NLP Analytics on IMDB Reviews

Atharva Khaire, Parita Danole

Abstract:

This project is aimed at predicting movie ratings based on the feedback submitted by reviewers. The objective is to perform various pyspark transformations to compile all the data into a single DataFrame and then proceed with NLP operations to generate an overall sentiment towards every single movie. We will be using Apache Airflow for the workflow and then testing various regression and classification models on the transformed data to determine the best model.

1. Introduction:

Sentiment analysis is the use of natural language processing, text analysis, computational linguistics, and biometrics to systematically identify, extract, quantify, and study affective states and subjective information. It is the process of analyzing digital text to determine if the emotional tone of the message is positive, negative, or neutral. Companies have large volumes of text data like emails, customer support chat transcripts, social media comments, and reviews that need to be analyzed to determine opinions about their product, service or idea.

For this project, we are using a Kaggle dataset of IMDb reviews. The Internet Movie Database (IMDb) is an online database containing information and statistics about movies, TV shows and video games. IMDb provides the review feature as a forum where users can freely express their opinions about movies or TV shows. The data is in JSON format, i.e, one JSON document represents one single review for a movie. Then, we will be authoring our own metric to generate a rating for that movie.

Apache Airflow is an open-source workflow management platform for data engineering pipelines. It is used for authoring, scheduling and monitoring data and computing workflows. AWS (Amazon Web Services) is a comprehensive, evolving cloud computing platform provided by Amazon that includes a mixture of infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS) and packaged-software-as-a-service (SaaS) offerings. It offers reliable, scalable, and inexpensive cloud computing services.

2. Project Objectives:

The primary objective of this project is to develop a system for predicting movie ratings through a multi-stage process. Initially, unstructured movie review data in JSON format will be transformed into a structured PySpark DataFrame, facilitating efficient analysis. Then Natural Language Processing (NLP) techniques using PySpark will be employed to conduct sentiment analysis on each review, capturing the overall feelings expressed by reviewers. The project aims to predict movie ratings based on the sentiments identified and automation of the entire workflow will be achieved through Apache Airflow that orchestrates tasks seamlessly. Finally, we will be experimenting with multiple regression and classification models that will be tested on the transformed data to identify the most effective model for predicting the movie ratings.

3. Methodology:

Apache Airflow -

Apache Airflow is an open source tool used for complex task orchestration. It is like a very smart scheduler that knows when to run tasks, how to run them and in the correct order. The most intriguing thing about airflow is the workflow is all determined by the code. We can write steps of our workflow in code which turns out to be very reliable for version control. It's all written in Python which makes it very easy to understand.

Airflow’s primary part is DAG - Directed Acyclic Graph. It's just an assembly of all the jobs we wish to complete, arranged to show the connections and interdependencies between them. A DAG can be thought of as the workflow's blueprint, with each job within it representing a precise location within a complicated structure similar to a block.

Airflow's robust monitoring and logging features are just another fantastic feature. Managing intricate workflows requires us to be fully aware of what's going on at all times. Airflow provides us with a thorough overview of our tasks, their status, and any potential problems. It resembles providing our data pipelines with a flight recorder.

Airflow has an online user interface. This makes managing and keeping an eye on our workflows incredibly easy. Right from our browser, we can literally examine our DAGs, check what's working and what hasn't, and delve deeply into logs.

Extracting TF-IDF features -

Tokenization

Function: Divides the text into discrete words, or tokens. "I love movies," for instance, becomes ["I", "love", "movies"].
Reasons for its significance: It's the initial step towards comprehending the text's structure. We can begin a thorough analysis of the text by dissecting sentences into their constituent terms.

Stop Words Removal

Function: eliminates terms that are frequently useless for analysis, such as "the," "is," "in," etc.
Reasons for its significance: Despite being commonly used, these words have little significance for the particular analysis. Eliminating them facilitates attention to other important terms.

Term Frequency Hashing

Function: By utilizing the hashing method, it transforms the filtered words into numerical feature vectors.
Reasons for its significance: Machine learning algorithms operate on numerical data rather than words. HashingTF allows one to do mathematical operations on words by converting text into a numerical format.

Inverse Document Frequency

Function: Calculates each word's weight inside the dataset. Ones that are more uncommon are given a higher weight than ones that are common.
Reasons for its significance: By taking this step, we can make sure that words that are frequently used in documents like "and" or "the" don't obscure more distinctive words that might be more pertinent to the study. It assists in drawing attention to the terms that, given the dataset's context, are more intriguing.

