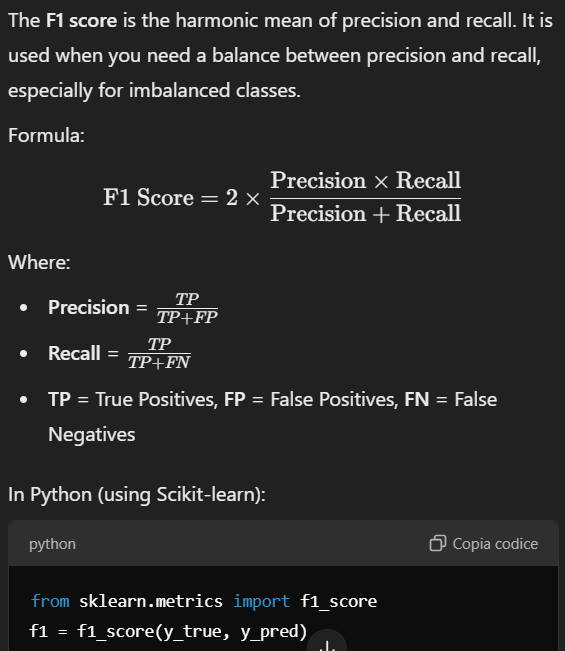
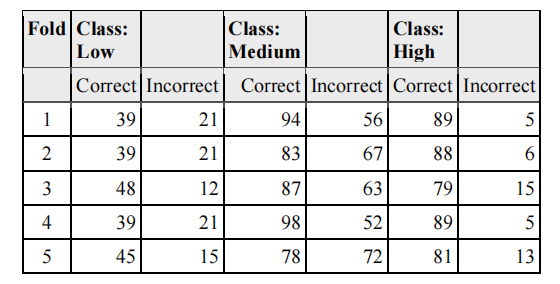
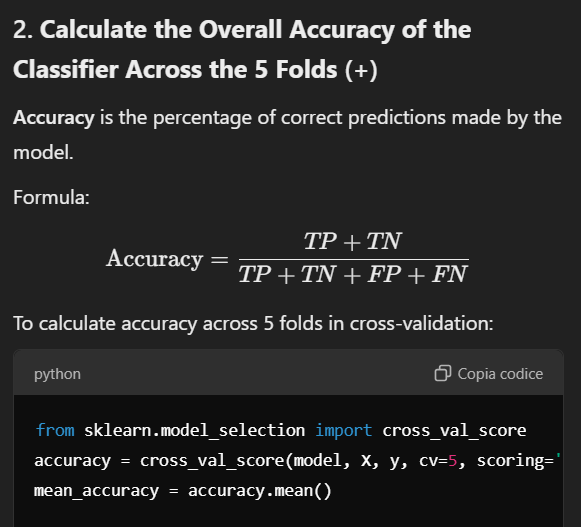
ML (2 day)

Calculate F1 measure ++



Calculate the overall accuracy of the classifier across the 5 folds +





### ****Step 1: Understanding the Table****

The table provides the following columns for each fold:

* **Class: Low, Medium, High**: The three different classes the model is predicting.
* **Correct**: The number of correct predictions for each class.
* **Incorrect**: The number of incorrect predictions for each class.

Here’s a breakdown of what’s given in the table:

| **Fold** | **Class: Low** | **Correct (Low)** | **Incorrect (Low)** | **Class: Medium** | **Correct (Medium)** | **Incorrect (Medium)** | **Class: High** | **Correct (High)** | **Incorrect (High)** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 39 | 21 | 94 | 56 | 89 | 5 | 39 | 21 | 83 |
| 2 | 39 | 21 | 83 | 67 | 88 | 6 | 39 | 21 | 78 |
| 3 | 48 | 12 | 87 | 63 | 79 | 15 | 39 | 21 | 81 |
| 4 | 39 | 21 | 98 | 52 | 89 | 5 | 39 | 21 | 81 |
| 5 | 45 | 15 | 78 | 72 | 81 | 13 | 39 | 21 | 89 |

The rows represent different folds in the cross-validation, and the columns give the counts for each class and the corresponding correct/incorrect predictions.

### ****Step 2: Calculate the Accuracy for Each Fold****

For each fold, we sum the correct and incorrect predictions for all classes and divide by the total number of predictions.

#### Example: Accuracy for Fold 1

* **Class: Low**: Correct = 39, Incorrect = 21
* **Class: Medium**: Correct = 94, Incorrect = 56
* **Class: High**: Correct = 89, Incorrect = 5

The **total correct predictions** = 39+94+89=22239 + 94 + 89 = 22239+94+89=222

The **total incorrect predictions** = 21+56+5=8221 + 56 + 5 = 8221+56+5=82

The **total predictions** = 222+82=304222 + 82 = 304222+82=304

Now, the accuracy for **Fold 1** is:

Repeat this for all 5 folds to get the accuracy for each fold.

### ****Step 3: Calculate the Mean Accuracy Across All Folds****

Once you calculate the accuracy for each fold, you can compute the **mean accuracy** across all folds to get a final measure of the model's performance.

### ****Final Output Example****:

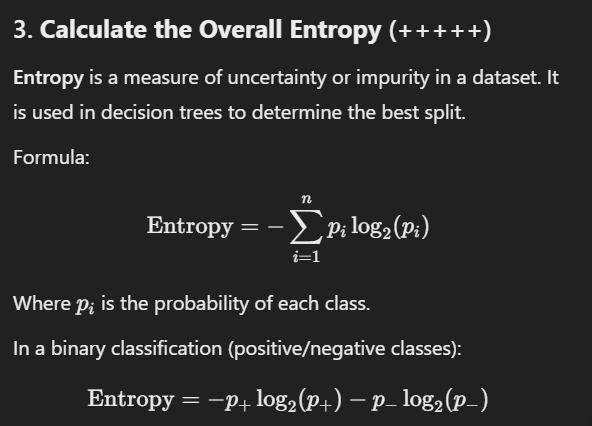
After processing, you might have results like this:

| **Metric** | **Fold 1** | **Fold 2** | **Fold 3** | **Fold 4** | **Fold 5** | **Mean Value** |
| --- | --- | --- | --- | --- | --- | --- |
| Accuracy | 73.03% | 72.00% | 74.10% | 75.50% | 71.80% | 73.09% |

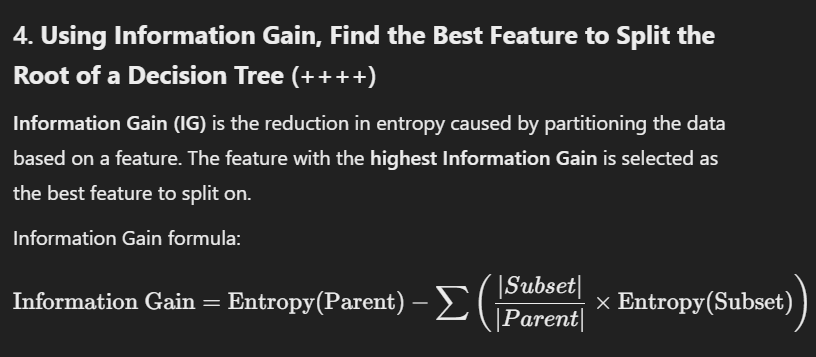
For skewed class distributions: ( aka imbalanced data)

1. Avoid relying solely on **accuracy**.
2. Use metrics like **precision**, **recall**, **F1 score**, **ROC-AUC**, and **PR-AUC**.
3. Analyze the confusion matrix for detailed insights.

Calculate the overall entropy +++++

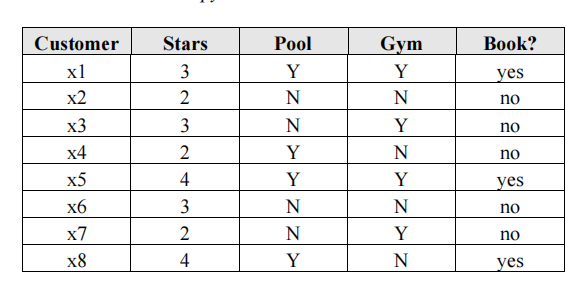


Using Information Gain, find the best feature to split the root of a Decision Tree classifier (or binary split) ++++



To determine the best feature to split the root of a **Decision Tree** classifier, we will use **Information Gain (IG)**, which is a metric that measures how much uncertainty (or entropy) is reduced when a feature is used to split the data.

### ****Steps to Calculate Information Gain:****

-

1. **Calculate the entropy of the entire dataset** (which we have already done).
2. **Calculate the entropy of each feature** by splitting the dataset based on the feature.
3. **Calculate the Information Gain** for each feature, which is the difference between the entropy of the dataset and the weighted entropy of the splits based on that feature.
4. **Select the feature with the highest Information Gain** as the best feature to split on at the root of the decision tree.

### ****Step 1: Overall Entropy (Already Calculated)****

From the previous step, we know the **entropy of the entire dataset** is approximately **0.955**.

### ****Step 2: Calculate the Entropy for Each Feature****

We will now calculate the **entropy** for each feature (Stars, Pool, and Gym) by splitting the dataset based on those features and calculating the weighted entropy of the resulting subsets.

Let's calculate the **entropy for each feature** one by one.

### ****Feature: Stars****

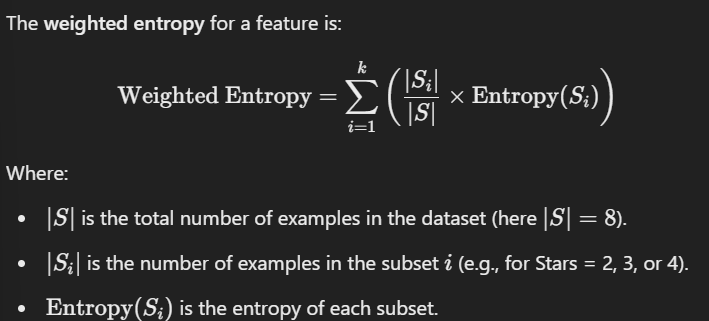
The **Stars** feature has 3 distinct values: **2, 3, 4**.

#### ****Split the dataset by Stars:****

* **Stars = 2**: Customers with 2 stars
  + (x2, x4, x6, x7) — 4 samples
  + **Book? = no**: 4 instances of "no", 0 instances of "yes"
  + Entropy = 0 (since all instances belong to the same class)
* **Stars = 3**: Customers with 3 stars
  + (x1, x3, x6) — 3 samples
  + **Book? = no**: 3 instances of "no", 0 instances of "yes"
  + Entropy = 0 (since all instances belong to the same class)
* **Stars = 4**: Customers with 4 stars
  + (x5, x8) — 2 samples
  + **Book? = yes**: 1 instance of "yes", 1 instance of "no"
  + Entropy = 1 (since both classes are equally represented)

### ****Weighted Entropy Formula****

* The **weighted entropy** for a feature is:



### ****Steps to Calculate Weights for Stars:****

#### 1. ****Subset Sizes for Stars****

The **Stars** feature has three distinct values: **2, 3, and 4**. Count the number of examples for each value:

* **Stars = 2**: ∣S2∣=3|S\_2| = 3∣S2​∣=3 (x2, x4, x7)
* **Stars = 3**: ∣S3∣=3|S\_3| = 3∣S3​∣=3 (x1, x3, x6)
* **Stars = 4**: ∣S4∣=2|S\_4| = 2∣S4​∣=2 (x5, x8)

#### 2. ****Calculate the Weight for Each Subset****

The weight is simply the fraction of examples in each subset relative to the total dataset.

Weight for Stars = 2=∣S2∣∣S∣=38=0.375\text{Weight for Stars = 2} = \frac{|S\_2|}{|S|} = \frac{3}{8} = 0.375Weight for Stars = 2=∣S∣∣S2​∣​=83​=0.375 Weight for Stars = 3=∣S3∣∣S∣=38=0.375\text{Weight for Stars = 3} = \frac{|S\_3|}{|S|} = \frac{3}{8} = 0.375Weight for Stars = 3=∣S∣∣S3​∣​=83​=0.375 Weight for Stars = 4=∣S4∣∣S∣=28=0.25\text{Weight for Stars = 4} = \frac{|S\_4|}{|S|} = \frac{2}{8} = 0.25Weight for Stars = 4=∣S∣∣S4​∣​=82​=0.25

### ****Weighted Entropy Calculation for Stars****

We already know:

* **Entropy(S\_2)** = 0 (all "no").
* **Entropy(S\_3)** = 0 (all "no").
* **Entropy(S\_4)** = 1 (equal split of "yes" and "no").

