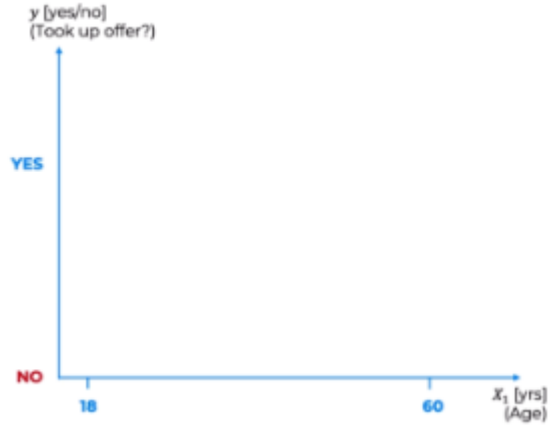
Essential Machine Learning Algorithms

Logistic Regression:

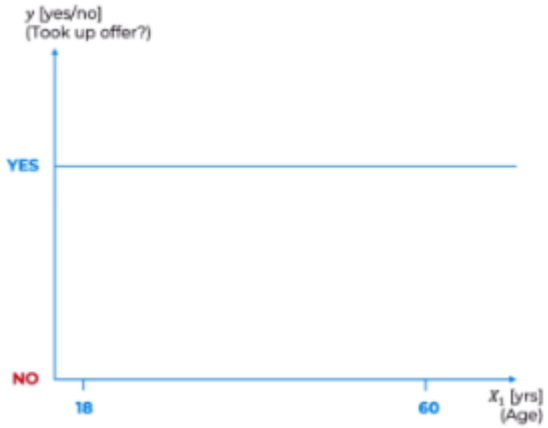
Logistic regression is a supervised machine learning algorithm used for binary classification tasks, predicting the probability of an outcome, event, or observation. The model is designed to predict a binary dependent variable, such as 'yes' or 'no,' '0' or '1,' or 'true' or 'false.' For instance, it could predict whether a person will survive an accident or not, or if a student will pass an exam, based on one or more independent variables.

Imagine you work in the marketing department of an e-commerce company, and you are tasked with predicting whether a customer will purchase a specific product, which is a categorical variable with 'yes' or 'no' as possible outcomes. You might expect this dependent variable based on an independent variable such as age. So, depending on their age, will they purchase the product your company is offering? In this scenario, the x-axis would represent age, and the y-axis would represent the binary outcome of 'yes' or 'no.'

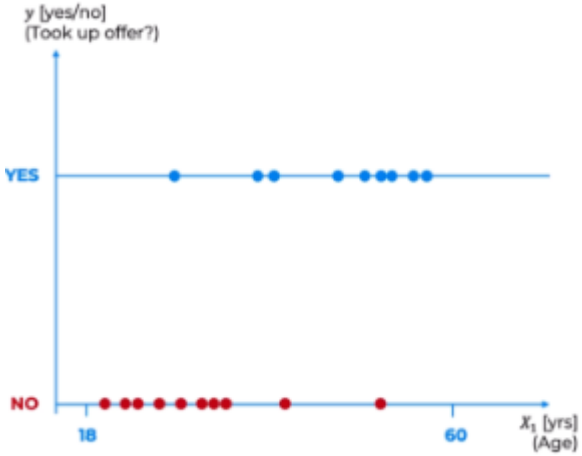
Let's say our x-axis spans from 18 to 60 years of age, and y-axis, reflects the binary outcome: "yes" (purchase) or "no" (no purchase). There are no grey areas here - a customer either buys the product or doesn't.



For illustrative purposes, we're going to add a horizontal line to represent this binary outcome clearly.



We got a certain number of observations in our dataset. Each observation represents a customer. So we know people's age and and their purchase behavior i.e when they were exposed to the offer, whether they purchased or didn't. the product, while those in blue did. And that's our dataset.



However, the relationship between age and purchase behavior might not be linear, meaning we can't draw a simple straight line through these points to distinguish between purchasers and non-purchasers. Here's where logistic regression comes in: it offers a mathematical framework to model this relationship. By applying the logistic regression equation to our data, we can determine the best-fitting curve that separates the two classes based on age, effectively capturing the purchase probability for each age group. This enables us to accurately predict future customers' purchase behavior based on age.

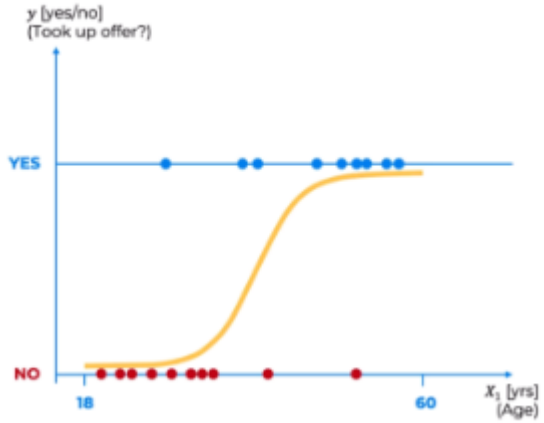
Logistic regression utilizes the logistic function to transform the linear combination of independent variables into a probability score bounded between 0 and 1.

The logistic regression equation is:

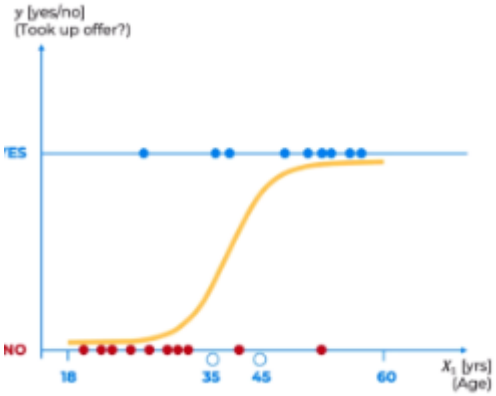
Where:

p is the probability

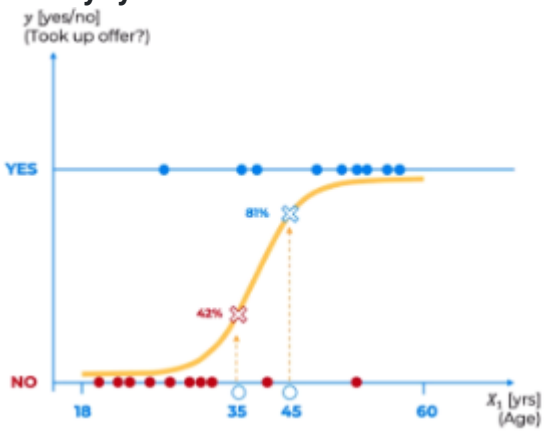
So let's look at the logistic regression curve. It's not a straight line, but rather an S-shaped curve, also known as a sigmoid curve. This curve represents the predicted probability of purchase at different ages.



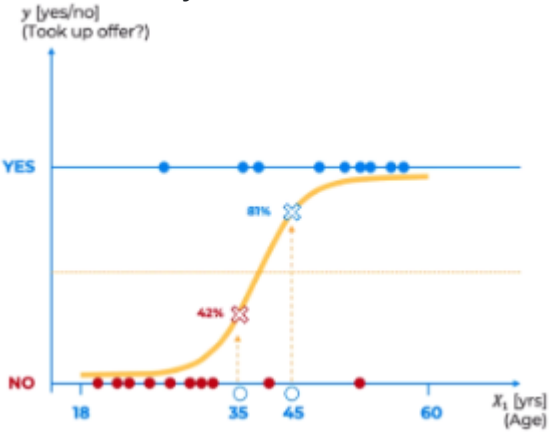
Now that we have the logistic regression curve, let's see how it works in action. Suppose we've constructed this model using our dataset via logistic regression. How does it translate to real-world applications with new observations? Let's consider two new observations: one individual aged 35 and another aged 45.



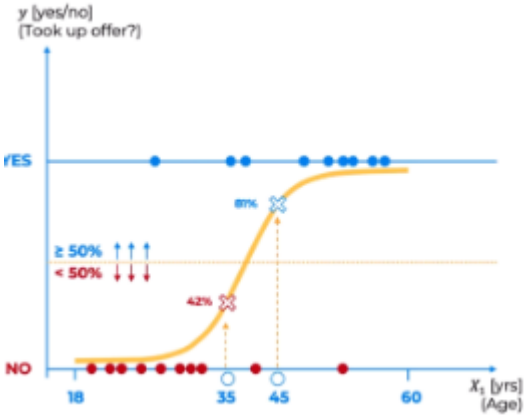
To apply our model, we project these values onto our logistic regression curve to determine where they fit. The logistic regression model then gives us the probability of each individual making a purchase. These probabilities range between 0 and 1, where 0 indicates "No" (no purchase) and 1 indicates "Yes" (purchase), with values in between representing the likelihood of purchase. For the 35-year-old, the model predicts a 42% chance of making a purchase. This means there is a 42% probability that this individual will take up the offer based on the model. Similarly, for the 45-year-old, the model predicts an 81% chance of making a purchase. This illustrates how the logistic regression model provides us with probabilities that an individual will say "yes" to the offer.



The probabilities we see (like the 42% and 81% in our example) are the "p" values in the logistic regression equation which represents the likelihood of a purchase.
We could use these probabilities directly in some use cases, which is one of the primary applications of logistic regression. However, in most situations, we need a binary outcome a simple yes or no. For those scenarios, we split our curve into two parts to classify the outcomes.



Anything above this line, with a probability of 50% or higher, gets classified as a "yes" (purchase) i.e a binary 1. Conversely, anything below the line, with a probability lower than 50%, gets classified as a "no" (no purchase) i.e a binary 0.

