lab17_spark_nlp_intro

Harito ID

2025-09-25

Lab 17: Building a Spark NLP Pipeline

Objectives:

- Understand and build a multi-stage Spark ML `Pipeline`.
- Read a real-world JSON dataset (`C4`) into a Spark `DataFrame`.
- Use different tokenizers (`RegexTokenizer`, `Tokenizer`) to process text.
- Apply feature engineering stages like `StopWordsRemover`, `HashingTF`, and `IDF`.
- Fit a pipeline to data and use it to transform text into feature vectors.
- Save the results to a local file.

Instructions:

1. **Setup:** To run the Spark project, you need to install Java (JDK) and `sbt` (Scala Build Tool). `sbt` will automatically manage and download the Spark libraries, so you **do not need to install Spark separately**.

   On Arch Linux: Open a terminal and run the following commands:

   ```
   # Update your system
   sudo pacman -Syu

   # Install Java (OpenJDK 17) and sbt
   sudo pacman -S jdk-openjdk sbt

   # Example:
   # Package (2)        Old Version  New Version  Net Change  Download Size

   # extra/jdk-openjdk  25.u36-1     25.u36-1       0.00 MiB
   # extra/sbt                      1:1.11.6-1    67.94 MiB      19.86 MiB
   #
   # Total Download Size:    19.86 MiB
   # Total Installed Size:  614.29 MiB
   # Net Upgrade Size:       67.94 MiB
   ```

   On Debian/Ubuntu:

   ```
   # Update and install
   sudo apt update
   sudo apt install openjdk-17-jdk sbt
   ```

   On macOS (using Homebrew):

```
# Install Java 17 and sbt
brew install openjdk@17 sbt
```

On Windows:

For Java Link or version higher. Download and run the .msi file. For sbt Link or version higher. Run the .msi file.

Verify Installation: After installation, check the versions to make sure everything is ready:

```
java -version
sbt --version

# In Windows PowerShell, use:
sbt sbtVersion
```

2. **Explore the Pipeline Code:**
   - Open the `spark_labs` directory.
   - Open the `src/main/scala/com/harito/spark/Lab17_NLPPipeline.scala` file. Make the name folder as your own.
   - Read through the code and comments to understand the different stages of the pipeline:
     1. **Data Loading:** This stage reads the `c4-train.00000-of-01024-30K.json.gz` dataset, which is a gzipped JSON file. It uses `spark.read.json()` to load the data into a Spark DataFrame. For faster execution during the lab, the data is limited to the first 1000 records.
     2. **Tokenization:** The `RegexTokenizer` is used here to split the input text into individual words or tokens based on a regular expression pattern. The code also includes a commented-out `Tokenizer` for a simpler, whitespace-based tokenization alternative.
     3. **Stop Word Removal:** The `StopWordsRemover` stage filters out common, less informative words (like “the”, “a”, “is”) from the tokenized list. This helps to reduce noise and improve the quality of the subsequent feature extraction.
     4. **TF-IDF Vectorization:** This is a two-step process to convert the filtered tokens into numerical feature vectors:
        - `HashingTF`: Converts the tokens into a fixed-size feature vector by hashing each token into an index and counting its occurrences (Term Frequency).
        - `IDF`: Scales down the importance of words that appear frequently across many documents (Inverse Document Frequency), giving more weight to unique and informative terms.

3. **Run the Application:**
   - Using a terminal, navigate to the `spark_labs` directory.
   - Run the command `sbt run`. `sbt` will compile the code, download all dependencies, and execute the application. This might take some time on the first run.
   - Observe the output in the console. The application will create a new file at `results/lab17_pipeline_output.txt` with the processed data.

Exercises:

1. **Switch Tokenizers:** In `Lab17_NLPPipeline.scala`, comment out the `RegexTokenizer` and uncomment the basic `Tokenizer`. Rerun the application and observe how the output tokens and vectors might change.
2. **Adjust Feature Vector Size:** Change the `numFeatures` in the `HashingTF` stage from `20000` to `1000`. How does this affect the resulting vectors?
3. **Extend the Pipeline:** The current pipeline stops after creating feature vectors. A common next step is to train a machine learning model. Add a `LogisticRegression` stage to the pipeline to classify the text (you may need to add a label column to the data first).
4. **Try a Different Vectorizer:** Comment out the `HashingTF` and `IDF` stages. In their place, implement a `Word2Vec` stage to generate word embeddings for each document.

Guide: Updating Scala and Spark Versions

Keeping libraries up-to-date is very important. All version management for this Spark project is handled in the `spark_labs/build.sbt` file.

Step 1: Find the Latest Versions

1. **Apache Spark:** Visit the official Apache Spark Downloads page. This page will always list the latest release. **Important:** pay attention to the Scala version Spark is pre-built for (e.g., "*for Scala 2.13*"). This determines the Scala version you should use in your project.
2. **Scala and other libraries:** MVNRepository is an excellent tool. You can search for any library (e.g., `org.apache.spark:spark-core`) to see all its released versions.

Step 2: Edit the `build.sbt` file

Let's assume you want to upgrade to Spark `4.0.0` (a hypothetical future release), and this version requires Scala `2.13.x`.

You would open the `spark_labs/build.sbt` file and change the corresponding lines.

Example:

Old `build.sbt` file:

```
ThisBuild / scalaVersion := "2.12.18"

lazy val root = (project in file("."))
  .settings(
    name := "spark-nlp-labs",
    libraryDependencies ++= Seq(
      "org.apache.spark" %% "spark-core" % "3.5.1",
      "org.apache.spark" %% "spark-sql" % "3.5.1",
      "org.apache.spark" %% "spark-mllib" % "3.5.1"
    )
  )
```

New `build.sbt` file (updated):

```
// A new Spark version might require Scala 2.13
ThisBuild / scalaVersion := "2.13.12"
```

```
// Tip: define a variable for the Spark version for easier changes
val sparkVersion = "4.0.0"

lazy val root = (project in file("."))
  .settings(
    name := "spark-nlp-labs",
    libraryDependencies ++= Seq(
      "org.apache.spark" %% "spark-core" % sparkVersion,
      "org.apache.spark" %% "spark-sql" % sparkVersion,
      "org.apache.spark" %% "spark-mllib" % sparkVersion
    )
  )
```

Step 3: Reload the Project

After you save the `build.sbt` file, `sbt` needs to download the new libraries.

- If you are using an IDE like IntelliJ, it will usually show a button to "Reload sbt project".
- If you are using the terminal, simply running any command like `sbt compile` will cause sbt to automatically detect the changes and download the new dependencies.

Use Scala and Spark (or Python and PySpark) to build data pipelines and perform machine learning tasks on large datasets

Building a Data Pipeline with Spark

This exercise requires you to build a fundamental data processing pipeline using Spark to prepare a large text dataset for machine learning tasks.

Requirements

Using **Scala and Spark (or Python and PySpark)**, follow these steps:

1. **Read the Data:**
   - Load the compressed file `c4-train.00000-of-01024-30K.json.gz` into a Spark DataFrame.
   - The file contains a series of JSON records, with each record being a separate line.
2. **Perform Text Preprocessing:**
   - **Tokenization:** Use `RegexTokenizer` or another tokenizer to split the text into individual words (tokens).
   - **Stop Word Removal:** Remove common, low-information words (such as "a," "the," "is," etc.) using `StopWordsRemover`.
3. **Vectorize the Data:**
   - Use `HashingTF` to transform the preprocessed text into term frequency vectors.
   - Apply `IDF` (Inverse Document Frequency) to weigh the vectors, reducing the importance of words that appear frequently across the entire dataset.
4. **Save the Results:**

- Persist the final feature vectors to a file at the path `results/lab17_pipeline_output.`

5. **Log the Process:**
    - Record key information about the job run, including start time, end time, and any errors, to a log file within the `log/` directory.

You can write the code in either Scala or Python. Make sure your code is well-structured, clearly commented, and follows good programming practices.