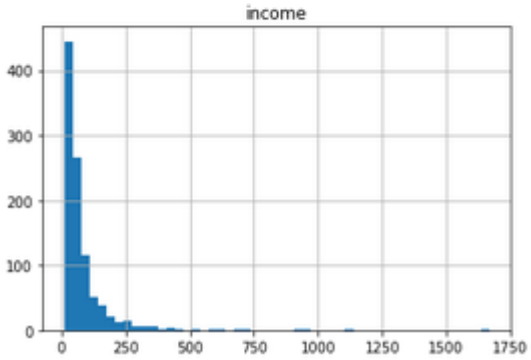
```
3    281
1    266
4    236
2    217
Name: custcat, dtype: int64
```

281 Plus Service, 266 Basic-service, 236 Total Service, and 217 E-Service customers

You can easily explore your data using visualization techniques:

```
df.hist(column='income', bins=50)
```

```
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x7f19107b22b0>]],
      dtype=object)
```



Normalize Data

Did you know? IBM Watson Studio lets you build and deploy an
in. Learn more here.

Data Standardization give data zero mean and unit variance, it is good practice, especially f

```
X = preprocessing.StandardScaler().fit(X).transform(X.astype(float))
X[0:5]
```

```
array([[-0.02696767, -1.055125  ,  0.18450456,  1.0100505 , -0.25303431,
        -0.12650641,  1.0877526 , -0.5941226 , -0.22207644, -1.03459817,
        -0.23065004],
       [ 1.19883553, -1.14880563, -0.69181243,  1.0100505 , -0.4514148 ,
         0.54644972,  1.9062271 , -0.5941226 , -0.22207644, -1.03459817,
         2.55666158],
       [ 1.19883553,  1.52109247,  0.82182601,  1.0100505 ,  1.23481934,
         0.35951747, -1.36767088,  1.78752803, -0.22207644,  0.96655883,
        -0.23065004],
       [-0.02696767, -0.11831864, -0.69181243, -0.9900495 ,  0.04453642,
        -0.41625141, -0.54919639, -1.09029981, -0.22207644,  0.96655883,
        -0.92747794],
       [-0.02696767, -0.58672182, -0.93080797,  1.0100505 , -0.25303431,
        -0.44429125, -1.36767088, -0.89182893, -0.22207644, -1.03459817,
         1.16300577]])
```

Train Test Split

Out of Sample Accuracy is the percentage of correct predictions that the model makes on data that tha
likelihood of being over-fit.

It is important that our models have a high, out-of-sample accuracy, because the purpose of any mode
approach called Train/Test Split. Train/Test Split involves splitting the dataset into training and testing sets

This will provide a more accurate evaluation on out-of-sample accuracy because the testing dataset is n

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (800, 11) (800,)
Test set: (200, 11) (200,)
```

K nearest neighbor (K-NN)

Import library

Classifier implementing the k-nearest neighbors vote.

```
]: from sklearn.neighbors import KNeighborsClassifier
```

Training

Lets start the algorithm with k=4 for now:

```
]: k = 4
#Train Model and Predict
neigh = KNeighborsClassifier(n_neighbors = k).fit(X_train,y_train)
neigh
```

```
]: KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                        metric_params=None, n_jobs=None, n_neighbors=4, p=2,
                        weights='uniform')
```

Predicting

we can use the model to predict the test set:

```
]: yhat = neigh.predict(X_test)
yhat[0:5]
```

```
]: array([1, 1, 3, 2, 4])
```

Accuracy evaluation

In multilabel classification, **accuracy classification score** function computes subset accuracy. This function is eq

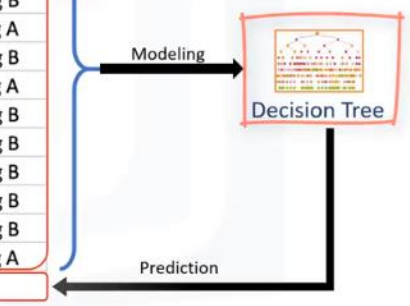
```
from sklearn import metrics
print("Train set Accuracy: ", metrics.accuracy_score(y_train, neigh.predict(X_train)))
print("Test set Accuracy: ", metrics.accuracy_score(y_test, yhat))
```

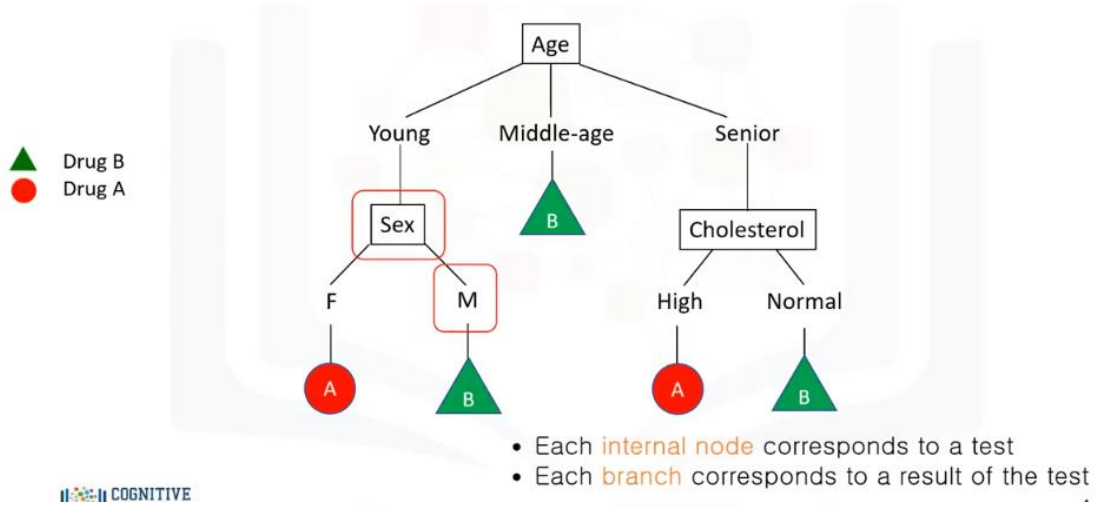
```
Train set Accuracy:  0.5475
Test set Accuracy:  0.32
```

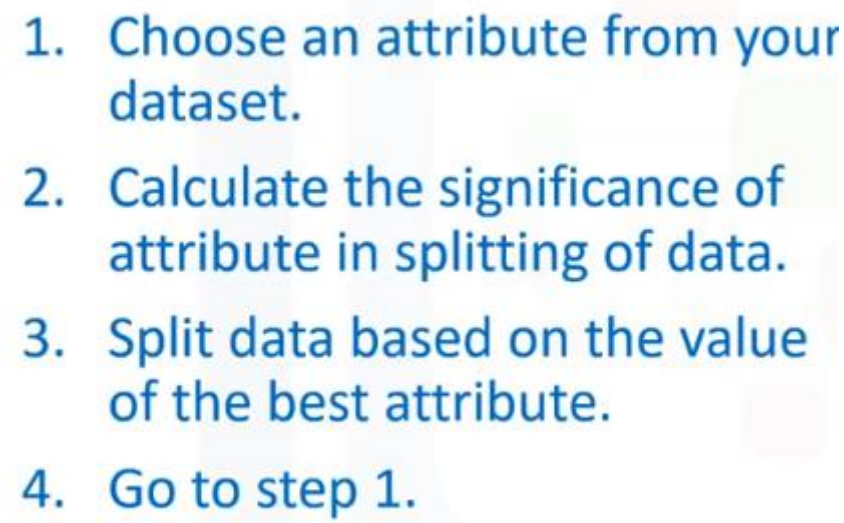
DECISION TREES

How to build a decision tree?

Patient ID	Age	Sex	BP	Cholesterol	Drug
p1	Young	F	High	Normal	Drug A
p2	Young	F	High	High	Drug A
p3	Middle-age	F	Hiigh	Normal	Drug B
p4	Senior	F	Normal	Normal	Drug B
p5	Senior	M	Low	Normal	Drug B
p6	Senior	M	Low	High	Drug A
p7	Middle-age	M	Low	High	Drug B
p8	Young	F	Normal	Normal	Drug A
p9	Young	M	Low	Normal	Drug B
p10	Senior	M	Normal	Normal	Drug B
p11	Young	M	Normal	High	Drug B
p12	Middle-age	F	Normal	High	Drug B
p13	Middle-age	M	High	Normal	Drug B
p14	Senior	F	Normal	High	Drug A
p15	Middle-age	F	Low	Normal	?

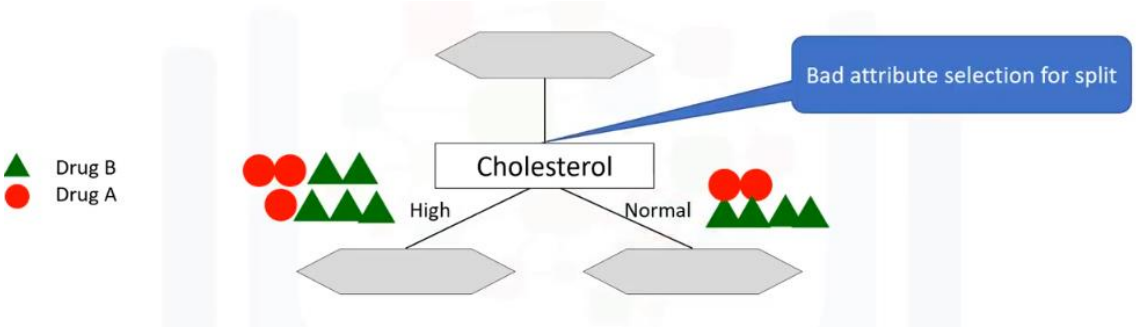


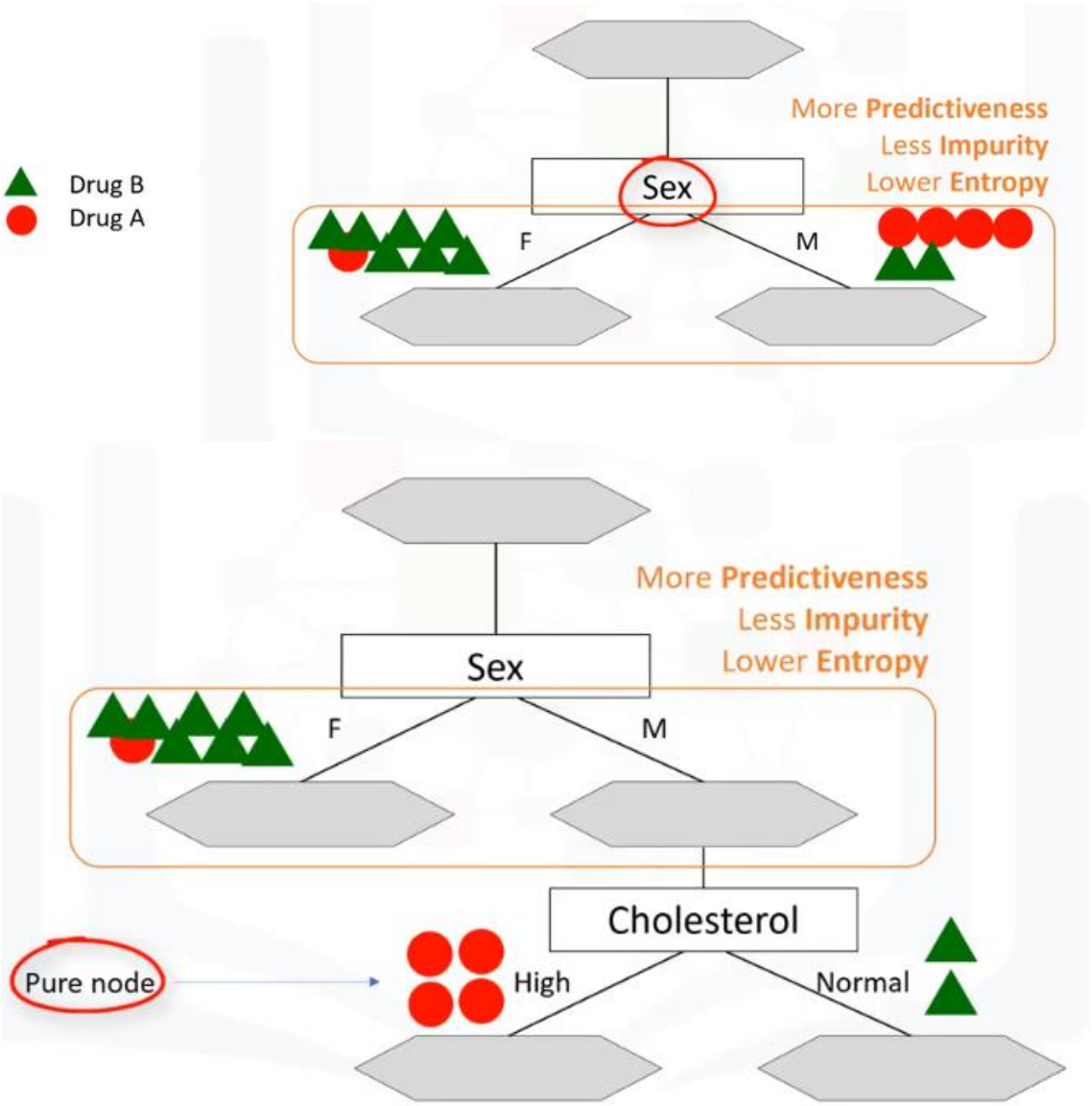




Hello, and welcome!
In this video, we're going to introduce and examine decision trees.
So let's get started
What exactly is a decision tree?
How do we use them to help us classify?
How can I grow my own decision tree?
These may be some of the questions that you have in mind from hearing the term, decision tree.
Hopefully, you'll soon be able to answer these questions, and many more, by watching this video!
Imagine that you're a medical researcher compiling data for a study.
You've already collected data about a set of patients, all of whom suffered from the same illness.
During their course of treatment, each patient responded to one of two medications; we'll call them Drug A and Drug B. Part of your job is to build a model to find out which drug might be appropriate for a future patient with the same illness.
The feature sets of this dataset are Age, Gender, Blood Pressure, and Cholesterol of our group of patients, and the target is the drug that each patient responded to.

It is a sample of binary classifiers, and you can use the training part of the dataset
to build a decision tree, and then, use it to predict the class of an unknown patient
… in essence, to come up with a decision on which drug to prescribe to a new patient.
Let's see how a decision tree is built for this dataset.
Decision trees are built by splitting the training set into distinct nodes, where one
node contains all of, or most of, one category of the data.
If we look at the diagram here, we can see that it's a patient classifier.
So, as mentioned, we want to prescribe a drug to a new patient, but the decision to choose
drug A or B, will be influenced by the patient's situation.
We start with the Age, which can be Young, Middle-aged, or Senior.
If the patient is Middle-aged, then we'll definitely go for Drug B.
On the other hand, if he is a Young or a Senior patient, we'll need more details to help
us determine which drug to prescribe.
The additional decision variables can be things such as Cholesterol levels, Gender or Blood
Pressure.
For example, if the patient is Female, then we will recommend Drug A, but if the patient
is Male, then we'll go for Drug B. As you can see, decision trees are about testing
an attribute and branching the cases, based on the result of the test.
Each internal node corresponds to a test.
And each branch corresponds to a result of the test.
And each leaf node assigns a patient to a class.
Now the question is how can we build such a decision tree?
Here is the way that a decision tree is built.
A decision tree can be constructed by considering the attributes one by one.
First, choose an attribute from our dataset.
Calculate the significance of the attribute in the splitting of the data.
In the next video, we will explain how to calculate the significance of an attribute,
to see if it's an effective attribute or not.
Next, split the data based on the value of the best attribute.
Then, go to each branch and repeat it for the rest of the attributes.
After building this tree, you can use it to predict the class of unknown cases or, in
our case, the proper Drug for a new patient based on his/her characterestics.
This concludes this video.
Thanks for watching!



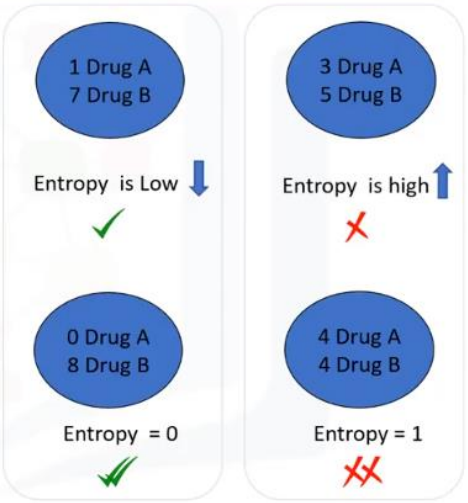


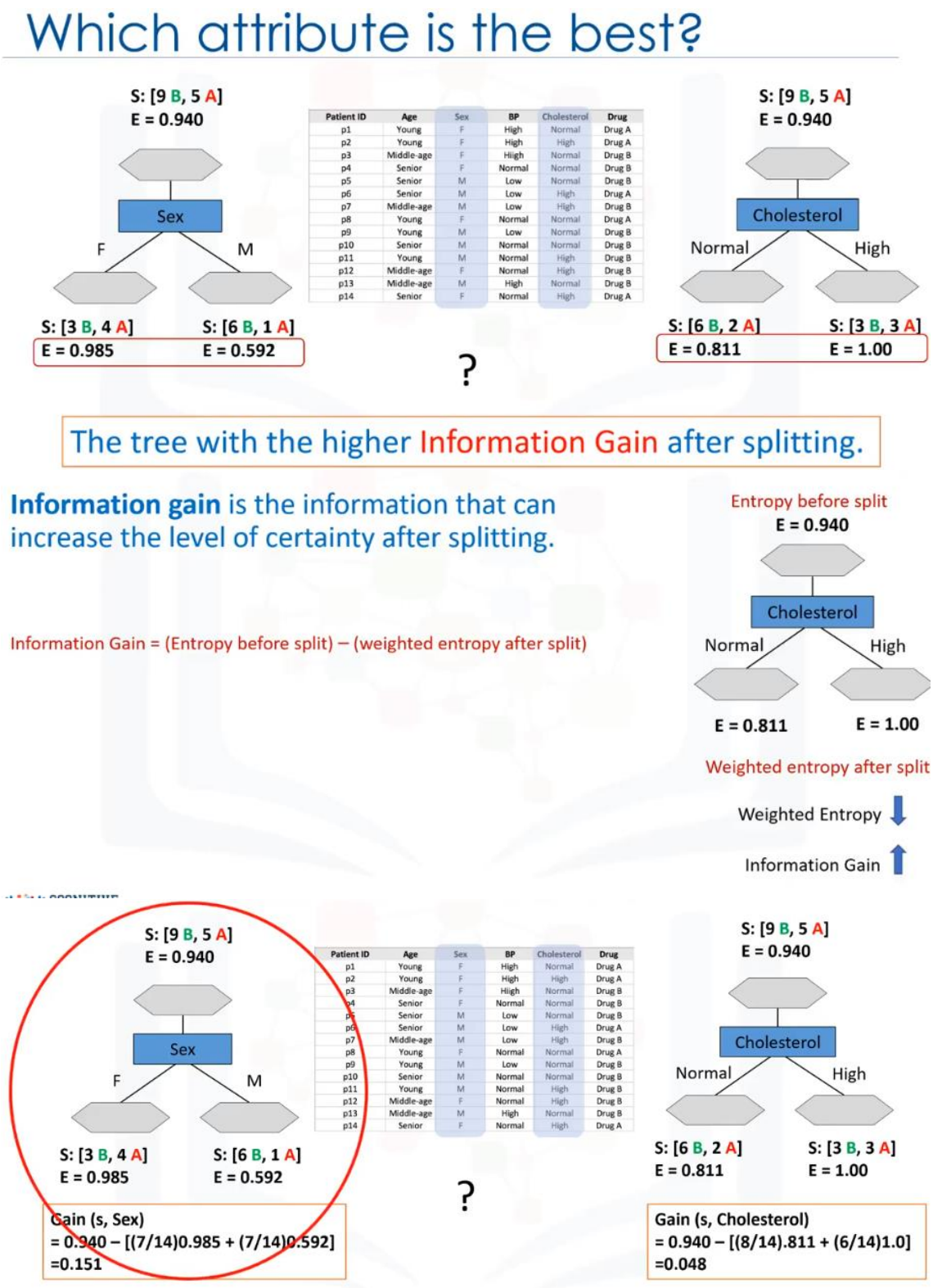
Entropy

- Measure of randomness or uncertainty

Entropy = - p(A)log(p(A)) - p(B)log(p(B))

The lower the Entropy, the less uniform the distribution, the purer the node.





Hello, and welcome!
In this video, we'll be covering the process of building decision trees.
So let's get started!
Consider the drug dataset again.
The question is, "How do we build the decision tree based on that dataset?"
Decision trees are built using recursive partitioning to classify the data.

Let's say we have 14 patients in our dataset.
The algorithm chooses the most predictive feature to split the data on.
What is important in making a decision tree, is to determine "which attribute is the best, or more predictive, to split data based on the feature."
Let's say we pick "Cholesterol" as the first attribute to split data.
It will split our data into 2 branches.
As you can see, if the patient has high "Cholesterol," we cannot say with high confidence that Drug B
might be suitable for him.
Also, if the Patient's "Cholesterol" is normal, we still don't have sufficient evidence or information to determine if either Drug A or Drug B is, in fact, suitable.
It is a sample of bad attribute selection for splitting data.
So, let's try another attribute.
Again, we have our 14 cases.
This time, we pick the "sex" attribute of patients.
It will split our data into 2 branches, Male and Female.
As you can see, if the patient is Female, we can say Drug B might be suitable for her with high certainty.
But, if the patient is Male, we don't have sufficient evidence or information to determine if Drug A or Drug B is suitable.
However, it is still a better choice in comparison with the "Cholesterol" attribute, because the result in the nodes are more pure.
It means, nodes that are either mostly Drug A or Drug B.
So, we can say the "Sex" attribute is more significant than "Cholesterol," or in other words, it's more predictive than the other attributes.
Indeed, "predictiveness" is based on decrease in "impurity" of nodes.
We're looking for the best feature to decrease the "impurity" of patients in the leaves, after splitting them up based on that feature.
So, the "Sex" feature is a good candidate in the following case, because it almost found the pure patients.
Let's go one step further.
For the Male patient branch, we again test other attributes to split the subtree.
We test "Cholesterol" again here.
As you can see, it results in even more pure leaves.
So, we can easily make a decision here.
For example, if a patient is "Male", and his "Cholesterol" is "High", we can certainly prescribe Drug A, but if it is "Normal", we can prescribe Drug B with high confidence.
As you might notice, the choice of attribute to split data is very important, and it is all about "purity" of the leaves after the split.
A node in the tree is considered "pure" if, in 100% of the cases, the nodes fall into a specific category of the target field.
In fact, the method uses recursive partitioning to split the training records into segments by minimizing the "impurity" at each step.
"Impurity" of nodes is calculated by "Entropy" of data in the node.
So, what is "Entropy"?
Entropy is the amount of information disorder, or the amount of randomness in the data.
The entropy in the node depends on how much random data is in that node and is calculated for each node.
In decision trees, we're looking for trees that have the smallest entropy in their nodes.
The entropy is used to calculate the homogeneity of the samples in that node.
If the samples are completely homogeneous the entropy is zero and if the samples are equally divided, it has an entropy of one.

This means, if all the data in a node are either Drug A or Drug B, then the entropy is zero, but if half of the data are Drug A and other half are B, then the entropy is one.
You can easily calculate the entropy of a node using the frequency table of the attribute through the Entropy formula, where P is for the proportion or ratio of a category, such as Drug A or B. Please remember, though, that you don't have to calculate these, as it's easily calculated by the libraries or packages that you use.
As an example, let's calculate the entropy of the dataset before splitting it.
We have 9 occurrences of Drug B and 5 of Drug A.
You can embed these numbers into the Entropy formula to calculate the impurity of the target attribute before splitting it.
In this case, it is 0.94.
So, what is entropy after splitting?
Now we can test different attributes to find the one with the most "predictiveness," which results in two more pure branches.
Let's first select the "Cholesterol" of the patient and see how the data gets split, based on its values.
For example, when it is "normal," we have 6 for Drug B, and 2 for Drug A.
We can calculate the Entropy of this node based on the distribution of drug A and B, which is 0.8 in this case.
But, when Cholesterol is "High," the data is split into 3 for drug B and 3 for drug A.
Calculating its entropy, we can see it would be 1.0.
We should go through all the attributes and calculate the "Entropy" after the split, and then chose the best attribute.
Ok, let's try another field.
Let's choose the Sex attribute for the next check.
As you can see, when we use the Sex attribute to split the data, when its value is "Female," we have 3 patients that responded to Drug B, and 4 patients that responded to Drug A.
The entropy for this node is 0.98 which is not very promising.
However, on other side of the branch, when the value of the Sex attribute is Male, the result is more pure with 6 for Drug B and only 1 for Drug A.
The entropy for this group is 0.59.
Now, the question is, between the Cholesterol and Sex attributes, which one is a better choice?
Which one is better as the first attribute to divide the dataset into 2 branches?
Or, in other words, which attribute results in more pure nodes for our drugs?
Or, in which tree, do we have less entropy after splitting rather than before splitting?
The "Sex" attribute with entropy of 0.98 and 0.59, or the "Cholesterol" attribute with entropy of 0.81 and 1.0 in its branches?
The answer is, "The tree with the higher information gain after splitting."
So, what is information gain?
Information gain is the information that can increase the level of certainty after splitting.
It is the entropy of a tree before the split minus the weighted entropy after the split by an attribute.
We can think of information gain and entropy as opposites.
As entropy, or the amount of randomness, decreases, the information gain, or amount of certainty, increases, and vice-versa.
So, constructing a decision tree is all about finding attributes that return the highest information gain.
Let's see how "information gain" is calculated for the Sex attribute.

As mentioned, the information gain is the entropy of the tree before the split, minus the weighted entropy after the split.
The entropy of the tree before the split is 0.94.
The portion of Female patients is 7 out of 14, and its entropy is 0.985.
Also, the portion of men is 7 out of 14, and the entropy of the Male node is 0.592.
The result of a square bracket here is the weighted entropy after the split.
So, the information gain of the tree if we use the "Sex" attribute to split the dataset is 0.151.
As you can see, we will consider the entropy over the distribution of samples falling under each leaf node, and we'll take a weighted average of that entropy – weighted by the proportion of samples falling under that leaf.
We can calculate the information gain of the tree if we use "Cholesterol" as well.
It is 0.48.
Now, the question is, "Which attribute is more suitable?"
Well, as mentioned, the tree with the higher information gain after splitting.
This means the "Sex" attribute.
So, we select the "Sex" attribute as the first splitter.
Now, what is the next attribute after branching by the "Sex" attribute?
Well, as you can guess, we should repeat the process for each branch, and test each of the other attributes to continue to reach the most pure leaves.
This is the way that you build a decision tree!
Thanks for watching!

>>Lab:

Import the Following Libraries:

- **numpy (as np)**
- **pandas**
- **DecisionTreeClassifier** from **sklearn.tree**

```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
```

now, read data using pandas dataframe:

```
my_data = pd.read_csv("drug200.csv", delimiter=",")
my_data[0:5]
```

	Age	Sex	BP	Cholesterol	Na_to_K	Drug
0	23	F	HIGH	HIGH	25.355	drugY
1	47	M	LOW	HIGH	13.093	drugC
2	47	M	LOW	HIGH	10.114	drugC
3	28	F	NORMAL	HIGH	7.798	drugX
4	61	F	LOW	HIGH	18.043	drugY

Pre-processing

Using **my_data** as the Drug.csv data read by pandas, declare the following variables:

- **X** as the **Feature Matrix** (data of my_data)
   y as the response vector (target)

Remove the column containing the target name since it doesn't contain numeric values.

```
X = my_data[['Age', 'Sex', 'BP', 'Cholesterol', 'Na_to_K']].values
X[0:5]
```

```
array([[23, 'F', 'HIGH', 'HIGH', 25.355],
       [47, 'M', 'LOW', 'HIGH', 13.093],
       [47, 'M', 'LOW', 'HIGH', 10.113999999999999],
       [28, 'F', 'NORMAL', 'HIGH', 7.797999999999999],
       [61, 'F', 'LOW', 'HIGH', 18.043]], dtype=object)
```

As you may figure out, some featurs in this dataset are catergorical such as **Sex** or **BP**. Unfortunately, Sklearn Decision Trees do not handle categorical variables. But still we can convert these features to numerical values. **pandas.get_dummies()** Convert categorical variable into dummy/indicator variables.

As you may figure out, some featurs in this dataset are catergorical such as **Sex** or **BP**. Unfortunately, Sklearn Decision Trees do not handle categorical variables. But still we can convert these features to numerical values. **pandas.get_dummies()** Convert categorical variable into dummy/indicator variables.

```
from sklearn import preprocessing
le_sex = preprocessing.LabelEncoder()
le_sex.fit(['F','M'])
X[:,1] = le_sex.transform(X[:,1])


le_BP = preprocessing.LabelEncoder()
le_BP.fit([ 'LOW', 'NORMAL', 'HIGH'])
X[:,2] = le_BP.transform(X[:,2])


le_Chol = preprocessing.LabelEncoder()
le_Chol.fit([ 'NORMAL', 'HIGH'])
X[:,3] = le_Chol.transform(X[:,3])

X[0:5]
```

```
array([[23, 0, 0, 0, 25.355],
       [47, 1, 1, 0, 13.093],
       [47, 1, 1, 0, 10.113999999999999],
```

Setting up the Decision Tree

We will be using **train/test split** on our **decision tree**. Let's import **train_test_split** from **sklearn.cross_validation**.

```
from sklearn.model_selection import train_test_split
```

Now **train_test_split** will return 4 different parameters. We will name them:
X_trainset, X_testset, y_trainset, y_testset

The **train_test_split** will need the parameters:
X, y, test_size=0.3, and random_state=3.

The **X** and **y** are the arrays required before the split, the **test_size** represents the ratio of the testing dataset, and the **random_state** ensures that we obtain the same splits.

```
X_trainset, X_testset, y_trainset, y_testset = train_test_split(X, y, test_size=0.3, random_state=3)
```

Modeling

We will first create an instance of the **DecisionTreeClassifier** called **drugTree**.
Inside of the classifier, specify *criterion="entropy"* so we can see the information gain of each node.

```
drugTree = DecisionTreeClassifier(criterion="entropy", max_depth = 4)
drugTree # it shows the default parameters
```

```
DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=4,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Next, we will fit the data with the training feature matrix **X_trainset** and training response vector **y_trainset**

```
drugTree.fit(X_trainset,y_trainset)
```

```
DecisionTreeClassifier(class_weight=None, criterion='entropy', max_depth=4,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

Prediction ¶

Let's make some **predictions** on the testing dataset and store it into a variable called **predTree**.

```
predTree = drugTree.predict(X_testset)
```

You can print out **predTree** and **y_testset** if you want to visually compare the prediction to the actual values.

```
print (predTree [0:5])
print (y_testset [0:5])
```

```
['drugY' 'drugX' 'drugX' 'drugX' 'drugX']
40     drugY
51     drugX
139    drugX
197    drugX
170    drugX
Name: Drug, dtype: object
```

Evaluation

Next, let's import **metrics** from sklearn and check the accuracy of our model.

