

BIG DATA PROCESSING AND ANALYTICS

UNIT 3

CONTENT



Map Reduce programming model
for distributed data
processing



Introduction to
Apache Spark:

RDDs
Data Frames
Spark SQL,



Machine learning
algorithms for
big data
analytics:

Classification
Regression
Clustering



Real-time analytics and stream
processing with Apache Flink.

MapReduce

MapReduce

- model for performing **Big Data analytics**.
- It is primarily designed for distributed platforms, like clusters.
- There are **three main implementations**:
 - **Google's proprietary version**.
 - **Hadoop** (open-source)
 - **Spark** (open-source)
- Other platforms stem from these basic frameworks.

Core Idea

- The core idea revolves around two main input functions:
 - **Map function**: Processes data in parallel using available resources.
 - **Reduce function**: Aggregates the processed

Distributed storage systems

- such as **HDFS** (Hadoop Distributed File System) and **Google File System (GFS)** work with MapReduce to handle input, intermediate, and output data.



MapReduce



MapReduce Workflow : involves three steps

- **Mapping Step:** Programmer implements the **Map function**.
- The **input dataset** is partitioned and stored in a **distributed file system (DFS)**.
- Each Map task processes a partition and generates **intermediate data**, which is stored locally on the worker nodes.

Shuffling Step:

- The **intermediate data** is distributed across the computational resources to prepare for the Reduce step.
- This step is often a **communication bottleneck** due to its all-to-all data

Reduce Step:

- The **Reduce function** is executed to generate the final output from the intermediate data.

Multistage MapReduce Jobs

- **Multistage MapReduce jobs** are common, where the output of one MapReduce job serves as the intermediate data for the next job.
- An example is **Yahoo! WebMap**, where a chain of MapReduce jobs processes web data.

Shortcoming of MapReduce

Rigid Structure Enforcing Map and Reduce Phases:

- MapReduce enforces a strict data processing flow that must fit into the Map and Reduce phases. This restricts flexibility, as other useful data processing operations, such as joins, filters, flatMap, groupByKey, unions, and intersections, are either missing or not natively

Lack of Comprehensive Workflows:

- Beyond simple Map and Reduce tasks, more complex workflows like data joins, filtering, and transformations are not well-supported. These operations often require custom coding or additional frameworks to handle efficiently.

Acyclic Data Flow:

- MapReduce is designed around an "acyclic data flow," where data flows from disk to disk, with intermediate results written to disk (HDFS). This design results in slow performance since each phase must read from and write to disk.

Stateless Nature of MapReduce:

- The architecture treats each node as a stateless machine. After each Map or Reduce phase, the system must save the results to disk, which can lead to significant inefficiencies, especially in tasks requiring frequent iterations, such as machine learning algorithms.

Shortcoming of

M

D

1

Inefficient for Iterative Tasks:

- Due to constant disk I/O between phases, MapReduce performs poorly for iterative tasks, such as those required in machine learning, where data needs to be processed repeatedly.

Java-Centric Support:

- The original MapReduce framework was primarily designed for Java. While other languages can be used through additional frameworks, the native support is Java-centric, and additional effort is required to support other languages.

Designed for Batch Processing:

- MapReduce was built for batch processing, where large amounts of data are processed in batches. It is not optimized for real-time or interactive data processing.

Lack of Support for Real-Time Processing:

- MapReduce does not natively support interactive data analysis or streaming data processing. Additional frameworks like Apache Spark or Apache Flink are required for such use cases.



Apache Spark



One Solution is Apache Spark



A Modern General Framework:

This framework addresses many of the limitations inherent in MapReduce, offering a more versatile and powerful solution for distributed data processing.



Integration with Hadoop Ecosystem:

It seamlessly integrates with the Hadoop ecosystem, utilizing existing components such as HDFS (Hadoop Distributed File System), YARN (Yet Another Resource Negotiator), HBase, S3 (Amazon's Simple Storage Service), and more. This ensures



Extensive Workflow Support:

Unlike MapReduce, this framework supports a wide variety of workflows, such as joins, filters, flatMap, distinct, groupByKey, reduceByKey, sortByKey, collect, count, first, and more. It offers around 30 efficient distributed operations, making



In-Memory Caching:

One of its standout features is the ability to cache data in memory, which significantly speeds up iterative operations. This is particularly beneficial for tasks like graph processing, machine learning algorithms, and any other

One Solution is Apache Spark



.Multi-Language Support:

The framework natively supports multiple popular programming languages, including Scala, Java, Python, and R, making it accessible to a wide range of developers.



Interactive Shells for Data Analysis:

It provides interactive shells for languages like Scala and Python, enabling exploratory data analysis and real-time interaction with data, which is a significant improvement over the batch processing nature of MapReduce.



User-Friendly Spark API:

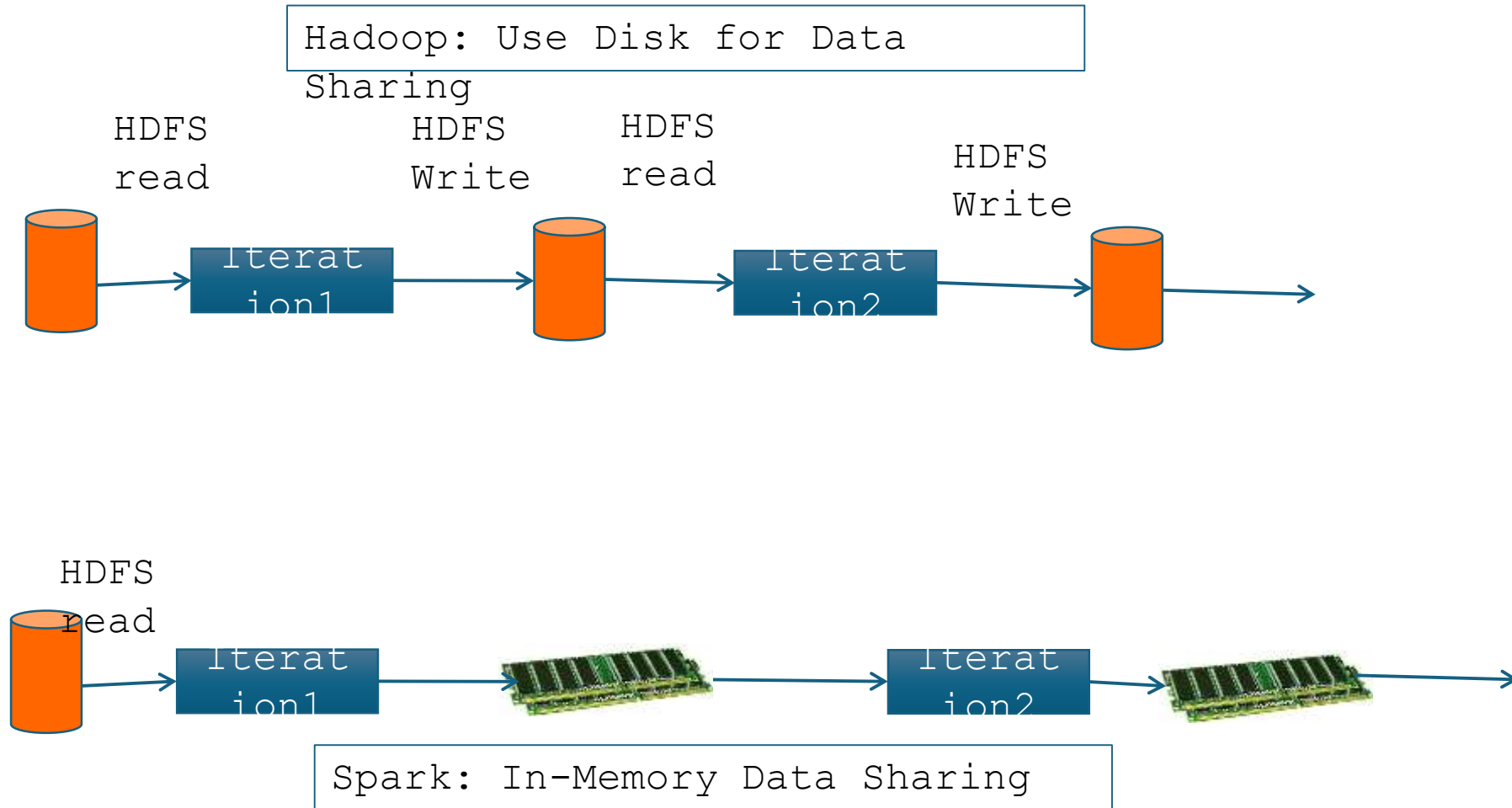
The Spark API is designed to be extremely simple and intuitive to use, allowing developers to quickly implement data processing tasks without needing extensive boilerplate code or complicated configurations.



Development and Origin:

Originally developed at UC Berkeley's AMPLab, it is now maintained and further developed by Databricks, a company founded by the creators of this framework. Databricks continues to push innovation in the big data processing field.

Spark Uses Memory instead of Disk



MapReduce vs Apache spark

Feature	MapReduce	Apache Spark
Developed By	Google (original implementation)	Apache Software Foundation
Processing Model	Disk-based (writes intermediate data to disk)	In-memory (stores intermediate data in memory)
Speed	Slower due to multiple read/write operations to disk	Faster due to in-memory processing
Ease of Use	Requires writing multiple MapReduce jobs for complex tasks	Higher-level APIs (e.g., DataFrames, Datasets) simplify code
API Support	Only supports Map and Reduce functions	Supports rich APIs: Map, Reduce, Join, Filter, etc.
Data Processing Type	Batch processing	Both batch and real-time (stream) processing
Latency	High latency due to disk I/O	Low latency because of in-memory operations
Fault Tolerance	High (uses HDFS for replication)	High (utilizes lineage graphs to recover lost data)
	Poor handling of iterative	

MapReduce vs Apache spark

Feature	MapReduce	Apache Spark
Fault Recovery	Recovery by restarting failed tasks and using HDFS replicas	Recovery using lineage and in-memory data
Supported Languages	Java, Python, C++	Java, Scala, Python, R, SQL
Machine Learning Libraries	Limited (Mahout)	Comprehensive support (MLlib, GraphX)
Use Case	Suitable for simple, large-scale batch processing	Suitable for iterative, real-time, and machine learning tasks
Resource Management	Uses Hadoop YARN	Can use Hadoop YARN, Mesos, or standalone cluster manager
Complexity	More complex for multi-step workflows (e.g., chaining MapReduce jobs)	Less complex for multi-step operations (integrates multiple steps in a single job)
Performance	Slower for iterative tasks or small datasets	Much faster due to DAG (Directed Acyclic Graph) execution

Spark vs MapReduce

MapReduce requires files to be stored in HDFS, Spark does not!

Spark also can perform operations up to 100x faster than MapReduce

So how does it achieve this speed?

MapReduce writes most data to disk after each map and reduce operation

Spark keeps most of the data in memory after each transformation

Spark can spill over to disk if the memory is filled

Spark

You can think of Spark as a flexible
alternative to MapReduce



Spark can use data stored in a variety of
formats

Cassand
ra

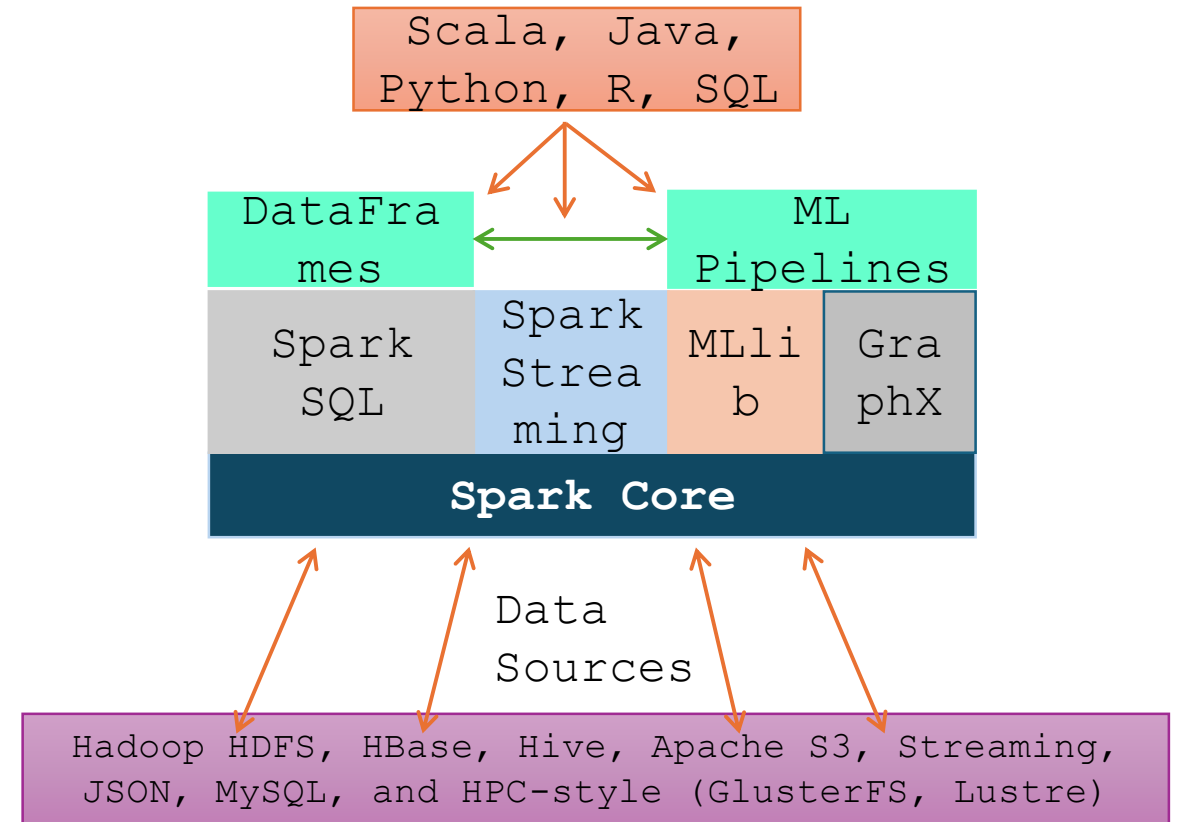
AWS S3

HDFS

And
more

Apache Spark

Apache Spark supports data analysis, machine learning, graphs, streaming data, etc. It can read/write from a range of data types and allows development in multiple languages.



Spark RDDs



At the core of Spark is the idea of a Resilient Distributed Dataset (RDD)



Resilient Distributed Dataset (RDD) has 4 main features:

Distributed Collection of Data

Fault-tolerant

Parallel operation - partitioned

Ability to use many data sources

SPARK RDD



Spark Core:

The foundational engine responsible for scheduling, memory management, fault recovery, and distributed processing.



Spark SQL:

Used for structured data processing, Spark SQL allows users to run SQL queries, read data from various sources, and join data from different formats.



Spark Streaming:

Provides scalable and fault-tolerant stream processing of live data streams.



MLlib:

A machine learning library for scalable and distributed machine learning tasks.