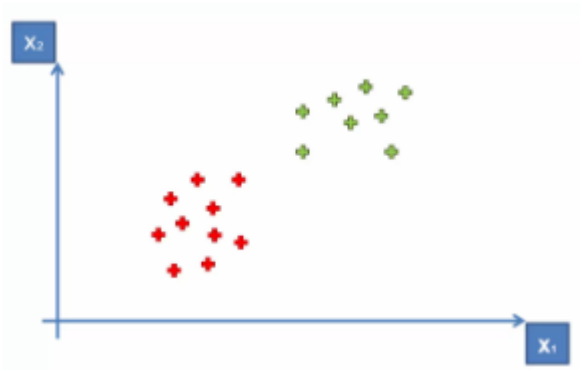


So, based on this logistic regression, we would conclude that the 35-year-old would not purchase the product because their probability is 42%, which is below the 50% threshold. In contrast, the 45-year-old would purchase the product because their probability is 81%, which is above the 50% threshold.
If we have multiple independent variables, like age, income, gender, and education level, our logistic regression equation would expand to incorporate these variables.
Where p is the probability, b0 is the intercept. X1, X2, X3, and X4 are the independent variables representing age, income, gender, and education level, respectively. And b1 , b2 , b3 and b4 are the coefficients of these independent variables.
In conclusion, logistic regression is a powerful tool for binary classification tasks. It models the relationship between one or more independent variables and a binary outcome, providing probabilities that can be used to make predictions. This method is particularly useful when the relationship between the variables is not linear, and it allows for the inclusion of multiple predictors to improve the accuracy of predictions.
Support Vector Machine

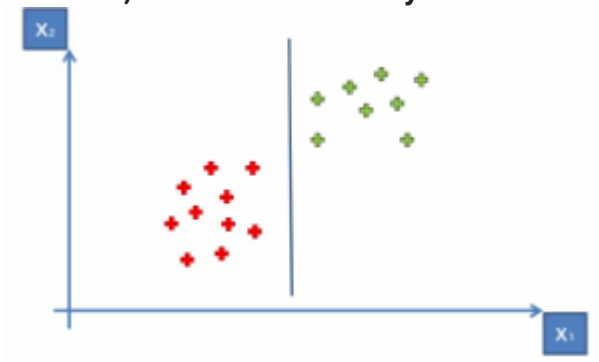
The next machine learning algorithm we will explore is Support Vector Machine (SVM). It is one of the most popular Supervised Learning algorithms, which is utilized for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning. Now let's understand how SVM works
In machine learning, SVM algorithms are designed to find the best line, or decision boundary, to separate data points into different categories. This line is called a hyperplane, and the SVM aims to find the hyperplane with the largest margin between the data points. This helps to ensure that the SVM can correctly classify new data points that it has not seen before.

$$ln\left(\frac{p}{1-p}\right) = b_0 + b_1X_1 + b_2X_2 + b_3X_3 + b_4X_4$$

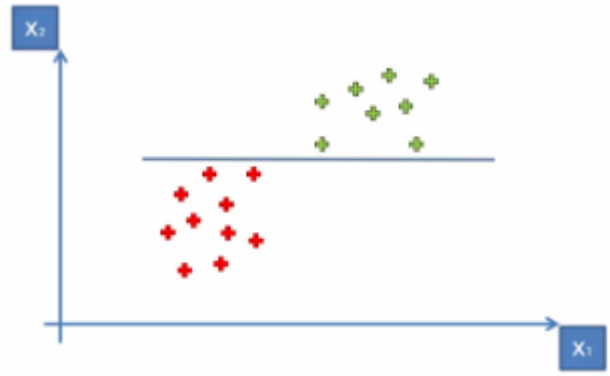
So here's how it works in a simpler setting. Imagine we have points plotted on a graph, like a map with two axes labeled x1 and x2. These points represent our data, and we already know which category each point belongs to, the green or red. But how do we draw a line that perfectly separates the green points from the red ones? This line, the decision boundary, is crucial because it allows us to classify new data points we encounter later. The SVM helps us find the best possible line that separates the existing data with the most space in between the categories. The wider this space, or margin, the easier it is for the SVM to sort new points into the right category correctly
So, how can we separate these points we see here?



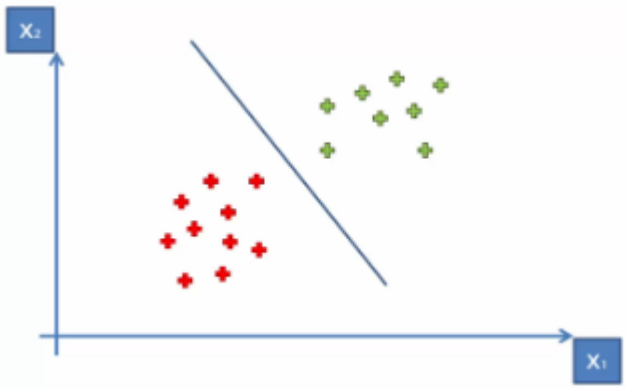
One way is to draw a line in our two-dimensional space and then say anything to the right of this line will be green, and anything to the left will be red. If a new point falls somewhere on this graph, we'll know right away if it's red or green because we'll know which side of the line it falls on.
However, there's another way! We can draw a horizontal line like that,



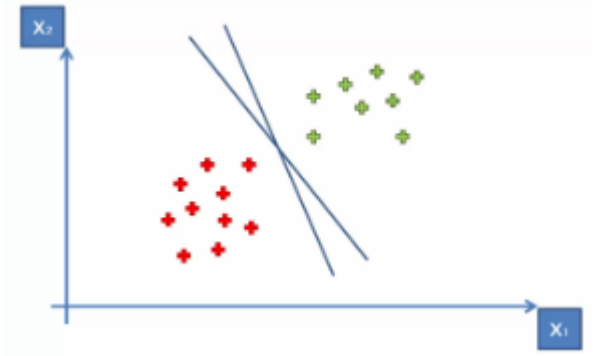
or a diagonal line like that,



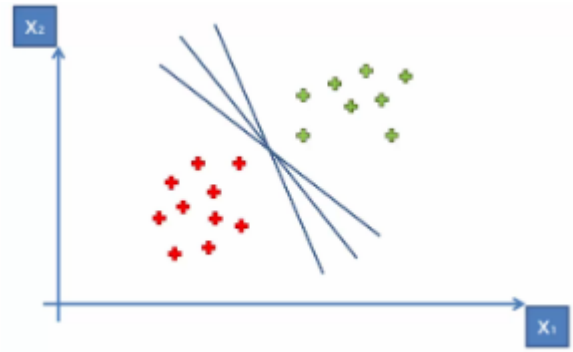
another diagonal line like that,



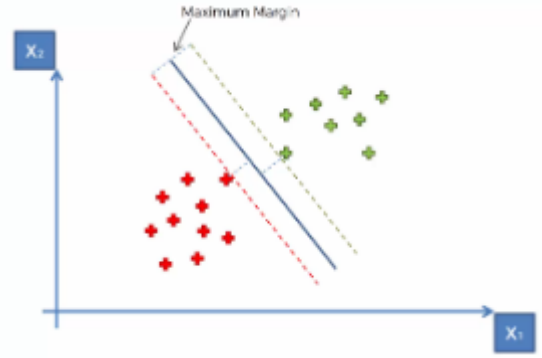
or even another diagonal line like that.



So we can create many different lines that will achieve the same result of separating our points into two classes.
But at the same time, they all will have different consequences in the future.
When we add new points, depending on where they fall, they'll be classified as green or red. So we want to find the optimal line, and that's what SVM is all about. It's about finding the best line, or decision boundary, which will help us separate our space into classes in the most robust way possible. Let's find out how the SVM actually searches for this best line!
Let's look at this graph to understand how the Support Vector Machine (SVM) works. Imagine a tug-of-war between the two classes (green and red). The SVM aims to create the widest possible gap between these two classes, which is called the margin.



In this graph, the solid blue line is the optimal separating line, or hyperplane, found by the SVM, that creates the maximum margin between the two classes. This margin is the distance between the hyperplane (solid blue line) and the nearest data points from each class, which are known as support vectors.



This hyperplane perfectly separates the two classes and is positioned to maximize the distance from the closest points of each class.

The support vectors are the critical elements of the training set. They are the data points closest to the hyperplane and directly influence its position and orientation. Without the support vectors, the SVM would not have a robust boundary to separate the classes effectively. Think of them like anchors for the decision boundary. These anchors are the data points that are closest to the separation line, and by moving them, we can significantly change the position of the line.

And hence we call them support vectors because they literally support the decision boundary by defining its position and margin.

The SVM aims to maximize the margin, which is the distance between the support vectors and the hyperplane. This maximization ensures that the decision boundary is as far as possible from any data point, reducing the risk of misclassification.

In this diagram, the dashed lines (red for one class and green for the other) represent the boundaries of the margin. The green dashed line represents the positive margin boundary, and the red dashed line represents the negative margin boundary. These boundaries are parallel to the hyperplane and pass through the support vectors.

So, that's how SVM works. Of course, there's some complicated mathematics behind it, but the essence of the intuitive part is exactly this: we're working with a linearly separable data set where we can draw a line to separate the two categories, and then search for the one with the maximum margin.

However, in the real world, data isn't always perfectly linearly separable. This is where things get a bit more complex, and SVMs utilize advanced techniques to handle non-linear data.

In essence, SVM is an extreme type of algorithm because it focuses on the data points closest to the decision boundary, known as support vectors, to construct its analysis. This approach allows SVM to create robust decision boundaries, which can be highly effective, especially in scenarios where there is limited data or complex data distributions. This distinguishing characteristic sets SVM apart from other machine learning algorithms and contributes to its versatility and applicability across various domains.

KNN

Next, we have the K-Nearest Neighbors algorithm, a commonly used machine-learning technique for classification and regression tasks. It works on the principle that data points with similar characteristics usually have similar labels or values. It's a fundamental method in machine learning, often used for its simplicity and effectiveness.

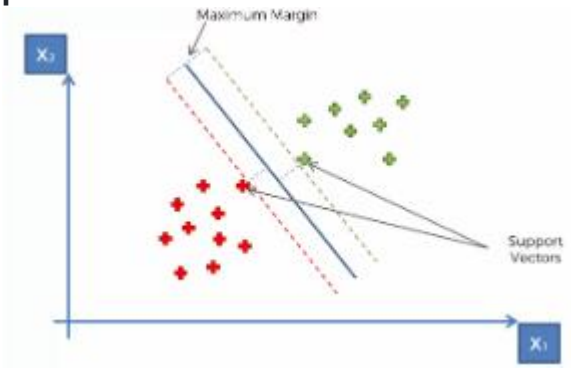
During the training phase, the KNN algorithm stores the entire training dataset as a reference. When making predictions, it calculates the distance between the input data point and all the training examples, using a chosen distance metric such as Euclidean distance.