Using the weights and entropies, calculate the weighted entropy:

Weighted EntropyStars=(0.375×0)+(0.375×0)+(0.25×1)\text{Weighted Entropy}\_{\text{Stars}} = \left( 0.375 \times 0 \right) + \left( 0.375 \times 0 \right) + \left( 0.25 \times 1 \right)Weighted EntropyStars​=(0.375×0)+(0.375×0)+(0.25×1) Weighted EntropyStars=0+0+0.25=0.25\text{Weighted Entropy}\_{\text{Stars}} = 0 + 0 + 0.25 = 0.25Weighted EntropyStars​=0+0+0.25=0.25

### ****Conclusion****

* The **weights** for each subset are proportional to the number of examples in each subset:
  + Weight for Stars = 2=0.375\text{Weight for Stars = 2} = 0.375Weight for Stars = 2=0.375
  + Weight for Stars = 3=0.375\text{Weight for Stars = 3} = 0.375Weight for Stars = 3=0.375
  + Weight for Stars = 4=0.25\text{Weight for Stars = 4} = 0.25Weight for Stars = 4=0.25
* The **weighted entropy** for Stars is **0.25**.

#### ****Calculate the weighted entropy for Stars:****

* Entropy of Stars = 2: 0×48=00 \times \frac{4}{8} = 00×84​=0
* Entropy of Stars = 3: 0×38=00 \times \frac{3}{8} = 00×83​=0
* Entropy of Stars = 4: 1×28=0.251 \times \frac{2}{8} = 0.251×82​=0.25

Total entropy for the **Stars** feature:

EntropyStars=0+0+0.25=0.25\text{Entropy}\_{\text{Stars}} = 0 + 0 + 0.25 = 0.25EntropyStars​=0+0+0.25=0.25

### ****Feature: Pool****

The **Pool** feature has 2 distinct values: **Y** (yes) and **N** (no).

#### ****Split the dataset by Pool:****

* **Pool = Y**: Customers with access to a pool
  + (x1, x4, x5, x8) — 4 samples
  + **Book? = yes**: 2 instances of "yes", 2 instances of "no"
  + Entropy = 1 (since both classes are equally represented)
* **Pool = N**: Customers without access to a pool
  + (x2, x3, x6, x7) — 4 samples
  + **Book? = no**: 4 instances of "no", 0 instances of "yes"
  + Entropy = 0 (since all instances belong to the same class)

#### ****Calculate the weighted entropy for Pool:****

* Entropy of Pool = Y: 1×48=0.51 \times \frac{4}{8} = 0.51×84​=0.5
* Entropy of Pool = N: 0×48=00 \times \frac{4}{8} = 00×84​=0

Total entropy for the **Pool** feature:

EntropyPool=0.5+0=0.5\text{Entropy}\_{\text{Pool}} = 0.5 + 0 = 0.5EntropyPool​=0.5+0=0.5

### ****Feature: Gym****

The **Gym** feature also has 2 distinct values: **Y** (yes) and **N** (no).

#### ****Split the dataset by Gym:****

* **Gym = Y**: Customers with access to a gym
  + (x1, x3, x5) — 3 samples
  + **Book? = yes**: 2 instances of "yes", 1 instance of "no"
  + Entropy = 0.918 (calculated as: −23log⁡223−13log⁡213=0.918-\frac{2}{3} \log\_2 \frac{2}{3} - \frac{1}{3} \log\_2 \frac{1}{3} = 0.918−32​log2​32​−31​log2​31​=0.918)
* **Gym = N**: Customers without access to a gym
  + (x2, x4, x6, x7) — 4 samples
  + **Book? = no**: 4 instances of "no", 0 instances of "yes"
  + Entropy = 0 (since all instances belong to the same class)

#### ****Calculate the weighted entropy for Gym:****

* Entropy of Gym = Y: 0.918×38=0.34350.918 \times \frac{3}{8} = 0.34350.918×83​=0.3435
* Entropy of Gym = N: 0×48=00 \times \frac{4}{8} = 00×84​=0

Total entropy for the **Gym** feature:

EntropyGym=0.3435+0=0.3435\text{Entropy}\_{\text{Gym}} = 0.3435 + 0 = 0.3435EntropyGym​=0.3435+0=0.3435

### ****Step 3: Calculate Information Gain for Each Feature****

Now that we have the **entropy of the entire dataset** (which is **0.955**), we can calculate the **Information Gain (IG)** for each feature.

**Information Gain** is the difference between the **entropy of the original dataset** and the **weighted entropy of the split** for each feature:

Information Gain=Entropy(S)−∑(∣Si∣∣S∣×Entropy(Si))\text{Information Gain} = \text{Entropy}(S) - \sum \left(\frac{|S\_i|}{|S|} \times \text{Entropy}(S\_i)\right)Information Gain=Entropy(S)−∑(∣S∣∣Si​∣​×Entropy(Si​))

Where:

* SSS is the dataset.
* SiS\_iSi​ represents the subsets of data after the split.

#### ****Information Gain for Stars****:

IGStars=0.955−0.25=0.705IG\_{\text{Stars}} = 0.955 - 0.25 = 0.705IGStars​=0.955−0.25=0.705

#### ****Information Gain for Pool****:

IGPool=0.955−0.5=0.455IG\_{\text{Pool}} = 0.955 - 0.5 = 0.455IGPool​=0.955−0.5=0.455

#### ****Information Gain for Gym****:

IGGym=0.955−0.3435=0.6115IG\_{\text{Gym}} = 0.955 - 0.3435 = 0.6115IGGym​=0.955−0.3435=0.6115

### ****Step 4: Conclusion****

Based on the **Information Gain** for each feature:

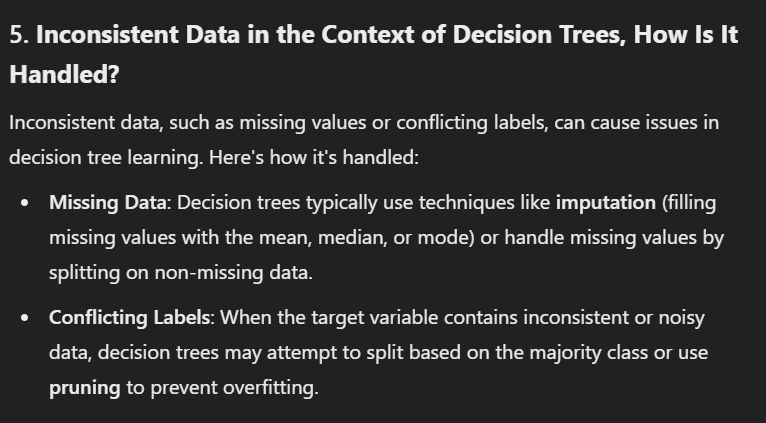
* **Stars** has the highest Information Gain (0.7050.7050.705), so it is the **best feature to split the root of the decision tree**.

Thus, **"Stars"** is the feature that would be selected to split the root node in the decision tree, as it provides the highest reduction in entropy.

### ****Final Results****:

* **Best feature to split on**: **Stars**
* **Information Gain for Stars**: **0.705**
* **Information Gain for Pool**: **0.455**
* **Information Gain for Gym**: **0.6115**

inconsistent data in the context of decision trees, how it is handled? +



**Pruning**:

* 1. **Overfitting** caused by inconsistent data can be controlled by **pruning** the tree. Pruning removes nodes or subtrees that don't improve generalization performance, reducing the impact of noisy or conflicting data points.

**Entropy/Gini Impurity**:

* 1. Decision trees inherently use metrics like **entropy** or **Gini impurity** to decide splits, even when conflicting labels exist. These metrics ensure that the tree splits in a way that minimizes impurity or maximizes information gain.

### ****What is Pruning?****

Pruning is a technique used in decision trees to reduce their size by removing parts of the tree that are not contributing significantly to the model’s accuracy. It helps in preventing **overfitting**, which occurs when the tree becomes too complex and starts capturing noise in the data rather than generalizable patterns.

#### ****Types of Pruning****:

**Pre-Pruning (Early Stopping)**:

* 1. The tree stops growing before it perfectly classifies the training data.
  2. Common pre-pruning strategies:
     1. Set a maximum depth for the tree.
     2. Require a minimum number of samples at a leaf node (min\_samples\_leaf).
     3. Set a threshold for the minimum impurity decrease (min\_impurity\_decrease).

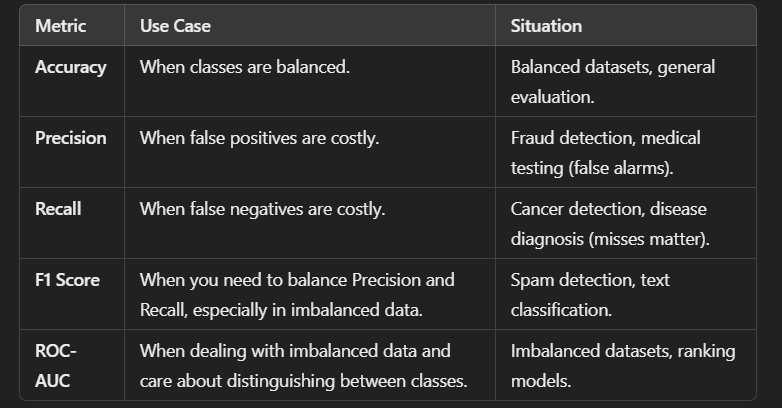
**Post-Pruning**:

* 1. The tree is fully grown first, and then nodes or subtrees are removed based on certain criteria.
  2. Common post-pruning strategies:
     1. **Cost Complexity Pruning**: Used in CART. A cost-complexity parameter (e.g., ccp\_alpha in Scikit-learn) is tuned to balance model complexity and performance.
     2. **Error-based Pruning**: Subtrees are removed if their removal doesn’t significantly increase error on a validation set.

#### ****Why Pruning is Important****:

* **Reduces Overfitting**: Removes unnecessary complexity from the model.
* **Improves Generalization**: Simplifies the model, making it perform better on unseen data.
* **Speeds Up Inference**: A smaller tree means faster predictions.

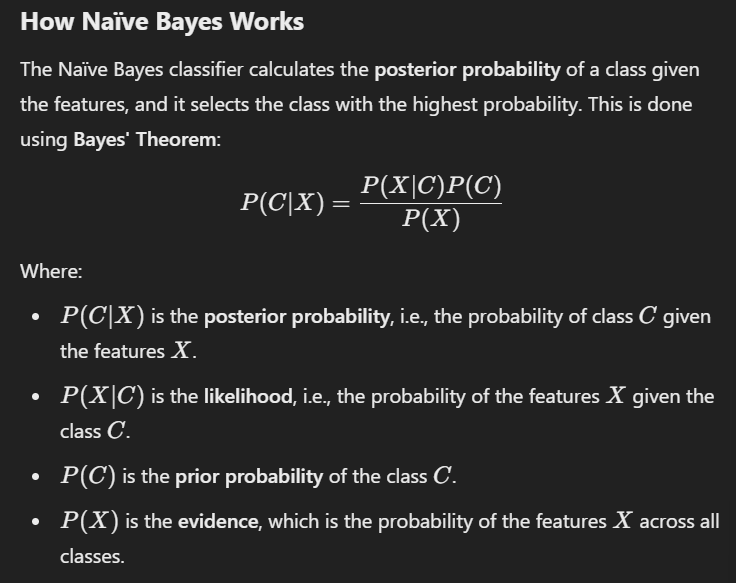
performance metric for evaluation ++

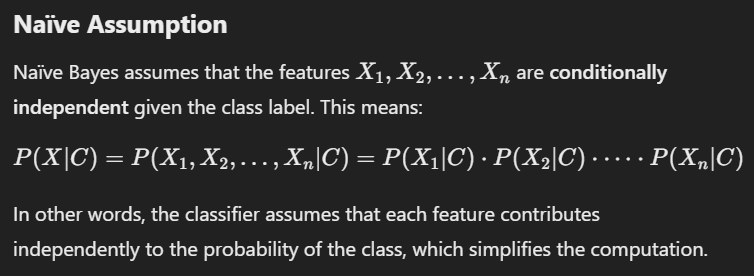


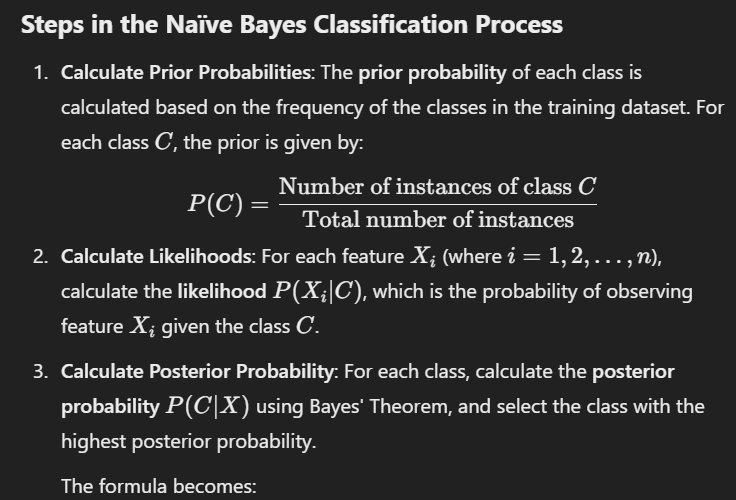
contingency table of conditional and prior probabilities that would be used by Naïve Bayes