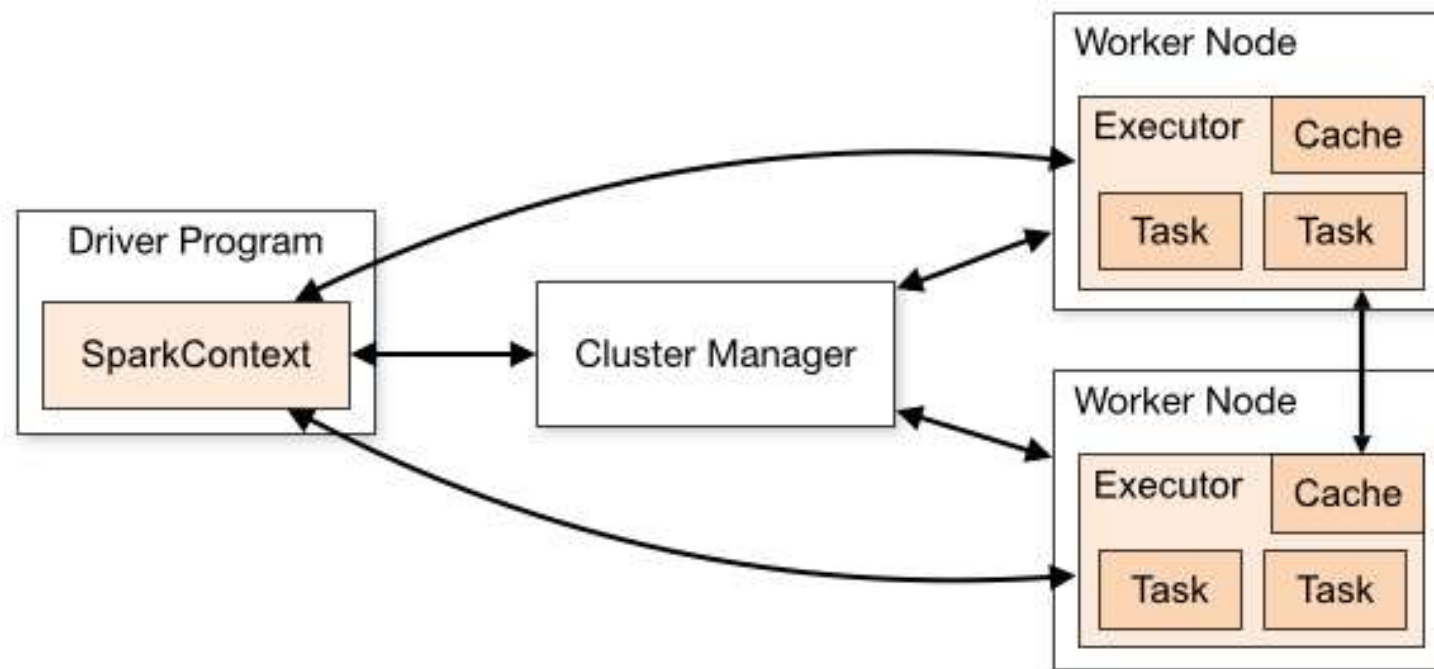


GraphX:

For processing and analysing graph data.

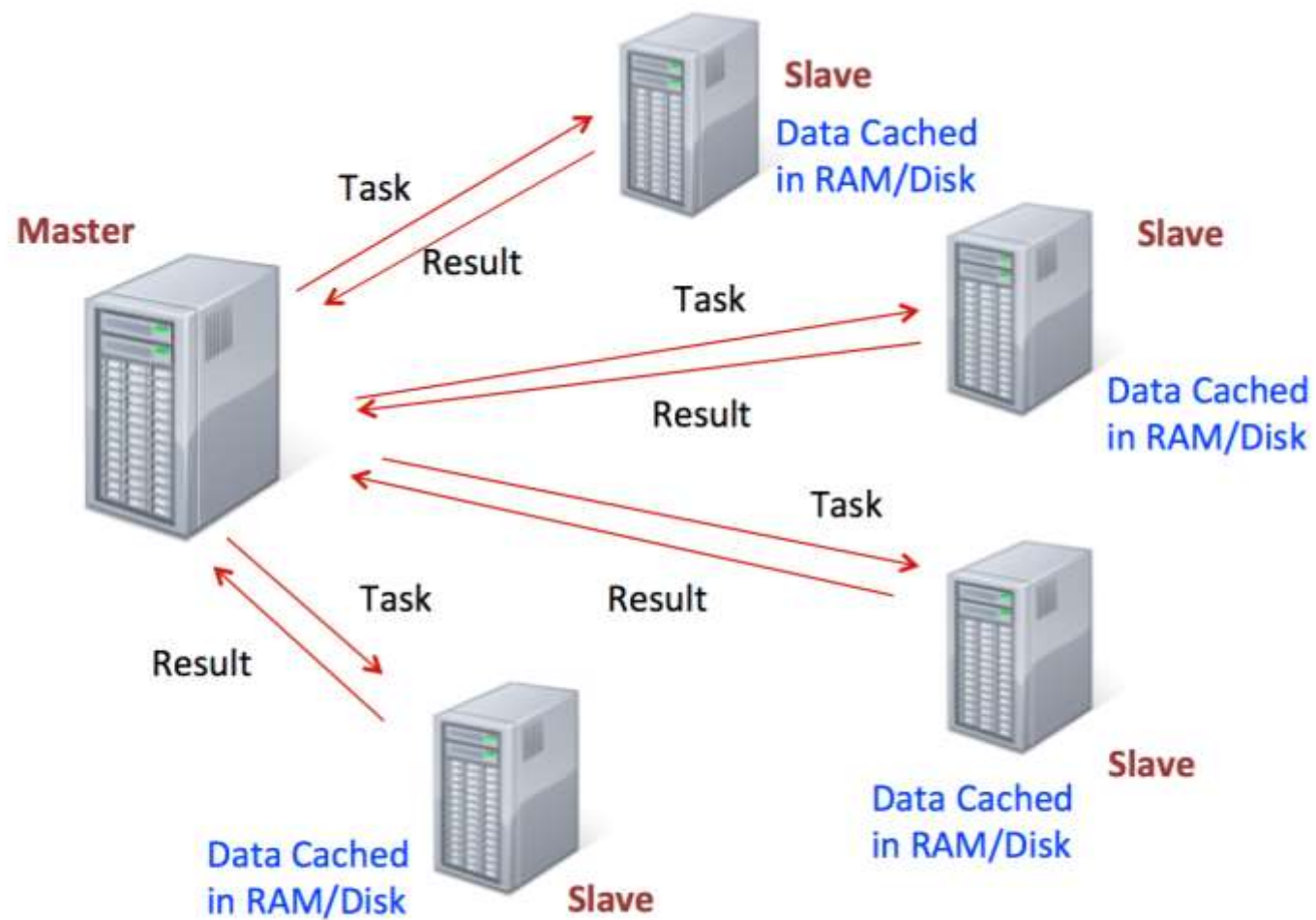


Spark RDDs





Spark RDDs





Spark RDDs

RDDs are immutable, lazily evaluated, and cacheable

There are two types of Spark operations:

- Transformations
- Actions

Transformations are basically a recipe to follow.

Actions actually perform what the recipe says to do and returns something back.

Spark RDDs



This
behaviour
carries over
to the syntax
when coding.



A lot of
times you
will write a
method call,
but won't see
anything as a
result until
you call the
action.



This makes
sense because
with a large
dataset, you
don't want to
calculate all
the
transformatio
ns until you
are sure you
want to
perform them!

Spark RDDs



When discussing Spark syntax you will see RDD versus DataFrame syntax show up.



With the release of Spark 2.0, Spark is moving towards a DataFrame based syntax, but keep in mind that the way files are being distributed can still be thought of as RDDs, it is just the typed out syntax that is changing

Spark DataFrames

Spark DataFrames are also now the standard way of using Spark's Machine Learning Capabilities.

Spark DataFrame documentation is still pretty new and can be sparse.

Spark DataFrames

Python and Spark



Spark DataFrames hold data in a column and row format.



Each column represents some feature or variable.



Each row represents an individual data point.

Python and Spark



Spark began with something known as the “RDD” syntax which was a little ugly and tricky to learn.



Now Spark 2.0 and higher has shifted towards a DataFrame syntax which is much cleaner and easier to work with!



Spark DataFrames can input and output data from a wide variety of sources.



We can then use these DataFrames to apply various transformations on the data.

Key Features of Spark DataFrames

Schema Information:

- DataFrames have schema information, meaning each column has a name and a type (string, integer, etc.).

Optimized Execution:

- They provide an abstraction that allows Spark's Catalyst Optimizer to optimize execution plans, resulting in more efficient queries.

Support for a Variety of Data Sources:

- DataFrames can be created from structured data files (e.g., CSV, JSON), tables in Hive, or external databases.

Integration with SQL:

- You can run SQL queries directly on DataFrames, which can integrate well with both structured and semi-structured data.

Lazy Execution:

- Like RDDs, DataFrames are lazily evaluated, meaning that transformations are not executed until an action is triggered.

Spark DataFrames Implementation Example in Python (PySpark)

```
from pyspark.sql import SparkSession
```

```
# Create a Spark session
```

```
spark = SparkSession.builder.appName("DataFrameExample").getOrCreate()
```

```
#The SparkSession object is the entry point to programming with  
DataFrames.
```

```
# Create a sample DataFrame from a list of tuples
```

```
data = [("Alice", 29), ("Bob", 31), ("Catherine", 25)]
```

```
columns = ["Name", "Age"]
```

```
df = spark.createDataFrame(data, columns)
```

```
# Show the DataFrame
```

```
df.show()
```

OUTPUT

```
+-----+----+  
|  Name|Age|  
+-----+----+  
| Alice| 29|  
|  Bob| 31|  
| Catherine| 25|  
+-----+----+
```


Loading Data from a CSV File

Load data from a CSV file into a DataFrame

```
df = spark.read.csv("path/to/file.csv",  
header=True, inferSchema=True)
```

Show the first 5 rows of the DataFrame

```
df.show(5)
```

#The inferSchema=True parameter allows Spark to automatically detect the schema of the file, and header=True treats the first row as a header.

Basic DataFrame Operations

- **Filtering:** Select rows based on conditions.
`df.filter(df.Age > 30).show()`
- **Selecting Columns:** You can select specific columns using the `select()` method.
`df.select("Name", "Age").show()`
- **Grouping and Aggregating Data:** You can group data by columns and perform aggregations like sum, average, etc.
`df.groupBy("Age").count().show()`
- **SQL Queries on DataFrames:**
Register the DataFrame as a SQL temporary view
`df.createOrReplaceTempView("people")`
Query using SQL
`spark.sql("SELECT Name, Age FROM people WHERE Age > 30").show()`
- **Writing DataFrame to Disk**
Save DataFrame to CSV:
`df.write.csv("output/path", header=True)`
Save DataFrame as Parquet:
`df.write.parquet("output/path")`

Spark SQL

01

Spark module that allows users to run SQL queries on large datasets stored in distributed systems.

02

It supports querying structured and semi-structured data and integrates well with

Spark's DataFrame API.

03

Spark SQL enables querying data via SQL as well as via the DataFrame API

04

Providing flexibility for working with data in a relational manner, even on distributed datasets.

Key Features of Spark SQL



Unified Data Access:

Spark SQL allows you to work with structured data from different sources like JSON, Parquet, Hive tables, JDBC, etc.



Integration with SQL:

You can use SQL queries seamlessly alongside Spark's native APIs.



Catalyst Optimizer:

Spark SQL uses an advanced query optimizer called the **Catalyst Optimizer**, which analyzes query plans and optimizes them for performance.



Compatibility with Hive:

Spark SQL can query existing Hive tables and access data stored in Hive's metadata store.



Supports UDFs (User-Defined Functions):

You can define your custom functions and use them in SQL queries.

Spark SQL Example

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("SparkSQLExample").getOrCreate()
# Load data from a CSV file into a DataFrame
df = spark.read.csv("path/to/people.csv", header=True, inferSchema=True)

# Show the DataFrame
df.show()
# Register DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

# Run SQL query
sqlDF = spark.sql("SELECT Name, Age FROM people WHERE Age > 30")
sqlDF.show()
```

```
# Using SQL functions to aggregate data
resultDF = spark.sql("SELECT AVG(Age) as Average_Age FROM people")
resultDF.show()
# Write DataFrame to Parquet format
df.write.parquet("people.parquet")

# Read the Parquet file into a DataFrame
parquetDF = spark.read.parquet("people.parquet")

# Create a SQL temporary view from the Parquet data
parquetDF.createOrReplaceTempView("parquet_people")

# Query the Parquet data using SQL
spark.sql("SELECT * FROM parquet_people WHERE Age > 30").show()
```

Using Hive with Spark SQL

```
# Enable Hive support
spark =
SparkSession.builder.appName("SparkSQLHiveExample").enableHiveSupport().
getOrCreate()

# Create a Hive table from a DataFrame
df.write.saveAsTable("people_hive_table")

# Query the Hive table
spark.sql("SELECT * FROM people_hive_table WHERE Age > 30").show()
```

User-Defined Functions (UDFs)

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Define a UDF to transform a column
def age_category(age):
    return "Senior" if age > 50 else "Adult"

# Register the UDF
spark.udf.register("ageCategory", age_category, StringType())

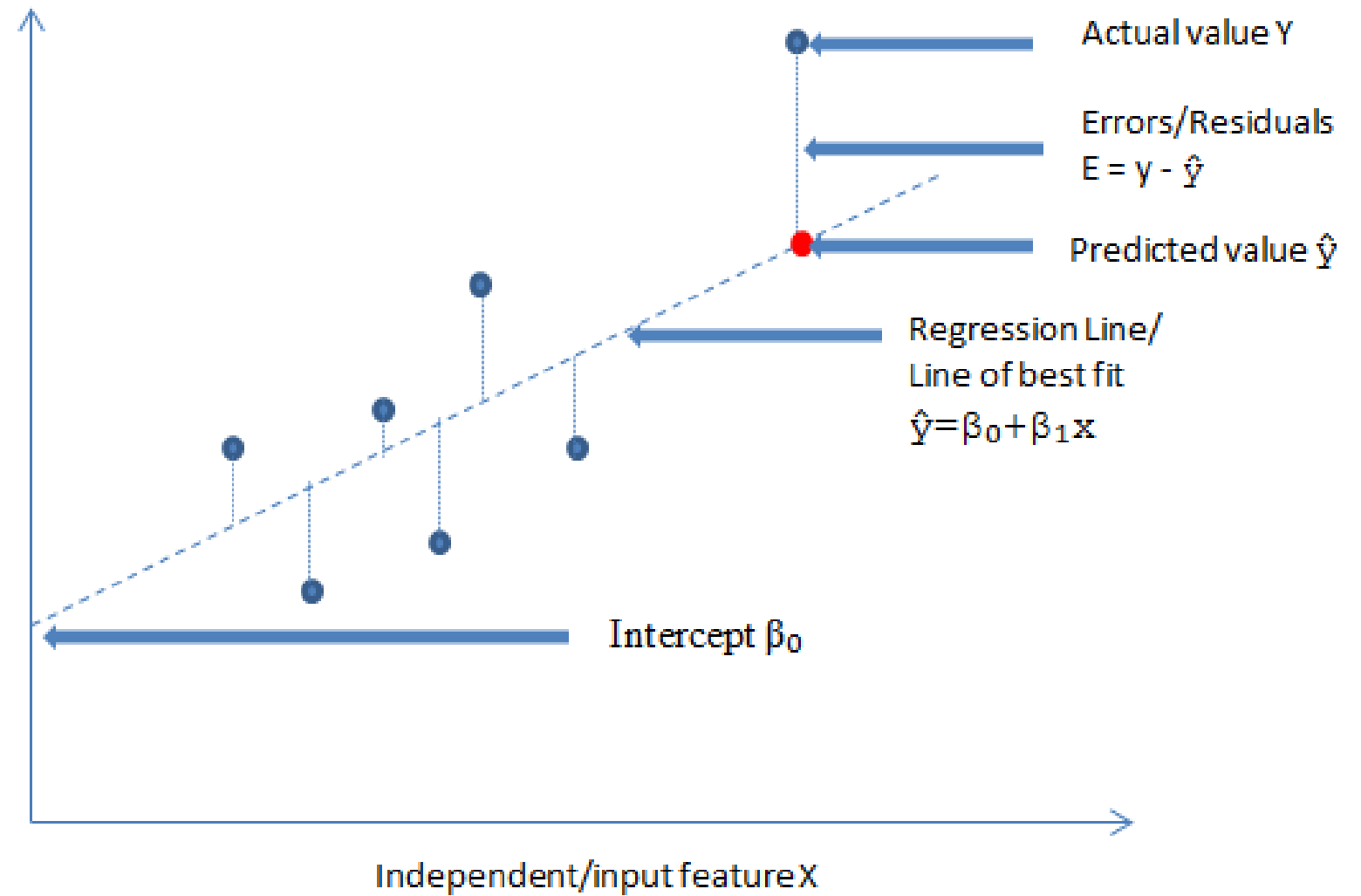
# Use the UDF in an SQL query
spark.sql("SELECT Name, Age, ageCategory(Age) as Category FROM
people").show()
```


Regression ML

- **Regression** is a statistical technique used in predictive modeling to determine the relationship between one dependent variable (target) and one or more independent variables (predictors).
- It's widely used in forecasting and predicting numerical values.
- **Types of Regression:**
 - 1.Linear Regression:**
 - 1.Models the relationship between the dependent and independent variable(s) by fitting a straight line to the data.
 - 2.Multiple Linear Regression:**
 - 1.Extends linear regression by using multiple predictors to predict the target.
 - 3.Polynomial Regression:**
 - 1.Models non-linear relationships by transforming predictors into higher-degree polynomials.
 - 4.Logistic Regression:**
 - 1.Used for binary classification (though technically not a regression model, it's often included under the term due to its similar approach).

a visual
representation of
a **simple linear**
regression model

Linear regressi on



Linear Regression

- The **regression line** captures the general trend of the data and provides predicted values \hat{Y} for given input values X .
- The **errors/residuals** indicate the difference between the actual observed values and the values predicted by the model.
- The **intercept** β_0 shows where the line crosses the Y-axis when $X = 0$.
- In practice, the goal of a linear regression model is to find the line that best fits the data by minimizing the residuals, usually using techniques like **least squares regression**.

