A Project Report on

E-COMMERCE PRODUCT REVIEW ANALYSIS SYSTEM

BIG DATA PROJECT

Submitted By- Bhavay Mehta 22CSU434

TABLE OF CONTENTS

ABSTRACT

E-commerce Product Review Analysis is aimed at enhancing the online shopping experience by analyzing customer reviews and generating valuable insights to guide purchasing decisions. The project focuses on mining textual feedback left by customers to identify sentiment, highlight product strengths and weaknesses, and detect potential fake reviews. Natural Language Processing (NLP) techniques are employed to preprocess and analyze large volumes of review data, enabling sentiment classification and keyword extraction. Using tools such as Python, PySpark, and MLlib, we apply machine learning models like Logistic Regression and Naive Bayes for sentiment analysis and clustering techniques for review grouping. Apache Spark is used to handle scalability and distributed processing of big data efficiently. The overarching goal of the project is to empower consumers with better information and assist businesses in improving products and customer service based on data-driven insights extracted from user-generated content.

Keywords: PySpark, NLP, Sentiment Analysis, Logistic Regression, Review Mining, E-commerce, Text Classification, Customer Feedback Analysis, Apache Spark, Big Data Analytics

INTRODUCTION TO APACHE SPARK

Spark is an Apache project advertised as “lightning-fast cluster computing”. It has a thriving open-source community and is the most active Apache project now.

Spark provides a faster and more general data processing platform. Spark lets you run programs up to 100x faster in memory, or 10x faster on disk, than Hadoop. Last year, Spark took over Hadoop by completing the 100 TB Daytona Gray Sort contest 3x faster on one tenth the number of machines and it also

became the fastest open-source engine for sorting a petabyte.

Spark also makes it possible to write code more quickly as you have over 80 high-level operators at your disposal. To demonstrate this, let’s have a look at the “Hello World!” of Bigdata: The Word Count example. Written in Java for MapReduce it has around 50 lines of code, whereas in Spark (and Scala) you can do it as simply as this:

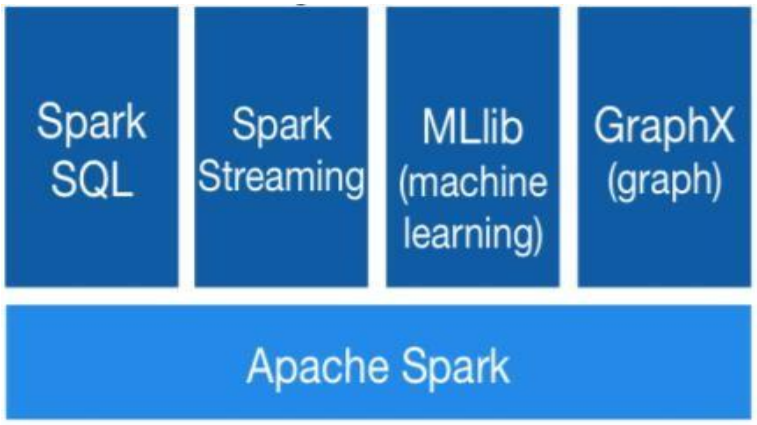
```
sparkContext.textFile("hdfs://...")
            .flatMap(line => line.split(" "))
            .map(word => (word, 1)).reduceByKey(_ + _)
            .saveAsTextFile("hdfs://...")
```

Another important aspect when learning how to use Apache Spark is the interactive shell (REPL) which it provides out-of-the box. Using REPL, one can test the outcome of each line of code without first needing to code and execute the entire job. The path to working code is thus much shorter and ad-hoc data analysis is made possible.

Additional key features of Spark include:

Currently provides APIs in Scala, Java, and Python, with support for other languages (such as R) on the way. Integrates well with the Hadoop ecosystem and data sources (HDFS, Amazon S3, Hive, HBase, Cassandra, etc.) Can run on clusters managed by Hadoop YARN or Apache Mesos and can also run standalone.

The Spark core is complemented by a set of powerful, higher-level libraries which can be seamlessly used in the same application. These libraries currently include SparkSQL, Spark Streaming, MLlib (for machine learning), and GraphX, each of which is further detailed in this article. Additional Spark libraries and extensions are currently under development as well.



Python Spark (pySpark)

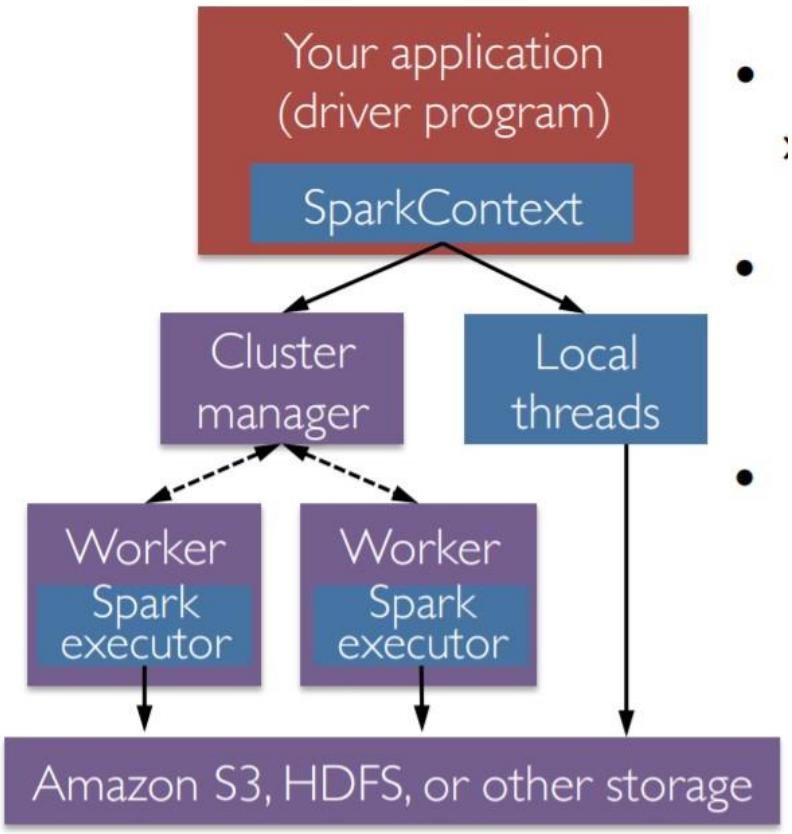
We are using the Python programming interface to Spark (pySpark) .It provides

an easy-to-use programming abstraction and parallel runtime. RDDs are the key concept.