Classification models -

Logistic Regression:

When used on IMDb reviews, multiclass logistic regression functions similarly to binary logistic regression but can accommodate more than two classes. It's an effective text classification tool, especially for sentiment analysis and topic grouping of movie reviews.

A softmax function is utilized in multiclass logistic regression in place of the typical logistic function used in binary classification. Because it can handle several classes and produce a probability distribution over the classes, the softmax function is utilized. The model calculates the probability $P(y=k|x)$ for each class k given an input review x.

A dataset of labeled reviews is used to train the model. During training, the goal is to minimize a loss function by modifying the model parameters, which are usually the weights assigned to each input feature. A popular loss function for multiclass logistic regression is the cross-entropy function. This function makes a comparison between the actual distribution (the true labels) and the anticipated probability distribution of the model.

Naive Bayes:

When used on IMDb reviews, naive Bayes classifiers are a collection of simple and effective text classification algorithms that support sentiment analysis and theme categorization. These classifiers rely on the 'naive' assumption of conditional independence between each pair of characteristics, given the value of the class variable, and the Bayes theorem.

The 'naive' assumption of independence between each pair of characteristics is applied by Bayes classifiers when applying the Bayes theorem. In terms of text categorization, this refers to the assumption that, given the review's category (such as "Positive," "Negative," or "Neutral," the presence of a given word in a review is independent of the presence of any other word.

Given a certain review, the classifier determines the likelihood of each category and then chooses the category with the highest probability in the context of IMDb reviews. To compute this, use the following formula:
$P(\text{Review}|\text{Category}) = P(\text{Review}|\text{Category}) \times P(\text{Category})/P(\text{Review})$, where $P(\text{Review}|\text{Category})$ is the evidence, $P(\text{Category})$ is the prior probability of each category, and $P(\text{Review})$ is the likelihood.

Decision Tree:

Applying decision trees to IMDb reviews offers a simple and efficient way to classify text, including arranging reviews according to sentiment or thematic material. A decision tree is a type of tree structure that resembles a flowchart, with each leaf node representing the result, the branch representing a decision rule, and the internal node

representing a feature or attribute. These results, in the context of IMDb reviews, would be the various classification categories (such as "Positive," "Negative," and "Neutral").

Choosing which features i.e, words or phrases from the reviews to employ at each decision point in the decision tree is an essential stage in the process of creating a decision tree. Commonly used feature selection criteria, such as information gain or Gini impurity, aid in determining which features best divide the dataset into its various categories.

The dataset is divided into subsets according to the feature that yields the maximum information gain (or lowest Gini impurity) in a top-down manner, beginning with the root node. Every derived subset undergoes a cyclical repetition of this process. When splitting stops adding value to the predictions or when every member of the subset at a node has the same value for the target variable, the recursion is finished.

Random Forest:

When used on IMDb reviews, Random Forest is a powerful and adaptable machine learning method for text classification. It can be used to group reviews according to sentiment, genre, or other thematic components. An ensemble learning technique called a Random Forest works by building a large number of decision trees during training and producing a class that is the mean of the classes (classification) of the individual trees.

To produce a prediction that is more reliable and accurate, a Random Forest constructs several decision trees and combines them. Every tree in the forest is constructed using a bootstrap sample, which is a sample selected from the training set with replacements. Moreover, the optimal split is determined from all input features or a random selection of them when dividing each node throughout the tree-building process. By adding variation, this strengthens the model beyond what a single decision tree might provide.

In conventional decision trees, all features have the potential to be utilized at each tree split. This procedure is further randomized using Random Forest, which divides a node without looking for the most important attribute. It looks through a random subset of features to find the best one. This produces a great deal of variability, which usually makes the model stronger.

Regression Models -

Linear Regression:

When it comes to IMDb reviews, linear regression is usually utilized for prediction jobs where the result is a continuous variable. Although linear regression is typically used to analyze numerical data, it can additionally be modified to handle text data by using different preparation methods.

A linear strategy to model the relationship between one or more independent variables (features extracted from the review text) and the dependent variable (movie ratings) is known as linear regression. The model presupposes a linear relationship between the continuous output variable (Y) and the input variables (X), which can be written as follows: $Y = B0 + B1 X1 + B2 X2 +...+Bn Xn + e$
The coefficients in this case are e0, e1,..., B0, B1,..., Bn, and the error term is e.

Decision Tree Regression:

Using features taken from the review text, decision tree regression can be used to predict a continuous variable, like a movie's rating, when applied to IMDb reviews. Decision tree regression is used for prediction tasks when the target variable is continuous and numerical, in contrast to decision tree classifiers, which are used for categorical outcomes.

