Exploring Movie Recommendations with Apache Spark

Andres Rocha, Andrew Chan, Sophia Chung,Yohan Sofian

June 4, 2024

Abstract

This report details the development and functioning of a movie recommendation system built using Apache Spark. The system preprocesses data, computes feature vectors, and applies machine learning techniques to recommend movies based on textual and numerical data.

1 High-Level Overview

Introduction

The objective of this project is to build a scalable movie recommendation system using Apache Spark. This system leverages text processing and machine learning to suggest movies similar to a given query.

Setup

The project is implemented in Scala and uses Apache Spark, an open-source unified analytics engine for large-scale data processing. The environment is set up as follows:

```
val spark = SparkSession.builder()
  .appName("MovieRecommendation")
  .master("local[*]")
  .getOrCreate()
```

```
Logger.getLogger("org").setLevel(Level.OFF)
Logger.getLogger("akka").setLevel(Level.OFF)
```

Data Loading and Preprocessing

Data is loaded from a CSV file containing top-rated movies. Each movie's 'overview' is preprocessed to replace nulls with a default string. Here, text preprocessing involves tokenization and removal of stop words to clean the data for further analysis.

Feature Engineering

The system creates feature vectors using several techniques:

- **TF-IDF:** Converts text data into a numeric form, reflecting the importance of words within the dataset.
- **Vector Assembler:** Combines features from different sources (TF-IDF vectors and numerical attributes like ratings and popularity) into a single feature vector.
- **Standard Scaler:** Standardizes features by scaling to unit variance.

Machine Learning Pipeline

A pipeline is constructed with various stages from tokenization to scaling. This pipeline automates the workflow of transforming and assembling data:

```
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, remover, hashingTF, idf, assembler, scaler))
```

Recommendation Engine

The system uses cosine similarity to find movies that are most similar to a given query. This measure helps identify movies with similar feature vectors, hence likely to be of interest to the user.

Example Query and Results

An example query is processed through the system to find movies similar to "Spider-Man: Across the Spider-Verse", demonstrating the effectiveness of the system.

title	overview	vote_average	vote_count	popularity	runtime	similarity
Spider-Man: Acros...	After reuniting w...	8.8	1160.0	2859.047	140.0	1.0
Midnight Cowboy	"Joe Buck is a wi...	7.5	1151.0	17.437	4.4785053E7	0.29027100842634684
Giant Spider	In a mysterious l...	7.2	57.0	18.372	84.0	0.19111307153145093
Eight Legged Freaks	The residents of ...	5.7	1059.0	346.156	99.0	0.18455005708099004
The Amazing Spide...	For Peter Parker,...	6.5	12057.0	163.998	141.0	0.17653226359680757

Conclusion

This movie recommendation system showcases the power of Apache Spark in handling and analyzing large datasets. Through effective preprocessing and feature engineering, it offers a robust platform for movie recommendations.

2 In-Depth Analysis of Code

In-depth Analysis of MovieRecommendation Code Author Name June 4, 2024

Introduction

This part of document provides an in-depth analysis of the `MovieRecommendation` code written in Scala. The code is designed to preprocess movie data and implement a recommendation system using Apache Spark and its machine learning library.

Imports and Setup

```scala
import org.apache.log4j.{Level, Logger, PropertyConfigurator}
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.ml.feature.{HashingTF, IDF,
    RegexTokenizer, StandardScaler, StopWordsRemover,
    VectorAssembler}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.ml.{Pipeline, PipelineModel}
```

Listing 1: Import Statements

Explanation

The code begins by importing necessary libraries and packages:

- `org.apache.log4j`: For logging configuration.
- `org.apache.spark.sql`: For Spark SQL and DataFrame operations.
- `org.apache.spark.ml.feature`: For various machine learning feature transformations.
- `org.apache.spark.ml.linalg`: For linear algebra operations.
- `org.apache.spark.ml`: For creating machine learning pipelines.