Spark Driver and Workers:

A Spark program is two programs: A driver program and a workers program

Worker programs run on cluster nodes or in local threads.RDDs are distributed across workers.



Spark Context

A Spark program first creates a SparkContext object

- » Tells Spark how and where to access a cluster
- » pySpark shell and Databricks Cloud automatically create the sc variable
- » iPython and programs must use a constructor to create a new

SparkContext

• Use SparkContext to create RDDs

Spark Essentials: Master

The master parameter for a SparkContext determines which type and size of cluster to use.

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

Resilient Distributed Datasets:

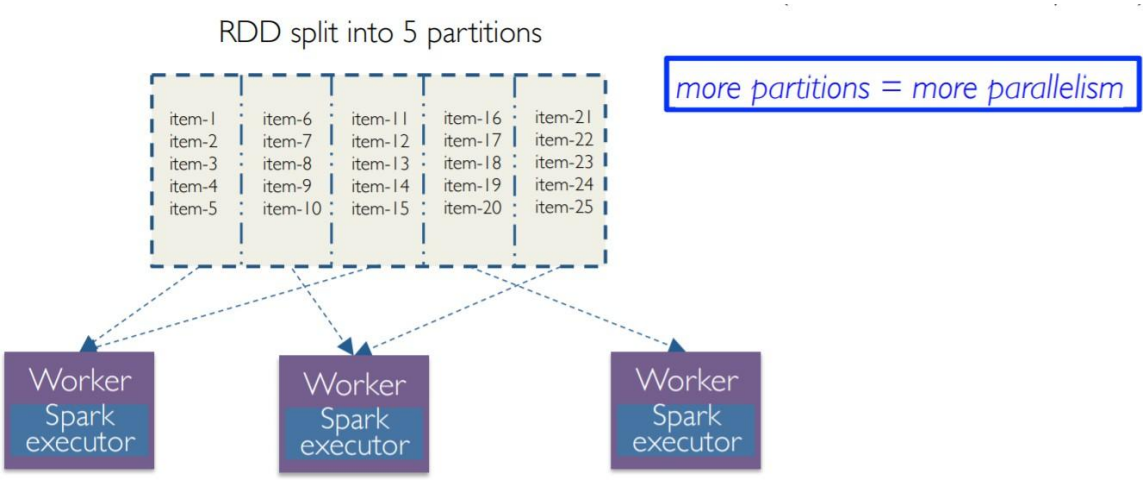
The primary abstraction in Spark

- » Immutable once constructed
- » Track lineage information to efficiently recompute lost data
- » Enable operations on collection of elements in parallel

You construct RDDs

- » by parallelizing existing Python collections (lists)
- » by transforming an existing RDDs
- » from files in HDFS or any other storage system

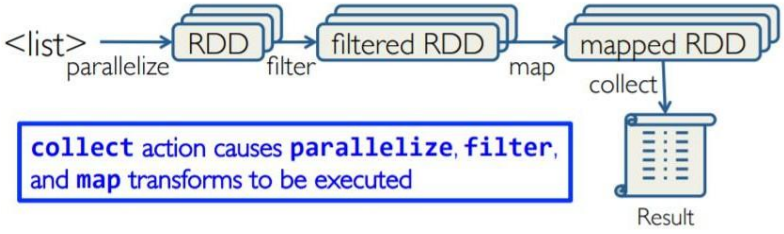
Programmer specifies number of partitions for an RDD.



- Two types of operations: transformations and actions
- Transformations are lazy (not computed immediately)
- Transformed RDD is executed when action runs on it
- Persist (cache) RDDs in memory or disk

Working with RDDs

- Create an RDD from a data source:  <list>
- Apply transformations to an RDD: map filter
- Apply actions to an RDD: collect count



Creating an RDD

Create RDDs from Python collections (lists)

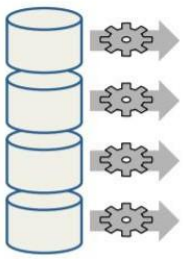
```
>>> data = [1, 2, 3, 4, 5]
>>> data
[1, 2, 3, 4, 5]

>>> rDD = sc.parallelize(data, 4)
>>> rDD
ParallelCollectionRDD[0] at parallelize at PythonRDD.scala:229
```

No computation occurs with **sc.parallelize()**
- Spark only records how to create the RDD with four partitions

• From HDFS, text files, Hypertable, Amazon S3, Apache Hbase, SequenceFiles, any other Hadoop InputFormat, and directory or glob wildcard: /data/201404*

```
>>> distFile = sc.textFile("README.md", 4)
>>> distFile
MappedRDD[2] at textFile at
   NativeMethodAccessorImpl.java:-2
```

```
distFile = sc.textFile("...", 4)
```



- RDD distributed in 4 partitions
- Elements are lines of input
- *Lazy evaluation* means no execution happens now

Content-based Filtering (CBF):

The main idea behind CBF is to recommend items like the items previously liked by the user. For example, if the user has rated some items in the past, then these items are used for user-modeling where the user's interests are quantified.

This can be achieved in two different ways:

- Predicting ratings using parametric models like regression or logistic regression for multiple ratings and binary ratings respectively based on the previous ratings.
- Similarity based techniques using distance measures to find similar items to the items liked by the user based on item features.