Here's how KNN uses these neighbors for different tasks:

In Classification, KNN checks the labels of those K closest neighbors. Whichever label shows up the most that becomes the label for the new data point. Like a voting system for closest neighbors!

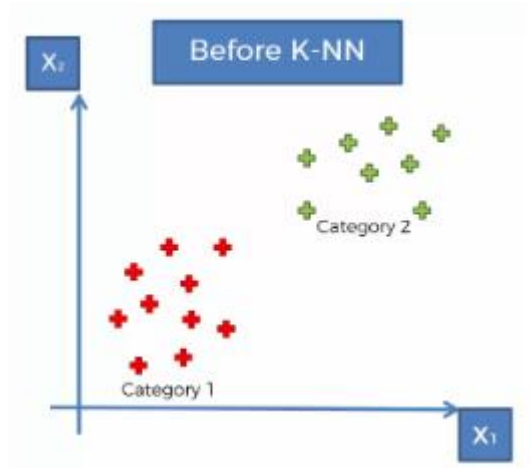
In Regression: Instead of a vote, KNN takes an average (or weighted average) of the values from those K closest neighbors. This becomes the predicted value for the new data point. Like taking an average answer from your closest classmates on a test.

To illustrate this, let's consider a scenario where we have two categories already present in our dataset.



we have identified two categories: Category 1 on the left side, which is red, and Category 2, which is green on the right. For simplicity's sake, we will consider two variables or two columns in our dataset, so all of this grouping is happening based on these two columns, X1 and X2.

Let's say we add a new data point to our data set. The question is, should it fall into the red category, or should it fall into the green category?



So how should we decide whether it's a red data point or whether it's a green data point? That's where the nearest neighbors algorithm will come to assist us.

So how does the K-Nearest Neighbors algorithm work? Well, we will build a step-by-step rule guide to using KNN. Then, after we've built it, we'll perform it manually to see how it works in action. You'll see that it's a very simple and intuitive algorithm!

First, we choose a number 'k' for our algorithm. This value determines how many neighbors we'll consider when making a prediction. There's no one-size-fits-all answer for k, and it can be 1, 2, 3, 4, 5, or any other number.

[Step 1: Choose the number of K neighbors]

For now, let's assume that the common default value for k to start with is 5.

[Step 2: Take the k nearest neighbors of the new data point, according to the Euclidean distance]

Next, we find the k nearest neighbors of our new data point. To determine "nearest," we'll calculate the Euclidean distance between the new data point and all the points in our training data. Euclidean distance is a common choice, but other distance metrics like Manhattan distance can also be used depending on the data.

[Step 3: Among the k neighbors, count the number of data points in each category]

Once we've identified the nearest neighbors, the next step is to count the number of data points in each category. This involves tallying how many data points belong to each category, whether you have two categories or more.

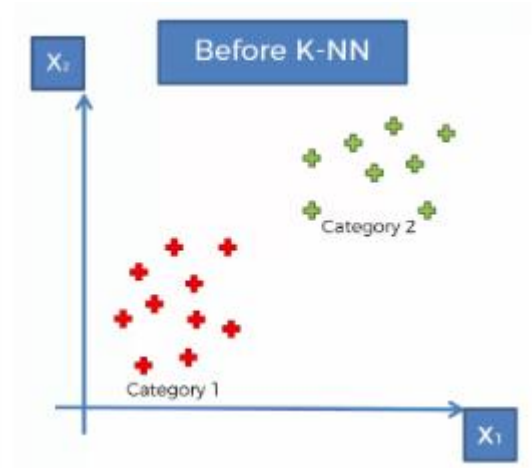
[Step 4: Assign the new data point to the category where you counted the most neighbors]

After counting, we assign the new data point to the category with the most neighbors. And then your model is ready.

It's as simple as that, and that's why it's known as K nearest neighbor.

Now that we understand the steps, let's perform a manual exercise to see KNN in action to really solidify this knowledge. So let's move onto that.

So here we have the new data point added to our scatter plot, as we saw previously.



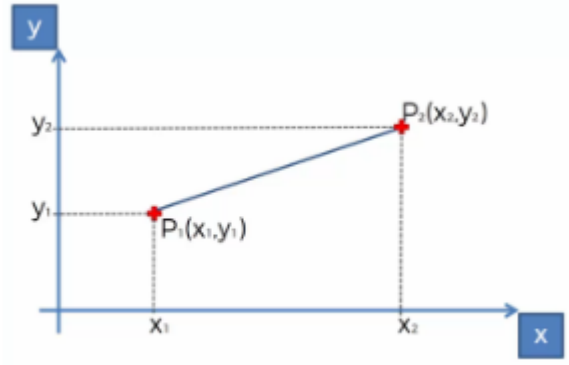
Now, how do we find the nearest neighbors of this new data point?
To find the nearest neighbors, we'll use the Euclidean distance formula. This formula tells us the straight-line distance between two points. The smaller the distance between our new data point and another point, the more similar they are likely to be. Well, let's start by examining the Euclidean distance that we'll use. Essentially, if we have two points, P1 and P2, then the distance between the two points is measured according to this formula:

$$d = \sqrt{((x1 - x2)^2 + (y1 - y2)^2)}$$

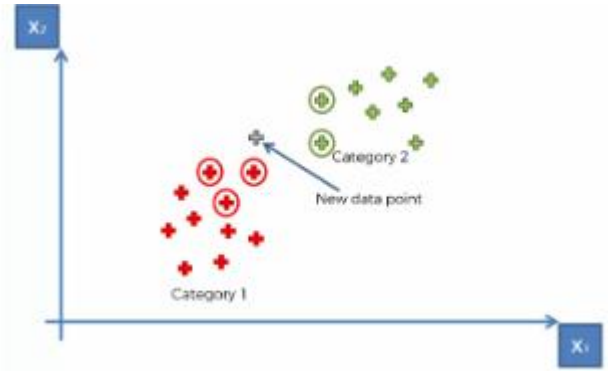
where:

- **d is the Euclidean distance**
- **(x1, y1) are the coordinates of the first point (P1)**
- **(x2, y2) are the coordinates of the second point (P2)**

Here we calculate the difference between the x-coordinates and y-coordinates of the two points. Then by squaring these differences, we ensure that negative values don't affect the final result negatively. Adding these squared differences and taking the square root of the sum gives us the straight-line distance between the two points, which is the Euclidean distance. Again, you could use any type of distance, but this is the geometrical distance and this is what we're going to stick to because Euclidean distance works well for many datasets.

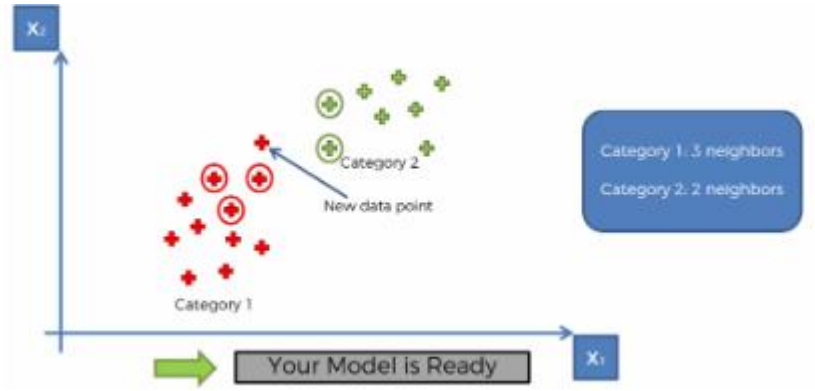


Referring to the graph on the screen, the next step is to find the 5 closest neighbors for our new data point. We'll calculate the Euclidean distance between our new point and all the existing data points. The points with the smallest distances will be our closest neighbors. So we can see that this is the closest one, the second closest, third closest, fourth closest, and fifth closest - which are marked on the graph.

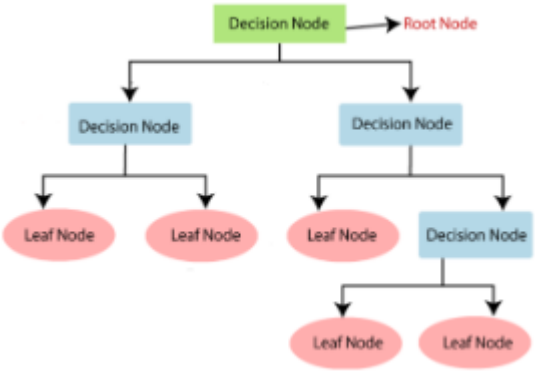


After this, among these k neighbors, let's count the number of data points in each category. In Category 1, which is in red, we have three neighbors, and in Category 2, which is in green, we have 2 neighbors, which can again be seen via the graph on the screen,
We now assign the new data point to the category where we counted the most neighbors. Since we have more neighbors in Category 1 (red), our new data point likely belongs there! Now we have classified this new point, and our model is ready.

And that's KNN It's a straightforward and easy-to-understand algorithm that allows us to classify new data points based on their similarity to existing data.
Decision Tree Classifier
We have the Decision Tree Algorithm. This is a popular method in supervised learning used for predicting results based on input data.
Imagine you're sorting fruits at a grocery store. You need a way to quickly decide which basket each fruit belongs in, like apples, oranges, or bananas. This is where a decision tree comes in. In machine learning, a decision tree is a helpful tool for making predictions based on data. It works by asking a series of simple questions to arrive at a final outcome. Just like you might ask, "Is the fruit red or yellow?" to decide if it's an apple or a banana, a decision tree asks questions about the data to categorize it. This type of learning is called supervised learning because the decision tree learns from existing data that are already labeled. For example, you might have a bunch of fruits that are already sorted, and the decision tree learns the questions to ask by looking at their features (round, yellow, etc.) and their corresponding labels (apple, banana). By following this approach, decision trees can be used to solve various problems, not just sorting fruit! They can be used to predict things like whether an email is spam or not spam (classification), or even estimate the price of a house based on its size and location (regression).
To understand decision trees better, it's important to know some key terminologies:
First we have is:
Root Node: it is the top node in a decision tree that represents the entire dataset, which is then split into two or more homogeneous sets. Next we have is
Decision Nodes also known as internal node. These nodes represent the features of the dataset and are used to make decisions based on their values, leading to further branches. Then we have
Leaf Nodes which is also known as terminal nodes, these nodes represent the final outcome or decision and do not split further.



