Module Name: Big Data Analytics and Data Visualisation

Module Code: 7153CEM

Assignment Title: Dataset Analysis and Visualization Using Big Data Program

Student Name: Yvonne Musinguzi

Student ID: 15094816

Abstract

This project explores classification of Netflix content as either a movie or TV show based on their descriptions. Exploratory Data Analysis (EDA) was conducted in Tableau to extract insights, followed by text preprocessing using tokenization, count vectorization, and IDF for feature extraction. Multiple classification models—Logistic Regression, Random Forest, Naive Bayes, Decision Tree, Gradient Boosting, and Support Vector Classifier—were trained and evaluated using accuracy, precision, recall, and F1 score. To address class imbalance , oversampling was applied. Hyperparameter tuning and 10-fold cross-validation were also used to optimize performance. The project highlights model comparisons and suggests improvements for future work, demonstrating the value of machine learning in content categorization for recommendation systems and media management.

Introduction and Background

Digital streaming platforms like Netflix use machine learning to personalize recommendations and enhance user experience (Gomez-Uribe & Hunt, 2015). Traditional systems focus on

metadata such as genre and ratings but often overlook the value in textual descriptions of content.

This project explores how textual descriptions can classify content as a movie or TV show, using natural language processing (NLP) techniques. By moving beyond metadata, it shows how NLP can improve content classification (Fucci et al., 2022), contributing to the growing field of content-aware machine learning models (Johnson & Zhang, 2017).

Relevance and Motivation for the Project

Motivated by Netflix's use of content descriptions, this project investigates whether descriptions alone can classify content. The study challenges metadata-based methods, demonstrating how text-based models can enhance classification tasks.

The project aligns with the growing interest in leveraging text-based metadata for context-aware classification and recommendations. By improving text understanding, the study aims to enhance personalized user experiences and inspire future research on automated decision-making in content management (Johnson & Zhang, 2017)

Related Work

Text classification, a fundamental task in Natural Language Processing (NLP), entails assigning predefined categories to unstructured text. Early approaches, including Naive Bayes and Support Vector Machines (SVMs) combined with feature engineering techniques like Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF), established the groundwork for effective text categorization. (Allam et al., 2025) emphasize these foundational methods in their paper, highlighting their significance in the evolution of machine learning for text classification.

(Mienye & Swart, 2024) explore the evolution and applications of deep learning models, focusing on foundational architectures like CNNs, RNNs, and LSTMs. They also examine the transformative impact of transformer-based models, particularly BERT, highlighting its role in improving text classification tasks. Hence, the review emphasizes the significant advancements in NLP and text categorization driven by these models.

The adoption of word embeddings, such as word2vec (Mikolov et al., 2013), marked a significant advancement in Natural Language Processing (NLP). Word2vec uses two model architectures, Continuous Bag of Words (CBOW) and Skip-gram, to predict words within a context, thereby capturing semantic relationships between words. This method maps words into a continuous vector space where semantically similar words are positioned closer together. The efficiency and performance of word2vec allow platforms like Netflix to process large datasets quickly and interpret text descriptions more accurately. This enhanced semantic understanding

leads to more personalized and relevant content recommendations, significantly improving user experience and engagement on streaming platforms.

While many studies have advanced genre prediction and sentiment analysis, fewer have focused on using textual metadata alone to classify content types like movies and TV shows. Models often combine textual descriptions with metadata (e.g., duration, cast). For instance, (Kumar et al., 2022) used plot summaries and other features for genre classification, without isolating descriptions.

This project fills the gap by exploring whether textual descriptions alone can predict content type. By applying advanced NLP techniques, this research contributes to content classification, personalization, and recommendation systems, aligning with trends in deep learning, contextual embeddings, and multi-modal learning in NLP.

Dataset

The Netflix Titles Dataset was obtained from Kaggle (Bansal, 2021), and it contains 8809 unique values (rows) and 12 features (columns) with majority having a string data type, one with data type and one numeric as seen in *Table 1* below

Table 1: Data Features and Types

Feature	**Type**	**Description**
show_id	String	A unique ID for each Movie/TV show
type	String	If it's a Movie/TV show
title	String	The name of the Movie or TV show
director	String	The director of Movie or TV show
cast	String	The cast (actors) of the Movie or TV show
country	String	The country from which the Movie or Show originates
date_added	Date	The date when the Movie or TV show was added to Netflix
release_year	Numeric	The Year when the Movie or TV show was first released
rating	String	How the Movie /TV show is rated (age group it belongs to) i.e. PG, PG-13
duration	String	The length of the Movie/TV show. Movie in minutes TV Show in seasons
listed_in	String	The Genre to which the Movie/TV show belong
description	String	A short summary of what the Movie/ TV show is about.

Exploratory Data Analysis and Visualization using Tableau

Class Imbalance in TV Shows and Movies (Packed Bubbles)

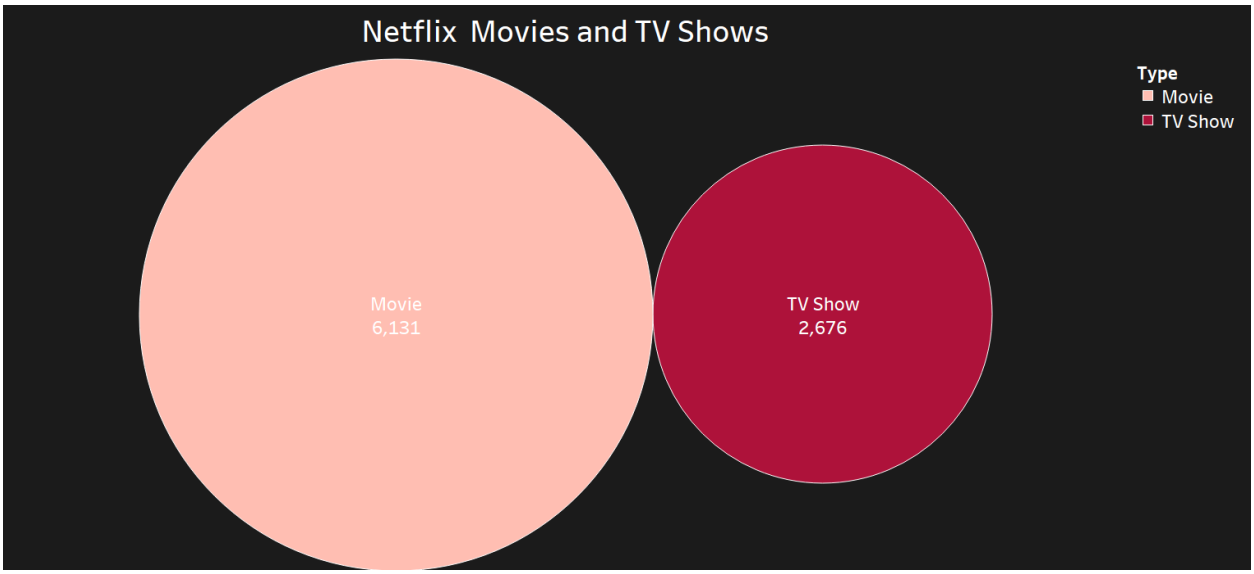


Figure 1: Class Imbalance in TV Shows and Movies (Packed Bubbles)

Figure 1 shows a packed bubbles visualization that highlights the class imbalance between TV shows and movies in the dataset. Each bubble represents a category (TV shows or movies), with bubble size showing the number of entries in each class. The larger bubbles indicate an overrepresentation of movies compared to TV shows.

To address this imbalance, oversampling techniques are applied to ensure both classes are equally represented before training the models. This will help improve the model's ability to learn from both categories and prevent biased predictions.

Movies and TV Shows Over the Years (Line Graph with Area)

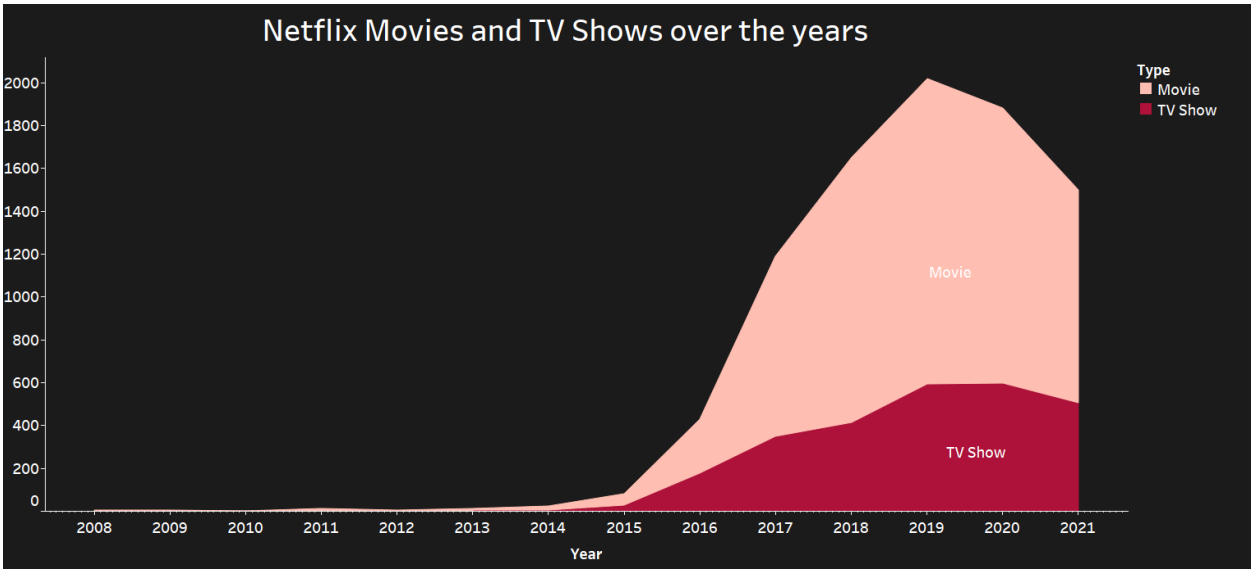


Figure 2: Movies and TV Shows Over the Years (Line Graph with Area)

Figure 2 shows The line graph with area shading shows the trends in Movie and TV Show releases over the years.

Key observations:

- Movie releases have steadily increased, outpacing TV Shows.
- TV Show releases have risen but remain lower than Movies.

This trend highlights changing media consumption patterns, useful for content classification or recommendation models.

Movie Duration Distribution

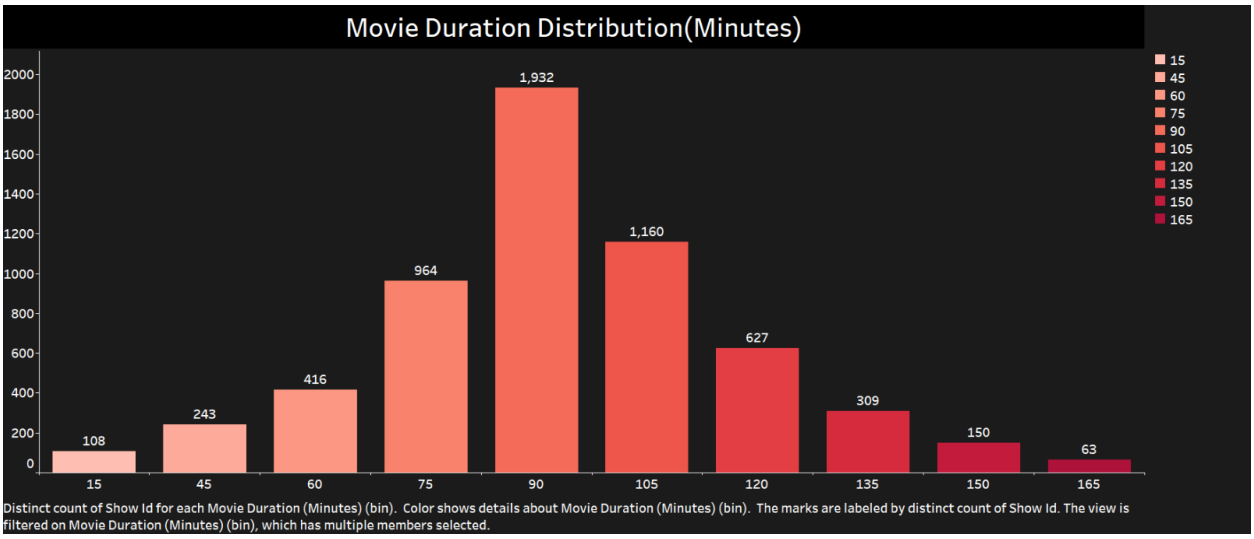


Figure 3: Movie Duration Distribution (Histogram)

This bar plot (*Figure 3*) shows movie duration distribution after preprocessing. Durations were binned into 15-minute intervals.

Key observations:

- Most movies are around 90 minutes.
- Few movies have extreme durations.

This reflects typical movie length and is useful for recommendation systems.

Top Ten Actors by Content

Figure 4: Top Ten Actors by Content

Figure 4 shows a visualization of the top ten actors based on their total appearances in TV shows and movies, with their pictures included. The "cast" feature was split using Python and analyzed in Tableau.

Key points:

- Actors are ranked by their total appearances.
- The x-axis shows actor names, and the y-axis shows their total number of appearances.
- Images help viewers quickly recognize them.

This provides insight into the most prominent actors in the entertainment industry. It is evident that they are mostly from the same Country / Geographic region, and this would imply that that region potentially produces many movies/TV shows featured on Netflix, but further analysis will be done to confirm this.