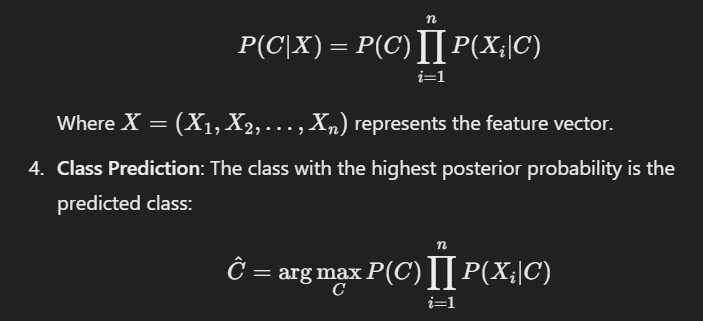
Naïve Bayes classifier +++

The **Naïve Bayes** **classifier** is a probabilistic machine learning algorithm based on **Bayes' Theorem**, and it is called "naïve" because it assumes that all features are **independent** of each other, which is often not the case in real-world data. Despite this simplifying assumption, Naïve Bayes can perform surprisingly well in many practical applications, especially in text classification tasks such as **spam detection** and **sentiment analysis**



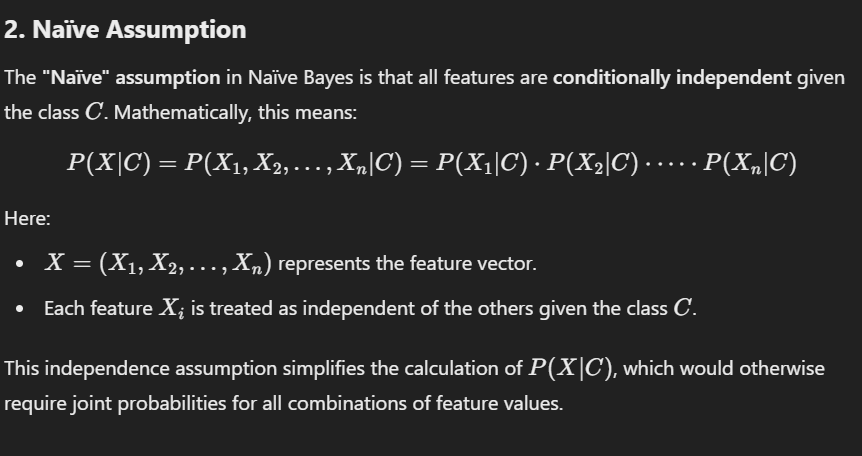


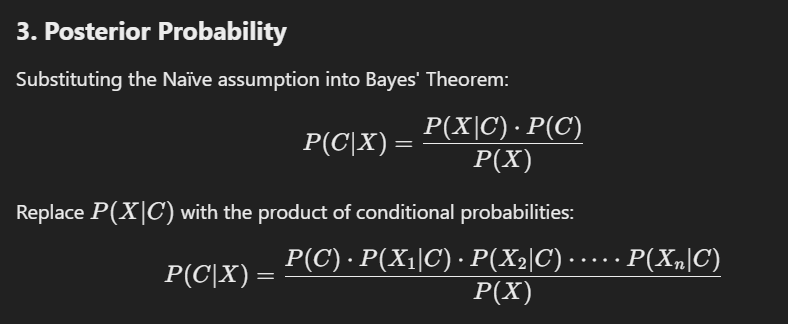


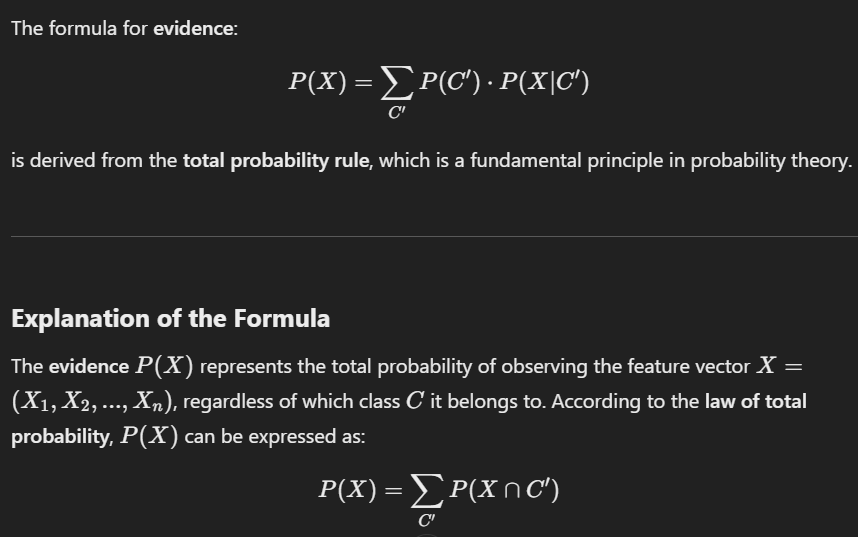
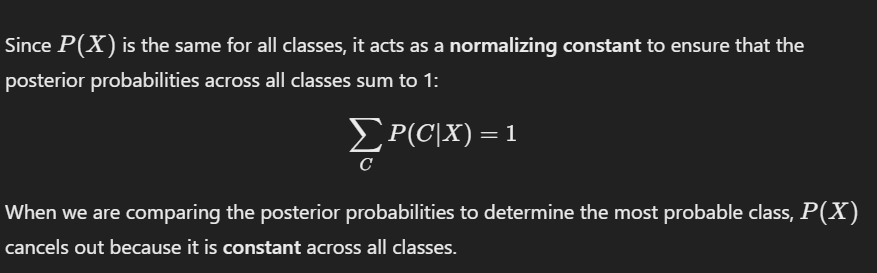
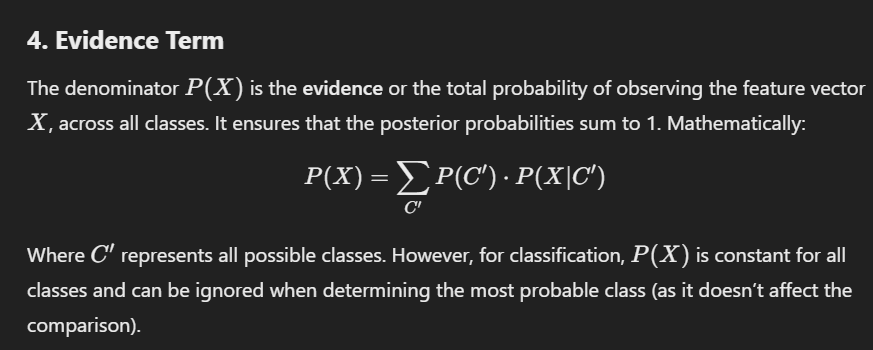


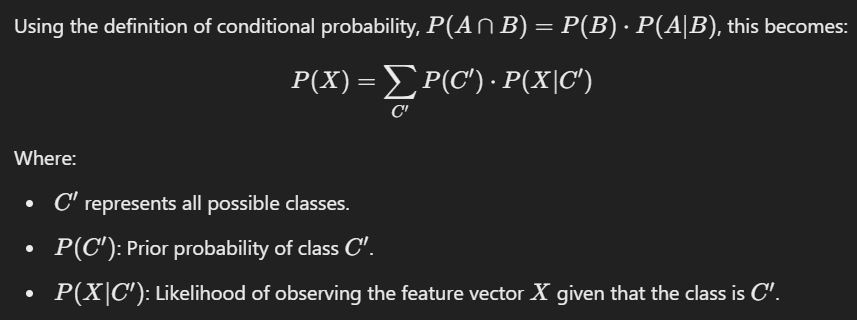
U get further explanations onwhy P(X) disappear on chat with gpt

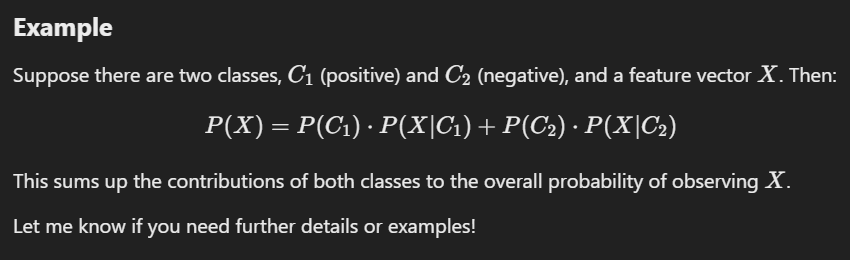
To understand how u get to final form



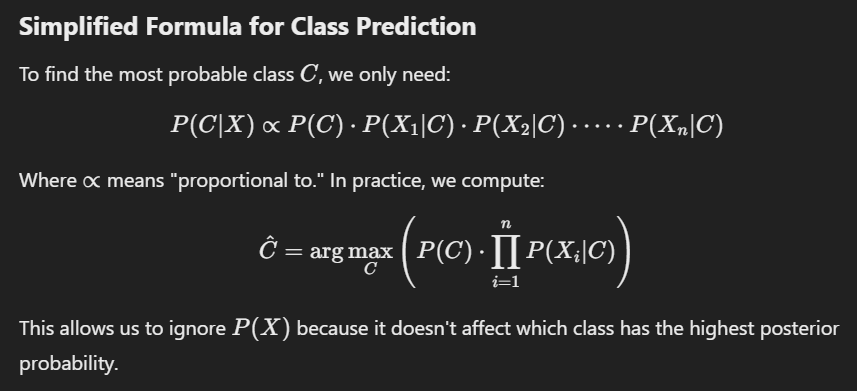








Therefore



### ****Contingency Table of Conditional and Prior Probabilities for Naïve Bayes****

A **contingency table** is a matrix that shows the relationship between two categorical variables, often used to compute the **prior probabilities** and **conditional probabilities** required for a **Naïve Bayes classifier**. It plays a crucial role in understanding and calculating probabilities for the model.

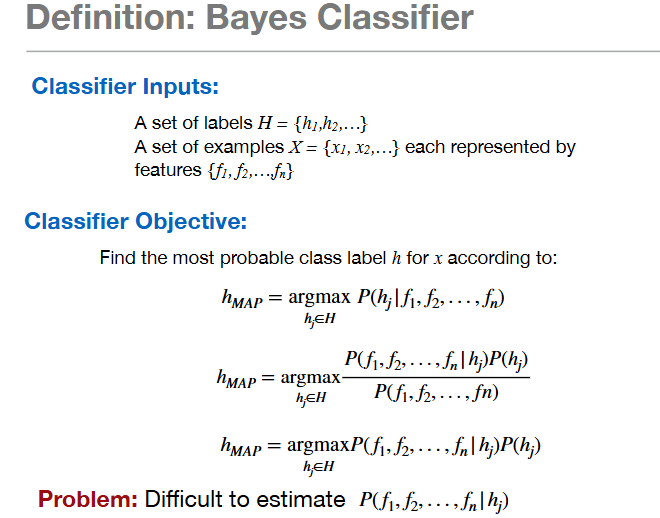
### ****Relation Between Contingency Table and Naïve Bayes****

The contingency table provides the raw counts needed to calculate the **prior** and **conditional probabilities**:

1. **Rows** represent the class labels (CCC).
2. **Columns** represent the features (XiX\_iXi​).
3. Each cell represents the count of instances where a particular feature value and class label co-occur.

### ****Defining the Naïve Bayes Model****

A **Naïve Bayes classifier** is a probabilistic machine learning model based on **Bayes' Theorem**, assuming **feature independence** (naïve assumption)



To build a Naïve Bayes classifier, we need to calculate the **prior probabilities** and the **conditional probabilities** of each feature given the class label (Flu?).

