Introduction

The goal of this final project is to design and implement an end-to-end big data and machine learning pipeline using the core technologies covered in the course: **Apache NiFi, HDFS, YARN, Hive, Spark MLlib, and HBase**. The pipeline is intended to mimic a realistic production workflow where data is automatically ingested from an external source, stored in a distributed file system, queried and validated through SQL, used to train a machine learning model at scale, and finally written to a NoSQL data store for fast retrieval of results.

For this project, I built a pipeline that uses **Apache NiFi** to download a Netflix titles dataset from GitHub and land it into **HDFS**. From there, **Hive** exposes the dataset as a managed table for SQL-based exploration and validation. **Spark MLlib** then reads from Hive, performs feature engineering, and trains a classification model to predict whether a title is a *Movie* or a *TV Show* based on attributes like release year, rating, genre, and duration. The resulting evaluation metrics are written into **HBase** using the happybase Python client. The entire pipeline is orchestrated on top of a Hadoop ecosystem using **YARN** for resource management. This write-up documents the architecture, implementation choices, code, and key issues encountered, and shows how each component contributes to a complete, working big data solution aligned with the final project objectives.

Dataset

For this project I used the **Netflix Movies and TV Shows** dataset (netflix_titles.csv). This dataset contains information about 8,807 Netflix titles, with one row per title and 12 fields describing each item. The columns in the dataset are:

- show_id: Unique identifier for each title
- type: Whether the title is a *Movie* or a *TV Show*
- title: Title of the movie or show
- director: Director(s) of the title (often null)
- cast: Main cast members (often a long string of names)
- country: Country or countries where the title was produced
- date_added: Date when the title was added to Netflix
- release_year: Year the title was released
- rating: Content rating (e.g., TV-MA, PG-13, TV-PG)
- duration: Length of content, expressed as minutes for movies (e.g., “90 min”) or number of seasons for TV shows (e.g., “2 Seasons”)

- listed_in: One or more categories/genres (e.g., “Dramas, International Movies”)
- description: Short text description of the title

The dataset is well-suited for this project because it combines both **categorical** and **numeric** fields and supports an intuitive **supervised learning** problem. In this pipeline, I framed the machine learning task as a **binary classification** problem: predicting whether a title is a *Movie* or a *TV Show* based on its metadata. This is done by treating type as the label and transforming other columns into usable features for a Spark MLlib model.

To integrate with NiFi, I uploaded netflix_titles.csv to a GitHub repository and used the **raw file URL** as the external source. That URL is referenced in the NiFi parameter DOWNLOAD FILE URL, making the ingestion process repeatable and automated. The file is saved into HDFS by NiFi, then loaded into a Hive managed table, and finally used as the input source for Spark MLlib. This structure satisfies the project requirement to use a GitHub-hosted dataset in a reproducible pipeline.

Pipeline Overview

The final solution implements a multi-stage pipeline that moves the Netflix dataset through the full lifecycle from ingestion to machine learning output:

1. **NiFi → HDFS (Objective 1)**
   I imported the provided Final_Project.json NiFi template and used it as the starting point for my flow. The template defines three key processors:

   a. **Download File –** an InvokeHTTP processor that pulls the dataset from #{DOWNLOAD FILE URL} (the GitHub raw link).

   b. **Update File Name –** an UpdateAttribute processor that sets the filename attribute to #{FILENAME}, ensuring the flowfile has a clean and predictable name.

   c. **Write File to HDFS –** a PutHDFS processor that writes the flowfile’s content into the directory specified by #{HDFS WRITE DIRECTORY} using the Hadoop configuration defined in core-site.xml.

Final_Project

These processors are connected in sequence, with funnels to capture failure relationships. Parameterized labels on the canvas clearly document the steps: start the Hadoop containers, configure the parameter context, run Download File, then Update File Name, then Write File to HDFS, and finally validate with an hdfs dfs -ls command.