Countries Producing Most Netflix Content

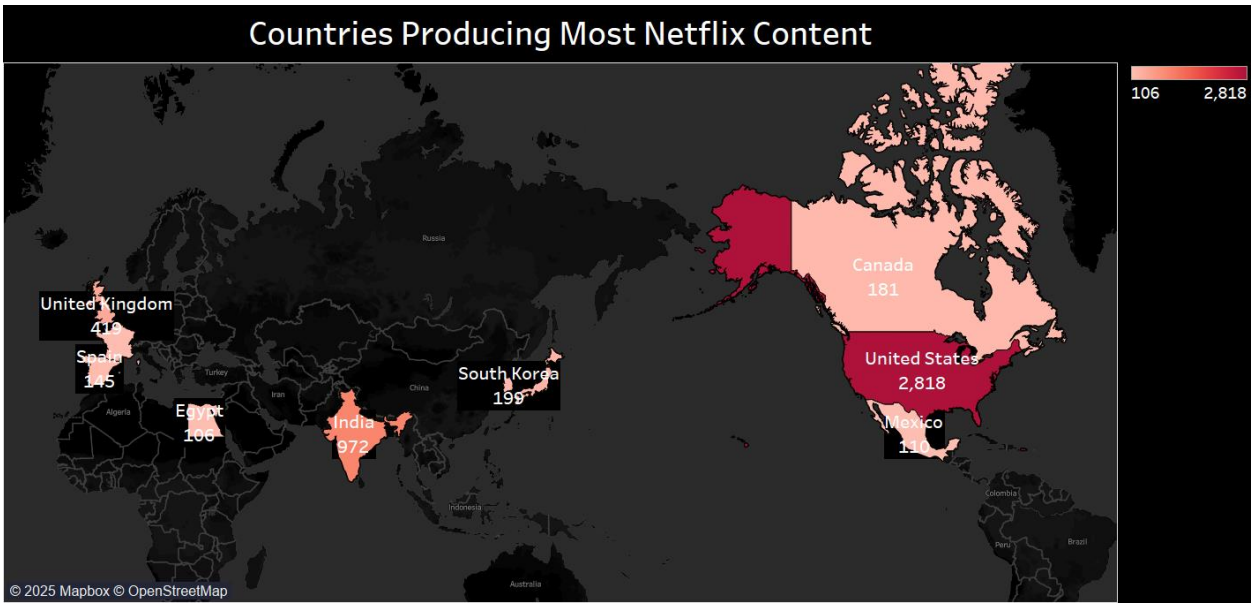


Figure 5: Countries Producing Most Netflix Content

Figure 5 displays a map of countries producing the most Netflix content, with the USA leading, followed by India. This trend likely correlates with *Figure 4*, which highlights the top 10 actors featured on Netflix—many of whom are predominantly associated India

Total Movies/TV Shows by Rating

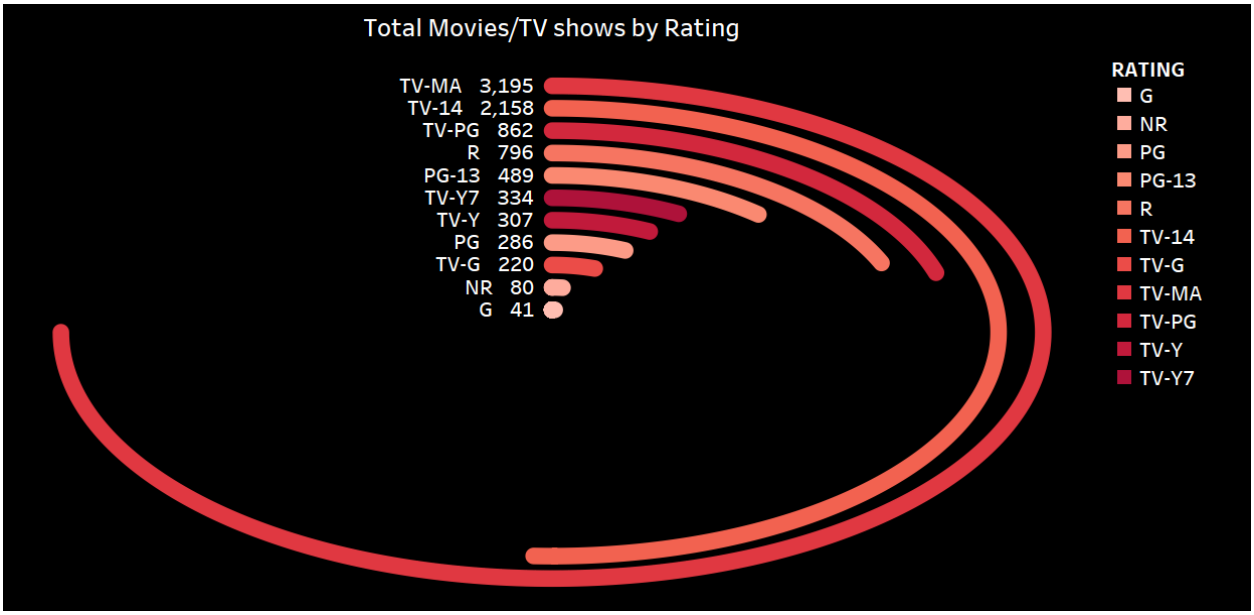


Figure 6: Total Movies/TV Shows by Rating

Figure 6 is a radial bar chart showing the total number of Movies and TV Shows across different rating categories. The most frequent rating is TV-MA, followed by TV-14, indicating a large portion of Netflix content is geared towards mature and teen audiences. The radial format was chosen for its visual appeal and ease of comparison.

Word Cloud of Most Common Words in Descriptions

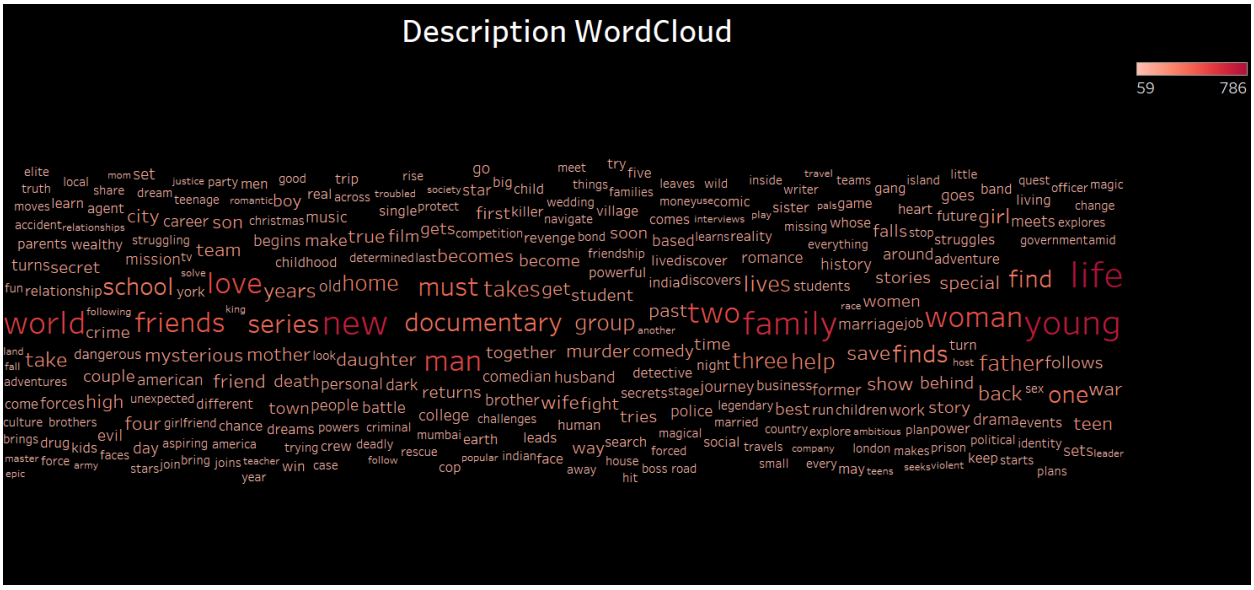


Figure 7: Word Cloud of Most Common Words in Descriptions

The word cloud highlights the most frequent words from the "description" column, with larger words appearing more often. Words like "family" and "life" suggest that many shows and movies focus on family dynamics and life experiences. Since I'm interested in using descriptions for classification, this visualization provides valuable insights into common themes that could aid in distinguishing between different types of content. narratives, which might be a key characteristic of the dataset. The word cloud thus highlights the common thematic elements, offering a quick visual insight into the nature of the content in the dataset.

Data Processing Steps

Objective: The goal of this project was to classify content as either a Movie or TV Show based on the description provided. The following preprocessing steps were applied to prepare the dataset for machine learning:

1. Column Selection and Null Handling: Only the type (Movie/TV Show) and description columns were selected.Rows with null values in either column were removed to ensure a clean dataset.

2. Initial Exploration: The dataset was grouped by the type column to examine class distribution.

3. Text Cleaning: The description column was cleaned with several steps:

    - Text was converted to lowercase for uniformity.
    - Non-alphanumeric characters (except spaces) were removed to standardize the input.

4. Feature Engineering and Preprocessing Pipeline

    - Tokenizer: Split descriptions into individual words.
    - StopWordsRemover: Removed common stop words to focus on meaningful words.
    - CountVectorizer: Converted words into numerical feature vectors.
    - IDF (Inverse Document Frequency): Applied IDF to reduce the influence of frequently occurring words.
    - StringIndexer: Encoded the type column (Movie/TV Show) into numerical labels for classification.

5. Train-Test Split: The dataset was split into training (80%) and testing (20%) subsets for model development and performance evaluation.

6. Addressing Class Imbalance: To address the imbalance between Movies and TV Shows, the minority class (TV Shows) from the training sample was oversampled to match the majority class (Movies).

Methodology / Experimental Set Up

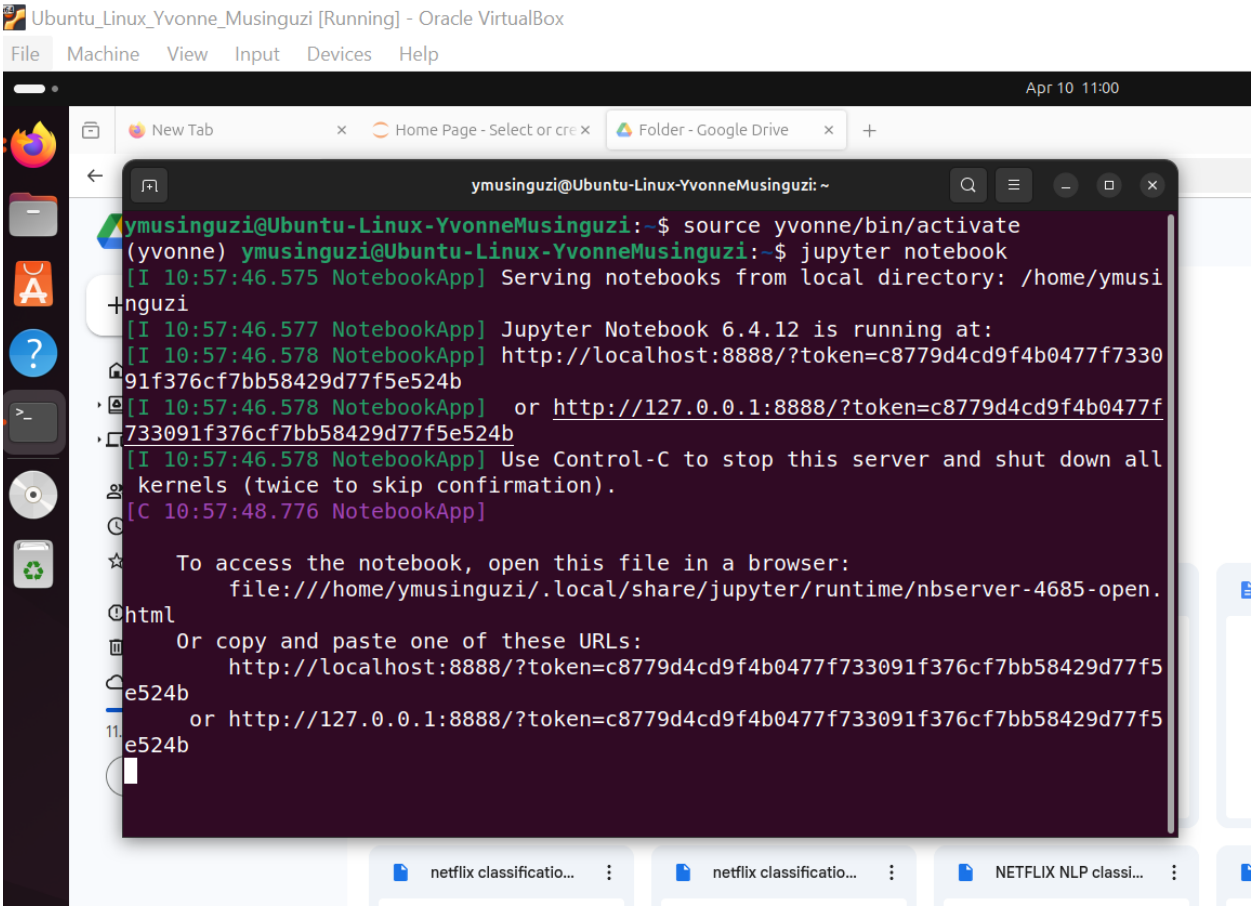
This project utilizes PySpark as the primary tool for data processing and model training. The work was carried out on both Linux and Windows environments, allowing for flexibility and seamless integration across platforms.

Environment Setup

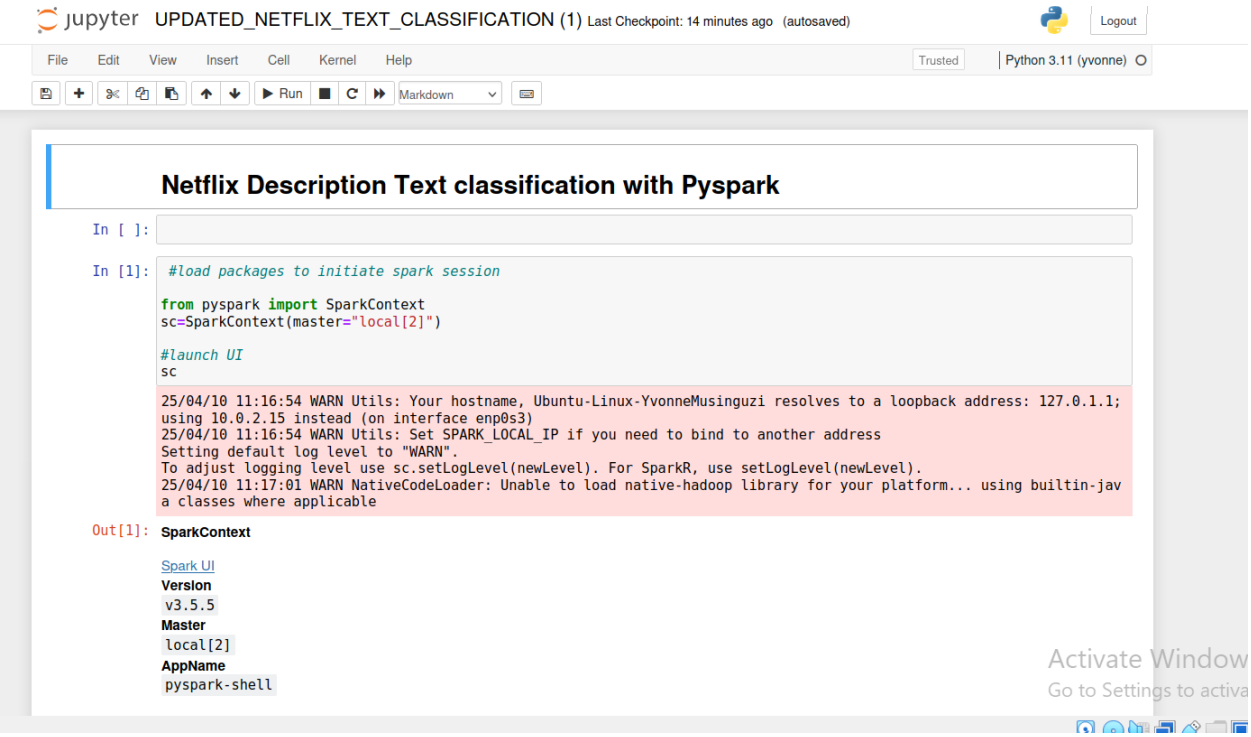
The project was developed in a virtual environment named ***(yvonne)*** on Linux, ensuring isolated dependencies. Key steps included:

- Installing PySpark and Jupyter Notebooks in the virtual environment.
- Activating the environment and using Jupyter for data processing and machine learning tasks.