* **Prior probability**
* **Conditional probabilities**

### ****Calculate Prior Probabilities****

### ****Calculate Conditional Probabilities****

Now we need to calculate the **conditional probabilities** for each feature given the class label (Flu = Yes or Flu = No).

Do that for all features to find contingency table

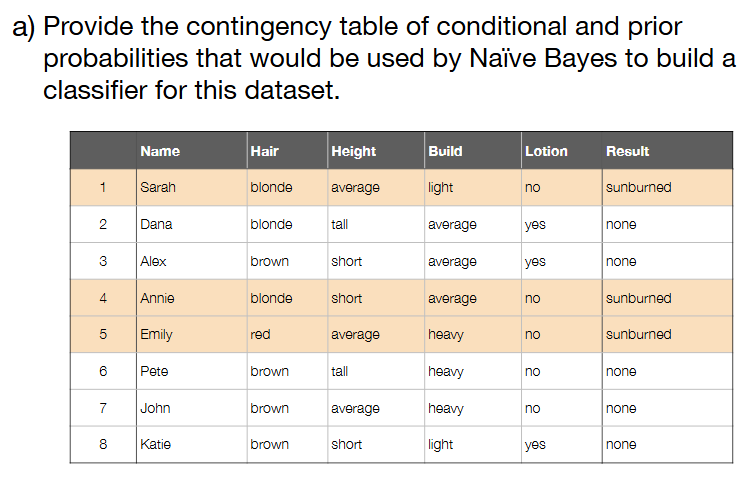
How to calculate predicted class in naive bayees in exercise ?

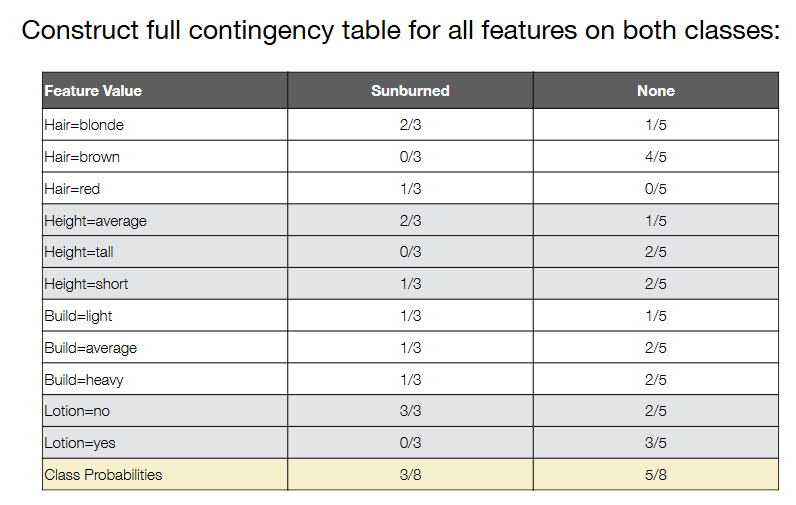
After u found the contingency table

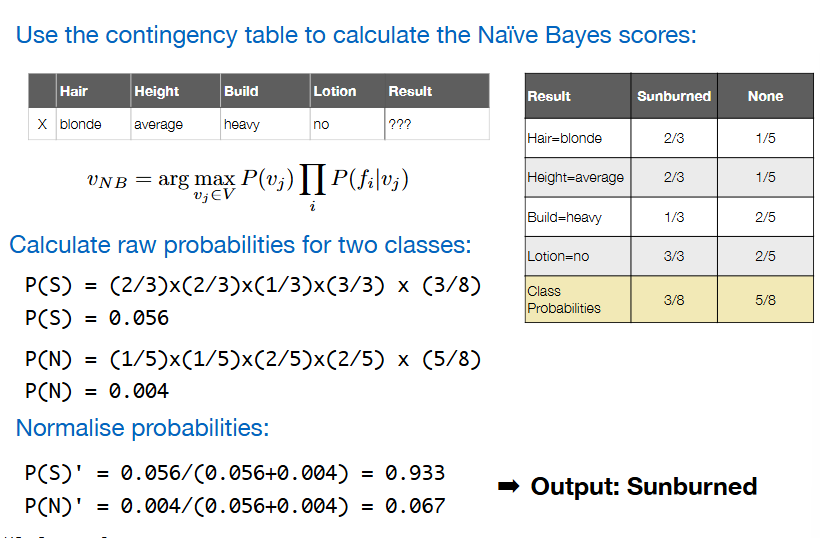
Calculate the postum (conditional) probability of the targets

And them normalize

Rmnbr to normalize C yes with C no if u use semplified form







The normalization used in this example is **probability normalization**, where the probabilities are scaled so that their total sum equals 1. This type of normalization is common in **Naïve Bayes** and other probabilistic models when comparing the probabilities of multiple hypotheses (class labels).

We have all other sorts of normalization that we will use later

4.(a) Given the nature of the ` AthleteSelection` data which  
would be the best of the Naive Bayes options in scikit-  
learn for that classification task?  
Gaussian Naive Bayes is an option because the features are real  
values - not counts or categories. The data is probably not  
exactly Gaussian but probably close enough.  
We could discretize the data and then use Categorical NB.  
20

kNN classifiers ( and distance weighted) (best k choice) (predict exercise) (difference btw unweighted kNN and weighted kNN)+++

distance functions++++

The **k-Nearest Neighbors (kNN)** algorithm is a simple, yet powerful, classification (and regression) algorithm that **makes predictions based on the **k closest data points**** to a given sample. It's a **non-parametric** algorithm, meaning it doesn't make any assumptions about the underlying distribution of the data.

Let's break down the key aspects of **kNN classifiers**, focusing on:

1. **Unweighted vs Weighted kNN**
2. **Choosing the Best k**
3. **Distance Functions**

### ****1. kNN Classifiers****:

**Unweighted kNN**: In this version of kNN, each of the **k nearest neighbors** contributes equally to the final prediction. For classification, the majority class among the k neighbors is predicted. For regression, the average of the neighbors' values is taken.

**Weighted kNN**: In this version of kNN, the influence of each neighbor is weighted by its **distance** to the query point. Closer neighbors have a higher influence on the prediction, which can sometimes improve performance, especially when the data is not uniformly distributed.

**Weighted kNN** is particularly useful when data points have different densities, as it accounts for the fact that closer points should contribute more to the final prediction.

### ****2. Choosing the Best k****:

The parameter **k** is the number of nearest neighbors to consider when making a prediction.

**How to choose k**:

* + **Small k (e.g., k=1)**: The model becomes more sensitive to noise, and overfitting is a concern. It might make overly specific predictions based on a single neighbor.
  + **Large k**: The model becomes smoother and may generalize better. However, too large a k can lead to underfitting, as the model may fail to capture the finer details of the data.

**Choosing the best k**: Typically, **cross-validation** is used to find the optimal value for k. A range of k values are tested, and the k that gives the best performance (e.g., highest accuracy) is selected.

* + For example:

python

Copia codice

from sklearn.model\_selection import cross\_val\_scorefrom sklearn.neighbors import KNeighborsClassifier

k\_values = range(1, 21)

cv\_scores = []

for k in k\_values:

knn = KNeighborsClassifier(n\_neighbors=k)

cv\_score = cross\_val\_score(knn, X, y, cv=5, scoring='accuracy')

cv\_scores.append(cv\_score.mean())

optimal\_k = k\_values[cv\_scores.index(max(cv\_scores))]print(f"Best k: {optimal\_k}")

This code finds the optimal k by using **cross-validation** to evaluate each value of k from 1 to 20.

* **Lazy Learning**: k-NN does not explicitly build a **model** during training. Instead:
  + It simply stores the training data.
  + The "training phase" is trivial since no abstraction or mathematical structure is created from the data.
* **Prediction**: The "model" is implicit—it uses the stored training data directly to compute distances to the query point and determine the closest neighbors.

### ****Contrast with Non-Lazy (Eager) Learning****:

**Eager Learning**:

* + Algorithms like Linear Regression, Support Vector Machines, or Neural Networks create an explicit model (e.g., a mathematical function or decision boundary).
  + Training involves finding parameters (e.g., weights, biases) that generalize the data.

**Lazy Learning** (e.g., k-NN):

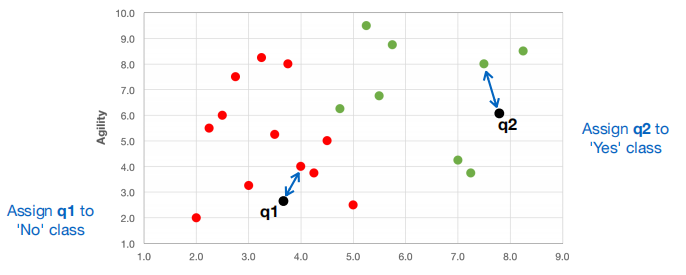
* + No such parameters or explicit function are learned.
  + The algorithm waits until prediction time to process the data.

.

### ****k-NN Classifier types****

**1-Nearest Neighbor (1-NN)**:

* 1. **Rule**: For a given query point q, find the single closest labeled example x from the training set.
  2. The label assigned to q is the same as x.
  3. **Key Characteristic**: Lazy learning—does not build a model but directly uses training data for classification.



**k-Nearest Neighbors (k-NN)**:

* 1. A generalization of 1-NN where the k closest neighbors are considered.
  2. **Decision Rule**: The majority label among the k nearest neighbors determines the label of q.

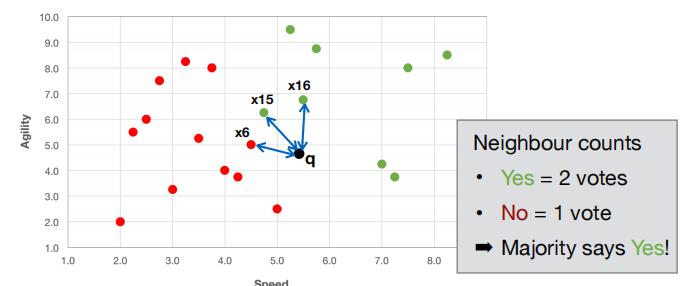
### ****Majority Voting in k-NN****

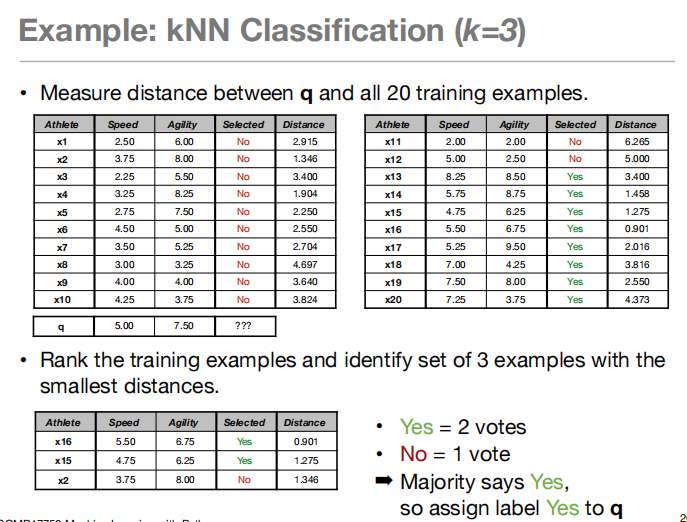
**Voting Process**:

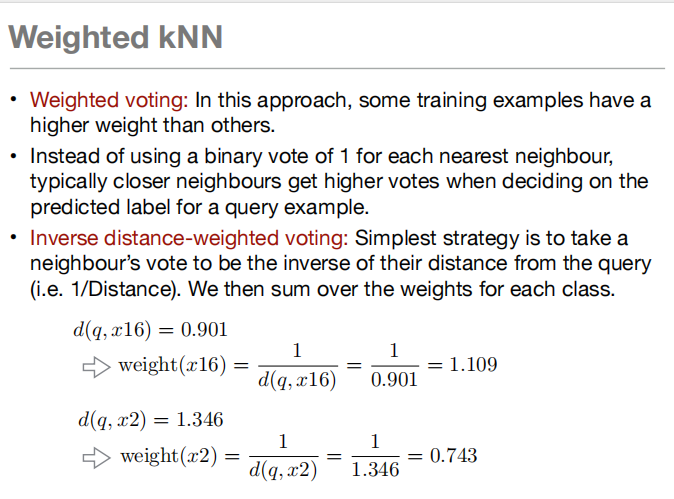
* 1. For k=3:
     1. Identify the 3 nearest neighbors of q using euclidian distance
     2. Count the votes for each label among these neighbors.
  2. Assign the label with the majority votes.