```
from sklearn import metrics
import matplotlib.pyplot as plt
print("DecisionTrees's Accuracy: ", metrics.accuracy_score(y_testset, predTree))
```

```
DecisionTrees's Accuracy:  0.9833333333333333
```

Accuracy classification score computes subset accuracy: the set of labels predicted for a sample must exactly match the corresponding set of labels in y_true.

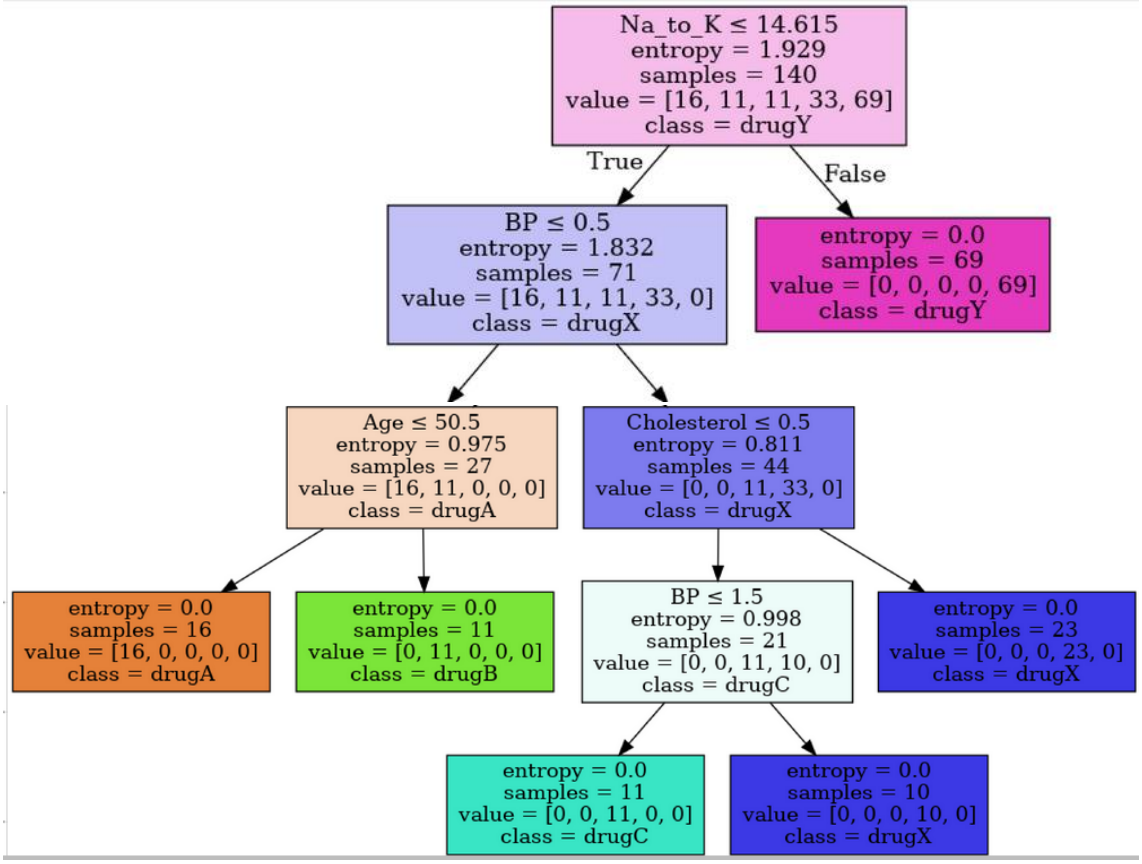
In multilabel classification, the function returns the subset accuracy. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.

Visualization

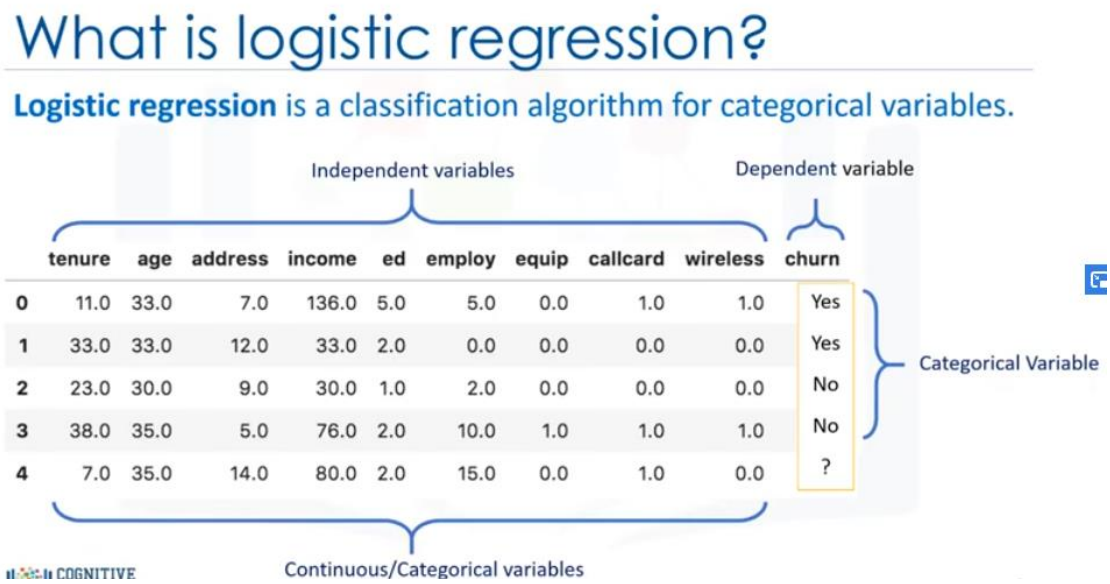
Lets visualize the tree

```
from sklearn.externals.six import StringIO
import pydotplus
import matplotlib.image as mpimg
from sklearn import tree
%matplotlib inline
```

```
dot_data = StringIO()
filename = "drugtree.png"
featureNames = my_data.columns[0:5]
targetNames = my_data["Drug"].unique().tolist()
out=tree.export_graphviz(drugTree,feature_names=featureNames, out_file=dot_data, class_names= np.unique(y_trainset), filled=True,  special_characters
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
graph.write_png(filename)
img = mpimg.imread(filename)
plt.figure(figsize=(100, 200))
plt.imshow(img,interpolation='nearest')
```



LOGISTIC REGRESSION

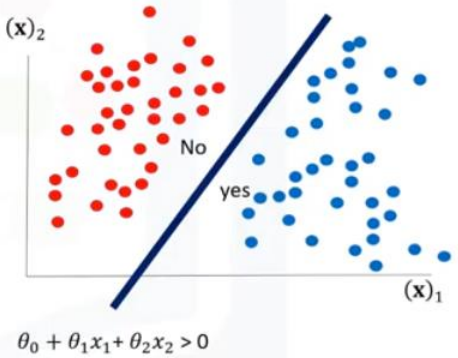


Logistic regression applications

- Predicting the probability of a person having a heart attack
- Predicting the mortality in injured patients
- Predicting a customer's propensity to purchase a product or halt a subscription
- Predicting the probability of failure of a given process or product
- Predicting the likelihood of a homeowner defaulting on a mortgage

When is logistic regression suitable?

- If your data is binary
  - 0/1, YES/NO, True/False
- If you need probabilistic results
- When you need a linear decision boundary



Building a model for customer churn

$\mathbf{x}$ y

	tenure	age	address	income	ed	employ	equip	callcard	wireless	churn
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	1.0
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	1.0
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	0.0
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	0.0

$X \in \mathbb{R}^{m \times n}$

$y \in \{0,1\}$

$\hat{y} = \text{P(y=1|x)}$

P(y=0|x) = 1- P(y=1|x)

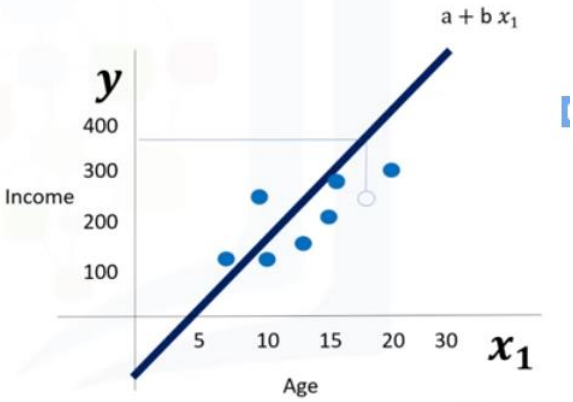
Hello, and welcome!
In this video we'll learn a machine learning method, called logistic regression, which is used for classification.
In examining this method, we'll specifically answer these three questions:
- What is logistic regression?
- What kind of problems can be solved by logistic regression?
- And, in which situations do we use logistic regression?
So, let's get started.
Logistic regression is a statistical and machine learning technique for classifying records of a dataset, based on the values of the input fields.
Let's say we have a telecommunication dataset that we'd would like to analyze, in order to understand which customers might leave us next month.
This is historical customer data where each row represents one customer.
Imagine that you're an analyst at this company and you have to find out who is leaving and why.
You'll use the dataset to build a model based on historical records and use it to predict the future churn within the customer group.
The data set includes information about: - Services that each customer has signed up for, - Customer account information,
- Demographic information about customers, like gender and age-range,
- And also Customers who've left the company within the last month.
The column is called Churn.
We can use logistic regression to build a model for predicting customer churn, using the given features.
In logistic regression, we use one or more independent variables such as tenure, age and income to predict an outcome, such as churn, which we call a dependent variable, representing whether or not customers will stop using the service.
Logistic regression is analogous to linear regression but tries to predict a categorical or discrete target field instead of a numeric one.
In linear regression, we might try to predict a continuous value of variables, such as the price of a house, blood pressure of patient, or fuel consumption of a car.
But, in logistic regression, we predict a variable which is binary, such as, Yes/No, TRUE/FALSE, successful or Not successful, pregnant/Not pregnant, and so on, all of which can all be coded as 0 or 1.
In logistic regression, dependent variables should be continuous; if categorical, they should be dummy or indicator-coded.
This means we have to transform them to some continuous value.
Please note that logistic regression can be used for both binary classification and multiclass classification, but for simplicity, in this video, we'll focus on binary classification.
Let's examine some applications of logistic regression before we explain how they work.
As mentioned, logistic regression is a type of classification algorithm, so it can be used in different situations, for example: - To predict the probability of a person having a heart attack within a specified time period, based on our knowledge of the person's age, sex, and body mass index.
- Or to predict the chance of mortality in an injured patient, or to predict whether a patient has a given disease, such as diabetes, based on observed characteristics of that patient, such as weight, height, blood pressure, and results of various blood tests, and so on.
- In a marketing context, we can use it to predict the likelihood of a customer purchasing a product or halting a subscription, as we've done in our churn example.
- We can also use logistic regression to predict the probability of failure of a given process, system, or product.

- We can even use it to predict the likelihood of a homeowner defaulting on a mortgage.
These are all good examples of problems that can be solved using logistic regression.
Notice that in all of these examples, not only do we predict the class of each case,
we also measure the probability of a case belonging to a specific class.
There are different machine algorithms which can classify or estimate a variable.
The question is, when should we use Logistic Regression?
Here are four situations in which Logistic regression is a good candidate:
First, when the target field in your data is categorical, or specifically, is binary,
such as 0/1, yes/no, churn or no churn, positive/negative, and so on.
Second, you need the probability of your prediction, for example, if you want to know what the probability is, of a customer buying a product.
Logistic regression returns a probability score between 0 and 1 for a given sample of
data.
In fact, logistic regressing predicts the probability of that sample, and we map the
cases to a discrete class based on that probability.
Third, if your data is linearly separable.
The decision boundary of logistic regression is a line or a plane or a hyper-plane.
A classifier will classify all the points on one side of the decision boundary as belonging
to one class and all those on the other side as belonging to the other class.
For example, if we have just two features (and are not applying any polynomial processing),
we can obtain an inequality like $\theta_0 + \theta_1 x_1 + \theta_2 x_2 > 0$, which is a half-plane,
easily plottable.
Please note that in using logistic regression, we can also achieve a complex decision boundary
using polynomial processing as well, which is out of scope here.
You'll get more insight from decision boundaries when you understand how logistic regression works.
Fourth, you need to understand the impact of a feature.
You can select the best features based on the statistical significance of the logistic
regression model coefficients or parameters.
That is, after finding the optimum parameters, a feature x with the weight θ_1 close to
0, has a smaller effect on the prediction, than features with large absolute values of
θ_1.
Indeed, it allows us to understand the impact an independent variable has on the dependent
variable while controlling other independent variables.
Let's look at our dataset again.
We define the independent variables as X, and dependent variable as Y.
Notice that, for the sake of simplicity, we can code the target or dependent values to
0 or 1.
The goal of logistic regression is to build a model to predict the class of each sample
(which in this case is a customer) as well as the probability of each sample belonging
to a class.
Given that, let's start to formalize the problem.
X is our dataset, in the space of real numbers of m by n, that is, of m dimensions or features
and n records.
And y is the class that we want to predict, which can be either zero or one.
Ideally, a logistic regression model, so called $\hat{y}$ (y-hat), can predict that the class of
a customer is 1, given its features x.
It can also be shown quite easily, that the probability of a customer being in class 0
can be calculated as 1 minus the probability that the class of the customer is 1.
Thanks for watching this video.

LOGISTIC REGRESSION VS LINEAR REGRESSION

Predicting customer income

tenure	age	address	income	ed	employ	equip	callcard	wireless	churn
11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	1
33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	1
23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	0.0	0
38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	1.0	0
7.0	35.0	14.0	80.0	2.0	15.0	0.0	1.0	0.0	0



COGNITIVE

Predicting churn using linear regression

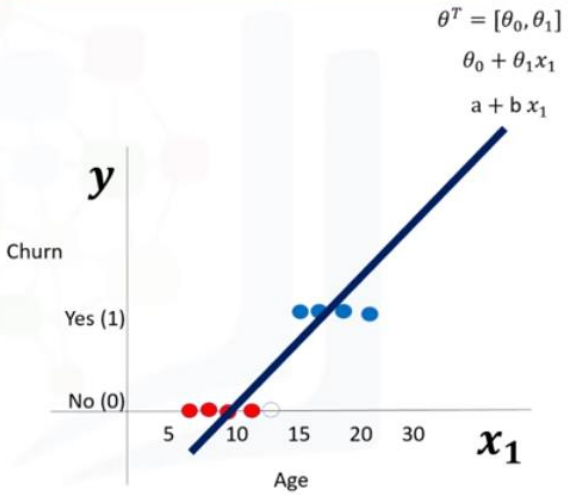
$\theta^T X = \theta_0 + \theta_1 x_1$

$\theta^T X = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$

$\theta^T = [\theta_0, \theta_1, \theta_2, \ldots]$ $\quad X = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \ldots \end{bmatrix}$

$\theta^T = [\theta_0, \theta_1]$

$\theta_0 + \theta_1 x_1$



COGNITIVE CLASS

Linear regression in classification problems?

$\theta^T X = \theta_0 + \theta_1 x_1$

$p_1 = [13]$ → $\theta^T X = -1 + 0.1 \,.\, x_1$

= -1 + 0.1 x 13

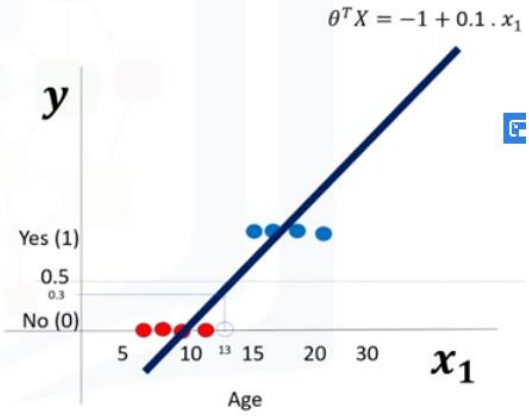
= 0.3

$\hat{y} = \begin{cases} 0 & \text{if } \theta^T X < 0.5 \\ 1 & \text{if } \theta^T X \geq 0.5 \end{cases}$

$\theta^T X = 0.3$

$\theta^T X < 0.5$ → Class 0

$\theta^T X = -1 + 0.1 \,.\, x_1$



Sigmoid function in logistic regression

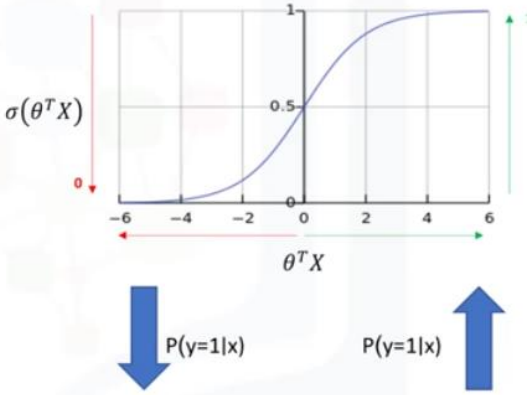
- Logistic Function

$$\sigma(\theta^T X) = \frac{1}{1 + e^{-\theta^T X}}$$

$\sigma(\theta^T X) = 1$

$\sigma(\theta^T X) = 0$

$[0, 1]$



Clarification of the customer churn model

What is the output of our model?

- P(Y=1|X)
- P(y=0|X) = 1 - P(y=1|x)

- P(Churn=1|income,age) = 0.8
- P(Churn=0|income,age) = 1 - 0.8 = 0.2

$\sigma(\theta^T X)$ → P(y=1|x)

$1 - \sigma(\theta^T X)$ → P(y=0|x)

The training process

$\sigma(\theta^T X)$ → P(y=1|x)

1. Initialize θ. — $\theta = [-1, 2]$
2. Calculate $\hat{y} = \sigma(\theta^T X)$ for a customer. — $\hat{y} = \sigma([-1, 2] \times [2, 5]) = 0.7$
3. Compare the output of $\hat{y}$ with actual output of customer, y, and record it as error. — Error = 1-0.7 = 0.3
4. Calculate the error for all customers. — $Cost = J(\theta)$
5. Change the θ to reduce the cost. — θ_{new}
6. Go back to step 2.

Hello, and welcome! In this video we'll learn more about training a logistic regression model. Also, we'll be discussing how to change the parameters of the model to better estimate the outcome. Finally, we talk about the cost function and gradient descent in logistic regression, as a way to optimize the model. So let's start.

The main objective of training in logistic regression, is to change the parameters of the model, so as to be the best estimation of the labels of the samples in the dataset, for example, the customer churn. "How do we do that?"
In brief, first we have to look at the cost function and see what the relation is between the cost function and the parameters θ. So, we should formulate the cost function. Then, using the derivative of the cost function, we can find how to change the parameters to reduce the cost, or rather, the error. Let's dive into it to see how it works.
But before I explain it, I should highlight for you that it needs some basic mathematical background to understand it. However, you shouldn't worry about it as most data science languages like Python, R and Scala have some packages or libraries that calculate these parameters for you. So, let's take a look at it.
Let's first find the cost function equation for a sample case.
To do this, we can use one of the customers in the churn problem.
There's normally a general equation for calculating the cost.
The cost function is the difference between the actual values of y and our model output, y^. This is a general rule for most cost functions in machine learning. We can show this as the "Cost of our model comparing it with actual labels," which is the "difference between the predicted value of our model and actual value of the target field," where the predicted value of our model is is σ(θ^T X). Usually, the square of this equation is used because of the possibility of the negative result, and for the sake of simplicity, half of this value is considered as the cost function, through the derivative process.
Now, we can write the cost function for all the samples in our training set; for example, for all customers, we can write it as the average sum of the cost functions of all cases. It is also called the mean squared error, and, as it is a function of a parameter vector θ, it is shown as J(θ).
Ok, good. We have the cost function. Now, how do we find or set the best weights or parameters that minimize this cost function? The answer is, "We should calculate the minimum point of this cost function and it'll show us the best parameters for our model."
Although we can find the minimum point of a function using the derivative of a function, there's not an easy way to find the global minimum point for such an equation.
Given this complexity, describing how to reach the global minimum for this equation, is outside the scope of this video. So, what is the solution?
Well, we should find another cost function instead; one which has the same behavior but is easier to find its minimum point.
Let's plot the desirable cost function for our model.
Recall that our model is y^. Our actual value is y, which equals 0 or 1, and our model tries to estimate it, as we want to find a simple cost function for our model. For a moment, assume that our desired value for y is 1. This means our model is best if it estimates y=1.
In this case, we need a cost function that returns 0 if the outcome of our model is 1, which is same as the actual label.
And the cost should keep increasing as the outcome of our model gets farther from 1.
And cost should be very large, if the outcome of our model is close to 0.
We can see that the –log() function provides such a cost function for us.
It means, if the actual value is 1, and the model also predicts 1, the –log() function returns 0 cost, but, if the prediction is smaller than 1, the –log() function returns a larger cost value. So, we can use the –log() function for calculating the cost of our logistic regression model.
So, if you recall, we previously noted that, in general, it is difficult to calculate the derivative of the cost function. Well, we can now change it with the -log of our model. We can easily prove that in the case that

desirable y is 1, the cost can be calculated as -log(y^), and in the case that desirable y is 0, the cost can be calculated as -log(1-y^).
Now we can plug it into our total cost function and rewrite it as this function.
So, this is the logistic regression cost function. As you can see for yourself, it penalizes situations in which the class is 0 and the model output is 1, and vice versa.
Remember, however, that y^ does not return a class as output, but it's a value of (0,1), which should be assumed as a probability. Now we can easily use this function to find the parameters of our model in such a way as to minimize the cost.
OK, let's recap what we've done. Our objective was to find a model that best estimates the actual labels. Finding the best model means finding the best parameters θ, for that model. So, the first question was, "How do we find the best parameters for our model?" Well, by finding and minimizing the cost function of our model. In other words, to minimize the J(θ) that we just defined. The next question is: "How do we minimize the cost function?" The answer is, using an optimization approach.
There are different optimization approaches, but we use one of the most famous and effective approaches here, gradient descent. The next question is: "What is gradient descent?"
Generally, gradient descent is an iterative approach to finding the minimum of a function.
Specifically, in our case, gradient descent is a technique to use the derivative of a cost function to change the parameter values, to minimize the cost or error.
Let's see how it works.
The main objective of gradient descent is to change the parameter values so as to minimize the cost. "How can gradient descent do that?"
Think of the parameters or weights in our model to be in a 2-dimensional space, for example, θ1, θ2, for 2 feature sets, age and income.
Recall the cost function, J, that we discussed in the previous slides.
We need to minimize the cost function J, which is a function of variables $\theta 1$, $\theta 2$.
So, let's add a dimension for the observed cost or error, J function.
Let's assume that if we plot the cost function based on all possible values of $\theta 1$, $\theta 2$, we can see something like this. It represents the error value for different values of parameters, that is, Error, which is a function of the parameters.
This is called your "error curve" or "error bowl" of your cost function.
Recall that we want to use this error bowl to find the best parameter values that result in minimizing the cost value. Now, the question is, "Which point is the best point for your cost function?" Yes, you should try to minimize your position on the "error curve." So, what should you do?
You have to find the minimum value of the cost by changing the parameters … but which way? Will you add some value to your weights, or deduct some value? And how much would that value be?
You can select random parameter values that locate a point on the bowl.
You can think of our starting point being the yellow point.
You change the parameters by $\Delta\theta 1$ and $\Delta\theta 2$, and take one step on the surface.
Let's assume we go down one step in the bowl.
As long as we're going downwards we can go one more step.
The steeper the slope, the further we can step.
And we can keep taking steps. As we approach the lowest point, the slope diminishes, so we can take smaller steps until we reach a flat surface.
This point is the minimum point of our curve, and the optimum $\theta 1$, $\theta 2$.
What are these steps really? I mean in which direction should we take these steps to make sure we descend, and how big should the steps be?
To find the direction and size of these steps, in other words, to find how to update the parameters, you should calculate the gradient of the cost function at that point.

The gradient is the slope of the surface at every point.
And, the direction of the gradient is the direction of the greatest uphill.
Now the question is, “How do we calculate the gradient of a cost function at a point?”
If you select a random point on this surface, for example, the yellow point, and take the partial derivative of J(θ) with respect to each parameter at that point, it gives you the slope of the move for each parameter at that point.
Now, if we move in the opposite direction of that slope, it guarantees that we go down in the error curve. For example, if we calculate the derivative of J with respect to $\theta 1$, we find out that it is a positive number.
This indicates that function is increasing as $\theta 1$ increases.
So, to decrease J, we should move in the opposite direction.
This means to move in the direction of the negative derivative for $\theta 1$, i.e. slope.
We have to calculate it for other parameters as well, at each step.
The gradient value also indicates how big of a step to take.
If the slope is large, we should take a large step because we’re far from the minimum.
If the slope is small we should to take a smaller step.
Gradient descent takes increasingly smaller steps towards the minimum with each iteration.
The partial derivative of the cost function J, is calculated using this expression.
If you want to know how the derivative of the J function is calculated, you need to know the derivative concept, which is beyond our scope here.
But to be honest, you don’t really need to remember all the details about it, as you can easily use this equation to calculate the gradients.
So, in a nutshell, this equation returns the slope of that point, and we should update the parameter in the opposite direction of the slope.
A vector of all these slopes is the gradient vector, and we can use this vector to change or update all the parameters. We take the previous values of the parameters and subtract the Error derivative. This results in the new parameters for θ that we know will decrease the cost. Also, we multiply the gradient value by a constant value Mu, which is called the learning rate.
Learning rate gives us additional control on how fast we move on the surface.
In sum, we can simply say, Gradient descent is like taking steps in the current direction of the slope, and the learning rate is like the length of the step you take.
So, these would be our new parameters. Notice that it’s an iterative operation and, in each iteration, we update the parameters and minimize the cost, until the ‘algorithm converge’ is on an acceptable minimum.
OK, let’s recap what we’ve done to this point, by going through the training algorithm again, step-by-step. Step 1. We initialize the parameters with random values. Step 2. We feed the cost function with the training set, and calculate the cost. We expect a high error rate as the parameters are set randomly. Step 3. We calculate the gradient of the cost function, keeping in mind that we have to use a partial derivative.
So, to calculate the gradient vector, we need all the training data to feed the equation for each parameter. Of course, this is an expensive part of the algorithm, but there are some solutions for this.
Step 4. We update the weights with new parameter values.
Step 5. Here we go back to step 2 and feed the cost function again, which has new parameters.
As was explained earlier, we expect less error as we’re going down the error surface.
We continue this loop until we reach a short value of cost, or some limited number of iterations.
Step 6. The parameters should be roughly found after some iterations.
This means the model is ready. And we can use it to predict the probability of a customer staying or leaving.
Thanks for watching this video!

General cost function

$\sigma(\theta^T X)$ → P(y=1|x)

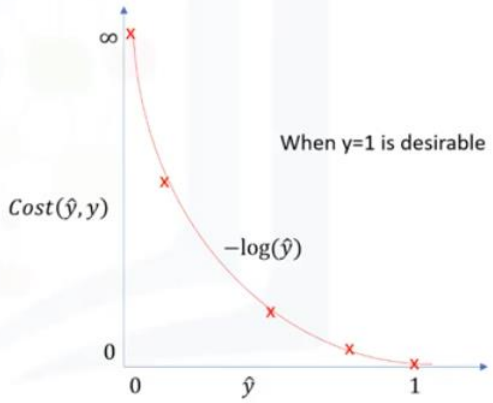
- Change the weight -> Reduce the cost
- Cost function

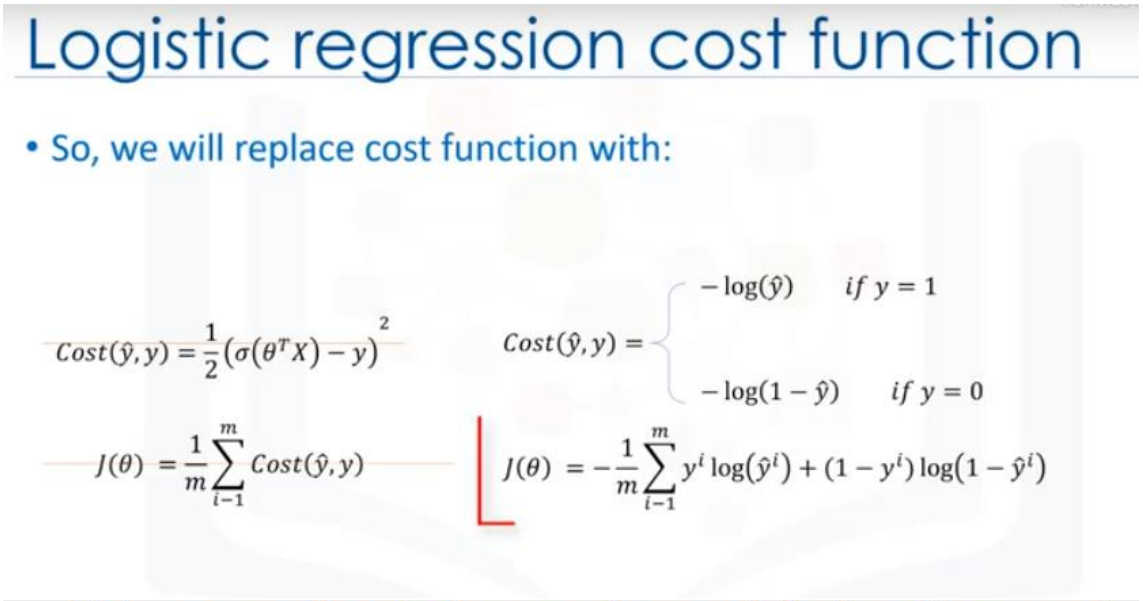
$$Cost(\hat{y}, y) = \frac{1}{2}(\sigma(\theta^T X) - y)^2$$

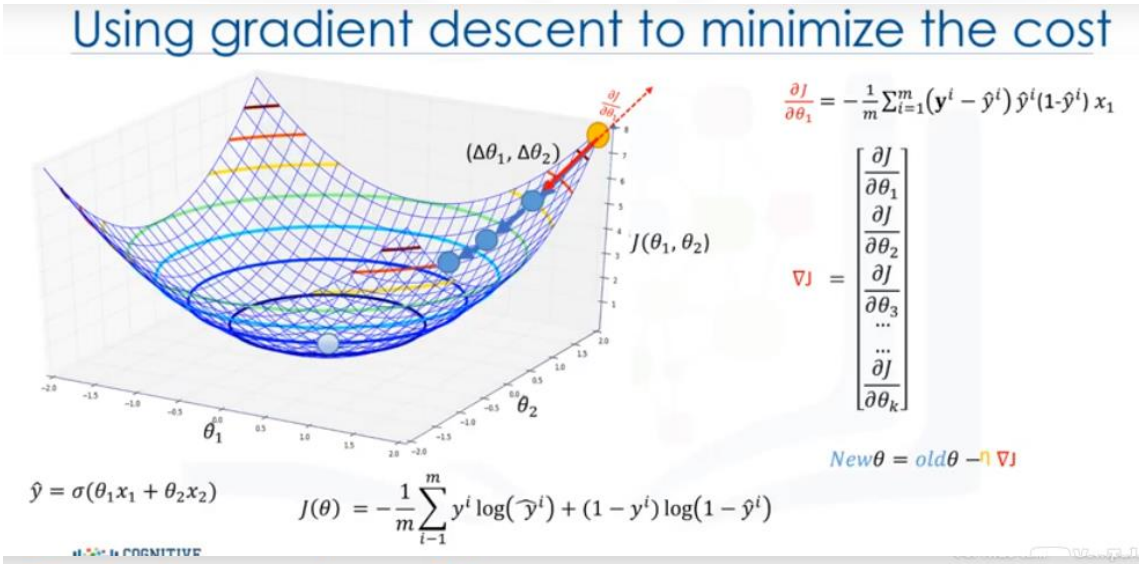
$$J(\theta) = \frac{1}{m}\sum_{i-1}^{m} Cost(\hat{y}, y)$$

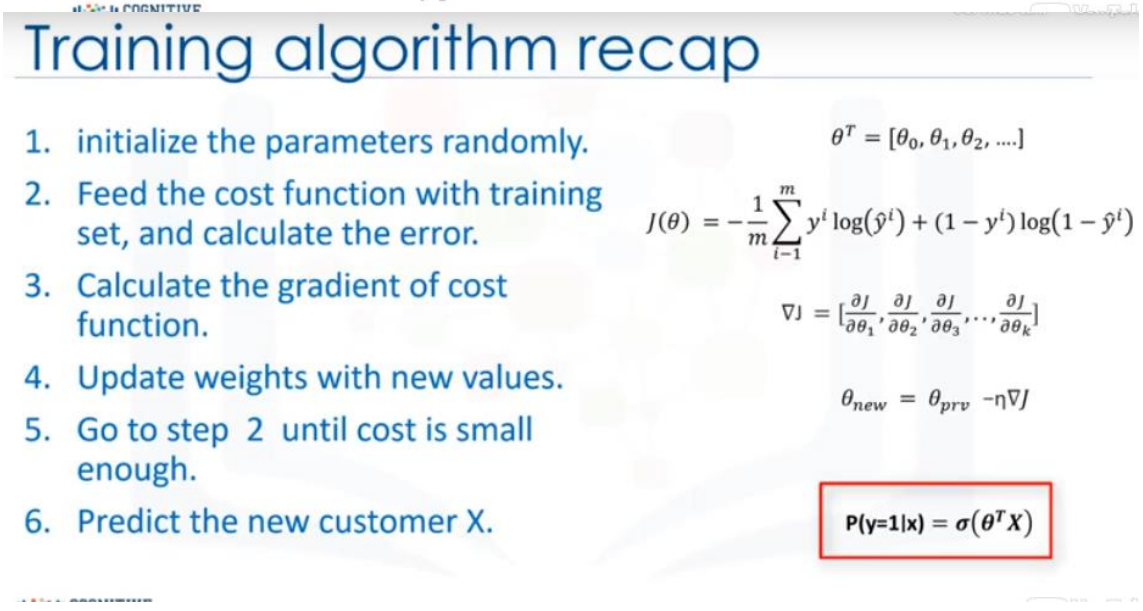
Plotting the cost function of the model

- Model $\hat{y}$
- Actual Value y=1 or 0
- If Y=1, and $\hat{y}$=1 → cost = 0
- If Y=1, and $\hat{y}$=0 → cost = large









Hello, and welcome! In this video we'll learn more about training a logistic regression model. Also, we'll be discussing how to change

the parameters of the model to better estimate the outcome.
Finally, we talk about the cost function and gradient descent in logistic regression, as a way to optimize the model. So let's start.
The main objective of training in logistic regression, is to change the parameters of the model, so as to be the best estimation of the labels of the samples in the dataset, for example, the customer churn. "How do we do that?"
In brief, first we have to look at the cost function and see what the relation is between the cost function and the parameters θ. So, we should formulate the cost function.
Then, using the derivative of the cost function, we can find how to change the parameters to reduce the cost, or rather, the error. Let's dive into it to see how it works.
But before I explain it, I should highlight for you that it needs some basic mathematical background to understand it. However, you shouldn't worry about it as most data science languages like Python, R and Scala have some packages or libraries that calculate these parameters for you. So, let's take a look at it.
Let's first find the cost function equation for a sample case.
To do this, we can use one of the customers in the churn problem.
There's normally a general equation for calculating the cost.
The cost function is the difference between the actual values of y and our model output, y^.
This is a general rule for most cost functions in machine learning. We can show this as the "Cost of our model comparing it with actual labels," which is the "difference between the predicted value of our model and actual value of the target field," where the predicted value of our model is is σ(θ^T X). Usually, the square of this equation is used because of the possibility of the negative result, and for the sake of simplicity, half of this value is considered as the cost function, through the derivative process.
Now, we can write the cost function for all the samples in our training set; for example, for all customers, we can write it as the average sum of the cost functions of all cases. It is also called the mean squared error, and, as it is a function of a parameter vector θ, it is shown as J(θ).
Ok, good. We have the cost function. Now, how do we find or set the best weights or parameters that minimize this cost function? The answer is, "We should calculate the minimum point of this cost function and it'll show us the best parameters for our model."
Although we can find the minimum point of a function using the derivative of a function, there's not an easy way to find the global minimum point for such an equation.
Given this complexity, describing how to reach the global minimum for this equation, is outside the scope of this video. So, what is the solution?
Well, we should find another cost function instead; one which has the same behavior but is easier to find its minimum point.
Let's plot the desirable cost function for our model.
Recall that our model is y^. Our actual value is y, which equals 0 or 1, and our model tries to estimate it, as we want to find a simple cost function for our model. For a moment, assume that our desired value for y is 1. This means our model is best if it estimates y=1.
In this case, we need a cost function that returns 0 if the outcome of our model is 1, which is same as the actual label.
And the cost should keep increasing as the outcome of our model gets farther from 1.
And cost should be very large, if the outcome of our model is close to 0.
We can see that the –log() function provides such a cost function for us.
It means, if the actual value is 1, and the model also predicts 1, the –log() function returns 0 cost, but, if the prediction is smaller than 1, the –log() function returns a larger cost value. So, we can use the –log() function for calculating the cost of our logistic regression model.

So, if you recall, we previously noted that, in general, it is difficult to calculate the derivative of the cost function. Well, we can now change it with the -log of our model. We can easily prove that in the case that desirable y is 1, the cost can be calculated as -log(y^), and in the case that desirable y is 0, the cost can be calculated as -log(1-y^).
Now we can plug it into our total cost function and rewrite it as this function.
So, this is the logistic regression cost function. As you can see for yourself, it penalizes situations in which the class is 0 and the model output is 1, and vice versa.
Remember, however, that y^ does not return a class as output, but it's a value of (0,1), which should be assumed as a probability. Now we can easily use this function to find the parameters of our model in such a way as to minimize the cost.
OK, let's recap what we've done. Our objective was to find a model that best estimates the actual labels. Finding the best model means finding the best parameters θ, for that model. So, the first question was, "How do we find the best parameters for our model?" Well, by finding and minimizing the cost function of our model. In other words, to minimize the J(θ) that we just defined. The next question is: "How do we minimize the cost function?" The answer is, using an optimization approach.
There are different optimization approaches, but we use one of the most famous and effective approaches here, gradient descent. The next question is: "What is gradient descent?"
Generally, gradient descent is an iterative approach to finding the minimum of a function.
Specifically, in our case, gradient descent is a technique to use the derivative of a cost function to change the parameter values, to minimize the cost or error.
Let's see how it works.
The main objective of gradient descent is to change the parameter values so as to minimize the cost. "How can gradient descent do that?"
Think of the parameters or weights in our model to be in a 2-dimensional space, for example, θ1, θ2, for 2 feature sets, age and income.
Recall the cost function, J, that we discussed in the previous slides.
We need to minimize the cost function J, which is a function of variables $\theta 1$, $\theta 2$.
So, let's add a dimension for the observed cost or error, J function.
Let's assume that if we plot the cost function based on all possible values of $\theta 1$, $\theta 2$, we can see something like this. It represents the error value for different values of parameters, that is, Error, which is a function of the parameters.
This is called your "error curve" or "error bowl" of your cost function.
Recall that we want to use this error bowl to find the best parameter values that result in minimizing the cost value. Now, the question is, "Which point is the best point for your cost function?" Yes, you should try to minimize your position on the "error curve." So, what should you do?
You have to find the minimum value of the cost by changing the parameters … but which way? Will you add some value to your weights, or deduct some value? And how much would that value be?
You can select random parameter values that locate a point on the bowl.
You can think of our starting point being the yellow point.
You change the parameters by $\Delta\theta 1$ and $\Delta\theta 2$, and take one step on the surface.
Let's assume we go down one step in the bowl.
As long as we're going downwards we can go one more step.
The steeper the slope, the further we can step.
And we can keep taking steps. As we approach the lowest point, the slope diminishes, so we can take smaller steps until we reach a flat surface.
This point is the minimum point of our curve, and the optimum $\theta 1$, $\theta 2$.
What are these steps really? I mean in which direction should we take these

steps to make sure we descend, and how big should the steps be?
To find the direction and size of these steps, in other words, to find how to update the parameters, you should calculate the gradient of the cost function at that point.
The gradient is the slope of the surface at every point.
And, the direction of the gradient is the direction of the greatest uphill.
Now the question is, "How do we calculate the gradient of a cost function at a point?"
If you select a random point on this surface, for example, the yellow point, and take the partial derivative of J(θ) with respect to each parameter at that point, it gives you the slope of the move for each parameter at that point.
Now, if we move in the opposite direction of that slope, it guarantees that we go down in the error curve. For example, if we calculate the derivative of J with respect to $\theta 1$, we find out that it is a positive number.
This indicates that function is increasing as $\theta 1$ increases.
So, to decrease J, we should move in the opposite direction.
This means to move in the direction of the negative derivative for $\theta 1$, i.e. slope.
We have to calculate it for other parameters as well, at each step.
The gradient value also indicates how big of a step to take.
If the slope is large, we should take a large step because we're far from the minimum.
If the slope is small we should to take a smaller step.
Gradient descent takes increasingly smaller steps towards the minimum with each iteration.
The partial derivative of the cost function J, is calculated using this expression.
If you want to know how the derivative of the J function is calculated, you need to know the derivative concept, which is beyond our scope here.
But to be honest, you don't really need to remember all the details about it, as you can easily use this equation to calculate the gradients.
So, in a nutshell, this equation returns the slope of that point, and we should update the parameter in the opposite direction of the slope.
A vector of all these slopes is the gradient vector, and we can use this vector to change or update all the parameters. We take the previous values of the parameters and subtract the Error derivative. This results in the new parameters for θ that we know will decrease the cost. Also, we multiply the gradient value by a constant value Mu, which is called the learning rate.
Learning rate gives us additional control on how fast we move on the surface.
In sum, we can simply say, Gradient descent is like taking steps in the current direction of the slope, and the learning rate is like the length of the step you take.
So, these would be our new parameters. Notice that it's an iterative operation and, in each iteration, we update the parameters and minimize the cost, until the 'algorithm converge' is on an acceptable minimum.
OK, let's recap what we've done to this point, by going through the training algorithm again, step-by-step. Step 1. We initialize the parameters with random values. Step 2. We feed the cost function with the training set, and calculate the cost. We expect a high error rate as the parameters are set randomly. Step 3. We calculate the gradient of the cost function, keeping in mind that we have to use a partial derivative.
So, to calculate the gradient vector, we need all the training data to feed the equation for each parameter. Of course, this is an expensive part of the algorithm, but there are some solutions for this.
Step 4. We update the weights with new parameter values.
Step 5. Here we go back to step 2 and feed the cost function again, which has new parameters.
As was explained earlier, we expect less error as we're going down the error surface.
We continue this loop until we reach a short value of cost, or some limited number of iterations.
Step 6. The parameters should be roughly found after some iterations.

This means the model is ready. And we can use it to predict the probability of a customer staying or leaving.
Thanks for watching this video!

>>Lab:

What is different between Linear and Logistic Regression?

While Linear Regression is suited for estimating continuous values (e.g. estimating house price), it is not the best tool for predicting the class of an observed data point. In order to estimate the class of a data point, we need some sort of guidance on what would be the **most probable class** for that data point. For this, we use **Logistic Regression**.

Recall linear regression:

As you know, __Linear regression__ finds a function that relates a continuous dependent variable, _y_, to some predictors (independent variables _x1_, _x2_, etc.). For example, Simple linear regression assumes a function of the form:

$$y = \theta 0 + \theta 1 * x1 + \theta 2 * x2 + \ldots$$

and finds the values of parameters _θ0_, _θ1_, _θ2_, etc, where the term _θ0_ is the "intercept". It can be generally shown as:

$$h_\theta(x) = \theta^T X$$

Logistic Regression is a variation of Linear Regression, useful when the observed dependent variable, *y*, is categorical. It produces a formula that predicts the probability of the class label as a function of the independent variables.

Logistic regression fits a special s-shaped curve by taking the linear regression and transforming the numeric estimate into a probability with the following function, which is called sigmoid function σ:

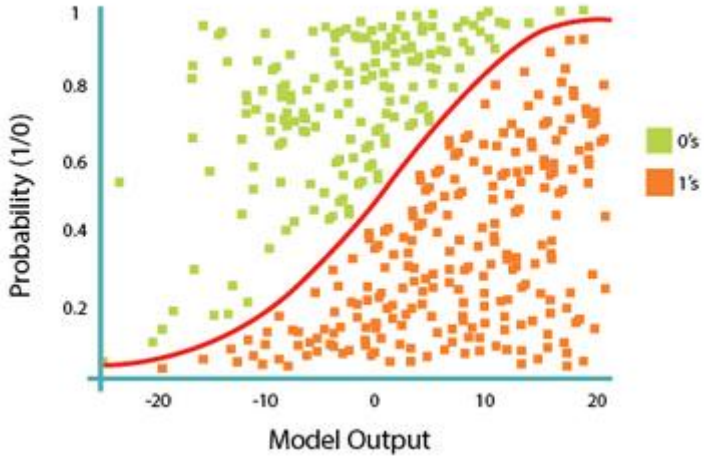
$$h_\theta(x) = \sigma(\theta^T X) = \frac{e^{(\theta 0 + \theta 1 * x1 + \theta 2 * x2 + \ldots)}}{1 + e^{(\theta 0 + \theta 1 * x1 + \theta 2 * x2 + \ldots)}}$$

Or:

$$ProbabilityOfaClass_1 = P(Y = 1|X) = \sigma(\theta^T X) = \frac{e^{\theta^T X}}{1 + e^{\theta^T X}}$$

In this equation, $\theta^T X$ is the regression result (the sum of the variables weighted by the coefficients), `exp` is the exponential function and $\sigma(\theta^T X)$ is the sigmoid or logistic function, also called logistic curve. It is a common "S" shape (sigmoid curve).

So, briefly, Logistic Regression passes the input through the logistic/sigmoid but then treats the result as a probability:



Customer churn with Logistic Regression

A telecommunications company is concerned about the number of customers leaving their land-line business for cable competitors. They need to understand who is leaving. Imagine that you're an analyst at this company and you have to find out who is leaving and why.

Lets first import required libraries:

```
import pandas as pd
import pylab as pl
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
%matplotlib inline
import matplotlib.pyplot as plt
```

Load Data From CSV File

```
churn_df = pd.read_csv("ChurnData.csv")
churn_df.head()
```

	tenure	age	address	income	ed	employ	equip	callcard	wi
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	
2	23.0	30.0	9.0	30.0	1.0	2.0	0.0	0.0	
3	38.0	35.0	5.0	76.0	2.0	10.0	1.0	1.0	

Data pre-processing and selection

Lets select some features for the modeling. Also we change the target data type to be integer, as it is a requirement by the skitlearn algorithm:

```
churn_df = churn_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip',   'callcard', 'wireless','churn']]
churn_df['churn'] = churn_df['churn'].astype('int')
churn_df.head()
```

	tenure	age	address	income	ed	employ	equip	callcard	wireless	churn
0	11.0	33.0	7.0	136.0	5.0	5.0	0.0	1.0	1.0	1
1	33.0	33.0	12.0	33.0	2.0	0.0	0.0	0.0	0.0	1

Lets define X, and y for our dataset:

```
X = np.asarray(churn_df[['tenure', 'age', 'address', 'income', 'ed', 'employ', 'equip']])
X[0:5]
```

```
array([[ 11.,  33.,   7., 136.,   5.,   5.,   0.],
       [ 33.,  33.,  12.,  33.,   2.,   0.,   0.],
       [ 23.,  30.,   9.,  30.,   1.,   2.,   0.],
       [ 38.,  35.,   5.,  76.,   2.,  10.,   1.],
       [  7.,  35.,  14.,  80.,   2.,  15.,   0.]])
```

```
y = np.asarray(churn_df['churn'])
y [0:5]
```

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

[8]:
```
from sklearn import preprocessing
X = preprocessing.StandardScaler().fit(X).transform(X)
X[0:5]
```

[8]:
```
array([[-1.13518441, -0.62595491, -0.4588971 ,  0.4751423 ,  1.6961288 ,
        -0.58477841, -0.85972695],
       [-0.11604313, -0.62595491,  0.03454064, -0.32886061, -0.6433592 ,
        -1.14437497, -0.85972695],
       [-0.57928917, -0.85594447, -0.261522  , -0.35227817, -1.42318853,
        -0.92053635, -0.85972695],
       [ 0.11557989, -0.47262854, -0.65627219,  0.00679109, -0.6433592 ,
        -0.02518185,  1.16316   ],
       [-1.32048283, -0.47262854,  0.23191574,  0.03801451, -0.6433592 ,
         0.53441472, -0.85972695]])
```

Train/Test dataset

Okay, we split our dataset into train and test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (160, 7) (160,)
Test set: (40, 7) (40,)
```

Lets build our model using **LogisticRegression** from Scikit-learn package. This function implements logistic regression and can use different numerical optimizers to find parameters, including 'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga' solvers. You can find extensive information about the pros and cons of these optimizers if you search it in internet.

The version of Logistic Regression in Scikit-learn, support regularization. Regularization is a technique used to solve the overfitting problem in machine learning models. **C** parameter indicates **inverse of regularization strength** which must be a positive float. Smaller values specify stronger regularization. Now lets fit our model with train set:

:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
LR = LogisticRegression(C=0.01, solver='liblinear').fit(X_train,y_train)
LR
```

:
```
LogisticRegression(C=0.01, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001, verbose=0, warm_start=False)
```

Now we can predict using our test set:

:
```
yhat = LR.predict(X_test)
yhat
```

predict_proba returns estimates for all classes, ordered by the label of classes. So, the first column is the probability of class 1, P(Y=1|X), and second column is probability of class 0, P(Y=0|X):

```
yhat_prob = LR.predict_proba(X_test)
yhat_prob
```

```
array([[0.54132919, 0.45867081],
       [0.60593357, 0.39406643],
       [0.56277713, 0.43722287],
```

Evaluation

jaccard index

Lets try jaccard index for accuracy evaluation. we can define jaccard as the size of the intersection divided by the size of the union of two label sets. If the entire set of predicted labels for a sample strictly match with the true set of labels, then the subset accuracy is 1.0; otherwise it is 0.0.

```
from sklearn.metrics import jaccard_similarity_score
jaccard_similarity_score(y_test, yhat)
```

```
0.75
```

confusion matrix

Another way of looking at accuracy of classifier is to look at **confusion matrix**.

```
from sklearn.metrics import classification_report, confusion_matrix
import itertools
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
  if normalize:
      cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
      print("Normalized confusion matrix")
  else:
      print('Confusion matrix, without normalization')

  print(cm)

  plt.imshow(cm, interpolation='nearest', cmap=cmap)
  plt.title(title)
  plt.colorbar()
  tick_marks = np.arange(len(classes))
  plt.xticks(tick_marks, classes, rotation=45)
  plt.yticks(tick_marks, classes)

  fmt = '.2f' if normalize else 'd'
  thresh = cm.max() / 2.
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
      plt.text(j, i, format(cm[i, j], fmt),
               horizontalalignment="center",
               color="white" if cm[i, j] > thresh else "black")
```

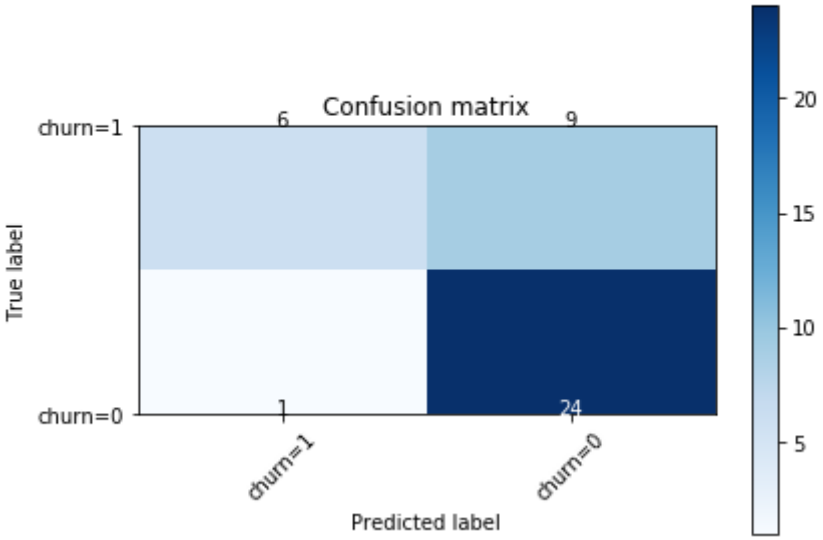
```
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
print(confusion_matrix(y_test, yhat, labels=[1,0]))
```

```
[[ 6  9]
 [ 1 24]]
```

```
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, yhat, labels=[1,0])
np.set_printoptions(precision=2)


# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=['churn=1','churn=0'],normalize= False,  title='Confusion matrix')
```

```
Confusion matrix, without normalization
[[ 6  9]
 [ 1 24]]
```

```
Confusion matrix, without normalization
[[ 6  9]
 [ 1 24]]
```



Look at first row. The firsr row is for customers whose actual churn value in test set is 1. As you can calculate, out of 40 customers, the churn value of 15 of them is 1. And out of these 15, the classifier correctly predicted 6 of them as 1, and 9 of them as 0.

It means, for 6 customers, the actual churn value were 1 in test set, and classifier also correctly predicted those as 1. However, while the actual label of 9 customers were 1, the classifier predicted those as 0, which is not very good. We can consider it as error of the model for first row.

What about the customers with churn value 0? Lets look at the second row. It looks like there were 25 customers whom their churn value were 0.

The classifier correctly predicted 24 of them as 0, and one of them wrongly as 1. So, it has done a good job in predicting the customers with churn value 0. A good thing about confusion matrix is that shows the model's ability to correctly predict or separate the classes. In specific case of binary classifier, such as this example, we can interpret these numbers as the count of true positives, false positives, true negatives, and false negatives.

```
[16]: print (classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       0.73      0.96      0.83        25
           1       0.86      0.40      0.55        15

   micro avg       0.75      0.75      0.75        40
   macro avg       0.79      0.68      0.69        40
weighted avg       0.78      0.75      0.72        40
```

Based on the count of each section, we can calculate precision and recall of each label:

- **Precision** is a measure of the accuracy provided that a class label has been predicted. It is defined by: precision = TP / (TP + FP)
- **Recall** is true positive rate. It is defined as: Recall = TP / (TP + FN)

So, we can calculate precision and recall of each class.

F1 score: Now we are in the position to calculate the F1 scores for each label based on the precision and recall of that label.

The F1score is the harmonic average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It is a good way to show that a classifer has a good value for both recall and precision.

And finally, we can tell the average accuracy for this classifier is the average of the f1-score for both labels, which is 0.72 in our case.

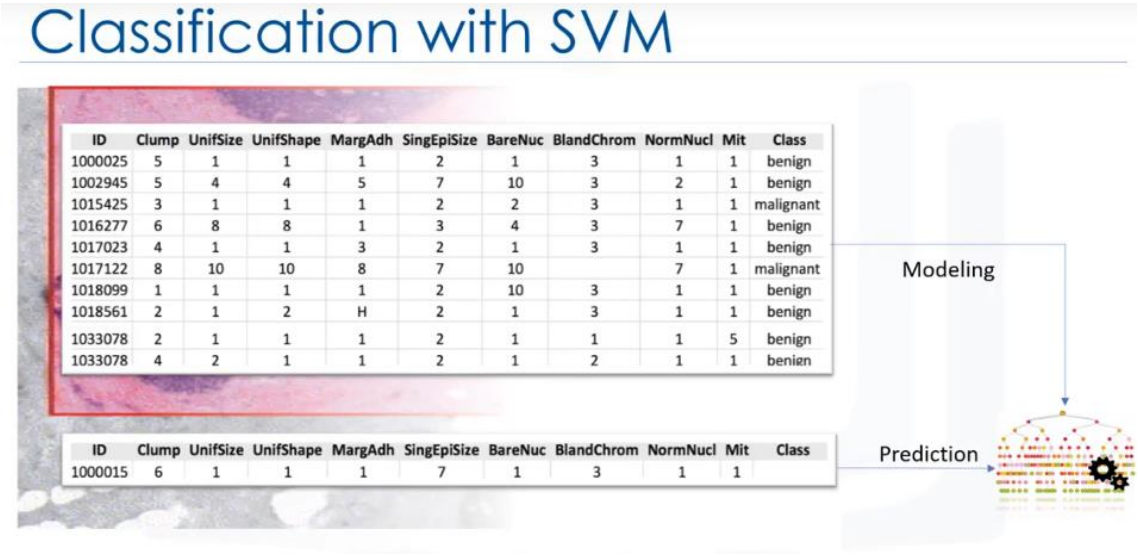
log loss

Now, lets try **log loss** for evaluation. In logistic regression, the output can be the probability of customer churn is yes (or equals to 1). This probability is a value between 0 and 1. Log loss(Logarithmic loss) measures the performance of a classifier where the predicted output is a probability value between 0 and 1.

```
from sklearn.metrics import log_loss
log_loss(y_test, yhat_prob)
```

```
0.6017092478101185
```

SUPPORT VECTOR MACHINE

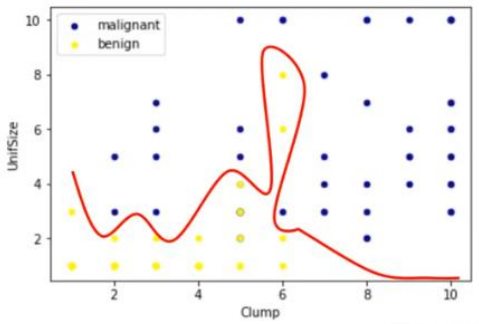


What is SVM?

SVM is a supervised algorithm that classifies cases by finding a separator.

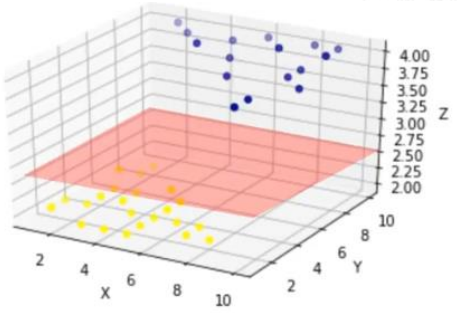
1. Mapping data to a **high-dimensional** feature space
2. Finding a **separator**

Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit	Class
5	1	1	1	2	1	3	1	1	benign
5	4	4	5	7	10	3	2	1	benign
3	1	1	1	2	2	3	1	1	malignant
6	8	8	1	3	4	3	7	1	benign
4	1	1	3	2	1	3	1	1	benign
8	10	10	8	7	10		7	1	malignant
1	1	1	1	2	10	3	1	1	benign
2	1	2	H	2	1	3	1	1	benign
2	1	1	1	2	1	1	1	5	benign
4	2	1	1	2	1	2	1	1	benign



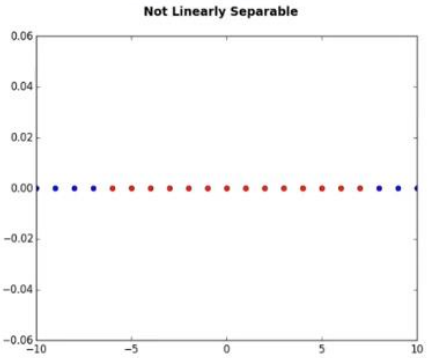
COGNITIVE

Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit	Class
5	1	1	1	2	1	3	1	1	benign
5	4	4	5	7	10	3	2	1	benign
3	1	1	1	2	2	3	1	1	malignant
6	8	8	1	3	4	3	7	1	benign
4	1	1	3	2	1	3	1	1	benign
8	10	10	8	7	10		7	1	malignant
1	1	1	1	2	10	3	1	1	benign
2	1	2	H	2	1	3	1	1	benign
2	1	1	1	2	1	1	1	5	benign
4	2	1	1	2	1	2	1	1	benign



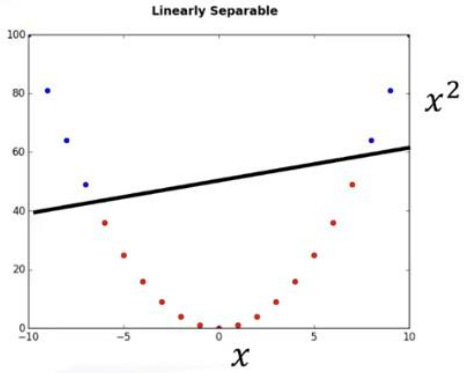
COGNITIVE

Data transformation



Kernelling:

- Linear
- Polynomial
- RBF
- Sigmoid

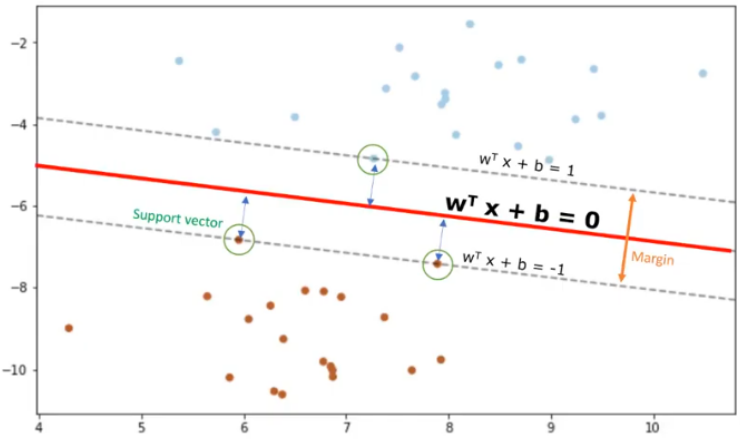


$$\phi(x) = [x, x^2]$$

COGNITIVE

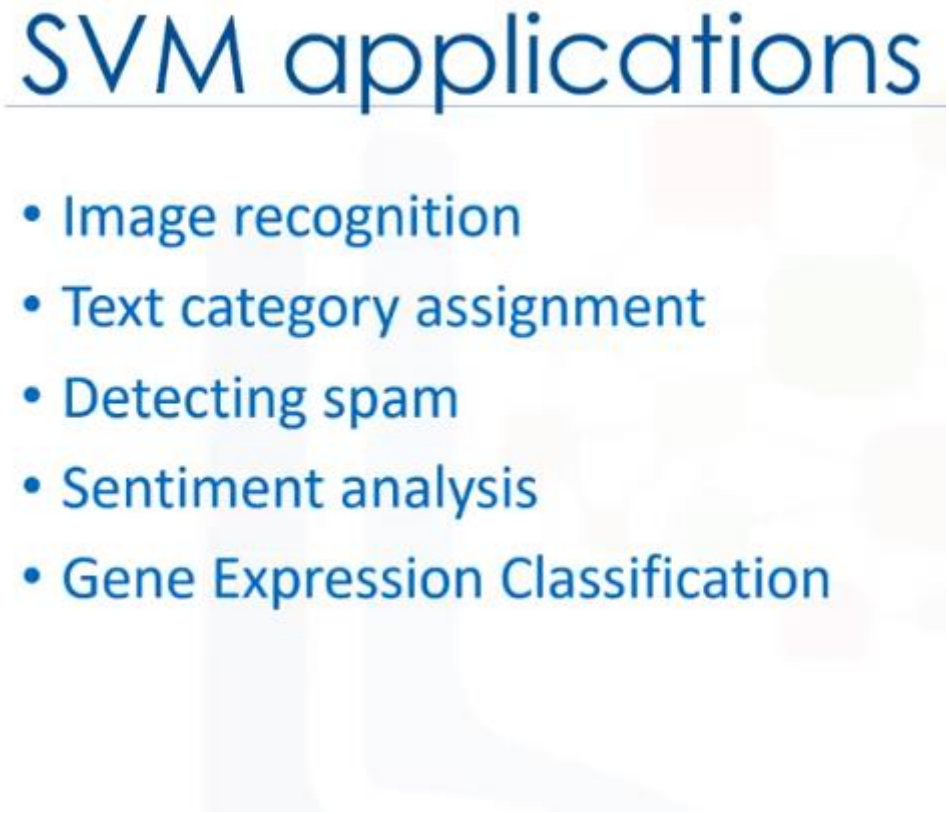
Using SVM to find the hyperplane

Find $\mathbf{w}$ and b such that
$\Phi(\mathbf{w}) = \frac{1}{2}\,\mathbf{w}^T\mathbf{w}$ is minimized;
and for all $\{(\mathbf{x_i}, y_i)\}$: $y_i\,(\mathbf{w}^T\mathbf{x_i} + b) \geq 1$



Pros and cons of SVM

- Advantages:
  - Accurate in high-dimensional spaces
  - Memory efficient
- Disadvantages:
  - Prone to over-fitting
  - No probability estimation
  - Small datasets

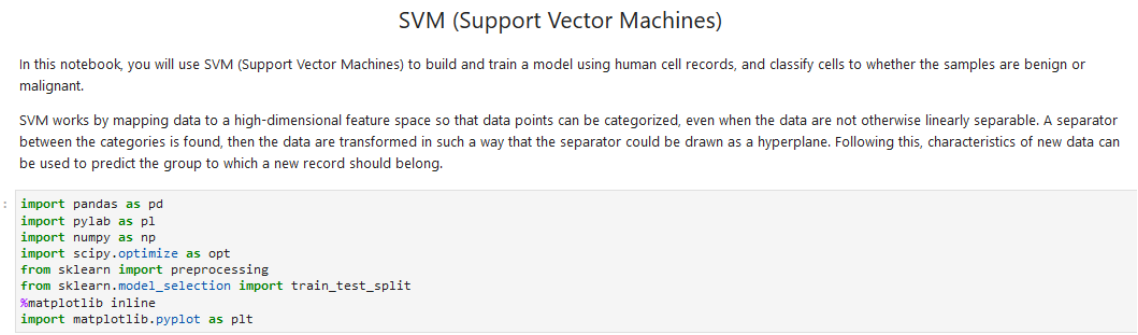


Hello, and welcome!
In this video we will learn a machine learning method called Support Vector Machine (or SVM), which is used for classification.
So let's get started.
Imagine that you've obtained a dataset containing characteristics of thousands of human cell samples extracted from patients who were believed to be at risk of developing cancer.
Analysis of the original data showed that many of the characteristics differed significantly between benign and malignant samples.
You can use the values of these cell characteristics in samples from other patients to give an early indication of whether a new sample might be benign or malignant.
You can use support vector machine, or SVM, as a classifier, to train your model to understand patterns within the data, that might show benign or malignant cells.
Once the model has been trained, it can be used to predict your new or unknown cell with rather high accuracy.
Now, let me give you a formal definition of SVM.
A Support Vector Machine is a supervised algorithm that can classify cases by finding a separator.
SVM works by first, mapping data to a high-dimensional feature space so that data points can be categorized,
even when the data are not otherwise linearly separable.
Then, a separator is estimated for the data.
The data should be transformed in such a way that a separator could be drawn as a hyperplane.
For example, consider the following figure, which shows the distribution of a small set
of cells, only based on their Unit Size and Clump thickness.
As you can see, the data points fall into two different categories.
It represents a linearly, non-separable, dataset.
The two categories can be separated with a curve, but not a line.
That is, it represents a linearly, non-separable dataset, which is the case for most real-world

datasets.
We can transfer this data to a higher dimensional space … for example, mapping it to a 3-dimensional
space.
After the transformation, the boundary between the two categories can be defined by a hyperplane.
As we are now in 3-dimensional space, the separator is shown as a plane.
This plane can be used to classify new or unknown cases.
Therefore, the SVM algorithm outputs an optimal hyperplane that categorizes new examples.
Now, there are two challenging questions to consider:
1) How do we transfer data in such a way that a separator could be drawn as a hyperplane?
and 2) How can we find the best/optimized hyperplane
separator after transformation?
Let's first look at "transforming data" to see how it works.
For the sake of simplicity, imagine that our dataset is 1-dimensional data, this means,
we have only one feature x.
As you can see, it is not linearly separable.
So, what we can do here?
Well, we can transfer it into a 2-dimensional space.
For example, your can increase the dimension of data by mapping x into a new space using
a function, with outputs x and x-squared.
Now, the data is linearly separable, right?
Notice that, as we are in a two dimensional space, the hyperplane is a line dividing a
plane into two parts where each class lays on either side.
Now we can use this line to classify new cases.
Basically, mapping data into a higher dimensional space is called kernelling.
The mathematical function used for the transformation is known as the kernel function, and can be
of different types, such as: Linear, Polynomial, Radial basis function (or RBF), and Sigmoid.
Each of these functions has its own characteristics, its pros and cons, and its equation, but the good news is that you don't need to know them, as most of them are already
implemented in libraries of data science programming languages.
Also, as there's no easy way of knowing which function performs best with any given dataset, we usually choose different functions in turn and compare the results.
Now, we get to another question, specifically, "How do we find the right or optimized separator after transformation?"
Basically, SVMs are based on the idea of finding a hyperplane that best divides a dataset into two classes, as shown here.
As we're in a 2-dimensional space, you can think of the hyperplane as a line that linearly separates the blue points from the red points.
One reasonable choice as the best hyperplane is the one that represents the largest separation, or margin, between the two classes.
So, the goal is to choose a hyperplane with as big a margin as possible.
Examples closest to the hyperplane are support vectors.
It is intuitive that only support vectors matter for achieving our goal; and thus, other
training examples can be ignored.
We try to find the hyperplane in such a way that it has the maximum distance to support vectors.
Please note, that the hyperplane and boundary decision lines have their own equations.
So, finding the optimized hyperplane can be formalized using an equation which involves quite a bit more math, so I'm not going to go through it here, in detail.
That said, the hyperplane is learned from training data using an optimization procedure

that maximizes the margin; and like many other problems, this optimization problem can also be solved by gradient descent, which is out of scope of this video.
Therefore, the output of the algorithm is the values 'w' and 'b' for the line.
You can make classifications using this estimated line.
It is enough to plug in input values into the line equation, then, you can calculate whether an unknown point is above or below the line.
If the equation returns a value greater than 0, then the point belongs to the first class, which is above the line, and vice versa.
The two main advantages of support vector machines are that they're accurate in high dimensional spaces; and, they use a subset of training points in the decision function (called support vectors), so it's also memory efficient.
The disadvantages of support vector machines include the fact that the algorithm is prone for over-fitting, if the number of features is much greater than the number of samples.
Also, SVMs do not directly provide probability estimates, which are desirable in most classification
problems.
And finally, SVMs are not very efficient computationally, if your dataset is very big, such as when you have more than one thousand rows.
And now, our final question is, "In which situation should I use SVM?"
Well, SVM is good for image analysis tasks, such as image classification and handwritten digit recognition.
Also SVM is very effective in text-mining tasks, particularly due to its effectiveness in dealing with high-dimensional data.
For example, it is used for detecting spam, text category assignment, and sentiment analysis.
Another application of SVM is in Gene Expression data classification, again, because of its power in high dimensional data classification.
SVM can also be used for other types of machine learning problems, such as regression, outlier detection, and clustering.
I'll leave it to you to explore more about these particular problems.
This concludes this video … Thanks for watching!

>>Lab:



SVM (Support Vector Machines)

In this notebook, you will use SVM (Support Vector Machines) to build and train a model using human cell records, and classify cells to whether the samples are benign or malignant.

SVM works by mapping data to a high-dimensional feature space so that data points can be categorized, even when the data are not otherwise linearly separable. A separator between the categories is found, then the data are transformed in such a way that the separator could be drawn as a hyperplane. Following this, characteristics of new data can be used to predict the group to which a new record should belong.

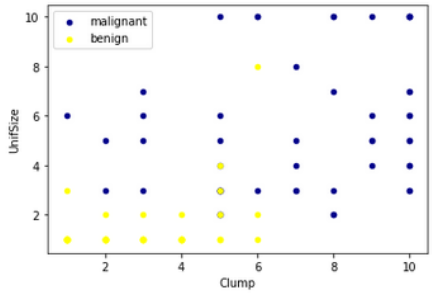
```
import pandas as pd
import pylab as pl
import numpy as np
import scipy.optimize as opt
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
%matplotlib inline
import matplotlib.pyplot as plt
```

Load Data From CSV File

```
cell_df = pd.read_csv("cell_samples.csv")
cell_df.head()
```

	ID	Clump	UnifSize	UnifShape	MargAdh	SingEpiSize	BareNuc	BlandChrom	NormNucl	Mit	Class
0	1000025	5	1	1	1	2	1	3	1	1	2
1	1002945	5	4	4	5	7	10	3	2	1	2
2	1015425	3	1	1	1	2	2	3	1	1	2
3	1016277	6	8	8	1	3	4	3	7	1	2
4	1017023	4	1	1	3	2	1	3	1	1	2

```
ax = cell_df[cell_df['Class'] == 4][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='DarkBlue', label='malignant');
cell_df[cell_df['Class'] == 2][0:50].plot(kind='scatter', x='Clump', y='UnifSize', color='Yellow', label='benign', ax=ax);
plt.show()
```



Data pre-processing and selection

Lets first look at columns data types:

```
cell_df.dtypes
```

```
ID              int64
Clump           int64
UnifSize        int64
UnifShape       int64
MargAdh         int64
SingEpiSize     int64
BareNuc        object
BlandChrom      int64
NormNucl        int64
Mit             int64
Class           int64
dtype: object
```

It looks like the **BareNuc** column includes some values that are not numerical. We can drop those rows:

```
cell_df = cell_df[pd.to_numeric(cell_df['BareNuc'], errors='coerce').notnull()]
cell_df['BareNuc'] = cell_df['BareNuc'].astype('int')
cell_df.dtypes
```

```
ID              int64
Clump           int64
UnifSize        int64
```

```
feature_df = cell_df[['Clump', 'UnifSize', 'UnifShape', 'MargAdh', 'SingEpiSize', 'BareNuc', 'BlandChrom', 'NormNucl', 'Mit']]
X = np.asarray(feature_df)
X[0:5]
```

```
array([[ 5,  1,  1,  1,  2,  1,  3,  1,  1],
       [ 5,  4,  4,  5,  7, 10,  3,  2,  1],
       [ 3,  1,  1,  1,  2,  2,  3,  1,  1],
       [ 6,  8,  8,  1,  3,  4,  3,  7,  1],
       [ 4,  1,  1,  3,  2,  1,  3,  1,  1]])
```

We want the model to predict the value of Class (that is, benign (=2) or malignant (=4)). As this field can have one of only two possible values, we need to change its measurement level to reflect this.

```
cell_df['Class'] = cell_df['Class'].astype('int')
y = np.asarray(cell_df['Class'])
y [0:5]
```

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

Train/Test dataset

Okay, we split our dataset into train and test set:

```
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2, random_state=4)
print ('Train set:', X_train.shape,  y_train.shape)
print ('Test set:', X_test.shape,  y_test.shape)
```

```
Train set: (546, 9) (546,)
Test set: (137, 9) (137,)
```

The SVM algorithm offers a choice of kernel functions for performing its processing. Basically, mapping data into a higher dimensional space is called kernelling. The mathematical function used for the transformation is known as the kernel function, and can be of different types, such as:

1. Linear
2. Polynomial
3. Radial basis function (RBF)
4. Sigmoid

Each of these functions has its characteristics, its pros and cons, and its equation, but as there's no easy way of knowing which function performs best with any given dataset, we usually choose different functions in turn and compare the results. Let's just use the default, RBF (Radial Basis Function) for this lab.