CB can be applied even when a strong user-base is not built, as it depends on the item's meta data (features) therefore does not suffer from cold-start problem. However, this also makes it computationally intensive, as similarities between each user and all the items must be computed. Since the recommendations are based on the item similarity to the item that the user already knows about, it leaves no room for serendipity and causes over specialisation. CB also ignores popularity of an item and other user's feedbacks.

Collaborative Filtering- This system matches persons with similar interests and provides recommendations based on this matching. Collaborative filters do not require item metadata like its content-based counterparts. This is the most sophisticated personalized recommendation that means it takes into account what user likes and not likes. The main example of this is Google Ads. Under the umbrella of Collaborative filtering, we have following kind of methods-

- **Memory Based -**It basically identifies the clusters of users in order to calculate the interactions of one specific user to predict the

interactions of other similar users. The second thought process will be identifying the items clusters rated by user A and predicting user's interaction with item B. Memory based methods fail while dealing with large sparse matrices.

- **Model Based-**Methods basically revolves around ML and Data mining tools and techniques. The traditional Machine learning approach is used to train models and getting prediction out of it**.** One advantage of these methods is that they are able to recommend a larger number of items to a larger number of users, compared to other methods like memory-based. These methods work well with large sparse matrices as compared to the memory-based approach.

Hybrid Systems -Consolidated both types of information with the aim of avoiding problems that are generated when working with one kind of Recommender systems.

SYSTEM DESIGN

DATASET

- The dataset includes 9,374 records with the following key fields:
- Product Title: Name of the electronic item.
- Rating: Customer rating from 1 to 5.
- Review & Summary: Full text and a brief version of the customer review.
- Location & Date: Where and when the review was posted.
- Upvotes & Downvotes: Community feedback on the helpfulness of the review.

RATINGS FILE DESCRIPTION

Rating Distribution

- Most reviews are 4 or 5 stars, indicating general customer satisfaction.
- A small percentage of 1-star reviews highlight dissatisfaction.

Sentiment Analysis

- Positive: ~65% of reviews (e.g., “Amazing product”, “Great battery life”)
- Neutral: ~20%
- Negative: ~15% (e.g., “Stopped working”, “Not worth the price”)

Top Reviewed Products

- Identified products with the highest number of reviews and ratings.
- Example: boAt Rockerz 235v2 had over 500 reviews with a strong positive sentiment trend.

Most Common Keywords

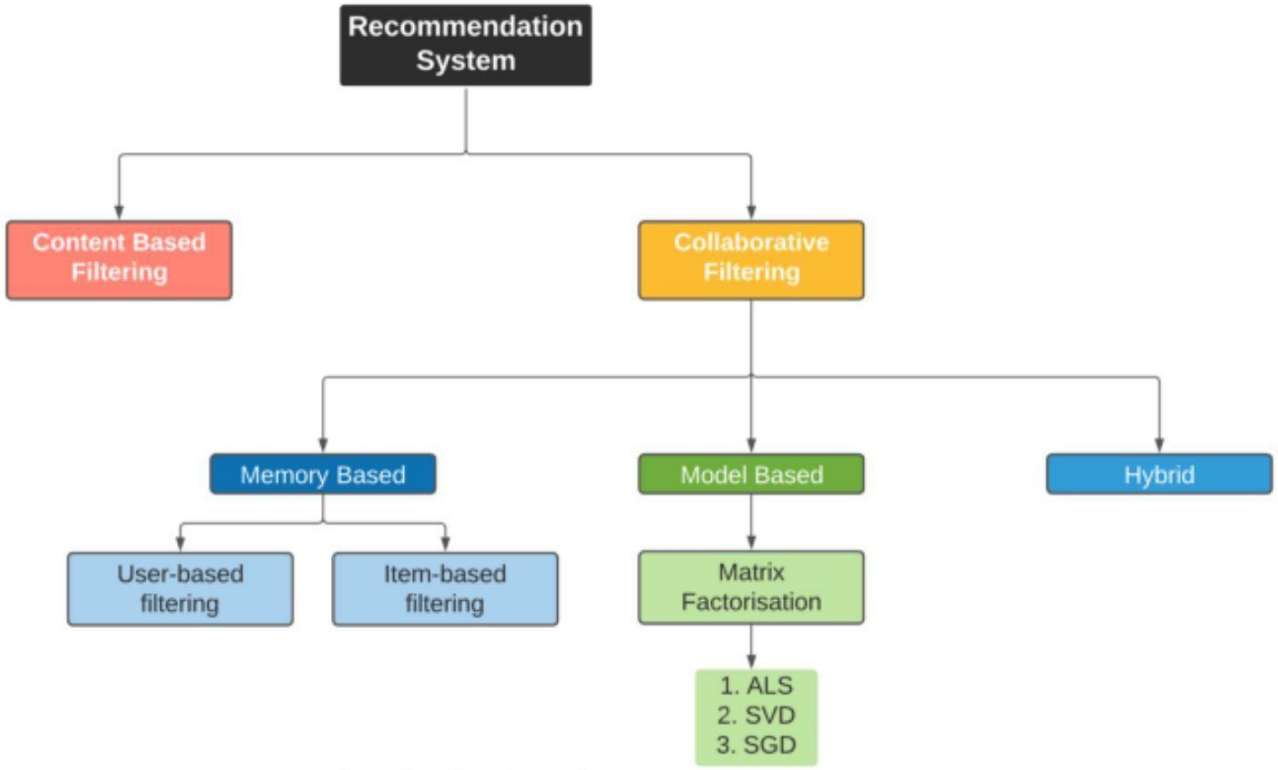
- Positive Reviews: "quality", "sound", "value", "battery", "fast"
- Negative Reviews: "defective", "stopped", "bad", "worst", "heating"

Review Helpfulness

- Analyzed upvotes/downvotes to identify the most engaging and trusted reviews.