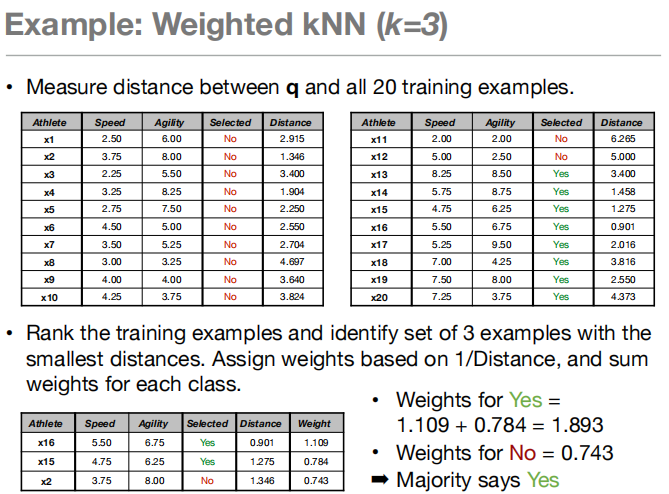
**Tie-Breaking**:

* 1. In case of a tie (e.g., k=4 with 2 votes each for two labels):
     1. Break the tie randomly.
     2. Alternatively, sum the distances and select the label with the smaller total distance.









### ****3. Distance Functions****:

In **kNN**, the choice of **distance function** is crucial because it defines how the "closeness" of data points is measured. Several distance functions can be used, and the choice of distance depends on the nature of the data.

#### ****Common Distance Functions****:

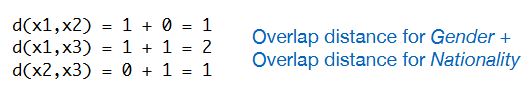
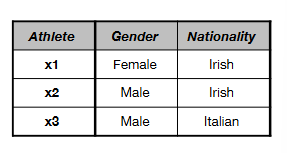
(more generic)  
Local distance function: Measure the  
distance between two examples based  
on a single feature

Global distance function: Measure the distance between two  
examples based on the combination of the local distances  
across all features

Overlap function: Simplest local distance  
measure. **Returns 0 if the two values for  
a feature are equal and 1 otherwise.**Generally suitable for categorical data

Hamming distance: Global distance function which is the sum of  
the overlap differences across all features - i.e. number of features  
on which two examples disagree.

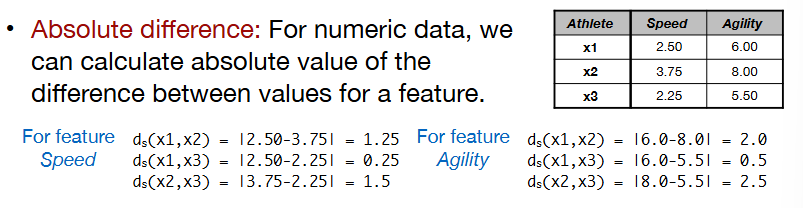
**Best for**: Categorical data with binary or nominal attributes.



python

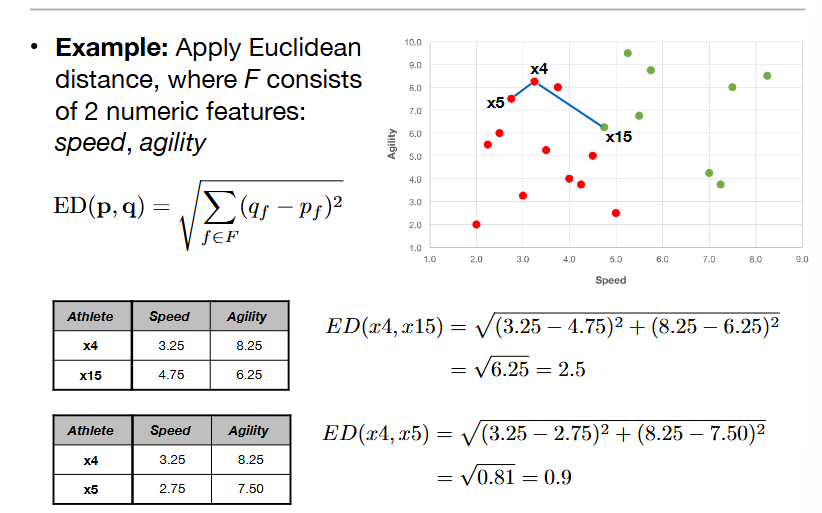
Copia codice

knn = KNeighborsClassifier(n\_neighbors=5, metric='hamming')



**Euclidean Distance** (most common)

* + Euclidean distance is the **straight-line distance** between two points in a multi-dimensional space (essentially, the length of the shortest path between the points).



**Best for**: Continuous numerical features where the relationship between the features is linear and distance-based.

**Python Example** (default in scikit-learn kNN):

python

Copia codice

from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n\_neighbors=5, metric='euclidean')

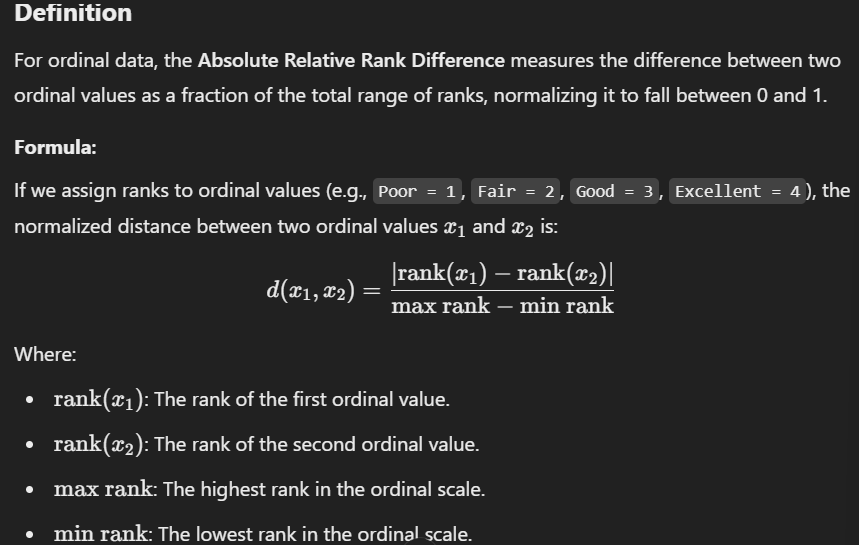
Wa if we have data type ordinal?

The **Absolute Relative Rank Difference** normalized as a local distance function is used for ordinal data, where the categories have a meaningful order but no specific numeric distance between them. This type of normalization is especially useful for datasets with ordinal variables such as {Poor, Fair, Good, Excellent}.

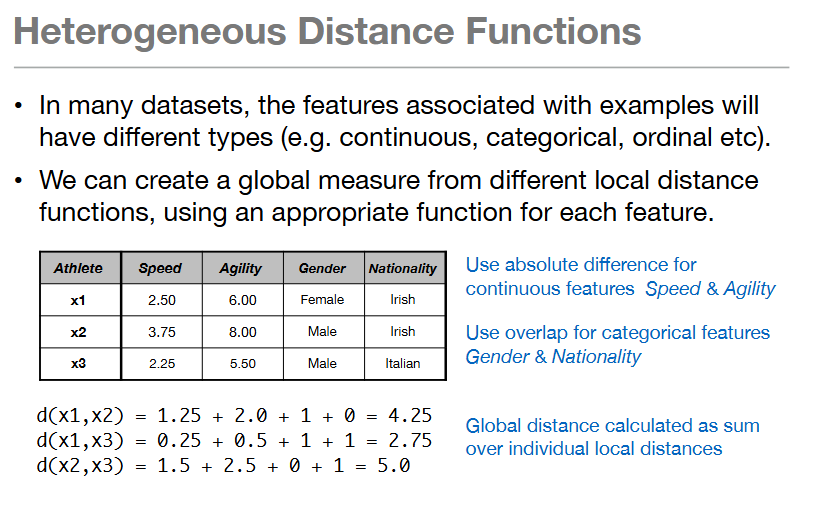
### ****Ordinal Data Characteristics****

Ordinal data:

1. Has a natural order (e.g., Poor < Fair < Good < Excellent).
2. Does not assume that the intervals between categories are equal (e.g., the "distance" between Fair and Good might not equal the "distance" between Poor and Fair).

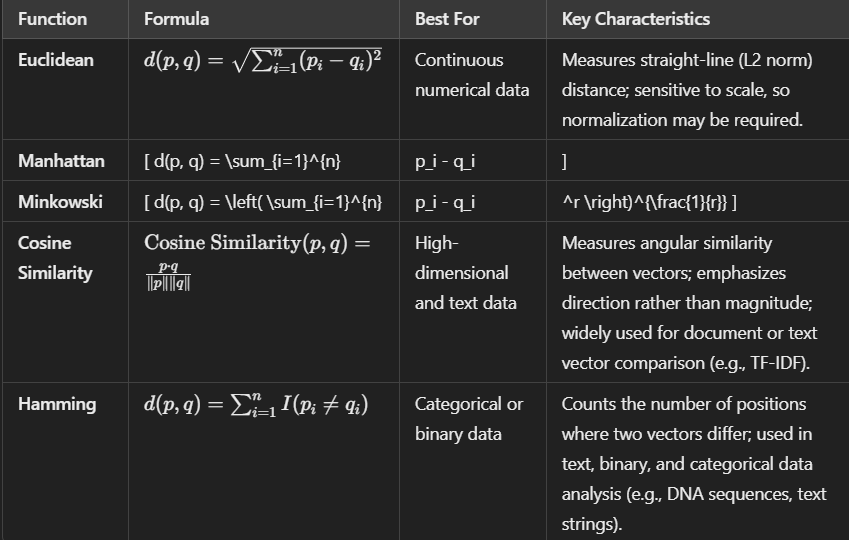


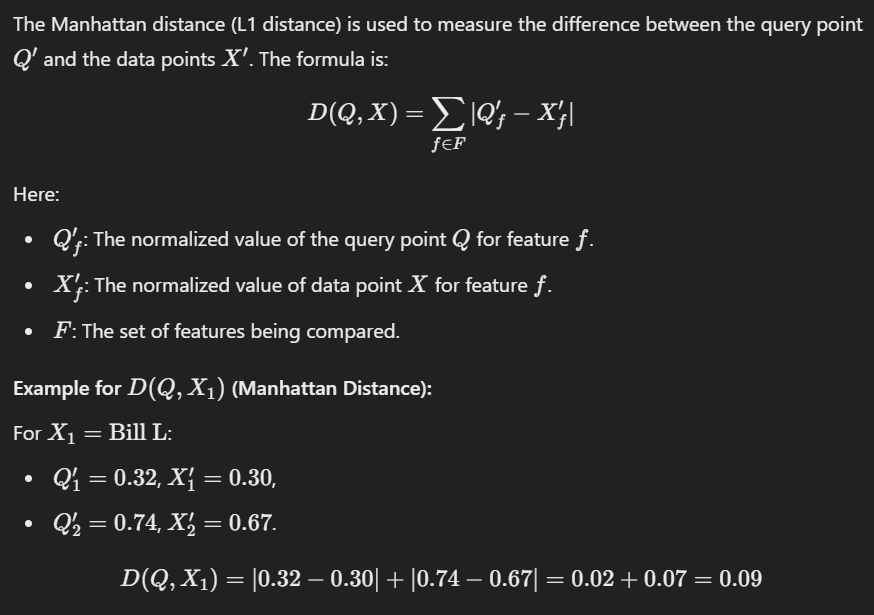
We have seen distance function for categoric data, for numeri data, wat if wer have both in our datasets?



### ****Choosing the Best Distance Function****:

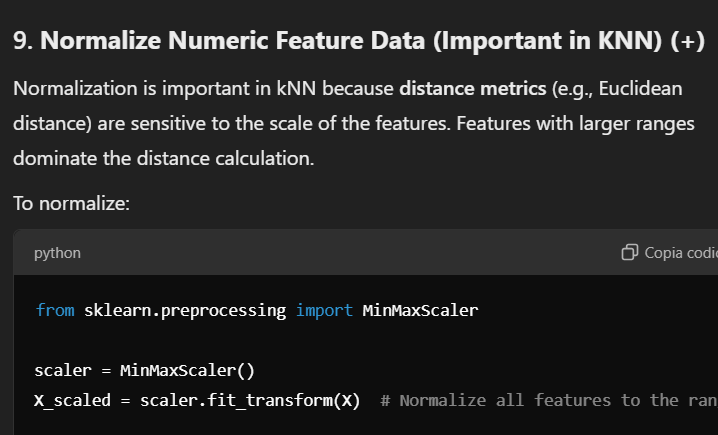
* **For numerical, continuous features**, **Euclidean distance** is most commonly used because it provides a natural "straight-line" distance between points.
* **For sparse or text-based data**, **Cosine similarity** is often preferred, as it focuses on the direction (similarity) rather than the exact distances between points.
* **For categorical data**, **Hamming distance** works well since it compares discrete values.
* **For grid-like, non-linear data**, **Manhattan distance** may perform better.