```
from sklearn import svm
clf = svm.SVC(kernel='rbf')
clf.fit(X_train, y_train)
```

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False)
```

After being fitted, the model can then be used to predict new values:

```
yhat = clf.predict(X_test)
yhat [0:5]
```

```
array([2, 4, 2, 4, 2])
```

Evaluation

```
from sklearn.metrics import classification_report, confusion_matrix
import itertools
```

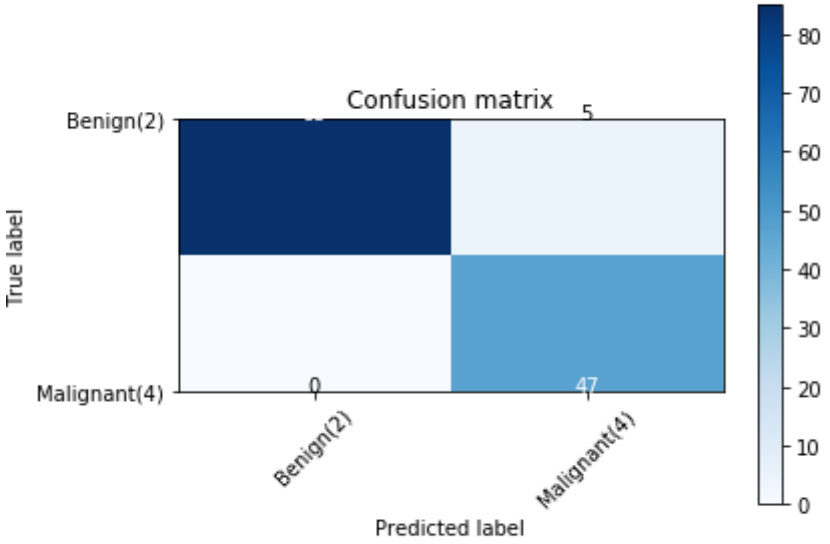
```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
```

```
print(cm)

plt.imshow(cm, interpolation='nearest', cmap=cmap)
plt.title(title)
plt.colorbar()
tick_marks = np.arange(len(classes))
plt.xticks(tick_marks, classes, rotation=45)
plt.yticks(tick_marks, classes)

fmt = '.2f' if normalize else 'd'
thresh = cm.max() / 2.
for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, format(cm[i, j], fmt),
             horizontalalignment="center",
             color="white" if cm[i, j] > thresh else "black")

plt.tight_layout()
plt.ylabel('True label')
plt.xlabel('Predicted label')
```

```
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, yhat, labels=[2,4])
np.set_printoptions(precision=2)

print (classification_report(y_test, yhat))

# Plot non-normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=['Benign(2)','Malignant(4)'],normalize= False,  title='Confusion matrix')
```

```
              precision    recall  f1-score   support

           2       1.00      0.94      0.97        90
           4       0.90      1.00      0.95        47

   micro avg       0.96      0.96      0.96       137
   macro avg       0.95      0.97      0.96       137
weighted avg       0.97      0.96      0.96       137

Confusion matrix, without normalization
[[85  5]
 [ 0 47]]
```

```
Confusion matrix, without normalization
[[85  5]
 [ 0 47]]
```



You can also easily use the **f1_score** from sklearn library:

You can also easily use the **f1_score** from sklearn library:

```
from sklearn.metrics import f1_score
f1_score(y_test, yhat, average='weighted')
```

```
0.9639038982104676
```

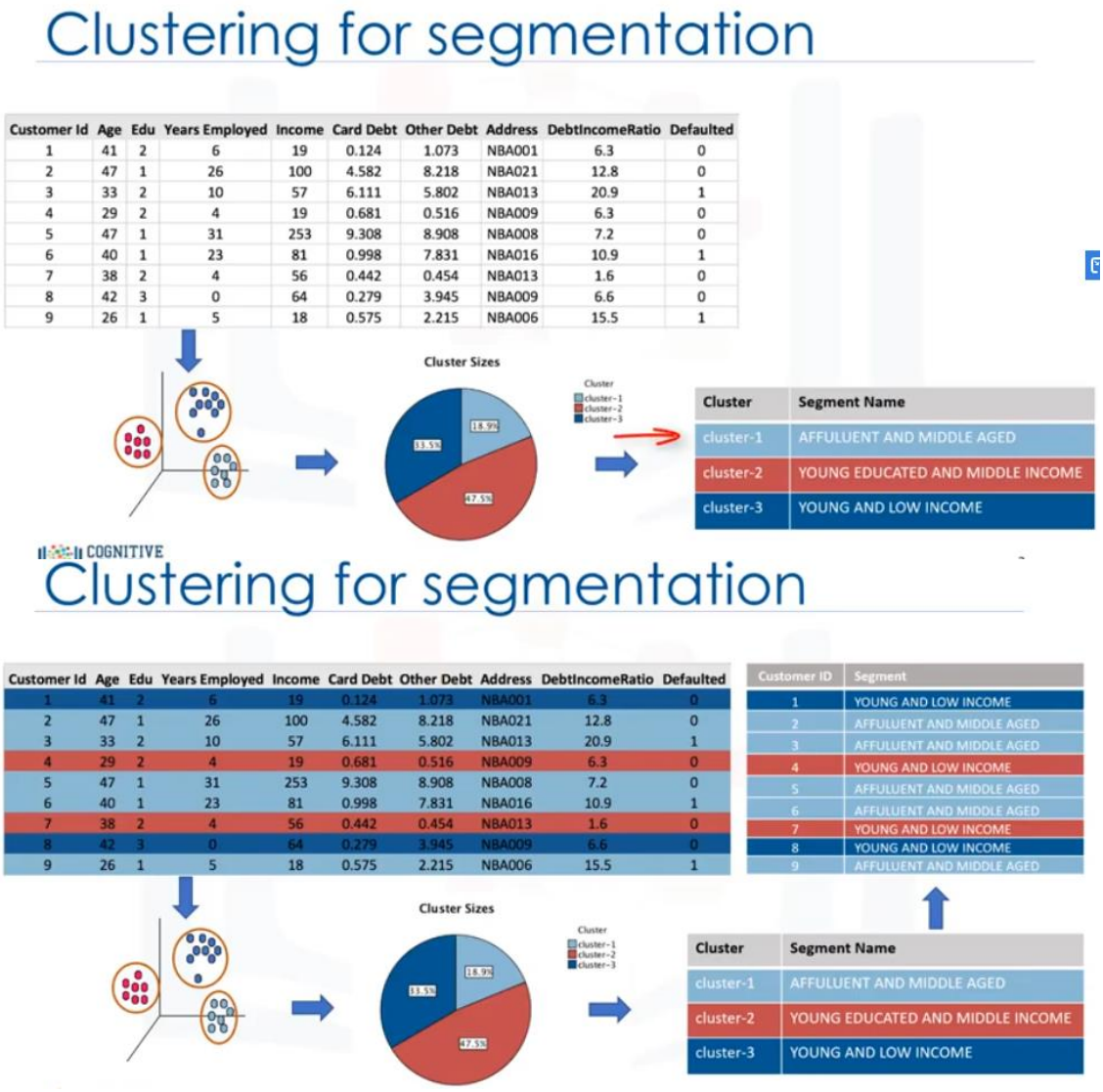
Lets try jaccard index for accuracy:

```
from sklearn.metrics import jaccard_similarity_score
jaccard_similarity_score(y_test, yhat)
```

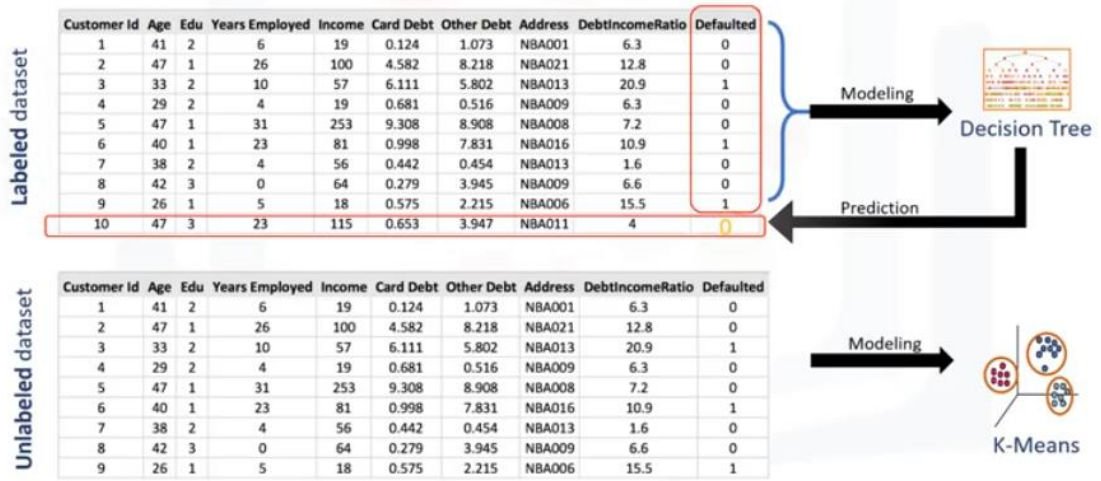
```
0.9635036496350365
```

MODULE 4 - CLUSTERING

INTRO



Clustering Vs. classification



Clustering applications

- PUBLICATION:
  - Auto-categorizing news based on their content
  - Recommending similar news articles
- MEDICINE:
  - Characterizing patient behavior
- BIOLOGY:
  - Clustering genetic markers to identify family ties

Hello, and welcome! In this video, we'll give you a high level introduction to clustering, its applications, and different types of clustering algorithms. Let's get started.

Imagine that you have a customer dataset, and you need to apply customer segmentation on this historical data. Customer segmentation is the practice of partitioning a customer base into groups of individuals that have similar characteristics.

It is a significant strategy as it allows a business to target specific groups of customers so as to more effectively allocate marketing resources.

For example, one group might contain customers who are high-profit and low-risk, that is, more likely to purchase products, or subscribe for a service.

Knowing this information allows a business to devote more time and attention to retaining these customers. Another group might include customers from non-profit organizations, and so on. A general segmentation process is not usually feasible for large volumes of varied data. Therefore, you need an analytical approach to deriving segments and groups from large data sets.

Customers can be grouped based on several factors: including age, gender, interests, spending habits, and so on. The important requirement is to use the available data to understand and identify how customers are similar to each other.

Let's learn how to divide a set of customers into categories, based on characteristics they share. One of the most adopted approaches that can be used for customer segmentation is clustering. Clustering can group data only "unsupervised," based on the similarity of customers to each other.

It will partition your customers into mutually exclusive groups, for example, into 3 clusters. The customers in each cluster are similar to each other demographically.

Now we can create a profile for each group, considering the common characteristics of each cluster. For example, the first group is made up of AFFULUENT AND MIDDLE AGED customers. The second is made up of YOUNG EDUCATED AND MIDDLE INCOME customers. And the third group includes YOUNG AND LOW INCOME customers. Finally, we can assign each individual in our dataset to one of these groups or segments of customers.

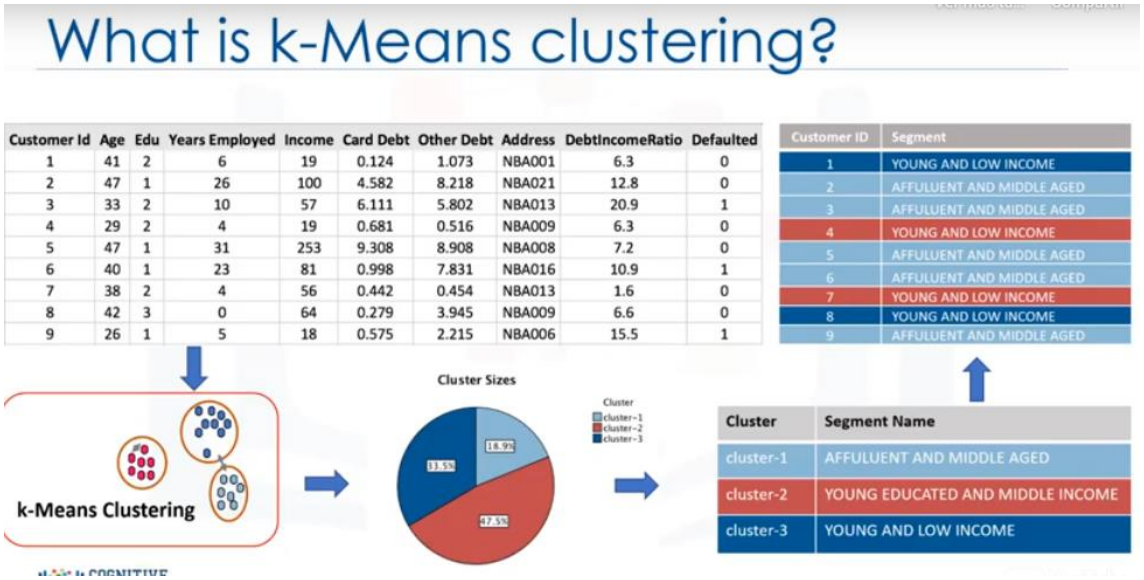
Now imagine that you cross-join this segmented dataset, with the dataset of the product or services that customers purchase from your company.

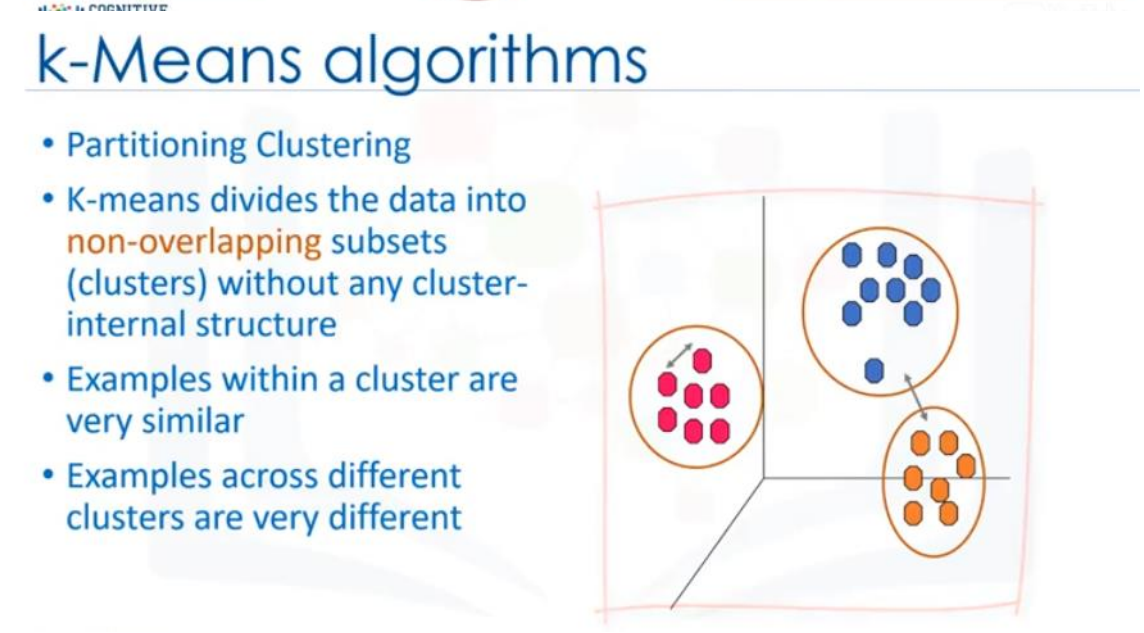
This information would really help to understand and predict the differences in individual

customers' preferences and their buying behaviors across various products.
Indeed, having this information would allow your company to develop highly personalized experiences for each segment. Customer segmentation is one of the popular usages of clustering. Cluster analysis also has many other applications in different domains. So let's first define clustering, and then we'll look at other applications.
Clustering means finding clusters in a dataset, unsupervised.
So, what is a cluster? A cluster is group of data points or objects in a dataset that are similar to other objects in the group, and dissimilar to data points in other clusters. Now, the question is, "What is different between clustering and classification?"
Let's look at our customer dataset again. Classification algorithms predict categorical class labels. This means, assigning instances to pre-defined classes such as "Defaulted" or "Non-Defaulted." For example, if an analyst wants to analyze customer data in order to know which customers might default on their payments, she uses a labeled dataset as training data, and uses classification approaches such as a decision tree, Support Vector Machines (or SVM), or, logistic regression to predict the default value for a new, or unknown customer. Generally speaking, classification is a supervised learning where each training data instance belongs to a particular class.
In clustering, however, the data is unlabelled and the process is unsupervised.
For example, we can use a clustering algorithm such as k-Means, to group similar customers as mentioned, and assign them to a cluster, based on whether they share similar attributes, such as age, education, and so on. While I'll be giving you some examples in different industries, I'd like you to think about more samples of clustering.
In the Retail industry, clustering is used to find associations among customers based on their demographic characteristics and use that information to identify buying patterns of various customer groups. Also, it can be used in recommendation systems to find a group of similar items or similar users, and use it for collaborative filtering, to recommend things like books or movies to customers.
In Banking, analysts find clusters of normal transactions to find the patterns of fraudulent credit card usage. Also, they use clustering to identify clusters of customers, for instance, to find loyal customers, versus churn customers.
In the Insurance industry, clustering is used for fraud detection in claims analysis, or to evaluate the insurance risk of certain customers based on their segments.
In Publication Media, clustering is used to auto-categorize news based on its content, or to tag news, then cluster it, so as to recommend similar news articles to readers.
In Medicine: it can be used to characterize patient behavior, based on their similar characteristics, so as to identify successful medical therapies for different illnesses.
Or, in Biology: clustering is used to group genes with similar expression patterns, or to cluster genetic markers to identify family ties.
If you look around, you can find many other applications of clustering, but generally, clustering can be used for one of the following purposes: exploratory data analysis, summary generation or reducing the scale, outlier detection, especially to be used for fraud detection, or noise removal, finding duplicates in datasets, or, as a pre-processing step for either prediction, other data mining tasks, or, as part of a complex system.
Let's briefly look at different clustering algorithms and their characteristics.
Partitioned-based clustering is a group of clustering algorithms that produces sphere-like clusters, such as k-Means, k-Median, or Fuzzy c-Means.
These algorithms are relatively efficient and are used for Medium and Large sized databases.
Hierarchical clustering algorithms produce trees of clusters, such as Agglomerative and Divisive algorithms. This group of algorithms are very intuitive

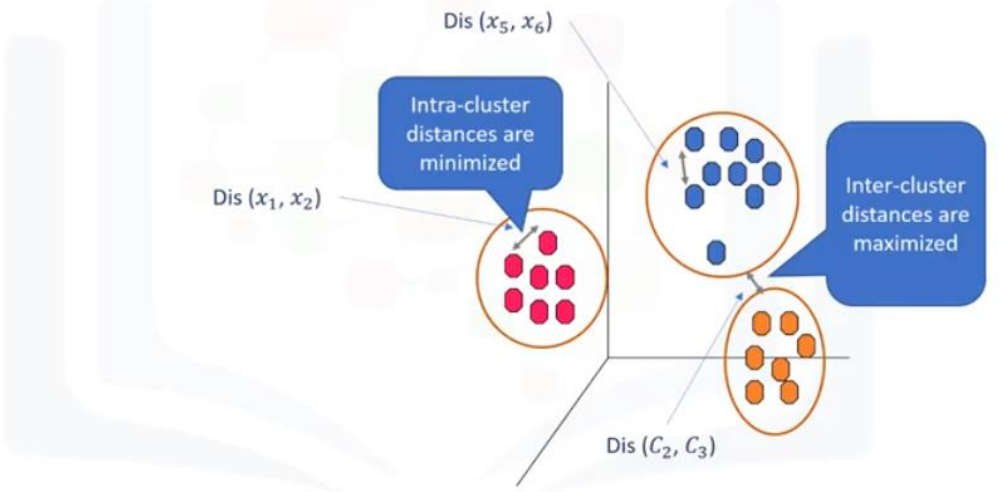
and are generally good for use with small size datasets.
Density based clustering algorithms produce arbitrary shaped clusters.
They are especially good when dealing with spatial clusters or when there is noise in your dataset, for example, the DBSCAN algorithm.
This concludes our video. Thanks for watching!

K-MEANS CLUSTERING





Determine the similarity or dissimilarity



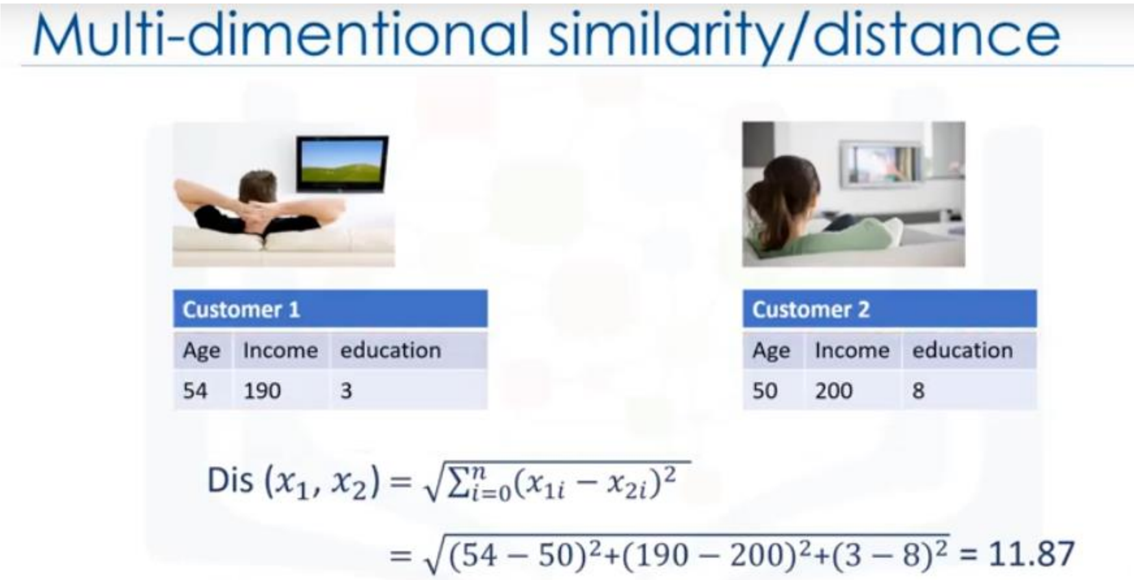
1-dimensional similarity/distance

Customer 1
Age
54

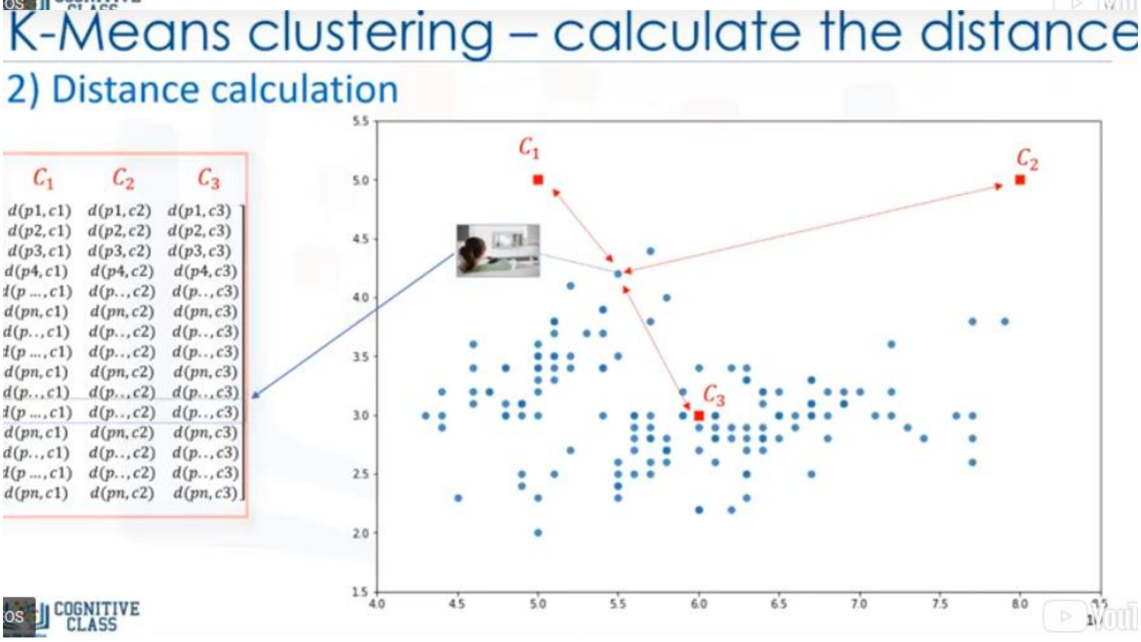
Customer 2
Age
50

$$\text{Dis}\,(x_1, x_2) = \sqrt{\sum_{i=0}^{n}(x_{1i} - x_{2i})^2}$$

$$\text{Dis}\,(x_1, x_2) = \sqrt{(34 - 30)^2} = 4$$

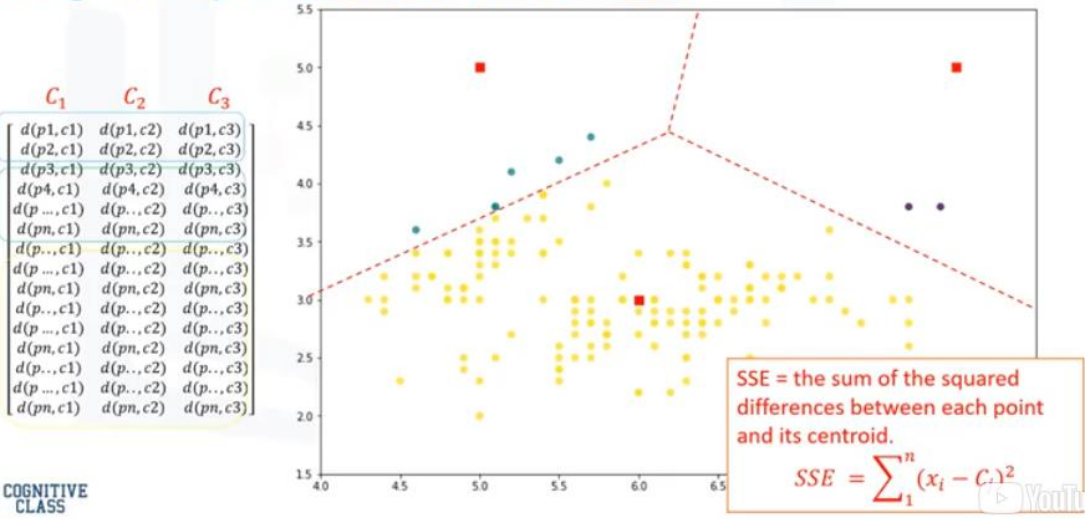

Multi-dimentional similarity/distance
Customer 1
Age Income education
54 190 3
Customer 2
Age Income education
50 200 8
$\text{Dis}(x_1, x_2) = \sqrt{\sum_{i=0}^{n}(x_{1i} - x_{2i})^2}$
$= \sqrt{(54-50)^2+(190-200)^2+(3-8)^2} = 11.87$


k-Means clustering – initialize k
1) Initialize k=3
centroids randomly
C_1
C_2
C_3
COGNITIVE CLASS


K-Means clustering – calculate the distance
2) Distance calculation
C_1 C_2 C_3
d(p1,c1) d(p1,c2) d(p1,c3)
d(p2,c1) d(p2,c2) d(p2,c3)
d(p3,c1) d(p3,c2) d(p3,c3)
d(p4,c1) d(p4,c2) d(p4,c3)
d(pn,c1) d(pn,c2) d(pn,c3)
C_1
C_2
C_3
COGNITIVE CLASS

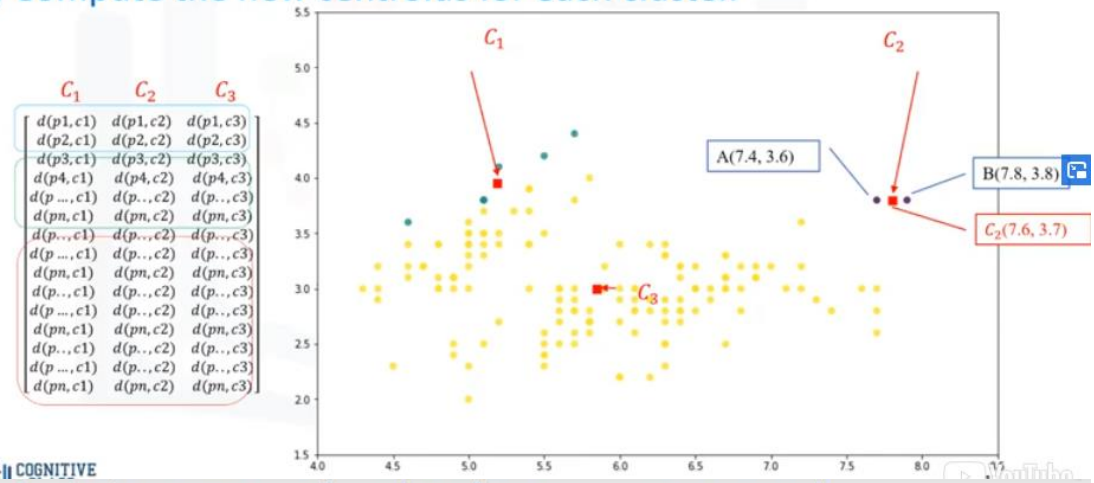
k-Means clustering – assign to centroid

3) Assign each point to the closest centroid



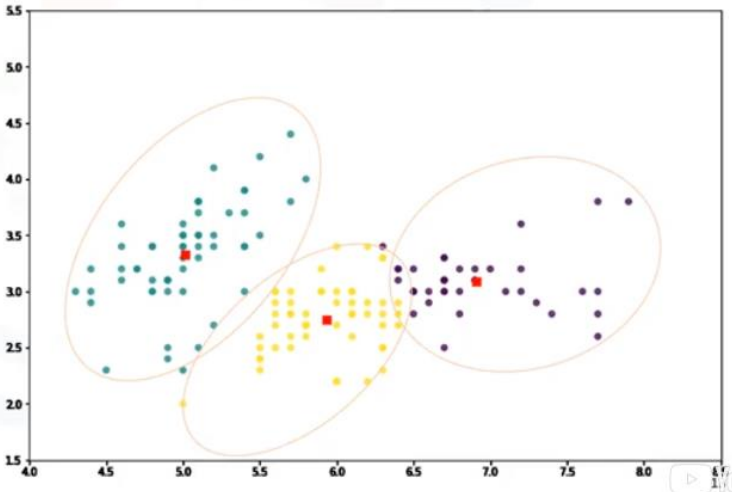
k-Means clustering – compute new centroids

4) Compute the new centroids for each cluster.



k-Means clustering – repeat

5) Repeat until there are no more changes.



Hello, and welcome! In this video, we'll be covering k-Means
clustering. So let's get started.
Imagine that you have a customer dataset, and you need to apply customer segmentation
on this historical data. Customer segmentation is the practice of partitioning
a customer base into groups of individuals that have similar characteristics.
One of the algorithms that can be used for customer segmentation is k-Means clustering.
k-Means can group data only "unsupervised," based on the similarity of customers to each other.
Let's define this technique more formally.
There are various types of clustering algorithms, such as partitioning, hierarchical, or density-based
clustering. k-Means is a type of partitioning clustering,
that is, it divides the data into k non-overlapping subsets (or clusters) without any cluster-internal
structure, or labels. This means, it's an unsupervised algorithm.
Objects within a cluster are very similar and objects across different clusters are
very different or dissimilar. As you can see, for using k-Means, we have
to find similar samples (for example, similar customers).
Now we face a couple of key questions. First, "How can we find the similarity of samples
in clustering?" And then, "How do we measure how similar two customers are with regard
to their demographics?"
Though the objective of k-Means is to form clusters in such a way that similar samples
go into a cluster, and dissimilar samples fall into different clusters, it can be shown
that instead of a similarity metric, we can use dissimilarity metrics.
In other words, conventionally, the distance of samples from each other is used to shape
the clusters. So, we can say, k-Means tries to minimize
the "intra-cluster" distances and maximize the "inter-cluster" distances.
Now, the question is, "How we can calculate the dissimilarity or distance of two cases,
such as two customers?"
Assume that we have two customers, we'll call them customer 1 and 2.
Let's also assume that we have only one feature for each of these two customers, and
that feature is Age. We can easily use a specific type of Minkowski
distance to calculate the distance of these two customers. Indeed, it is the Euclidian
distance. Distance of x1 from x2 is root of 34 minus
30 power 2, which is 4. What about if we have more than one feature,
for example Age and Income?
For example, if we have income and age for each customer, we can still use the same formula,
but this time in a 2-dimensional space.
Also, we can use the same distance matrix for multi-dimensional vectors.
Of course, we have to normalize our feature set to get the accurate dissimilarity measure.
There are other dissimilarity measures as well that can be used for this purpose, but
it is highly dependent on data type and also the domain that clustering is done for it.
For example, you may use Euclidean distance, cosine similarity, average distance, and so
on. Indeed, the similarity measure highly controls
how the clusters are formed, so it is recommended to understand the domain knowledge of your
dataset, and data type of features, and then choose the meaningful distance measurement.
Now, let's see how k-Means clustering works. For the sake of simplicity, let's assume
that our dataset has only two features, the age and income of customers.
This means, it's a 2-dimentional space. We can show the distribution of customers

using a scatterplot. The y-axes indicates Age and the x-axes shows
Income of customers. We try to cluster the customer dataset into
distinct groups (or clusters) based on these two dimensions.
In the first step, we should determine the number of clusters.
The key concept of the k-Means algorithm is that it randomly picks a center point for
each cluster. It means, we must initialize k, which represents
"number of clusters." Essentially, determining the number of clusters
in a data set, or k, is a hard problem in k-Means that we will discuss later.
For now, let's put k equals 3 here, for our sample dataset.
It is like we have 3 representative points for our clusters.
These 3 data points are called centroids of clusters, and should be of same feature size
of our customer feature set. There are two approaches to choose these centroids:
1) We can randomly choose 3 observations out of the dataset and use these observations
as the initial means. Or, 2) We can create 3 random points as centroids
of the clusters, which is our choice that is shown in this plot with red color.
After the initialization step, which was defining the centroid of each cluster, we have to assign
each customer to the closest center. For this purpose, we have to calculate the
distance of each data point (or in our case, each customer) from the centroid points.
As mentioned before, depending on the nature of the data and the purpose for which clustering
is being used, different measures of distance may be used to place items into clusters.
Therefore, you will form a matrix where each row represents the distance of a customer
from each centroid. It is called the "distance-matrix."
The main objective of k-Means clustering is to minimize the distance of data points from
the centroid of its cluster and maximize the distance from other cluster centroids.
So, in this step we have to find the closest centroid to each data point.
We can use the distance-matrix to find the nearest centroid to data points.
Finding the closest centroids for each data point, we assign each data point to that cluster.
In other words, all the customers will fall to a cluster, based on their distance from
centroids. We can easily say that it does not result
in good clusters, because the centroids were chosen randomly from the first.
Indeed, the model would have a high error. Here, error is the total distance of each
point from its centroid. It can be shown as within-cluster sum of squares
error. Intuitively, we try to reduce this error.
It means we should shape clusters in such a way that the total distance of all members
of a cluster from its centroid be minimized. Now, the question is, "How we can turn it
into better clusters, with less error?"
Okay, we move centroids. In the next step, each cluster center will
be updated to be the mean for data points in its cluster. Indeed, each centroid moves
according to their cluster members. In other words, the centroid of each of the
3 clusters becomes the new mean. For example, if Point A coordination is 7.4
and 3.6, and Point B features are 7.8 and 3.8, the new centroid of this cluster with
2 points, would be the average of them, which is 7.6 and 3.7.
Now we have new centroids. As you can guess, once again, we will have
to calculate the distance of all points from the new centroids.
The points are re-clustered and the centroids move again.
This continues until the centroids no longer move.
Please note that whenever a centroid moves, each point's distance to the centroid needs
to be measured again.
Yes, k-Means is an iterative algorithm, and we have to repeat steps 2 to 4 until the algorithm
converges. In each iteration, it will move the centroids,
calculate the distances from new centroids, and assign the data points to the nearest
centroid. It results in the clusters with minimum error,

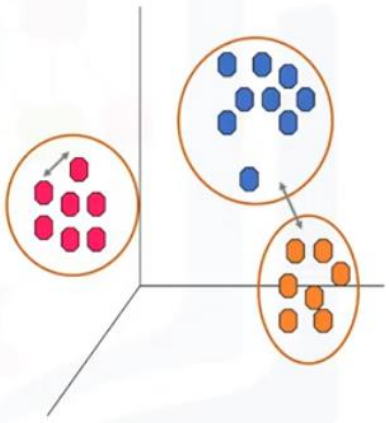
or the most dense clusters. However, as it is a heuristic algorithm, there
is no guarantee that it will converge to the global optimum, and the result may depend
on the initial clusters. It means this algorithm is guaranteed to converge
to a result, but the result may be a local optimum (i.e. not necessarily the best possible
outcome). To solve this problem, it is common to run
the whole process, multiple times, with different starting conditions.
This means, with randomized starting centroids, it may give a better outcome.
And as the algorithm is usually very fast, it wouldn't be any problem to run it multiple times.
Thanks for watching this video!

k-Means clustering algorithm

1. Randomly placing *k* centroids, one for each cluster.
2. Calculate the distance of each point from each centroid.
3. Assign each data point (object) to its closest centroid, creating a cluster.
4. Recalculate the position of the *k* centroids.
5. Repeat the steps 2-4, until the centroids no longer move.

k-Means accuracy

- External approach
  - Compare the clusters with the ground truth, if it is available.
- Internal approach
  - Average the distance between data points within a cluster.