Sub Tree: A subtree in a decision tree is essentially a smaller decision tree within the larger tree.



Branches: We've talked about how decision trees use questions to make predictions. But how do we move from one question to the next? That's where branches come in. Think of them as pathways in the tree, guiding you based on your answers, or in other words branches (also known as edges) represent the connections between nodes, illustrating how decisions are determined based on specific conditions or circumstances.

Next, we have is splitting.

Splitting: Splitting is the process of dividing a node into two or more sub-nodes by applying a decision criterion.

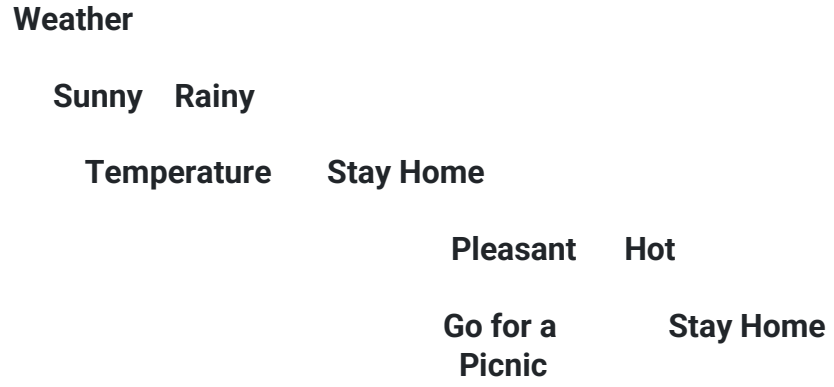
Parent node and child node: A parent node serves as the original node from which a split originates, while child nodes are formed as a result of this division.

So, how does the Decision Tree algorithm work? Decision trees operate by recursively partitioning the data into subsets, each time selecting the most informative feature to make the split.

To construct a decision tree, we start by creating the root node, denoted as S, which contains the entire dataset. Next, we employ an Attribute Selection Measure (ASM) to determine the best attribute within the dataset. Using this attribute, we partition S into subsets, each representing possible values for the chosen attribute. Subsequently, we create decision tree nodes based on these divisions, with each node representing the selected attribute. This process continues recursively as we generate new decision trees using the subsets created earlier. We iterate until reaching a point where further classification is not possible, resulting in the creation of leaf nodes, marking the end of the decision tree.

It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.

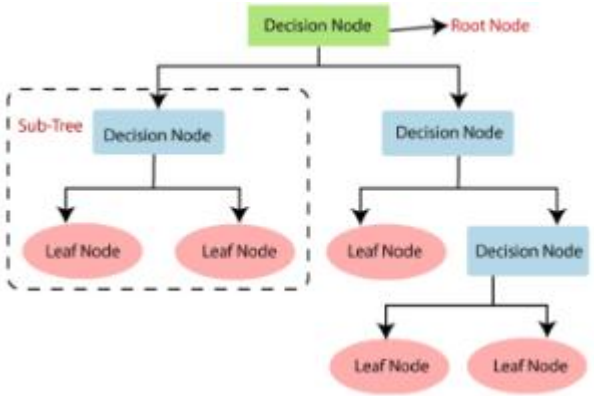
We can apply the same decision tree process to everyday scenarios, like deciding whether to go for a picnic. In this case, the root node represents the entire dataset, with weather identified as the most important factor using Attribute Selection Measure (ASM). It splits into two branches based on the weather: Sunny or Rainy. Under the "Sunny" branch, a decision node for Temperature (Hot or Pleasant) is created. The final decision is represented by leaf nodes: "Go for a Picnic" if sunny and pleasant, "Stay Home" if sunny and hot, and "Stay Home" again if rainy, as no further decision is necessary based on temperature for rainy weather.
Here's the corresponding decision tree diagram:



Next, the question arises: how do we select the best attribute for the root node and sub-nodes using ASM mentioned previously? To solve this problem ASM has two popular techniques:

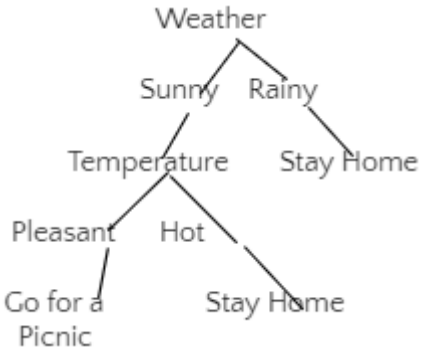
- **Information Gain**
- **Gini Index**

Information Gain:
Information gain is the measurement of changes in entropy after splitting a dataset based on an attribute. It tells us how much more organized or certain the data becomes after the split. Imagine a box containing different types of fruits. When the box has only apples (all the same kind), the entropy (uncertainty) is low because the box is very organized. However, if there's a mix of apples, oranges, and bananas, the entropy is high due to the increased uncertainty about the contents. Splitting the data using an attribute is essentially sorting the fruits. Information gain tells us how much better organized the fruits become after a particular sorting method, such as splitting by color or size. The better the attribute separates the data, the lower the entropy becomes in the resulting groups. In the context of building a decision tree, we calculate the information gain for each potential split at each step. The algorithm aims to find the attribute that results in the highest information gain, meaning the data becomes most organized after splitting on that attribute. This attribute is then used to create the primary branch at the current node.
Now, let's see how to calculate information gain. This measure is computed using a formula derived from entropy calculations.
To calculate entropy, we use the formula:



where pi is the probability of each class in the dataset T.

Next, to calculate the information gain for an attribute, we measure the entropy before and after the split and then compute the difference. The formula for information gain (IG) is:



Here, T is the original dataset, A is the attribute being evaluated, Tv is the subset of T where the attribute A has the value v, and T is the number of instances in T.

Now let's consider a dataset, calculate the entropy, and then calculating information gain. We'll use a simple dataset to make the concepts clear.

Color	Size	Type
Red	Small	Apple
Red	Small	Apple
Red	Small	Apple
Yellow	Small	Banana
Yellow	Large	Banana
Green	Small	Apple
Green	Large	Watermelon
Green	Large	Watermelon
Yellow	Small	Banana
Yellow	Small	Banana

Now let’s calculate the Entropy. As discussed previously, entropy is a measure of the uncertainty or randomness in a dataset. We calculate entropy for the "Type" attribute in our dataset.

First, count the occurrences of each type of fruit:

- **Apples: 4**
- **Bananas: 4**
- **Watermelons: 2**

Total fruits: 4+4+2=10
Next, calculate the probabilities:

- **p(Apple)= 4/10= 0.4**
- **p(Banana)= 4/10= 0.4**
- **p(Watermelon)= 2/10= 0.2**

Now, compute the entropy:
Entropy(T)= -(0.4 log20.4+0.4 log20.4+0.2 log20.2)
After calculating the logarithms and products, and summing these values we'll get:
Entropy(T)=−(−0.52877−0.52877−0.46439)≈1.522
Next we'll calculate information gain, and as previously discussed Information gain measures how much uncertainty in the dataset is reduced after splitting it based on an attribute. We calculate information gain for the "Color" attribute.
First Split the dataset by "Color", so we have for:

- **Red: 3 Apples**
- **Yellow: 4 Bananas**
- **Green: 1 Apple, 2 Watermelons**

Next we Calculate the entropy for each of these subsets:

- **Red (3 Apples):**

Entropy(Red)=0
Since there's only one type of fruit (apples) in this subset, the entropy is 0. There's no uncertainty because all instances belong to the same class. Next

- **Yellow (4 Bananas):**

Entropy(Yellow)=0
Similar to the "Red" subset, there's only one type of fruit (bananas) here, so the entropy is also 0.

- **Green (1 Apple, 2 Watermelons):**

In this subset, there are two types of fruits: apples and watermelons. We calculate the probability of each class:
p(Apple)=1/3
p(Watermelon)=2/3
Then, we use these probabilities to calculate the entropy:
Entropy(Green)=-(1/3log21/3 + 2/3log22/3)
Entropy(Green)≈0.91829
Next we calculate the weighted average entropy after the split:

- **Red: 3/10 of the dataset**
- **Yellow: 4/10 of the dataset**
- **Green: 3/10 of the dataset**

Entropy after split ≈ (3/10 x 0) +(4/10 x 0) + (3/10 x 0.91829) ≈ 0.275
And now we calculate the information gain:
And since we know
Information Gain=Entropy(T)−Entropy after split
Then
Information Gain≈1.522−0.275≈1.247

So, the information gain for splitting by the "Color" attribute is approximately 1.247. This tells us how much more organized the dataset becomes when we split it by color.

By following this process and utilizing the information gain formula, you can calculate entropy and information gain for other attributes as well, choosing the one with the highest information gain to make the best split in constructing a decision tree.

Now, let's calculate the entropy and information gain for the other feature, which is size.

Let's Split the dataset by "Size", so we have for:

For the "Small" subset:

- **Apples: 4**
- **Bananas: 3**

Total fruits=4+3=7

Probability for "Small":

p(Apple)=4/7

p(Banana)=3/7

Now we will use these probabilities to calculate the entropy for 'Small':

Entropy(Small)=-(4/7 + 3/7) ≈ 0.985

Next for the "Large" subset:

- **Bananas: 1**
- **Watermelons: 2**

Total fruits=1+2=3

Then Probability for "Large":

p(Bananas)=1/3

p(Watermelon)=2/3

Now we will use these probabilities to calculate the entropy for 'Large':

Entropy(Large)=-(1/3 + 2/3) ≈ 0.918

Next, let's calculate the weighted average entropy after the split:

Entropy after split ≈ (7/10 x 0.985) + (3/10 x 0.918) ≈ 0.9649

Finally, let's calculate the information gain.