Top helpful reviews usually had balanced, detailed feedback

FLOW CHART


Recommendation System
Content Based Filtering
Collaborative Filtering
Memory Based
Model Based
Hybrid
User-based filtering
Item-based filtering
Matrix Factorisation
1. ALS
2. SVD
3. SGD

1

IMPLEMENTATION

Matrix Factorization

In collaborative filtering, **matrix factorization** is the state-of-the-art solution for sparse data problem

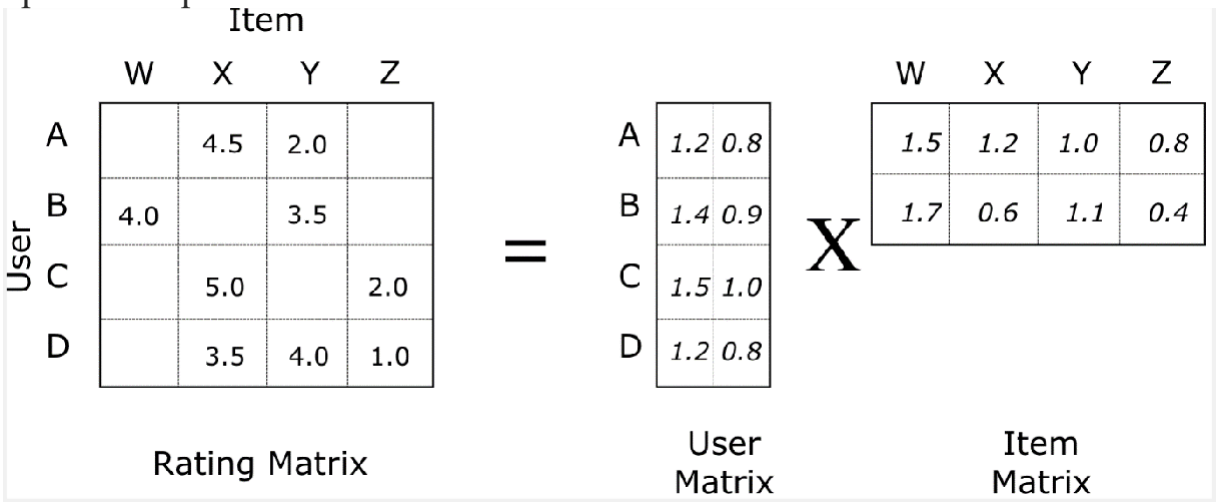


Fig: Matrix Factorization

Matrix factorization is a technique where the user-item interaction matrix is decomposed into two lower-dimensional matrices:

- A **user matrix** where each row represents a user and columns represent latent features.
- An **item (product) matrix** where each row represents latent features and columns represent items (e.g., products).

In the context of product review analysis, matrix factorization helps model hidden patterns in customer preferences and product features. The predicted rating r^ui\hat{r}_{ui}r^ui for a user u and item i is computed as the dot product of their respective latent vectors.

In the sparse user-item interaction matrix, the predicted rating user u will give item i is computed as:

$$\tilde{r}_{ui} = \sum_{f=0}^{nfactors} H_{u,f} W_{f,i}$$

where H is user matrix, W is item matrix

Rating of item i given by user u can be expressed as a dot product of the user latent vector and the item latent vector.

Notice in above formula, the number of **latent factors** can be tuned via cross-validation. **Latent factors** are the features in the lower dimension latent space projected from user-item interaction matrix. The idea behind matrix factorization is to use latent factors to represent user preferences or movie topics in a much lower dimension space. Matrix factorization is one of very effective **dimension reduction** techniques in machine learning.