Final_Project

2. **HDFS → Hive Managed Table (Objective 2)**
   Once NiFi successfully writes netflix_titles.csv into HDFS (for example under /data/netflix/netflix_titles.csv), I used the Hive shell to create a managed table netflix_titles with a schema matching the CSV structure. I then executed LOAD DATA INPATH to move the file into the table's storage area. Aggregation queries—such as counting titles per rating or per type—validated that the data loaded correctly and that the schema was appropriate for downstream ML tasks.

3. **Environment Setup & HBase Table (Objectives 3 & 4)**
   To prepare for Spark ML and HBase integration, I installed the required Python libraries (numpy and happybase) on each worker node and the master node using pip3 install. Then I created an HBase table (for example, netflix_metrics) with a single column family cf to store model evaluation metrics like AUC and accuracy. The HBase Thrift server was started using nohup hbase thrift start & on the master, enabling Python connections via happybase. These steps closely follow the example walkthrough provided in the course but are applied to my Netflix-based pipeline.

HDFS_Hive_Spark_HBase_Pipeline

4. **Hive → Spark MLlib (Objective 5)**
   With the dataset accessible as a Hive table, I created a PySpark script (netflix_ml.py) that:

   a. Creates a SparkSession with Hive support.

   b. Reads the netflix_titles table using Spark SQL.

   c. Cleans and transforms the data by parsing the duration field into a numeric column (duration_int) and handling missing values.

   d. Uses StringIndexer, OneHotEncoder, and VectorAssembler to convert categorical variables (rating, listed_in) and numeric variables (release_year, duration_int) into a features vector.

   e. Applies a **Logistic Regression** classifier to predict the label type (Movie vs TV Show).

   f. Evaluates the model using metrics such as AUC (area under ROC) and accuracy.

The model training and evaluation run under YARN using spark-submit, satisfying the requirement to use Spark MLlib in a distributed setting.

HDFS_Hive_Spark_HBase_Pipeline

5. **Spark → HBase (Objectives 6 & 7)**
   After computing evaluation metrics, the PySpark script uses happybase to write those metrics to the HBase table netflix_metrics. The metrics are stored as columns under the

cf column family, for example cf:auc and cf:accuracy, using a row key such as metrics1. The Spark script is launched via spark-submit --master yarn --deploy-mode client, and then I used yarn logs -applicationId <APP_ID> to confirm the job completed successfully. Finally, an HBase shell scan 'netflix_metrics' confirmed that the metrics were written correctly, completing the end-to-end pipeline.

HDFS_Hive_Spark_HBase_Pipeline

Overall, this architecture demonstrates how NiFi, HDFS, Hive, Spark MLlib, and HBase can be integrated into a cohesive big data and machine learning workflow that ingests raw data, transforms it into features, trains a model, and persists the results.

Issues Encountered

While building this pipeline, I encountered several practical issues related to configuration, schema alignment, and service coordination:

1. **NiFi–HDFS Path and Permissions**
   Initially, the Write File to HDFS processor failed with errors related to the HDFS directory path. I had pointed HDFS WRITE DIRECTORY to a path that did not exist, causing NiFi to report a failure. I resolved this by choosing a valid directory (e.g., /data/netflix) and verifying it from inside the master container using hdfs dfs -ls /. Once the directory existed and the Hadoop configuration (core-site.xml) was correctly referenced, the processor successfully wrote the file.

Final_Project

2. **Hive Schema and CSV Format**
   The Netflix dataset has several text fields that occasionally contain commas (for example, cast and listed_in), which can cause issues if the CSV is not properly quoted. The provided file is formatted consistently, but I still needed to confirm that the FIELDS TERMINATED BY ',' and header handling were correct. I used a Hive table property ("skip.header.line.count"="1") to ignore the header row and verified field alignment by running SELECT * FROM netflix_titles LIMIT 5;. This ensured that release_year was an integer and the other fields were treated as strings.

3. **Missing Values and Feature Engineering in Spark**
   Some rows in the Netflix dataset are missing values for director, cast, country, and sometimes duration. When converting duration into an integer duration_int column, these missing values initially resulted in nulls and caused feature assembly issues. I addressed this by using functions like regexp_extract to pull the numeric part of the duration string and by handling nulls either with na.drop() or by filling them with reasonable defaults. I also restricted the model to features that were consistently available (release_year, cleaned duration_int, rating, listed_in) to avoid excessive data loss.