And since we know

Information Gain=Entropy(*T*)—Entropy after split

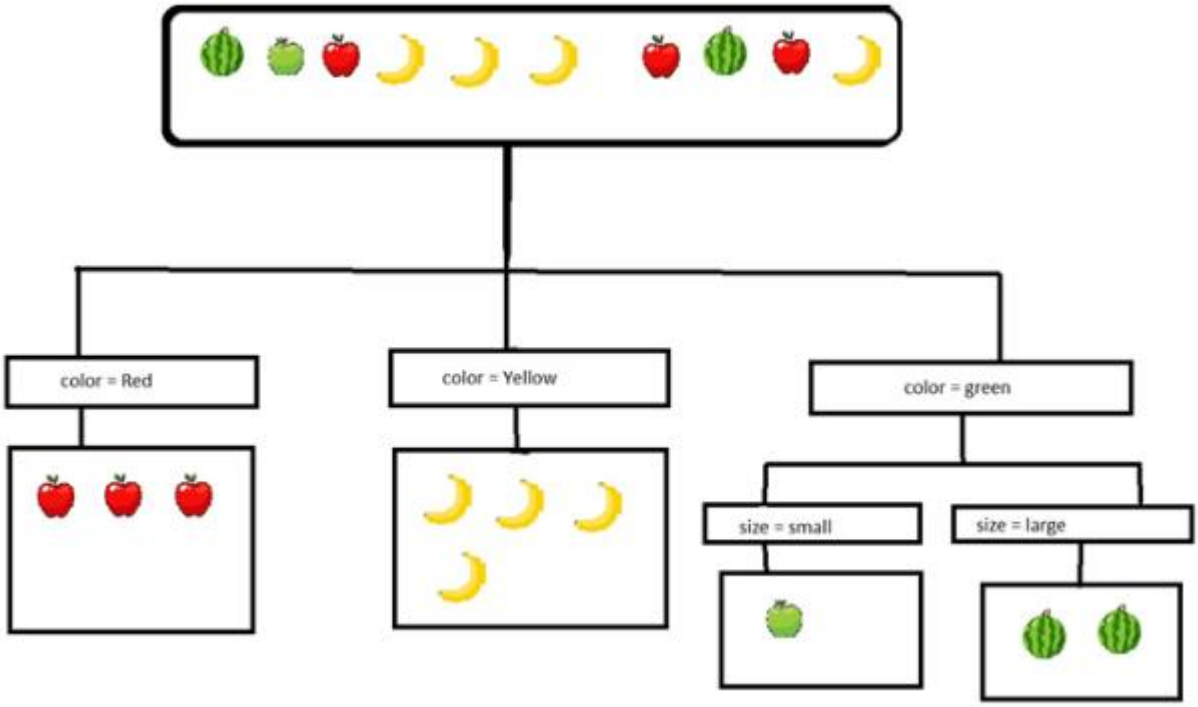
Information Gain≈1.522— 0.9649≈ 0.5571

So, the information gain for splitting by the "Size" attribute is approximately 0.5571.

Now let's compare the information gains:

- **Information Gain for splitting by "Color": 1.247**
- **Information Gain for splitting by "Size": 0.5571**

The higher the information gain, the better the attribute is for splitting the dataset and organizing it. In this case, splitting by "Color" results in a higher information gain, indicating that it provides a better organization of the dataset compared to splitting by "Size". Therefore we would choose "Color" as it leads to a better split in the dataset and reduces more uncertainty. We could further split this subset based on another attribute, such as Size, to create more distinct groups.



For the subset where Color = Red:
There are only Apples in this subset, further splitting is not required. We have reached a leaf node, xand the decision is made based on this subset.
For the subset where Color = Yellow:
Again, similar to the subset with Red color, there are only Bananas in this subset. No further splitting is needed.
Next for the subset where Color = Green:
Here, we have a mix of Apples and Watermelons. We could proceed to split this subset further based on another attribute, like Size, in order to generate additional distinct groups. Therefore we split the Green subset further based on the Size attribute into smaller and larger fruits. This split further separates apples from watermelons. And once again leaf node is reached, no further splitting occurs.
This sequence of splits represents how a decision tree algorithm uses information gain to organize and classify the dataset. By recursively choosing the attribute that provides the highest information gain, the decision tree algorithm constructs a tree that effectively partitions the dataset into distinct groups, ultimately leading to accurate predictions or classifications
Now, let's move on to the second technique for selecting the best attribute: the Gini Index.
Gini Index:
The Gini index, also known as Gini impurity, helps assess the quality of splits in decision trees. It does this by measuring the level of impurity within a node, indicating how well the data is separated based on the target variable.
Essentially, it assesses how well the data is grouped. CART (Classification and Regression Tree), a type of decision tree algorithm, utilizes this measure, applicable to both classification (categorizing data into classes) and regression (predicting continuous values). In decision tree construction, attributes yielding lower Gini

indices are favored over those with higher indices. Therefore, during the tree-building process, attributes leading to lower Gini indices are prioritized as they result in purer groups. A lower Gini index signifies greater purity, indicating a more homogenous group, whereas a higher index suggests more mixed data. It's crucial to note that the Gini index reflects the probability of misclassifying a randomly chosen element based on the distribution of labels within the node. For instance let's say we have two boxes: one with mostly apples and a few oranges, and the other with mostly oranges and a few apples. If we randomly pick a fruit from one of these boxes, there's a higher chance we might get it wrong because the boxes aren't neatly sorted. The Gini index helps us measure this chance of guessing wrong. If the fruits are very mixed up, the Gini index is high, indicating a higher probability of misclassifying a randomly chosen fruit. But if the fruits are well-sorted, the Gini index is low, meaning there's a lower chance of guessing wrong. So, when we're building our decision tree, we want to split our data into boxes (or nodes) in a way that minimizes this chance of misclassification, and the Gini index helps us make that decision.

As for the CART algorithm, it typically generates binary splits, dividing the data into two parts based on the selected attributes. Imagine you have a box of mixed fruits: apples and oranges. The CART algorithm helps you split this box into two smaller boxes based on a chosen attribute, like color. For example, one box could have only apples, and the other could have only oranges. This way, the fruits are divided into two parts (binary split) to make the groups more homogenous. While the analogy of a box containing mixed fruits attempts to illustrate a high Gini index, it's important to emphasize that the Gini index primarily concerns the distribution of classes within a node. For example, if a node (box) has 90% apples and 10% oranges, it has a lower Gini index compared to a node with 50% apples and 50% oranges. The Gini index measures how mixed the classes (apples and oranges) are within each node, rather than the diversity of attributes like color or size.

Gini index can be calculated using the below formula:

Gini Impurity=1-(pi2)

Where p(i) is the probability of a specific class, and the summation is done for all classes present in the dataset.

Using information gain and the Gini index are two different methods to measure the quality of a split in decision trees. Both have their own advantages and are used for different reasons based on the context and specific requirements of the machine learning task.

Random Forest

Building on the foundation of decision trees, the next algorithm to consider is Random Forest. This versatile technique can be used for both classification tasks, where we predict a category, and regression tasks. It enhances predictive performance by combining multiple decision trees to reduce overfitting and improve generalization. Overfitting occurs when a model learns the training data too well, capturing noise and details that do not generalize to new, unseen data.

But first, we'll learn about a new concept known as Ensemble Learning.

Imagine you have a group of friends trying to predict the weather. Each friend might have a different idea (sunny, rainy, cloudy). By combining all of their predictions, you might be able to come up with a more accurate forecast. Ensemble learning is similar to this idea. It combines multiple learning algorithms to improve the overall predictive performance.

By leveraging the strengths and compensating for the weaknesses of each individual model, Ensemble Learning can provide more accurate and reliable results than any single model alone.

The models used in Ensemble Learning can be different or the same. A great example of this is the Random Forest method, which we'll explore next. Random forest method combines a lot of decision tree methods. Instead of relying on just one decision tree, we run this process multiple times, creating a 'forest' of decision trees. Let's delve into the step-by-step process to understand how it works.

The 1st step is to select a subset of K data points from the training set. This injects randomness into the process and helps prevent overfitting.

Step 2 is to build a decision tree using only this subset.

Next is Step 3, where we define the number of trees (N) we want to create in the forest.

We then repeat step 1 and 2 multiple times, creating a collection of decision trees, each based on a different random subset of the data. The number of trees determines the density of the forest and can significantly impact the model's performance.

And lastly in step 4 when presented with a new data point, we send it down each tree in the forest. Each tree makes a prediction about the category the data point belongs to. And finally, we take a democratic vote. The category that receives the most votes from the individual trees becomes the final prediction for the new data point.

That's how Random Forest works.

So essentially, Random Forest leverages the wisdom of the crowd. Even though individual trees might not be perfect, by combining their predictions and taking a majority vote, we can often achieve a more accurate and robust classification.

In contrast to Random Forest's ensemble approach of combining multiple models, linear regression takes a different approach to achieving accurate predictions. So the next algorithm we'll explore is Linear Regression.

Linear Regression

Linear regression is a supervised machine learning algorithm that computes the linear relationship between the dependent variable (the variable we want to predict) and one or more independent variables (the variables we use for prediction) by fitting a linear equation to the data we have collected. When there is only one independent variable, it is known as Simple Linear Regression, and when there are multiple independent variables, it is called Multiple Linear Regression.

Imagine you're trying to guess someone's height based on their shoe size. Linear regression is a tool that helps you do this by finding a straight line that best fits the data. This data could be the shoe sizes and heights of people you already know.

The height you want to guess is called the dependent variable.

The shoe size you use for guessing is the independent variable.

This kind of guess with just one thing to predict like height from shoe size then it is called simple linear regression.

If you have more things to consider for your guess like adding weight and arm span to predict height, then that's called multiple linear regression. But even with more variables, the idea stays the same - finding a straight line to make the best possible prediction.

The strength of linear regression lies in its clarity. The model’s equation provides straightforward coefficients that illustrate how each independent variable affects the

dependent variable, aiding in a better understanding of the data. It serves as the alphabet of machine learning, forming the foundation for many more complex algorithms. Even advanced techniques like support vector machines owe a debt to linear regression! Additionally, it helps ensure our data makes sense by allowing researchers to test key assumptions before drawing conclusions.