Linear Regression

- **Regression Line (Line of Best Fit) :**

- The dashed line is the **regression line** which represents the relationship between the independent (input) variable **X** and the dependent (target) variable **Y**.
- The equation of the line is:
- $\hat{Y} = \beta_0 + \beta_1 X$ where:
 - \hat{Y} is the **predicted value** of Y,
 - β_0 is the **intercept**, representing the point where the line crosses the Y-axis,
 - β_1 is the **slope**, representing the rate of change of Y with respect to X.

- **Independent/Input Feature (X) :**

- The X-axis represents the **independent variable** or feature, denoted as **X**. It is the input feature we use to predict the dependent variable.

- **Dependent/Target Feature (Y) :**

- The Y-axis represents the **dependent variable** or the target we are trying to predict, denoted as **Y**. It depends on the input feature **X**.

Linear Regression

- **Predicted Value \hat{Y} :**
 - The **red dot** on the regression line represents the **predicted value** \hat{Y} for a given X. This is the value predicted by the regression model for a particular input value of **X**.
- **Actual Value (Y):**
 - The **blue dots** above and below the regression line represent the **actual values** of Y for the corresponding X values. These are the true target values in the dataset.
- **Errors/Residuals (E):**
 - The **vertical lines** between the actual data points (blue dots) and the regression line represent the **residuals** or **errors**, denoted as E.
 - Residual (Error) is the difference between the actual value Y and the predicted value
 - \hat{Y} : $E = Y - \hat{Y}$
 - These errors indicate how far off the predictions are from the actual data points. In linear regression, the goal is to minimize the sum of these residuals.

Logistic Regression



Not all labels are continuous, sometimes you need to predict categories, this is known as classification.



Logistic Regression is one of the basic ways to perform classification (don't be confused by the word "regression")

Logistic Regression

We want to learn about Logistic Regression as a method for **Classification**.

Some examples of classification problems:

- Spam versus "Ham" emails
- Loan Default (yes/no)
- Disease Diagnosis

Above were all examples of Binary Classification

Background

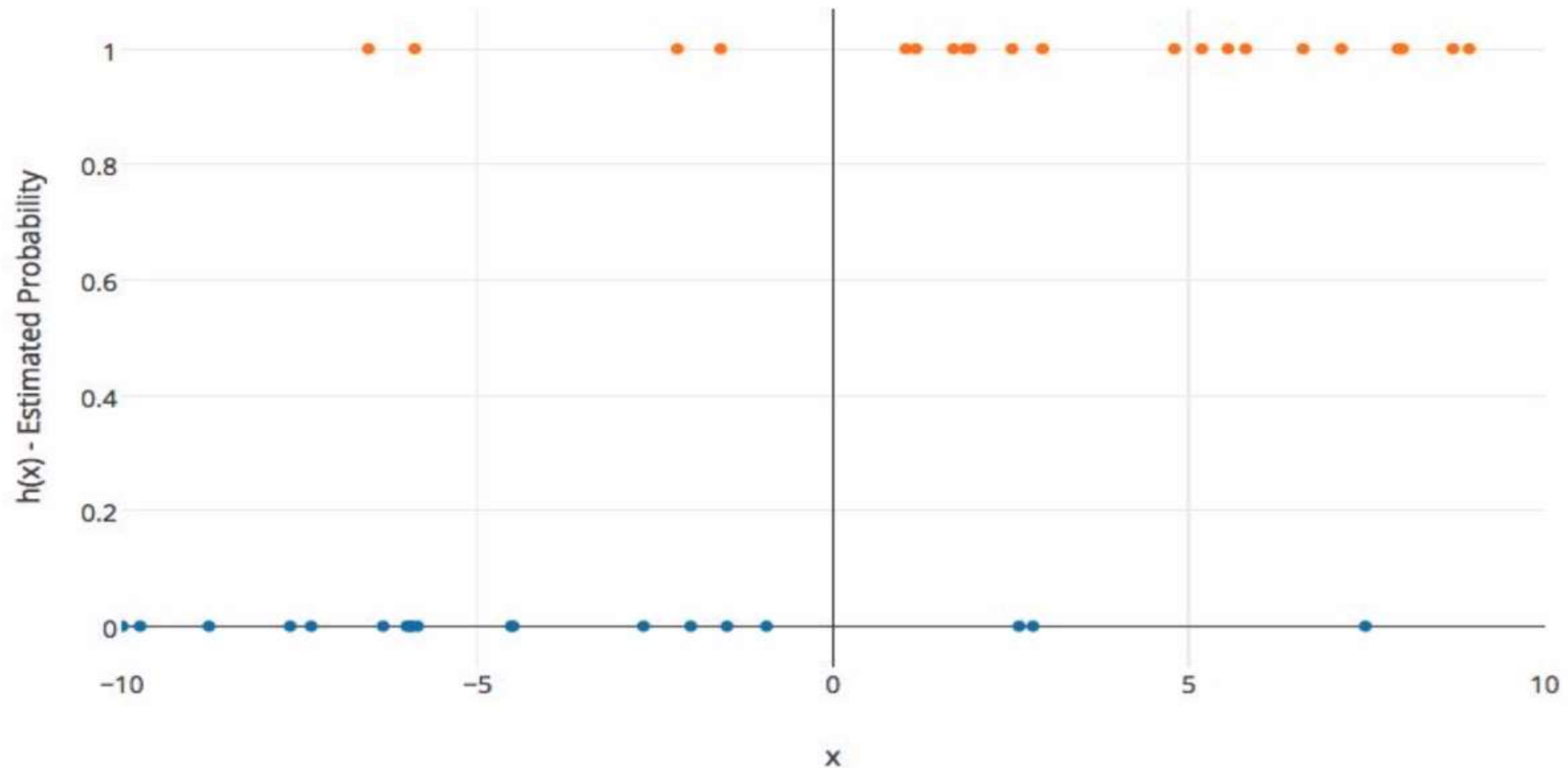
So far we've only seen regression problems where we try to predict a continuous value.

Although the name may be confusing at first, logistic regression allows us to solve **classification** problems, where we are trying to predict discrete categories.

The convention for binary classification is to have two classes 0 and 1.

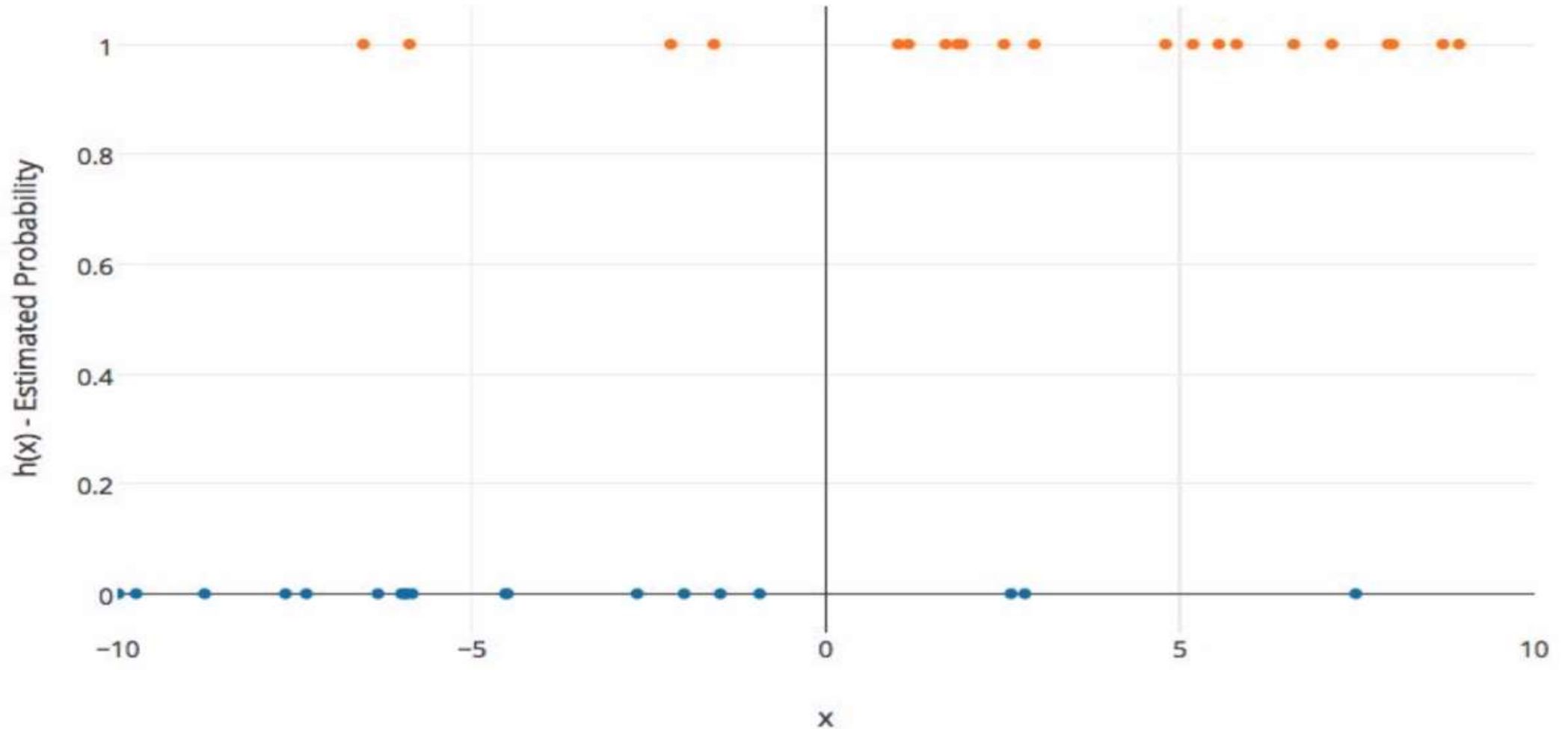
Background

- Imagine we plotted out some categorical data against one feature.



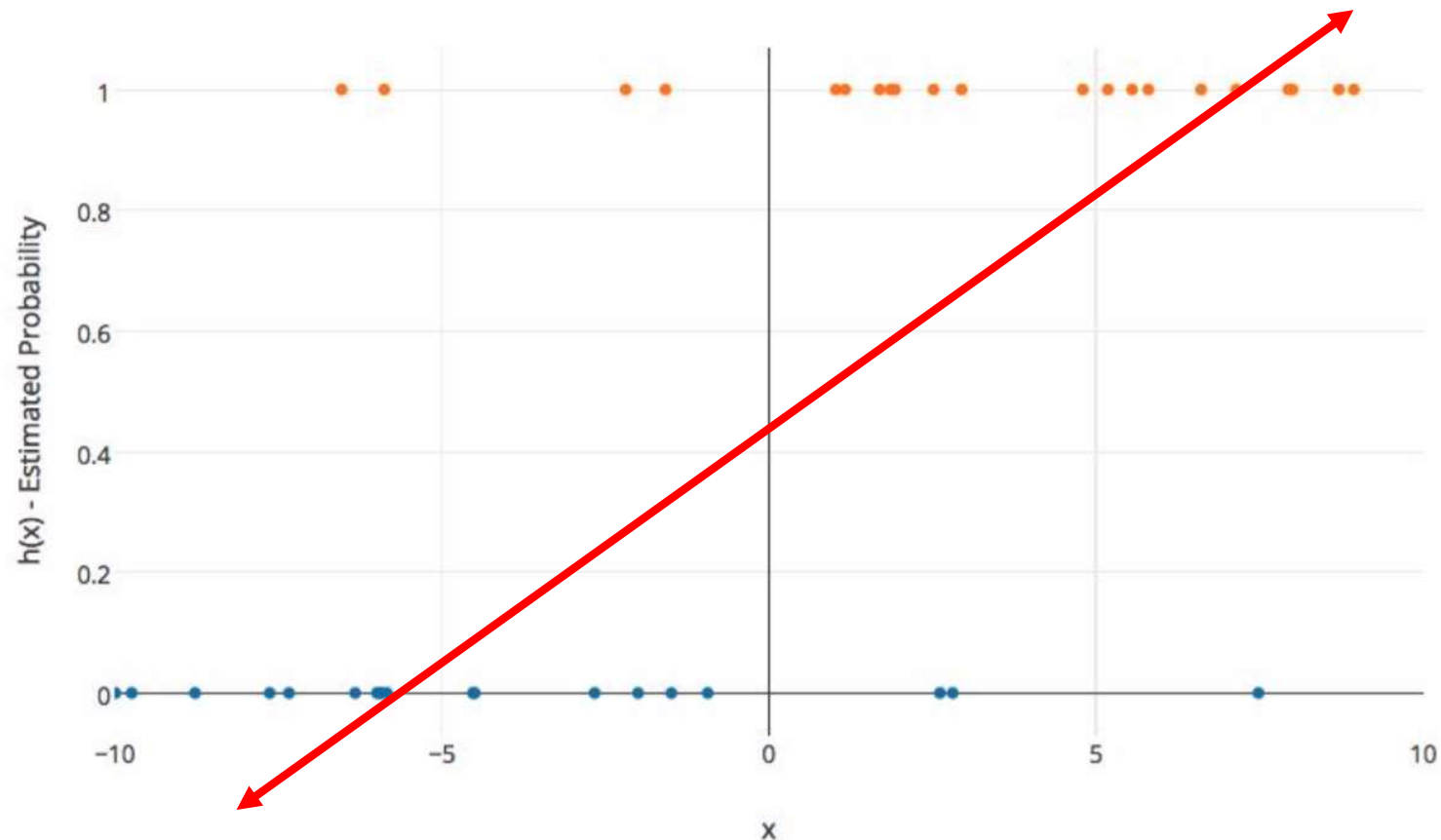
Background

- The X axis represents a feature value and the Y axis represents the probability of belonging to class 1.