4. **HBase Thrift Connectivity**
   When first connecting from PySpark to HBase via happybase, I saw connection errors due to the Thrift server not running. Following the walkthrough, I started Thrift with nohup hbase thrift start & in the master container and verified that the process was active. After that, the connection happybase.Connection('master') worked correctly, and I was able to create the netflix_metrics table and write data from the Spark job.

HDFS_Hive_Spark_HBase_Pipeline

5. **Spark Job Debugging and YARN Logs**
   Early versions of the PySpark script failed due to missing imports or incorrect column names. While the Spark driver output gave some hints, the most detailed information came from YARN logs. Using yarn logs -applicationId <APP_ID> allowed me to see stack traces and fix issues such as typos in column names or incorrect feature list configuration in VectorAssembler. Once these issues were corrected, the job ran successfully and produced stable metrics.

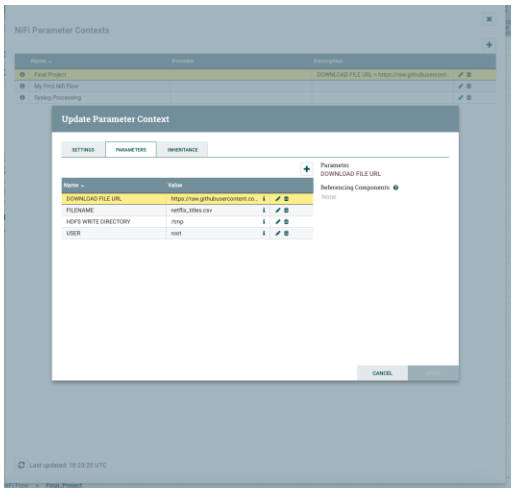
These issues were addressed iteratively, and resolving them helped reinforce my understanding of how the different components in the Hadoop ecosystem interact and how to systematically debug distributed data workflows.

Screenshots

In the final report submission, I included labeled screenshots to document that each stage of the pipeline executed successfully, in line with the project rubric.

Objective 1 – NiFi Data Ingestion into HDFS (NiFi + HDFS)

- **Screenshot 1:** NiFi canvas showing the Final_Project process group with the three processors: *Download File (InvokeHTTP)*, *Update File Name (UpdateAttribute)*, and *Write File to HDFS (PutHDFS)*, along with the labeled steps.

- 

NiFi Parameter Contexts

Name	Provider	Description
Final Project		DOWNLOAD FILE URL = https://raw.githubusercont...
My First NiFi Flow		
Syslog Processing		

Update Parameter Context

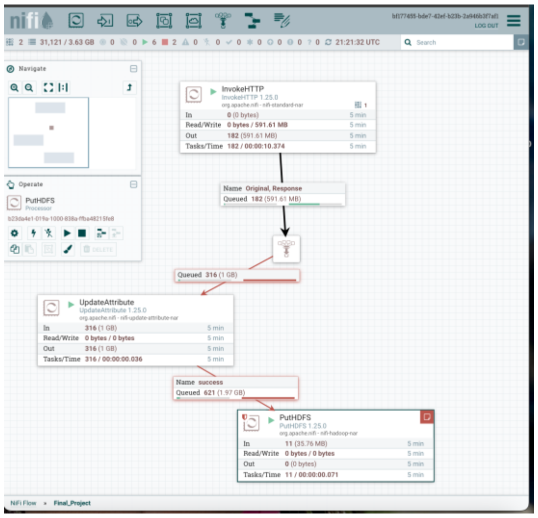
Name	Value
DOWNLOAD FILE URL	https://raw.githubusercontent.co...
FILENAME	netflix_titles.csv
HDFS WRITE DIRECTORY	/tmp
USER	root

Parameter
DOWNLOAD FILE URL
Referencing Components
None

Last updated: 18:23:20 UTC

NiFi Flow » Final_Project

- **Screenshot 2:** NiFi flow running with data visible in the queues between processors, indicating successful transfer of flowfiles.