The equation for simple linear regression is: Y = a + bX, where:

- **Y is the dependent variable (what we're trying to predict)**
- **X is the independent variable (what we're using for prediction)**
- **a is the intercept**
- **b is the slope**

In multiple linear regression, we're dealing with more than one independent variable and one dependent variable. The equation for multiple linear regression is:

$$Y = a + b_1X_1 + b_2X_2 + ... + bnXn$$

where:

- **Y represents the dependent variable**
- $X_1, X_2,..., Xn$ **represent multiple independent variables**
- **a represents the intercept**
- $b_1, b_2,..., bn$ **represent the slopes**

The goal of the algorithm in both simple and multiple linear regression is to find the best-fit line (or hyperplane), which captures the relationship between the independent variable(s) and the dependent variable. In simple linear regression, this involves one independent variable (X) and one dependent variable (Y), while in multiple linear regression, there are multiple independent variables. This best-fit line helps summarize how the X values influence the Y values. Regression analysis helps you understand the relationship between variables and predict the value of Y based on X.

So, what exactly is a best-fit line? It's the line (or hyperplane in the case of multiple regression) that minimizes the error between the predicted values and the actual values of the dependent variable. The model achieves this by adjusting the intercept and slope (or slopes in the case of multiple regression) to find the combination that minimizes this error.

This minimization is typically achieved by defining a cost function, also known as a loss function or error function. The error function quantifies the difference between the predicted values and the actual values of the dependent variable. The goal is to choose the parameters of the model (slope and intercept in simple linear regression, coefficients in multiple linear regression) that minimize this error function. In simple linear regression, the most commonly used error function is the Mean Squared Error (MSE). The MSE measures the average squared difference between the predicted value (denoted as pred) and the true value (denoted as y) across all data points.

Step-by-step way to simply understand it:

Imagine you have some data points scattered on a graph. The best-fit line is like drawing a line through those points in a way that it gets as close as possible to each one. This line helps us understand the relationship between two things, like for example how temperature affects ice cream sales.

So we want our line to be really good at predicting the actual values, which involves minimizing error. So, we adjust the position of the line until it's closest to all the data points. This minimizes the difference between what the line predicts and what's actually observed. And think of the error function like a scorecard that tells us how

well our line is doing. It measures how far off our predictions are from the real values. Our goal is to make this score as low as possible. Mean Squared Error (MSE) is a specific way to calculate our score. We take the difference between each predicted value and the real value, square it (to get rid of negative values), then average all those squared differences. So, MSE gives us an idea of how spread out the errors are. Now, imagine you're trying to find the lowest point in a valley, but you're blindfolded. You take small steps downhill, feeling the slope with your feet. Gradient descent is like that. We start with random values for our line's position, then keep tweaking them slightly in the direction that reduces the error function (MSE) until we reach the lowest point. And we repeat this process over and over, adjusting the line a bit each time, until we can't improve it anymore. Each adjustment makes our line better at predicting the real values. This iterative optimization ensures that our model continuously refines its parameters, gradually converging towards the most accurate representation of the relationship between variables.

So, in simple terms, linear regression is about finding the best line through our data points by adjusting it until it predicts the real values as closely as possible. We do this by using a scoring system (error function) and a method (gradient descent) that guides us towards the best line. It's like fine-tuning a radio until you get the clearest signal.

****

The formula for MSE is:

The model aims to minimize this MSE by adjusting the intercept and slopes in the case of simple and multiple linear regression, respectively. By finding the best combination of intercept and slopes, the model optimally fits the data and provides the best estimate for the dependent variable based on the independent variables.

The model aims to minimize this MSE by adjusting the intercept and slopes in the case of simple and multiple linear regression, respectively, through an iterative optimization process known as gradient descent. By iteratively updating the parameters using gradient descent, the algorithm gradually moves towards the optimal values that minimize the error function.

In summary, linear regression aims to find the best-fit line that minimizes the error between predicted and actual values of the dependent variable. This is achieved by adjusting the intercept and slopes through gradient descent, iteratively minimizing a error function such as Mean Squared Error. Through this process, the model optimally captures the relationship between independent and dependent variables, enabling accurate predictions of the dependent variable based on the independent variables.

So far we've been focusing on supervised learning.In supervised learning, our data comes with a handy advantage - labels! Imagine you have a bunch of pictures, but instead of just pictures, each one has a label that tells you exactly what it is. For example, in image classification, you might have pictures of cats, each labeled as "cat."

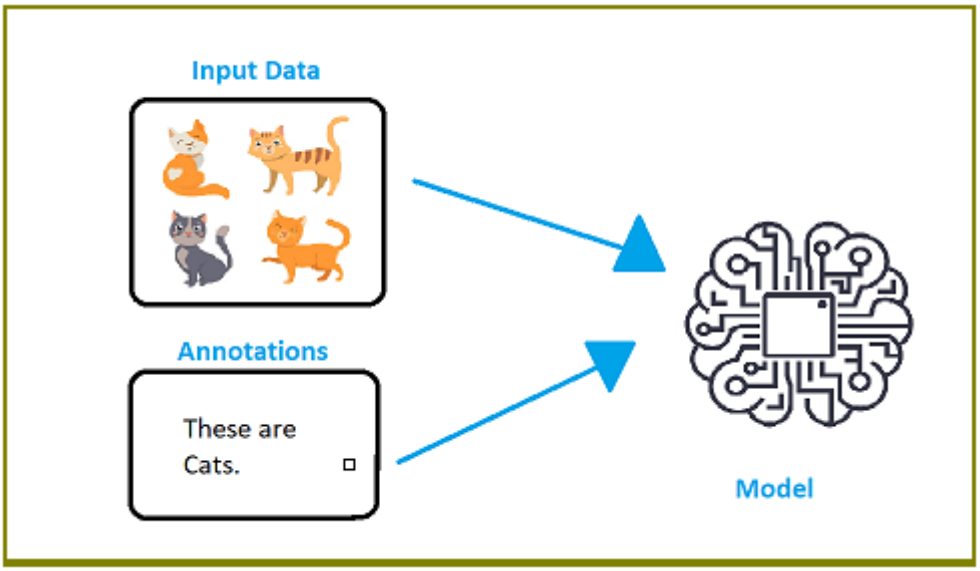
These labeled examples become our training data. We feed them to our machine learning model, kind of like a student studying for a test. The model analyzes the data, including the labels, and learns the relationship between the input (the image) and the desired output (the label "cat").

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \text{pred}_i)^2$$

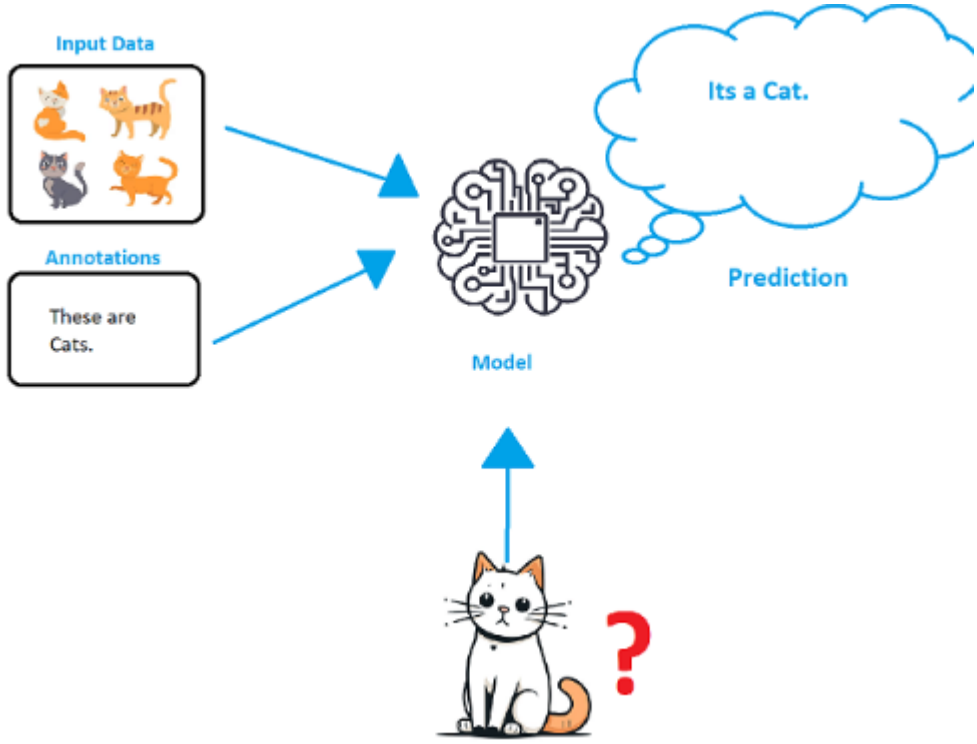
Where:

- n is the number of data points.
- y_i represents the true value of the dependent variable for the ith data point.
- pred_i represents the predicted value of the dependent variable for the ith data point.

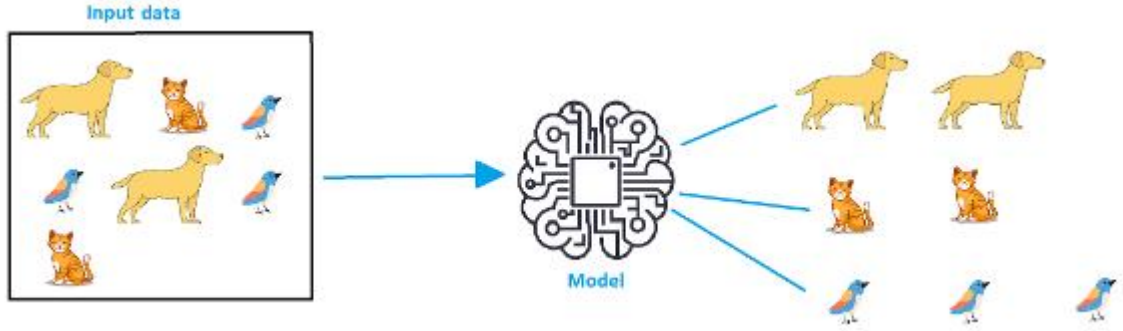
Once trained, the model can then take a completely new image, one it's never seen before, and based on what it learned from the labeled examples, predict what that image is. So, if you show it a picture of an cat, it might say, "Hey, that looks like an cat!"



In unsupervised learning, the model has to explore the data on its own, looking for hidden patterns and relationships. For instance, you might give the model pictures of cats, dogs, and birds. The model wouldn't know what a cat, dog, or bird is initially. But by analyzing the images, it might recognize similarities between the cat pictures (furry, pointy ears) and group them together. It might do the same for the dogs and birds based on their own unique features.