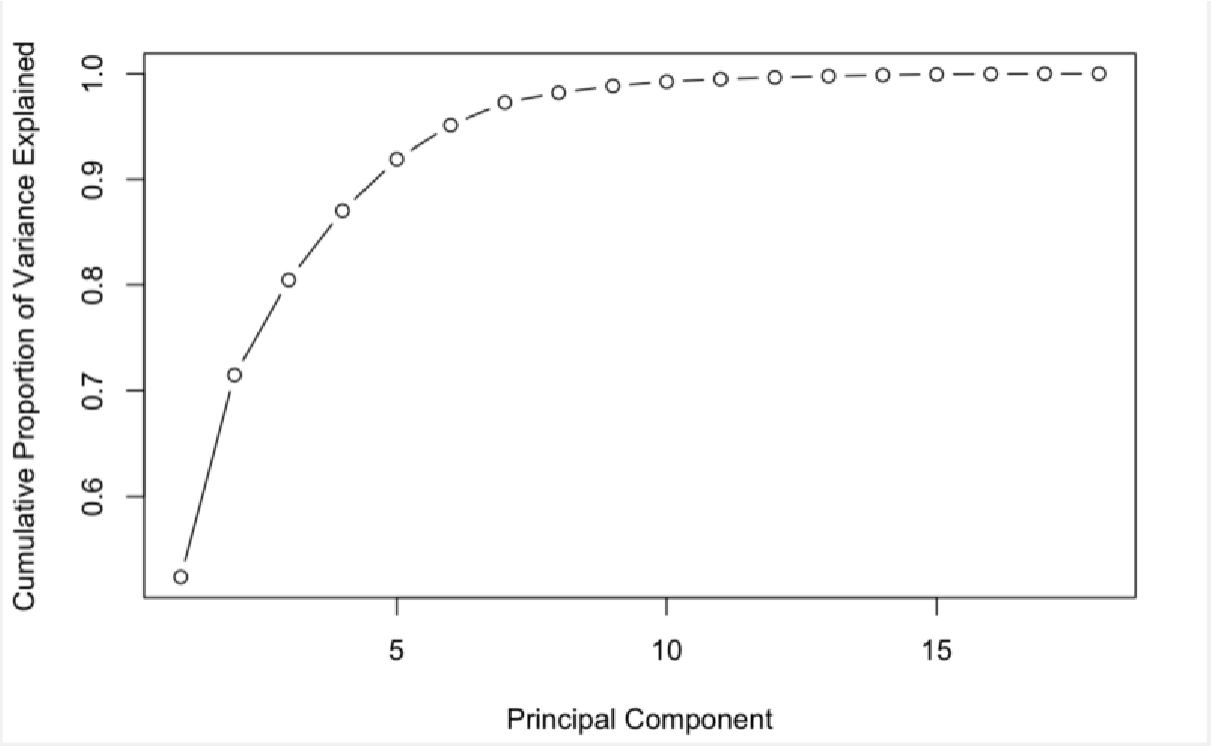


Fig: Variance Explained by Components In PCA

Very much like the concept of **components** in **PCA**, the number of latent factors determines the amount of abstract information that we want to store in a lower dimension space. A matrix factorization with one latent factor is equivalent to a most popular or top popular recommender (e.g. recommends the items with the most interactions without any personalization). Increasing the number of latent factors will improve personalization, until the number of factors becomes too high, at which point the model starts to overfit. A common strategy to avoid overfitting is to add **regularization terms** to the objective function.

The objective of matrix factorization is to minimize the error between true rating and predicted rating:

$$\arg\min_{H,W} \|R - \tilde{R}\|_{\mathrm{F}} + \alpha\|H\| + \beta\|W\|$$

where H is user matrix, W is item matrix

Once we have an objective function, we just need a training routine (eg, gradient descent) to complete the implementation of a matrix factorization algorithm. This implementation is actually called **Funk SVD**. It is named after Simon Funk, who he shared his findings with the research community during Netflix prize challenge in 2006.

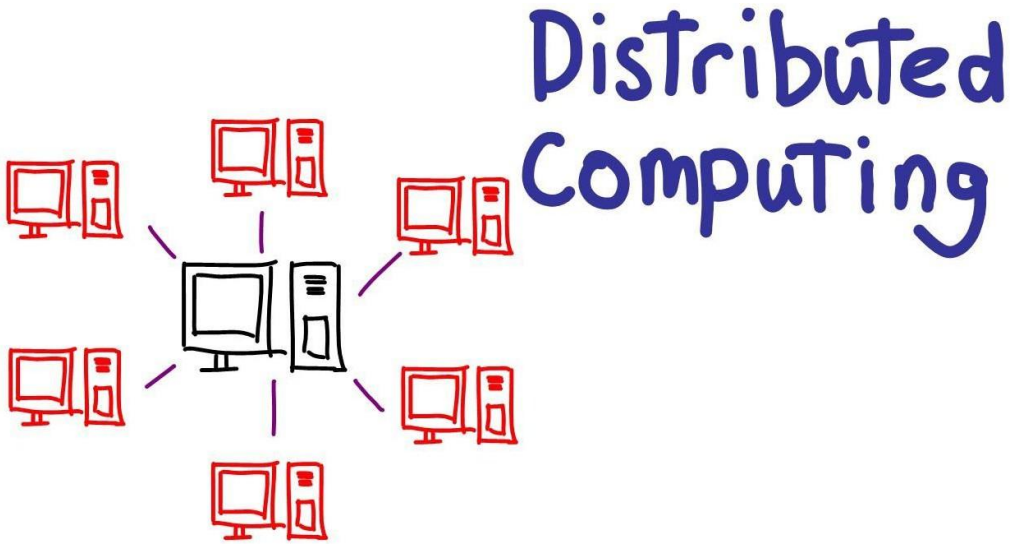


Fig: Scaling Machine Learning Applications With Distributed Computing

Although Funk SVD was very effective in matrix factorization with single machine during that time, it's not **scalable** as the amount of data grows today. With terabytes or even petabytes of data, it's impossible to load data with such size into a single machine. So we need a machine learning model (or framework) that can train on dataset spreading across from cluster of machines.

Alternating Least Square (ALS) with Spark ML

Alternating Least Square (ALS) is also a matrix factorization algorithm and it runs itself in a parallel fashion. ALS is implemented in Apache Spark ML and built for a larger-scale collaborative filtering problems. ALS is doing a pretty

good job at solving scalability and sparseness of the Ratings data, and it's simple and scales well to very large datasets.

Some high-level ideas behind ALS are:

- Its objective function is slightly different than Funk SVD: ALS uses **L2 regularization** while Funk uses **L1 regularization.**
- Its training routine is different: ALS minimizes **two loss functions alternatively**; It first holds user matrix fixed and runs gradient descent with item matrix; then it holds item matrix fixed and runs gradient descent with user matrix.
- Its scalability: ALS runs its gradient descent in **parallel** across multiple partitions of the underlying training data from a cluster of machines

SGD Algorithm for MF

Input: training matrix V, the number of features K, regularization parameter λ, learning rate ϵ
Output: row related model matrix W and column related model matrix H

Initialize W, H to $UniformReal(0, \frac{1}{\sqrt{K}})$
repeat
  for random $V_{ij} \in V$ **do**
    $error = W_{i*}H_{*j} - V_{ij}$
    $W_{i*} = W_{i*} - \epsilon(error \cdot H_{*j}^{\intercal} + \lambda W_{i*})$
    $H_{*j} = H_{*j} - \epsilon(error \cdot W_{i*}^{\intercal} + \lambda H_{*j})$
  end for
until convergence

Fig: Pseudocode for SGD In Matrix Factorization

Like many machine learning algorithms, Alternating Least Squares (ALS) has several key hyperparameters that must be tuned for optimal performance. These can be tuned using hold-out validation or cross-validation.

Key Hyperparameters in ALS:

- **maxIter:** Maximum number of iterations (e.g., 10)
- **rank:** Number of latent factors in the model (e.g., 20)
- **regParam:** Regularization parameter to prevent overfitting (e.g., 0.05)

Hyperparameter tuning is a repetitive yet essential task in machine learning pipelines. Automating this tuning process can significantly accelerate model development.