Hello, and welcome! In this video, we'll look at k-Means accuracy
and characteristics. Let's get started.
Let's define the algorithm more concretely before we talk about its accuracy.
A k-Means algorithm works by randomly placing k centroids, one for each cluster.
The farther apart the clusters are placed, the better.
The next step is to calculate the distance of each data point (or object) from the centroids.
Euclidean distance is used to measure the distance from the object to the centroid.
Please note, however, that you can also use different types of distance measurements,
not just Euclidean distance. Euclidean distance is used because it's
the most popular. Then, assign each data point (or object) to
its closest centroid, creating a group. Next, once each data point has been classified
to a group, recalculate the position of the k centroids. The new centroid position is
determined by the mean of all points in the group.
Finally, this continues until the centroids no longer move.
Now the question is, "How can we evaluate the 'goodness' of the clusters formed
by k-Means?" In other words, "How do we calculate the accuracy
of k-Means clustering?" One way is to compare the clusters with the
ground truth, if it's available. However, because k-Means is an unsupervised
algorithm, we usually don't have ground truth in real world problems to be used.
But, there is still a way to say how bad each cluster is, based on the objective of the
k-Means. This value is the average distance between
data points within a cluster. Also, average of the distances of data points
from their cluster centroids can be used as a metric of error for the clustering algorithm.
Essentially, determining the number of clusters in a data set, or k, as in the k-Means algorithm,

is a frequent problem in data clustering. The correct choice of k is often ambiguous, because it's very dependent on the shape and scale of the distribution of points in a data set. There are some approaches to address this problem, but one of the techniques that is commonly used, is to run the clustering across the different values of K, and looking at a metric of accuracy for clustering.
This metric can be "mean distance between data points and their cluster centroid," which indicate how dense our clusters are, or to what extend we minimized the error of clustering.
Then looking at the change of this metric, we can find the best value for k.
But the problem is that with increasing the number of clusters, the distance of centroids to data points will always reduce. This means, increasing K will always decrease the "error."
So, the value of the metric as a function of K is plotted and the "elbow point" is determined, where the rate of decrease sharply shifts. It is the right K for clustering.
This method is called the "elbow" method.
So, let's recap k-Means clustering: k-Means is a partitioned-based clustering, which is:
a) Relatively efficient on medium and large sized datasets;
b) Produces sphere-like clusters, because the clusters are shaped around the centroids;
and c) Its drawback is that we should pre-specify the number of clusters, and this is not an easy task.
Thanks for watching!

>>lab:

Introduction

There are many models for **clustering** out there. In this notebook, we will be presenting the model that is considered the one of the simplest model among them. Despite its simplicity, the **K-means** is vastly used for clustering in many data science applications, especially useful if you need to quickly discover insights from **unlabeled data**. In this notebook, you learn how to use k-Means for customer segmentation.

Some real-world applications of k-means:

- Customer segmentation
- Understand what the visitors of a website are trying to accomplish
- Pattern recognition
- Machine learning
- Data compression

In this notebook we practice k-means clustering with 2 examples:

- k-means on a random generated dataset
- Using k-means for customer segmentation

Import libraries

Lets first import the required libraries. Also run **%matplotlib inline** since we will be plotting in this section.

```
import random
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets.samples_generator import make_blobs
%matplotlib inline
```

k-Means on a randomly generated dataset

Lets create our own dataset for this lab!

First we need to set up a random seed. Use **numpy's random.seed()** function, where the seed will be set to **0**

```
np.random.seed(0)
```

Next we will be making *random clusters* of points by using the **make_blobs** class. The **make_blobs** class can take in many inputs, but we will be using these specific ones.

Input

- **n_samples**: The total number of points equally divided among clusters.
  - Value will be: 5000
- **centers**: The number of centers to generate, or the fixed center locations.
  - Value will be: [[4, 4], [-2, -1], [2, -3],[1,1]]
- **cluster_std**: The standard deviation of the clusters.
  - Value will be: 0.9

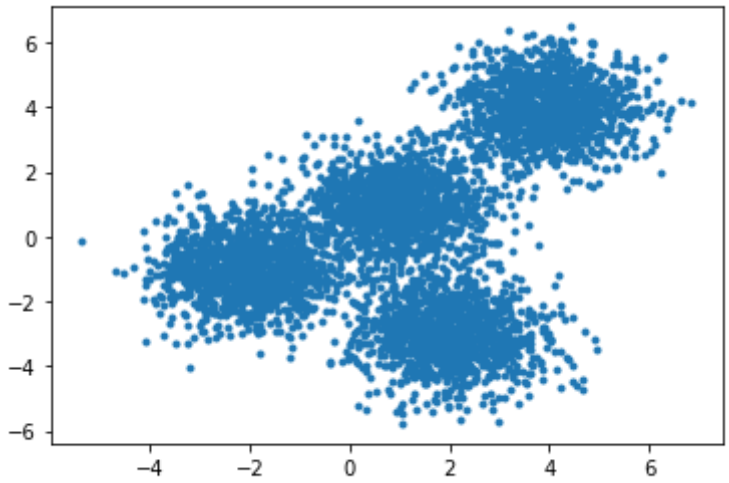
Output

- **X**: Array of shape [n_samples, n_features]. (Feature Matrix)
  - The generated samples.
- **y**: Array of shape [n_samples]. (Response Vector)
  - The integer labels for cluster membership of each sample.

```
X, y = make_blobs(n_samples=5000, centers=[[4,4], [-2, -1], [2, -3], [1, 1]], cluster_std=0.9)
```

Display the scatter plot of the randomly generated data.

```
plt.scatter(X[:, 0], X[:, 1], marker='.')
```



Setting up K-Means ¶

Now that we have our random data, let's set up our K-Means Clustering.

The KMeans class has many parameters that can be used, but we will be using these three:

- **init**: Initialization method of the centroids.
  - Value will be: "k-means++"
  - k-means++: Selects initial cluster centers for k-mean clustering in a smart way to speed up convergence.
- **n_clusters**: The number of clusters to form as well as the number of centroids to generate.
  - Value will be: 4 (since we have 4 centers)
- **n_init**: Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
  - Value will be: 12

Initialize KMeans with these parameters, where the output parameter is called **k_means**.

```
k_means = KMeans(init = "k-means++", n_clusters = 4, n_init = 12)
```

```
k_means.fit(X)
```

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=4, n_init=12, n_jobs=None, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

Now let's grab the labels for each point in the model using KMeans' **.labels_** attribute and save it as **k_means_labels**

```
k_means_labels = k_means.labels_
k_means_labels
```

```
array([0, 3, 3, ..., 1, 0, 0], dtype=int32)
```

We will also get the coordinates of the cluster centers using KMeans' **.cluster_centers_** and save it as **k_means_cluster_centers**

```
k_means_cluster_centers = k_means.cluster_cen
k_means_cluster_centers
```

```
array([[-2.03743147, -0.99782524],
       [ 3.97334234,  3.98758687],
       [ 0.96900523,  0.98370298],
       [ 1.99741008, -3.01666822]])
```

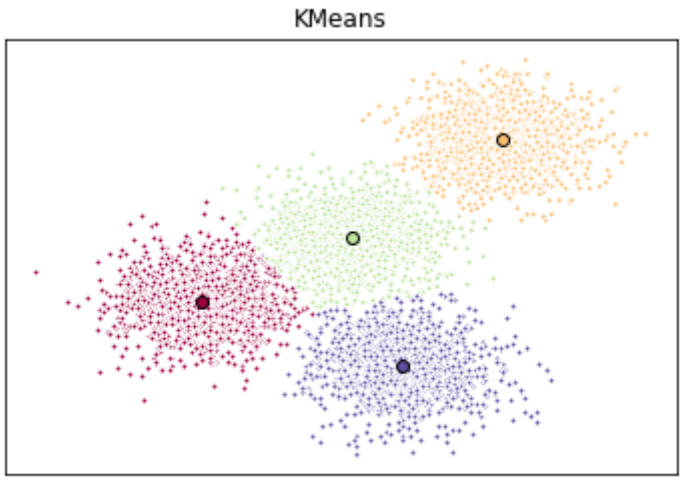
Creating the Visual Plot

So now that we have the random data generated and the KMeans model initialized, let's plot them and see what it looks like!

Please read through the code and comments to understand how to plot the model.

```
# Initialize the plot with the specified dimensions.
fig = plt.figure(figsize=(6, 4))

# Colors uses a color map, which will produce an array of colors based on
# the number of labels there are. We use set(k_means_labels) to get the
# unique labels.
colors = plt.cm.Spectral(np.linspace(0, 1, len(set(k_means_labels))))

# Create a plot
ax = fig.add_subplot(1, 1, 1)

# For loop that plots the data points and centroids.
# k will range from 0-3, which will match the possible clusters that each
# data point is in.
for k, col in zip(range(len([[4,4], [-2, -1], [2, -3], [1, 1]])), colors):

    # Create a list of all data points, where the data poitns that are
    # in the cluster (ex. cluster 0) are labeled as true, else they are
    # labeled as false.
    my_members = (k_means_labels == k)

    # Define the centroid, or cluster center.
    cluster_center = k_means_cluster_centers[k]
```

```
    # Plots the datapoints with color col.
    ax.plot(X[my_members, 0], X[my_members, 1], 'w', markerfacecolor=col, marker='.')

    # Plots the centroids with specified color, but with a darker outline
    ax.plot(cluster_center[0], cluster_center[1], 'o', markerfacecolor=col,  markeredgecolor='k', markersize=6)

# Title of the plot
ax.set_title('KMeans')

# Remove x-axis ticks
ax.set_xticks(())

# Remove y-axis ticks
ax.set_yticks(())

# Show the plot
plt.show()
```



Customer Segmentation with K-Means

Imagine that you have a customer dataset, and you need to apply customer segmentation on this historical data. Customer segmentation is the practice of partitioning a customer base into groups of individuals that have similar characteristics. It is a significant strategy as a business can target these specific groups of customers and effectively allocate marketing resources. For example, one group might contain customers who are high-profit and low-risk, that is, more likely to purchase products, or subscribe for a service. A business task is to retaining those customers. Another group might include customers from non-profit organizations. And so on.

Lets download the dataset. To download the data, we will use `!wget`. To download the data, we will use `!wget` to download it from IBM Object Storage.
Did you know? When it comes to Machine Learning, you will likely be working with large datasets. As a business, where can you host your data? IBM is offering a unique opportunity for businesses, with 10 Tb of IBM Cloud Object Storage: Sign up now for free

Load Data From CSV File

Before you can work with the data, you must use the URL to get the Cust_Segmentation.csv.

```
import pandas as pd
cust_df = pd.read_csv("Cust_Segmentation.csv")
cust_df.head()
```

	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	Address	DebtIncomeRatio
0	1	41	2	6	19	0.124	1.073	0.0	NBA001	6.3
1	2	47	1	26	100	4.582	8.218	0.0	NBA021	12.8
2	3	33	2	10	57	6.111	5.802	1.0	NBA013	20.9
3	4	29	2	4	19	0.681	0.516	0.0	NBA009	6.3
4	5	47	1	31	253	9.308	8.908	0.0	NBA008	7.2

Pre-processing

As you can see, **Address** in this dataset is a categorical variable. k-means algorithm isn't directly applicable to categorical variables because Euclidean distance function isn't really meaningful for discrete variables. So, lets drop this feature and run clustering.

```
df = cust_df.drop('Address', axis=1)
df.head()
```

	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	DebtIncomeRatio
0	1	41	2	6	19	0.124	1.073	0.0	6.3
1	2	47	1	26	100	4.582	8.218	0.0	12.8
2	3	33	2	10	57	6.111	5.802	1.0	20.9
3	4	29	2	4	19	0.681	0.516	0.0	6.3
4	5	47	1	31	253	9.308	8.908	0.0	7.2

Normalizing over the standard deviation

Now let's normalize the dataset. But why do we need normalization in the first place? Normalization is a statistical method that helps mathematical-based algorithms to interpret features with different magnitudes and distributions equally. We use **tandardScaler()** to normalize our dataset.

```
from sklearn.preprocessing import StandardScaler
X = df.values[:,1:]
X = np.nan_to_num(X)
Clus_dataSet = StandardScaler().fit_transform(X)
Clus_dataSet
```

```
array([[ 0.74291541,  0.31212243, -0.37878978, ..., -0.59048916,
        -0.52379654, -0.57652509],
       [ 1.48949049, -0.76634938,  2.5737211 , ...,  1.51296181,
        -0.52379654,  0.39138677],
       [-0.25251804,  0.31212243,  0.2117124 , ...,  0.80170393,
```

Modeling

In our example (if we didn't have access to the k-means algorithm), it would be the same as guessing that each customer group would have certain age, income, education, etc, with multiple tests and experiments. However, using the K-means clustering we can do all this process much easier.

Lets apply k-means on our dataset, and take look at cluster labels.

```
clusterNum = 3
k_means = KMeans(init = "k-means++", n_clusters = clusterNum, n_init = 12)
k_means.fit(X)
labels = k_means.labels_
print(labels)
```

```
[1 2 1 1 0 2 1 2 1 2 2 1 1 1 1 1 1 1 2 1 1 1 1 2 2 2 1 1 2 1 2 1 1 1 1 1 1
 1 1 2 1 2 1 0 1 2 1 1 1 2 2 1 1 2 2 1 1 1 2 1 2 1 2 2 1 1 2 1 1 1 2 2 2 1
 1 1 1 1 2 1 2 2 0 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 2 1 1 1 1 1 1 2 1
 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 2 1 2 1
 1 1 1 1 1 1 2 1 2 2 1 2 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 2 1
```

Insights

We assign the labels to each row in dataframe.

```
df["Clus_km"] = labels
df.head(5)
```

	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	DebtIncomeRatio	Clus_km
0	1	41	2	6	19	0.124	1.073	0.0	6.3	1
1	2	47	1	26	100	4.582	8.218	0.0	12.8	2
2	3	33	2	10	57	6.111	5.802	1.0	20.9	1
3	4	29	2	4	19	0.681	0.516	0.0	6.3	1
4	5	47	1	31	253	9.308	8.908	0.0	7.2	0

We can easily check the centroid values by averaging the features in each cluster.

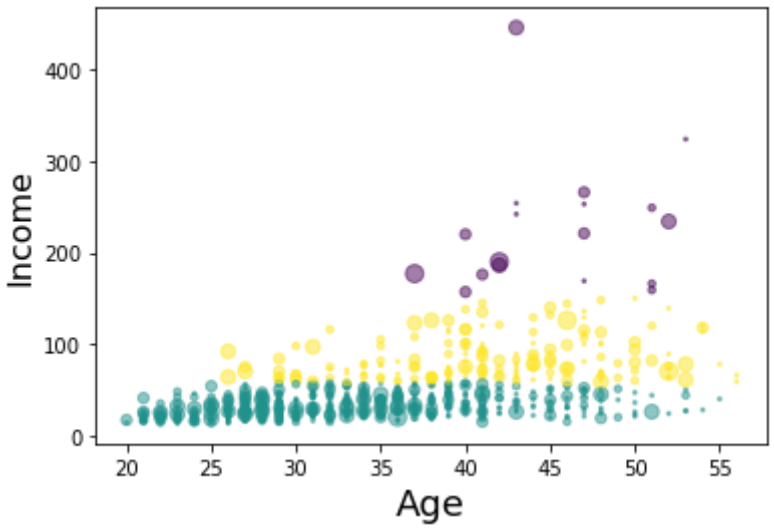
We can easily check the centroid values by averaging the features in each cluster.

```
df.groupby('Clus_km').mean()
```

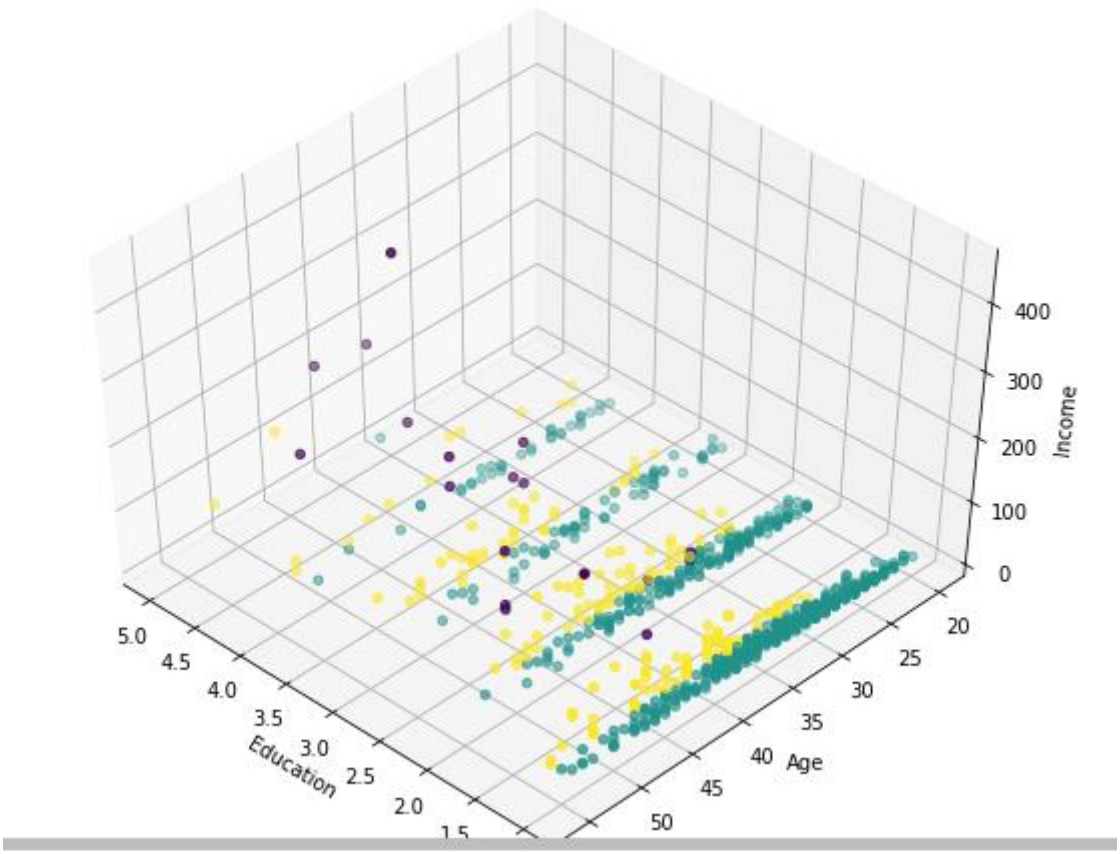
Clus_km	Customer Id	Age	Edu	Years Employed	Income	Card Debt	Other Debt	Defaulted	DebtIncomeRatio
0	410.166667	45.388889	2.666667	19.555556	227.166667	5.678444	10.907167	0.285714	7.322222
1	432.006154	32.967692	1.613846	6.389231	31.204615	1.032711	2.108345	0.284658	10.095385
2	403.780220	41.368132	1.961538	15.252747	84.076923	3.114412	5.770352	0.172414	10.725824

Now, lets look at the distribution of customers based on their age and income:

```
area = np.pi * ( X[:, 1])**2
plt.scatter(X[:, 0], X[:, 3], s=area, c=labels.astype(np.float), alpha=0.5)
plt.xlabel('Age', fontsize=18)
plt.ylabel('Income', fontsize=16)

plt.show()
```



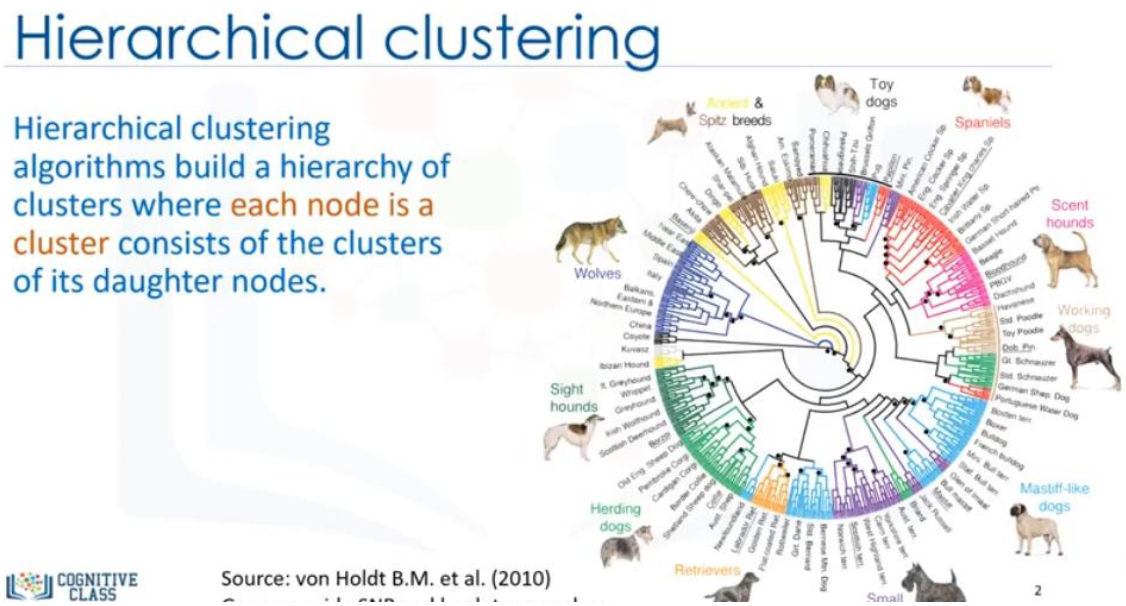

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(1, figsize=(8, 6))
plt.clf()
ax = Axes3D(fig, rect=[0, 0, .95, 1], elev=48, azim=134)

plt.cla()
# plt.ylabel('Age', fontsize=18)
# plt.xlabel('Income', fontsize=16)
# plt.zlabel('Education', fontsize=16)
ax.set_xlabel('Education')
ax.set_ylabel('Age')
ax.set_zlabel('Income')

ax.scatter(X[:, 1], X[:, 0], X[:, 3], c= labels.astype(np.float))
```

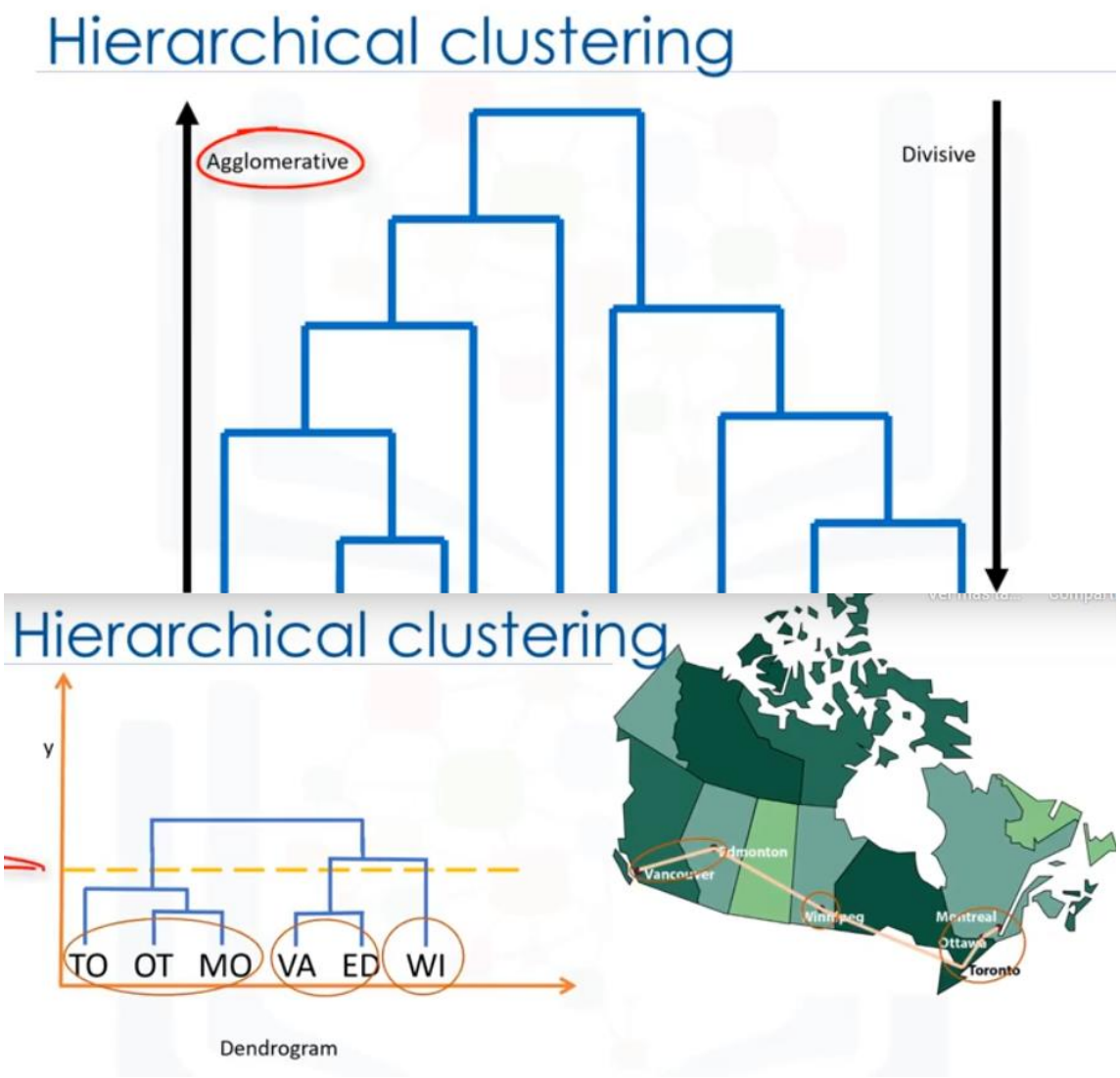


k-means will partition your customers into mutually exclusive groups, for example, into 3 clusters. The customers in each cluster are similar to each other demographically. Now we can create a profile for each group, considering the common characteristics of each cluster. For example, the 3 clusters can be:

- AFFLUENT, EDUCATED AND OLD AGED
- MIDDLE AGED AND MIDDLE INCOME
- YOUNG AND LOW INCOME

HIERACHICAL CLUSTERING





Hello, and welcome! In this video, we'll be covering Hierarchical clustering. So let's get started.

Let's look at this chart. An international team of scientists, led by UCLA biologists, used this dendrogram to report genetic data from more than 900 dogs from 85 breeds -- and more than 200 wild gray wolves worldwide, including populations from North America, Europe, the Middle East, and East Asia.

They used molecular genetic techniques to analyze more than 48,000 genetic markers.

This diagram, shows hierarchical clustering of these animals based on the similarity in their genetic data. Hierarchical clustering algorithms build a hierarchy of clusters where each node is a cluster consisting of the clusters of its daughter nodes.

Strategies for hierarchical clustering generally fall into two types: Divisive and Agglomerative.

Divisive is top-down, so you start with all observations in a large cluster and break it down into smaller pieces. Think about divisive as "dividing" the cluster.

Agglomerative is the opposite of divisive, so it is bottom-up, where each observation starts in its own cluster and pairs of clusters are merged together as they move up the hierarchy.

Agglomeration means to amass or collect things, which is exactly what this does with the cluster.

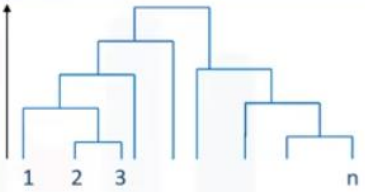
The Agglomerative approach is more popular among data scientists and so it is the main subject of this video.

Let's look at a sample of Agglomerative clustering.

This method builds the hierarchy from the individual elements by progressively merging clusters. In our example, let's say we want to cluster
6 cities in Canada based on their distances from one another.
They are: Toronto, Ottawa, Vancouver, Montreal, Winnipeg, and Edmonton.
We construct a distance matrix at this stage, where the numbers in the row i column j is the distance between the i and j cities. In fact, this table shows the distances between each pair of cities.
The algorithm is started by assigning each city to its own cluster.
So, if we have 6 cities, we have 6 clusters, each containing just one city.
Let's note each city by showing the first two characters of its name.
The first step is to determine which cities -- let's call them clusters from now on -- to merge into a cluster. Usually, we want to take the two closest clusters according to the chosen distance. Looking at the distance matrix, Montreal and
Ottawa are the closest clusters. So, we make a cluster out of them.
Please notice that we just use a simple 1-dimentional distance feature here, but our object can be multi-dimensional, and distance measurement can be either Euclidean, Pearson, average distance, or many others, depending on data type and domain knowledge.
Anyhow, we have to merge these two closest cities in the distance matrix as well.
So, rows and columns are merged as the cluster is constructed.
As you can see in the distance matrix, rows and columns related to Montreal and Ottawa cities are merged as the cluster is constructed. Then, the distances from all cities to this new merged cluster get updated. But how? For example, how do we calculate the distance from Winnipeg to the Ottawa-Montreal cluster? Well, there are different approaches, but let's assume, for example, we just select the distance from the centre of the Ottawa-Montreal cluster to Winnipeg. Updating the distance matrix, we now have
one less cluster. Next, we look for the closest clusters once
again. In this case, Ottawa-Montreal and Toronto
are the closest ones, which creates another cluster.
In the next step, the closest distance is between the Vancouver cluster and the Edmonton cluster. Forming a new cluster, their data in the matrix
table gets updated. Essentially, the rows and columns are merged
as the clusters are merged and the distance updated.
This is a common way to implement this type of clustering, and has the benefit of caching distances between clusters.
In the same way, agglomerative algorithm proceeds by merging clusters.
And we repeat it until all clusters are merged and the tree becomes completed.
It means, until all cities are clustered into a single cluster of size 6.
Hierarchical clustering is typically visualized as a dendrogram as shown on this slide.
Each merge is represented by a horizontal line.
The y-coordinate of the horizontal line is the similarity of the two clusters that were merged, where cities are viewed as singleton clusters.
By moving up from the bottom layer to the top node, a dendrogram allows us to reconstruct the history of merges that resulted in the depicted clustering.
Essentially, Hierarchical clustering does not require a pre-specified number of clusters.
However, in some applications we want a partition of disjoint clusters just as in flat clustering.
In those cases, the hierarchy needs to be cut at some point.
For example here, cutting in a specific level of similarity, we create 3 clusters of similar cities.
This concludes this video. Thanks for watching.

Agglomerative algorithm

1. Create *n* clusters, one for each data point
2. Compute the Proximity Matrix
3. **Repeat**
   i. Merge the two closest clusters
   ii. Update the proximity matrix
4. **Until** only a single cluster remains



$$\begin{bmatrix} 0 & & & & \\ d(2,1) & 0 & & & \\ d(3,1) & d(3,2) & 0 & & \\ \vdots & \vdots & \vdots & & \\ d(n,1) & d(n,2) & \dots & \dots & 0 \end{bmatrix}$$

Similarity/Distance

Patient 1		
Age	BMI	BP
54	190	120

Patient 2		
Age	BMI	BP
50	200	125

Dis (p1,p2)

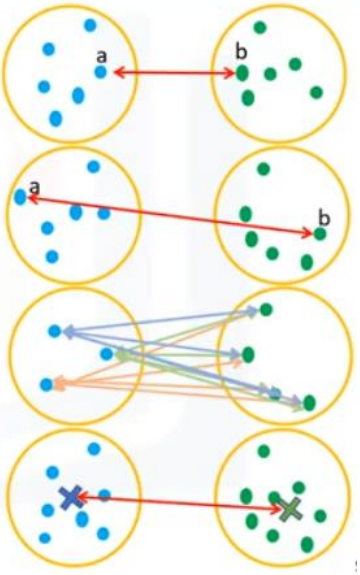
$$= \sqrt{\sum_{i=0}^{n}(x_i - y_i)^2}$$

$$= \sqrt{(54-50)^2+(190-200)^2+(120-125)^2}$$

$$= 11.87$$

Distance between clusters

- Single-Linkage Clustering
  - Minimum distance between clusters
- Complete-Linkage Clustering
  - Maximum distance between clusters
- Average Linkage Clustering
  - Average distance between clusters
- Centroid Linkage Clustering
  - Distance between cluster centroids



COGNITIVE CLASS

Advantages vs. disadvantages

Advantages	Disadvantages
Doesn't required number of clusters to be specified.	Can never undo any previous steps throughout the algorithm.
Easy to implement.	Generally has long runtimes.
Produces a dendrogram, which helps with understanding the data.	Sometimes difficult to identify the number of clusters by the dendrogram.

Hierarchical clustering Vs. *K*-means

K-means	Hierarchical Clustering
1. Much more efficient	1. Can be slow for large datasets
2. Requires the number of clusters to be specified	2. Does not require the number of clusters to run
3. Gives only one partitioning of the data based on the predefined number of clusters	3. Gives more than one partitioning depending on the resolution
4. Potentially returns different clusters each time it is run due to random initialization of centroids	4. Always generates the same clusters

Hello, and welcome! In this video, we'll be covering more details about Hierarchical clustering. Let's get started.
Let's look at Agglomerative algorithm for Hierarchical Clustering. Remember that Agglomerative clustering is a bottom-up approach.

Let's say our dataset has n data points. First, we want to create n clusters, one for each data point. Then each point is assigned as a cluster.

Next, we want to compute the distance/proximity matrix, which will be an n by n table.

After that, we want to iteratively run the following steps until the specified cluster number is reached, or until there is only one cluster left.

First, MERGE the two nearest clusters. (Distances are computed already in the proximity matrix.)

Second, UPDATE the proximity matrix with the new values.

We stop after we've reached the specified number of clusters, or there is only one cluster remaining, with the result stored in a dendrogram. So, in the proximity matrix, we have to measure the distances between clusters, and also merge the clusters that are "nearest."

So, the key operation is the computation of the proximity between the clusters with one point, and also clusters with multiple data points.

At this point, there are a number of key questions that need to be answered.

For instance, "How do we measure the distances between these clusters and How do we define the 'nearest' among clusters?" We also can ask, "Which points do we use?"

First, let's see how to calculate the distance between 2 clusters with 1 point each.

Let's assume that we have a dataset of patients, and we want to cluster them using hierarchy clustering. So, our data points are patients, with a feature set of 3 dimensions. For example, Age, Body Mass Index (or BMI), and Blood Pressure. We can use different distance measurements to calculate the proximity matrix. For instance, Euclidean distance.

So, if we have a dataset of n patients, we can build an n by n dissimilarly-distance matrix. It will give us the distance of clusters with 1 data point. However, as mentioned, we merge clusters in Agglomerative clustering. Now, the question is, "How can we calculate the distance between clusters when there are multiple patients in each cluster?"

We can use different criteria to find the closest clusters, and merge them.

In general, it completely depends on the data type, dimensionality of data, and most importantly, the domain knowledge of the dataset. In fact, different approaches to defining the distance between clusters, distinguish the different algorithms.

As you might imagine, there are multiple ways we can do this.

The first one is called Single-Linkage Clustering. Single linkage is defined as the shortest distance between 2 points in each cluster, such as point "a" and "b".

Next up is Complete-Linkage Clustering. This time, we are finding the longest distance between points in each cluster, such as the distance between point "a" and "b".

The third type of linkage is Average Linkage Clustering, or the mean distance.

This means we're looking at the average distance of each point from one cluster to every point in another cluster. The final linkage type to be reviewed is Centroid Linkage Clustering. Centroid is the average of the feature sets of points in a cluster. This linkage takes into account the centroid of each cluster when determining the minimum distance.

There are 3 main advantages to using hierarchical clustering.

First, we do not need to specify the number of clusters required for the algorithm.

Second, hierarchical clustering is easy to implement.

And third, the dendrogram produced is very useful in understanding the data.

There are some disadvantages as well. First, the algorithm can never undo any previous steps. So for example, the algorithm clusters 2 points, and later on we see that the connection was not a good one, the program cannot undo that

step. Second, the time complexity for the clustering
can result in very long computation times, in comparison with efficient algorithms, such k-Means. Finally, if we have a large dataset, it can
become difficult to determine the correct number of clusters by the dendrogram.
Now, let's compare Hierarchical clustering with k-Means.
K-Means is more efficient for large datasets. In contrast to k-Means, Hierarchical clustering does not require the number of clusters to be specified.
Hierarchical clustering gives more than one partitioning depending on the resolution, whereas k-Means gives only one partitioning of the data.
Hierarchical clustering always generates the same clusters, in contrast with k-Means that returns different clusters each time it is run due to random initialization of centroids.
Thanks for watching!

>>Lab:

[1]:
```
import numpy as np
import pandas as pd
from scipy import ndimage
from scipy.cluster import hierarchy
from scipy.spatial import distance_matrix
from matplotlib import pyplot as plt
from sklearn import manifold, datasets
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets.samples_generator import make_blobs
%matplotlib inline
```

Generating Random Data

We will be generating a set of data using the **make_blobs** class.

Input these parameters into make_blobs:

- **n_samples**: The total number of points equally divided among clusters.
  - Choose a number from 10-1500
- **centers**: The number of centers to generate, or the fixed center locations.
  - Choose arrays of x,y coordinates for generating the centers. Have 1-10 centers (ex. centers=[[1,1], [2,5]])
- **cluster_std**: The standard deviation of the clusters. The larger the number, the further apart the clusters
  - Choose a number between 0.5-1.5

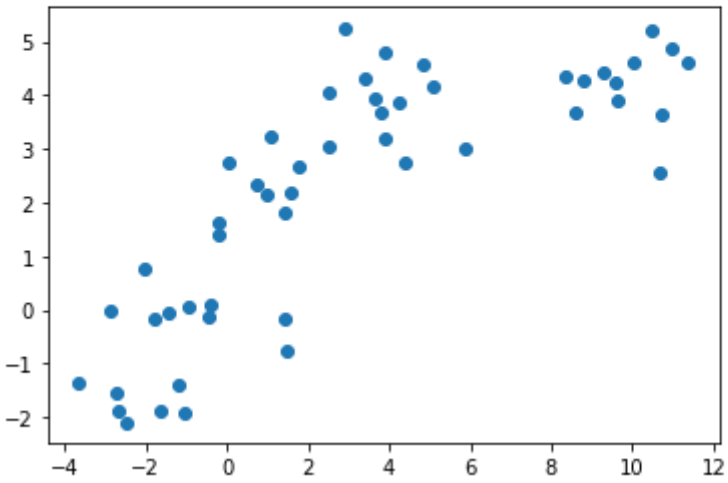
Save the result to **X1** and **y1**.

:
```
X1, y1 = make_blobs(n_samples=50, centers=[[4,4], [-2, -1], [1, 1], [10,4]], cluster_std=0.9)
```

Plot the scatter plot of the randomly generated data

```
plt.scatter(X1[:, 0], X1[:, 1], marker='o')
```

```
<matplotlib.collections.PathCollection at 0x7f69889cf358>
```



Agglomerative Clustering

We will start by clustering the random data points we just created.

The **Agglomerative Clustering** class will require two inputs:

- **n_clusters**: The number of clusters to form as well as the number of centroids to generate.
  - Value will be: 4
- **linkage**: Which linkage criterion to use. The linkage criterion determines which distance to use between sets of observation. The algorithm will merge the pairs of cluster that minimize this criterion.
  - Value will be: 'complete'
  - **Note**: It is recommended you try everything with 'average' as well

Save the result to a variable called **agglom**

```
agglom = AgglomerativeClustering(n_clusters = 4, linkage = 'average')
```

Fit the model with **X2** and **y2** from the generated data above.