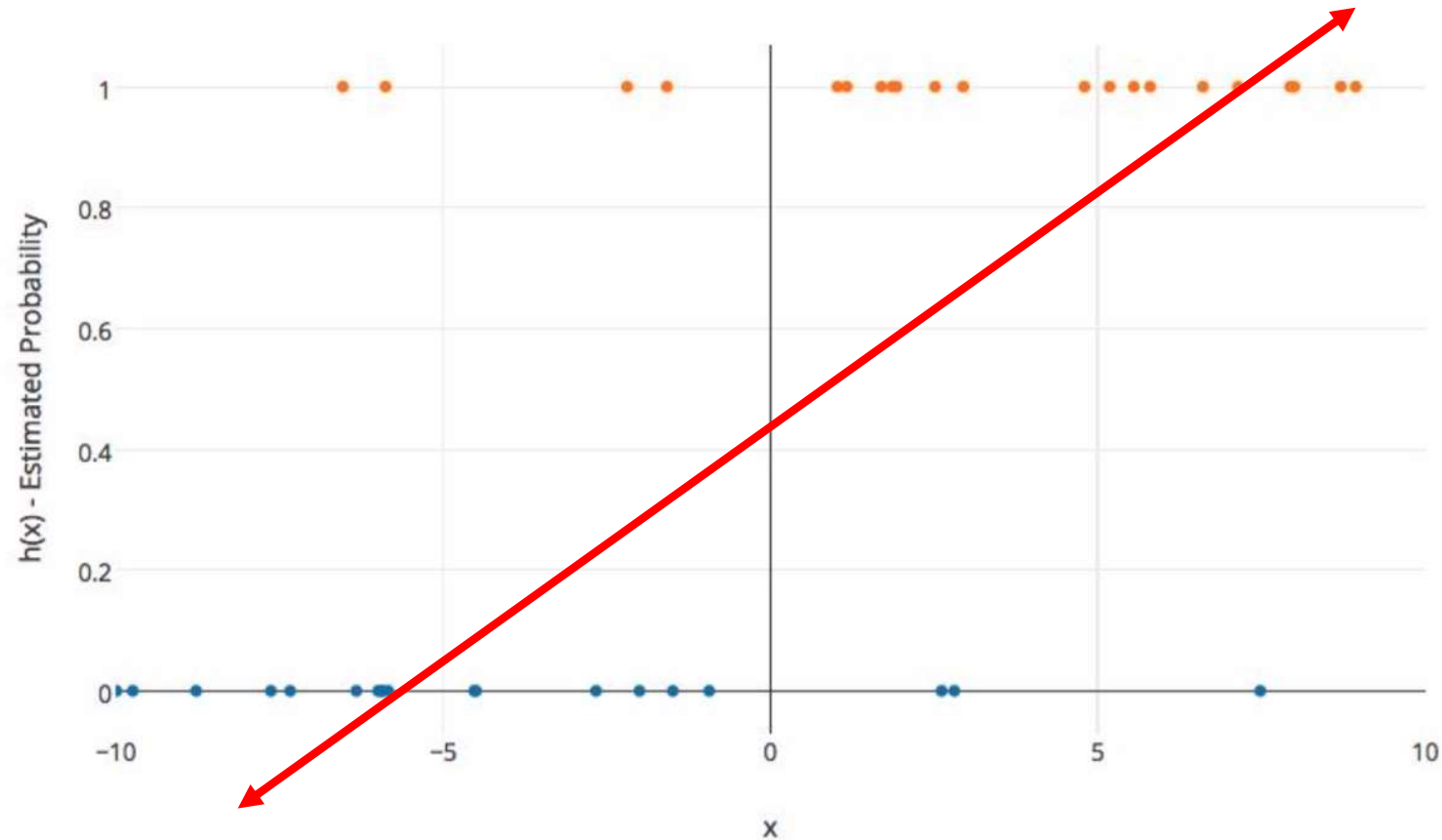
Background

- We can't use a normal linear regression model on binary groups. It won't lead to a good fit:



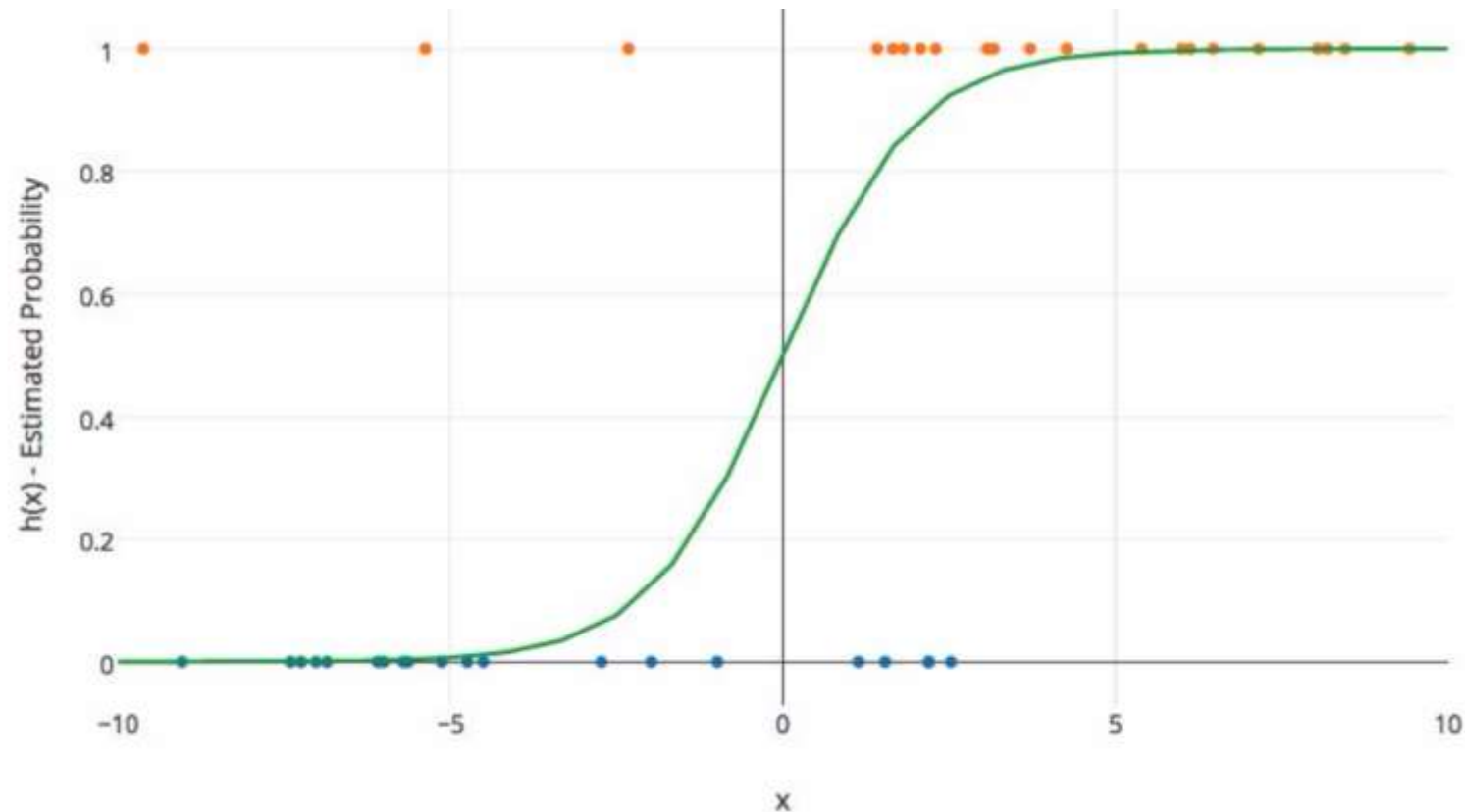
Background

- We need a function that will fit binary categorical data!



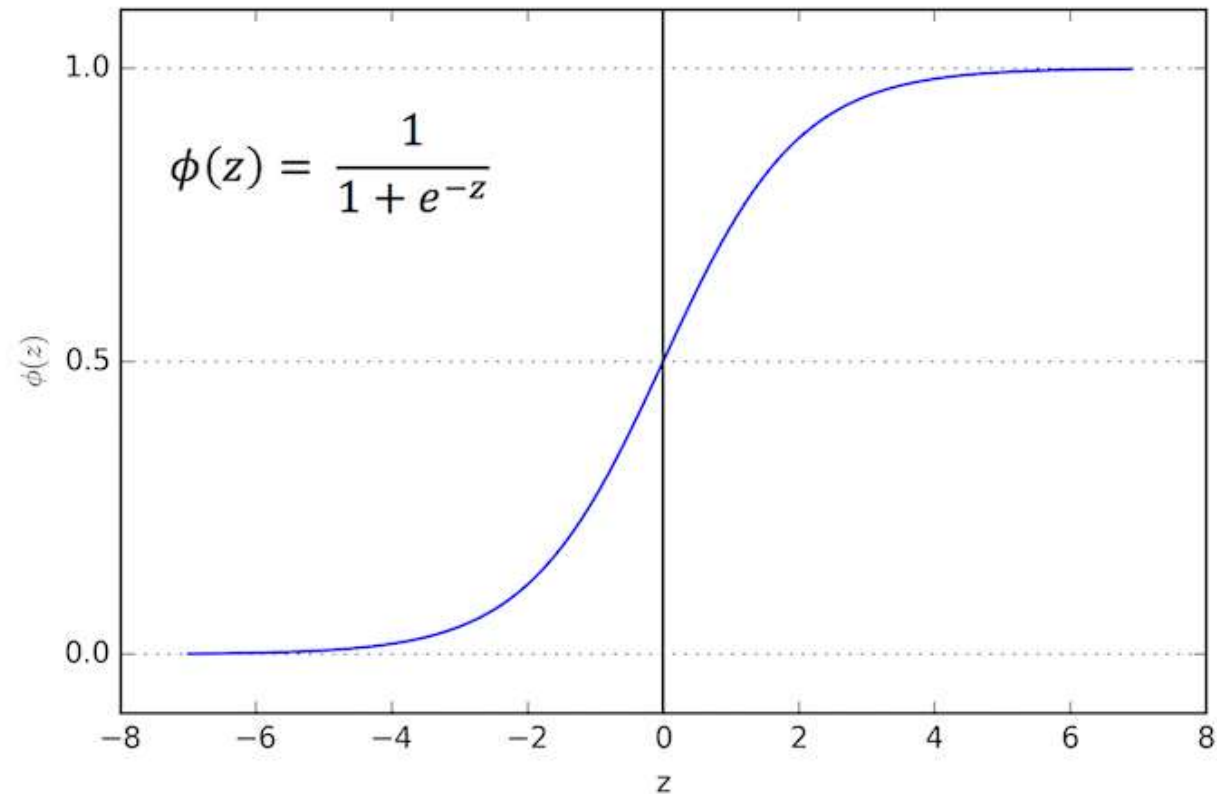
Background

- It would be great if we could find a function with this sort of behavior:



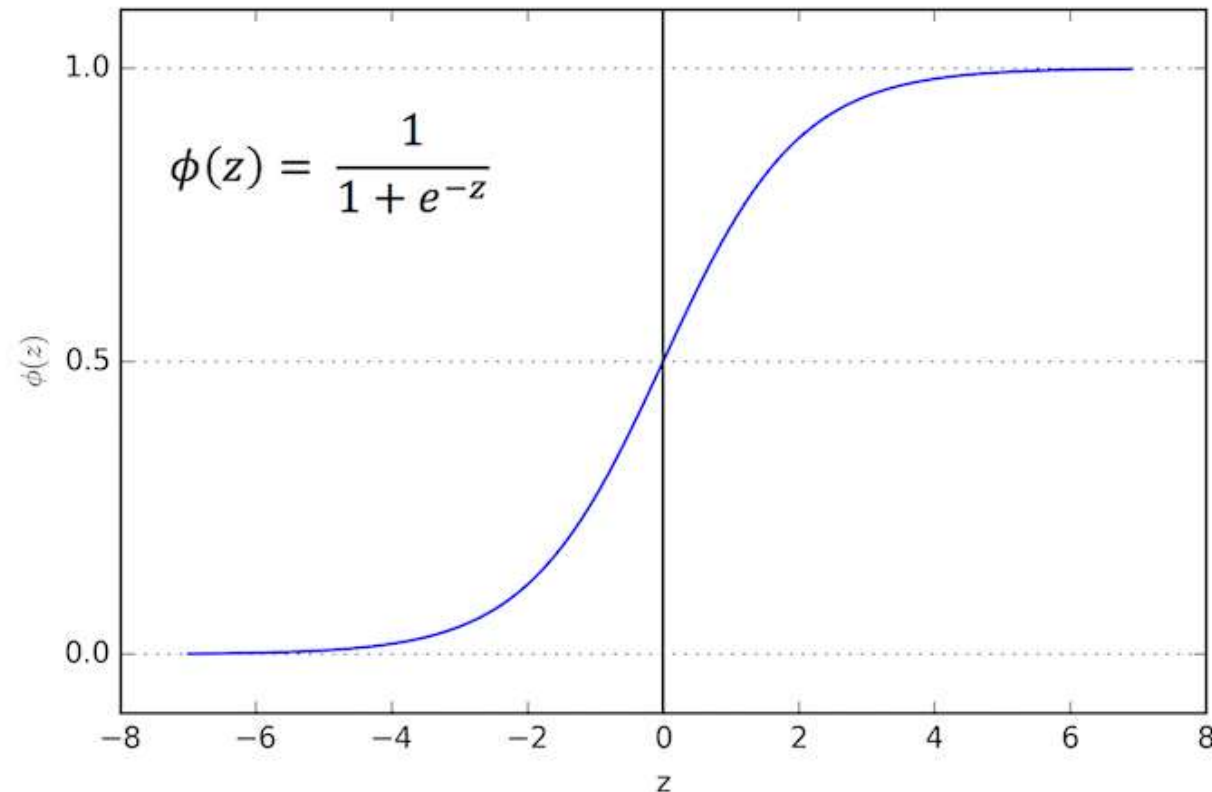
Sigmoid Function

- The Sigmoid (aka Logistic) Function takes in any value and outputs it to be between 0 and 1.



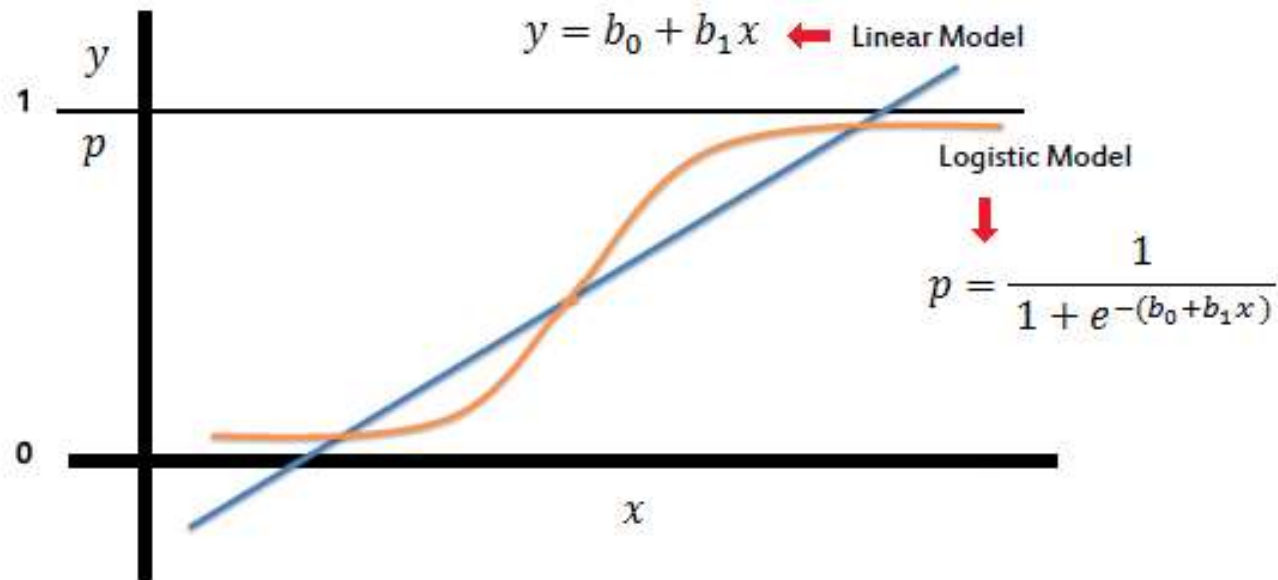
Sigmoid Function

- This means we can take our Linear Regression Solution and place it into the Sigmoid Function.



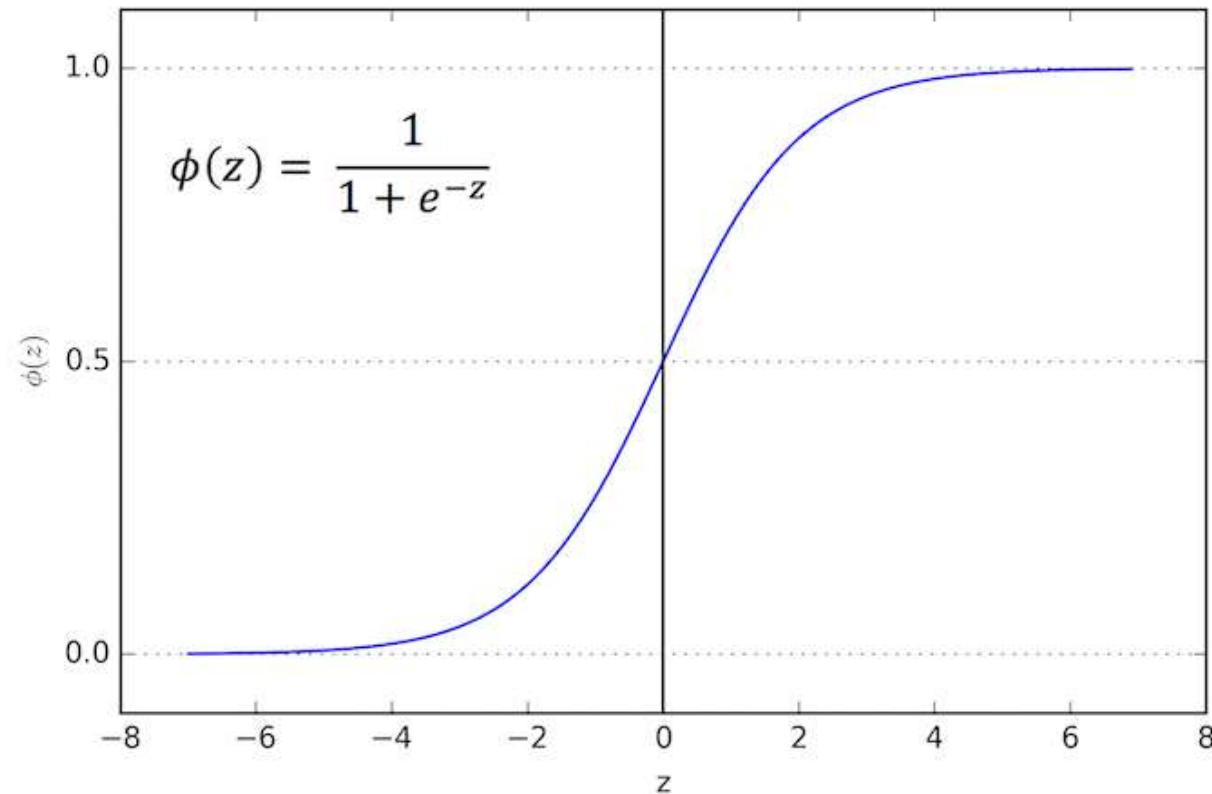
Sigmoid Function

- This means we can take our Linear Regression Solution and place it into the Sigmoid Function.



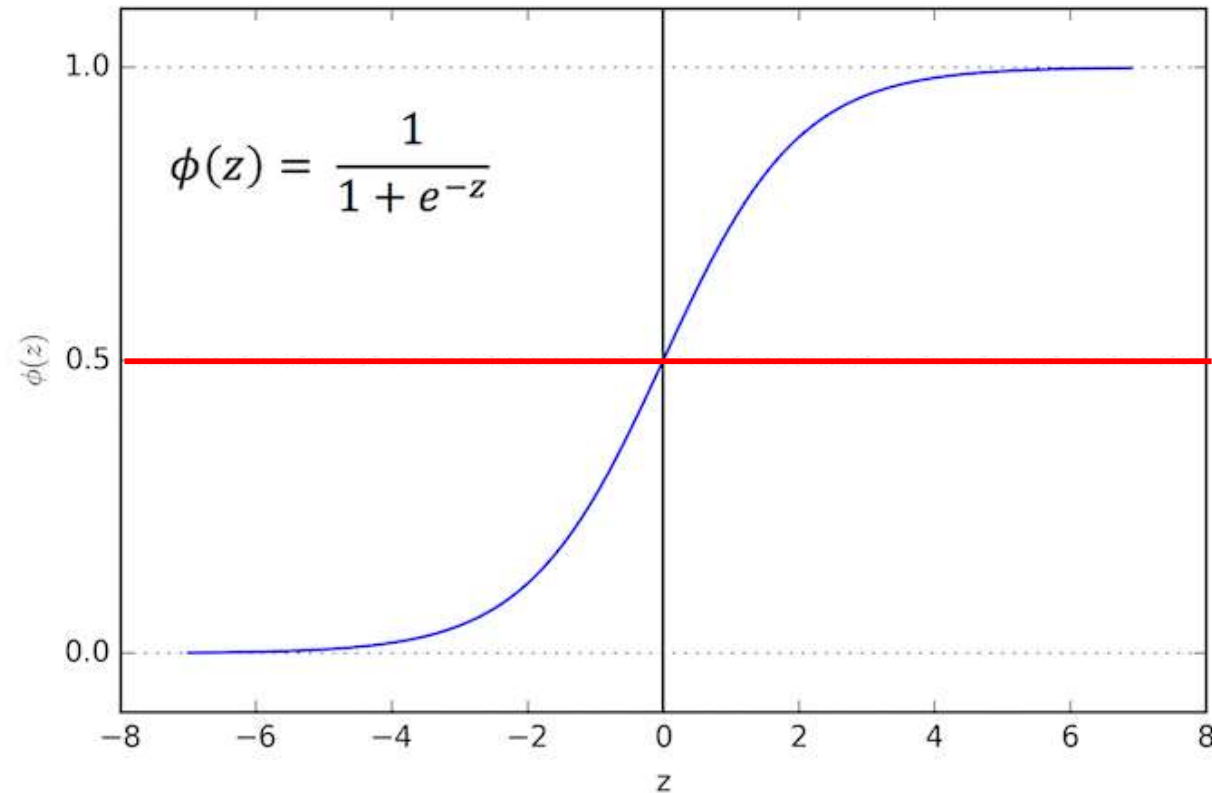
Sigmoid Function

- This results in a probability from 0 to 1 of belonging in the 1 class.



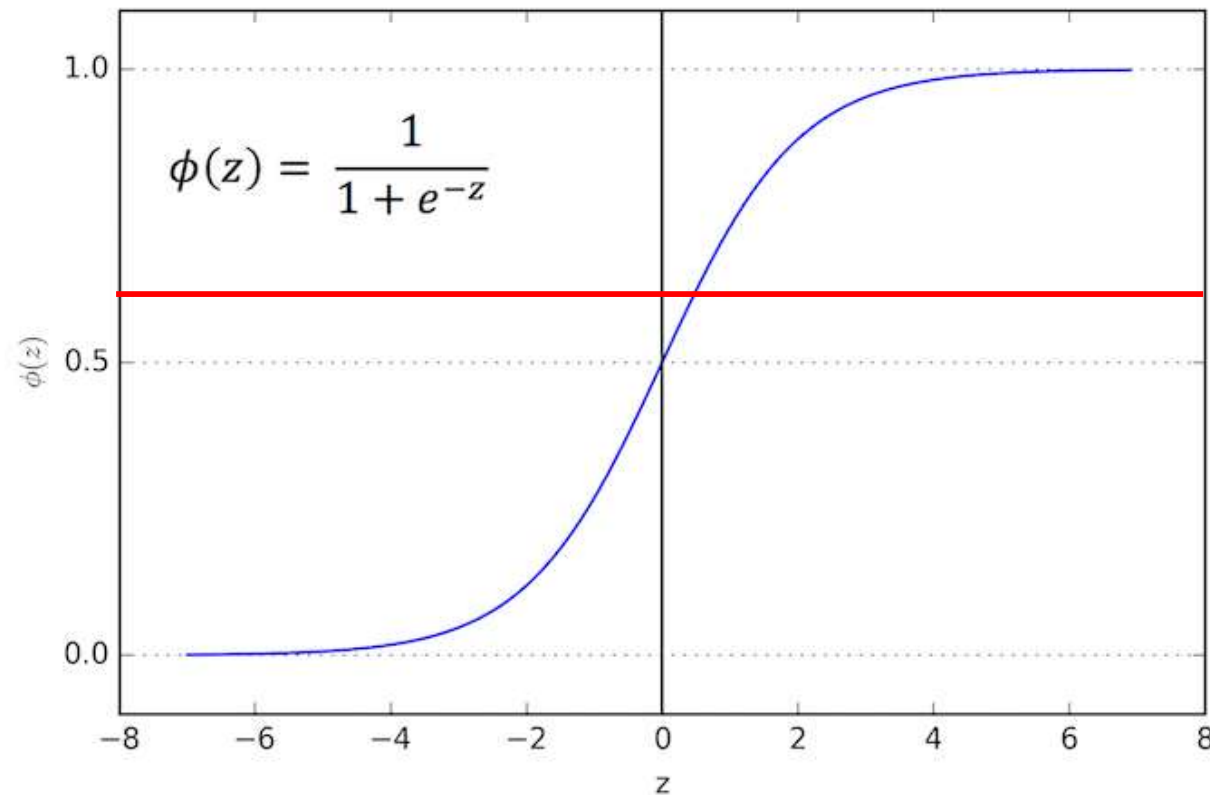
Sigmoid Function

- We can set a cutoff point at 0.5, anything below it results in class 0, anything above is class 1.



Review

- We use the logistic function to output a value ranging from 0 to 1. Based off of this probability we assign a class.



Model Evaluation

- After you train a logistic regression model on some training data, you will evaluate your model's performance on some test data.
- You can use a confusion matrix to evaluate classification models.

Confusion Matrix

		predicted condition	
total population		prediction positive	prediction negative
true condition	condition positive	True Positive (TP)	False Negative (FN) (type II error)
	condition negative	False Positive (FP) (Type I error)	True Negative (TN)

Confusion Matrix

		predicted condition		
total population		prediction positive	prediction negative	Prevalence = $\frac{\Sigma \text{ condition positive}}{\Sigma \text{ total population}}$
true condition	condition positive	True Positive (TP)	False Negative (FN) (type II error)	True Positive Rate (TPR), Sensitivity, Recall, Probability of Detection = $\frac{\Sigma \text{ TP}}{\Sigma \text{ condition positive}}$
	condition negative	False Positive (FP) (Type I error)	True Negative (TN)	False Positive Rate (FPR), Fall-out, Probability of False Alarm = $\frac{\Sigma \text{ FP}}{\Sigma \text{ condition negative}}$
		Positive Predictive Value (PPV), Precision = $\frac{\Sigma \text{ TP}}{\Sigma \text{ prediction positive}}$	False Omission Rate (FOR) = $\frac{\Sigma \text{ FN}}{\Sigma \text{ prediction negative}}$	Positive Likelihood Ratio (LR+) = $\frac{\text{TPR}}{\text{FPR}}$
		False Discovery Rate (FDR) = $\frac{\Sigma \text{ FP}}{\Sigma \text{ prediction positive}}$	Negative Predictive Value (NPV) = $\frac{\Sigma \text{ TN}}{\Sigma \text{ prediction negative}}$	Negative Likelihood Ratio (LR-) = $\frac{\text{FNR}}{\text{TNR}}$
Accuracy = $\frac{\Sigma \text{ TP} + \Sigma \text{ TN}}{\Sigma \text{ total population}}$				

Model Evaluation



The main point to remember with the confusion matrix and the various calculated metrics is that they are all fundamentally ways of comparing the predicted values versus the true values.



What constitutes “good” metrics, will really depend on the specific situation!

Model Evaluation

- We can use a confusion matrix to evaluate our model.
- For example, imagine testing for disease.

n=165	Predicted: NO	Predicted: YES
	Actual: NO	Actual: YES
	50	10
	5	100

Example: Test for presence of disease
NO = negative test = False = 0
YES = positive test = True = 1

Confusion Matrix

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Basic Terminology:

- True Positives (TP)
- True Negatives (TN)
- False Positives (FP)
- False Negatives (FN)

Confusion Matrix

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Accuracy:

- Overall, how often is it **correct**?
- $(TP + TN) / \text{total} = 150/165 = 0.91$

Confusion Matrix

n=165	Predicted: NO	Predicted: YES	
Actual: NO	TN = 50	FP = 10	60
Actual: YES	FN = 5	TP = 100	105
	55	110	

Misclassification Rate
(Error Rate):

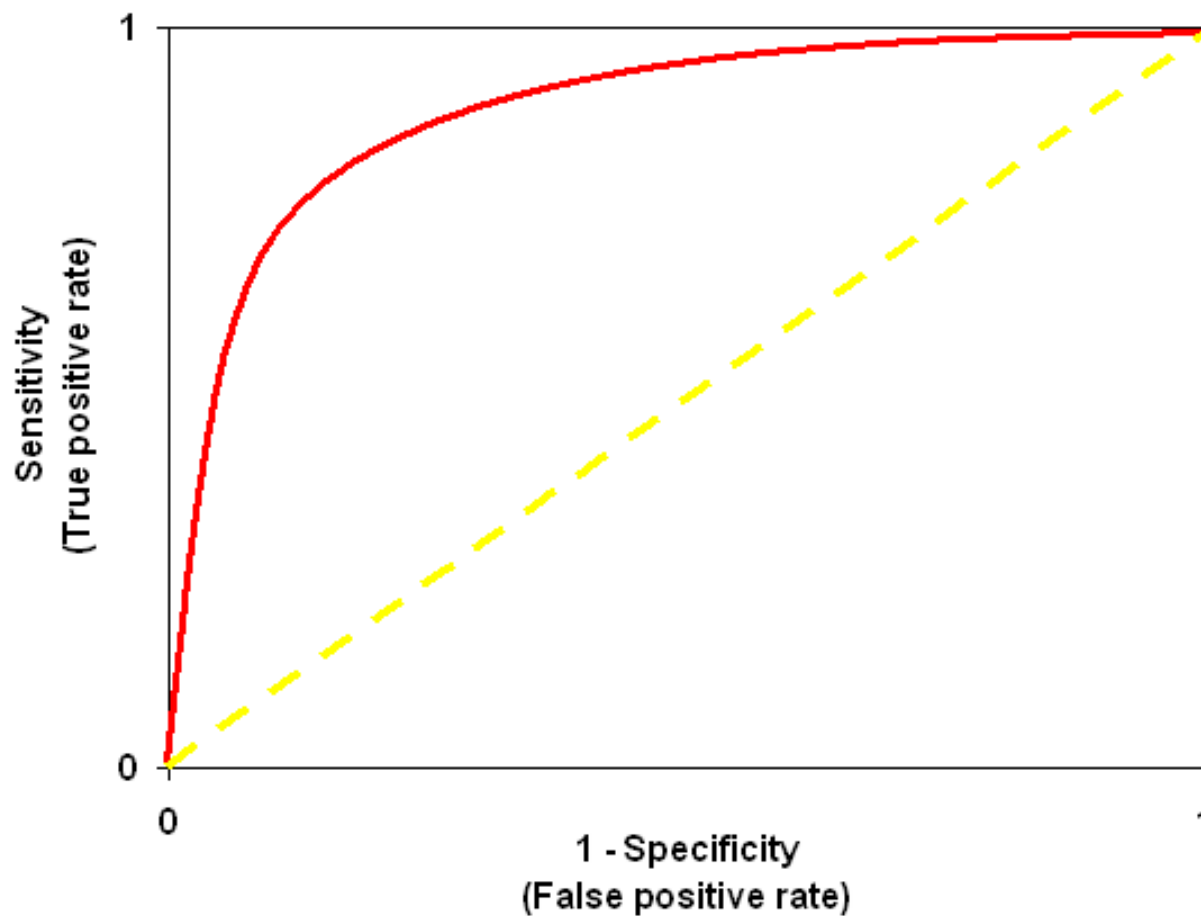
- Overall, how often is it **wrong**?
- $(FP + FN) / \text{total} = 15/165 = 0.09$

Model Evaluation

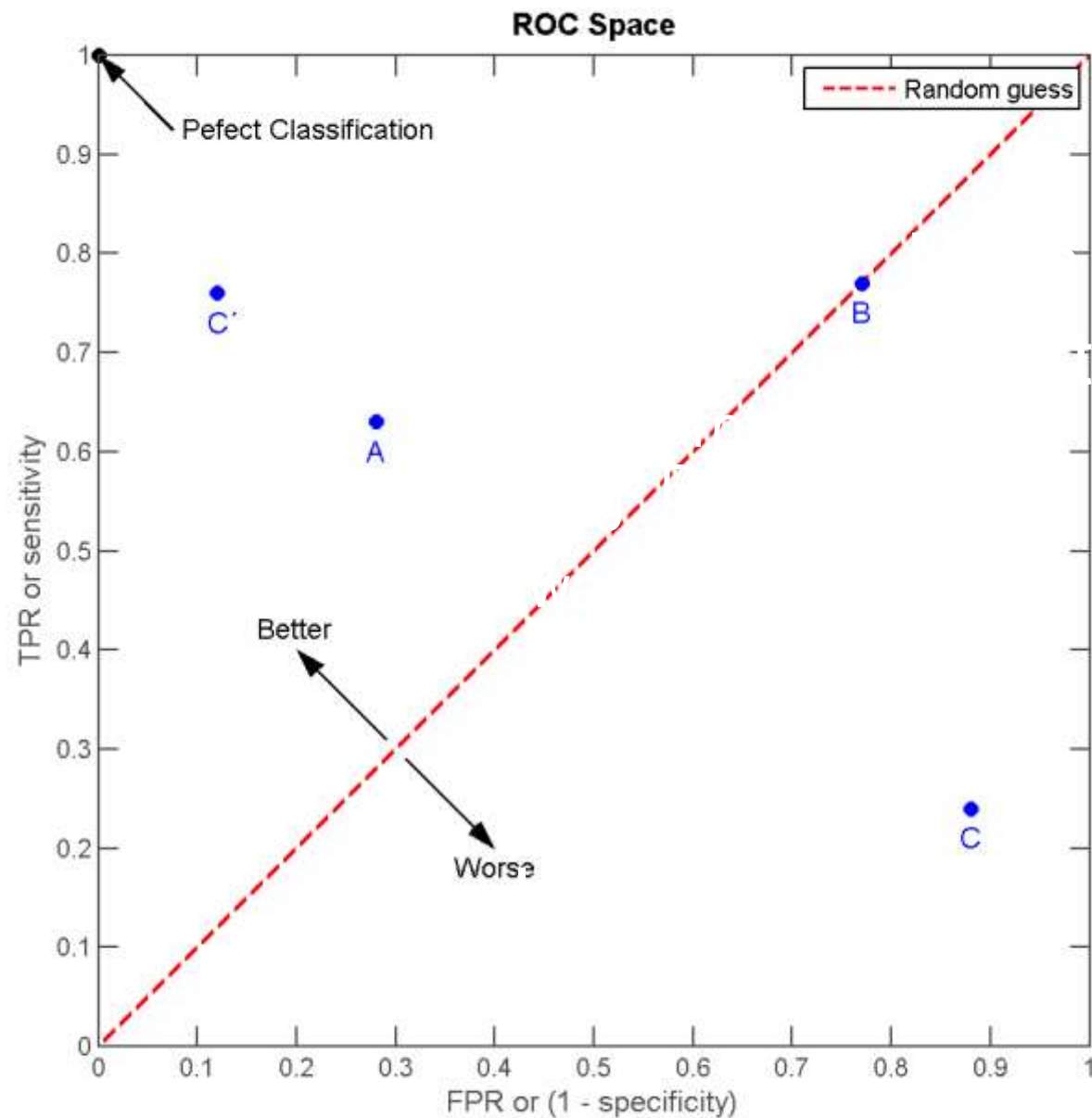
- Binary classification has some of its own special classification metrics.
- These include visualizations of metrics from the confusion matrix.
- The Receiver Operator Curve (ROC) curve was developed during World War II to help analyze radar data.

Model Evaluation

- The ROC curve:



Model Evaluation



Predicting House Prices

- We'll use the **California Housing Dataset**, which contains various features like crime rate, average number of rooms per dwelling, and more to predict house prices.



Import Libraries & Load and Explore the Dataset

```
# Import necessary libraries

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import
train_test_split

from sklearn.linear_model import
LinearRegression

from sklearn.metrics import
mean_squared_error, r2_score

from sklearn.datasets import
fetch_california_housing

# Load the California Housing dataset

housing = fetch_california_housing()

# Convert to pandas DataFrame for easier
manipulation

data = pd.DataFrame(housing.data,
columns=housing.feature_names)

data['PRICE'] = housing.target
```

```
# Display the first 5 rows of the dataset

print("First 5 rows of the dataset:")

print(data.head())

# Select features and target

X = data[['MedInc']] # Independent
variable (Median Income)

y = data['PRICE'] # Dependent variable
(House price)
```

Features Explanation:

- CRIM:** Crime rate per capita.
- RM:** Average number of rooms per dwelling.
- LSTAT:** Percentage of lower status population.
- PRICE:** Median value of homes in \$1000s.

Import Libraries & Load and Explore the Dataset

```
# Split the data into training and testing sets (80% training, 20% testing)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,  
random_state=42)
```

```
# Initialize and train the Linear Regression model
```

```
model = LinearRegression()
```

```
model.fit(X_train, y_train)
```

```
# Predict house prices for the test set
```

```
y_pred = model.predict(X_test)
```

```
# Evaluate the model
```

```
mse = mean_squared_error(y_test, y_pred)
```

```
r2 = r2_score(y_test, y_pred)
```

```
print(f'\nMean Squared Error: {mse:.2f}')
```

```
print(f'R-squared: {r2:.2f}')
```

First 5 rows of the dataset:

	MedInc	HouseAge	AveRooms	...
	Latitude	Longitude	PRICE	
0	8.3252	41.0	6.984127	...
	37.88	-122.23	4.526	
1	8.3014	21.0	6.238137	...
	37.86	-122.22	3.585	
2	7.2574	52.0	8.288136	...
	37.85	-122.24	3.521	
3	5.6431	52.0	5.817352	...
	37.85	-122.25	3.413	
4	3.8462	52.0	6.281853	...
	37.85	-122.25	3.422	

Import Libraries & Load and Explore the Dataset

```
# Visualize the results
plt.figure(figsize=(8, 6))

plt.scatter(X_test, y_test,
            color='blue', label='Actual Prices')

plt.plot(X_test, y_pred,
         color='red', linewidth=2,
         label='Predicted Prices')

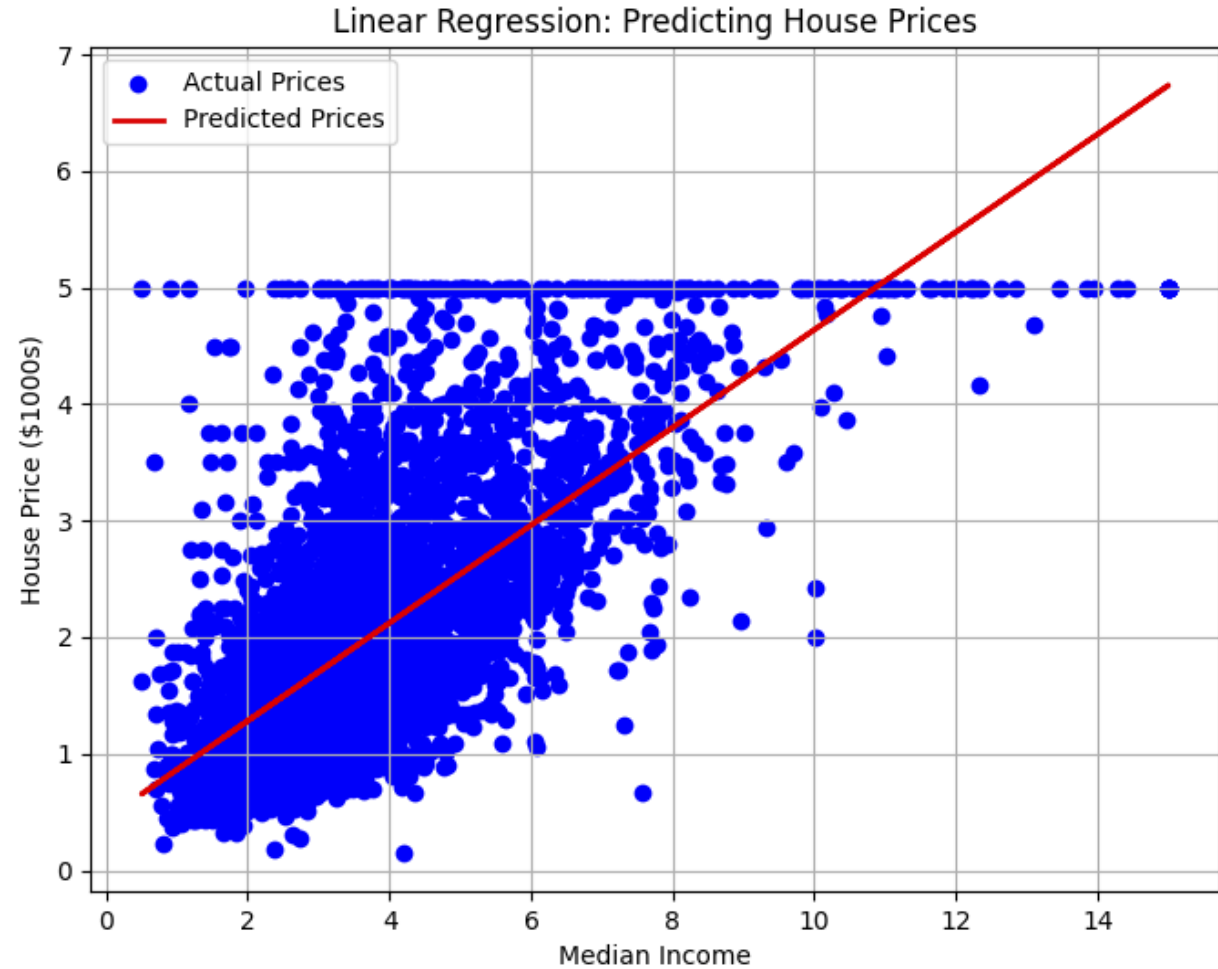
plt.xlabel('Median Income')
plt.ylabel('House Price ($1000s)')

plt.title('Linear Regression:
Predicting House Prices')

plt.legend()

plt.grid(True)

plt.show()
```





Mean Squared Error (MSE)

measures the average of the squares of the errors between predicted and actual values. Lower values indicate better fit.



R-squared

measures how well the independent variable(s) explain the variance in the dependent variable.

It ranges from 0 to 1, where 1 indicates a perfect fit. In this example, R-squared = 0.48 means the model explains 48% of the variance in house prices.

ROC Curves (Receiver Operating Characteristic Curves)

ROC curves

- used to evaluate the performance of binary classification models by plotting the True Positive Rate (TPR) against the False Positive Rate (FPR) at various

Use Case: ROC curves are commonly used when there is a class imbalance. A higher AUC means the model is better at distinguishing between positive and negative classes.

True Positive Rate (TPR):

- Also known as sensitivity or recall, it's the ratio of correctly predicted positive observations

$$TPR = \frac{TP}{TP + FN}$$

False Positive Rate (FPR):

- The ratio of incorrectly predicted positive observations to all actual negatives.

$$FPR = \frac{FP}{FP + TN}$$

AUC (Area Under the Curve):

- Measures the overall performance of the model. An AUC of 1 indicates a perfect model, while an AUC of 0.5 represents random guessing.

Breast Cancer Wisconsin dataset

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import
load_breast_cancer

from sklearn.model_selection import
train_test_split

from sklearn.preprocessing import
StandardScaler

from sklearn.linear_model import
LogisticRegression

from sklearn.metrics import roc_curve,
roc_auc_score

# Load the Breast Cancer dataset

data = load_breast_cancer()

X = data.data

y = data.target

print(f"Feature names: {data.feature_names}")

print(f"Number of samples: {X.shape[0]}")

print(f"Number of features: {X.shape[1]}")
```

```
# Split the data into training and testing sets (70%
train, 30% test)

X_train, X_test, y_train, y_test =
train_test_split(

    X, y, test_size=0.3, random_state=42,
    stratify=y

)

# Standardize the features

scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)

# Initialize and train the Logistic Regression model

model =
LogisticRegression(solver='liblinear',
random_state=42)

model.fit(X_train, y_train)

Feature names: ['mean radius' 'mean texture'
'mean perimeter' ... 'worst fractal
dimension']
Number of samples: 569
```

Breast Cancer Wisconsin dataset

```
# Predict probabilities for the test set
y_scores = model.predict_proba(X_test)[: , 1] # Probability of the
positive class

# Compute ROC curve metrics
fpr, tpr, thresholds = roc_curve(y_test, y_scores)

# Compute AUC (Area Under the Curve)
auc_score = roc_auc_score(y_test, y_scores)

print(f"AUC Score: {auc_score:.2f}")
```

AUC Score: 0.99

Plot the ROC Curve

```
plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='blue',
label=f'ROC Curve (AUC = {auc_score:.2f})')

plt.plot([0, 1], [0, 1], color='red',
linestyle='--', label='Random Guessing')

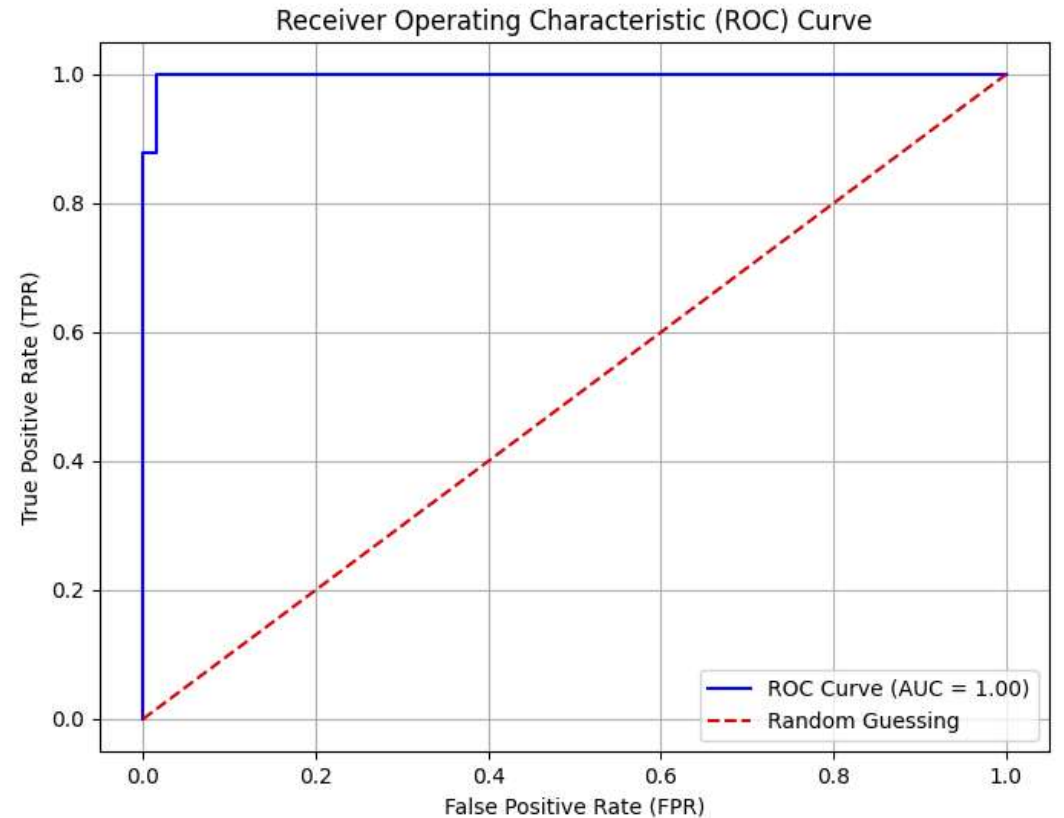
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')

plt.title('Receiver Operating Characteristic (ROC) Curve')

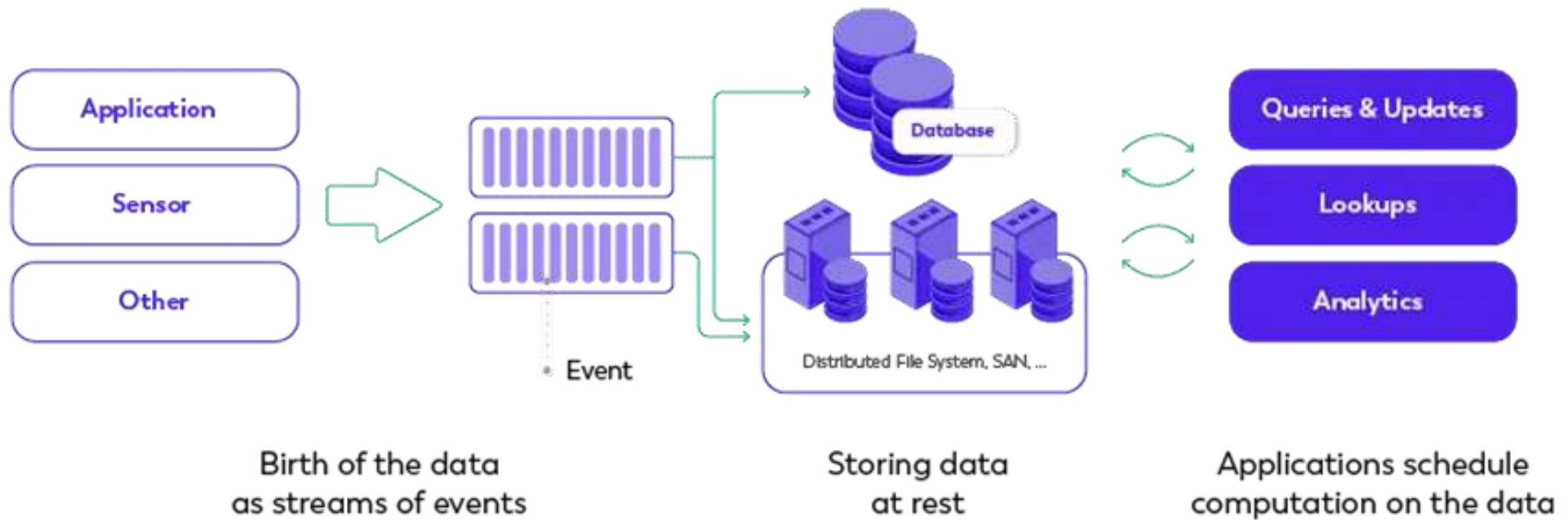
plt.legend(loc='lower right')

plt.grid(True)