Implementing ALS Recommender System

Now that we have a robust model for analyzing product reviews and generating recommendations, the next step is to operationalize it.

Instead of movies, our goal is to recommend **products** based on customer interaction data (e.g., product ratings, reviews, or purchase history). Here's how a simple MVP (Minimum Viable Product) recommendation system can be structured:

Workflow:

1. **User Input**: A new user provides ratings or interactions with a few known products.
2. **Model Update**: The ALS model is retrained or updated with the new user-product data.
3. **Prediction Base**: The system gathers the full set of products to evaluate.
4. **Inference**: The system predicts ratings for all products for that specific user using the ALS model.
5. **Recommendation Output**: The top N products with the highest predicted ratings are presented as personalized recommendations.

CODE

Importing required libraries:

SparkSession:
Used to create a Spark session which allows interaction with Apache Spark functionality. You can create DataFrames, register them as tables, run SQL queries, cache tables, and load data formats like Parquet.
This is the starting point for any PySpark application:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("EcommerceProductRecommendation").getOrCreate()
```

ALS (Alternating Least Squares):
ALS is used to build collaborative filtering models based on implicit or explicit feedback. In our case, we use it to **predict user preferences for products** by factoring the user-product interaction matrix into two lower-rank matrices:

$$R \approx U \times P^T$$

Where **R** is the rating matrix, **U** is the user latent feature matrix, and **P** is the product latent feature matrix.
RegressionEvaluator:
Evaluates the performance of the ALS model, typically using Root Mean Square Error (RMSE) for the predicted vs. actual ratings.
pyspark.sql.functions as F:
A collection of built-in PySpark functions (such as col, when, lit, etc.) used for data transformation and preparation tasks.

```
[ ] df_clean = df_spark.filter((df_spark["review"].isNotNull()) & (df_spark["rating"].isNotNull()))

def get_sentiment(text):
    try:
        polarity = TextBlob(text).sentiment.polarity
        if polarity > 0.1:
            return "Positive"
        elif polarity < -0.1:
            return "Negative"
        else:
            return "Neutral"
    except:
        return "Neutral"

sentiment_udf = udf(get_sentiment, StringType())
df_sentiment = df_clean.withColumn("sentiment", sentiment_udf(col("review")))

[ ] sentiment_pd = df_sentiment.groupBy("sentiment").count().toPandas()
fig1 = px.pie(sentiment_pd, names="sentiment", values="count", hole=0.4, title="Sentiment Distribution")
fig1.show()
```

```
!pip install pyspark
```

```
Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.5)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.11/dist-packages (from pyspark) (0.10.9.7)
```

```
[ ] from pyspark.sql import SparkSession
    from pyspark.sql.types import *
    from pyspark.sql import functions as F
    from pyspark.ml.recommendation import ALS
    from pyspark.ml.evaluation import RegressionEvaluator

    import pandas as pd
    import matplotlib.pyplot as plt
    import seaborn as sns
```

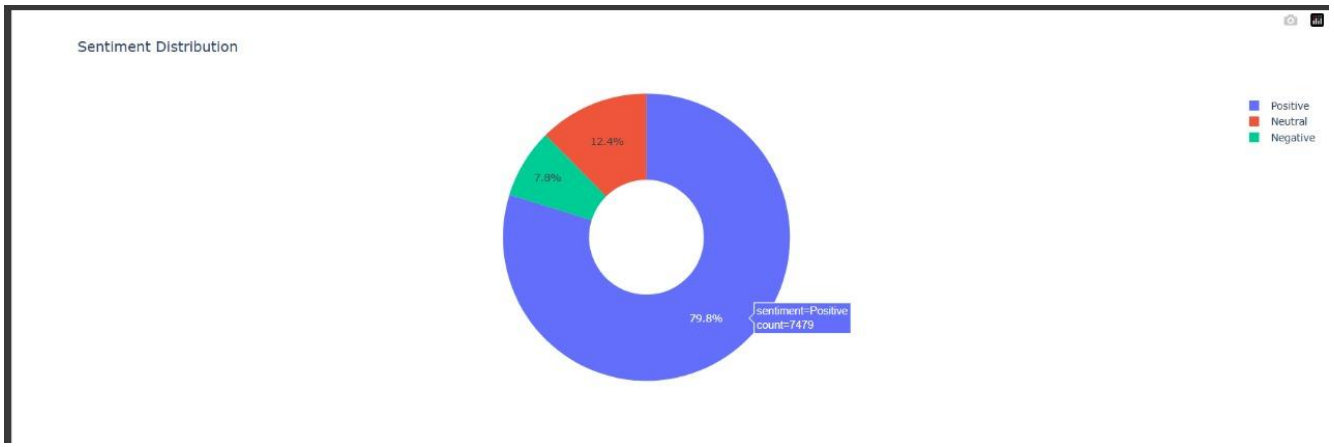
```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf, col
from pyspark.sql.types import StringType

from textblob import TextBlob
import pandas as pd
import plotly.express as px
import matplotlib.pyplot as plt
from wordcloud import WordCloud
```

```
[ ] spark = SparkSession.builder \
        .appName("Flipkart Review Sentiment Analysis") \
        .getOrCreate()
```

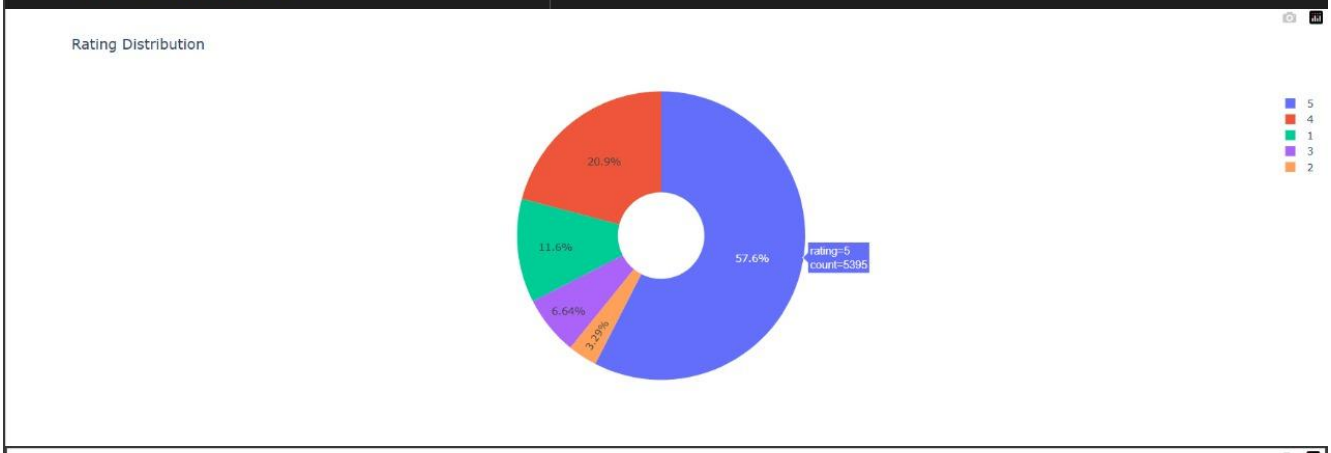
```
[ ] df_spark = spark.read.csv("/content/Flipkart_Reviews - Electronics.csv", header=True, inferSchema=True)
```

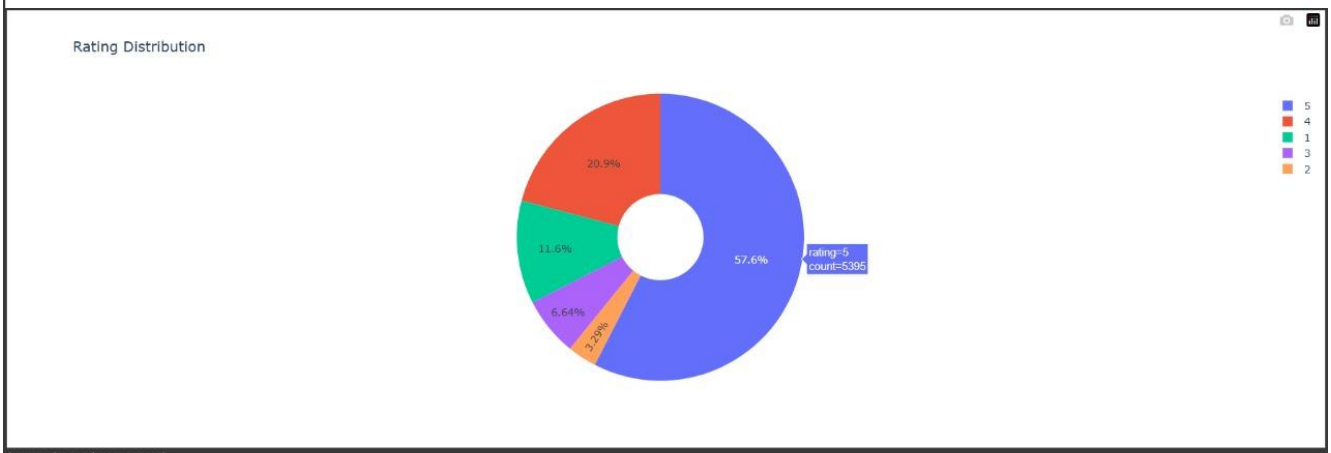
```
[ ] df_sentiment.printSchema()
```

```
root
 |-- product_id: string (nullable = true)
 |-- product_title: string (nullable = true)
 |-- rating: integer (nullable = true)
 |-- summary: string (nullable = true)
 |-- review: string (nullable = true)
 |-- location: string (nullable = true)
 |-- date: string (nullable = true)
 |-- upvotes: string (nullable = true)
 |-- downvotes: string (nullable = true)
 |-- sentiment: string (nullable = true)
```



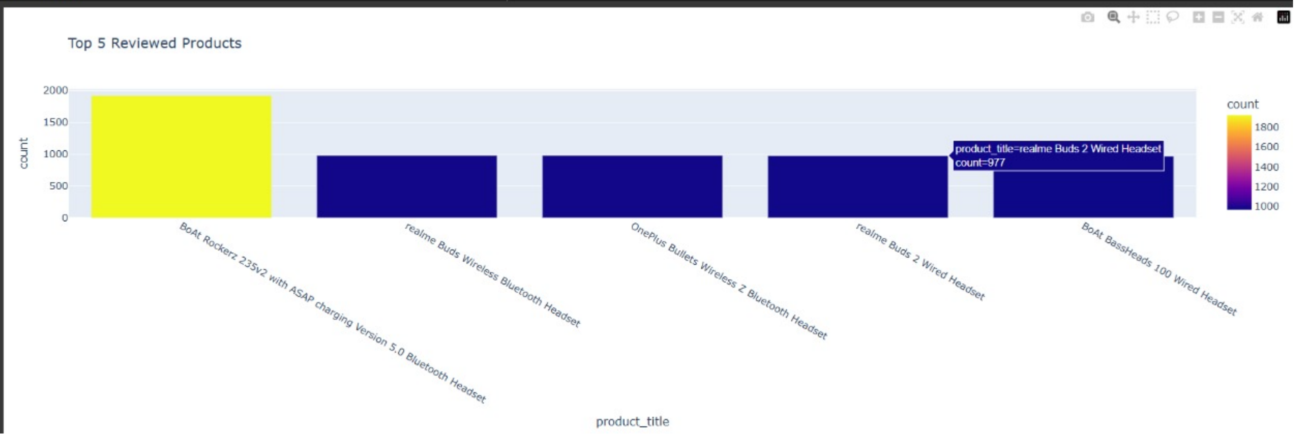
```python
rating_pd = df_sentiment.groupBy("rating").count().orderBy("rating").toPandas()
fig2 = px.pie(rating_pd, names="rating", values="count", hole=0.3, title="Rating Distribution")
fig2.show()
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, round