```
]: agglom.fit(X1,y1)
```

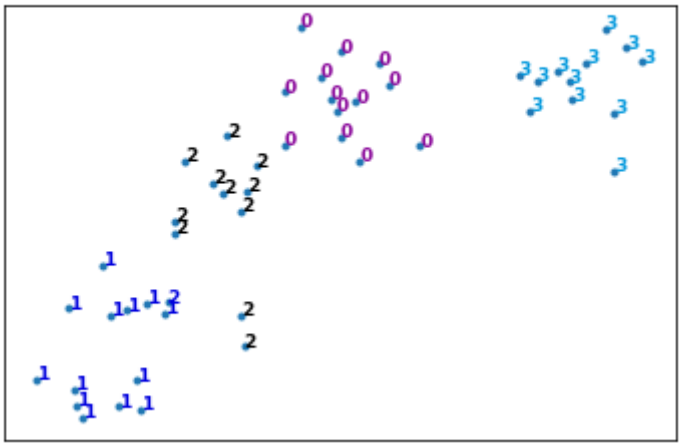
```
]: AgglomerativeClustering(affinity='euclidean', compute_full_tree='auto',
                           connectivity=None, linkage='average', memory=None,
                           n_clusters=4, pooling_func='deprecated')
```

Run the following code to show the clustering!

Remember to read the code and comments to gain more understanding on how the plotting works.

```
# Create a figure of size 6 inches by 4 inches.
plt.figure(figsize=(6,4))

# These two lines of code are used to scale the data points down,
# Or else the data points will be scattered very far apart.

# Create a minimum and maximum range of X1.
x_min, x_max = np.min(X1, axis=0), np.max(X1, axis=0)

# Get the average distance for X1.
X1 = (X1 - x_min) / (x_max - x_min)

# This loop displays all of the datapoints.
for i in range(X1.shape[0]):
    # Replace the data points with their respective cluster value
    # (ex. 0) and is color coded with a colormap (plt.cm.spectral)
    plt.text(X1[i, 0], X1[i, 1], str(y1[i]),
             color=plt.cm.nipy_spectral(agglom.labels_[i] / 10.),
             fontdict={'weight': 'bold', 'size': 9})

# Remove the x ticks, y ticks, x and y axis
plt.xticks([])
plt.yticks([])
#plt.axis('off')
```

```
plt.show()
```



Dendrogram Associated for the Agglomerative Hierarchical Clustering

Remember that a **distance matrix** contains the **distance from each point to every other point of a dataset** .

Use the function **distance_matrix**, which requires **two inputs**. Use the Feature Matrix, **X2** as both inputs and save the distance matrix to a variable called **dist_matrix**

Remember that the distance values are symmetric, with a diagonal of 0's. This is one way of making sure your matrix is correct.

(print out dist_matrix to make sure it's correct)

```
dist_matrix = distance_matrix(X1,X1)
print(dist_matrix)
```

```
[[0.         0.58794359 0.27103729 ... 0.57727761 0.87371431 0.12574668]
 [0.58794359 0.         0.85742914 ... 0.19108683 0.37911917 0.57555213]
```

Using the **linkage** class from hierarchy, pass in the parameters:

- The distance matrix
- 'complete' for complete linkage

Save the result to a variable called **Z**

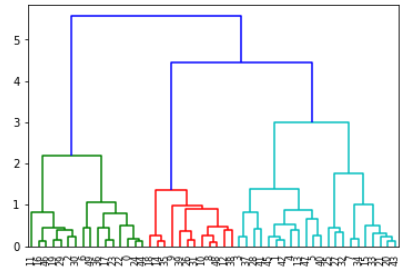
```
Z = hierarchy.linkage(dist_matrix, 'complete')
```

A Hierarchical clustering is typically visualized as a dendrogram as shown in the following cell. Each merge is represented by a horizontal line. The y-coordinate of the horizontal line is the similarity of the two clusters that were merged, where cities are viewed as singleton clusters. By moving up from the bottom layer to the top node, a dendrogram allows us to reconstruct the history of merges that resulted in the depicted clustering.

Next, we will save the dendrogram to a variable called **dendro**. In doing this, the dendrogram will also be displayed. Using the **dendrogram** class from hierarchy, pass in the parameter:

- Z

```
dendro = hierarchy.dendrogram(Z)
```



Clustering on Vehicle dataset

Imagine that an automobile manufacturer has developed prototypes for a new vehicle. Before introducing the new model into its range, the manufacturer want to determine which existing vehicles on the market are most like the prototypes--that is, how vehicles can be grouped, which group is the most similar with the model, and therefore which models they will be competing against.

Our objective here, is to use clustering methods, to find the most distinctive clusters of vehicles. It will summarize the existing vehicles and help manufacture to make decision about new models simply.

Read data

lets read dataset to see what features the manufacturer has collected about the existing models.

```
filename = 'cars_clus.csv'

#Read csv
pdf = pd.read_csv(filename)
print ("Shape of dataset: ", pdf.shape)

pdf.head(5)
```

```
Shape of dataset:  (159, 16)
```

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width	length	curb_wgt	fuel_cap	mpg	lnsales	partition
0	Acura	Integra	16.919	16.360	0.000	21.500	1.800	140.000	101.200	67.300	172.400	2.639	13.200	28.000	2.828	0.0
1	Acura	TL	39.384	19.875	0.000	28.400	3.200	225.000	108.100	70.300	192.900	3.517	17.200	25.000	3.673	0.0
2	Acura	CL	14.114	18.225	0.000	*null*	3.200	225.000	106.900	70.600	192.000	3.470	17.200	26.000	2.647	0.0
3	Acura	RL	8.588	29.725	0.000	42.000	3.500	210.000	114.600	71.400	196.600	3.850	18.000	22.000	2.150	0.0
4	Audi	A4	20.397	22.255	0.000	23.990	1.800	150.000	102.600	68.200	178.000	2.998	16.400	27.000	3.015	0.0

```
print ("Shape of dataset before cleaning: ", pdf.size)
pdf[[ 'sales', 'resale', 'type', 'price', 'engine_s',
       'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap',
       'mpg', 'lnsales']] = pdf[['sales', 'resale', 'type', 'price', 'engine_s',
       'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap',
       'mpg', 'lnsales']].apply(pd.to_numeric, errors='coerce')
pdf = pdf.dropna()
pdf = pdf.reset_index(drop=True)
print ("Shape of dataset after cleaning: ", pdf.size)
pdf.head(5)
```

```
Shape of dataset before cleaning:  2544
Shape of dataset after cleaning:  1872
```

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width	length	curb_wgt	fuel_cap	mpg	lnsales	partition
0	Acura	Integra	16.919	16.360	0.0	21.50	1.8	140.0	101.2	67.3	172.4	2.639	13.2	28.0	2.828	0.0
1	Acura	TL	39.384	19.875	0.0	28.40	3.2	225.0	108.1	70.3	192.9	3.517	17.2	25.0	3.673	0.0
2	Acura	RL	8.588	29.725	0.0	42.00	3.5	210.0	114.6	71.4	196.6	3.850	18.0	22.0	2.150	0.0
3	Audi	A4	20.397	22.255	0.0	23.99	1.8	150.0	102.6	68.2	178.0	2.998	16.4	27.0	3.015	0.0

Feature selection

Lets select our feature set:

```
featureset = pdf[['engine_s',  'horsepow', 'wheelbas', 'width', 'length', 'curb_wgt', 'fuel_cap', 'mpg']]
```

Normalization

Now we can normalize the feature set. **MinMaxScaler** transforms features by scaling each feature to a given range. It is by default (0, 1). That is, this estimator scales and translates each feature individually such that it is between zero and one.

```
from sklearn.preprocessing import MinMaxScaler
x = featureset.values #returns a numpy array
min_max_scaler = MinMaxScaler()
feature_mtx = min_max_scaler.fit_transform(x)
feature_mtx [0:5]
```

```
array([[0.11428571, 0.21518987, 0.18655098, 0.28143713, 0.30625832,
        0.2310559 , 0.13364055, 0.43333333],
       [0.31428571, 0.43037975, 0.3362256 , 0.46107784, 0.5792277 ,
```

Clustering using Scipy

In this part we use Scipy package to cluster the dataset:

First, we calculate the distance matrix.

```
import scipy
leng = feature_mtx.shape[0]
D = scipy.zeros([leng,leng])
for i in range(leng):
    for j in range(leng):
        D[i,j] = scipy.spatial.distance.euclidean(feature_mtx[i], feature_mtx[j])
```

We use **complete** for our case, but feel free to change it to see how the results change.

```
import pylab
import scipy.cluster.hierarchy
Z = hierarchy.linkage(D, 'complete')
```

Essentially, Hierarchical clustering does not require a pre-specified number of clusters. However, in some applications we want a partition of disjoint clusters just as in flat clustering. So you can use a cutting line:

```
from scipy.cluster.hierarchy import fcluster
max_d = 3
clusters = fcluster(Z, max_d, criterion='distance')
clusters
```

```
array([ 1,  5,  5,  6,  5,  4,  6,  5,  5,  5,  5,  5,  4,  4,  5,  1,  6,
        5,  5,  5,  4,  2, 11,  6,  6,  5,  6,  5,  1,  6,  6, 10,  9,  8,
        9,  3,  5,  1,  7,  6,  5,  3,  5,  3,  8,  7,  9,  2,  6,  6,  5,
        4,  2,  1,  6,  5,  2,  7,  5,  5,  5,  4,  4,  3,  2,  6,  6,  5,
        7,  4,  7,  6,  6,  5,  3,  5,  5,  6,  5,  4,  4,  1,  6,  5,  5,
        5,  6,  4,  5,  4,  1,  6,  5,  6,  6,  5,  5,  5,  7,  7,  7,  2,
        2,  1,  2,  6,  5,  1,  1,  1,  7,  8,  1,  1,  6,  1,  1],
      dtype=int32)
```

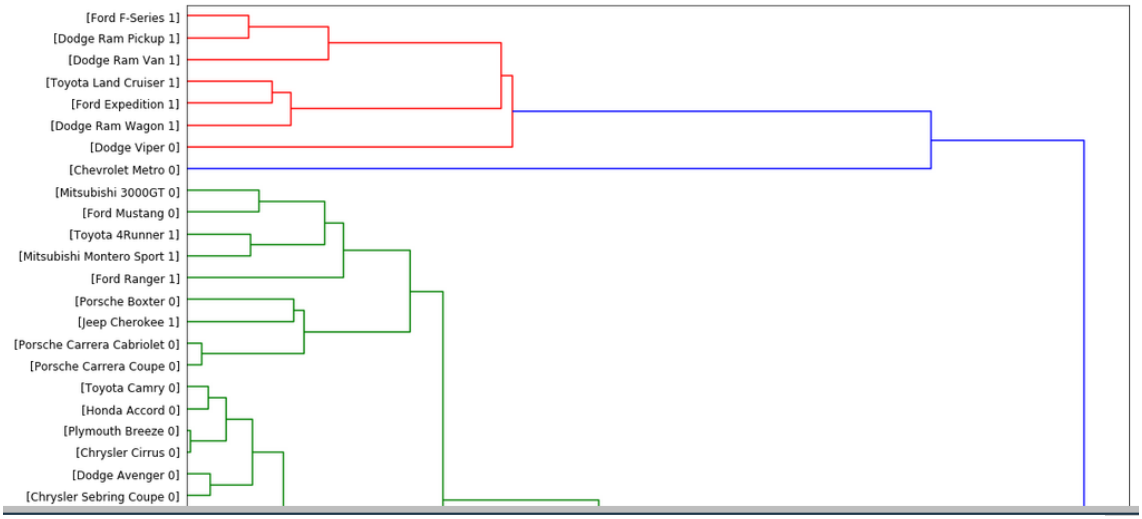
Also, you can determine the number of clusters directly:

```
from scipy.cluster.hierarchy import fcluster
k = 5
clusters = fcluster(Z, k, criterion='maxclust')
clusters
```

```
array([1, 3, 3, 3, 3, 2, 3, 3, 3, 3, 3, 2, 2, 3, 1, 3, 3, 3, 3, 2, 1,
```

Now, plot the dendrogram:

```
fig = pylab.figure(figsize=(18,50))
def llf(id):
    return '[%s %s %s]' % (pdf['manufact'][id], pdf['model'][id], int(float(pdf['type'][id])) )

dendro = hierarchy.dendrogram(Z,  leaf_label_func=llf, leaf_rotation=0, leaf_font_size =12, orientation = 'right')
```



Clustering using scikit-learn

Lets redo it again, but this time using scikit-learn package:

```
dist_matrix = distance_matrix(feature_mtx,feature_mtx)
print(dist_matrix)
```

```
[[0.         0.57777143 0.75455727 ... 0.28530295 0.24917241 0.18879995]
 [0.57777143 0.         0.22798938 ... 0.36087756 0.66346677 0.62201282]
 [0.75455727 0.22798938 0.         ... 0.51727787 0.81786095 0.77930119]
 ...
 [0.28530295 0.36087756 0.51727787 ... 0.         0.41797928 0.35720492]
 [0.24917241 0.66346677 0.81786095 ... 0.41797928 0.         0.15212198]
 [0.18879995 0.62201282 0.77930119 ... 0.35720492 0.15212198 0.        ]]
```

Now, we can use the 'AgglomerativeClustering' function from scikit-learn library to cluster the dataset. The AgglomerativeClustering performs a hierarchical clustering using a bottom up approach. The linkage criteria determines the metric used for the merge strategy:

- Ward minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.
- Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.
- Average linkage minimizes the average of the distances between all observations of pairs of clusters.

```
agglom = AgglomerativeClustering(n_clusters = 6, linkage = 'complete')
agglom.fit(feature_mtx)
agglom.labels_
```

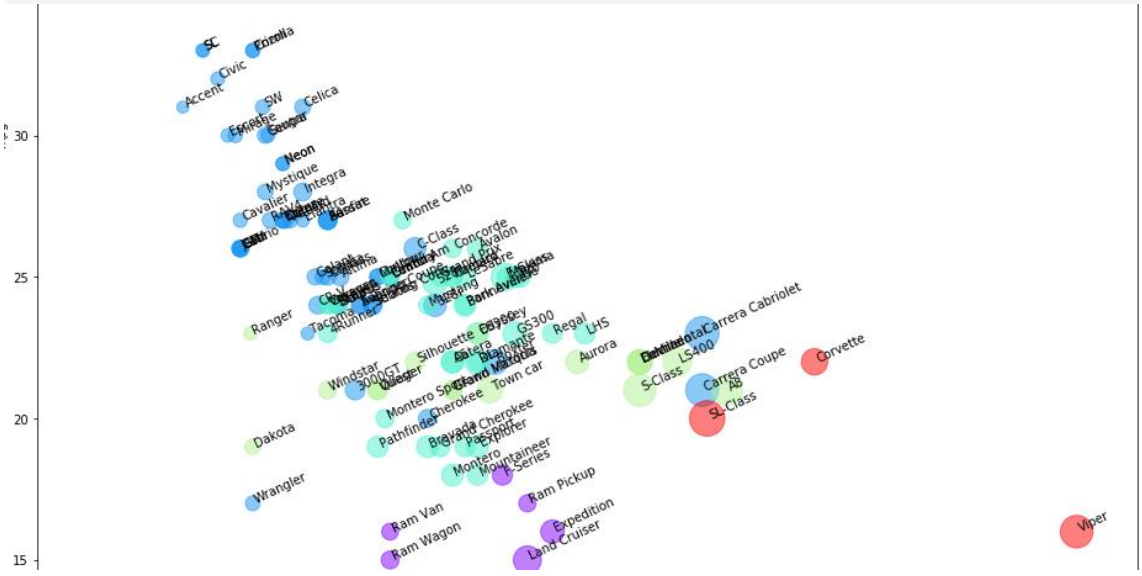
```
array([1, 2, 2, 1, 2, 3, 1, 2, 2, 2, 2, 2, 3, 3, 2, 1, 1, 2, 2, 2, 5, 1,
       4, 1, 1, 2, 1, 2, 1, 1, 1, 5, 0, 0, 0, 3, 2, 1, 2, 1, 2, 3, 2, 3,
       0, 3, 0, 1, 1, 1, 2, 3, 1, 1, 1, 2, 1, 1, 2, 2, 2, 3, 3, 3, 1, 1,
```

And, we can add a new field to our dataframe to show the cluster of each row:

```
pdf['cluster_'] = agglom.labels_
pdf.head()
```

	manufact	model	sales	resale	type	price	engine_s	horsepow	wheelbas	width	length	curb_wgt	fuel_cap	mpg	lnsales	partition	cluster_
0	Acura	Integra	16.919	16.360	0.0	21.50	1.8	140.0	101.2	67.3	172.4	2.639	13.2	28.0	2.828	0.0	1
1	Acura	TL	39.384	19.875	0.0	28.40	3.2	225.0	108.1	70.3	192.9	3.517	17.2	25.0	3.673	0.0	2
2	Acura	RL	8.588	29.725	0.0	42.00	3.5	210.0	114.6	71.4	196.6	3.850	18.0	22.0	2.150	0.0	2
3	Audi	A4	20.397	22.255	0.0	23.99	1.8	150.0	102.6	68.2	178.0	2.998	16.4	27.0	3.015	0.0	1
4	Audi	A6	18.780	23.555	0.0	33.95	2.8	200.0	108.7	76.1	192.0	3.561	18.5	22.0	2.933	0.0	2

```
import matplotlib.cm as cm
n_clusters = max(agglom.labels_)+1
colors = cm.rainbow(np.linspace(0, 1, n_clusters))
cluster_labels = list(range(0, n_clusters))

# Create a figure of size 6 inches by 4 inches.
plt.figure(figsize=(16,14))

for color, label in zip(colors, cluster_labels):
    subset = pdf[pdf.cluster_ == label]
    for i in subset.index:
            plt.text(subset.horsepow[i], subset.mpg[i],str(subset['model'][i]), rotation=25)
    plt.scatter(subset.horsepow, subset.mpg, s= subset.price*10, c=color, label='cluster'+str(label),alpha=0.5)
#    plt.scatter(subset.horsepow, subset.mpg)
plt.legend()
plt.title('Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg')
```



```
pdf.groupby(['cluster_','type'])['cluster_'].count()
```

```
cluster_  type
0         1.0      6
1         0.0     47
          1.0      5
2         0.0     27
          1.0     11
3         0.0     10
          1.0      7
4         0.0      1
5         0.0      3
Name: cluster_, dtype: int64
```

Now we can look at the characterestics of each cluster:

```
agg_cars = pdf.groupby(['cluster_','type'])['horsepow','engine_s','mpg','price'].mean()
agg_cars
```

		horsepow	engine_s	mpg	price
cluster_	**type**				
0	**1.0**	211.666667	4.483333	16.166667	29.024667
1	**0.0**	146.531915	2.246809	27.021277	20.306128
	1.0	145.000000	2.580000	22.200000	17.009200
2	**0.0**	203.111111	3.303704	24.214815	27.750593
	1.0	182.090909	3.345455	20.181818	26.265364
3	**0.0**	256.500000	4.410000	21.500000	42.870400
	1.0	160.571429	3.071429	21.428571	21.527714
4	**0.0**	55.000000	1.000000	45.000000	9.235000
5	**0.0**	365.666667	6.233333	19.333333	66.010000

It is obvious that we have 3 main clusters with the majority of vehicles in those.

Cars:

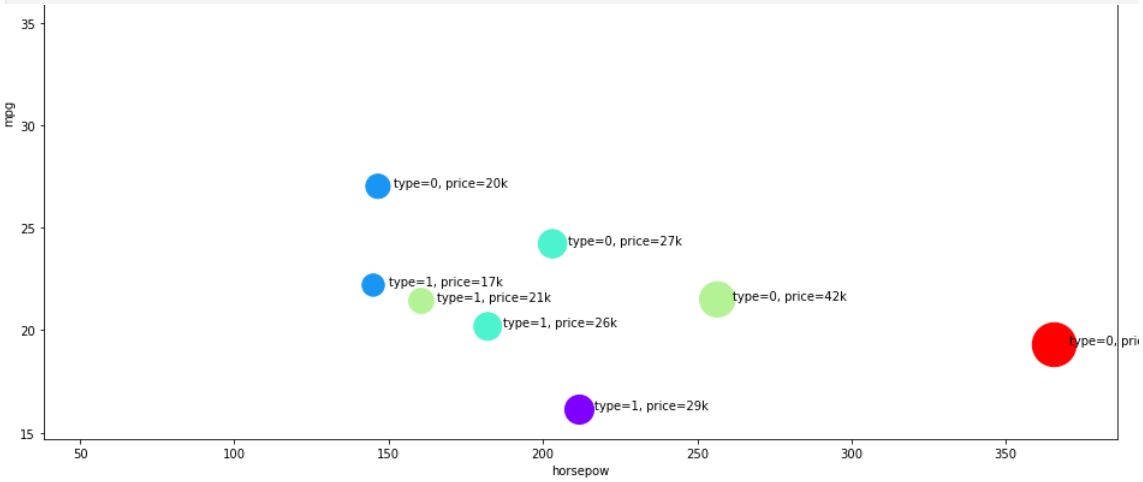
- Cluster 1: with almost high mpg, and low in horsepower.
- Cluster 2: with good mpg and horsepower, but higher price than average.
- Cluster 3: with low mpg, high horsepower, highest price.

Trucks:

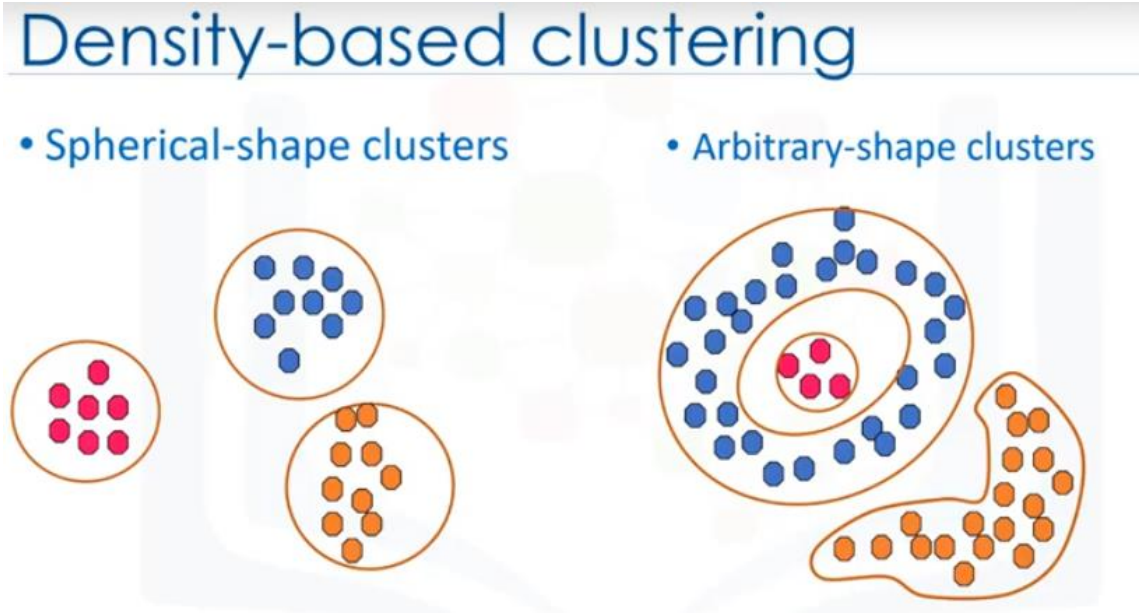
- Cluster 1: with almost highest mpg among trucks, and lowest in horsepower and price.
- Cluster 2: with almost low mpg and medium horsepower, but higher price than average.
- Cluster 3: with good mpg and horsepower, low price.

Please notice that we did not use **type** , and **price** of cars in the clustering process, but Hierarchical clustering could forge the clusters and discriminate them with quite high accuracy.

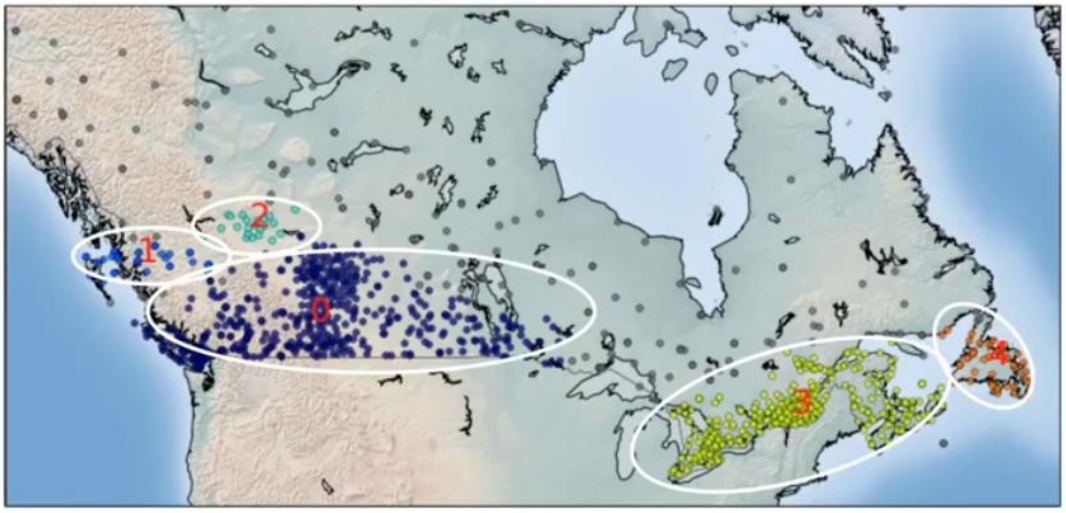
```
plt.figure(figsize=(16,10))
for color, label in zip(colors, cluster_labels):
    subset = agg_cars.loc[(label,),]
    for i in subset.index:
        plt.text(subset.loc[i][0]+5, subset.loc[i][2], 'type='+str(int(i)) + ', price='+str(int(subset.loc[i][3]))+'k')
    plt.scatter(subset.horsepow, subset.mpg, s=subset.price*20, c=color, label='cluster'+str(label))
plt.legend()
plt.title('Clusters')
plt.xlabel('horsepow')
plt.ylabel('mpg')
```



DBSCAN CLUSTERING

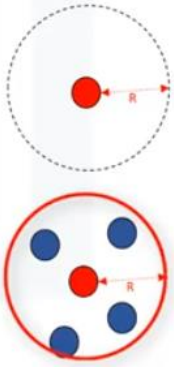


DBSCAN for class identification

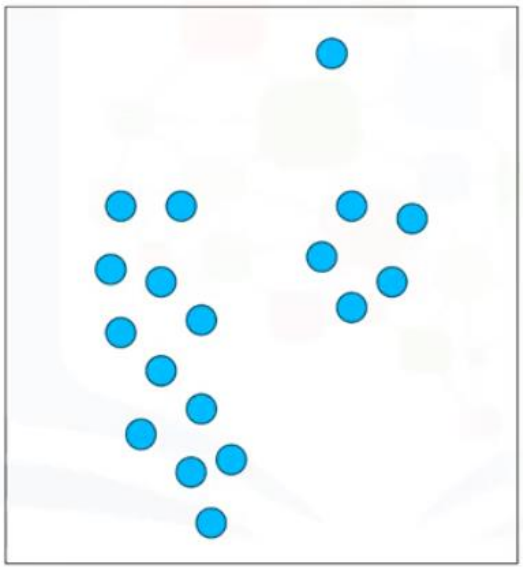


What is DBSCAN?

- DBSCAN (**D**ensity-**B**ased **S**patial **C**lustering of **A**pplications with **N**oise
  - Is one of the most common clustering algorithms
  - Works based on density of objects
- R (**R**adius of neighborhood)
  - Radius (R) that if includes enough number of points within, we call it a dense area
- M (**M**in number of neighbors)
  - The minimum number of data points we want in a neighborhood to define a cluster



How DBSCAN works

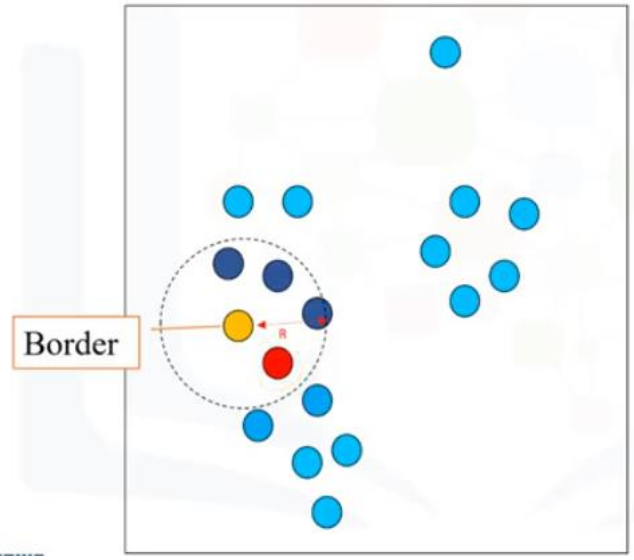


Each point is either:

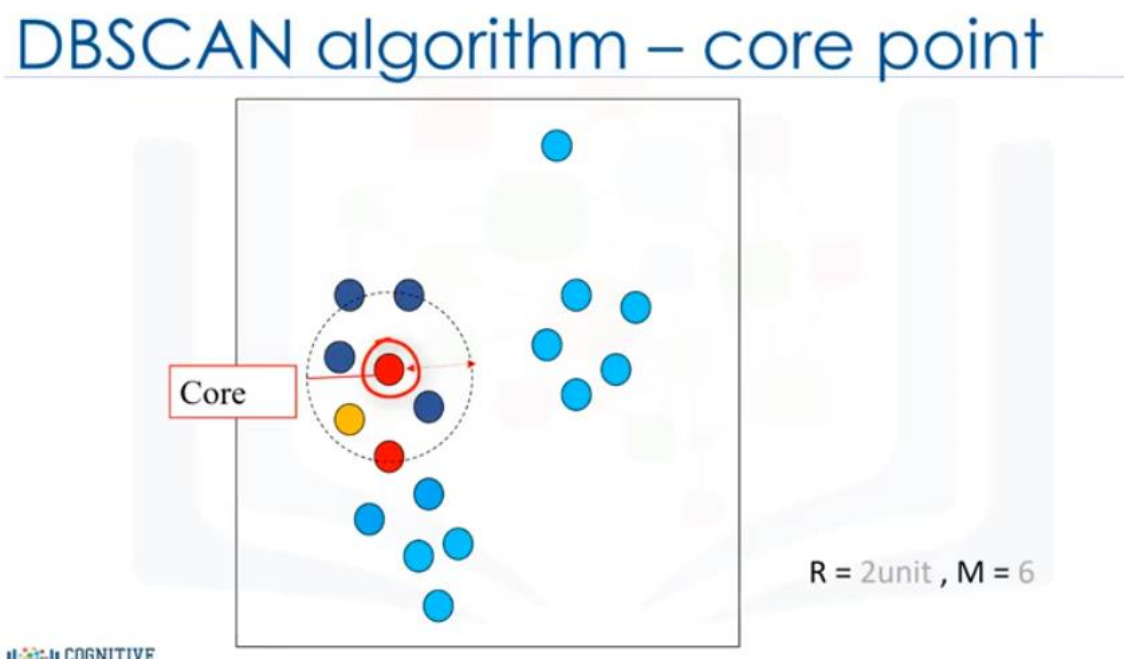
- *core point*
- *border point*
- *outlier point*

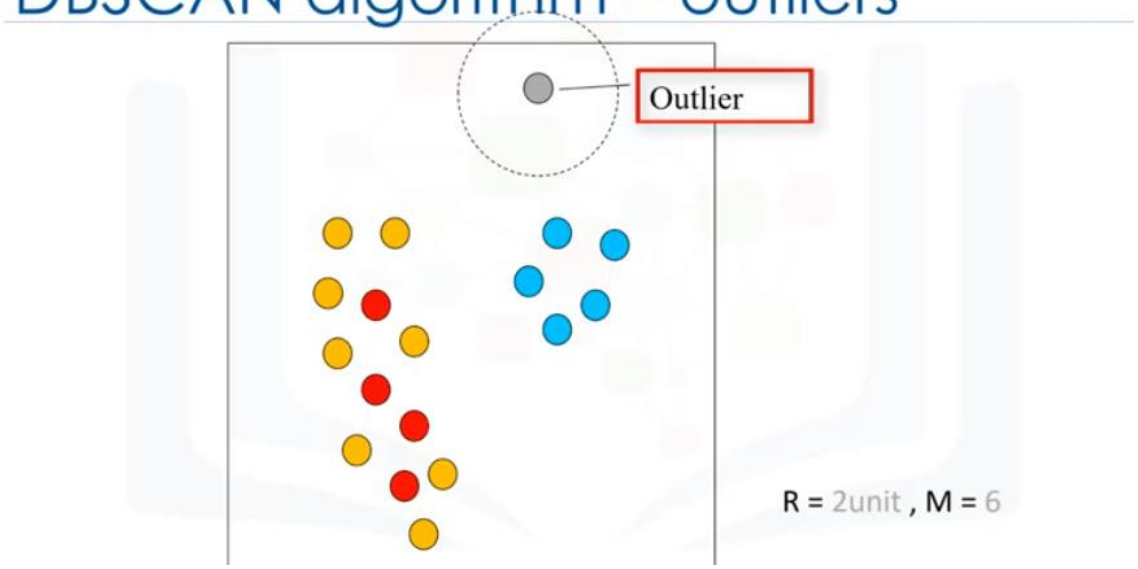
R = 2unit , M = 6

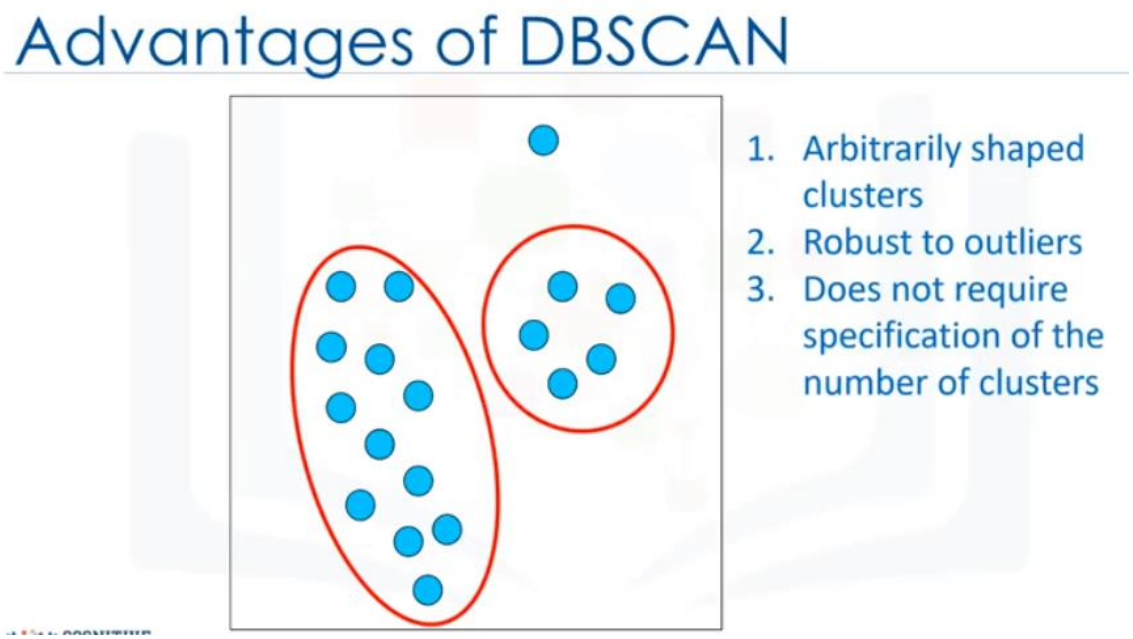
DBSCAN algorithm – border points?



R = 2unit , M = 6


DBSCAN algorithm – core point
Core
R = 2unit , M = 6


DBSCAN algorithm – outliers
Outlier
R = 2unit , M = 6



Hello, and welcome! In this video, we'll be covering DBSCAN,
a density-based clustering algorithm, which is appropriate to use when examining spatial data. So let's get started.
Most of the traditional clustering techniques, such as k-means, hierarchical, and fuzzy clustering, can be used to group data in an un-supervised way.
However, when applied to tasks with arbitrary shape clusters, or clusters within clusters, traditional techniques might not be able to achieve good results.
That is, elements in the same cluster might not share enough similarity -- or the performance may be poor.
Additionally, while partitioning-based algorithms, such as K-Means, may be easy to understand and implement in practice, the algorithm has no notion of outliers.
That is, all points are assigned to a cluster, even if they do not belong in any.
In the domain of anomaly detection, this causes problems as anomalous points will be assigned to the same cluster as "normal" data points. The anomalous points pull the cluster centroid towards them, making it harder to classify them as anomalous points.
In contrast, Density-based clustering locates regions of high density that are separated from one another by regions of low density. Density, in this context, is defined as the number of points within a specified radius. A specific and very popular type of density-based clustering is DBSCAN. DBSCAN is particularly effective for tasks
like class identification on a spatial context. The wonderful attribute of the DBSCAN algorithm is that it can find out any arbitrary shape cluster without getting affected by noise.
For example, this map shows the location of weather stations in Canada.
DBSCAN can be used here to find the group of stations, which show the same weather conditions.
As you can see, it not only finds different arbitrary shaped clusters, it can find the
denser part of data-centered samples by ignoring less-dense areas or noises.
Now, let's look at this clustering algorithm to see how it works.
DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise.
This technique is one of the most common clustering algorithms, which works based on density of
object. DBSCAN works on the idea is that if a particular
point belongs to a cluster, it should be near to lots of other points in that cluster.
It works based on 2 parameters: Radius and Minimum Points.

R determines a specified radius that, if it includes enough points within it, we call
it a "dense area." M determines the minimum number of data points
we want in a neighborhood to define a cluster.
Let's define radius as 2 units. For the sake of simplicity, assume it as radius
of 2 centimeters around a point of interest. Also, let's set the minimum point, or M,
to be 6 points including the point of interest. To see how DBSCAN works, we have to determine
the type of points. Each point in our dataset can be either a
core, border, or outlier point. Don't worry, I'll explain what these points
are, in a moment. But the whole idea behind the DBSCAN algorithm
is to visit each point, and find its type first.
Then we group points as clusters based on their types.
Let's pick a point randomly. First we check to see whether it's a core
data point.
So, what is a core point? A data point is a core point if, within R-neighborhood
of the point, there are at least M points. For example, as there are 6 points in the
2-centimeter neighbor of the red point, we mark this point as a core point.
Ok, what happens if it's NOT a core point?
Let's look at another point. Is this point a core point? No.
As you can see, there are only 5 points in this neighborhood, including the yellow point.
So, what kind of point is this one? In fact, it is a "border" point.
What is a border point? A data point is a BORDER point if:
a. Its neighborhood contains less than M data points, or
b. It is reachable from some core point. Here, Reachability means it is within R-distance
from a core point. It means that even though the yellow point
is within the 2-centimeter neighborhood of the red point, it is not by itself a core
point, because it does not have at least 6 points in its neighborhood.
We continue with the next point. As you can see it is also a core point.
And all points around it, which are not core points, are border points.
Next core point.
And next core point.
Let's take this point. You can see it is not a core point, nor is
it a border point. So, we'd label it as an outlier.
What is an outlier? An outlier is a point that: Is not a core
point, and also, is not close enough to be reachable from a core point.
We continue and visit all the points in the dataset and label them as either Core, Border,
or Outlier.
The next step is to connect core points that are neighbors, and put them in the same cluster.
So, a cluster is formed as at least one core point, plus all reachable core points, plus
all their borders. It simply shapes all the clusters and finds
outliers
as well.
Let's review this one more time to see why DBSCAN is cool.
DBSCAN can find arbitrarily shaped clusters. It can even find a cluster completely surrounded
by a different cluster. DBSCAN has a notion of noise, and is robust
to outliers. On top of that, DBSCAN makes it very practical
for use in many really world problems because it does not require one to specify the number
of clusters, such as K in k-Means.
This concludes this video. Thanks for watching!

>>Lab:

Density-Based Clustering

Most of the traditional clustering techniques, such as k-means, hierarchical and fuzzy clustering, can be used to group data without supervision.

However, when applied to tasks with arbitrary shape clusters, or clusters within cluster, the traditional techniques might be unable to achieve good results. That is, elements in the same cluster might not share enough similarity or the performance may be poor. Additionally, Density-based Clustering locates regions of high density that are separated from one another by regions of low density. Density, in this context, is defined as the number of points within a specified radius.

In this section, the main focus will be manipulating the data and properties of DBSCAN and observing the resulting clustering.

Import the following libraries:

- **numpy as np**
- **DBSCAN** from **sklearn.cluster**
- **make_blobs** from **sklearn.datasets.samples_generator**
- **StandardScaler** from **sklearn.preprocessing**
- **matplotlib.pyplot as plt**

Remember **%matplotlib inline** to display plots