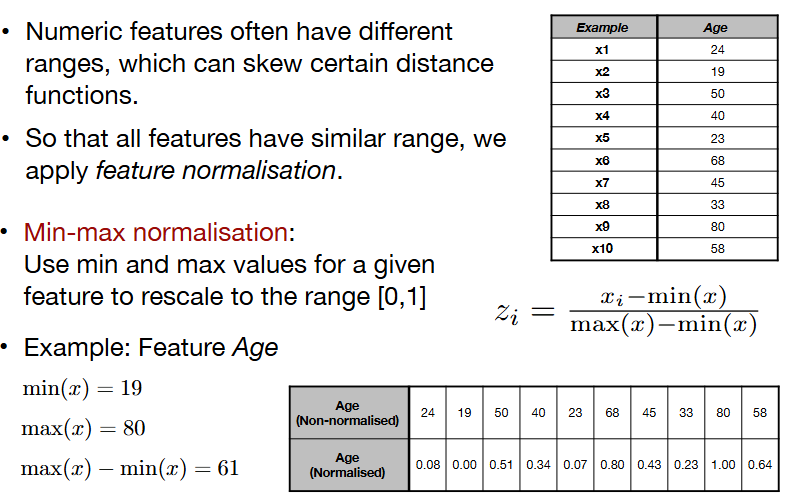




窗体顶端

Normalize numeric feature data ( important in KNN )+





Second gli es. Posso pure scegliere albitrarialmente min, max

### ****Curse of Dimensionality in k-NN****

The **curse of dimensionality** refers to the challenges and inefficiencies that arise when working with high-dimensional data

**Key Issues**:

* 1. **Distance Metrics**: In high dimensions, all points become similarly distant, making it hard to identify true neighbors.
  2. **Data Sparsity**: High-dimensional spaces spread out data,· the volume of the space grows exponentially, Data points become sparse, and local density assumptions fail, making neighborhood-based approaches like k-NN less effective.
  3. **Computational Cost**: Calculating distances and finding neighbors becomes slow.
  4. **Overfitting**: Irrelevant features dilute meaningful patterns, k-NN treats all dimensions equally, so irrelevant dimensions dilute meaningful relationships and increase noise sensitivity.

All leading to incorrect neighbor selection

**Mitigation Strategies**:

* 1. **Dimensionality Reduction**: Use PCA or t-SNE to reduce dimensions.
  2. **Feature Selection**: Retain only relevant features.
  3. **Weighted Distance Metrics**: Emphasize important features.
  4. **Normalization**: Scale features to avoid dominance by any single feature.

### ****Summary****:

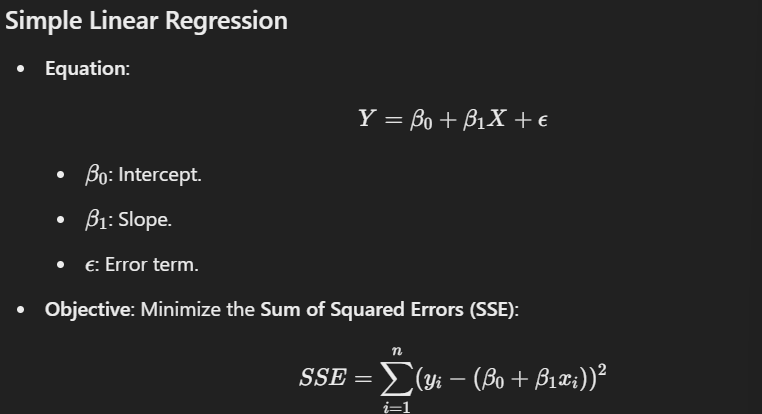
In high dimensions, k-NN struggles due to the curse of dimensionality, which reduces distance metric effectiveness, increases sparsity, and complicates computation. Preprocessing and feature engineering are essential to mitigate these challenges.

Classification: predict a value  
belonging to a class  
Regression: predict a value  
belonging to a continuous set

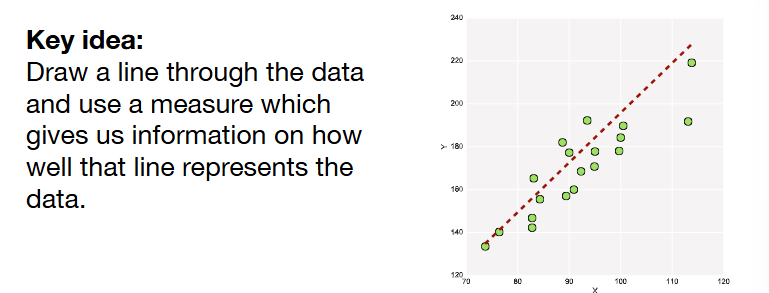
**Definition**: **Regression** is a statistical method used to predict a dependent variable (target) based on one or more independent variables (predictors).

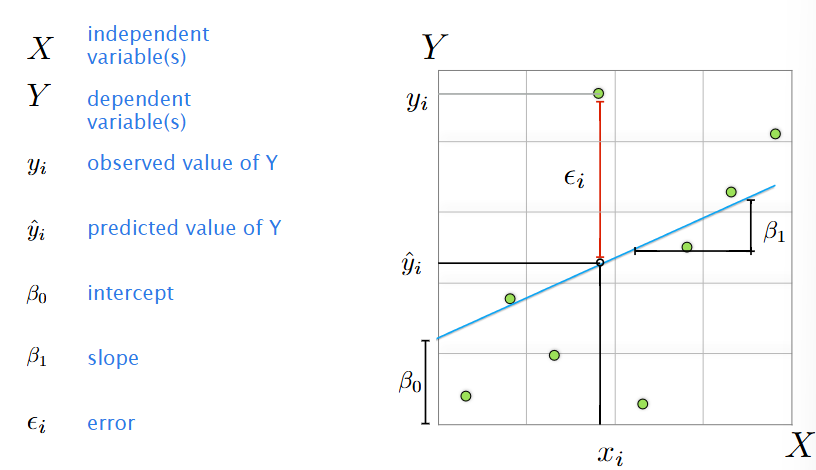
Terminology:

* Linear regression: We assume the output variable Y models a  
  linear relationship between the input variables X1, X2 , ...
* Simple Linear Regression: There is just one input variable X.  
  Changes is Y are caused by changes in X.
* We assume that the dependent variable Y is a linear  
  function of X
* Multiple Linear Regression: There is more than one input  
  variable



• β0: estimated average value of Y when X is 0 (i.e. intercept).  
• β1: estimated change in the average value of Y, if we make a  
unit change in X (i.e. slope).





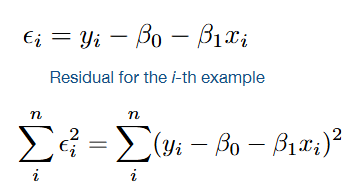
**Sum of Squared Errors (SSE)** is indeed a type of **loss function** in the context of regression analysis.

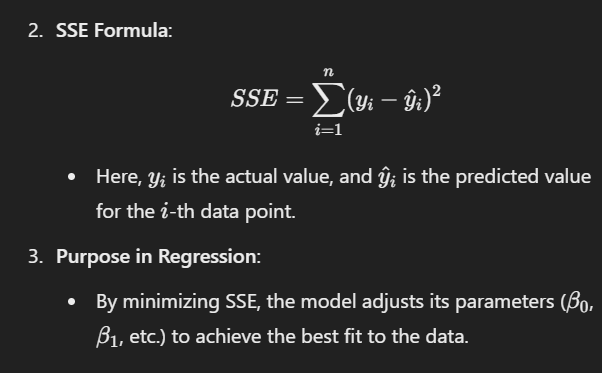
More specifically:

SSE is A **loss function** during the training phase, as the optimization process seeks to minimize it (througth methods like gradient descent)

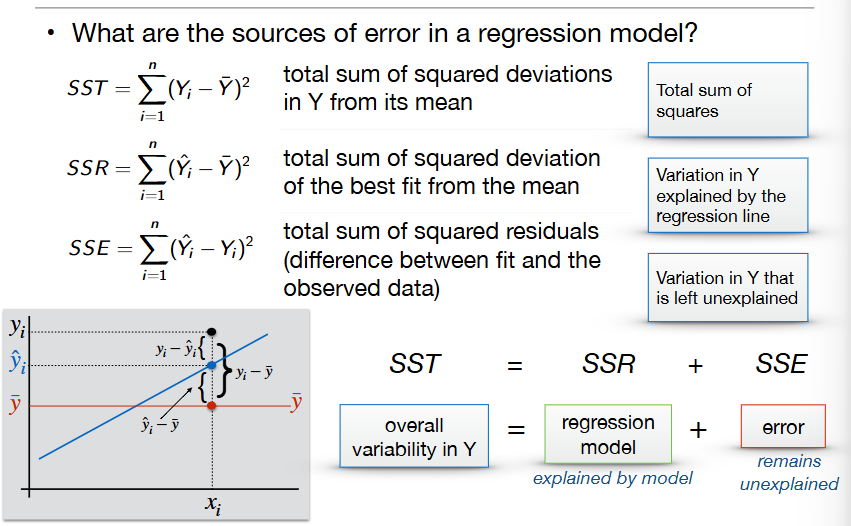
SSE is an **error measure** after training, as it evaluates the residual variance in predictions.

Another form of SSE





There are other form of Error measure



### Workflow of Simple Linear Regression (with Optimization)

**Understand the Problem**:

* 1. Define the target variable (yyy) and predictor (xxx).
  2. Verify an approximate linear relationship between xxx and yyy.

**Collect and Prepare Data**:

* 1. Gather data with xxx and yyy.
  2. Preprocess: Handle missing values, normalize features, and check for outliers.

**Explore the Data**:

* 1. Visualize using scatterplots.
  2. Compute correlation to check the strength of the linear trend.

**Split Data**:

* 1. Divide into training and testing sets (e.g., 80%-20%).

**Model Building and Optimization**:

* 1. **Model Equation**: y=β0+β1x+ϵy = \beta\_0 + \beta\_1x + \epsilony=β0​+β1​x+ϵ.
     1. β0\beta\_0β0​: Intercept, β1\beta\_1β1​: Slope.
  2. **Optimization**: Minimize the sum of squared errors (SSE) using least squares to estimate β0\beta\_0β0​ and β1\beta\_1β1​.

**Evaluate the Model**:

* 1. Metrics: R2R^2R2 (explained variance) and Mean Squared Error (MSE).
  2. Check residual plots for assumptions like homoscedasticity and linearity.

**Make Predictions**:

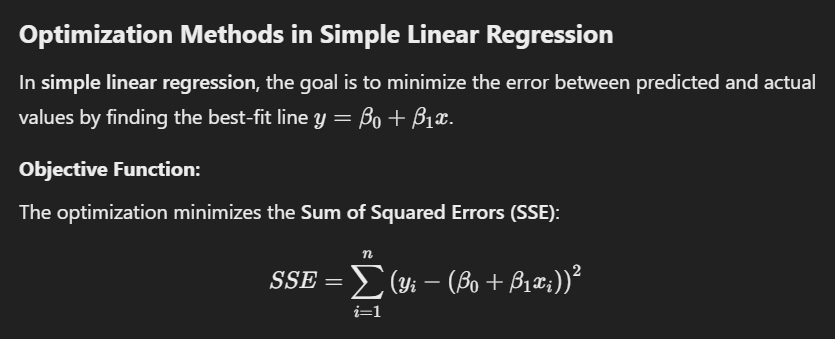
* 1. Apply y^=β0+β1x\hat{y} = \beta\_0 + \beta\_1xy^​=β0​+β1​x to test or new data.

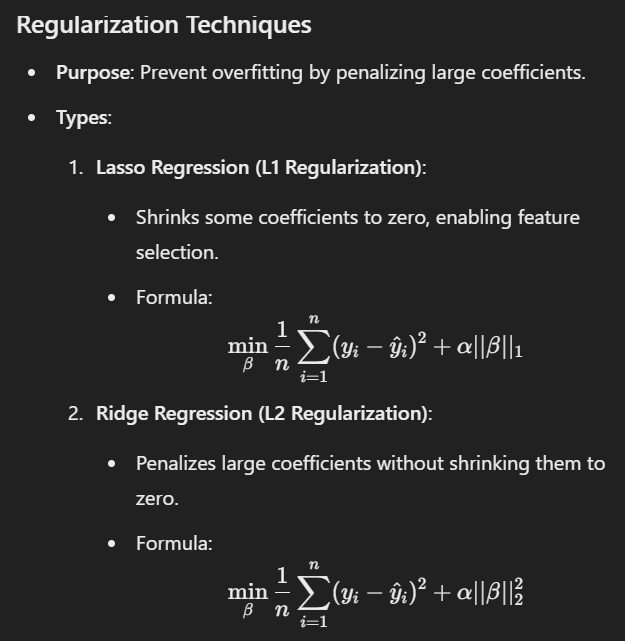
**Interpret Results**:

* 1. Assess the slope (β1\beta\_1β1​) for the relationship's significance.
  2. Use p-values or confidence intervals for statistical relevance.

**Deploy or Iterate**:

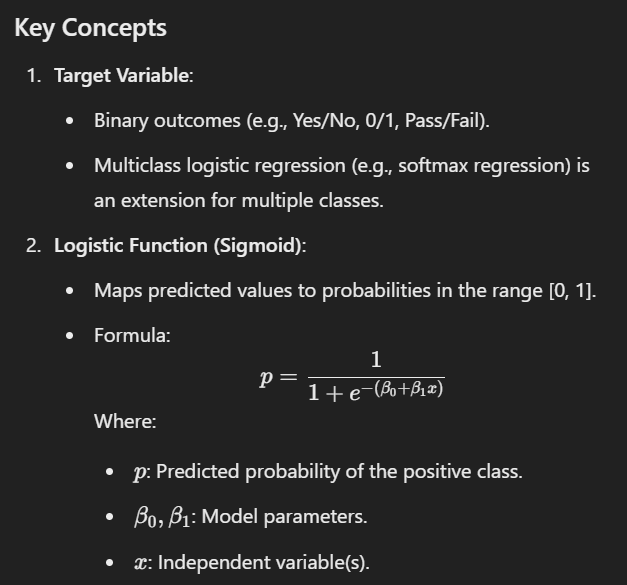
* 1. Deploy if results meet expectations.
  2. Refine the model if necessary (e.g., transformations or additional features).

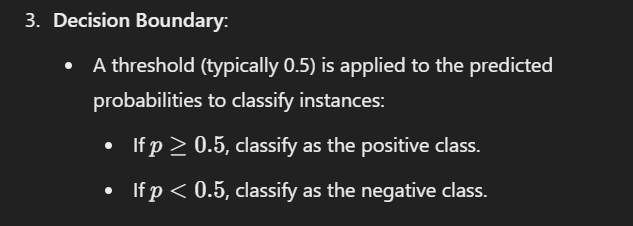


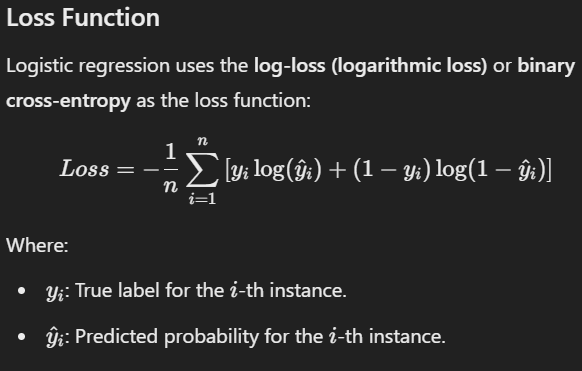


Its used both in linear and logistic regression

**Logistic Regression** is a statistical method used for binary classification problems. Unlike linear regression, which predicts continuous values, logistic regression predicts the probability of an instance belonging to a particular class.







### 

**Logistic regression involves a classification step,** which requires additional considerations like probability thresholds and classification metrics.

While the **workflows** for simple linear regression and logistic regression are conceptually similar, they diverge significantly in the model-building, optimization, and evaluation stages due to differences in their objectives and output types.

**Optimization**:

* 1. Logistic regression uses optimization techniques (e.g., **gradient descent**) to minimize the loss function

### Gradient Descent

* **Purpose**: Optimization algorithm to minimize a loss function.
* **Process**:
  1. Compute gradient of the loss function.
  2. Update parameters iteratively.
  3. Stop when convergence criteria are met.

**Variants**:

stochastic gradient descent optimisation difference from the standard gradient descent algorithm and difference to batch gradient descent ++

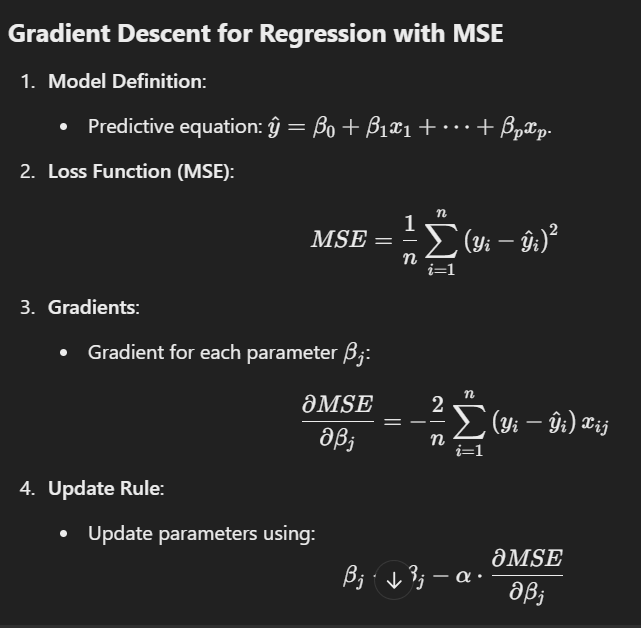
**Batch Gradient Descent (Standard Gradient Descent)**:

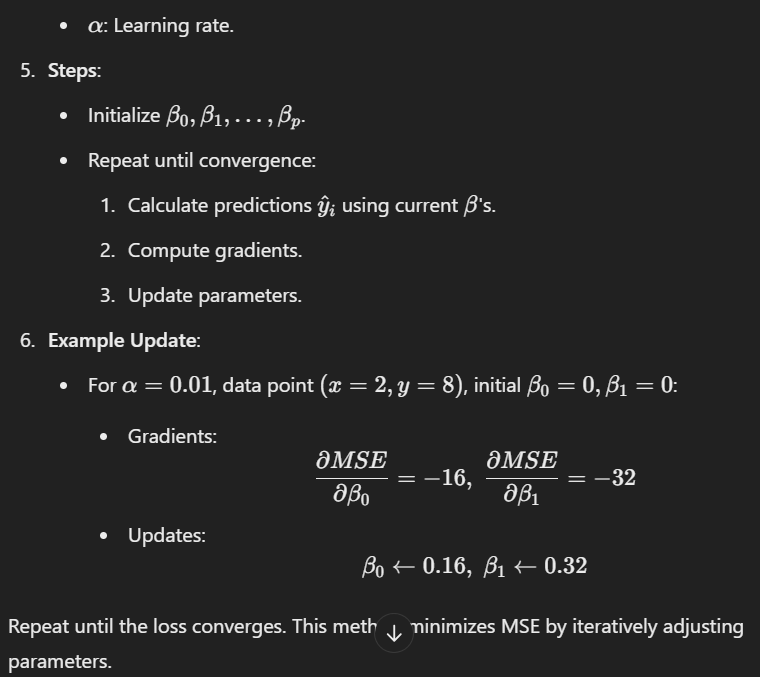
* + Computes gradients using the **entire training dataset**.
  + Iteratively updates parameters after processing all training examples.

**Stochastic Gradient Descent (SGD)**:

* + Computes gradients using **one random training example** at a time.
  + Parameters are updated after processing each single training sample.

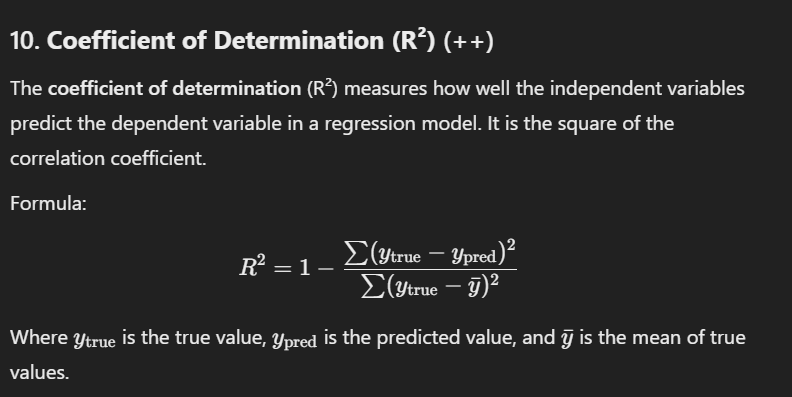
Gradient descend algorithm for regression model and update steps after every iteration ++





Coefficent of determination +

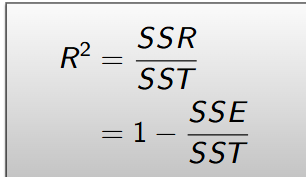
The **coefficient of determination**, denoted as R^2, is a key metric in regression analysis that quantifies the proportion of variance in the dependent variable(Y) that is predictable from the independent variables(X)





(observerd = true ; while predicted = pred)

### ****In another point of view****



### ****Interpretation of High**** R^2

**Proportion of Variance Explained**:

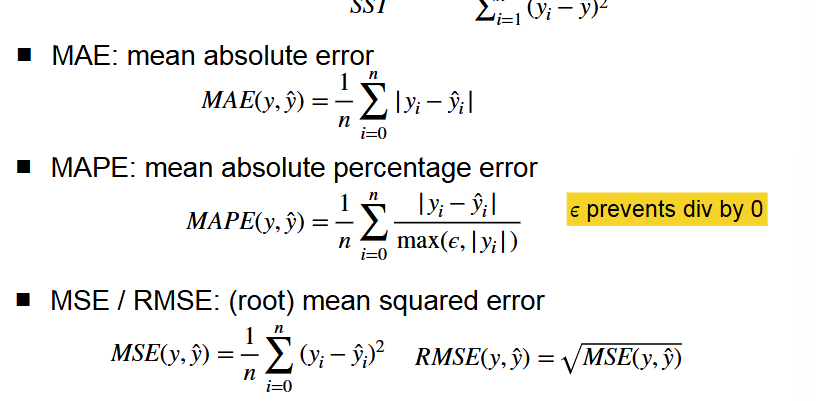
R2=0.9R^2 = 0.9R2=0.9 (or 90%) means that 90% of the variability in yyy is accounted for by the predictors in the model.

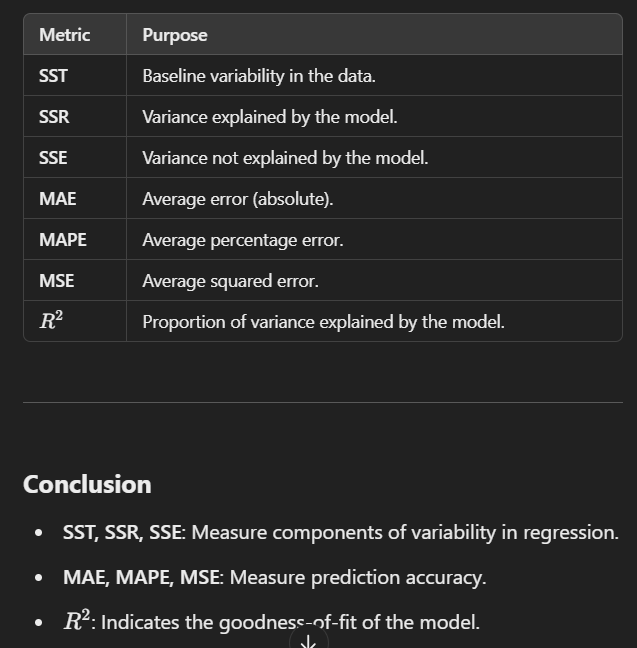
* 1. The remaining 10% is due to factors not included in the model (e.g., noise or omitted variables).