This setup ensured the correct Python packages were used throughout the project.

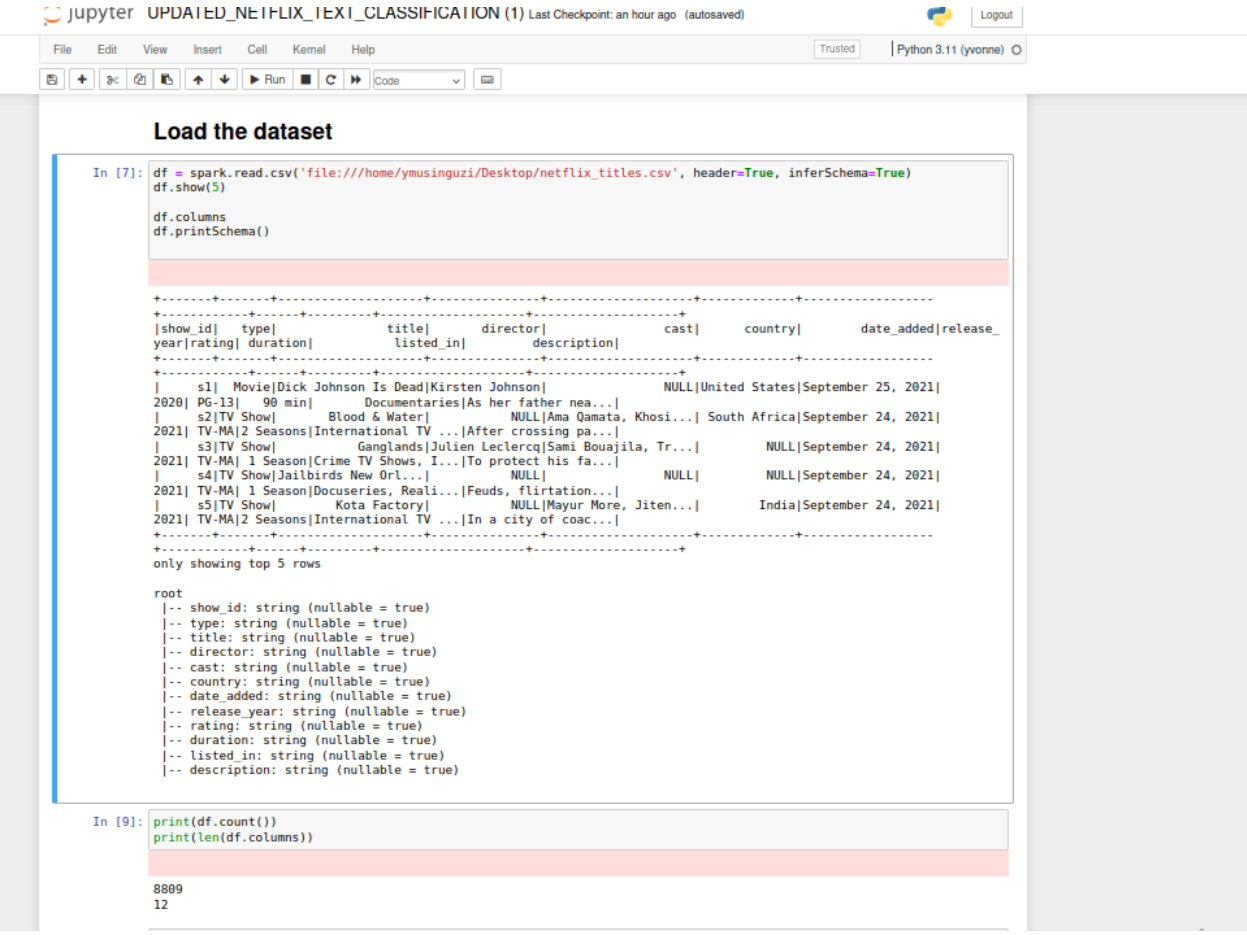


Setting up Spark Session

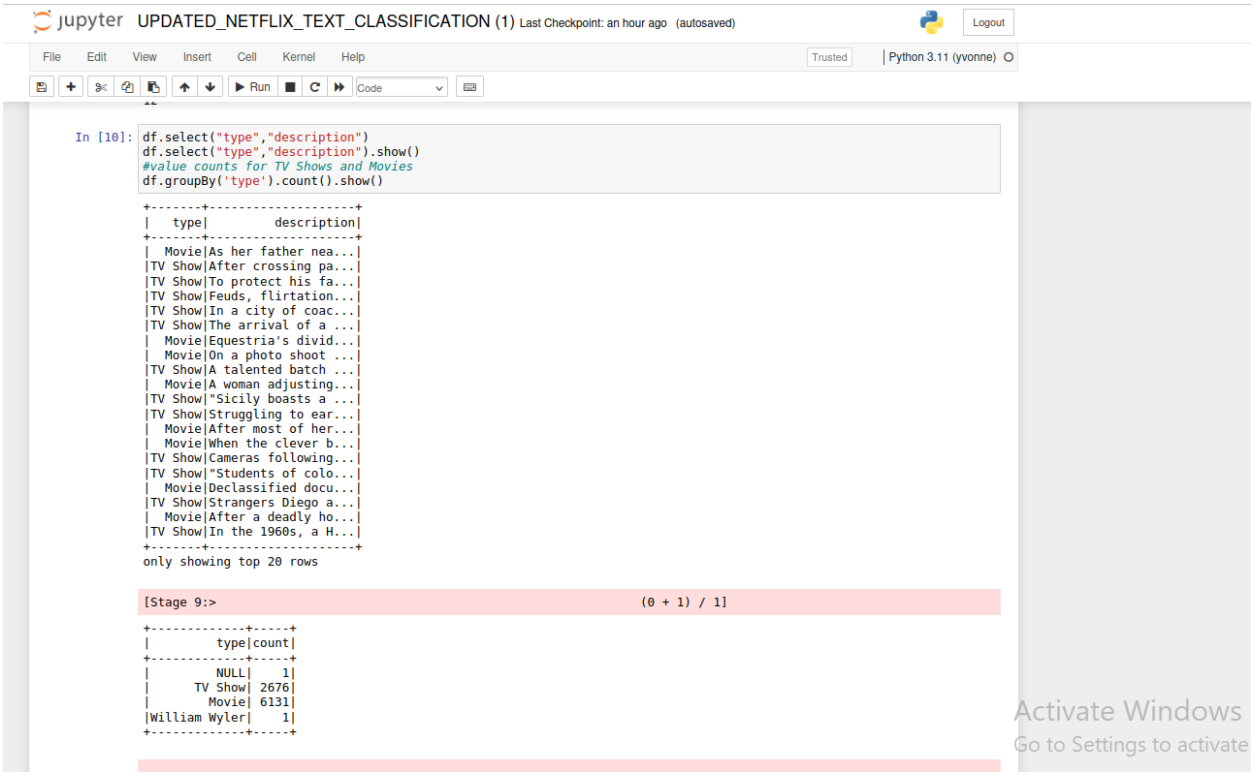


Data Preprocessing and Cleaning

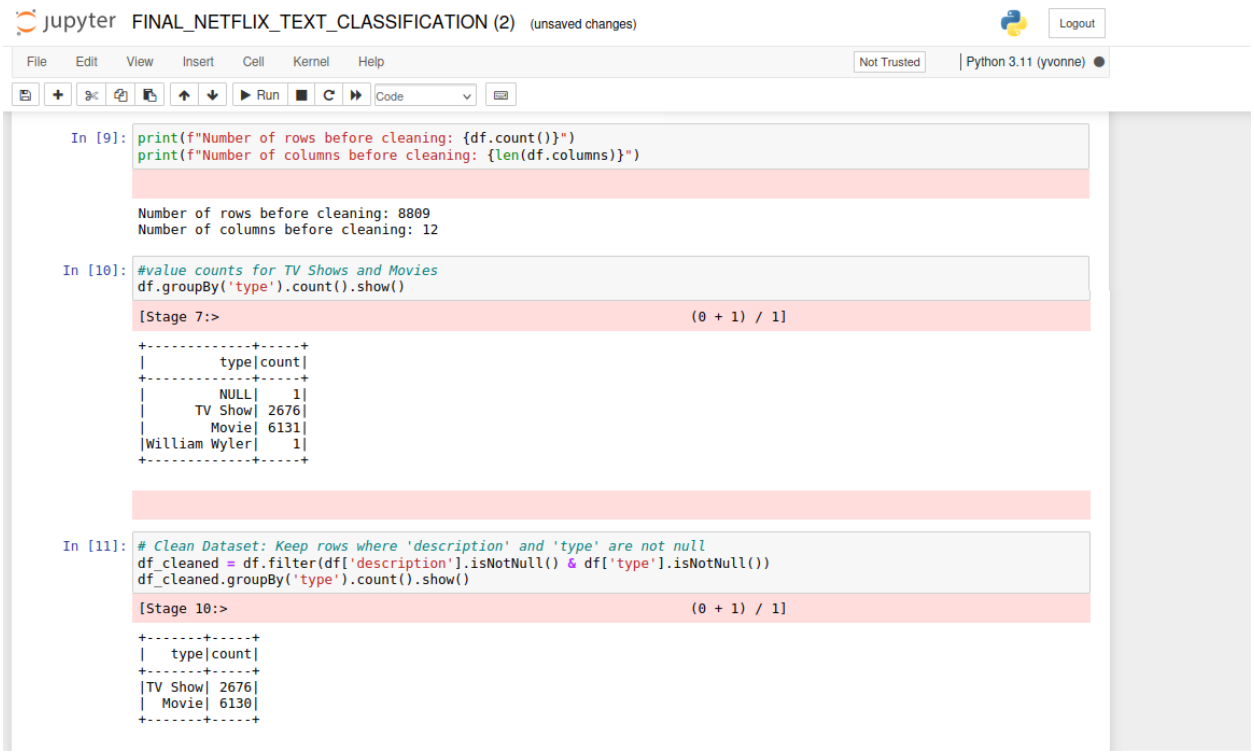
Loading of the dataset :The goal is to clean and prepare the Netflix dataset for classification. The dataset was loaded into PySpark using spark.read.csv, which automatically inferred the schema of the data.



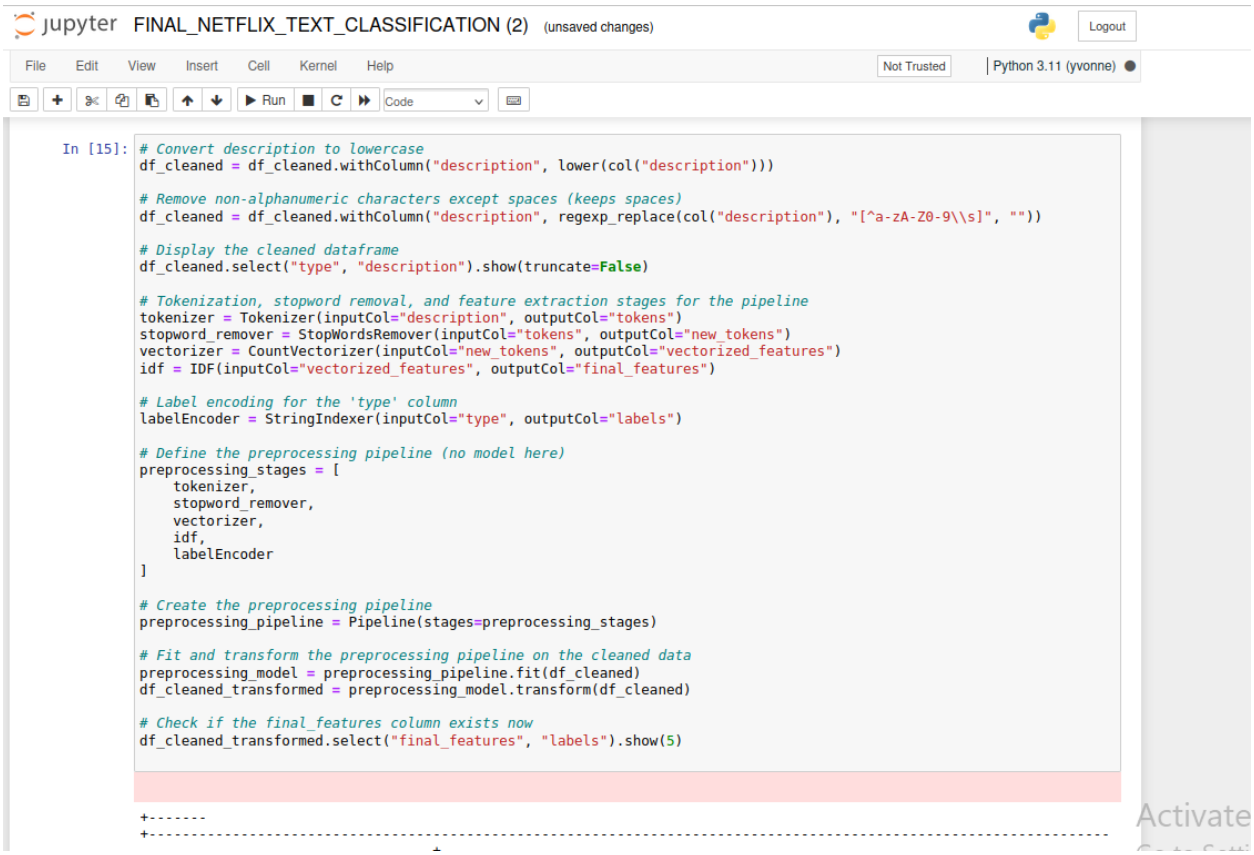
- Exploration and Inspection: The first few records of the dataset were displayed to inspect the columns and values.