```
# Notice: For visualization of map, you need basemap package.
# if you dont have basemap install on your machine, you can use the following line to install it
# !conda install -c conda-forge  basemap==1.1.0  matplotlib==2.2.2  -y
# Notice: you maight have to refresh your page and re-run the notebook after installation
```

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.datasets.samples_generator import make_blobs
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
%matplotlib inline
```

Data generation

The function below will generate the data points and requires these inputs:

- **centroidLocation**: Coordinates of the centroids that will generate the random data.
  - Example: input: [[4,3], [2,-1], [-1,4]]
- **numSamples**: The number of data points we want generated, split over the number of centroids (# of centroids defined in centroidLocation)
  - Example: 1500
- **clusterDeviation**: The standard deviation between the clusters. The larger the number, the further the spacing.
  - Example: 0.5

```
def createDataPoints(centroidLocation, numSamples, clusterDeviation):
    # Create random data and store in feature matrix X and response vector y.
    X, y = make_blobs(n_samples=numSamples, centers=centroidLocation,
                                cluster_std=clusterDeviation)

    # Standardize features by removing the mean and scaling to unit variance
    X = StandardScaler().fit_transform(X)
    return X, y
```

Use **createDataPoints** with the **3 inputs** and store the output into variables **X** and **y**.

```
X, y = createDataPoints([[4,3], [2,-1], [-1,4]] , 1500, 0.5)
```

Modeling

DBSCAN stands for Density-Based Spatial Clustering of Applications with Noise. This technique is one of the most common clustering algorithms which works based on density of object. The whole idea is that if a particular point belongs to a cluster, it should be near to lots of other points in that cluster.

It works based on two parameters: Epsilon and Minimum Points
Epsilon determine a specified radius that if includes enough number of points within, we call it dense area
minimumSamples determine the minimum number of data points we want in a neighborhood to define a cluster.

```
epsilon = 0.3
minimumSamples = 7
db = DBSCAN(eps=epsilon, min_samples=minimumSamples).fit(X)
labels = db.labels_
labels
```

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

Distinguish outliers

Lets Replace all elements with 'True' in core_samples_mask that are in the cluster, 'False' if the points are outliers.

```
# Firts, create an array of booleans using the labels from db.
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
core_samples_mask
```

```
array([ True,  True,  True, ...,  True,  True,  True])
```

```
# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_clusters_
```

```
3
```

```
# Remove repetition in labels by turning it into a set.
unique_labels = set(labels)
unique_labels
```

```
{-1, 0, 1, 2}
```

Data visualization

```
# Create colors for the clusters.
colors = plt.cm.Spectral(np.linspace(0, 1, len(unique_labels)))
colors
```

```
array([[0.61960784, 0.00392157, 0.25882353, 1.        ],
       [0.99346405, 0.74771242, 0.43529412, 1.        ],
       [0.74771242, 0.89803922, 0.62745098, 1.        ],
       [0.36862745, 0.30980392, 0.63529412, 1.        ]])
```

```
# Plot the points with colors
for k, col in zip(unique_labels, colors):
    if k == -1:
        # Black used for noise.
        col = 'k'

    class_member_mask = (labels == k)

    # Plot the datapoints that are clustered
    xy = X[class_member_mask & core_samples_mask]
    plt.scatter(xy[:, 0], xy[:, 1],s=50, c=col, marker=u'o', alpha=0.5)

    # Plot the outliers
    xy = X[class_member_mask & ~core_samples_mask]
    plt.scatter(xy[:, 0], xy[:, 1],s=50, c=col, marker=u'o', alpha=0.5)
```

Weather Station Clustering using DBSCAN & scikit-learn

DBSCAN is specially very good for tasks like class identification on a spatial context. The wonderful attribute of DBSCAN algorithm is that it can find out any arbitrary shape cluster without getting affected by noise. For example, this following example cluster the location of weather stations in Canada. <Click 1> DBSCAN can be used here, for instance, to find the group of stations which show the same weather condition. As you can see, it not only finds different arbitrary shaped clusters, can find the denser part of data-centered samples by ignoring less-dense areas or noises.

let's start playing with the data. We will be working according to the following workflow:

1. Loading data

- Overview data
- Data cleaning
- Data selection
- Clusteing

2- Load the dataset

We will import the .csv then we creates the columns for year, month and day.

```
import csv
import pandas as pd
import numpy as np

filename='weather-stations20140101-20141231.csv'

#Read csv
pdf = pd.read_csv(filename)
pdf.head(5)
```

	Stn_Name	Lat	Long	Prov	Tm	DwTm	D	Tx	DwTx	Tn	...	DwP	P%N	S_G	Pd	BS	DwBS	BS%	HDD	CDD	Stn_No
0	CHEMAINUS	48.935	-123.742	BC	8.2	0.0	NaN	13.5	0.0	1.0	...	0.0	NaN	0.0	12.0	NaN	NaN	NaN	273.3	0.0	1011500
1	COWICHAN LAKE FORESTRY	48.824	-124.133	BC	7.0	0.0	3.0	15.0	0.0	-3.0	...	0.0	104.0	0.0	12.0	NaN	NaN	NaN	307.0	0.0	1012040
2	LAKE COWICHAN	48.829	-124.052	BC	6.8	13.0	2.8	16.0	9.0	-2.5	...	9.0	NaN	NaN	11.0	NaN	NaN	NaN	168.1	0.0	1012055
3	DISCOVERY ISLAND	48.425	-123.226	BC	NaN	NaN	NaN	12.5	0.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1012475

3-Cleaning

Lets remove rows that dont have any value in the **Tm** field.

```
pdf = pdf[pd.notnull(pdf["Tm"])]
pdf = pdf.reset_index(drop=True)
pdf.head(5)
```

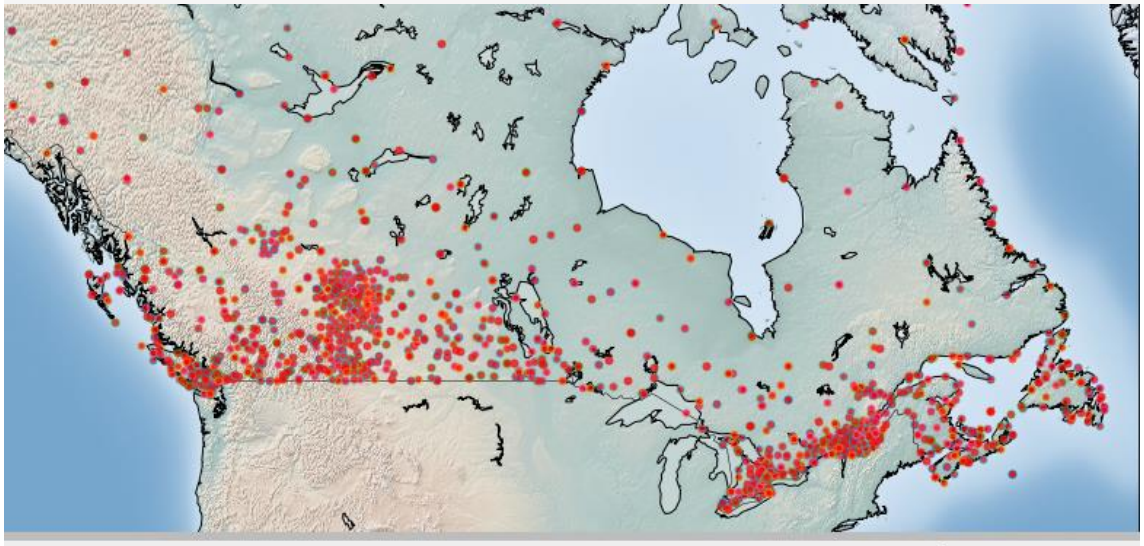
	Stn_Name	Lat	Long	Prov	Tm	DwTm	D	Tx	DwTx	Tn	...	DwP	P%N	S_G	Pd	BS	DwBS	BS%	HDD	CDD	Stn_No
0	CHEMAINUS	48.935	-123.742	BC	8.2	0.0	NaN	13.5	0.0	1.0	...	0.0	NaN	0.0	12.0	NaN	NaN	NaN	273.3	0.0	1011500
1	COWICHAN LAKE FORESTRY	48.824	-124.133	BC	7.0	0.0	3.0	15.0	0.0	-3.0	...	0.0	104.0	0.0	12.0	NaN	NaN	NaN	307.0	0.0	1012040
2	LAKE COWICHAN	48.829	-124.052	BC	6.8	13.0	2.8	16.0	9.0	-2.5	...	9.0	NaN	NaN	11.0	NaN	NaN	NaN	168.1	0.0	1012055
3	DUNCAN KELVIN CREEK	48.735	-123.728	BC	7.7	2.0	3.4	14.5	2.0	-1.0	...	2.0	NaN	NaN	11.0	NaN	NaN	NaN	267.7	0.0	1012573
4	ESQUIMALT HARBOUR	48.432	-123.439	BC	8.8	0.0	NaN	13.1	0.0	1.9	...	8.0	NaN	NaN	12.0	NaN	NaN	NaN	258.6	0.0	1012710

4-Visualization

Visualization of stations on map using basemap package. The matplotlib basemap toolkit is a library for plotting 2D data on maps in Python. Basemap does not do any plotting on it's own, but provides the facilities to transform coordinates to a map projections.

Please notice that the size of each data points represents the average of maximum temperature for each station in a year.

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams['figure.figsize'] = (14,10)

llon=-140
ulon=-50
llat=40
pdf = pdf[(pdf['Long'] > llon) & (pdf['Long'] < ulon) & (pdf['Lat'] > llat) &(pdf['Lat'] < ulat)]

my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
# my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To collect data based on stations

xs,ys = my_map(np.asarray(pdf.Long), np.asarray(pdf.Lat))
pdf['xm']= xs.tolist()
#Visualization1
for index,row in pdf.iterrows():
#   x,y = my_map(row.Long, row.Lat)
   my_map.plot(row.xm, row.ym,markerfacecolor =([1,0,0]),  marker='o', markersize= 5, alpha = 0.75)
#plt.text(x,y,stn)
plt.show()
```



5- Clustering of stations based on their location i.e. Lat & Lon

DBSCAN form sklearn library can runs DBSCAN clustering from vector array or distance matrix. In our case, we pass it the Numpy array Clus_dataSet to find core samples of high density and expands clusters from them.

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler
sklearn.utils.check_random_state(1000)
Clus_dataSet = pdf[['xm','ym']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = StandardScaler().fit_transform(Clus_dataSet)

# Compute DBSCAN
db = DBSCAN(eps=0.15, min_samples=10).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
pdf["Clus_Db"]=labels

realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)
clusterNum = len(set(labels))
```

```
# A sample of clusters
pdf[["Stn_Name","Tx","Tm","Clus_Db"]].head(5)
```

	Stn_Name	Tx	Tm	Clus_Db
0	CHEMAINUS	13.5	8.2	0
1	COWICHAN LAKE FORESTRY	15.0	7.0	0
2	LAKE COWICHAN	16.0	6.8	0
3	DUNCAN KELVIN CREEK	14.5	7.7	0
4	ESQUIMALT HARBOUR	13.1	8.8	0

As you can see for outliers, the cluster label is -1

```
set(labels)
```

```
{-1, 0, 1, 2, 3, 4}
```

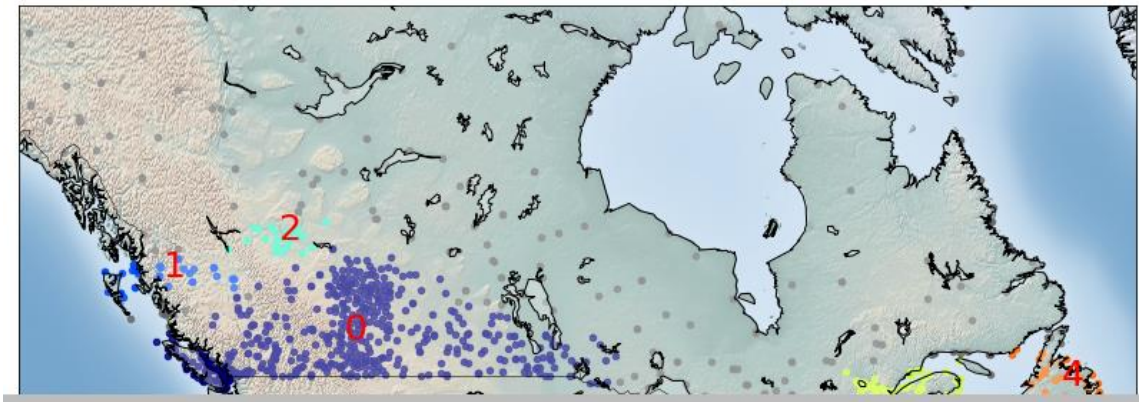
6- Visualization of clusters based on location

Now, we can visualize the clusters using basemap:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams['figure.figsize'] = (14,10)

my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
#my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
my_map.shadedrelief()

# To create a color map
colors = plt.get_cmap('jet')(np.linspace(0.0, 1.0, clusterNum))
```

```
#Visualization1
for clust_number in set(labels):
    c=(([0.4,0.4,0.4]) if clust_number == -1 else colors[np.int(clust_number)])
    clust_set = pdf[pdf.Clus_Db == clust_number]
    my_map.scatter(clust_set.xm, clust_set.ym, color =c,  marker='o', s= 20, alpha = 0.85)
    if clust_number != -1:
        cenx=np.mean(clust_set.xm)
        ceny=np.mean(clust_set.ym)
        plt.text(cenx,ceny,str(clust_number), fontsize=25, color='red',)
        print ("Cluster "+str(clust_number)+', Avg Temp: '+ str(np.mean(clust_set.Tm)))
```

```
Cluster 0, Avg Temp: -5.538747553816046
Cluster 1, Avg Temp: 1.9526315789473685
Cluster 2, Avg Temp: -9.195652173913045
Cluster 3, Avg Temp: -15.300833333333333
Cluster 4, Avg Temp: -7.769047619047619
```



7- Clustering of stations based on their location, mean, max, and min Temperature

In this section we re-run DBSCAN, but this time on a 5-dimensional dataset:

```
from sklearn.cluster import DBSCAN
import sklearn.utils
from sklearn.preprocessing import StandardScaler
sklearn.utils.check_random_state(1000)
Clus_dataSet = pdf[['xm','ym','Tx','Tm','Tn']]
Clus_dataSet = np.nan_to_num(Clus_dataSet)
Clus_dataSet = StandardScaler().fit_transform(Clus_dataSet)

# Compute DBSCAN
db = DBSCAN(eps=0.3, min_samples=10).fit(Clus_dataSet)
core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True
labels = db.labels_
pdf["Clus_Db"]=labels

realClusterNum=len(set(labels)) - (1 if -1 in labels else 0)
clusterNum = len(set(labels))
```

```
# A sample of clusters
pdf[["Stn_Name","Tx","Tm","Clus_Db"]].head(5)
```

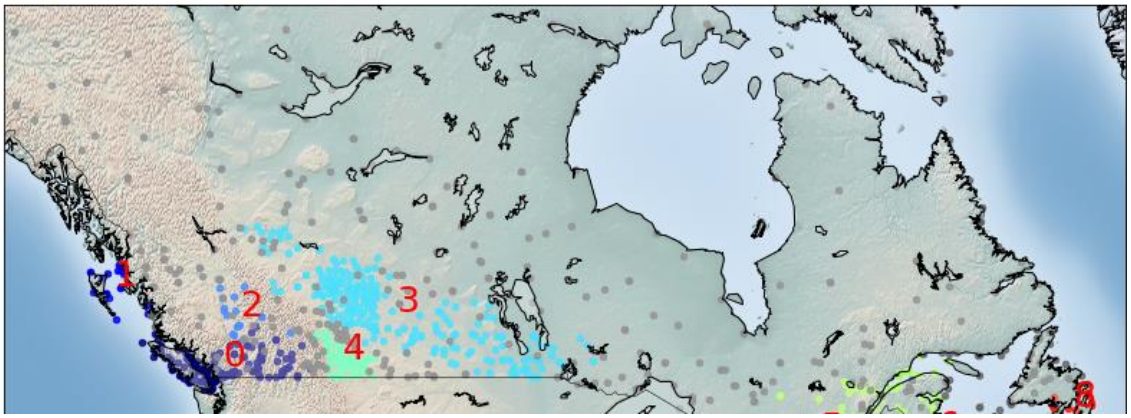
	Stn_Name	Tx	Tm	Clus_Db
0	CHEMAINUS	13.5	8.2	0
1	COWICHAN LAKE FORESTRY	15.0	7.0	0
2	LAKE COWICHAN	16.0	6.8	0
3	DUNCAN KELVIN CREEK	14.5	7.7	0
4	ESQUIMALT HARBOUR	13.1	8.8	0

8- Visualization of clusters based on location and Temperture

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
from pylab import rcParams
%matplotlib inline
rcParams['figure.figsize'] = (14,10)

my_map = Basemap(projection='merc',
            resolution = 'l', area_thresh = 1000.0,
            llcrnrlon=llon, llcrnrlat=llat, #min longitude (llcrnrlon) and latitude (llcrnrlat)
            urcrnrlon=ulon, urcrnrlat=ulat) #max longitude (urcrnrlon) and latitude (urcrnrlat)

my_map.drawcoastlines()
my_map.drawcountries()
#my_map.drawmapboundary()
my_map.fillcontinents(color = 'white', alpha = 0.3)
```

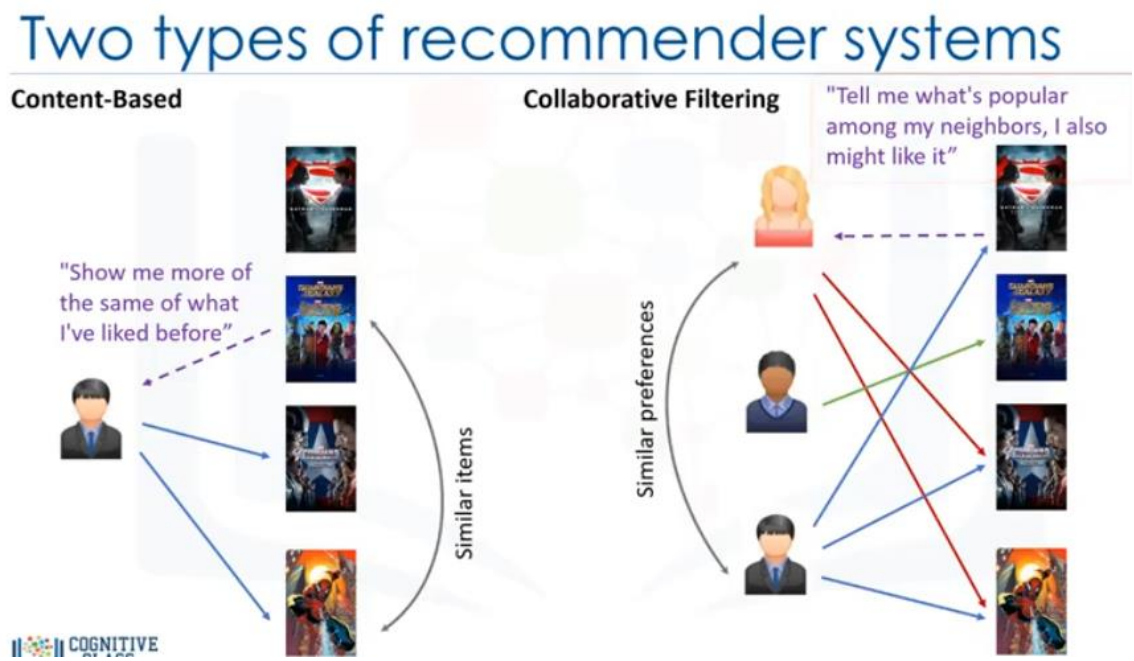
```
# To create a color map
colors = plt.get_cmap('jet')(np.linspace(0.0, 1.0, clusterNum))



#Visualization1
for clust_number in set(labels):
    c=(([0.4,0.4,0.4]) if clust_number == -1 else colors[np.int(clust_number)])
    clust_set = pdf[pdf.Clus_Db == clust_number]
    my_map.scatter(clust_set.xm, clust_set.ym, color =c,  marker='o', s= 20, alpha = 0.85)
    if clust_number != -1:
        cenx=np.mean(clust_set.xm)
        ceny=np.mean(clust_set.ym)
        plt.text(cenx,ceny,str(clust_number), fontsize=25, color='red',)
        print ("Cluster "+str(clust_number)+', Avg Temp: '+ str(np.mean(clust_set.Tm)))
```

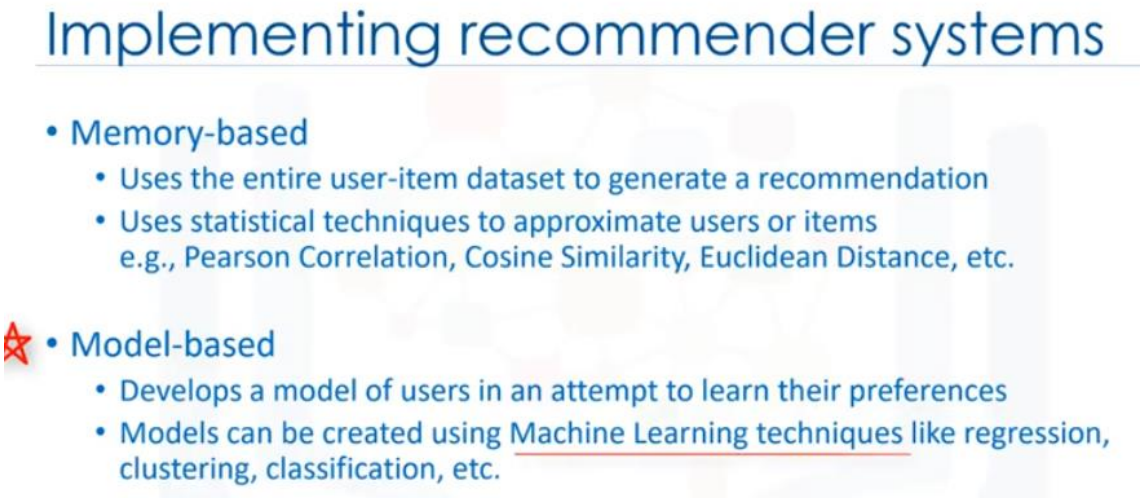
```
Cluster 0, Avg Temp: 6.221192052980132
Cluster 1, Avg Temp: 6.790000000000001
Cluster 2, Avg Temp: -0.49411764705882344
Cluster 3, Avg Temp: -13.87720930232558
Cluster 4, Avg Temp: -4.186274509803922
Cluster 5, Avg Temp: -16.301503759398496
Cluster 6, Avg Temp: -13.599999999999998
Cluster 7, Avg Temp: -9.753333333333334
Cluster 8, Avg Temp: -4.258333333333334
```



MODULE 5 – RECOMMENDER SYSTEMS

RECOMEMDER SYSTEMS





Hello, and welcome! In this video, we'll be going through a
quick introduction to recommendation systems. So, let's get started.
Even though peoples' tastes may vary, they generally follow patterns.
By that, I mean that there are similarities in the things that people tend to like … or
another way to look at it, is that people tend to like things in the same category or
things that share the same characteristics. For example, if you've recently purchased
a book on Machine Learning in Python and you've enjoyed reading it, it's very likely that
you'll also enjoy reading a book on Data Visualization.
People also tend to have similar tastes to those of the people they're close to in
their lives. Recommender systems try to capture these patterns
and similar behaviors, to help predict what else you might like.
Recommender systems have many applications that I'm sure you're already familiar
with. Indeed, Recommender systems are usually at
play on many websites. For example, suggesting books on Amazon and
movies on Netflix. In fact, everything on Netflix's website
is driven by customer selection. If a certain movie gets viewed frequently

enough, Netflix's recommender system ensures that that movie gets an increasing number of recommendations.

Another example can be found in a daily-use mobile app, where a recommender engine is used to recommend anything from where to eat, or, what job to apply to.

On social media, sites like Facebook or LinkedIn, regularly recommend friendships.

Recommender systems are even used to personalize your experience on the web.

For example, when you go to a news platform website, a recommender system will make note of the types of stories that you clicked on and make recommendations on which types of stories you might be interested in reading, in future.

There are many of these types of examples and they are growing in number every day.

So, let's take a closer look at the main benefits of using a recommendation system.

One of the main advantages of using recommendation systems is that users get a broader exposure

to many different products they might be interested in.

This exposure encourages users towards continual usage or purchase of their product.

Not only does this provide a better experience for the user but it benefits the service provider, as well, with increased potential revenue and better security for its customers.

There are generally 2 main types of recommendation systems: Content-based and collaborative filtering.

The main difference between each, can be summed up by the type of statement that a consumer

might make. For instance, the main paradigm of a Content-based

recommendation system is driven by the statement: "Show me more of the same of what I've liked before."

Content-based systems try to figure out what

a user's favorite aspects of an item are, and then make recommendations on items that share those aspects.

Collaborative filtering is based on a user saying, "Tell me what's popular among my neighbors because I might like it too." Collaborative filtering techniques find similar groups of users, and provide recommendations based on similar tastes within that group.

In short, it assumes that a user might be interested in what similar users are interested in.

Also, there are Hybrid recommender systems,

which combine various mechanisms.

In terms of implementing recommender systems, there are 2 types: Memory-based and Model-based.

In memory-based approaches, we use the entire user-item dataset to generate a recommendation

system. It uses statistical techniques to approximate

users or items. Examples of these techniques include: Pearson

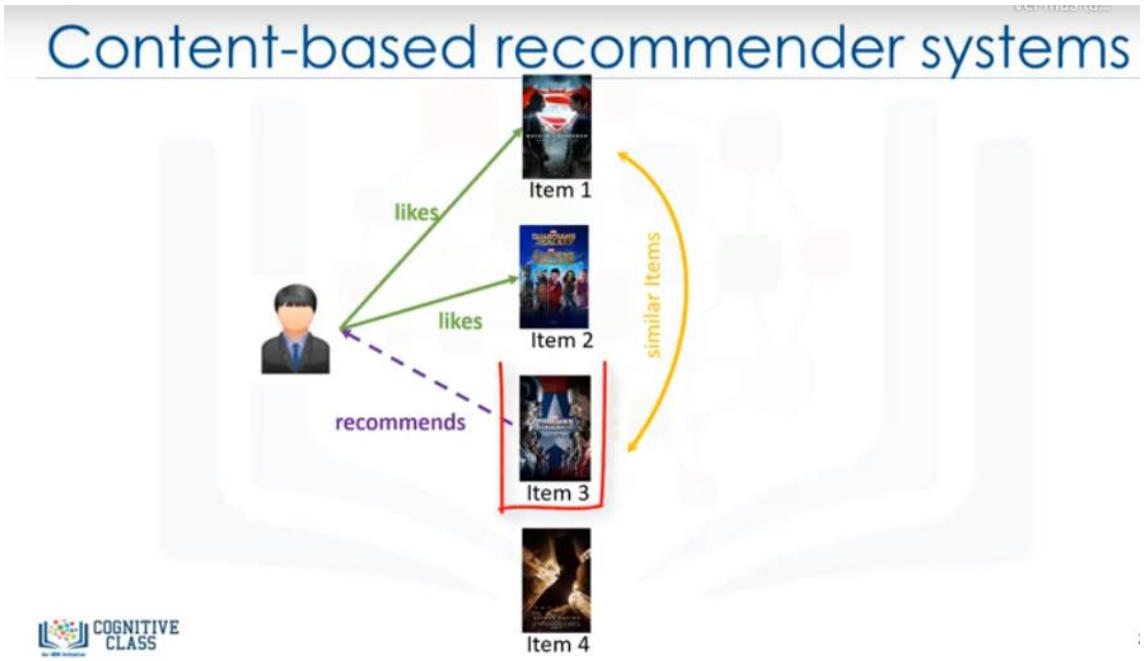
Correlation, Cosine Similarity and Euclidean Distance, among others.