Let's understand this concept in a business context. Imagine you have a dataset containing information about your customers' annual income and their spending scores with annual income on the x-axis and spending score on the y-axis. When you plot this data where each customer would be represented by a dot on the graph, it might form a scatter plot.
At first, without any existing groups of customers, the plot might just look like random dots with no clear patterns.



Now, suppose you want to identify distinct groups of customers based on their spending behavior but without any predefined categories. This is where clustering comes into play. By applying clustering algorithms to the data, you can automatically group customers with similar spending patterns and income levels together.

These groups, called clusters, can reveal hidden segments within your customer base. For instance, you might identify a cluster of high-income customers with high spending scores, another cluster of low-income customers with moderate spending scores, and so on.

Once you have these customer segments, you can delve deeper into understanding their characteristics and behaviors which is incredibly valuable for businesses.

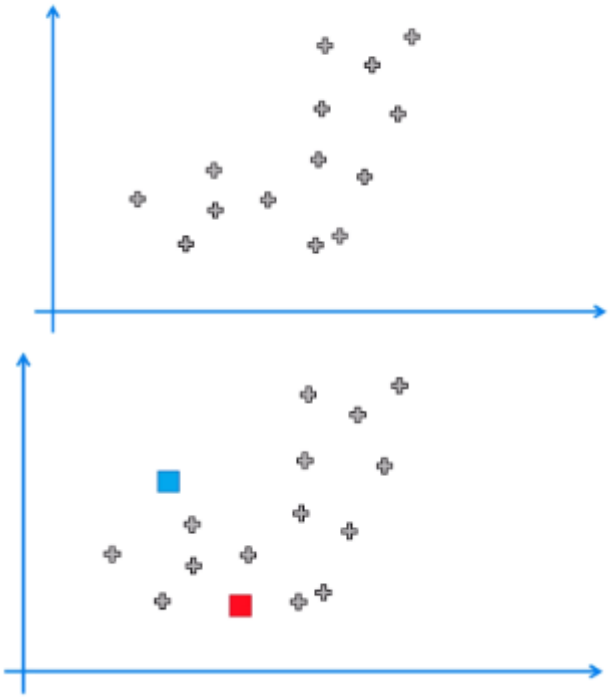
You can analyze why certain groups emerge and what this means for your business. For example, you might find that high-income customers with high spending scores prefer luxury products or personalized services, while low-income customers with moderate spending scores are more price-sensitive. Armed with this knowledge, you can tailor your marketing strategies accordingly. You can create targeted promotions, offers, and reminders tailored to each customer segment's preferences and needs. This enables you to optimize your marketing efforts, improve customer satisfaction, and ultimately drive business growth by better serving the diverse needs of your customer base.

So that's clustering, a powerful technique that offers valuable insights into your customer base. It's a departure from the supervised learning we discussed earlier, where the model learns from labeled data. Instead, clustering allows the model to explore unlabeled data and uncover hidden structures autonomously.

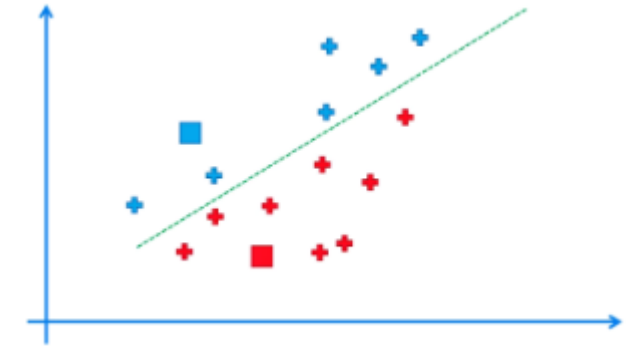
Now, let's delve into one of the most widely used clustering algorithms: K-means clustering.

K-Means Clustering is an Unsupervised Learning algorithm, which groups the unlabeled dataset into different clusters by a process known as clustering. Here K defines the number of pre-defined clusters that need to be created in the process, as if K=2, there will be two clusters, and for K=3, there will be three clusters, and so on. It is an iterative algorithm that divides the unlabeled dataset into k different clusters in such a way that each dataset belongs to only one group that has similar properties. So, here we have a scatter plot of our data points, and we want K-means clustering to create clusters. Since we don't have any training data (because we are dealing with an unlabeled dataset without predefined categories), we only have this data and want to create the clusters.

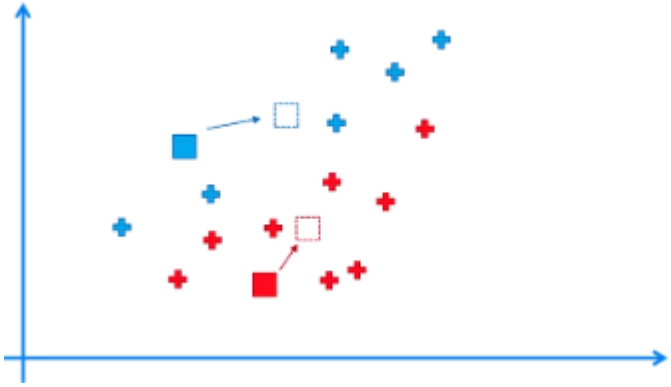
So how does it work? The first step is to decide how many clusters you want. We'll understand how to make this decision in a while, but for now, let's say we decide on two clusters. For each cluster, you need to place a randomly located centroid on the scatter plot wherever you like. It doesn't have to be one of the existing data points.



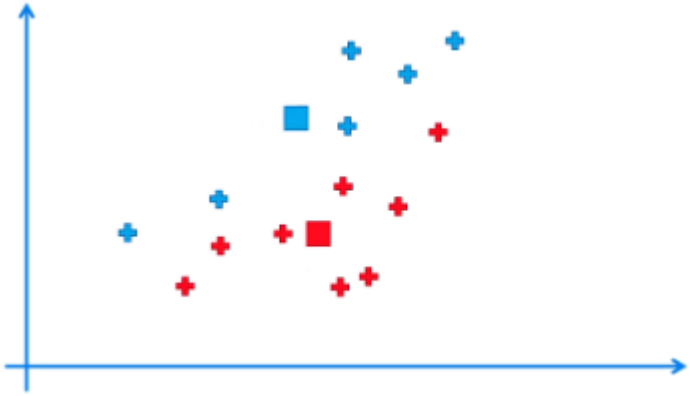
Now, what happens next is that K-means will assign each of the data points to the closest centroid. This can be done easily by drawing an equidistant line between the centroids. Any data point above this line is assigned to the blue centroid, and any data point below it is assigned to the red centroid.
Now, the next step is to calculate the center of mass or center of gravity for each of the preliminary clusters that we have identified. The centroid is not included in this calculation. For example, for the blue cluster, we need to take the average of all the x-coordinates and the average of all the y-coordinates of the data points within that cluster. This will give us the position of the new center of mass.



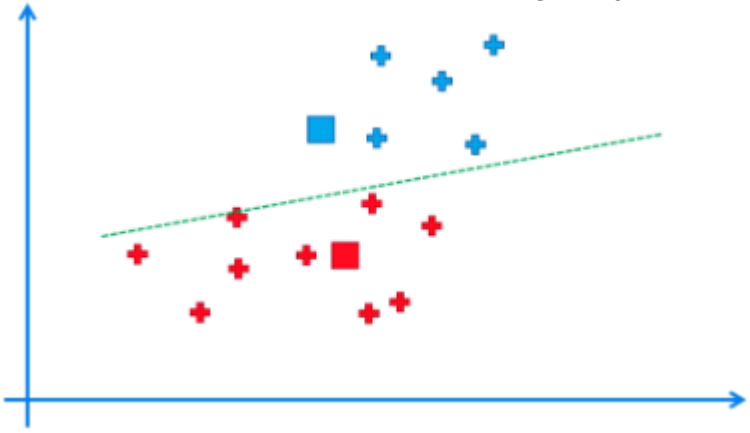
And then we move the centroid to those positions.



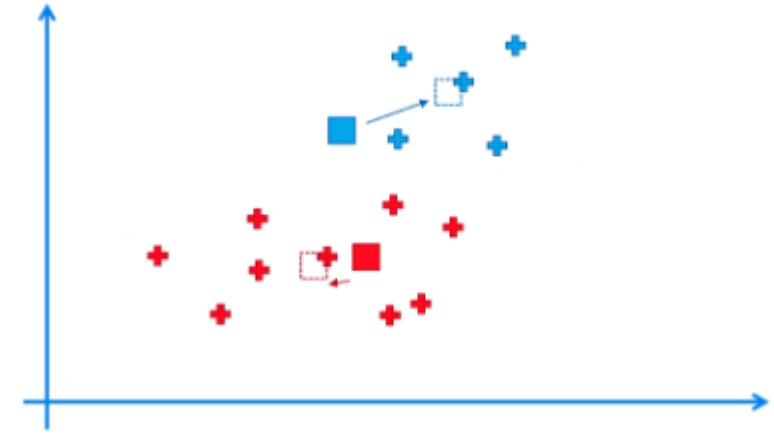
Once the centroids have moved, we repeat the process: reassign each data point to the closest centroid. Again, we draw an equidistant line between the centroids, then change the colors of the data points and reassign them accordingly.



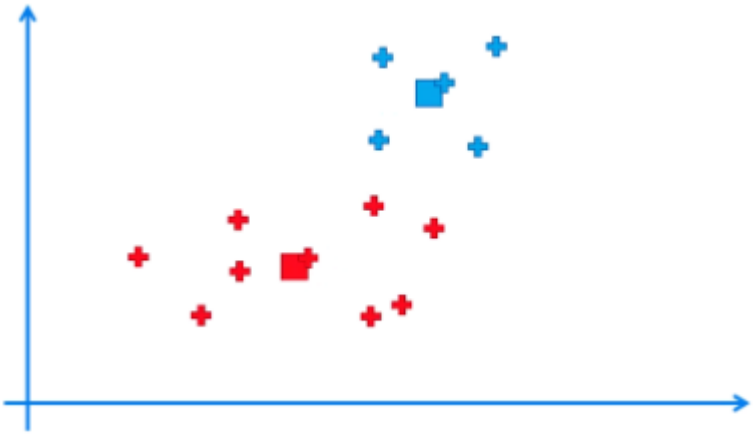
After that, we calculate the center of gravity for each cluster again.