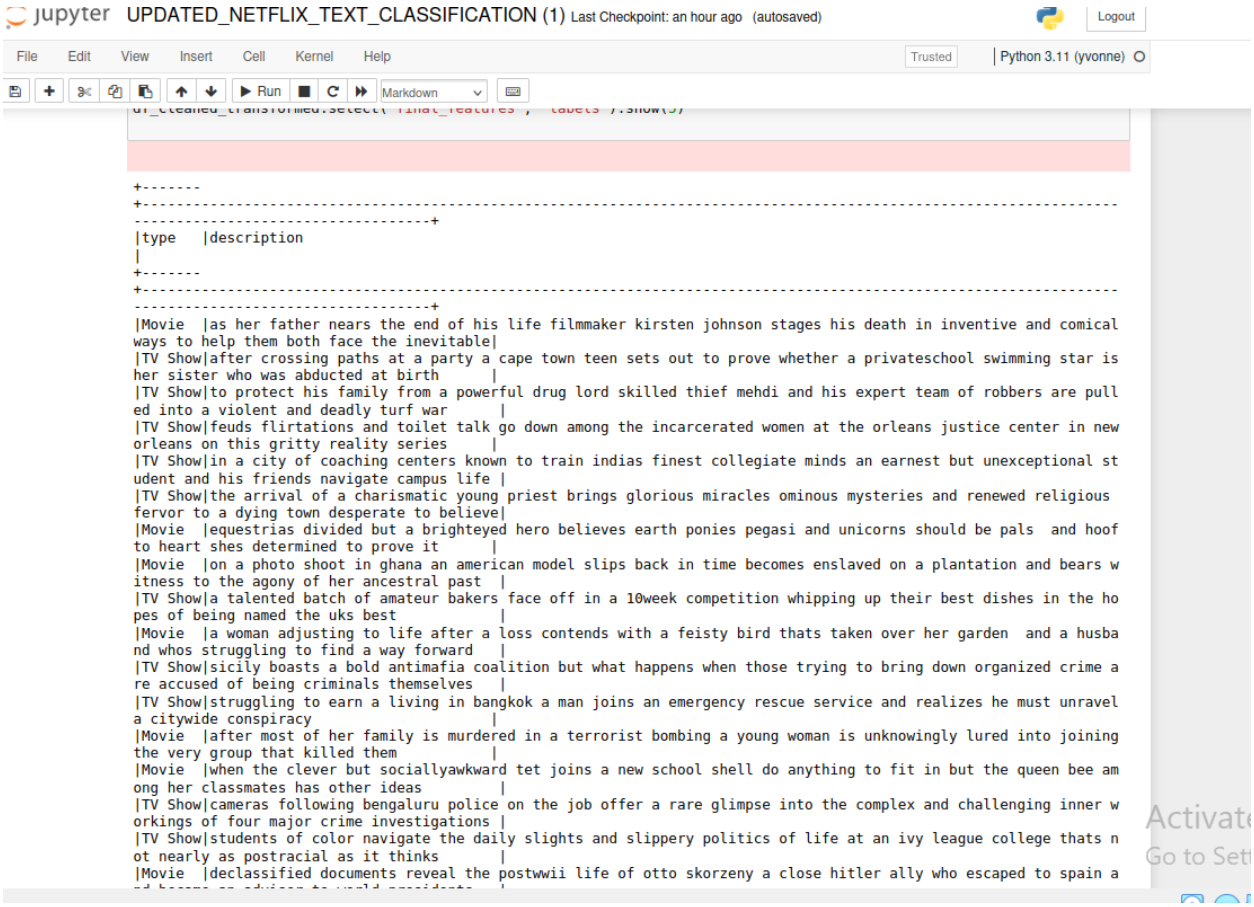
- Cleaning the Data: Data cleaning focused on the type and description columns. The goal was to filter out rows where either of these columns contained null values. It is evident that the classes are imbalanced.



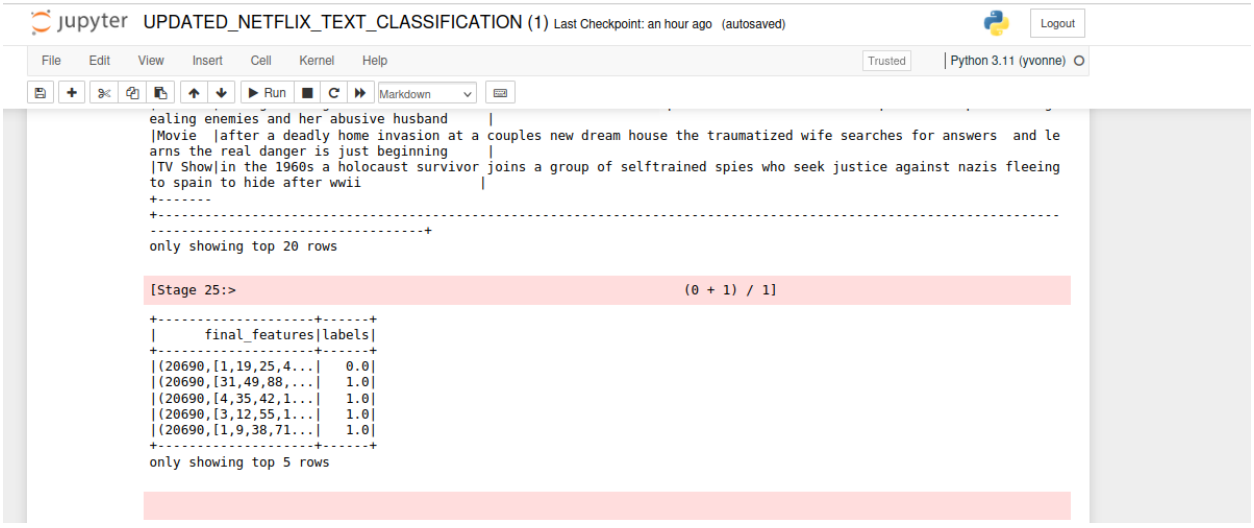
Feature Extraction



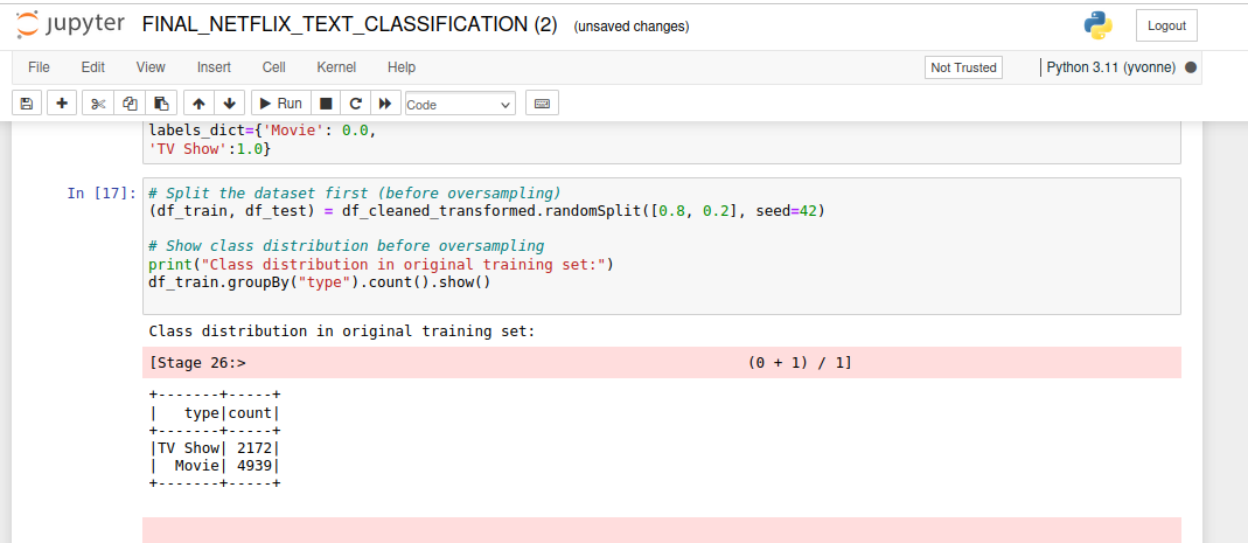
- The description column was cleaned by converting text to lowercase and removing non-alphanumeric characters to standardize it for analysis.
- Descriptions were split into tokens (individual words) using the Tokenizer.
- Common stop words (e.g., "the", "and") were removed using the StopWordsRemover.
- A CountVectorizer converted tokens into numerical feature vectors.
- IDF was applied to weigh the importance of terms.
- Label Encoding: The StringIndexer converted the type column (target variable) into numerical labels for classification.



Final Features

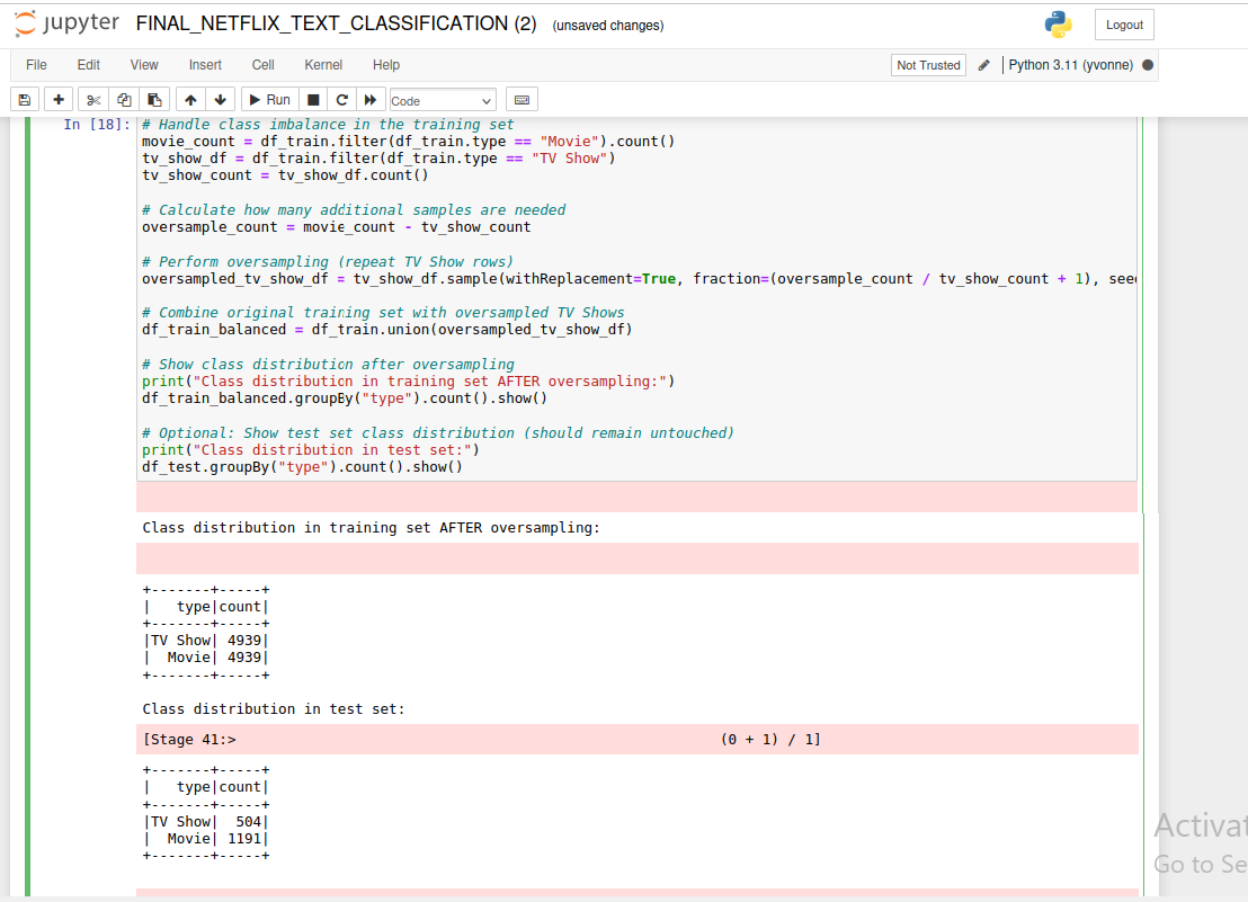


Splitting the Data

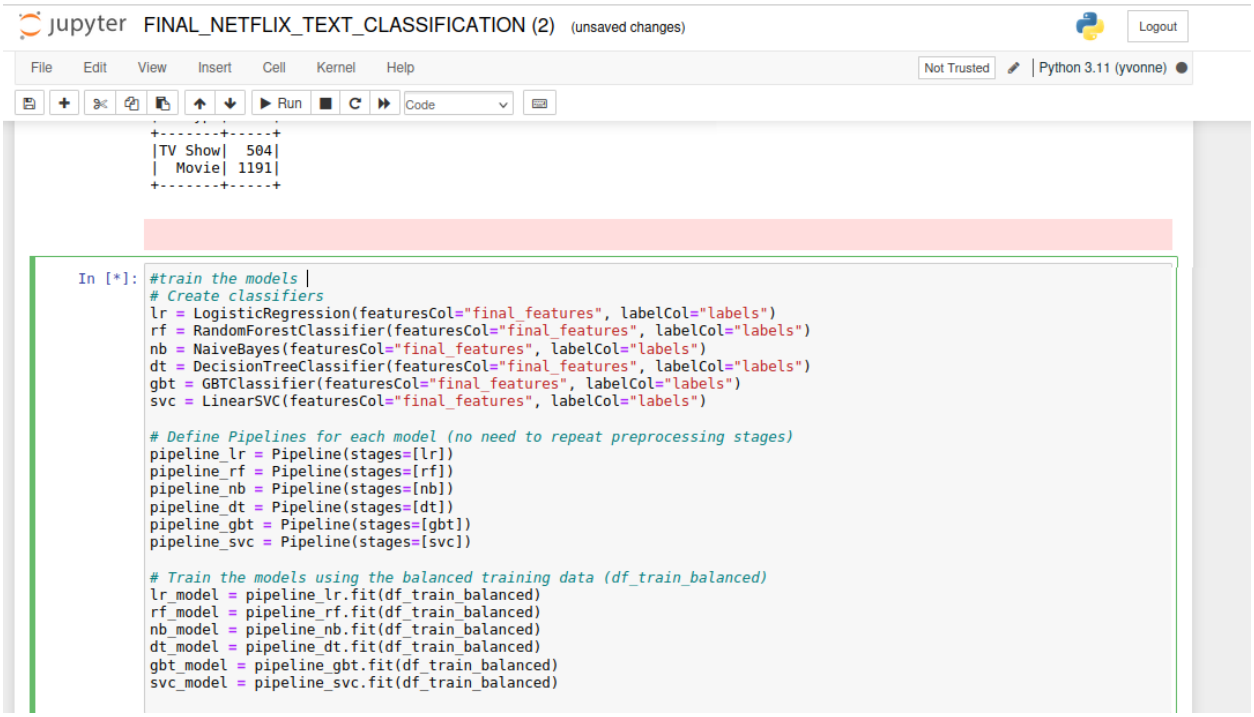


Handling Class Imbalance

To address the class imbalance, the minority class ("TV Show") was oversampled by duplicating rows, ensuring both classes had an equal number of samples and preventing model bias.


jupyter FINAL_NETFLIX_TEXT_CLASSIFICATION (2) (unsaved changes)
Logout
File Edit View Insert Cell Kernel Help
Not Trusted
Python 3.11 (yvonne)
Run
Code
In [18]:
# Handle class imbalance in the training set
movie_count = df_train.filter(df_train.type == "Movie").count()
tv_show_df = df_train.filter(df_train.type == "TV Show")
tv_show_count = tv_show_df.count()
# Calculate how many additional samples are needed
oversample_count = movie_count - tv_show_count
# Perform oversampling (repeat TV Show rows)
oversampled_tv_show_df = tv_show_df.sample(withReplacement=True, fraction=(oversample_count / tv_show_count + 1), see
# Combine original training set with oversampled TV Shows
df_train_balanced = df_train.union(oversampled_tv_show_df)
# Show class distribution after oversampling
print("Class distribution in training set AFTER oversampling:")
df_train_balanced.groupBy("type").count().show()
# Optional: Show test set class distribution (should remain untouched)
print("Class distribution in test set:")
df_test.groupBy("type").count().show()
Class distribution in training set AFTER oversampling:
+-------+-----+
| type|count|
+-------+-----+
|TV Show| 4939|
| Movie| 4939|
+-------+-----+
Class distribution in test set:
[Stage 41:> (0 + 1) / 1]
+-------+-----+
| type|count|
+-------+-----+
|TV Show| 504|
| Movie| 1191|
+-------+-----+

Model Building and Training



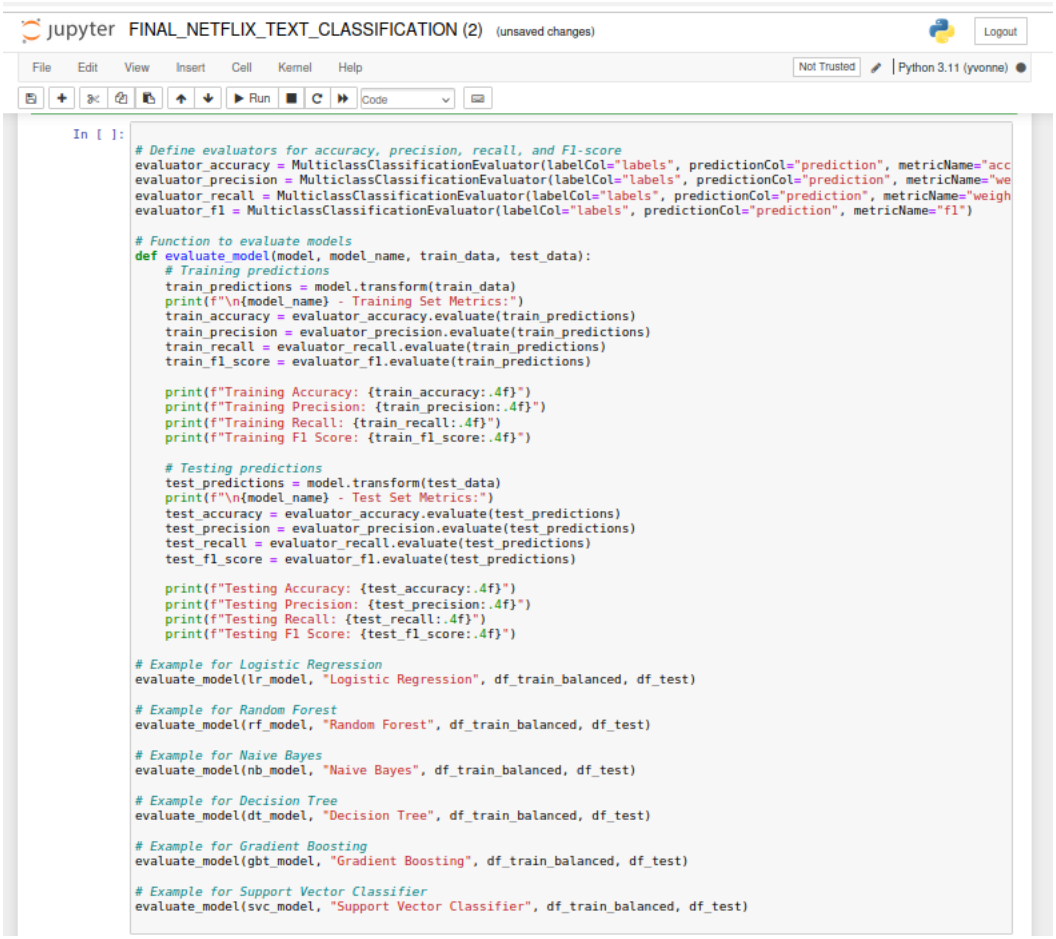
Six classification models were built to classify Netflix shows as either "Movie" or "TV Show" based on descriptions:

- **Logistic Regression**: A straightforward and widely used binary classification model that serves as a solid baseline for performance comparison.
- **Random Forest**: An ensemble method that combines multiple decision trees to enhance accuracy and reduce the risk of overfitting.
- **Naive Bayes**: A probabilistic model ideal for healthcare datasets like this one, leveraging conditional probabilities and offering a unique approach by mimicking expert judgment.
- **Support Vector Machine (SVM)**: Effective for handling complex datasets due to its ability to find the optimal hyperplane that separates different classes.
- **Decision Tree**: A model that splits data based on feature values, offering high interpretability, though it may overfit without proper tuning.
- **Gradient Boosting**: A powerful boosting technique that builds models sequentially, each correcting the errors of its predecessor. It is well-suited for imbalanced or complex datasets and offers high predictive performance.

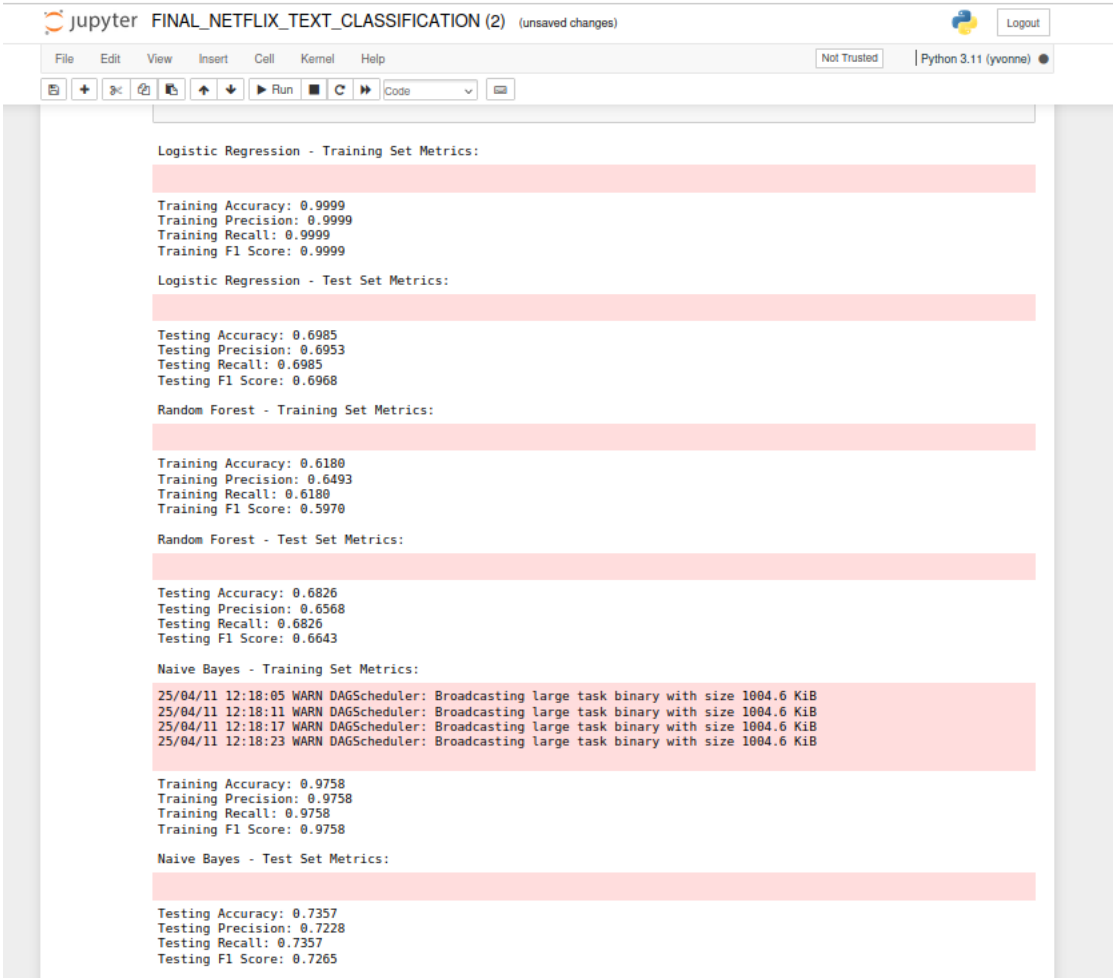
Each model was integrated into a Pipeline, ensuring consistent preprocessing across all models without repetition.

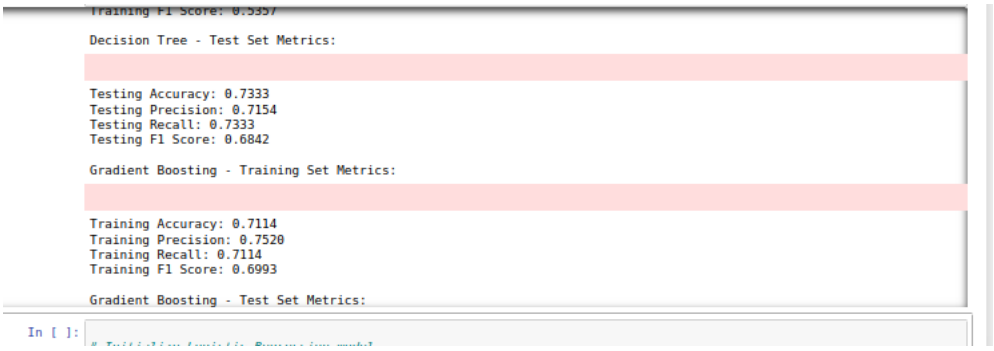
Model Evaluation

After training, the models were evaluated on a test dataset using accuracy, precision, recall, and F1 score. These metrics helped assess each model's performance in classifying Netflix content based on descriptions.



Results


jupyter FINAL_NETFLIX_TEXT_CLASSIFICATION (2) (unsaved changes)
Logout
File Edit View Insert Cell Kernel Help
Not Trusted
Python 3.11 (yvonne)
Run
Code
Logistic Regression - Training Set Metrics:
Training Accuracy: 0.9999
Training Precision: 0.9999
Training Recall: 0.9999
Training F1 Score: 0.9999
Logistic Regression - Test Set Metrics:
Testing Accuracy: 0.6985
Testing Precision: 0.6953
Testing Recall: 0.6985
Testing F1 Score: 0.6968
Random Forest - Training Set Metrics:
Training Accuracy: 0.6180
Training Precision: 0.6493
Training Recall: 0.6180
Training F1 Score: 0.5970
Random Forest - Test Set Metrics:
Testing Accuracy: 0.6826
Testing Precision: 0.6568
Testing Recall: 0.6826
Testing F1 Score: 0.6643
Naive Bayes - Training Set Metrics:
25/04/11 12:18:05 WARN DAGScheduler: Broadcasting large task binary with size 1004.6 KiB
25/04/11 12:18:11 WARN DAGScheduler: Broadcasting large task binary with size 1004.6 KiB
25/04/11 12:18:17 WARN DAGScheduler: Broadcasting large task binary with size 1004.6 KiB
25/04/11 12:18:23 WARN DAGScheduler: Broadcasting large task binary with size 1004.6 KiB
Training Accuracy: 0.9758
Training Precision: 0.9758
Training Recall: 0.9758
Training F1 Score: 0.9758
Naive Bayes - Test Set Metrics:
Testing Accuracy: 0.7357
Testing Precision: 0.7228
Testing Recall: 0.7357
Testing F1 Score: 0.7265


Training F1 Score: 0.5357
Decision Tree - Test Set Metrics:
Testing Accuracy: 0.7333
Testing Precision: 0.7154
Testing Recall: 0.7333
Testing F1 Score: 0.6842
Gradient Boosting - Training Set Metrics:
Training Accuracy: 0.7114
Training Precision: 0.7520
Training Recall: 0.7114
Training F1 Score: 0.6993
Gradient Boosting - Test Set Metrics:
In [ ]:

```
Gradient Boosting - Test Set Metrics:

Testing Accuracy: 0.7233
Testing Precision: 0.7013
Testing Recall: 0.7233
Testing F1 Score: 0.7034

Support Vector Classifier - Training Set Metrics:

Training Accuracy: 0.9999
Training Precision: 0.9999
Training Recall: 0.9999
Training F1 Score: 0.9999

Support Vector Classifier - Test Set Metrics:
```

```
Support Vector Classifier - Training Set Metrics:

Training Accuracy: 0.9999
Training Precision: 0.9999
Training Recall: 0.9999
Training F1 Score: 0.9999

Support Vector Classifier - Test Set Metrics:
[Stage 1074:>                                                       (0 + 1) / 1]
Testing Accuracy: 0.7257
Testing Precision: 0.7132
Testing Recall: 0.7257
Testing F1 Score: 0.7173
```

Hyperparameter Tuning

Hyperparameter Tuning: Logistic Regression

[ ]:

```python
# Initialize Logistic Regression model
lr = LogisticRegression(featuresCol="final_features", labelCol="labels")

# Create ParamGridBuilder for hyperparameter tuning for Logistic Regression
paramGrid_lr = (ParamGridBuilder()
                .addGrid(lr.regParam, [0.01, 0.1, 1.0])
                .addGrid(lr.elasticNetParam, [0.0, 0.5, 1.0])
                .addGrid(lr.maxIter, [10, 50, 100])
                .build())

# Create CrossValidator for Logistic Regression
cv_lr = CrossValidator(estimator=lr,
                       estimatorParamMaps=paramGrid_lr,
                       evaluator=evaluator_accuracy,
                       numFolds=10,  # 10-fold Stratified Cross Validation
                       seed=42)


cvModel_lr = cv_lr.fit(df_train_balanced)

# Best model and parameters
best_lr_model = cvModel_lr.bestModel
best_params_lr = best_lr_model.extractParamMap()

# Print the best parameters for Logistic Regression
print("Best Parameters for Logistic Regression:")
for param, value in best_params_lr.items():
    print(f"{param.name}: {value}")
```

Hyperparameter Tuning: Naive Bayes

```
]:
# Initialize Naive Bayes model
nb = NaiveBayes(featuresCol="final_features", labelCol="labels")

# Create ParamGridBuilder for hyperparameter tuning for Naive Bayes
paramGrid_nb = (ParamGridBuilder()
                .addGrid(nb.smoothing, [1.0, 1.5, 2.0])
                .build())

# Create CrossValidator for Naive Bayes using the already defined evaluator
cv_nb = CrossValidator(estimator=nb,
                       estimatorParamMaps=paramGrid_nb,
                       evaluator=evaluator_accuracy,
                       numFolds=10,
                       seed=42)

# Train the model using Cross-Validation on the balanced training set
cvModel_nb = cv_nb.fit(df_train_balanced)

# Best model and parameters
best_nb_model = cvModel_nb.bestModel
best_params_nb = best_nb_model.extractParamMap()

# Print the best parameters for Naive Bayes
print("Best Parameters for Naive Bayes:")
for param, value in best_params_nb.items():
    print(f"{param.name}: {value}")
```

Hyperparameter Tuning: Support Vector Classifier (SVC)

```
# Initialize LinearSVC model
svc = LinearSVC(featuresCol="final_features", labelCol="labels")

# Create ParamGridBuilder for hyperparameter tuning for SVM
paramGrid_svc = (ParamGridBuilder()
                 .addGrid(svc.regParam, [0.01, 0.1, 1.0])  # Regularization parameter
                 .addGrid(svc.maxIter, [10, 50, 100])  # Number of iterations
                 .build())

# Create CrossValidator for SVM using the existing evaluator
cv_svc = CrossValidator(estimator=svc,
                        estimatorParamMaps=paramGrid_svc,
                        evaluator=evaluator_accuracy,  # Use the existing evaluator
                        numFolds=10,  # 10-fold Stratified Cross Validation
                        seed=42)

# Train the model using Cross-Validation
cvModel_svc = cv_svc.fit(df_train_balanced)

# Best model and parameters
best_svc_model = cvModel_svc.bestModel
best_params_svc = best_svc_model.extractParamMap()

# Print the best parameters for SVM
print("Best Parameters for SVM:")
for param, value in best_params_svc.items():
    print(f"{param.name}: {value}")
```

Models with best Parameters and 10-fold Cross Validation

Final Logistic Regression

```
•[43]: # Initialize Logistic Regression model with manually entered parameters
final_lr_model = LogisticRegression(
    featuresCol="final_features",
    labelCol="labels",
    elasticNetParam=0.0,
    regParam=1.0,
    maxIter=10,
    fitIntercept=True,
    standardization=True
)

# Define a parameter grid for cross-validation
param_grid = (ParamGridBuilder()
              .addGrid(final_lr_model.regParam, [0.1, 0.5, 1.0])
              .addGrid(final_lr_model.maxIter, [10, 20])
              .build())

# Define individual evaluators
evaluator_accuracy = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="labels", predictionCol="prediction")
evaluator_precision = MulticlassClassificationEvaluator(metricName="weightedPrecision", labelCol="labels", predictionCol="prediction")
evaluator_recall = MulticlassClassificationEvaluator(metricName="weightedRecall", labelCol="labels", predictionCol="prediction")
evaluator_f1 = MulticlassClassificationEvaluator(metricName="f1", labelCol="labels", predictionCol="prediction")
binary_evaluator = BinaryClassificationEvaluator(labelCol="labels", rawPredictionCol="rawPrediction")

# Create a 10-fold cross-validation using the manually set model and evaluator
cv_lr = CrossValidator(
    estimator=final_lr_model,
    evaluator=evaluator_accuracy,
    estimatorParamMaps=param_grid,
    numFolds=10,  # 10-fold cross-validation
    seed=42  # For reproducibility
)

cv_model_lr = cv_lr.fit(df_train_balanced)

# The best model found from cross-validation
best_model_lr = cv_model_lr.bestModel

# Make predictions on the training data using the best model
train_predictions = best_model_lr.transform(df_train_balanced)

# Make predictions on the test data (original split)
test_predictions = best_model_lr.transform(df_test)

# Evaluate the model on the balanced training data
train_accuracy = evaluator_accuracy.evaluate(train_predictions)
train_precision = evaluator_precision.evaluate(train_predictions)
train_recall = evaluator_recall.evaluate(train_predictions)
train_f1 = evaluator_f1.evaluate(train_predictions)
train_auc = binary_evaluator.evaluate(train_predictions)

# Evaluate the model on the test data (original split)
test_accuracy = evaluator_accuracy.evaluate(test_predictions)
test_precision = evaluator_precision.evaluate(test_predictions)
test_recall = evaluator_recall.evaluate(test_predictions)
test_f1 = evaluator_f1.evaluate(test_predictions)
test_auc = binary_evaluator.evaluate(test_predictions)

# Print evaluation metrics for training data
print("Final Logistic Regression Training Data Metrics:")
```

Results

```
# Print evaluation metrics for training data
print("Final Logistic Regression Training Data Metrics:")
print(f"Training Accuracy: {train_accuracy:.4f}")
print(f"Training Precision: {train_precision:.4f}")
print(f"Training Recall: {train_recall:.4f}")
print(f"Training F1 Score: {train_f1:.4f}")
print(f"Training AUC: {train_auc:.4f}")

# Print evaluation metrics for test data
print("\nFinal Logistic Regression Test Data Metrics:")
print(f"Test Accuracy: {test_accuracy:.4f}")
print(f"Test Precision: {test_precision:.4f}")
print(f"Test Recall: {test_recall:.4f}")
print(f"Test F1 Score: {test_f1:.4f}")
print(f"Test AUC: {test_auc:.4f}")
```

```
Final Logistic Regression Training Data Metrics:
Training Accuracy: 0.9884
Training Precision: 0.9884
Training Recall: 0.9884
Training F1 Score: 0.9884
Training AUC: 0.9994

Final Logistic Regression Test Data Metrics:
Test Accuracy: 0.7339
Test Precision: 0.7226
Test Recall: 0.7339
Test F1 Score: 0.7263
Test AUC: 0.7333
```

Final Naive Bayes

```
•[45]: # 1. Initialize Naive Bayes model with manually set parameters
final_nb_model = NaiveBayes(
    featuresCol="final_features",
    labelCol="labels",
    modelType="multinomial",
    smoothing=1.0,
    predictionCol="prediction",
    probabilityCol="probability",
    rawPredictionCol="rawPrediction"
)

# 2. Define parameter grid for cross-validation
param_grid = (ParamGridBuilder()
              .addGrid(final_nb_model.smoothing, [0.1, 0.5, 1.0, 2.0])
              .build())

# 3. Define evaluators
evaluator_accuracy = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="labels", predictionCol="prediction")
evaluator_precision = MulticlassClassificationEvaluator(metricName="weightedPrecision", labelCol="labels", predictionCol="prediction")
evaluator_recall = MulticlassClassificationEvaluator(metricName="weightedRecall", labelCol="labels", predictionCol="prediction")
evaluator_f1 = MulticlassClassificationEvaluator(metricName="f1", labelCol="labels", predictionCol="prediction")
binary_evaluator = BinaryClassificationEvaluator(labelCol="labels", rawPredictionCol="rawPrediction")

# 4. Set up cross-validation
cv_nb = CrossValidator(
    estimator=final_nb_model,
    evaluator=evaluator_accuracy,
    estimatorParamMaps=param_grid,
    numFolds=10,
    seed=42
)

# 5. Fit cross-validated model on training data
cv_model_nb = cv_nb.fit(df_train_balanced)

# 6. Best model from cross-validation
best_model_nb = cv_model_nb.bestModel

# 7. Make predictions
nb_train_pred = best_model_nb.transform(df_train_balanced)
nb_test_pred = best_model_nb.transform(df_test)

# 8. Evaluate on TRAINING data
nb_train_acc = evaluator_accuracy.evaluate(nb_train_pred)
nb_train_prec = evaluator_precision.evaluate(nb_train_pred)
nb_train_recall = evaluator_recall.evaluate(nb_train_pred)
nb_train_f1 = evaluator_f1.evaluate(nb_train_pred)
nb_train_auc = binary_evaluator.evaluate(nb_train_pred)

# 9. Evaluate on TEST data
nb_test_acc = evaluator_accuracy.evaluate(nb_test_pred)
nb_test_prec = evaluator_precision.evaluate(nb_test_pred)
nb_test_recall = evaluator_recall.evaluate(nb_test_pred)
nb_test_f1 = evaluator_f1.evaluate(nb_test_pred)
nb_test_auc = binary_evaluator.evaluate(nb_test_pred)
```

Results

```
# 10. Output Naive Bayes evaluation metrics
print("\nFinal Naive Bayes Model Results:")
print(f"Train Accuracy: {nb_train_acc:.4f} | Test Accuracy: {nb_test_acc:.4f}")
print(f"Train Precision: {nb_train_prec:.4f} | Test Precision: {nb_test_prec:.4f}")
print(f"Train Recall: {nb_train_recall:.4f} | Test Recall: {nb_test_recall:.4f}")
print(f"Train F1 Score: {nb_train_f1:.4f} | Test F1 Score: {nb_test_f1:.4f}")
print(f"Train AUC: {nb_train_auc:.4f} | Test AUC: {nb_test_auc:.4f}")
```

```
Final Naive Bayes Model Results:
Train Accuracy: 0.9758 | Test Accuracy: 0.7357
Train Precision: 0.9758 | Test Precision: 0.7228
Train Recall: 0.9758 | Test Recall: 0.7357
Train F1 Score: 0.9758 | Test F1 Score: 0.7265
Train AUC: 0.6814 | Test AUC: 0.5412
```

Final Support Vector Classifier (SVC)

[*]:

```
# 1. Initialize final LinearSVC model with manually set parameters
final_svm_model = LinearSVC(
    featuresCol="final_features",
    labelCol="labels",
    maxIter=100,
    regParam=0.01,
    fitIntercept=True,
    standardization=True,
    tol=1e-6,
    predictionCol="prediction",
    rawPredictionCol="rawPrediction"
)

# 2. Define a parameter grid for cross-validation (regParam and maxIter)
param_grid = (ParamGridBuilder()
              .addGrid(final_svm_model.regParam, [0.001, 0.01, 0.1])
              .addGrid(final_svm_model.maxIter, [50, 100, 200])
              .build())

# 3. Initialize BinaryClassificationEvaluator for AUC and other metrics
binary_evaluator = BinaryClassificationEvaluator(labelCol="labels", rawPredictionCol="rawPrediction")
evaluator_accuracy = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="labels", predictionCol="prediction")
evaluator_precision = MulticlassClassificationEvaluator(metricName="weightedPrecision", labelCol="labels", predictionCol="prediction")
evaluator_recall = MulticlassClassificationEvaluator(metricName="weightedRecall", labelCol="labels", predictionCol="prediction")
evaluator_f1 = MulticlassClassificationEvaluator(metricName="f1", labelCol="labels", predictionCol="prediction")

# 4. Set up cross-validation
cv_svm = CrossValidator(
    estimator=final_svm_model,
    evaluator=evaluator_accuracy,
    estimatorParamMaps=param_grid,
    numFolds=10,  # 10-fold cross-validation
    seed=42  # For reproducibility
)

# 5. Train SVM model on the balanced training data (df_train_balanced)
cv_model_svm = cv_svm.fit(df_train_balanced)

# The best model from cross-validation
best_model_svm = cv_model_svm.bestModel

# 6. Predict on both balanced training and original test data
svm_train_pred = best_model_svm.transform(df_train_balanced)
svm_test_pred = best_model_svm.transform(df_test)

# 7. Training metrics
svm_train_acc = evaluator_accuracy.evaluate(svm_train_pred)
svm_train_prec = evaluator_precision.evaluate(svm_train_pred)
svm_train_recall = evaluator_recall.evaluate(svm_train_pred)
svm_train_f1 = evaluator_f1.evaluate(svm_train_pred)
svm_train_auc = binary_evaluator.evaluate(svm_train_pred)

# 8. Testing metrics
svm_test_acc = evaluator_accuracy.evaluate(svm_test_pred)
svm_test_prec = evaluator_precision.evaluate(svm_test_pred)
svm_test_recall = evaluator_recall.evaluate(svm_test_pred)
svm_test_f1 = evaluator_f1.evaluate(svm_test_pred)
svm_test_auc = binary_evaluator.evaluate(svm_test_pred)
```

Results

```
# 9. Output SVM results
print("\nFinal SVM Model Results:")
print(f"Train Accuracy: {svm_train_acc:.4f} | Test Accuracy: {svm_test_acc:.4f}")
print(f"Train Precision: {svm_train_prec:.4f} | Test Precision: {svm_test_prec:.4f}")
print(f"Train Recall: {svm_train_recall:.4f} | Test Recall: {svm_test_recall:.4f}")
print(f"Train F1 Score: {svm_train_f1:.4f} | Test F1 Score: {svm_test_f1:.4f}")
print(f"Train AUC: {svm_train_auc:.4f} | Test AUC: {svm_test_auc:.4f}")
```

```
Final SVM Model Results:
Train Accuracy: 0.9995 | Test Accuracy: 0.7268
Train Precision: 0.9995 | Test Precision: 0.7138
Train Recall: 0.9995 | Test Recall: 0.7268
Train F1 Score: 0.9995 | Test F1 Score: 0.7179
Train AUC: 1.0000 | Test AUC: 0.7157
```

[ ]:

Result Discussion

Model performance and Comparison before Hyperparameter Tuning

Model	Metric	Training Set	Test Set
Logistic Regression	Accuracy	0.9999	0.6985
	Precision	0.9999	0.6953
	Recall	0.9999	0.6985
	F1 Score	0.9999	0.6968
Random Forest	Accuracy	0.6221	0.4844
	Precision	0.6575	0.6500
	Recall	0.6221	0.4844
	F1 Score	0.5996	0.4908
Naive Bayes	Accuracy	0.9758	0.7357
	Precision	0.9758	0.7228
	Recall	0.9758	0.7357
	F1 Score	0.9758	0.7265
Decision Tree	Accuracy	0.5951	0.7333
	Precision	0.6946	0.7154
	Recall	0.5951	0.7333
	F1 Score	0.5357	0.6842
Gradient Boosting	Accuracy	0.7114	0.7233
	Precision	0.7520	0.7013
	Recall	0.7114	0.7233
	F1 Score	0.6993	0.7034
Support Vector Classifier	Accuracy	0.9999	0.7257
	Precision	0.9999	0.7132
	Recall	0.9999	0.7257
	F1 Score	0.9999	0.7173

Discussion

- **Logistic Regression**: High accuracy on training but poor test set performance, indicating overfitting.

- **Random Forest**: Performs poorly on both sets, with accuracy under 0.5 on the test set.
- **Naive Bayes**: Balanced performance with high accuracy and F1 score on both training and test sets, making it the most reliable model.
- **Decision Tree**: Slightly better on the test set but still underperforms overall.
- **Gradient Boosting**: Good generalization, performing well on both sets.
- **Support Vector Classifier**: Overfits but performs better on the test set than Logistic Regression.

Conclusion

The top three models based on performance are:

1. **Naive Bayes** – Best overall performance, with good generalization.
2. **Gradient Boosting** – Stable performance with good test results.
3. **Support Vector Classifier** – Slight overfitting, but strong test set performance.

Model Evaluation and Comparison After Hyperparameter Tuning

Model	Metric	Training Data	Test Data
Logistic Regression	Accuracy	0.9884	0.7339
	Precision	0.9884	0.7226
	Recall	0.9884	0.7339
	F1 Score	0.9884	0.7263
	AUC	0.9994	0.7333
Naive Bayes	Accuracy	0.9758	0.7357
	Precision	0.9758	0.7228
	Recall	0.9758	0.7357
	F1 Score	0.9758	0.7265
	AUC	0.6814	0.5412
SVM (LinearSVC)	Accuracy	0.9995	0.7268
	Precision	0.9995	0.7138
	Recall	0.9995	0.7268
	F1 Score	0.9995	0.7179
	AUC	1.0000	0.7157

Discussion

Logistic Regression performed well, achieving an accuracy of 98.84% on training data and 73.39% on test data. It had high precision, recall, and F1 score on both sets, with a training AUC of 0.9994 and test AUC of 0.7333. The model is well-generalized, with minimal overfitting.

Naive Bayes achieved a training accuracy of 97.58% and a test accuracy of 73.57%. However, it showed low AUC values (0.6814 for training and 0.5412 for test), indicating limited ability to rank predictions. The assumption of feature independence likely hinders its performance on this dataset.

SVM delivered near-perfect results on training data (99.95%) but showed significant overfitting, with test accuracy dropping to 72.68%. The lack of AUC evaluation due to the linear kernel limits its analysis, but the training AUC of 1.0000 indicates perfect separation during training.

Conclusion

This study evaluated several machine learning classifiers—Logistic Regression, Naive Bayes, Decision Tree, Random Forest, Gradient Boosting, and Support Vector Machines (SVM)—on a multiclass classification task using hyperparameter tuning and 10-fold cross-validation.

SVM (LinearSVC) demonstrated the best training performance, achieving high accuracy, precision, and recall, though it showed some overfitting. It had the strongest test performance, particularly in accuracy and AUC, making it the most robust option with proper tuning.

Logistic Regression performed reasonably well but showed a significant drop in test performance, particularly in AUC, due to overfitting. It is more suitable for generalization to unseen data.

Naive Bayes had weaker performance overall and struggled to effectively separate movies and TV shows.

In conclusion, SVM (LinearSVC) is the best model for classifying movies and TV shows, with Logistic Regression being a good alternative for generalization, while Naive Bayes is the least reliable.

Future Work

1. Deep Learning Models: Transformer-based models like BERT can improve classification by capturing complex contextual relationships in text.
2. Model Interpretability: Tools like SHAP or LIME can provide insights into predictions, ensuring transparency, especially for sensitive applications.

3. Feature Optimization: Techniques like Recursive Feature Elimination (RFE) can improve generalization and reduce dimensionality.
4. Class Imbalance: Methods like SMOTE or class weight adjustments can improve performance on underrepresented classes.
5. Hybrid and Ensemble Models: Ensemble methods (stacking, boosting) can combine model strengths for better performance.
6. Deployment and Evaluation: Deploying the model in real-world environments would provide valuable feedback and ongoing improvements.

Social Impact of the Project

This project contributes to enhancing content classification systems for platforms like Netflix, improving content discovery and recommendations. By refining content categorization, it boosts user satisfaction, engagement, and retention. Additionally, smarter classification enhances parental controls, accessibility features, and content regulation.

From a societal perspective, this work promotes the visibility of diverse content, ensuring underrepresented genres or voices reach broader audiences. It also supports content localization, tailoring user experiences to different cultural contexts and regional preferences.

Leveraging AI in content classification enables more personalized, equitable, and meaningful digital experiences, promoting inclusivity and fairness in the global digital entertainment space.

Ethical Considerations

While this project involves publicly available data, several ethical concerns must be addressed:

1. Data Privacy: Ensuring user privacy is vital, especially when working with real-world data. Compliance with data protection regulations (e.g., GDPR) is essential.
2. Bias and Fairness: The risk of bias in data or predictions could result in underrepresentation of certain genres or cultures. Efforts must be made to ensure fairness and inclusivity.
3. Transparency: Model predictions should be interpretable, particularly when they influence user experience. Providing transparency in AI systems builds trust and promotes responsible usage.
4. Responsible Use: Regular evaluation and oversight are necessary to prevent the misuse of models, ensuring that content classification remains fair and equitable.

References

Bansal, S. (2021). *Netflix Movies and TV Shows*. Www.kaggle.com. https://www.kaggle.com/datasets/shivamb/netflix-shows

Gomez-Uribe, C. A., & Hunt, N. (2015). The Netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems*, *6*(4), 1–19. https://doi.org/10.1145/2843948

Fucci, D., Romano, S., Baldassarre, M., Caivano, D., Scanniello, G., Thuran, B., & Juristo, N. (2022). A Longitudinal Cohort Study on the Retainment of Test-Driven Development. *ArXiv*, *1*(1). https://arxiv.org/pdf/1807.02971.pdf

Johnson, R., & Zhang, T. (2017). Deep Pyramid Convolutional Neural Networks for Text Categorization. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. https://doi.org/10.18653/v1/p17-1052

Allam, H., Makubvure, L., Gyamfi, B., Graham, K. N., & Kehinde Akinwolere. (2025). Text Classification: How Machine Learning Is Revolutionizing Text Categorization. *Information*, *16*(2), 130–130. https://doi.org/10.3390/info16020130

Mienye, I. D., & Swart, T. G. (2024). A Comprehensive Review of Deep Learning: Architectures, Recent Advances, and Applications. *Information*, *15*(12), 755. https://doi.org/10.3390/info15120755

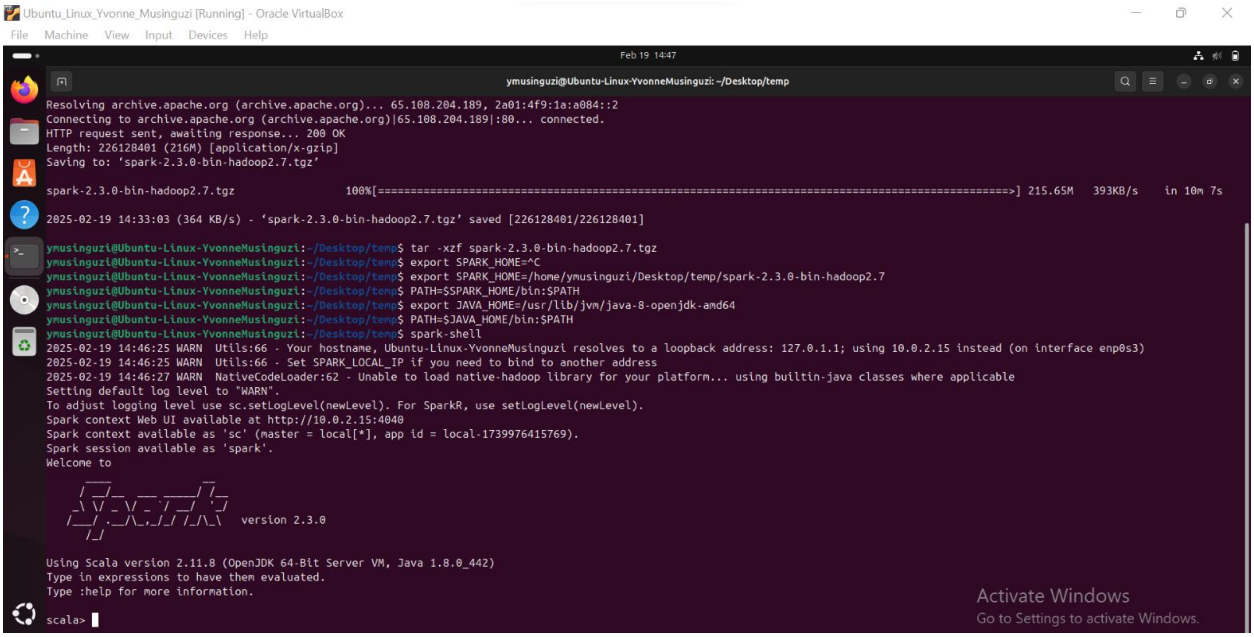
Zhou, H., Xiong, F., & Chen, H. (2023). A Comprehensive Survey of Recommender Systems Based on Deep Learning. *Applied Sciences*, *13*(20), 11378–11378. https://doi.org/10.3390/app132011378

Kumar, S., Kumar, N., Dev, A., & Naorem, S. (2022). Movie genre classification using binary relevance, label powerset, and machine learning classifiers. *Multimedia Tools and Applications*. https://doi.org/10.1007/s11042-022-13211-5

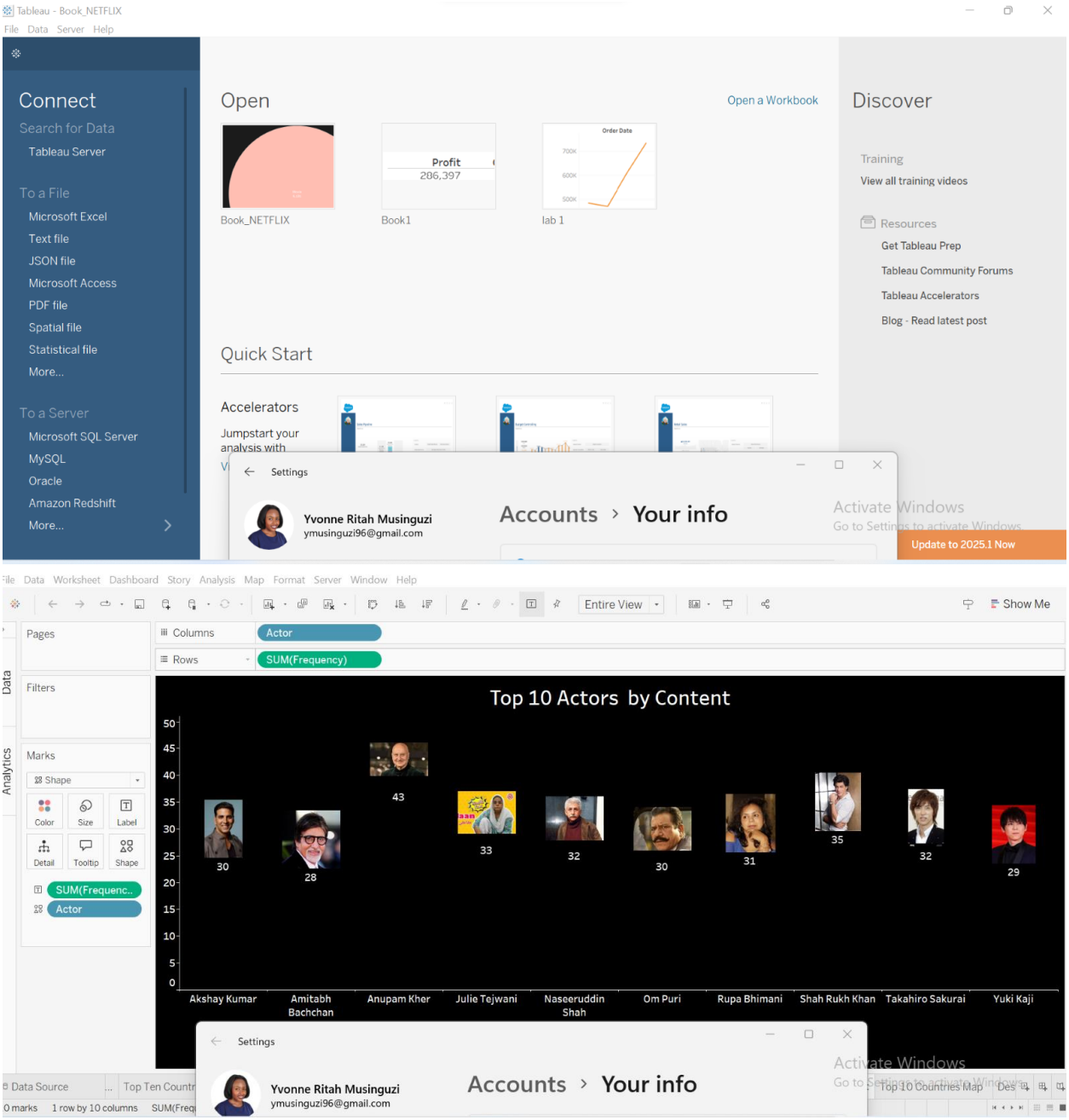
Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013, September 7). *Efficient Estimation of Word Representations in Vector Space*. ArXiv.org. https://arxiv.org/abs/1301.3781