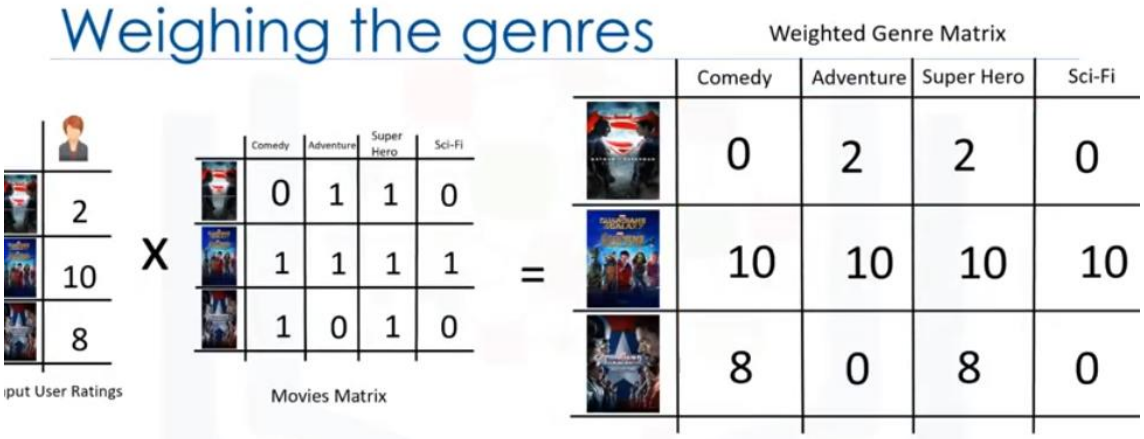
In model-based approaches, a model of users is developed in an attempt to learn their preferences. Models can be created using Machine Learning

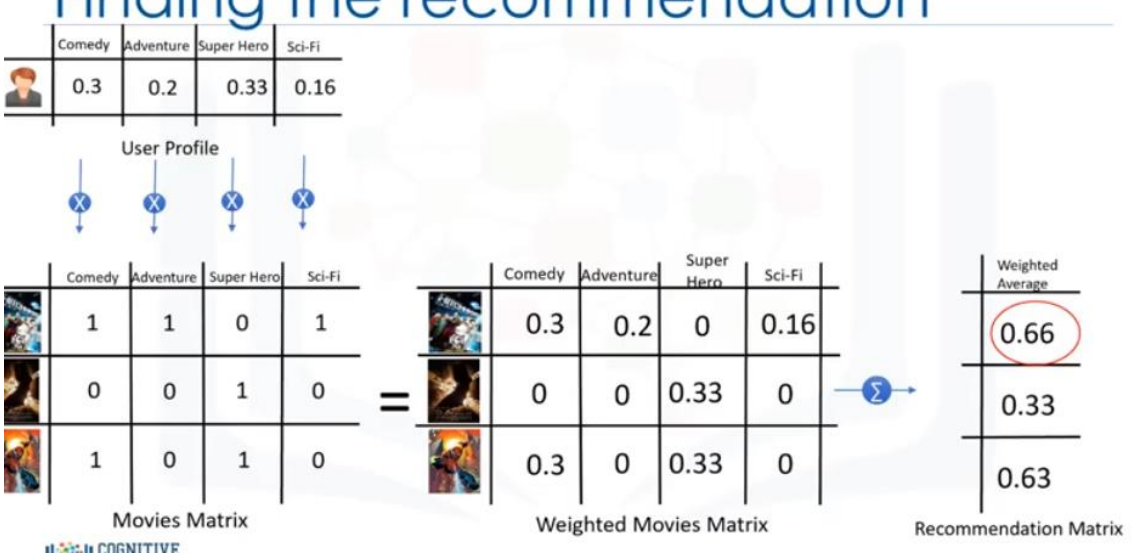
techniques like regression, clustering, classification, and so on.

This is the end of our video. Thanks for watching!

CONTENT-BASED







Hello, and welcome! In this video, we'll be covering content-based recommendation systems. So let's get started.

A content-based recommendation system tries to recommend items to users, based on their

profile. The user's profile revolves around that
user's preferences and tastes. It is shaped based on user ratings, including
the number of times that user has clicked on different items or perhaps, even liked
those items. The recommendation process is based on the
similarity between those items. Similarity, or closeness of items, is measured
based on the similarity in the content of those items.
When we say content, we're talking about things like the item's category, tag, genre,
and so on. For example, if we have 4 movies, and if the
user likes or rates the first 2 items, and if item 3 is similar to item 1, in terms of
their genre, the engine will also recommend item 3 to the user.
In essence, this is what content-based recommender system engines do.
Now, let's dive into a content-based recommender system to see how it works.
Let's assume we have a dataset of only 6 movies.
This dataset shows movies that our user has watched, and also the genre of each of the
movies. For example, "Batman versus Superman"
is in the Adventure, Super Hero genre. And "Guardians of the Galaxy" is in Comedy,
Adventure, Super Hero, and Science-Fiction genres.
Let's say the user has watched and rated 3 movies so far and she has given a rating
of 2 out of 10 to the first movie, 10 out of 10 to the second movie, and an 8 out of
10 to the third. The task of the recommender engine is to recommend
one of the 3 candidate movies to this user. Or, in other words, we want to predict what
the user's possible rating would be, of the 3 candidate movies if she were to watch them.
To achieve this, we have to build the user profile.
First, we create a vector to show the user's ratings for the movies that she's already
watched. We call it "input user ratings." Then, we encode the movies through the
"One Hot Encoding" approach. Genre of movies are used here as a feature set.
We use the first 3 movies to make this matrix,
which represents the movie 'feature-set' matrix.
If we multiply these 2 matrices, we can get the "weighted feature set" for the movies.
Let's take a look at the result. This matrix is also called the "Weighted
Genre Matrix," and represents the interests of the user for each genre based on the movies
that she's watched. Now, given the weighted genre matrix, we can
shape the profile of our active user. Essentially, we can aggregate the weighted
genres, and then normalize them to find the user profile.
It clearly indicates that she likes "super hero" movies more than other genres.
We use this profile to figure out what movie is proper to recommend to this user.
Recall that we also had 3 candidate movies for recommendation, that haven't been watched
by the user. We encode these movies as well.
Now we're in the position where we have to figure out which of them is most suited
to be recommended to the user.
To do this, we simply multiply the user-profile matrix by the candidate movie matrix, which
results in the "weighted movies" matrix. It shows the weight of each genre, with respect
to the user profile. Now, if we aggregate these weighted ratings,
we get the active user's possible interest-level in these 3 movies.
In essence, it's our "recommendation" list, which we can sort to rank the movies,
and recommend them to the user. For example, we can say that the "Hitchhiker's
Guide to the Galaxy" has the highest score in our list, and is proper to recommend to
the user. Now you can come back and fill the predicted
ratings for the user.
So, to recap what we've discussed so far, the recommendation in a content-based system,
is based on user's tastes, and the content or feature set items.
Such a model is very efficient. However, in some cases it doesn't work.

For example, assume that we have a movie in the “drama” genre, which the user has never watched. So, this genre would not be in her profile.
Therefore, she’ll only get recommendations related to genres that are already in her profile, and the recommender engine may never recommend any movie within other genres. This problem can be solved by other types of recommender systems such as "Collaborative Filtering.”
Thanks for watching!

>>Lab:

Acquiring the Data

To acquire and extract the data, simply run the following Bash scripts:
Dataset acquired from GroupLens. Lets download the dataset. To download the data, we will use `!wget` . To download the data, we will use `!wget` to download it from IBM Object Storage.
Did you know? When it comes to Machine Learning, you will likely be working with large datasets. As a business, where can you host your data? IBM is offering a unique opportunity for businesses, with 10 Tb of IBM Cloud Object Storage: Sign up now for free

```
!wget -O moviedataset.zip https://s3-api.us-geo.objectstorage.softlayer.net/cf-courses-data/CognitiveClass/ML0101ENv3/labs/moviedataset.zip
print('unziping ...')
!unzip -o -j moviedataset.zip
```

Preprocessing

First, let's get all of the imports out of the way:

```
#Dataframe manipulation library
import pandas as pd
#Math functions, we'll only need the sqrt function so let's import only that
from math import sqrt
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

Now let's read each file into their Dataframes:

```
#Storing the movie information into a pandas dataframe
movies_df = pd.read_csv('movies.csv')
#Storing the user information into a pandas dataframe
ratings_df = pd.read_csv('ratings.csv')
#Head is a function that gets the first N rows of a dataframe. N's default is 5.
movies_df.head()
```

	movieId	title	genres
0	1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
1	2	Jumanji (1995)	Adventure|Children|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama|Romance
4	5	Father of the Bride Part II (1995)	Comedy

Let's also remove the year from the **title** column by using pandas' replace function and store in a new **year** column.

```
#Using regular expressions to find a year stored between parentheses
#We specify the parantheses so we don't conflict with movies that have years in their titles
movies_df['year'] = movies_df.title.str.extract('(\(\d\d\d\d\))',expand=False)
#Removing the parentheses
movies_df['year'] = movies_df.year.str.extract('(\d\d\d\d)',expand=False)
#Removing the years from the 'title' column
movies_df['title'] = movies_df.title.str.replace('(\(\d\d\d\d\))', '')
#Applying the strip function to get rid of any ending whitespace characters that may have appeared
movies_df['title'] = movies_df['title'].apply(lambda x: x.strip())
movies_df.head()
```

	movieId	title	genres	year
0	1	Toy Story	Adventure|Animation|Children|Comedy|Fantasy	1995
1	2	Jumanji	Adventure|Children|Fantasy	1995
2	3	Grumpier Old Men	Comedy|Romance	1995
3	4	Waiting to Exhale	Comedy|Drama|Romance	1995
4	5	Father of the Bride Part II	Comedy	1995

With that, let's also split the values in the **Genres** column into a **list of Genres** to simplify future use. This can be achieved by applying Python's split string function on the correct column.

```
#Every genre is separated by a | so we simply have to call the split function on |
movies_df['genres'] = movies_df.genres.str.split('|')
movies_df.head()
```

	movieId	title	genres	year
0	1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995
1	2	Jumanji	[Adventure, Children, Fantasy]	1995
2	3	Grumpier Old Men	[Comedy, Romance]	1995
3	4	Waiting to Exhale	[Comedy, Drama, Romance]	1995
4	5	Father of the Bride Part II	[Comedy]	1995

Since keeping genres in a list format isn't optimal for the content-based recommendation system technique, we will use the One Hot Encoding technique to convert the list of genres to a vector where each column corresponds to one possible value of the feature. This encoding is needed for feeding categorical data. In this case, we store every different genre in columns that contain either 1 or 0. 1 shows that a movie has that genre and 0 shows that it doesn't. Let's also store this dataframe in another variable since genres won't be important for our first recommendation system.

```
#Copying the movie dataframe into a new one since we won't need to use the genre information in our first case.
moviesWithGenres_df = movies_df.copy()

#For every row in the dataframe, iterate through the list of genres and place a 1 into the corresponding column
for index, row in movies_df.iterrows():
    for genre in row['genres']:
        moviesWithGenres_df.at[index, genre] = 1
#Filling in the NaN values with 0 to show that a movie doesn't have that column's genre
```

Next, let's look at the ratings dataframe.

```
ratings_df.head()
```

	userId	movieId	rating	timestamp
0	1	169	2.5	1204927694
1	1	2471	3.0	1204927438
2	1	48516	5.0	1204927435
3	2	2571	3.5	1436165433
4	2	109487	4.0	1436165496

Every row in the ratings dataframe has a user id associated with at least one movie, a rating and a timestamp showing when they reviewed it. We won't be needing the timestamp column, so let's drop it to save on memory.

```
#Drop removes a specified row or column from a dataframe
ratings_df = ratings_df.drop('timestamp', 1)
```

	userId	movieId	rating
0	1	169	2.5
1	1	2471	3.0
2	1	48516	5.0
3	2	2571	3.5
4	2	109487	4.0

Now, let's take a look at how to implement **Content-Based** or **Item-Item recommendation systems**. This technique attempts to figure out what a user's favourite aspects of an item is, and then recommends items that present those aspects. In our case, we're going to try to figure out the input's favorite genres from the movies and ratings given.

Let's begin by creating an input user to recommend movies to:

Notice: To add more movies, simply increase the amount of elements in the **userInput**. Feel free to add more in! Just be sure to write it in with capital letters and if a movie starts with a "The", like "The Matrix" then write it in like this: 'Matrix, The' .

```
userInput = [
            {'title':'Breakfast Club, The', 'rating':5},
            {'title':'Toy Story', 'rating':3.5},
            {'title':'Jumanji', 'rating':2},
            {'title':"Pulp Fiction", 'rating':5},
            {'title':'Akira', 'rating':4.5}
         ]
inputMovies = pd.DataFrame(userInput)
inputMovies
```

	title	rating
0	Breakfast Club, The	5.0
1	Toy Story	3.5
2	Jumanji	2.0
3	Pulp Fiction	5.0
4	Akira	4.5

Add movieId to input user

With the input complete, let's extract the input movies's ID's from the movies dataframe and add them into it.

We can achieve this by first filtering out the rows that contain the input movies' title and then merging this subset with the input dataframe. We also drop unnecessary columns for the input to save memory space.

```
#Filtering out the movies by title
inputId = movies_df[movies_df['title'].isin(inputMovies['title'].tolist())]
#Then merging it so we can get the movieId. It's implicitly merging it by title.
inputMovies = pd.merge(inputId, inputMovies)
#Dropping information we won't use from the input dataframe
inputMovies = inputMovies.drop('genres', 1).drop('year', 1)
#Final input dataframe
#If a movie you added in above isn't here, then it might not be in the original
#dataframe or it might spelled differently, please check capitalisation.
inputMovies
```

	movieId	title	rating
0	1	Toy Story	3.5
1	2	Jumanji	2.0
2	296	Pulp Fiction	5.0
3	1274	Akira	4.5

We're going to start by learning the input's preferences, so let's get the subset of movies that the input has watched from the Dataframe containing genres defined with binary values.

```
#Filtering out the movies from the input
userMovies = moviesWithGenres_df[moviesWithGenres_df['movieId'].isin(inputMovies['movieId'].tolist())]
userMovies
```

	movieId	title	genres	year	Adventure	Animation	Children	Comedy	Fantasy	Romance	...	Horror	Mystery	Sci-Fi	IMAX	Documentary	War	Musical
0	1	Toy Story	[Adventure, Animation, Children, Comedy, Fantasy]	1995	1.0	1.0	1.0	1.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	2	Jumanji	[Adventure, Children, Fantasy]	1995	1.0	0.0	1.0	0.0	1.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0
293	296	Pulp Fiction	[Comedy, Crime, Drama, Thriller]	1994	0.0	0.0	0.0	1.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0

We'll only need the actual genre table, so let's clean this up a bit by resetting the index and dropping the movieId, title, genres and year columns.

```
#Resetting the index to avoid future issues
userMovies = userMovies.reset_index(drop=True)
#Dropping unnecessary issues due to save memory and to avoid issues
userGenreTable = userMovies.drop('movieId', 1).drop('title', 1).drop('genres', 1).drop('year', 1)
userGenreTable
```

	Adventure	Animation	Children	Comedy	Fantasy	Romance	Drama	Action	Crime	Thriller	Horror	Mystery	Sci-Fi	IMAX	Documentary	War	M
0	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
1	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
2	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	
3	1.0	1.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	
4	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

Now we're ready to start learning the input's preferences!

To do this, we're going to turn each genre into weights. We can do this by using the input's reviews and multiplying them into the input's genre table and then summing up the resulting table by column. This operation is actually a dot product between a matrix and a vector, so we can simply accomplish by calling Pandas's "dot" function.

```
inputMovies['rating']
```

```
0    3.5
1    2.0
2    5.0
3    4.5
4    5.0
Name: rating, dtype: float64
```

```
#Dot produt to get weights
userProfile = userGenreTable.transpose().dot(inputMovies['rating'])
#The user profile
userProfile
```

```
Adventure            10.0
Animation             8.0
Children              5.5
Comedy               13.5
```

Now, we have the weights for every of the user's preferences. This is known as the User P preferences.

Let's start by extracting the genre table from the original dataframe:

```
#Now let's get the genres of every movie in our original dataframe
genreTable = moviesWithGenres_df.set_index(moviesWithGenres_df['movieId'])
#And drop the unnecessary information
genreTable = genreTable.drop('movieId', 1).drop('title', 1).drop('genres', 1)
genreTable.head()
```

movieId	Adventure	Animation	Children	Comedy	Fantasy	Romance	Drama	Action	Cr
1	1.0	1.0	1.0	1.0	1.0	0.0	0.0	0.0	
2	1.0	0.0	1.0	0.0	1.0	0.0	0.0	0.0	
3	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0	
4	0.0	0.0	0.0	1.0	0.0	1.0	1.0	0.0	

Mod

```
genreTable.shape
```

```
(34208, 20)
```

With the input's profile and the complete list of movies and their genres in hand, we're going to take the weighted average of every movie based on the input profile and recommend the top twenty movies that most satisfy it.

```
#Multiply the genres by the weights and then take the weighted average
recommendationTable_df = ((genreTable*userProfile).sum(axis=1))/(userProfile.sum())
recommendationTable_df.head()
```

```
movieId
1    0.594406
2    0.293706
3    0.188811
4    0.328671
5    0.188811
dtype: float64
```

```
#Sort our recommendations in descending order
recommendationTable_df = recommendationTable_df.sort_values(ascending=False)
#Just a peek at the values
recommendationTable_df.head()
```

```
movieId
5018      0.748252
26093     0.734266
27344     0.720280
148775    0.685315
6902      0.678322
dtype: float64
```

Now here's the recommendation table!

```
#The final recommendation table
movies_df.loc[movies_df['movieId'].isin(recommendationTable_df.head(20).keys())]
```

```
#The final recommendation table
movies_df.loc[movies_df['movieId'].isin(recommendationTable_df.head(20).keys())]
```

	movieId	title	genres	year
664	673	Space Jam	[Adventure, Animation, Children, Comedy, Fanta...	1996
1824	1907	Mulan	[Adventure, Animation, Children, Comedy, Drama...	1998
2902	2987	Who Framed Roger Rabbit?	[Adventure, Animation, Children, Comedy, Crime...	1988
4923	5018	Motorama	[Adventure, Comedy, Crime, Drama, Fantasy, Mys...	1991
6793	6902	Interstate 60	[Adventure, Comedy, Drama, Fantasy, Mystery, S...	2002
8605	26093	Wonderful World of the Brothers Grimm, The	[Adventure, Animation, Children, Comedy, Drama...	1962

Advantages and Disadvantages of Content-Based Filtering ¶

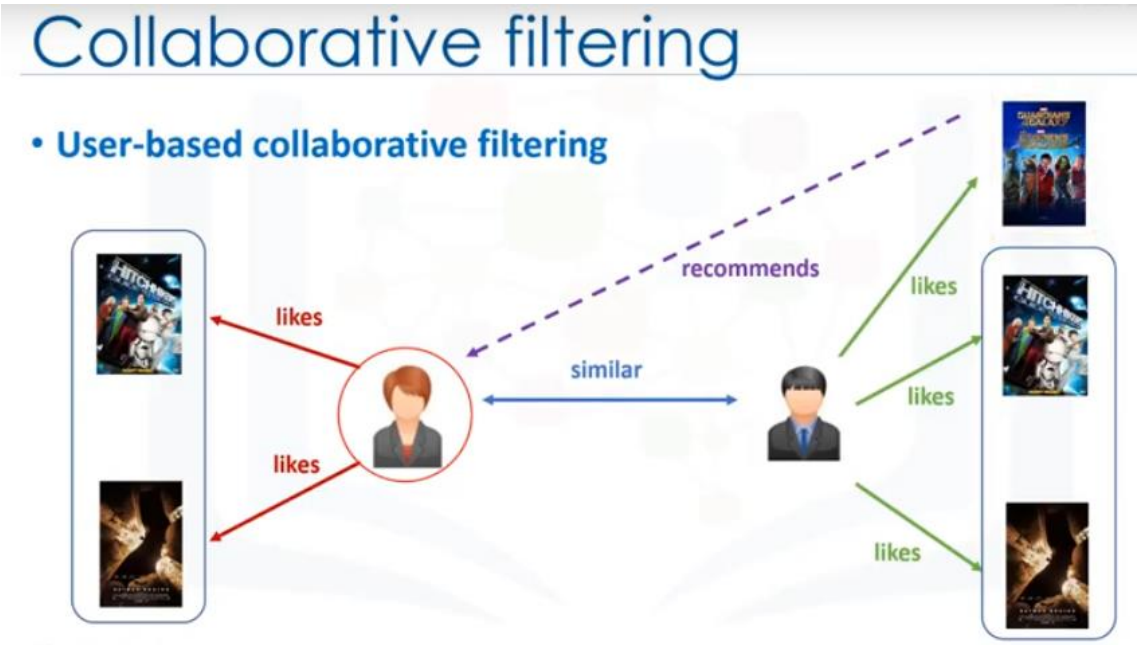
Advantages

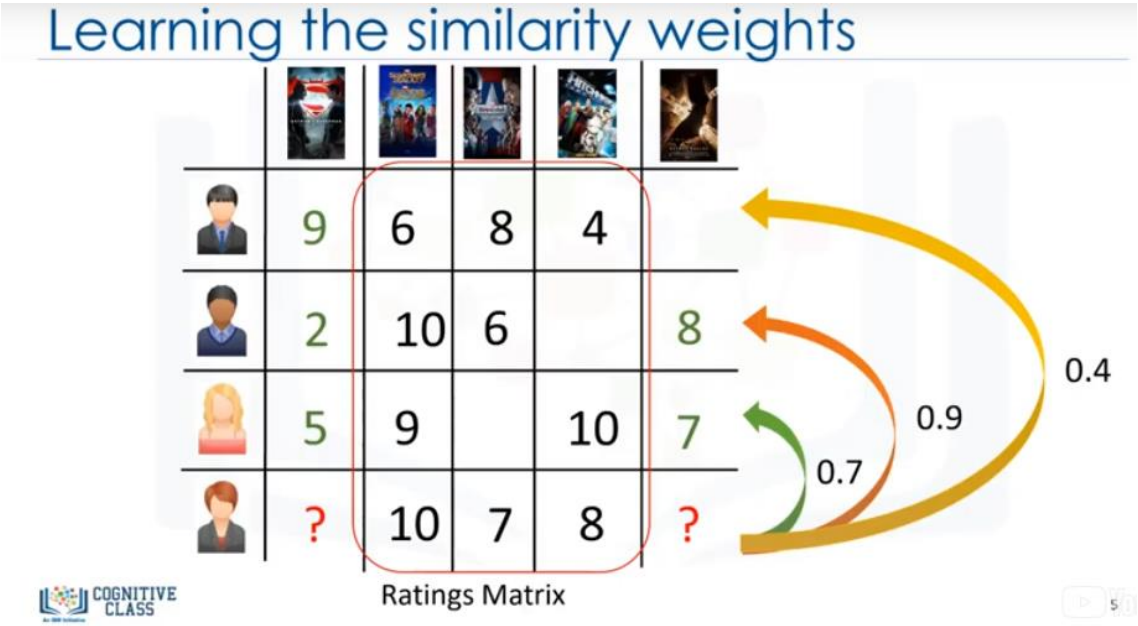
- Learns user's preferences
- Highly personalized for the user

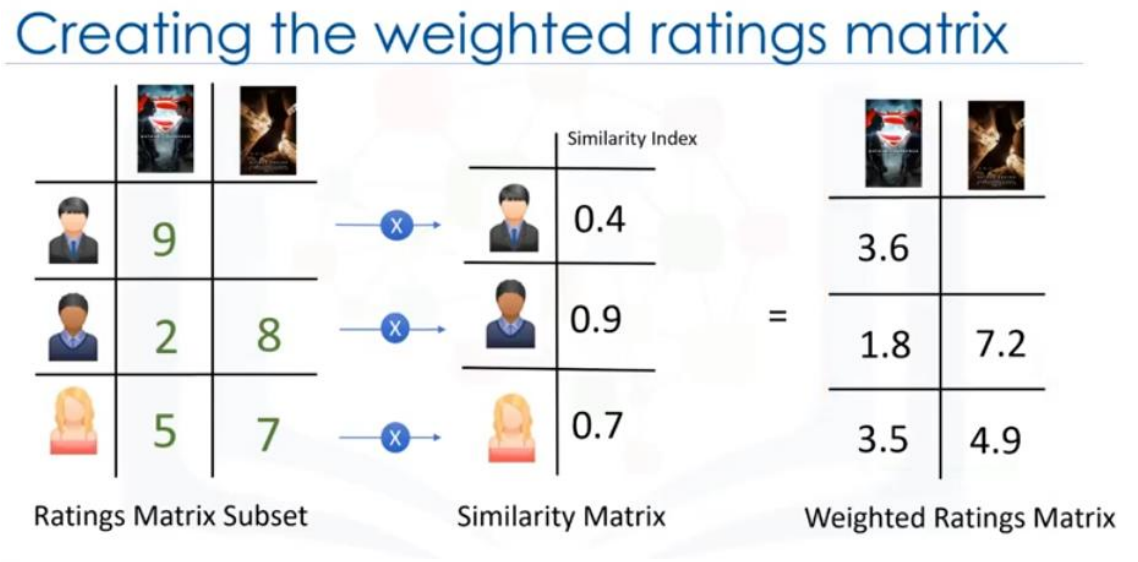
Disadvantages

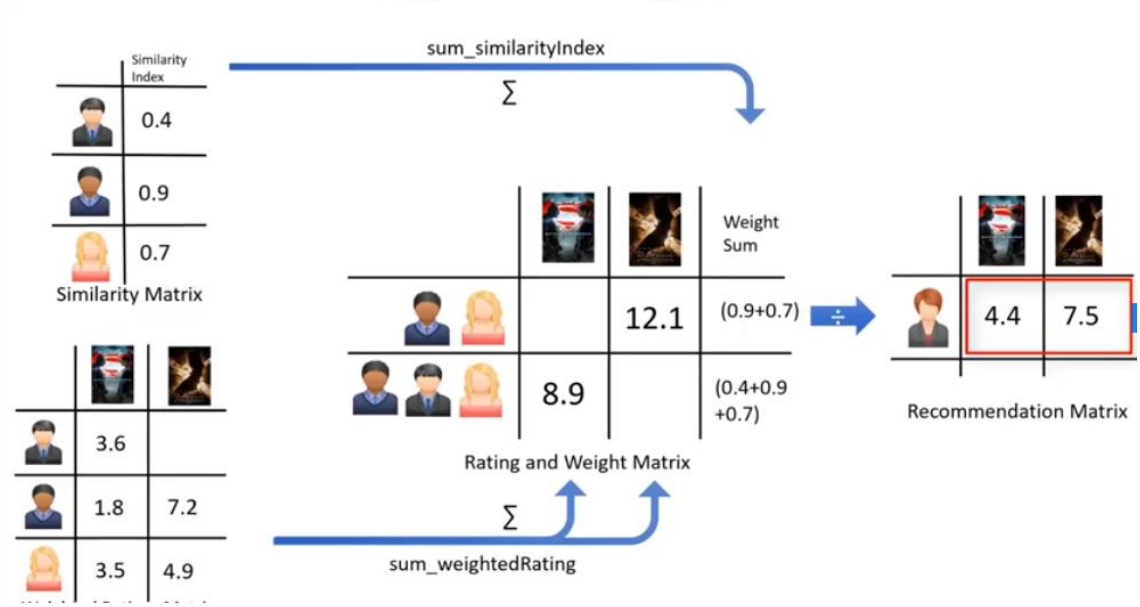
- Doesn't take into account what others think of the item, so low quality item recommendations might happen
- Extracting data is not always intuitive
- Determining what characteristics of the item the user dislikes or likes is not always obvious

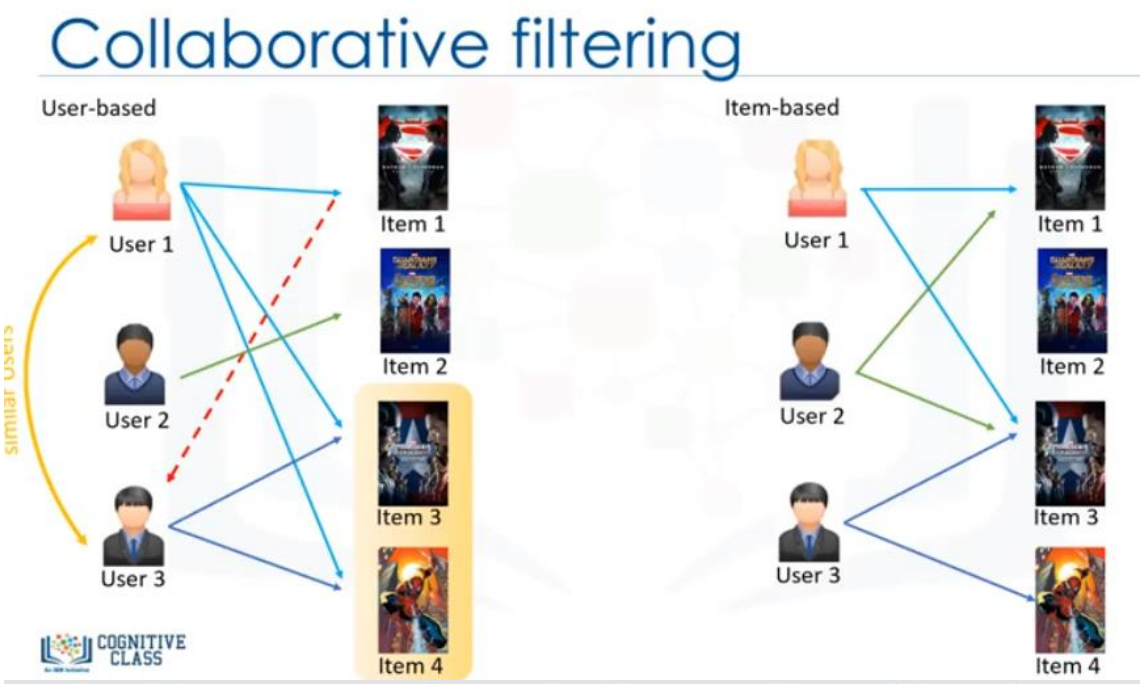
COLLABORATIVE FILTERING

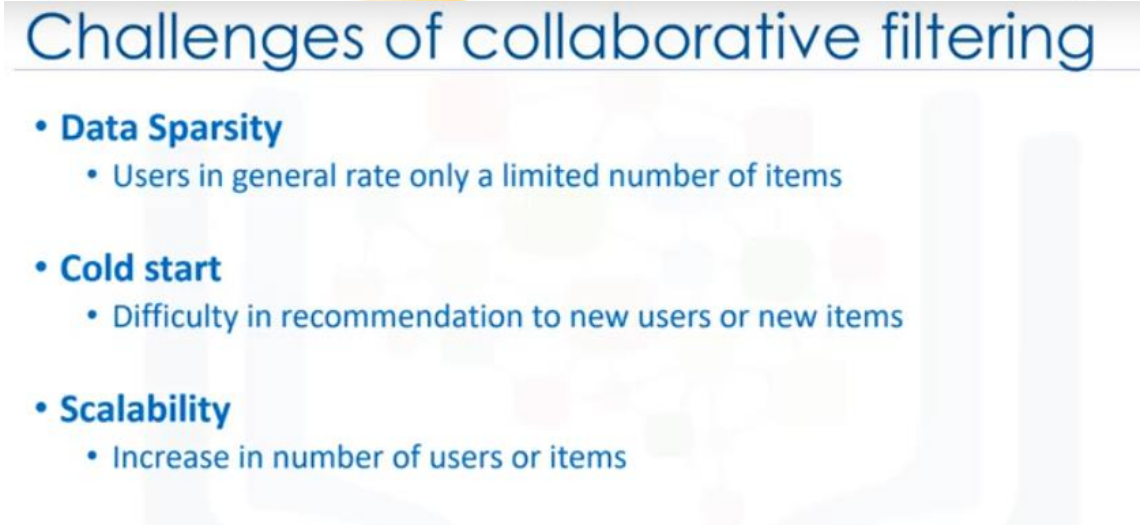



Learning the similarity weights
9
6
8
4
2
10
6
8
5
9
10
7
?
10
7
8
?
0.4
0.9
0.7
Ratings Matrix
COGNITIVE CLASS


Creating the weighted ratings matrix
9
2
8
5
7
Similarity Index
0.4
0.9
0.7
=
3.6
1.8
7.2
3.5
4.9
Ratings Matrix Subset
Similarity Matrix
Weighted Ratings Matrix


Similarity Index
0.4
0.9
0.7
Similarity Matrix
sum_similarityIndex
Σ
Weight Sum
12.1
(0.9+0.7)
8.9
(0.4+0.9 +0.7)
Rating and Weight Matrix
4.4
7.5
Recommendation Matrix
3.6
1.8
7.2
3.5
4.9
Σ
sum_weightedRating





Hello, and welcome! In this video, we'll be covering a recommender system technique called, Collaborative filtering. So let's get started.
Collaborative filtering is based on the fact that relationships exist between products and people's interests. Many recommendation systems use Collaborative filtering to find these relationships and to give an accurate recommendation of a product that the user might like or be interested in.
Collaborative filtering has basically two approaches: User-based and Item-based.
User-based collaborative filtering is based on the user's similarity or neighborhood.
Item-based collaborative filtering is based on similarity among items.
Let's first look at the intuition behind the "user-based" approach.
In user-based collaborative filtering, we have an active user for whom the recommendation is aimed. The collaborative filtering engine, first looks for users who are similar, that is, users who share the active user's rating patterns. Collaborative filtering bases this similarity on things like history, preference, and choices that users make when buying, watching, or enjoying something. For example, movies that similar users have rated highly. Then, it uses the ratings from these similar users to predict the possible ratings by the active user for a movie that she had not previously

watched. For instance, if 2 users are similar or are
neighbors, in terms of their interest in movies, we can recommend a movie to the active user that her neighbor has already seen. Now, let's dive into the algorithm to see
how all of this works.
Assume that we have a simple user-item matrix, which shows the ratings of 4 users for 5 different
movies. Let's also assume that our active user has
watched and rated 3 out of these 5 movies. Let's find out which of the two movies that our active user hasn't watched, should be recommended to her.
The first step is to discover how similar the active user is to the other users.
How do we do this? Well, this can be done through several different
statistical and vectorial techniques such as distance or similarity measurements, including Euclidean Distance, Pearson Correlation, Cosine Similarity, and so on.
To calculate the level of similarity between 2 users, we use the 3 movies that both the users have rated in the past. Regardless of what we use for similarity measurement, let's say, for example, the similarity, could be 0.7, 0.9, and 0.4 between the active user and other users. These numbers represent similarity weights,
or proximity of the active user to other users in the dataset.
The next step is to create a weighted rating matrix.
We just calculated the similarity of users to our active user in the previous slide.
Now we can use it to calculate the possible opinion of the active user about our 2 target movies. This is achieved by multiplying the similarity
weights to the user ratings. It results in a weighted ratings matrix, which
represents the user's neighbour's opinion about our 2 candidate movies for recommendation.
In fact, it incorporates the behaviour of other users and gives more weight to the ratings of those users who are more similar to the active user.
Now we can generate the recommendation matrix by aggregating all of the weighted rates.
However, as 3 users rated the first potential movie, and 2 users rated the second movie, we have to normalize the weighted rating values. We do this by dividing it by the sum of the similarity index for users. The result is the potential rating that our
active user will give to these movies, based on her similarity to other users.
It is obvious that we can use it to rank the movies for providing recommendation to our active user.
Now, let's examine what's different between "User-based" and "Item-based" Collaborative filtering: In the User-based approach, the recommendation
is based on users of the same neighborhood, with whom he or she shares common preferences.
For example, as User1 and User3 both liked Item 3 and Item 4, we consider them as similar or neighbor users, and recommend Item 1, which is positively rated by User1 to User3.
In the item-based approach, similar items build neighborhoods on the behavior of users.
(Please note, however, that it is NOT based on their content).
For example, Item 1 and Item 3 are considered neighbors, as they were positively rated by both User1 and User2. So, Item 1 can be recommended to User 3 as
he has already shown interest in Item3. Therefore, the recommendations here are based on the items in the neighborhood that a user might prefer.
Collaborative filtering is a very effective recommendation system, however, there are some challenges with it as well. One of them is Data Sparsity.
Data sparsity happens when you have a large dataset of users, who generally, rate only a limited number of items. As mentioned, collaborative-based recommenders
can only predict scoring of an item if there are other users who have rated it.
Due to sparsity, we might not have enough ratings in the user-item dataset, which makes it impossible to provide proper recommendations. Another issue to keep in mind is something called 'cold start.' Cold start refers to the difficulty the recommendation

system has when there is a new user and, as such, a profile doesn't exist for them yet.
Cold start can also happen when we have a new item, which has not received a rating.
Scalability can become an issue, as well. As the number of users or items increases
and the amount of data expands, Collaborative filtering algorithms will begin to suffer
drops in performance, simply due to growth in the similarity computation.
There are some solutions for each of these challenges, such as using hybrid-based recommender systems, but they are out of scope of this course.
Thanks for watching!

>>Lab: