Lý thuyết chung về Spark

1. Kiến trúc cơ bản của Spark

Spark Components

A few blue squares with words

Description automatically generated

|  |  |
| --- | --- |
|  | Uses |
| Spark Core | * The Spark Core is the heart of Spark and performs the core functionality. * It holds the components for task scheduling, fault recovery, interacting with storage systems and memory management. |
| Spark SQL structured data | * The Spark SQL is built on the top of Spark Core. It provides support for structured data. * It allows to query the data via SQL (Structured Query Language) as well as the Apache Hive variant of SQL?called the HQL (Hive Query Language). * It supports JDBC and ODBC connections that establish a relation between Java objects and existing databases, data warehouses and business intelligence tools. * It also supports various sources of data like Hive tables, Parquet, and JSON. |
| Spark Streaming real-time | * Spark Streaming is a Spark component that supports scalable and fault-tolerant processing of streaming data. * It uses Spark Core's fast scheduling capability to perform streaming analytics. * It accepts data in mini-batches and performs RDD transformations on that data. * Its design ensures that the applications written for streaming data can be reused to analyze batches of historical data with little modification. * The log files generated by web servers can be considered as a real-time example of a data stream. |
| MLib machine learning | * The MLlib is a Machine Learning library that contains various machine learning algorithms. * These include correlations and hypothesis testing, classification and regression, clustering, and principal component analysis. * It is nine times faster than the disk-based implementation used by Apache Mahout. |
| GraphX graph processing | * The GraphX is a library that is used to manipulate graphs and perform graph-parallel computations. * It facilitates to create a directed graph with arbitrary properties attached to each vertex and edge. * To manipulate graph, it supports various fundamental operators like subgraph, join Vertices, and aggregate Messages. |

1. Các khái niệm cơ bản trong Spark:
   1. DataFrames:

DataFrame is a distributed collection of data organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames are one of the core data structures in Spark, and they are used for a variety of tasks, including data analysis, machine learning, and real-time streaming.

Key features of DataFrames:

* Distributed: DataFrames are distributed across multiple machines, which allows them to be processed in parallel. This makes them ideal for processing large datasets.
* Structured: DataFrames are structured, which means that each column has a name and a type. This makes them easy to understand and work with.
* Optimized: DataFrames are optimized for performance. Spark uses a variety of techniques to optimize DataFrame operations, including lazy evaluation and code generation.

Benefits of using DataFrames:

* Ease of use: DataFrames are easy to use, with a simple and intuitive API.
* Scalability: DataFrames can be used to process large datasets efficiently.
* Performance: DataFrames are optimized for performance, and they can be used to process data quickly.
* Versatility: DataFrames can be used for a variety of tasks, including data analysis, machine learning, and real-time streaming.
  1. Partitions

A partition is a logical division of data that is stored on a single machine. Partitions are the basic unit of parallelism in Spark, and they are used to distribute data processing across multiple machines.

Benefits of Partitioning (Partitioning offers several benefits for Spark processing):

* Parallelism: By dividing data into partitions, Spark can process multiple partitions in parallel, significantly improving performance for large datasets.
* Shuffle Optimization: Partitioned data reduces the amount of data that needs to be shuffled between machines during operations like joins and aggregations.
* Efficient File I/O: Partitioning aligns with the file system's block structure, enabling efficient reading and writing of partitioned data.
* Improved Query Performance: Partitioning allows Spark to filter and group data based on the partition key, reducing the amount of data that needs to be processed.

Partitioning Strategies (Spark offers two main strategies for partitioning data):

* Range Partitioning: Partitions are created based on a range of values in a specific column. This is suitable for columns with a continuous distribution of values, such as timestamps or numerical values.
* Hash Partitioning: Partitions are created by hashing values in a specific column. This is useful for columns with a categorical distribution of values, such as user IDs or product categories.
  1. Transformation

A transformation is an operation that takes an existing dataset and produces a new dataset. Transformations are lazy, meaning that they do not actually compute their results until an action is called. This allows Spark to optimize the execution of transformations by only computing the results that are actually needed.

Types of Transformations:

There are two main types of transformations in Spark: narrow transformations and wide transformations.

Narrow transformations only shuffle data within partitions. This means that the partition structure of the input dataset is preserved in the output dataset. Examples of narrow transformations include filter(), map(), and takeSample(). A diagram of a flowchart

Description automatically generated

Wide transformations shuffle data between partitions. This means that the partition structure of the output dataset is different from the partition structure of the input dataset. Examples of wide transformations include distinct(), groupByKey(), and sortByKey(). A drawing of a network of paper

Description automatically generated with medium confidence

Lazy Evaluation of Transformations

One of the key features of Spark transformations is that they are lazy.

This means that they do not actually compute their results until an action is called. This has several benefits:

* Spark can optimize the execution of transformations by only computing the results that are actually needed.
* Spark can cache the results of transformations, which can improve the performance of subsequent queries.
* Spark can handle failures gracefully by recomputing the results of transformations that have failed.
  1. Action

An action is an operation that triggers the computation of a dataset and returns a result to the driver program. Unlike transformations, which only define a computation path but do not actually execute, actions force Spark to compute and produce a result.

Actions are essential for working with the results of Spark computations, as they allow you to access, manipulate, and store the data. They play a crucial role in data analysis, machine learning, and other Spark applications.

Key Characteristics of Actions:

* Trigger Computation: Actions initiate the execution of a Spark job, causing Spark to materialize the results of transformations and return them to the driver program.
* Return a Result: Actions always return a value, either a simple scalar value or a more complex data structure, such as a collection or a DataFrame.
* Materialize Transformations: Actions materialize the results of transformations, meaning that the computations defined by transformations are actually executed when an action is called.
* Force Partitioning: Actions can force partitioning of data, which is important for certain operations like saveAsTextFile(), where each partition is written to a separate file.

Common Actions in Spark:

* collect(): Collects all elements of an RDD or DataFrame into a local collection, such as an array or a list.
* count(): Returns the total number of elements in an RDD or DataFrame.
* reduce(): Aggregates the elements of an RDD using a specified binary function.
* takeSample(): Collects a random sample of elements from an RDD or DataFrame.
* saveAsTextFile(): Writes the elements of an RDD or DataFrame to a text file, with each element written as a separate line.
* saveAsTable(): Saves an RDD or DataFrame to an external table, such as a Hive or Parquet table.
* foreach(): Applies a function to each element of an RDD or DataFrame.
* show(): Displays a portion of an RDD or DataFrame in the console.

Impact of Actions on Spark Execution:

* Lazy Evaluation: Actions trigger the execution of transformations, but transformations themselves are lazy, meaning they are not evaluated until an action is called.
* Materialization Overhead: Actions incur some overhead due to materializing the results of transformations and returning them to the driver program.
* Job Execution: Actions trigger the creation and execution of Spark jobs, which involve distributing computations across multiple worker machines.
* Serialization and Deserialization: Actions may involve serialization and deserialization of data, which can impact performance for large datasets.

Optimizing Actions in Spark:

* Minimize Actions: Only use actions when necessary to access or store the results of computations.
* Cache Intermediate Results: Cache the results of transformations to avoid recomputing them for multiple actions.
* Utilize Partitioning: Consider partitioning data strategically to improve the efficiency of certain actions.
* Tune Spark Configuration: Adjust Spark configuration parameters to optimize job execution and action performance.

An action is an operation that triggers the computation of a dataset. Actions are not lazy, which means that they will always compute their results, even if the results are not needed. Examples of actions include collect(), count(), and saveAsTextFile().

1. Data Source
   1. Parquet

Parquet is a columnar file format that offers high storage efficiency, fast query performance, and support for schema evolution. It is a popular choice for large-scale data analytics, machine learning, and data warehousing applications.

* 1. ORC

ORC is another columnar file format that offers similar advantages to Parquet, including efficient storage, fast queries, and schema evolution support. It is a good alternative to Parquet and is widely supported by various tools and frameworks.

* 1. JSON

JSON is a text-based data format that is commonly used for storing semi-structured data. It is easy to use and human-readable, but it is less efficient for storage and queries compared to columnar formats like Parquet and ORC.

* 1. CSV

CSV is a simple text-based format for storing tabular data. It is universally supported by various tools and frameworks, making it a convenient choice for data exchange and quick data analysis. However, it is the least efficient format in terms of storage and query performance.

Pros and cons

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Feature | Parquet | ORC | JSON | CSV |
| Storage Efficiency | Highly efficient due to columnar storage and compression | Efficient columnar storage and compression | Less efficient due to row-based storage | Least efficient due to row-based storage and no compression |
| Query Performance | Faster queries due to columnar storage and predicate pushdown | Good query performance due to columnar storage and predicate pushdown | Slower queries due to row-based storage | Slowest queries due to row-based storage and no compression |
| Schema Evolution | Supports schema evolution with minimal data loss | Supports schema evolution with minimal data loss | Less flexible for schema changes | Does not support schema evolution |
| Compatibility | Widely supported by various tools and frameworks | Widely supported by various tools and frameworks | Widely supported by various tools and frameworks | Universally supported by various tools and frameworks |
| Ease of Use | Slightly more complex to set up and use | Slightly more complex to set up and use | Simple and easy to use | Extremely simple and easy to use |
| Use Cases | Ideal for large-scale data analytics, machine learning, and data warehousing | Ideal for large-scale data analytics, machine learning, and data warehousing | Suitable for semi-structured data, configuration files, and web applications | Suitable for simple data exchange, logging, and quick data analysis |

In summary, the choice of data source format depends on the specific requirements of the application. For large-scale data analytics and machine learning, Parquet or ORC are often preferred due to their superior performance and schema evolution capabilities. For semi-structured data, JSON is a suitable choice. For simple data exchange or quick analysis, CSV can be a convenient option.

1. Những vấn đề nghiêm trọng thường gặp khi lập trình trong 1 cụm Hadoop với Spark trên dữ liệu lớn (Slow Aggregation, Slow join, Slow writing, etc...). Nguyên nhân và cách giải quyết.
   1. Slow Aggregation

If you have a slow aggregation, start by reviewing the issues in the “Slow Tasks” sec‐ tion before proceeding. Having tried those, you might continue to see the same prob‐ lem.

Signs and symptoms

• Slow tasks during a groupBy call.

• Jobs after the aggregation are slow, as well.

Potential treatments

Unfortunately, this issue can’t always be solved. Sometimes, the data in your job just has some skewed keys, and the operation you want to run on them needs to be slow.

• Increasing the number of partitions, prior to an aggregation, might help by reducing the number of different keys processed in each task.

• Increasing executor memory can help alleviate this issue, as well. If a single key has lots of data, this will allow its executor to spill to disk less often and finish faster, although it may still be much slower than executors processing other keys.

• If you find that tasks after the aggregation are also slow, this means that your dataset might have remained unbalanced after the aggregation. Try inserting a repartition call to partition it randomly.

• Ensuring that all filters and SELECT statements that can be are above the aggrega‐ tion can help to ensure that you’re working only on the data that you need to be working on and nothing else. Spark’s query optimizer will automatically do this for the structured APIs.

• Ensure null values are represented correctly (using Spark’s concept of null) and not as some default value like " " or "EMPTY". Spark often optimizes for skipping nulls early in the job when possible, but it can’t do so for your own placeholder values.

• Some aggregation functions are also just inherently slower than others. For instance, collect\_list and collect\_set are very slow aggregation functions because they must return all the matching objects to the driver, and should be avoided in performance-critical code.

* 1. Slow join

Joins and aggregations are both shuffles, so they share some of the same general symptoms as well as treatments.

Signs and symptoms

• A join stage seems to be taking a long time. This can be one task or many tasks.

• Stages before and after the join seem to be operating normally.

Potential treatments

• Many joins can be optimized (manually or automatically) to other types of joins. We covered how to select different join types in Chapter 8.

• Experimenting with different join orderings can really help speed up jobs, especially if some of those joins filter out a large amount of data; do those first.

• Partitioning a dataset prior to joining can be very helpful for reducing data movement across the cluster, especially if the same dataset will be used in multiple join operations. It’s worth experimenting with different prejoin partitioning. Keep in mind, again, that this isn’t “free” and does come at the cost of a shuffle.

• Slow joins can also be caused by data skew. There’s not always a lot you can do here, but sizing up the Spark application and/or increasing the size of executors can help, as described in earlier sections.

• Ensuring that all filters and select statements that can be are above the join can help to ensure that you’re working only on the data that you need for the join.