# Step 1: Start Spark session
spark = SparkSession.builder.appName("RatingSummary").getOrCreate()

# Step 2: Load data
df = spark.read.csv("/content/Flipkart_Reviews - Electronics.csv", header=True, inferSchema=True)

# Step 3: Filter non-null ratings
df = df.filter(col("rating").isNotNull())

# Step 4: Count of each rating
rating_summary = df.groupBy("rating") \
                   .agg(count("*").alias("count"))

# Step 5: Get total number of reviews
total_reviews = df.count()

# Step 6: Add percentage column
rating_summary = rating_summary.withColumn(
    "percentage", round((col("count") / total_reviews) * 100, 2)
)

# Step 7: Show the summary
rating_summary.orderBy("rating").show()
```






```
+------+-----+----------+
|rating|count|percentage|
+------+-----+----------+
|     1| 1090|     11.63|
|     2|  308|      3.29|
|     3|  622|      6.64|
|     4| 1959|      20.9|
|     5| 5395|     57.55|
+------+-----+----------+
```

```python
top_products = df_sentiment.groupBy("product_title").count().orderBy(col("count").desc()).limit(5).toPandas()
fig3 = px.bar(top_products, x="product_title", y="count", color="count", title="Top 5 Reviewed Products")
fig3.show()
```




```python
df_helpful = df_sentiment.withColumn("helpfulness", col("upvotes") - col("downvotes"))
most_helpful = df_helpful.orderBy(col("helpfulness").desc()).select("product_title", "review", "rating", "helpfulness").limit(5)
most_helpful.show(truncate=False)
```

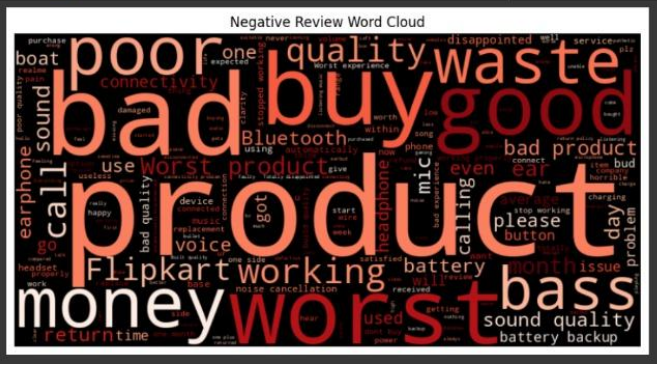
```
+--------------------------------------------+----------------------------------------------------------------------------------------
|product_title                               |review
+--------------------------------------------+----------------------------------------------------------------------------------------
|realme Buds Wireless Bluetooth Headset      |I will tell you some hidden features that you might have not heard and will summarize my review as follows first:👉Sound quality is 4/5👉Build and quality 5/5
|realme Buds Wireless Bluetooth Headset      |PROS: best in budget, with a good bass most loved thing is 🔋 backup. 1hour charge gives minimum 10 ✌ 🎧 enjoymentMagnetic attaching helps more to improve bat
|BoAt BassHeads 100 Wired Headset            |nice product with beautiful design and fast delivery.... once again thank you flipkart
|realme Buds 2 Wired Headset                 |💛All in One🎧😍👍👍👍
|OnePlus Bullets Wireless Z Bluetooth Headset|LONG BUT WORTH READING.Honest review after one week. Just go for it. Dont get affected by negative reviews. Totally worth every penny and far ahead of its compe
+--------------------------------------------+----------------------------------------------------------------------------------------
```

```python
positive_text = " ".join(df_sentiment.filter(df_sentiment.sentiment == "Positive").select("review").rdd.flatMap(lambda x: x).collect())
negative_text = " ".join(df_sentiment.filter(df_sentiment.sentiment == "Negative").select("review").rdd.flatMap(lambda x: x).collect())
```

```python
wordcloud_pos = WordCloud(background_color="white", width=800, height=400).generate(positive_text)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud_pos, interpolation="bilinear")
plt.axis("off")
plt.title("Positive Review Word Cloud")
plt.show()
```

```python
wordcloud_pos = WordCloud(background_color="white", width=800, height=400).generate(positive_text)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud_pos, interpolation="bilinear")
plt.axis("off")
plt.title("Positive Review Word Cloud")
plt.show()
```

```
wordcloud_neg = WordCloud(background_color="black", colormap="Reds", width=800, height=400).generate(negative_text)
plt.figure(figsize=(10, 5))
plt.imshow(wordcloud_neg, interpolation="bilinear")
plt.axis("off")
plt.title("Negative Review Word Cloud")
plt.show()
```



CONCLUSION

The **E-commerce Product Review Analysis** system recommends products to users based on their preferences and interactions using collaborative filtering techniques. Recommender systems serve as intelligent information filtering tools that aim to predict user ratings or interests in various products by analyzing large volumes of data. This approach enhances user experience by offering personalized product suggestions and has become a critical component of modern e-commerce platforms.

FUTURE ENHANCEMENTS

In the future, the system can be extended by integrating product data from multiple online marketplaces or e-commerce platforms. Additional features such as **real-time review sentiment analysis**, **dynamic pricing trends**, and **product availability across platforms (like Amazon, Flipkart, etc.)** can be incorporated. The model can also include customer subscription types (e.g., Prime, Plus) and recommend products accordingly. Including metrics such as **view counts, cart additions, or wishlist trends** can further refine the recommendation quality.

REFERENCES

- Build Recommendation System with PySpark using ALS Matrix Factorisation – Towards Data Science
- Crafting Recommendation Engine in PySpark – Medium
- Recommendation System in Python using ALS Algorithm and Apache Spark – Mediu