**Goodness of Fit**:

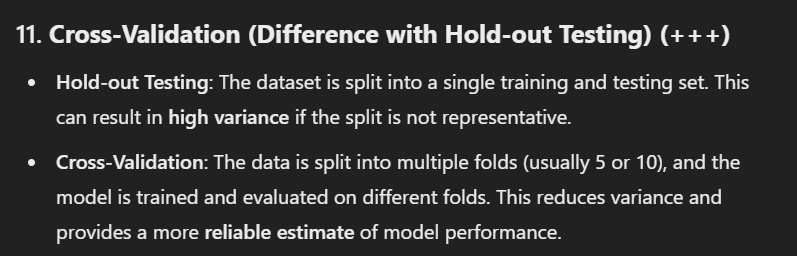
* 1. A high R2R^2R2 suggests that the regression model fits the data well.
  2. it tells us how well our regression line matches the real data. The closer R 2 is to 1, the better the fit.

We have also these others Error Measures

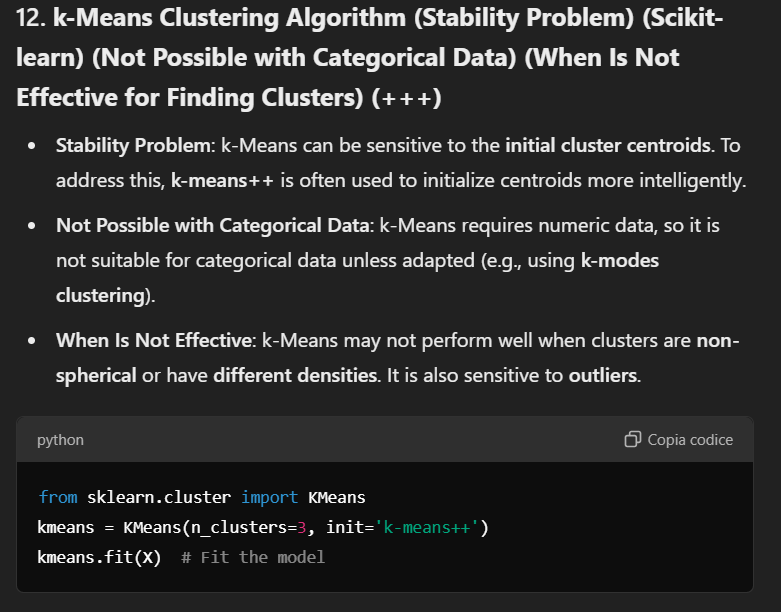




Cross validation ( difference with hold-out testing)+ +



k-Means clustering algorithm (stability problem) (scikitlearn) ( not possible with categorical data) ( when is not effective for finding clusters) ++++



why training a multi-layer feedforward neural network is

considerably more difficult than training a single layer network+

Even in a simple single layer Feedforward Neural Network the units

(neurons) will have a fixed bias input. What is the reason for this bias

nput? +

Explain in terms of the dynamics of a single neuron why it is that

single layer perceptrons are only able to learn patterns that are

linearly separable

concept of ‘graph cut’ in spectral clustering +

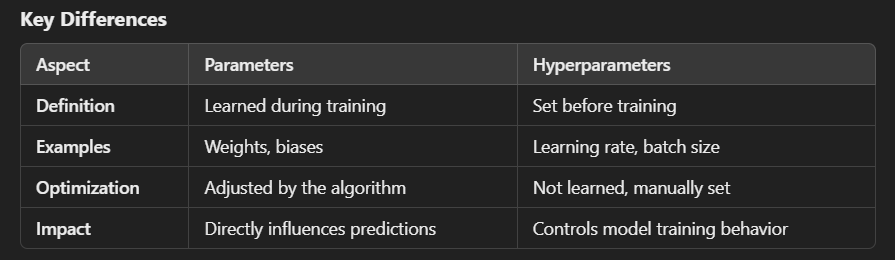
method to convert data in a feature vector format to an affinity matrix +

Spectral cluster (applied on tabular data)

difference between hyperparameters and other parameters in neural networks (example) ++

Parameters are values learned by the model during the training process.

While Hyperparameters are external configurations set before training and are not updated during training.



### ****Example: Neural Network****

* **Parameters**:
  + Weight matrix (W) connecting input layer to hidden layers.
  + Bias vector (b) added to each layer’s output.
* **Hyperparameters**:
  + Number of hidden layers.
  + Number of neurons per layer.
  + Activation function used in each layer (e.g., ReLU, sigmoid).

· **Hyperparameter Tuning**:

Performed to find the optimal configuration that maximizes model performance.

* Common techniques include **grid search**, **random search**, and **Bayesian optimization**.

· **Parameter Learning**:

Achieved through backpropagation and gradient descent.

Parameters are updated iteratively to minimize the loss function.

diversity role in ensemble classification(bagging, boosting) ++++++++

### ****Role of Diversity in Ensemble Classification****

**Importance of Diversity**:

* 1. Reduces errors by ensuring models complement each other.
  2. Prevents redundancy, improves generalization, and smooths out individual model errors.

**In Bagging**:

* 1. **Achieved by**: Random sampling with replacement (bootstrap samples) and using unstable models (e.g., decision trees).
  2. **Impact**: Reduces variance, prevents overfitting, and combines predictions through voting or averaging.

**In Boosting**:

* 1. **Achieved by**: Sequential training, focusing on misclassified examples by adjusting weights.
  2. **Impact**: Reduces both bias and variance, ensuring models specialize in correcting previous errors but risks overfitting noisy data.

**Comparison**:

* 1. **Bagging**: Uses randomness for diversity, models trained independently.
  2. **Boosting**: Creates diversity by focusing on harder examples, models trained sequentially.

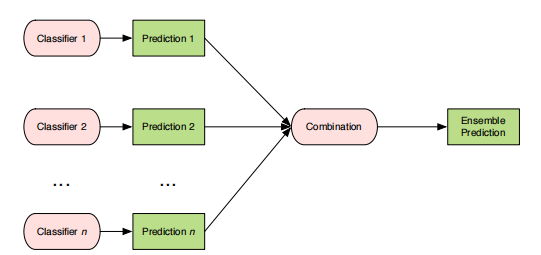
Ensemble methods : baggin /boosting

what differentiates the different ensemble members in a

Boosting Ensemble:

Boosting ensembles differ because their members adapt dynamically, focusing on correcting errors of previous models, and rely on weighted examples and models for more targeted learning. This sequential approach sets them apart from other ensemble methods like Bagging.

Ensembles classification Combines predictions from multiple classifiers to improve accuracy.



**Condorcet Jury Theorem**: If individual voters (classifiers) have a probability of being correct > 50%, the ensemble becomes more accurate as the group grows, assuming diversity among classifiers.

**Diversity Role**: Diverse models reduce errors by ensuring that mistakes of one model are corrected by others. The accuracy plateaus if diversity decreases.

How do we generate the different base classiefier that complements each other? And how do we combine their outputs?

**Random sampling** is a technique used in ensemble methods to create diversity among base classifiers and reduces overfitting:

**• Simple random sampling:** A subset of items is chosen (with or without replacement) from a larger population. Each item is chosen by chance, and each member of the population has an equal probability of being chosen.

#### 3. ****Ensemble Generation Techniques****

**Bagging (Bootstrap Aggregating)**:

* + Builds models by sampling training data **with replacement**.
  + Encourages diversity by training each model on different random data subsets.
  + Reduces **variance**, making it effective for unstable models (e.g., decision trees).

**Boosting**:

* + Sequentially trains models, focusing on **misclassified examples, so that later classifiers are trained to better predict class labels for examples that earlier ones performed poorly on**

**Misclassified examples** refer to data points in a dataset that a model predicts incorrectly. In classification tasks, these are instances where the predicted class label does not match the true class label.

Train and test a classifier to identify the misclassified example. For the next classifier, focus on correctly classifying those "difficult" examples.

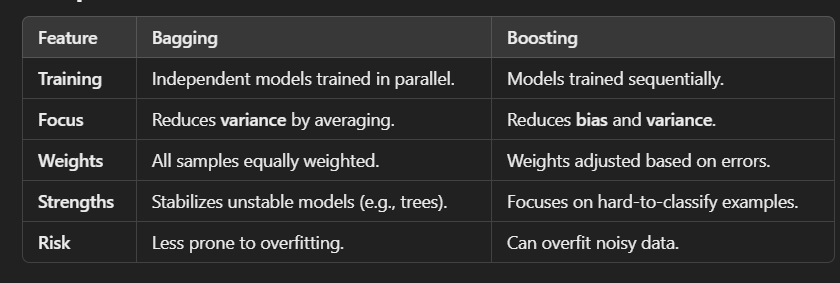
we emphasise certain training examples by assigning weights to training instances.

· Initially, all examples have equal weights, but weights are adjusted dynamically after each iteration.

· Misclassified examples get higher weights, making them more likely to influence the next model.

Each model's contribution to the final prediction is weighted based on its accuracy (lower error → higher weight in final ensemble).

* + Reduces both **bias** and **variance** but can overfit noisy data.

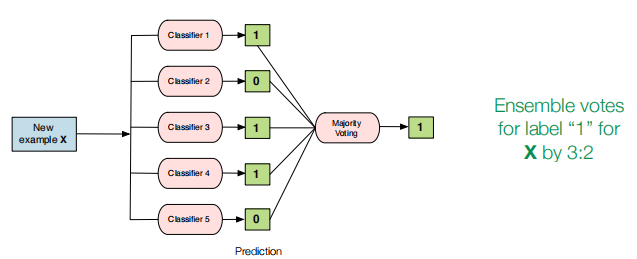


#### 4. ****Ensemble Combination Methods****

Simplest way to combine the output of multiple classifiers is to use

majority voting

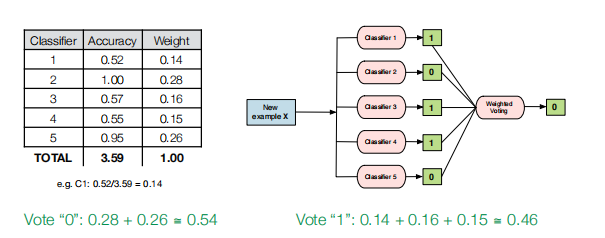
* **Voting**:
  + **Majority Voting**: Each model votes equally, and the majority class wins.
  + **Weighted Voting**: Models with higher accuracy are given more weight.



If individual classifiers do not give equal performance, we should give more influence to better classifiers.

• Weighted Voting Combination:

• Rather than treating every classifier’s vote equally, we weight each classifier’s vote based on its accuracy/error.



#### 5. ****Random Forests****

* Combines **Bagging** and **Random Subspacing**:
  + Randomly selects subsets of features at each decision tree split, improving diversity and reducing overfitting.
  + Builds hundreds of decision trees, each using a bootstrap sample and random feature subset.

#### 6. ****Boosting Variants****

* Boosting builds models iteratively, focusing on harder-to-classify examples.
* **AdaBoost**: Adjusts weights of data points based on errors of the previous classifier.
* **Gradient Boosting**: Uses gradient descent to minimize errors for the next model.

#### 7. ****Bias-Variance Tradeoff in Ensembles****

* **Bagging**: Reduces variance by averaging predictions across diverse models.
* **Boosting**: Reduces both bias and variance but risks overfitting noisy data.

#### 8. ****XGBoost****:

* A highly optimized Gradient Boosting framework.
* Popular in machine learning competitions due to its efficiency and predictive power.

### Diversity's Role in Ensemble Classification (Bagging, Boosting)

1. **Definition**: Diversity refers to how differently the base classifiers behave, particularly in terms of their errors.
2. **Importance**:
   * Prevents correlated errors: If all classifiers make the same mistakes, the ensemble cannot improve.
   * Ensures complementary strengths: Some classifiers perform well on specific subsets of data, while others excel elsewhere.
3. **Bagging**:
   * Achieves diversity by training models on different random subsets of data.
   * Helps reduce **variance** by ensuring models are exposed to different perspectives.
4. **Boosting**:
   * Achieves diversity by focusing on hard-to-classify examples.
   * Models are trained sequentially, correcting errors from previous models.

### What Differentiates Ensemble Members in Boosting?

* **Focus on Misclassified Examples**:
  + Each subsequent model is trained to correct errors made by the previous model.
* **Dynamic Weight Adjustments**:
  + Training instances are weighted; misclassified examples are given higher weights.
* **Error-Driven Training**:
  + Boosting optimizes the training process to reduce bias while targeting difficult areas of the data space.

By employing these mechanisms, Boosting creates specialized and complementary classifiers that collectively form a robust ensemble.

T-test, p-value +

generalisation in the context of supervised learning, with refer

ence to the problem of overfitting.( causes) (scikit) (strsategies to prevent it) ++

overfitting and underfitting (in supervised learning)

bias terms and variance in a neural network architecture++