• Ensure that null values are handled correctly (that you’re using null) and not some default value like " " or "EMPTY", as with aggregations.

• Sometimes Spark can’t properly plan for a broadcast join if it doesn’t know any statistics about the input DataFrame or table. If you know that one of the tables that you are joining is small, you can try to force a broadcast (as discussed in Chapter 8), or use Spark’s statistics collection commands to let it analyze the table.

* 1. Slow writing

Slow I/O can be difficult to diagnose, especially with networked file systems.

Signs and symptoms

• Slow reading of data from a distributed file system or external system.

• Slow writes from network file systems or blob storage.

Potential treatments

• Turning on speculation (set spark.speculation to true) can help with slow reads and writes. This will launch additional tasks with the same operation in an attempt to see whether it’s just some transient issue in the first task. Speculation is a powerful tool and works well with consistent file systems. However, it can cause duplicate data writes with some eventually consistent cloud services, such as Amazon S3, so check whether it is supported by the storage system connector you are using.

• Ensuring sufficient network connectivity can be important—your Spark cluster may simply not have enough total network bandwidth to get to your storage system.

• For distributed file systems such as HDFS running on the same nodes as Spark, make sure Spark sees the same hostnames for nodes as the file system. This will enable Spark to do locality-aware scheduling, which you will be able to see in the “locality” column in the Spark UI.

Ngôn ngữ Spark

Select:

df.select(expr1, expr2, ...)

example:

1. Selecting specific columns:

df.select("id", "name", "age")

1. Selecting columns using aliases:

df.select("id AS customer\_id", "name", "age")

1. Selecting expressions:

df.select("id", "name", expr("age + 5").alias("age\_in\_5\_years"))

1. Selecting all columns:

df.select("\*")

Filter:

df.filter(condition)

Join:

|  |  |
| --- | --- |
| Inner joins | keep rows with keys that exist in the left and right datasets |
| Outer joins | keep rows with keys in either the left or right datasets |
| Left outer joins | keep rows with keys in the left dataset |
| Right outer joins | keep rows with keys in the right dataset |
| Left semi joins | keep the rows in the left, and only the left, dataset where the key appears in the right dataset |
| Left anti joins | keep the rows in the left, and only the left, dataset where they do not appear in the right dataset |
| Natural joins | perform a join by implicitly matching the columns between the two datasets with the same names) |
| Cross (or Cartesian) joins | match every row in the left dataset with every row in the right dataset |

Pivot:

Spark SQL provides pivot() function to rotate the data from one column into multiple columns (transpose row to column). It is an aggregation where one of the grouping columns values transposed into individual columns with distinct data.

Window function

Window functions in PySpark are a powerful tool for performing aggregations and calculations over a set of rows that are related to the current row. They are often used to compute running totals, ranks, percentiles, and other aggregates that are dependent on a specific window of data.

Basic Concepts

* Window: A window defines a set of rows over which a window function will be applied. Windows can be defined based on partition columns, ordering columns, and frame boundaries.
* Partition columns: Partition columns divide the data into groups, and the window function is applied to each partition separately.
* Ordering columns: Ordering columns define the order of rows within each partition. This is important for functions like rank() and percentile\_rank() that depend on row order.
* Frame boundaries: Frame boundaries define the range of rows over which the window function will be applied. The frame can be unbounded, meaning it includes all rows in the partition, or it can be bounded by a specific number of rows or a range of values.

Common Window Functions

* row\_number(): Assigns a row number to each row within a partition, starting from 1.
* rank(): Assigns a rank to each row within a partition, based on its value relative to other rows in the partition.
* dense\_rank(): Assigns a rank to each row within a partition, without gaps. This means that if there are ties for a rank, all tied rows will be assigned the same rank.
* percent\_rank(): Assigns a percentile rank to each row within a partition. This means that a row with a percentile rank of 0.5 is in the 50th percentile of the partition.
* ntile(): Assigns a bucket number to each row within a partition, dividing the rows into a specified number of buckets.

• Cho trước một bảng population với 2 cột là population\_type và salary. Với mỗi population type, tìm ra 3 người có lương cao nhất.

window = Window.partitionBy(df['population\_type']).orderBy(df['salary'].desc())

df.select('\*', rank().over(window).alias('rank')).filter(col('rank') <= 3).show()

Algorithm – Mllib

1. Correlation: định nghĩa, cách tính correlation của 2 biến ngẫu nhiên, 2 series

"Correlation" is a statistical term that describes the degree to which two random variables are linearly related to each other. It is used to assess whether a change in one variable predicts a change in another in a predictable way.

Definition:

Correlation Coefficient: This is a numerical value that measures the strength and direction of the linear relationship between two variables. This value ranges from -1 to +1.

* +1: Perfect positive linear relationship. As one variable increases, the other also increases.
* 0: No linear relationship.
* -1: Perfect negative linear relationship. As one variable increases, the other decreases.Cách Tính Correlation: Correlation của Hai Biến Ngẫu Nhiên: Sử dụng công thức Pearson correlation coefficient

A math equations and formulas

Description automatically generated with medium confidence

* Correlation measures linear relationships only and does not necessarily indicate a cause-and-effect relationship.
* A high correlation coefficient does not always mean that one variable causes a change in another.

1. Principal Component Analysis (PCA): Ý nghĩa, cách dùng. Nắm vững cách tính eigenvector, eigenvalue của 1 ma trận.

Principal Component Analysis (PCA) is a statistical method used for dimensionality reduction by transforming data into a new set of "principal components" while retaining most of the information in the original data. It's a useful tool for analyzing large datasets, where reducing the number of variables can make the data more manageable and interpretable.

Significance of PCA:

* Dimensionality Reduction: PCA helps reduce the number of variables without losing much information.
* Data Analysis: It aids in uncovering relationships among variables and highlights variations in the data.
* Computational Efficiency: Reducing the number of variables can enhance computational efficiency in machine learning models and data analysis.

How to Use PCA:

* Standardize the Data: First, the data needs to be standardized so that each variable has a mean of 0 and a standard deviation of 1.
* Compute the Covariance Matrix: Calculate the covariance matrix from the standardized data.
* Calculate Eigenvalues and Eigenvectors: Find the eigenvalues and eigenvectors of the covariance matrix.
* Select Principal Components: The eigenvectors corresponding to the largest eigenvalues are chosen as the principal components.
* Transform the Data: Use these principal components to transform the original data into a new set with reduced dimensions.

Calculating Eigenvectors and Eigenvalues of a Matrix:

Definitions:

* Eigenvector: A vector that does not change direction when applied by a matrix.
* Eigenvalue: A value that, when multiplied with the eigenvector, results in a vector in the same direction but potentially different in length.

How to Calculate:

* Solve the equation:, where A is the matrix, ,is the eigenvector, and is the eigenvalue.
* Find by solving the determinant of (where I is the identity matrix).
* Use the found λ values to solve the equation and find the corresponding eigenvectors.

PCA is a crucial tool in multivariate data analysis and is widely used in various fields like finance, biology, and machine learning.

1. Các phương pháp cơ bản trong Data Preprocessing với Spark.

Data preprocessing in PySpark, the Python API for Apache Spark, involves a series of steps to clean and transform raw data into a suitable format for analysis and modeling. PySpark provides a rich set of tools within its library, making it efficient for handling large-scale data in a distributed manner. Here's an overview of common data preprocessing tasks in PySpark:

1. Data Import and Understanding: Loading Data: PySpark allows reading data from various sources like CSV, JSON, Parquet, and databases. Exploring Data: Functions like show(), printSchema(), and describe() help in understanding the data's structure, content, and statistical properties.
2. Data Cleaning: Handling Missing Values: Utilize functions like dropna() to remove rows with missing data, or fillna() to replace them with a specific value or a derived statistic (mean, median). Removing Duplicates: Use dropDuplicates() to eliminate duplicate rows.
3. Data Transformation: Normalization and Standardization: The StandardScaler and Normalizer in PySpark MLlib are used for scaling numerical features. Encoding Categorical Variables: Convert categorical variables into numeric format using StringIndexer, OneHotEncoder, etc. Feature Extraction: For textual data, PySpark offers functions like Tokenizer, HashingTF, and IDF for feature extraction.
4. Data Integration and Aggregation: Merging Data: Combine dataframes using operations like join(). Aggregating Data: Use groupBy() for aggregating data based on certain features.
5. Data Reduction: Dimensionality Reduction: Apply PCA (Principal Component Analysis) or other techniques to reduce the number of features. Filtering Data: Filter out unnecessary data or select specific features using select() or filter().
6. Logistic Regression: Nắm vững thuật toán logistic regression: công thức tính y\_prediction trong thuật toán, loss function, gradient descent

Logistic Regression is a statistical method used for binary classification. It's a fundamental algorithm in machine learning, especially useful when the dependent variable is categorical. Here's an overview covering its aspects:

Basic Concept:

* Purpose: Logistic Regression is used to predict the probability of a binary outcome (1/0, True/False, Yes/No) based on one or more predictor variables (features).
* Output: It outputs probabilities, which are then transformed into a binary format for classification purposes.

Mathematical Foundation:

A yellow rectangular object with black text

Description automatically generated

Loss Function in Logistic Regression

In logistic regression, the goal is to find the best fitting model to describe the relationship between the dichotomous characteristic of interest (dependent variable) and a set of independent variables. The loss function is a measure of how well a given model does this.

Binary Cross-Entropy Loss: Also known as log loss, it is the most common loss function used in logistic regression. It measures the performance of a classification model whose output is a probability value between 0 and 1. The loss increases as the predicted probability diverge from the actual label.

Formula: For a binary classification problem, the loss function can be defined as:

A black text with black letters

Description automatically generated

Gradient Descent

Gradient descent is an optimization algorithm used for minimizing the loss function in logistic regression. It is used to update the parameters (coefficients) of the logistic regression model.

Basic Idea: The basic idea behind gradient descent is to update the model parameters iteratively to minimize the loss function. This is done by moving the parameters in the direction of the steepest decrease in loss.

Procedure:

* Initialization: Start with initial values for the model parameters.
* Compute Gradient: Calculate the gradient of the loss function with respect to each parameter. The gradient is a vector that points in the direction of the greatest increase of the function.
* Update Parameters: Adjust the parameters in the opposite direction of the gradient. The size of the step is determined by the learning rate.
* Iterate: Repeat the process until the loss function converges to a minimum value.

Learning Rate: The learning rate determines the size of the steps taken to reach the minimum. A too-small learning rate makes the process slow, while a too-large learning rate can lead to overshooting the minimum.

Convergence: The process continues until the algorithm reaches a point where the loss function does not decrease significantly with each iteration, indicating convergence.

Challenges in Gradient Descent

* Choosing a Proper Learning Rate: If the learning rate is too high, the algorithm might overshoot the minimum. If it's too low, the algorithm will take too long to converge.
* Local Minima and Saddle Points: In more complex models, gradient descent can get stuck in local minima or saddle points, although this is less of a problem in logistic regression.
* Scaling of Features: Gradient descent requires careful feature scaling for efficient and correct convergence.

Variants of Gradient Descent

* Batch Gradient Descent: Uses the entire dataset to calculate the gradient of the loss function.
* Stochastic Gradient Descent (SGD): Uses a single data point at each iteration to calculate the gradient. It can be faster but is also noisier.
* Mini-batch Gradient Descent: A compromise between batch and stochastic gradient descent. It uses a subset of data to calculate the gradient and is often used in practice for its balance between efficiency and accuracy.

1. Decision Tree: Nắm vững thuật toán, khái niệm Entropy, Information Value.

Basic Concept

A decision tree is a flowchart-like tree structure where an internal node represents a feature (or attribute), the branch represents a decision rule, and each leaf node represents the outcome. The topmost node in a decision tree is known as the root node. It learns to partition on the basis of the attribute value. It partitions the tree in a recursive manner called recursive partitioning. This flowchart-like structure helps in decision making.

Key Components:

* Root Node: It represents the entire population or sample and this further gets divided into two or more homogeneous sets.
* Splitting: It is a process of dividing a node into two or more sub-nodes.
* Decision Node: When a sub-node splits into further sub-nodes, it is called a decision node.
* Leaf/Terminal Node: Nodes with no children (no further split) are called Leaf or Terminal nodes.
* Pruning: The process of removing sub-nodes of a decision node is called pruning.
* Branch/Sub-Tree: A sub-section of the entire tree is called a branch or sub-tree.
* Parent and Child Node: A node, which is divided into sub-nodes is called the parent node of sub-nodes, whereas sub-nodes are the child of the parent node.

How Decision Trees Work:

* Selection of the Best Attribute: Use a statistical measure, like Gini impurity or entropy in case of classification, and variance reduction in case of regression, to identify the attribute that best separates the samples into homogeneous subgroups.
* Splitting: Create a decision node specifying the selected attribute, and divide the dataset into subsets that contain possible values for this attribute.
* Recursive Splitting: Repeat this process recursively for each child node until one of the conditions for stopping is reached (such as a maximum tree depth, or no further improvement is possible).
* Tree Pruning: Sometimes the tree might overfit the data, leading to poor generalization. Pruning methods are used to avoid this, such as minimal cost-complexity pruning.

Advantages

* Easy to Understand: Decision trees are simple to interpret and visualize.
* Minimal Data Preprocessing: Requires less data cleaning compared to other techniques.
* Can Handle Both Numeric and Categorical Data: Works well with both types of data.
* Non-Parametric Method: No assumptions about the space distribution and the classifier structure.

Disadvantages

* Overfitting: Decision trees can create over-complex trees that do not generalize well to new data.
* Instability: Small variations in the data can result in a completely different tree.
* Greedy Algorithms: Decision tree algorithms are considered greedy because they might settle for a locally optimal solution.

Variants and Extensions

* Random Forest: An ensemble of decision trees, which averages or combines the results of individual trees to improve overall performance and reduce overfitting.
* Boosted Trees: Techniques like Gradient Boosting use decision trees as weak learners in an iterative way to improve model predictions.

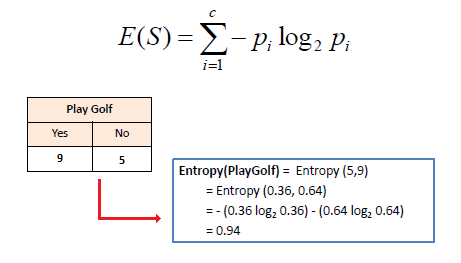
Applications

* Decision trees are used in a wide range of fields from medicine, manufacturing, finance, to astronomy. They are particularly useful for operational and strategic planning decisions.
* They are used for classification tasks like credit scoring, diagnosis, and customer segmentation, as well as for regression tasks in predicting housing prices, stock prices, etc.

Entropy

Entropy is a statistical measure of impurity or disorder in a set of data. It's rooted in information theory, where it quantifies the amount of uncertainty or surprise associated with random variables.

Entropy Formula:



Interpretation:

Low entropy (close to 0) means the set is mostly homogeneous.

High entropy (up to 1 in binary classification, higher in multi-class problems) indicates a heterogeneous or diverse set.

Information Gain:

Information Gain, on the other hand, measures the reduction in this entropy or unpredictability, brought about by partitioning the data based on an attribute. In other words, it quantifies how much knowing an attribute's value reduces the uncertainty about the class.

Entropy Formula:

A math equations on a white background

Description automatically generated

Interpretation:

* A high information gain for an attribute means that knowing the attribute's value gives us a lot of information about what class a data point belongs to.
* In decision tree algorithms like ID3 or C4.5, the attribute with the highest information gain is chosen to split the data at each node.

Relationship Between Entropy and Information Gain:

* Entropy provides a measure of the impurity or randomness in a dataset.
* Information Gain measures the effectiveness of an attribute in reducing that uncertainty or impurity.
* The decision tree algorithm selects the attributes that lead to the largest information gain (i.e., the greatest reduction in entropy) at each node.

Practical Use in Decision Trees:

Selecting Splits: In a decision tree, you want to split your data to make your subsets as pure as possible. The attribute that results in the highest information gain is chosen to split the dataset at each node.

Stopping Criteria: These concepts also help in determining when to stop growing the tree. For instance, if an attribute results in a perfect classification (entropy = 0 in the subsets), the branch can be stopped.

Limitations:

Computationally Intensive: Calculating entropy and information gain for every attribute at every decision point can be computationally expensive for large datasets.

Biased Towards Certain Attributes: Information gain can be biased towards attributes with a larger number of distinct values (like ID numbers or unique strings).