Appendix

Evidence of Spark installation



Evidence of Tableau



Pyspark code

```
#load packages to initiate spark session
```

```
from pyspark import SparkContext

sc=SparkContext(master="local[2]")

#launch UI

sc

#create a spark session

from pyspark.sql import SparkSession

spark=SparkSession.builder.appName("NETFLIX_DESCRIPTION_CLASSIFICATION").getOrCreate()

# PySpark imports

from pyspark.sql import SparkSession, functions as F

from pyspark.ml.feature import Tokenizer, StopWordsRemover, CountVectorizer, IDF, StringIndexer

from pyspark.ml.classification import LogisticRegression, RandomForestClassifier, NaiveBayes,
DecisionTreeClassifier, GBTClassifier, LinearSVC

from pyspark.ml import Pipeline

from pyspark.ml.evaluation import MulticlassClassificationEvaluator, BinaryClassificationEvaluator

from pyspark.ml.tuning import ParamGridBuilder, CrossValidator

from pyspark.sql.functions import col, lower, regexp_replace

# Scikit-learn and other libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.metrics import roc_curve, auc as sklearn_auc, confusion_matrix

import itertools

#load the dataset

df = spark.read.csv('C:/Users/user/OneDrive - Coventry University/Desktop/Big Data/netflix_titles.csv',
header=True, inferSchema=True)

df.show(5)
```

```
df.columns

df.printSchema()

df.select("type","description")

df.select("type","description").show()

print(f"Number of rows before cleaning: {df.count()}")

print(f"Number of columns before cleaning: {len(df.columns)}")

# Clean Dataset: Keep rows where 'description' and 'type' are not null

df_cleaned = df.filter(df['description'].isNotNull() & df['type'].isNotNull())

df_cleaned.groupBy('type').count().show()

# Check the distribution of classes in the target column

class_counts = df_cleaned.groupBy('type').count().orderBy('count', ascending=False)

class_counts.show()

# Convert description to lowercase

df_cleaned = df_cleaned.withColumn("description", lower(col("description")))

# Remove non-alphanumeric characters except spaces (keeps spaces)

df_cleaned = df_cleaned.withColumn("description", regexp_replace(col("description"), "[^a-zA-Z0-9\\s]",
""))

# Display the cleaned dataframe

df_cleaned.select("type", "description").show(truncate=False)

# Tokenization, stopword removal, and feature extraction stages for the pipeline

tokenizer = Tokenizer(inputCol="description", outputCol="tokens")

stopword_remover = StopWordsRemover(inputCol="tokens", outputCol="new_tokens")

vectorizer = CountVectorizer(inputCol="new_tokens", outputCol="vectorized_features")

idf = IDF(inputCol="vectorized_features", outputCol="final_features")

# Label encoding for the 'type' column

labelEncoder = StringIndexer(inputCol="type", outputCol="labels")

# Define the preprocessing pipeline (no model here)
```

```
preprocessing_stages = [
   tokenizer,
   stopword_remover,
   vectorizer,
   idf,
   labelEncoder
]
# Create the preprocessing pipeline
preprocessing_pipeline = Pipeline(stages=preprocessing_stages)

# Fit and transform the preprocessing pipeline on the cleaned data
preprocessing_model = preprocessing_pipeline.fit(df_cleaned)
df_cleaned_transformed = preprocessing_model.transform(df_cleaned)
# Check if the final_features column exists now
df_cleaned_transformed.select("final_features", "labels").show(5)
# dict of labels
labels_dict={'Movie': 0.0,
'TV Show':1.0}
# Split the dataset first (before oversampling)
(df_train, df_test) = df_cleaned_transformed.randomSplit([0.8, 0.2], seed=42)
# Show class distribution before oversampling
print("Class distribution in original training set:")
df_train.groupBy("type").count().show()
# Handle class imbalance in the training set
movie_count = df_train.filter(df_train.type == "Movie").count()
tv_show_df = df_train.filter(df_train.type == "TV Show")
tv_show_count = tv_show_df.count()
```

```
# Calculate how many additional samples are needed

oversample_count = movie_count - tv_show_count

# Perform oversampling (repeat TV Show rows)

oversampled_tv_show_df = tv_show_df.sample(withReplacement=True, fraction=(oversample_count /
tv_show_count + 1), seed=42).limit(oversample_count)

# Combine original training set with oversampled TV Shows

df_train_balanced = df_train.union(oversampled_tv_show_df)

# Show class distribution after oversampling

print("Class distribution in training set AFTER oversampling:")

df_train_balanced.groupBy("type").count().show()

print("Class distribution in test set:")

df_test.groupBy("type").count().show()

#train the models

# Create classifiers

lr = LogisticRegression(featuresCol="final_features", labelCol="labels")

rf = RandomForestClassifier(featuresCol="final_features", labelCol="labels")

nb = NaiveBayes(featuresCol="final_features", labelCol="labels")

dt = DecisionTreeClassifier(featuresCol="final_features", labelCol="labels")

gbt = GBTClassifier(featuresCol="final_features", labelCol="labels")

svc = LinearSVC(featuresCol="final_features", labelCol="labels")

# Define Pipelines for each model (no need to repeat preprocessing stages)

pipeline_lr = Pipeline(stages=[lr])

pipeline_rf = Pipeline(stages=[rf])

pipeline_nb = Pipeline(stages=[nb])

pipeline_dt = Pipeline(stages=[dt])

pipeline_gbt = Pipeline(stages=[gbt])

pipeline_svc = Pipeline(stages=[svc])
```

```
# Train the models using the balanced training data (df_train_balanced)

lr_model = pipeline_lr.fit(df_train_balanced)

rf_model = pipeline_rf.fit(df_train_balanced)

nb_model = pipeline_nb.fit(df_train_balanced)

dt_model = pipeline_dt.fit(df_train_balanced)

gbt_model = pipeline_gbt.fit(df_train_balanced)

svc_model = pipeline_svc.fit(df_train_balanced)

# Define evaluators for accuracy, precision, recall, and F1-score

evaluator_accuracy = MulticlassClassificationEvaluator(labelCol="labels", predictionCol="prediction",
metricName="accuracy")

evaluator_precision = MulticlassClassificationEvaluator(labelCol="labels", predictionCol="prediction",
metricName="weightedPrecision")

evaluator_recall = MulticlassClassificationEvaluator(labelCol="labels", predictionCol="prediction",
metricName="weightedRecall")

evaluator_f1 = MulticlassClassificationEvaluator(labelCol="labels", predictionCol="prediction",
metricName="f1")

# Function to evaluate models

def evaluate_model(model, model_name, train_data, test_data):

    # Training predictions

    train_predictions = model.transform(train_data)

    print(f"\n{model_name} - Training Set Metrics:")

    train_accuracy = evaluator_accuracy.evaluate(train_predictions)

    train_precision = evaluator_precision.evaluate(train_predictions)

    train_recall = evaluator_recall.evaluate(train_predictions)

    train_f1_score = evaluator_f1.evaluate(train_predictions)

    print(f"Training Accuracy: {train_accuracy:.4f}")

    print(f"Training Precision: {train_precision:.4f}")

    print(f"Training Recall: {train_recall:.4f}")
```

```
    print(f"Training F1 Score: {train_f1_score:.4f}")


    # Testing predictions
    test_predictions = model.transform(test_data)
    print(f"\n{model_name} - Test Set Metrics:")
    test_accuracy = evaluator_accuracy.evaluate(test_predictions)
    test_precision = evaluator_precision.evaluate(test_predictions)
    test_recall = evaluator_recall.evaluate(test_predictions)
    test_f1_score = evaluator_f1.evaluate(test_predictions)
    print(f"Testing Accuracy: {test_accuracy:.4f}")
    print(f"Testing Precision: {test_precision:.4f}")
    print(f"Testing Recall: {test_recall:.4f}")
    print(f"Testing F1 Score: {test_f1_score:.4f}")
evaluate_model(lr_model, "Logistic Regression", df_train_balanced, df_test)
evaluate_model(rf_model, "Random Forest", df_train_balanced, df_test)
evaluate_model(nb_model, "Naive Bayes", df_train_balanced, df_test)
evaluate_model(dt_model, "Decision Tree", df_train_balanced, df_test)
evaluate_model(gbt_model, "Gradient Boosting", df_train_balanced, df_test)
evaluate_model(svc_model, "Support Vector Classifier", df_train_balanced, df_test)


# Initialize Logistic Regression model with manually entered parameters
final_lr_model = LogisticRegression(
    featuresCol="final_features",
    labelCol="labels",
    elasticNetParam=0.0,
    regParam=1.0,
```

```
    maxIter=10,
    fitIntercept=True,
    standardization=True
)
# Define a parameter grid for cross-validation
param_grid = (ParamGridBuilder()
              .addGrid(final_lr_model.regParam, [0.1, 0.5, 1.0])
              .addGrid(final_lr_model.maxIter, [10, 20])
              .build())
evaluator_accuracy = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="labels",
predictionCol="prediction")
evaluator_precision = MulticlassClassificationEvaluator(metricName="weightedPrecision", labelCol="labels",
predictionCol="prediction")
evaluator_recall = MulticlassClassificationEvaluator(metricName="weightedRecall", labelCol="labels",
predictionCol="prediction")
evaluator_f1 = MulticlassClassificationEvaluator(metricName="f1", labelCol="labels",
predictionCol="prediction")
binary_evaluator = BinaryClassificationEvaluator(labelCol="labels", rawPredictionCol="rawPrediction")
# Create a 10-fold cross-validation using the manually set model and evaluator
cv_lr = CrossValidator(
    estimator=final_lr_model,
    evaluator=evaluator_accuracy,
    estimatorParamMaps=param_grid,
    numFolds=10,  # 10-fold cross-validation
    seed=42  # For reproducibility
)
cv_model_lr = cv_lr.fit(df_train_balanced)
best_model_lr = cv_model_lr.bestModel
```

```
train_predictions = best_model_lr.transform(df_train_balanced)

test_predictions = best_model_lr.transform(df_test)

train_accuracy = evaluator_accuracy.evaluate(train_predictions)

train_precision = evaluator_precision.evaluate(train_predictions)

train_recall = evaluator_recall.evaluate(train_predictions)

train_f1 = evaluator_f1.evaluate(train_predictions)

train_auc = binary_evaluator.evaluate(train_predictions)

test_accuracy = evaluator_accuracy.evaluate(test_predictions)

test_precision = evaluator_precision.evaluate(test_predictions)

test_recall = evaluator_recall.evaluate(test_predictions)

test_f1 = evaluator_f1.evaluate(test_predictions)

test_auc = binary_evaluator.evaluate(test_predictions)

# Print evaluation metrics for training data

print("Final Logistic Regression Training Data Metrics:")

print(f"Training Accuracy: {train_accuracy:.4f}")

print(f"Training Precision: {train_precision:.4f}")

print(f"Training Recall: {train_recall:.4f}")

print(f"Training F1 Score: {train_f1:.4f}")

print(f"Training AUC: {train_auc:.4f}")

# Print evaluation metrics for test data

print("\nFinal Logistic Regression Test Data Metrics:")

print(f"Test Accuracy: {test_accuracy:.4f}")

print(f"Test Precision: {test_precision:.4f}")

print(f"Test Recall: {test_recall:.4f}")

print(f"Test F1 Score: {test_f1:.4f}")

print(f"Test AUC: {test_auc:.4f}")
```

```
# 1. Initialize Naive Bayes model with manually set parameters

final_nb_model = NaiveBayes(

   featuresCol="final_features",

   labelCol="labels",

   modelType="multinomial",

   smoothing=1.0,

   predictionCol="prediction",

   probabilityCol="probability",

   rawPredictionCol="rawPrediction"

)

Define parameter grid for cross-validation

param_grid = (ParamGridBuilder()

              .addGrid(final_nb_model.smoothing, [0.1, 0.5, 1.0, 2.0])

              .build())

evaluator_accuracy = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="labels",
predictionCol="prediction")

evaluator_precision = MulticlassClassificationEvaluator(metricName="weightedPrecision", labelCol="labels",
predictionCol="prediction")

evaluator_recall = MulticlassClassificationEvaluator(metricName="weightedRecall", labelCol="labels",
predictionCol="prediction")

evaluator_f1 = MulticlassClassificationEvaluator(metricName="f1", labelCol="labels",
predictionCol="prediction")

binary_evaluator = BinaryClassificationEvaluator(labelCol="labels", rawPredictionCol="rawPrediction")

cv_nb = CrossValidator(

   estimator=final_nb_model,

   evaluator=evaluator_accuracy,

   estimatorParamMaps=param_grid,
```

```
    numFolds=10,
    seed=42
)
cv_model_nb = cv_nb.fit(df_train_balanced)
best_model_nb = cv_model_nb.bestModel
nb_train_pred = best_model_nb.transform(df_train_balanced)
nb_test_pred = best_model_nb.transform(df_test)
nb_train_acc = evaluator_accuracy.evaluate(nb_train_pred)
nb_train_prec = evaluator_precision.evaluate(nb_train_pred)
nb_train_recall = evaluator_recall.evaluate(nb_train_pred)
nb_train_f1 = evaluator_f1.evaluate(nb_train_pred)
nb_train_auc = binary_evaluator.evaluate(nb_train_pred)
nb_test_acc = evaluator_accuracy.evaluate(nb_test_pred)
nb_test_prec = evaluator_precision.evaluate(nb_test_pred)
nb_test_recall = evaluator_recall.evaluate(nb_test_pred)
nb_test_f1 = evaluator_f1.evaluate(nb_test_pred)
nb_test_auc = binary_evaluator.evaluate(nb_test_pred)
print("\nFinal Naive Bayes Model Results:")
print(f"Train Accuracy: {nb_train_acc:.4f} | Test Accuracy: {nb_test_acc:.4f}")
print(f"Train Precision: {nb_train_prec:.4f} | Test Precision: {nb_test_prec:.4f}")
print(f"Train Recall: {nb_train_recall:.4f} | Test Recall: {nb_test_recall:.4f}")
print(f"Train F1 Score: {nb_train_f1:.4f} | Test F1 Score: {nb_test_f1:.4f}")
print(f"Train AUC: {nb_train_auc:.4f} | Test AUC: {nb_test_auc:.4f}")


final_svm_model = LinearSVC(
    featuresCol="final_features",
    labelCol="labels",
```

```
    maxIter=100,
    regParam=0.01,
    fitIntercept=True,
    standardization=True,
    tol=1e-6,
    predictionCol="prediction",
    rawPredictionCol="rawPrediction"
)

# 2. Define a parameter grid for cross-validation (regParam and maxIter)
param_grid = (ParamGridBuilder()
              .addGrid(final_svm_model.regParam, [0.001, 0.01, 0.1])
              .addGrid(final_svm_model.maxIter, [50, 100, 200])
              .build())
# 3. Initialize BinaryClassificationEvaluator for AUC and other metrics
binary_evaluator = BinaryClassificationEvaluator(labelCol="labels", rawPredictionCol="rawPrediction")
evaluator_accuracy = MulticlassClassificationEvaluator(metricName="accuracy", labelCol="labels",
predictionCol="prediction")
evaluator_precision = MulticlassClassificationEvaluator(metricName="weightedPrecision", labelCol="labels",
predictionCol="prediction")
evaluator_recall = MulticlassClassificationEvaluator(metricName="weightedRecall", labelCol="labels",
predictionCol="prediction")
evaluator_f1 = MulticlassClassificationEvaluator(metricName="f1", labelCol="labels",
predictionCol="prediction")
# 4. Set up cross-validation
cv_svm = CrossValidator(
    estimator=final_svm_model,
    evaluator=evaluator_accuracy,
    estimatorParamMaps=param_grid,
```

```
    numFolds=10,  # 10-fold cross-validation
    seed=42  # For reproducibility
)
# 5. Train SVM model on the balanced training data (df_train_balanced)
cv_model_svm = cv_svm.fit(df_train_balanced)
# The best model from cross-validation
best_model_svm = cv_model_svm.bestModel
# 6. Predict on both balanced training and original test data
svm_train_pred = best_model_svm.transform(df_train_balanced)
svm_test_pred = best_model_svm.transform(df_test)
# 7. Training metrics
svm_train_acc = evaluator_accuracy.evaluate(svm_train_pred)
svm_train_prec = evaluator_precision.evaluate(svm_train_pred)
svm_train_recall = evaluator_recall.evaluate(svm_train_pred)
svm_train_f1 = evaluator_f1.evaluate(svm_train_pred)
svm_train_auc = binary_evaluator.evaluate(svm_train_pred)
# 8. Testing metrics
svm_test_acc = evaluator_accuracy.evaluate(svm_test_pred)
svm_test_prec = evaluator_precision.evaluate(svm_test_pred)
svm_test_recall = evaluator_recall.evaluate(svm_test_pred)
svm_test_f1 = evaluator_f1.evaluate(svm_test_pred)
svm_test_auc = binary_evaluator.evaluate(svm_test_pred)
# 9. Output SVM results
print("\nFinal SVM Model Results:")
print(f"Train Accuracy: {svm_train_acc:.4f} | Test Accuracy: {svm_test_acc:.4f}")
print(f"Train Precision: {svm_train_prec:.4f} | Test Precision: {svm_test_prec:.4f}")
```

```
print(f"Train Recall: {svm_train_recall:.4f} | Test Recall: {svm_test_recall:.4f}")

print(f"Train F1 Score: {svm_train_f1:.4f} | Test F1 Score: {svm_test_f1:.4f}")

print(f"Train AUC: {svm_train_auc:.4f} | Test AUC: {svm_test_auc:.4f}")
```