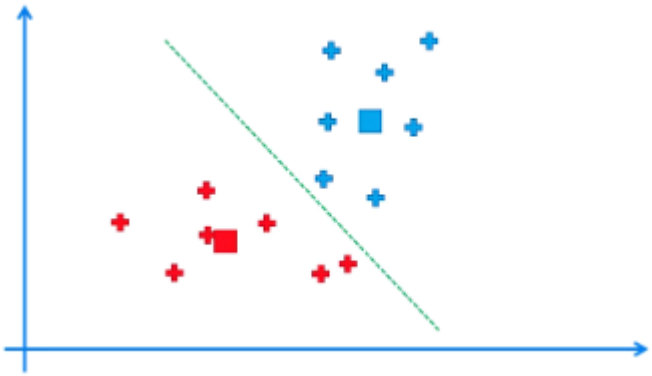
Move the centroids and do the process again.



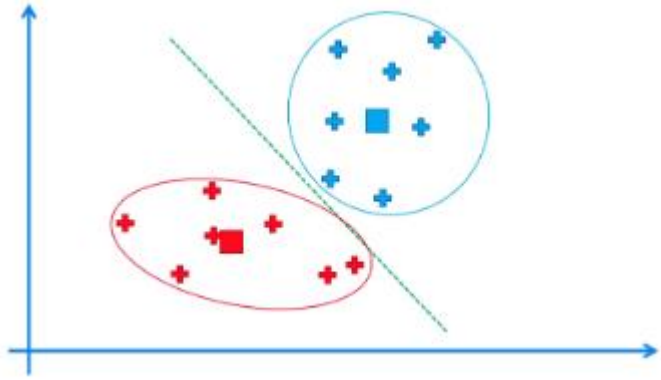
Reassign, calculate the center of mass, move the centroids, and repeat until we reach a point where further iterations do not change the assignments of the data points. Like in this state where we've drawn the equidistant line and already all the blue points are above and all the red points are below, we have reached the end of the K-means clustering step-by-step process.



And those are our final centroids! So that's how K-means clustering works. As you can see, it's a very simple and effective algorithm.



And now we have our two clusters, and the next step is to interpret what they mean. We can do this by analyzing the characteristics of the data points within each cluster and drawing insights from a business and domain knowledge perspective.

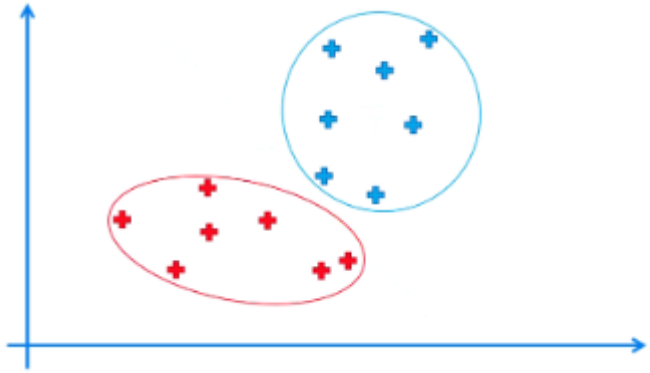


For example, if we're clustering customer data, we might find one cluster representing high-value customers and another representing potential churn risk. This information can then be used to inform targeted marketing campaigns or customer retention strategies.

Now that we understand how K-means clustering works, the question arises: how do we decide how many clusters to select in that initial step? It's up to us! The elbow method is one approach to help us make this decision.

However, sometimes you might already know the ideal number of clusters based on your domain knowledge and the problem statement. If that information isn't available, the elbow method is a good way to find the optimal number of clusters.

So, the elbow method requires us to look at this equation:



Where Pi represents a data point within a specific cluster.

C1 , C2 and so on represent the centroids of the corresponding clusters.

So the equation used in the elbow method is the within-cluster sum of squares or WCSS. It basically looks at the distance between each data point and its centroid within a cluster, and squares this distance. To calculate this distance, various methods like Euclidean distance or Manhattan distance can be employed.

For example, imagine you have two data points in a cluster: (1,2) and (4,6). Let's say the centroid (center) of this cluster is (3,4).

Now, we calculate the distance from each point to the centroid:

Distance from (1,2) to centroid (3,4) is $\sqrt{[(3-1)^2 + (4-2)^2]} = \sqrt{(4+4)} = \sqrt{8}$

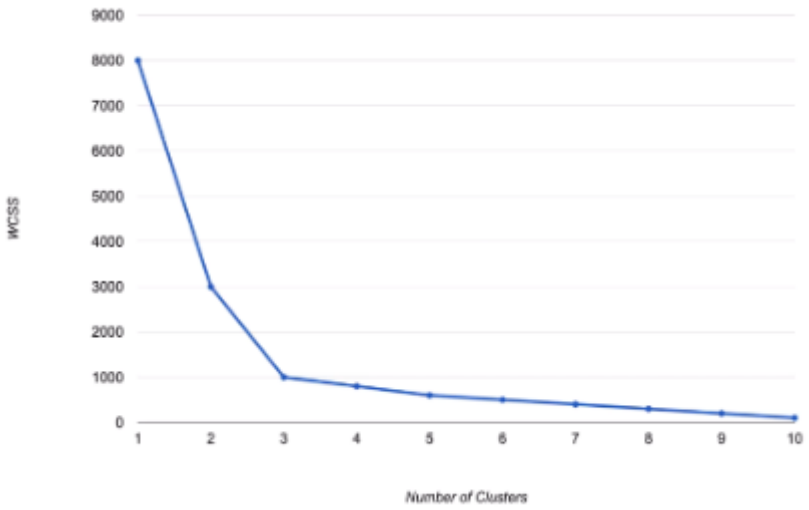
Now Square of the distance from (1,2) to centroid (3,4) = $(\sqrt{8})^2 = 8$

Next Distance from (4,6) to centroid (3,4) is $\sqrt{[(4-3)^2 + (6-4)^2]} = \sqrt{(1+4)} = \sqrt{5}$

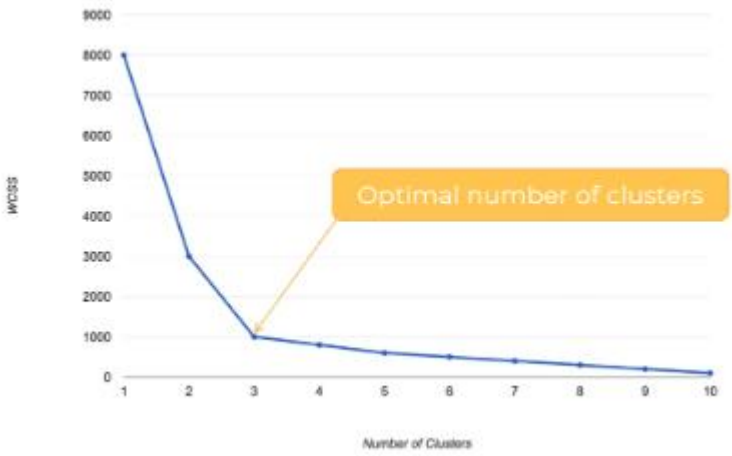
And Square of the distance from (4,6) to centroid (3,4) = $(\sqrt{5})^2 = 5$

Now, when we add these squared distances together, we get 8 + 5 = 13. This represents the within-cluster sum of squares (WCSS) for this example.

So in elbow method it performs K-means clustering on a dataset with different numbers of clusters (K values), typically ranging from 1 to 10. For each value of K, it calculates the within-cluster sum of squares (WCSS) value, which measures the compactness of the clusters.
So, two things to point out here. First, to calculate all these WCSS values for different options, we actually need the clusters to already exist. This means that every time, we first run the K-means clustering algorithm, and then we calculate WCSS. It's a bit backwards – we don't use the elbow method first to find the optimal number of clusters and then do K-means. Instead, we run K-means multiple times, finding WCSS for every setup with different numbers of clusters, like 2, 3, 4, 5, and so on, and then apply the elbow method.
Second, it's important to note that the more clusters we have, the smaller the WCSS becomes. So, we can keep increasing the number of clusters until we reach the maximum number of clusters, which is equal to the number of data points we have. At this point, the WCSS will actually be exactly 0 because each data point becomes its own centroid, and the distance is 0.
The plot that we build from this displays a curve illustrating the relationship between the calculated WCSS values (Within-Cluster Sum of Squares) and the number of clusters K. On the y-axis, we have WCSS, representing the compactness of the clusters, while on the x-axis, we have the number of clusters. This curve helps us visualize how the WCSS changes as we vary the number of clusters.



As you can see, it drops off all the way down to 0 as we discussed.
And the elbow method is very simple; it's actually a visual method. When you look at this graph, you search for the kink or the elbow. There it is. So, that is the optimal number of clusters.



And so that's how elbow method works.
Let's simplify this with an example:
Imagine you have a bunch of data points that represent customers' locations in a city. You want to group these customers into clusters based on their proximity to each other. The elbow method helps you decide how many clusters (K values) you should use.
So, you start with K = 1, meaning you're trying to group all the customers into one big cluster. You calculate the within-cluster sum of squares (WCSS), which tells you how spread out the customers are within this single cluster.
Next, you try K = 2, meaning you're dividing the customers into two clusters. You calculate the WCSS again for these two clusters.
You continue this process, trying K values from 1 to, let's say, 3 or 4. Each time, you calculate the WCSS, which tells you how compact or spread out the clusters are for that particular K value.
Now, two important things to note here:
Number of Clusters (K): The elbow method helps you decide the optimal number of clusters. You're looking for the K value where adding more clusters doesn't significantly decrease the WCSS anymore. It's like bending your arm to find the point where you feel the most 'elbow'.
Within-Cluster Sum of Squares (WCSS): This is a measure of how compact the clusters are. Lower WCSS values indicate that the points within each cluster are closer to each other, which is what you want in a good clustering. So, as you increase the number of clusters, you expect the WCSS to decrease because the clusters become more compact. However, at some point, adding more clusters won't decrease the WCSS much, indicating that adding more clusters doesn't significantly improve the clustering quality.
So, by plotting the number of clusters (K) against the corresponding WCSS values, you can visually identify the 'elbow point', which helps you determine the optimal number of clusters for your dataset.