plt.show()
```



STREAM PROCESSING



Real-time stream processing



the continuous processing of data as it is generated, enabling immediate analysis and decision-making.



Instead of waiting for data to accumulate and then processing it in batches (as with traditional batch processing),



stream processing handles events as they arrive, providing insights, responses, or transformations instantly or within milliseconds to seconds.

Key Characteristics



Continuous Data Flow:

Data is processed as soon as it arrives, typically in an unbounded and continuous stream.

It's ideal for environments where data is generated constantly, such as sensor networks, financial markets, or



Low Latency:

there is minimal delay between data ingestion and the processing of that data.

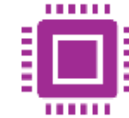
This allows for immediate insights and actions.



Event-Driven:

Data in real-time stream processing is typically in the form of discrete events, such as user clicks, financial transactions, or sensor readings.

Systems respond to these events almost instantly, triggering



Stateful Processing:

Stream processors often maintain a state across different events.

counting occurrences of events, tracking patterns over time, or joining streams of data.

A system like **Apache Flink** is



Windowing:

In stream processing, data is often aggregated over windows of time (e.g., every 5 minutes) or a certain number of events (e.g., after every 100 events).

This allows for meaningful analysis and summarization of the

Use Cases of Real-Time Stream Processing



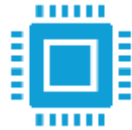
Fraud Detection:

real-time stream processing can detect suspicious transactions instantly and flag or block them before harm is done.



Real-Time Analytics:

For online platforms, clickstream analysis in real-time helps provide instant insights into user behavior, enabling dynamic content adjustments or recommendations.



Internet of Things (IoT):

Devices and sensors continuously generate data. Real-time processing allows for immediate actions, such as shutting down a machine in case of overheating, or predictive maintenance based on continuous



Financial Market

Stream Analysis:

processing is critical in stock exchanges, where real-time data processing is necessary to monitor market movements and execute trades based on live data.



Real-Time Recommendations

- Services like Netflix and Spotify use real-time stream processing to analyze user interactions and provide personalized recommendations immediately.

Technologies Used for Real-Time Stream Processing

Apache Flink:

- Optimized for real-time, stateful stream processing with low-latency and fault tolerance features.

Apache Kafka:

- Often used as a distributed message broker, handling data streaming between producers and consumers.

Apache Storm:

- Another real-time stream processing framework, designed for distributed and fault-tolerant processing.

Apache Spark Structured Streaming:

- A micro-batch processing framework for near real-time analytics, which offers a simpler approach to streaming.

Telecom Churn Prediction

Scenario:

- A telecom company wants to predict customer churn in real-time to reduce attrition.
- They implement a real-time churn prediction system based on customer usage patterns and interactions with support.

Data Source:

- Stream customer call logs, data usage, service complaints, and billing information through **Kafka**.

Flink for Stream Processing:

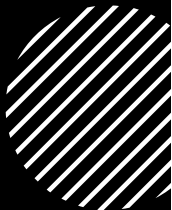


- Use **Flink** to preprocess the data, calculate real-time metrics (e.g., call duration, internet usage over the past 24 hours), and detect patterns (e.g., customers frequently contacting support).
- Real-time features such as "days since last payment" or "increase in service complaints" are

Churn Prediction Model:

- A pre-trained **logistic regression** model is integrated into the Flink pipeline to predict the

Real-Time Action:

- If the churn probability is high, the system triggers an SMS offering a discount or enhanced service. If the customer is still dissatisfied, it escalates to a human support team for intervention.



Churn Predictio n



Churn prediction is the process of identifying customers who are likely to stop using a service or product.



In real-time, this involves processing and analyzing customer data as it is generated, allowing companies to intervene promptly and reduce customer attrition.



Real-time churn prediction enables businesses, such as telecom companies, online platforms, and subscription services, to take immediate action, such as offering incentives or sending notifications to at-risk customers.

Telecom company wants to predict customer churn



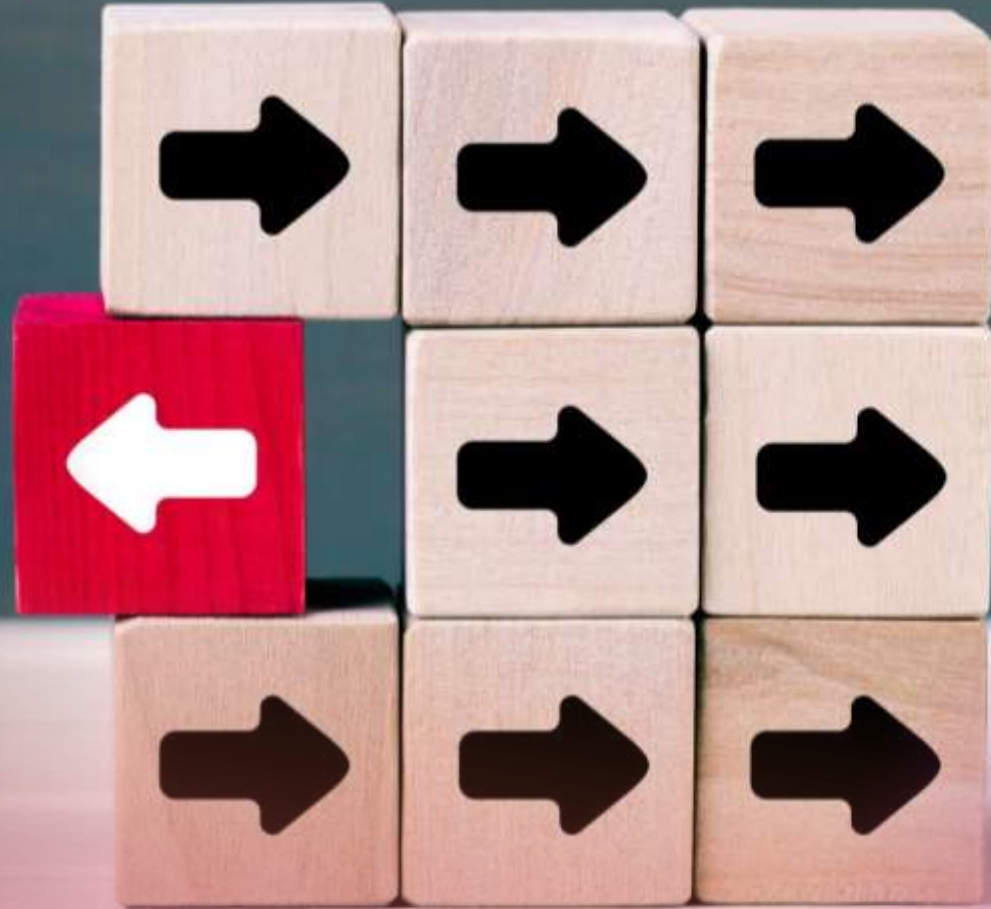
The telecom company wants to prevent churn by analyzing real-time customer data, such as:

- Call durations.
- Data usage.
- Interaction with customer support.
- Billing and payment history.
- Network issues or service complaints.



The goal is to use this data to predict which customers are likely to stop using the service soon and intervene with offers, discounts, or personalized communication before they leave.

Step-by-Step
Real-Time
Churn
Prediction
Process



Data Collection (Real-Time Ingestion)

Data Sources: The telecom company continuously collects real-time data from multiple sources:

Call logs (e.g., dropped calls, call durations).

Data usage patterns (e.g., internet usage, session time).

Support interactions (e.g., complaints, service requests).

Billing information (e.g., late payments, subscription renewals).



Tool: Apache Kafka is used to stream this real-time data into the processing pipeline.



Example: A customer calls telecom support several times in a short period due to network issues, raising the probability of churn.


Preprocessing and Feature Engineering

Data from Kafka is consumed by **Apache Flink** for real-time processing.



Flink cleans and transforms the data and generates relevant features, such as:

Time since last call.	Number of dropped calls in the last week.	Total data consumption in the last month.	Frequency of customer support interactions.	Payment delays or missed payments.
-----------------------	---	---	---	------------------------------------



Example: A feature like "increased support calls in the last 7 days" or "dropped calls in the last 24 hours" is calculated for every customer.

Churn Prediction Model (Real-Time Scoring)

Machine Learning Model:

- A pre-trained machine learning model (e.g., logistic regression, random forest, or deep learning) is used to predict the churn probability.
- This model is trained on historical data of customer behavior, past churn events, and retention outcomes.

Real-Time Model Scoring:

- As data is processed in real-time, the features generated by Flink are passed to the model to predict whether a customer is likely to churn.
 - **Tools:** The model can be deployed using **Flink ML** or served using **TensorFlow Serving** for real-time predictions.

Example:

- The system computes a churn probability of 85% for a customer who has reported multiple network issues, decreased their data usage, and had delayed payments in the last month.

Real-Time Actions and Interventions

Intervention Strategy: If the churn probability exceeds a threshold (e.g., 80%), the system triggers an action in real-time:

Sending personalized offers (e.g., discounts, data packages).

Escalating to customer service for a follow-up call.

Sending retention emails or SMS messages.



Tools: Integration with CRM systems, email/SMS services, or automated chatbots for direct communication.



Example: If the model predicts that a customer is at high risk of churning, the system might immediately send an SMS offering a 10% discount on the next month's bill or provide additional data usage for free.

Continuous Monitoring and Feedback Loop

Monitoring Customer Behavior:

- After the intervention, customer behavior is monitored in real-time to determine whether the churn prediction was accurate or if the intervention was effective.
- If the customer continues to use the service or accepts the offer, this positive outcome is tracked.
- If the customer still shows signs of churn, further actions may be taken, such as direct outreach by the customer support team.

Feedback and Model Retraining:

- The outcomes of interventions (e.g., customer retention or churn) are fed back into the machine learning pipeline to improve the model.
- Periodically, the model is retrained using updated data to reflect the latest customer behavior trends.

Example:

- If the customer accepts the discount offer and continues using the service, the system tracks this behavior and adjusts the churn model to recognize the effectiveness of the discount in preventing churn.

Real-Time Churn Prediction Pipeline (Using Apache Kafka and Apache Flink)



Data Stream Ingestion:

Kafka streams customer activity data in real-time to the processing pipeline.



Data Preprocessing and Feature Engineering:

Apache Flink consumes the Kafka stream, applies transformations, cleans data, and computes real-time features (e.g., "time since last purchase").



Model Scoring:

The real-time features are passed to a pre-trained model (deployed using TensorFlow Serving or Flink ML), which predicts the churn probability.



Real-Time Intervention:

If the churn probability exceeds a certain threshold (e.g., 80%), the system triggers immediate actions, such as sending offers through a CRM system or escalating customer support interactions.



Continuous Monitoring and Feedback:

The pipeline is monitored continuously, and feedback (e.g., whether the customer accepted the offer or stayed) is used to retrain the model periodically.

Customer Lifetime Value (CLV) Estimation



Customer Lifetime Value (CLV) is a metric that estimates the total revenue a business can reasonably expect from a customer throughout their relationship.



It helps companies understand the long-term value of their customers, guiding decisions on marketing, sales, and customer retention strategies.



Accurate CLV estimation allows businesses to optimize customer acquisition costs and focus on retaining the most valuable customers.

Components of CLV



Average Purchase Value (APV) :

The average amount a customer spends on a purchase.



Purchase Frequency (PF) :

The number of purchases a customer makes over a specific period.



Customer Lifespan (CL) :

The length of time a customer continues to make purchases from the business.



Gross Margin (GM) :

The percentage of revenue retained after deducting the cost of goods sold (COGS).

CLV Formula



CLV can be estimated using the formula:

$$CLV = APV \times PF \times CL \times GM$$



However, **real-time CLV** estimation considers dynamic customer behavior, and data is constantly updated as new customer interactions occur.

E-Commerce CLV Estimation

Scenario:

- A large e-commerce company wants to estimate the real-time CLV of its customers to personalize marketing strategies and optimize spending on customer acquisition.

Data Sources:

- The company collects real-time data from its website, including transaction history, browsing behavior, and customer interactions with promotional

Data Processing:

- Using **Apache Kafka** and **Flink**, the company streams customer transaction data and calculates real-time metrics like purchase frequency and average order value.

E-Commerce CLV Estimation



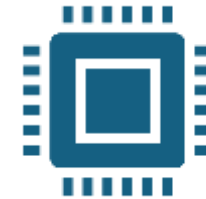
CLV Model:

A pre-trained regression model is deployed using **TensorFlow Serving**. The model estimates CLV based on customer purchase history, browsing patterns, and responses to promotions.



Personalized Marketing:

Customers with high CLV are offered loyalty rewards or special discounts, while those with lower CLV receive personalized re-engagement offers via email or SMS.



Real-Time Adjustment:

The CLV model continuously updates based on new customer data, allowing the company to adjust its marketing strategies dynamically.

Benefits of Real-Time CLV Estimation

- 1. Personalized Marketing:** Businesses can tailor marketing campaigns to individual customers, maximizing revenue potential.
- 2. Improved Resource Allocation:** By understanding which customers are most valuable, companies can focus resources on retention and growth for those segments.
- 3. Optimized Acquisition:** Companies can adjust customer acquisition strategies, focusing on acquiring high-CLV customers.
- 4. Increased Retention:** Personalized offers based on CLV predictions help retain valuable customers and reduce churn.
- 5. Business Growth:** CLV estimation improves decision-making, enhancing profitability and long-term business growth.



Apache Flink



What is Apache Flink?

open-source, distributed engine for stateful processing over unbounded (streams) and bounded (batches) data sets.

Stream processing applications are designed to run continuously, with minimal downtime, and process data as it is ingested.

Apache Flink is designed for low latency processing, performing computations in-memory, for high availability, removing single point of failures, and to scale horizontally.

Apache Flink's features include advanced state management with exactly-once consistency guarantees, event-time processing semantics with sophisticated out-of-order and late data handling.

Apache Flink has been developed for streaming-first, and offers a unified programming interface for both stream and batch processing.

Use of Apache Flink

Event-driven applications

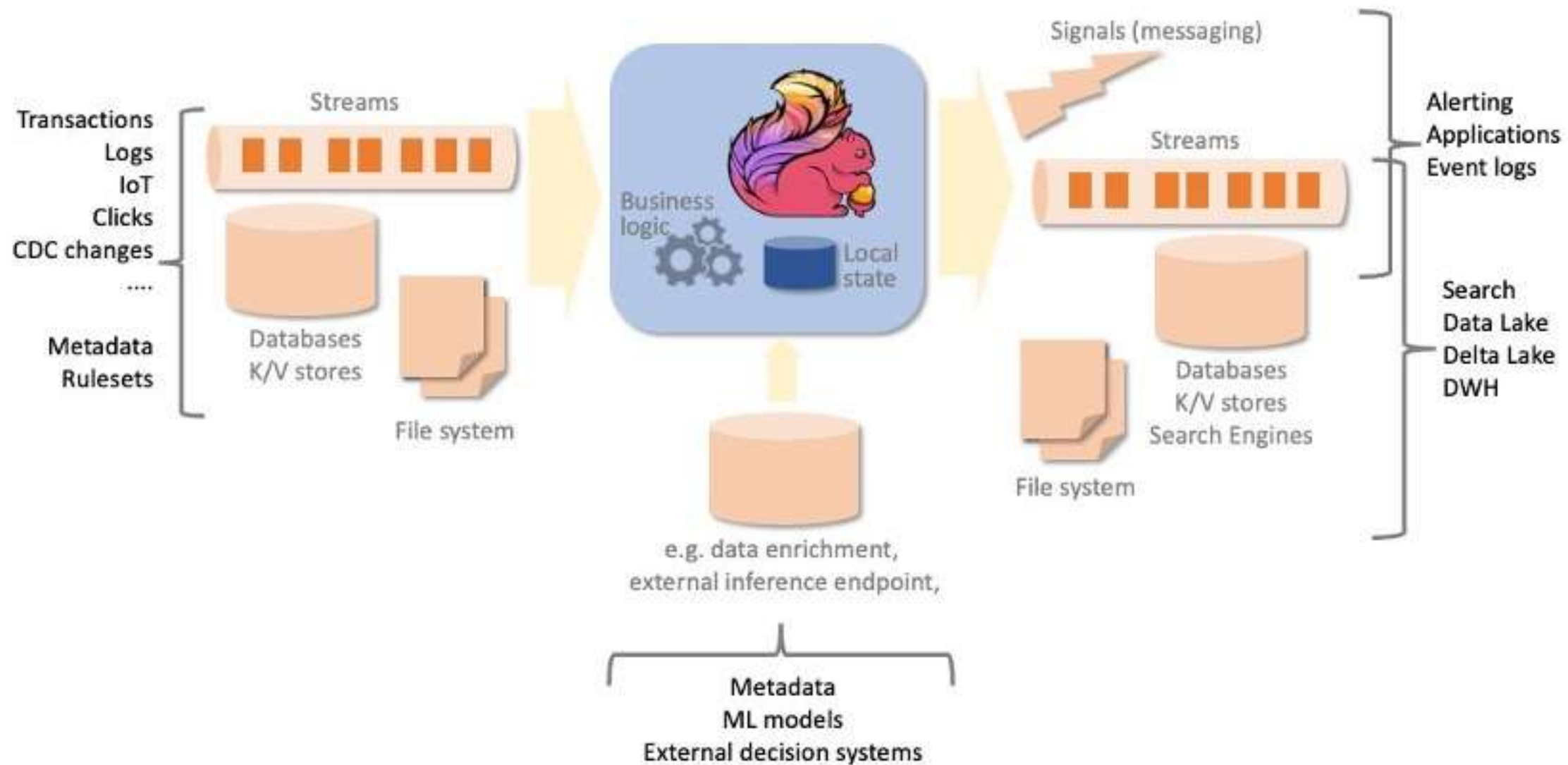
- ingesting events from one or more event streams and executing computations, state updates or external actions.
- Stateful processing allows implementing logic beyond the Single Message Transformation, where the results depend on the history of ingested events.

Data Analytics applications

- extracting information and insights from data.
- Traditionally executed by querying finite data sets, and re-running the queries or amending the results to incorporate new data.
- With Apache Flink, the analysis can be executed by continuously updating, streaming queries or processing ingested events in real-time, continuously emitting and updating the results.

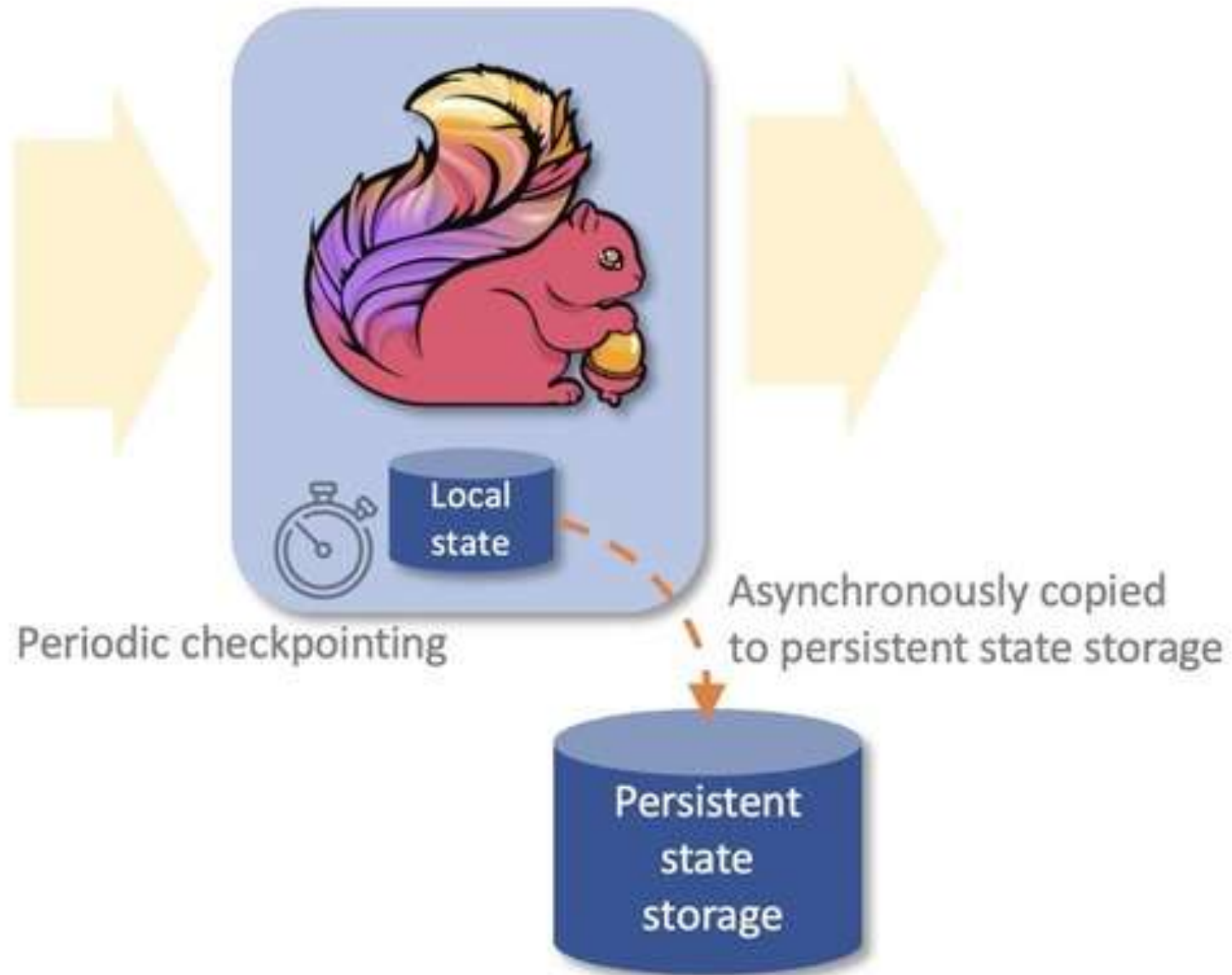
Data pipelines applications

- transforming and enriching data to be moved from one data storage to another.
- Traditionally, extract-transform-load (ETL) is executed periodically, in batches.
- With Apache Flink, the process can operate continuously, moving the data with low latency to their destination.



How does Apache Flink work?

- Flink is a high throughput, low latency stream processing engine.
- A Flink application consists of an arbitrary complex acyclic dataflow graph, composed of streams and transformations.
- Data is ingested from one or more data sources and sent to one or more destinations.
- Source and destination systems can be streams, message queues, or datastores, and include files, popular database and search engines.
- Transformations can be stateful, like aggregations over time windows or complex pattern detection.
- Fault tolerance is achieved by two separate mechanisms:
 - automatic and periodic checkpointing of the application state, copied to a persistent storage,
 - to allow automatic recovery in case of failure;
 - on-demand savepoints, saving a consistent image of the execution state, to allow stop-and-resume, update or fork your Flink job, retaining the application state across stops and restarts.
 - Checkpoint and savepoint mechanisms are asynchronous, taking a consistent snapshot of the state without “stopping the world”, while the application keeps processing events.



What are the benefits of Apache Flink?

Process both unbounded (streams) and bounded (batches) data sets

- Apache Flink can process both unbounded and bounded data sets, i.e., streams and batch data.
- Unbounded streams have a start but are virtually infinite and never end.
- Processing can theoretically never stop.
- Bounded data, like tables, are finite and can be processed from the beginning to the end in a finite time.
- Apache Flink provides algorithms and data structures to support both bounded and unbounded processing through the same programming interface.
- Applications processing bounded data will end their execution when reaching

Run applications at scale

- Apache Flink is designed to run stateful applications at virtually any scale. Processing is parallelized to thousands of tasks, distributed multiple machines, concurrently.
- State is also partitioned and distributed horizontally, allowing to maintain several terabytes across multiple machines.
- State is checkpointed to a persistent storage incrementally.

What are the benefits of Apache Flink?

In-memory performance

- Data flowing through the application and state are partitioned across multiple machines.

Exactly-once state consistency

- Applications beyond single message transformations are stateful.
- The business logic needs to remember events or intermediate results.
- Apache Flink guarantees consistency of the internal state, even in case of failure and across application stop and restart.
- The effect of each message on the internal state is always applied exactly-once, regardless the application may receive duplicates from the data source on recovery or

Wide range of connectors

- Apache Flink has a number of proven connectors to popular messaging and streaming systems, data stores, search engines, and file system.
- Some examples are Apache Kafka, Amazon Kinesis Data Streams, Amazon SQS, Active MQ, Rabbit MQ, NiFi, OpenSearch and Elasticsearch, DynamoDB, HBase, and any database providing JDBC client.



Apache Kafka



elasticsearch



Amazon
DynamoDB



Amazon S3



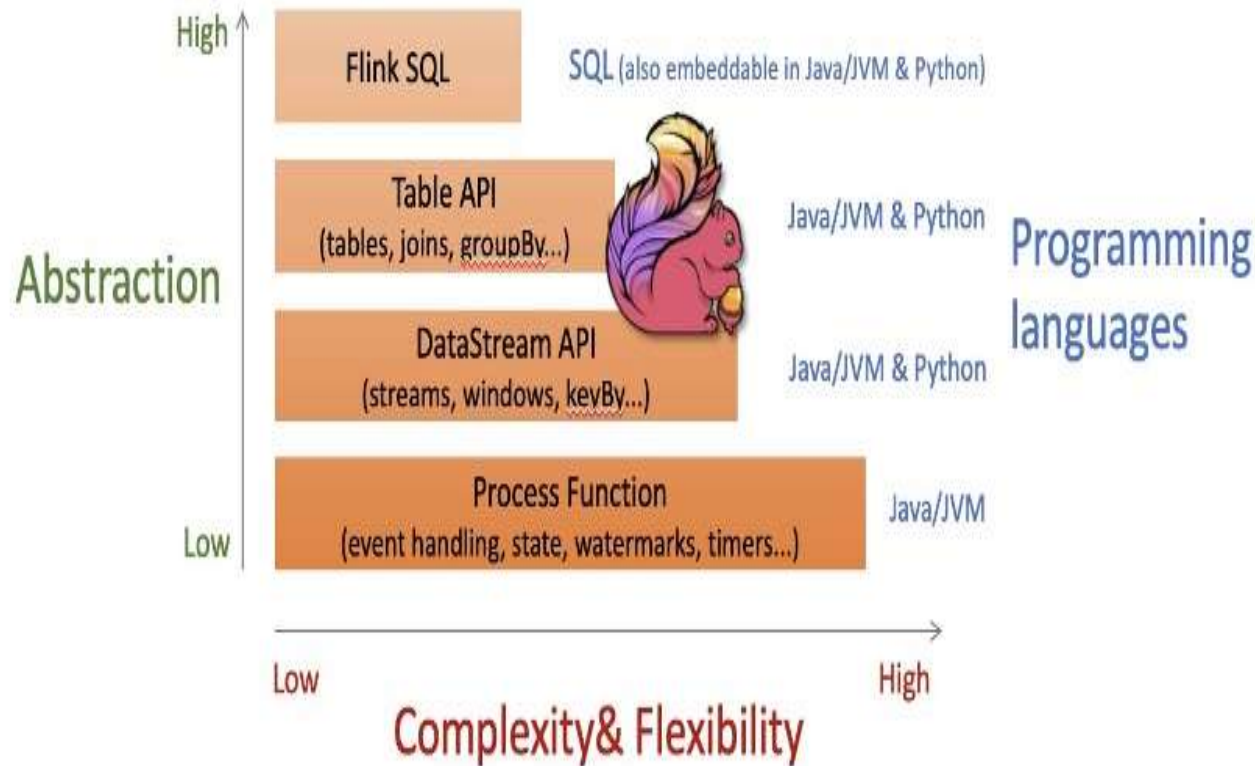
Apache Cassandra



Amazon Kinesis
Data Streams



Multiple levels of abstractions



- Apache Flink offers multiple level of abstraction for the programming interface.
- From higher level streaming SQL and Table API, using familiar abstractions like table, joins and group by.
- The DataStream API offers a lower level of abstraction but also more control, with the semantics of streams, windowing and mapping.
- And finally, the ProcessFunction API offers fine control on the processing of each message and direct control of the state.
- All programming interfaces work seamlessly with both unbounded (streams) and bounded (tables) date sets.
- Different levels of abstractions can be used in the same application, as the right tool to solve each problem.



NortonLifeLock™



POSHMARK