Main Object and Spark Session

```scala
object MovieRecommendation {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder()
      .appName("MovieRecommendation")
      .master("local[*]")
      .getOrCreate()

    Logger.getLogger("org").setLevel(Level.OFF)
    Logger.getLogger("akka").setLevel(Level.OFF)

    PropertyConfigurator.configure("src/main/resources/log4j.
  ↪ properties")
```

Listing 2: Main Object and Spark Session

Explanation

- The main object `MovieRecommendation` contains the entry point of the application.
- A Spark session is created using `SparkSession.builder()` with the application name "MovieRecommendation" and master set to local mode.
- Logging levels for Spark and Akka are set to OFF to reduce console output noise.
- The logging configuration is loaded from a properties file.

Data Loading and Cleaning

```scala
    import spark.implicits._

    val df = spark.read.option("header", "true")
      .csv("top_1000_popular_movies_tmdb.csv")
      .withColumn("vote_average", $"vote_average".cast("
  ↪ double"))
      .withColumn("vote_count", $"vote_count".cast("double"))
      .withColumn("popularity", $"popularity".cast("double"))
      .withColumn("runtime", $"runtime".cast("double"))

```

```
    val defaultOverview = "No overview available"
    val cleanedDF = df.na.fill(Map("overview" -> defaultOverview))
```

Listing 3: Data Loading and Cleaning

Explanation

- The dataset is loaded from a CSV file with headers.
- Columns `vote_average`, `vote_count`, `popularity`, and `runtime` are cast to double for numerical operations.
- Missing or empty `overview` fields are filled with a default value "No overview available".

Handling Missing Values

```
    val voteAverageMedian = cleanedDF.stat.approxQuantile("vote_average", Array(0.5), 0.001).head
    val voteCountMedian = cleanedDF.stat.approxQuantile("vote_count", Array(0.5), 0.001).head
    val popularityMedian = cleanedDF.stat.approxQuantile("popularity", Array(0.5), 0.001).head
    val runtimeMedian = cleanedDF.stat.approxQuantile("runtime", Array(0.5), 0.001).head

    val filledDF = cleanedDF.na.fill(Map(
      "vote_average" -> voteAverageMedian,
      "vote_count" -> voteCountMedian,
      "popularity" -> popularityMedian,
      "runtime" -> runtimeMedian
    ))
```

Listing 4: Handling Missing Values

Explanation

- Median values for `vote_average`, `vote_count`, `popularity`, and `runtime` are calculated.
- Missing values in these columns are filled with their respective medians.

Text Preprocessing

```
val tokenizer = new RegexTokenizer()
  .setInputCol("overview")
  .setOutputCol("tokens")
  .setPattern("\\W")

val remover = new StopWordsRemover()
  .setInputCol("tokens")
  .setOutputCol("filtered_tokens")

val hashingTF = new HashingTF()
  .setInputCol("filtered_tokens")
  .setOutputCol("raw_features")
  .setNumFeatures(10000)

val idf = new IDF()
  .setInputCol("raw_features")
  .setOutputCol("tfidf_features")
```

Listing 5: Text Preprocessing

Explanation

- `RegexTokenizer`: Tokenizes the `overview` text into words.
- `StopWordsRemover`: Removes common stop words from the tokens.
- `HashingTF`: Converts the tokens into raw feature vectors of fixed size (10,000).
- `IDF`: Applies the Inverse Document Frequency transformation to the raw features to produce TF-IDF features.

Feature Assembly and Scaling

```
val assembler = new VectorAssembler()
  .setInputCols(Array("tfidf_features", "vote_average", "vote_count", "popularity", "runtime"))
  .setOutputCol("features")
  .setHandleInvalid("skip")

val scaler = new StandardScaler()
```

```
      .setInputCol("features")
      .setOutputCol("scaled_features")
      .setWithStd(true)
      .setWithMean(false)
```

Listing 6: Feature Assembly and Scaling

Explanation

- `VectorAssembler`: Combines the TF-IDF features with numerical columns into a single feature vector.
- `StandardScaler`: Scales the features to have unit standard deviation without centering them around the mean.

Pipeline and Model Training

```
val pipeline = new Pipeline().setStages(Array(tokenizer,
↪ remover, hashingTF, idf, assembler, scaler))

val model = pipeline.fit(filledDF)
val processedDF = model.transform(filledDF)
```

Listing 7: Pipeline and Model Training

Explanation

- A pipeline is created with the stages: tokenizer, remover, hashingTF, idf, assembler, and scaler.
- The pipeline is fitted to the filled DataFrame and used to transform it, producing the processed DataFrame.

Cosine Similarity and Nearest Neighbors

```
def cosineSimilarity(v1: Vector, v2: Vector): Double = {
  val dotProduct = v1.toArray.zip(v2.toArray).map { case
↪ (x, y) => x * y }.sum
  val normA = math.sqrt(v1.toArray.map(x => x * x).sum)
  val normB = math.sqrt(v2.toArray.map(x => x * x).sum)
```

```
    dotProduct / (normA * normB)
  }

  def findNearestNeighbors(query: String, numericalData: Array[Double], k: Int = 5): DataFrame = {
    val queryDF = Seq((query, numericalData(0), numericalData(1), numericalData(2), numericalData(3))).toDF("overview", "vote_average", "vote_count", "popularity", "runtime")
    val queryProcessedDF = model.transform(queryDF)
    val queryFeatures = queryProcessedDF.select("scaled_features").first().getAs[Vector]("scaled_features")

    val similarities = processedDF.select("title", "overview", "vote_average", "vote_count", "popularity", "runtime", "scaled_features").as[(String, String, Double, Double, Double, Double, Vector)].map {
      case (title, overview, voteAvg, voteCount, popularity, runtime, features) =>
        val similarity = cosineSimilarity(queryFeatures, features)
        (title, overview, voteAvg, voteCount, popularity, runtime, similarity)
    }

    val nearestNeighbors = similarities.sort($"_7".desc).take(k)
    spark.createDataFrame(nearestNeighbors).toDF("title", "overview", "vote_average", "vote_count", "popularity", "runtime", "similarity")
  }
```

Listing 8: Cosine Similarity and Nearest Neighbors

Explanation

- `cosineSimilarity`: Computes the cosine similarity between two feature vectors.
- `findNearestNeighbors`: Finds the top `k` nearest neighbors to a given query movie based on the cosine similarity of their features.
- A query DataFrame is created and transformed using the pipeline

model.

- Cosine similarities between the query movie and all movies in the processed DataFrame are computed.
- The top `k` most similar movies are returned as a DataFrame.

Example Query and Result Display

```scala
    val query = "After reuniting with Gwen Stacy, Brooklyns full-time, friendly neighborhood Spider-Man is catapulted across the Multiverse, where he encounters the Spider Society, a team of Spider-People charged with protecting the Multiverses very existence. But when the heroes clash on how to handle a new threat, Miles finds himself pitted against the other Spiders and must set out on his own to save those he loves most."
    val numericalData = Array(8.8, 1160, 2859.047, 140)
    val nearestNeighbors = findNearestNeighbors(query, numericalData)

    nearestNeighbors.show()
    spark.stop()
  }
}
```

Listing 9: Example Query and Result Display

title	overview	vote_average	vote_count	popularity	runtime	similarity
Spider-Man: Acros...	After reuniting w...	8.8	1160.0	2859.047	140.0	1.0
Midnight Cowboy	"Joe Buck is a wi...	7.5	1151.0	17.437	4.4785053E7	0.29027100842634684
Giant Spider	In a mysterious l...	7.2	57.0	18.372	84.0	0.19111307153145093
Eight Legged Freaks	The residents of ...	5.7	1059.0	346.156	99.0	0.18455005708099004
The Amazing Spide...	For Peter Parker,...	6.5	12057.0	163.998	141.0	0.17653226359680757

Explanation

- An example query movie description and associated numerical data are provided.

- The `findNearestNeighbors` function is called to find the nearest neighbors to the example query.
- The results are displayed using `show()` and the Spark session is stopped.