- 

InvokeHTTP
InvokeHTTP 1.25.0
org.apache.nifi - nifi-standard-nar

In	0 (0 bytes)	5 min
Read/Write	0 bytes / 591.61 MB	5 min
Out	182 (591.61 MB)	5 min
Tasks/Time	182 / 00:00:10.374	5 min

Name Original, Response
Queued 182 (591.61 MB)

Queued 316 (1 GB)

UpdateAttribute
UpdateAttribute 1.25.0
org.apache.nifi - nifi-update-attribute-nar

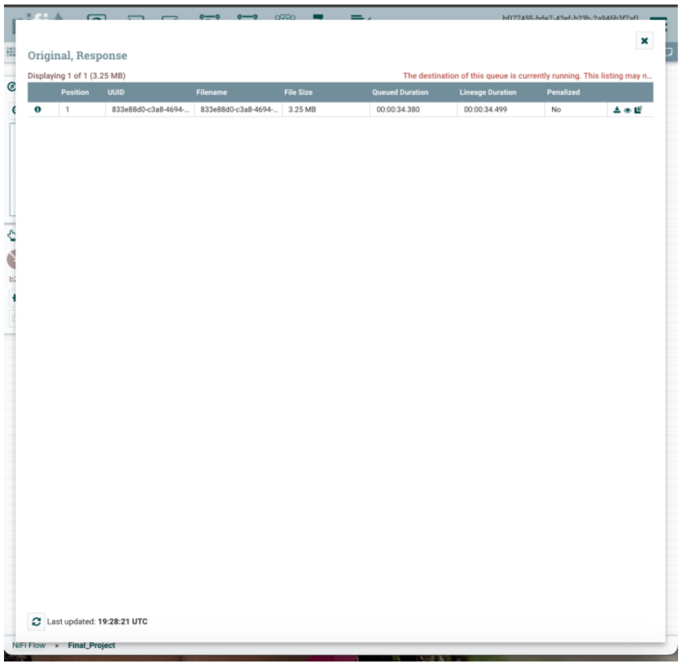
In	316 (1 GB)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	316 (1 GB)	5 min
Tasks/Time	316 / 00:00:00.036	5 min

Name success
Queued 621 (1.97 GB)

PutHDFS
PutHDFS 1.25.0
org.apache.nifi - nifi-hadoop-nar

In	11 (35.76 MB)	5 min
Read/Write	0 bytes / 0 bytes	5 min
Out	0 (0 bytes)	5 min
Tasks/Time	11 / 00:00:00.071	5 min

2 31,121 / 3.63 GB 0 0 6 2 0 0 0 0 0 0 0 21:21:32 UTC
bf177455-bde7-42ef-b23b-2a946b3f7af1
LOG OUT
Search
Navigate
Operate
PutHDFS
Processor
b23da4e1-019a-1000-838a-ffba48215fe8
DELETE
NiFi Flow » Final_Project

- 

Original, Response

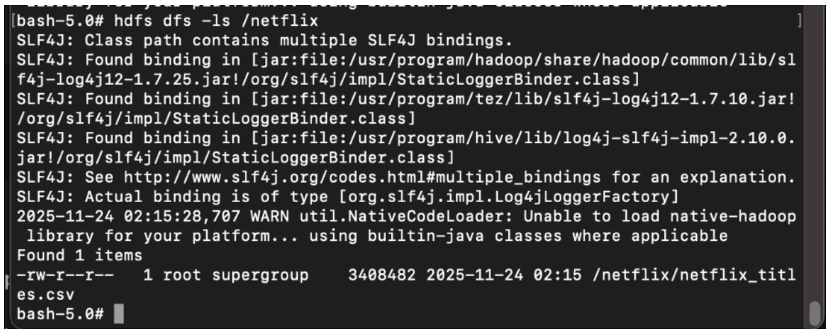
Displaying 1 of 1 (3.25 MB)

The destination of this queue is currently running. This listing may n...

	Position	UUID	Filename	File Size	Queued Duration	Lineage Duration	Penalized	
	1	833e88d0-c3a8-4694-...	833e88d0-c3a8-4694-...	3.25 MB	00:00:34.380	00:00:34.499	No	

Last updated: 19:28:21 UTC

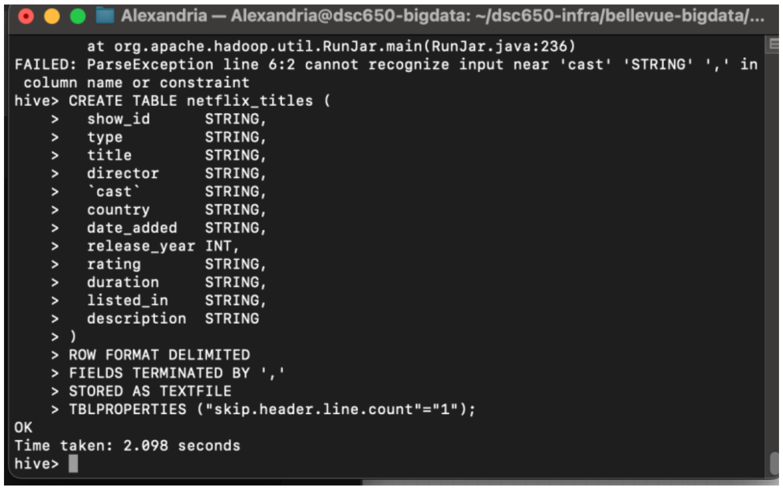
- **Screenshot 3:** Terminal output of hdfs dfs -ls /data/netflix (or the chosen directory) showing netflix_titles.csv present in HDFS.

- 

```
bash-5.0# hdfs dfs -ls /netflix
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/program/hadoop/share/hadoop/common/lib/sl
f4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/program/tez/lib/slf4j-log4j12-1.7.10.jar!
/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/program/hive/lib/log4j-slf4j-impl-2.10.0.
jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
2025-11-24 02:15:28,707 WARN util.NativeCodeLoader: Unable to load native-hadoop
 library for your platform... using builtin-java classes where applicable
Found 1 items
-rw-r--r--   1 root supergroup    3408482 2025-11-24 02:15 /netflix/netflix_titl
es.csv
bash-5.0# 
```

Objective 2 – Hive Managed Table (Hive)

- **Screenshot 4:** Hive shell with the CREATE TABLE netflix_titles statement.

- 
Alexandria — Alexandria@dsc650-bigdata: ~/dsc650-infra/bellevue-bigdata/...

```
        at org.apache.hadoop.util.RunJar.main(RunJar.java:236)
FAILED: ParseException line 6:2 cannot recognize input near 'cast' 'STRING' ',' in
 column name or constraint
hive> CREATE TABLE netflix_titles (
    >   show_id      STRING,
    >   type         STRING,
    >   title        STRING,
    >   director     STRING,
    >   `cast`       STRING,
    >   country      STRING,
    >   date_added   STRING,
    >   release_year INT,
    >   rating       STRING,
    >   duration     STRING,
    >   listed_in    STRING,
    >   description  STRING
    > )
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY ','
    > STORED AS TEXTFILE
    > TBLPROPERTIES ("skip.header.line.count"="1");
OK
Time taken: 2.098 seconds
hive> 
```

- **Screenshot 5:** Query output from SELECT * FROM netflix_titles LIMIT 10; showing properly loaded rows.

- **Screenshot 6:** Aggregation query, for example:
  SELECT type, rating, COUNT(*) AS cnt FROM netflix_titles GROUP BY type, rating ORDER BY cnt DESC LIMIT 10;

Objective 3 – Environment Setup (Python Libraries + Thrift)

- **Screenshot 7:** pip3 install numpy happybase commands and successful installation output in worker1, worker2, and master containers.

HDFS_Hive_Spark_HBase_Pipeline

- **Screenshot 8:** nohup hbase thrift start & command running on the master container, confirming the Thrift server is active.

Objective 4 – HBase Table Creation (HBase)

- **Screenshot 9:** HBase shell commands showing create 'netflix_metrics', 'cf'.

- **Screenshot 10:** scan 'netflix_metrics' immediately after creation, showing an empty result set.

Objective 5 – PySpark ML Code (Spark MLlib)

- **Screenshot 11:** Terminal window showing the netflix_ml.py script running and printing training and evaluation output.
- **Screenshot 12:** Console output showing metrics such as AUC and accuracy.

Objective 6 – Spark Submit & Output (YARN)

- **Screenshot 13:** The spark-submit command used to run the job on YARN (e.g., spark-submit --master yarn --deploy-mode client --name NetflixTypeClassification netflix_ml.py).
- **Screenshot 14:** Snippet of YARN logs or driver logs showing the job successfully finishing.

Objective 7 – HBase Scan Verification (HBase)

- **Screenshot 15:** HBase shell scan 'netflix_metrics' showing the row key (e.g., metrics1) and columns cf:auc and cf:accuracy populated with the Spark-computed values.

Code

This section summarizes the key code used in the project: NiFi template usage, Hive DDL and queries, Spark MLlib script, and HBase commands. The full versions of these scripts are also stored in my GitHub repository, as required.

1. NiFi Template and Parameters (Final_Project.json)

I used the provided Final_Project.json NiFi template without structural changes. The main configuration was done via **Parameter Context** named “Final Project” with the following parameters:

Final_Project

- DOWNLOAD FILE URL = https://raw.githubusercontent.com/taylorduncan/dsc650/refs/heads/main/netflix_titles.csv
- FILENAME = netflix_titles.csv
- HDFS WRITE DIRECTORY = /data/netflix
- USER = output of whoami on the master VM

The flow order is:

1. **Download File (InvokeHTTP)** – GET request to #{DOWNLOAD FILE URL}, generating a flowfile with the CSV content.

2. **Update File Name (UpdateAttribute)** – sets filename to #{FILENAME} so the output file has a clear name.

3. **Write File to HDFS (PutHDFS)** – writes the flowfile to #{HDFS WRITE DIRECTORY} using Hadoop configuration from /home/#{USER}/dsc650-infra/bellevue-bigdata/nifi/hadoopconf/core-site.xml.

2. Hive DDL and Sample Queries

Create Managed Table

```
CREATE TABLE netflix_titles (
   show_id      STRING,
   type         STRING,
   title        STRING,
   director     STRING,
   cast         STRING,
   country      STRING,
   date_added   STRING,
   release_year INT,
   rating       STRING,
   duration     STRING,
   listed_in    STRING,
   description  STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE
TBLPROPERTIES ("skip.header.line.count"="1");
```

Load Data from HDFS into Hive

```
LOAD DATA INPATH '/data/netflix/netflix_titles.csv'
INTO TABLE netflix_titles;
```

Aggregation Query (Schema Validation)

```
SELECT type, rating, COUNT(*) AS cnt
FROM netflix_titles
```

```
GROUP BY type, rating
ORDER BY cnt DESC
LIMIT 10;
```

3. HBase Table and Thrift Setup

Create HBase Table

```
hbase shell

create 'netflix_metrics', 'cf'

scan 'netflix_metrics'  -- initially empty
exit
```

Start Thrift Server (Master Container)

```
nohup hbase thrift start &
```

This setup follows the example process from the course walkthrough, but with a custom table name for Netflix metrics.

HDFS_Hive_Spark_HBase_Pipeline

4. PySpark ML Script (netflix_ml.py)

Below is a condensed version of the PySpark script used for model training and HBase writes. It predicts whether a title is a movie or TV show (type) using features derived from release_year, duration, rating, and listed_in:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import regexp_extract, col, when
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator,
MulticlassClassificationEvaluator
import happybase

# 1. Spark Session with Hive support
spark = SparkSession.builder \
    .appName("NetflixTypeClassification") \
    .enableHiveSupport() \
    .getOrCreate()

# 2. Load data from Hive
df = spark.sql("""
    SELECT type, rating, listed_in, release_year, duration
    FROM netflix_titles
""")

# 3. Parse duration into integer (minutes or seasons as a count)
df = df.withColumn    "duration_int",
regexp_extract(col("duration"), r'(\\d+)', 1).cast("int")
)

# Drop rows with missing critical fields
df = df.na.drop(subset=["type", "rating", "listed_in", "release_year", "duration_int"])

# 4. Index and encode categorical features
label_indexer = StringIndexer(inputCol="type", outputCol="label", handleInvalid="keep")

cat_cols = ["rating", "listed_in"]
indexers = [    StringIndexer(inputCol=c, outputCol=c + "_idx", handleInvalid="keep")
    for c in cat_cols
]

encoder = OneHotEncoder(
    inputCols=[c + "_idx" for c in cat_cols],
    outputCols=[c + "_oh" for c in cat_cols]
)

# 5. Assemble features
assembler = VectorAssembler(
    inputCols=["release_year", "duration_int"] + [c + "_oh" for c in cat_cols],
    outputCol="features"
)

# 6. Logistic Regression model
lr = LogisticRegression(featuresCol="features", labelCol="label")

# 7. Build pipeline
pipeline = Pipeline(stages=[label_indexer] + indexers + [encoder, assembler, lr])

# 8. Train/test split
train, test = df.randomSplit([0.7, 0.3], seed=42)
```

```
# 9. Fit model and evaluate
model = pipeline.fit(train)
predictions = model.transform(test)

binary_eval = BinaryClassificationEvaluator(
    labelCol="label",
    rawPredictionCol="rawPrediction",
    metricName="areaUnderROC"
)
multiclass_eval = MulticlassClassificationEvaluator(
    labelCol="label",
    predictionCol="prediction",
    metricName="accuracy"
)

auc = binary_eval.evaluate(predictions)
accuracy = multiclass_eval.evaluate(predictions)

print(f"AUC: {auc}")
print(f"Accuracy: {accuracy}")

# 10. Write metrics to HBase via happybase
metrics_data = [
    ("metrics1", "cf:auc", str(auc)),
    ("metrics1", "cf:accuracy", str(accuracy))
]

def write_to_hbase_partition(partition):
    connection = happybase.Connection('master')
    connection.open()
    table = connection.table('netflix_metrics')
    for row_key, column, value in partition:
        table.put(row_key, {column: value.encode("utf-8")})
    connection.close()

rdd = spark.sparkContext.parallelize(metrics_data)
rdd.foreachPartition(write_to_hbase_partition)

spark.stop()
```
This script follows the same general pattern as the provided MLlib example in the course materials (Spark session → Hive read → feature engineering → model training → metrics → HBase write), but it is customized to the Netflix dataset and uses a classification algorithm (LogisticRegression) that makes sense for predicting type.

HDFS_Hive_Spark_HBase_Pipeline

Spark Submit Command

```
spark-submit \
  --master yarn \
  --deploy-mode client \
  --name NetflixTypeClassification \
  netflix_ml.py
```

Conclusion

This project demonstrates a complete, working big data and machine learning pipeline built on top of the Hadoop ecosystem. Starting from a GitHub-hosted Netflix titles dataset, I used **Apache NiFi** to automate ingestion into **HDFS**, **Hive** to structure and query the data, **Spark MLlib** to build and evaluate a binary classification model, and **HBase** to store the model's evaluation metrics for fast retrieval. Each step maps directly to the course's final project objectives, from NiFi flow design and HDFS confirmation through Hive schema design, environment setup, HBase integration, Spark ML training, and final HBase verification.

Beyond satisfying the rubric, the project highlights how these technologies can be combined in real-world scenarios where data needs to flow reliably from ingestion to insight. I gained practical experience configuring multi-component services, debugging distributed jobs, and applying machine learning in a scalable way using Spark. The resulting pipeline is modular and extensible: the same flow could be adapted to new datasets, additional features, or different models simply by adjusting the NiFi parameters, Hive DDL, and Spark script. Completing this project has strengthened my understanding of modern data engineering and machine learning platforms and has prepared me to work with similar end-to-end architectures in professional environments.