Building a tree with internal nodes representing features, branches representing decision rules, and leaf nodes representing the outcome is known as decision tree regression. In the resulting subsets, the algorithm chooses the features and separates them in a way that minimizes the variance of the target variable. Regression trees often have leaf nodes with continuous values, in contrast to decision trees used for classification. The target values of the cases that fall into each leaf are often represented by the mean or median at that leaf.

Random Forest Regression:

Using features taken from the review text, Random Forest regression is a strong and adaptable ensemble learning technique that may be used to predict continuous variables, such as the average rating of a movie, from IMDb reviews. By generating an ensemble of trees, a Random Forest regression model improves on the idea of decision tree regression and helps produce predictions that are more reliable and accurate overall.

During training, Random Forest regression builds a number of decision trees. Every tree is constructed using a distinct random sample of the data, and a random subset of

features is taken into account at each tree split. This unpredictability lowers the chance of overfitting and aids in the creation of a diversified collection of trees.

The algorithm chooses splits that reduce the variation within each node for each tree. Creating branches that lead to subsets of the data with comparable objective values. For a given input, each tree in a random forest regression predicts a continuous result. Usually, the average of all the trees in the forest's projections yields the final prediction.

Gradient Boosted Tree Regression:

When employed on IMDb reviews, gradient boosting tree (GBT) regression is an advanced and potent ensemble machine learning method for predicting continuous variables, like a movie's rating, based on textual elements taken from the reviews. Gradient Boosting builds trees one after the other, with each new tree in the ensemble helping to rectify the mistakes produced by the older ones.

Gradient Boosting develops one tree at a time, in contrast to Random Forest, which grows its trees concurrently. The goal of building each new tree is to reduce the errors or residuals from the ones that came before it. In essence, every tree is picking up lessons from the mistakes made by the trees that came before it.

Gradient Boosting uses an objective that is to minimize a differentiable loss function. This is often the mean squared error in a regression. By building trees that anticipate the loss's negative gradient, the approach applies gradient descent to minimize this loss. The final result of GBT regression is the sum of the predictions made by each tree, which in turn predicts a value. Usually, the combination is a weighted total, with the weights being acquired through training.

K-means Clustering -

When used on IMDb reviews, K-means clustering is a well-liked unsupervised machine learning method for organizing reviews into clusters according to textual content similarities. K-means recognizes innate structures in the data, classifying reviews into clusters without prior knowledge of the categories, in contrast to supervised learning techniques that require labeled data.

To begin, the algorithm randomly initializes k centroids. These centroids, which stand for the cluster centers, are positioned in the same spatial coordinate system as the reviews. Based on a distance metric, usually Euclidean distance, each review is assigned to the closest centroid. The assignment is predicated on the idea that evaluations belonging to

the same cluster ought to be as comparable as feasible. The mean of all reviews inside a cluster is used to recalculate the centroids once all reviews have been assigned to clusters. The centroids are moved in this stage to a location that better represents each cluster.

Iteratively repeat cluster assignment and centroid updation until the centroids settle and the reviews' cluster assignments no longer vary noticeably. The clusters are refined through this iterative procedure to achieve maximum internal homogeneity and maximum inter-cluster differentiation.

LDA -

When applied to IMDb reviews, Latent Dirichlet Allocation (LDA) is an advanced unsupervised machine learning method for identifying abstract concepts in the textual data. Assuming that documents (in this case, IMDB reviews) are mixes of themes and topics are mixtures of words, LDA is a sort of topic modeling. This method makes a valuable contribution to the analysis and classification of IMDb reviews by assisting in the discovery of latent theme structures within extensive text sets.

Using the vectorized data, LDA is trained. As it proceeds through each review, the algorithm designates each word to one of the K subjects. Then, iteratively updating these assignments, it does so by taking into account the number of times a word has been associated with a topic in all papers as well as the number of times a topic has been associated with a document.

LDA produces two results: first, a distribution of themes is expressed for each review. In other words, the model offers a list of the most pertinent subjects for each review together with an indication of how relevant each topic is. Second, the distribution of words for each topic shows which words are most relevant to that particular issue.

4. Implementation with Apache Airflow:

This project required a systematic approach to how certain processes need to be done. So we decided to integrate the tasks with Airflow. Apache Airflow is a task scheduling and management tool which organizes processes in compliance with how we want to run code. So for example, if we are running a classification model, we first need to process the data so that it is understandable by the model. We will just schedule the preprocessing tasks before the model training. For this project, we decided to author 3 DAGs (Directed Acyclic Graph).

- Imdb_preprocessing
- Imdb_model
- Imdb_eda

Now before we dive deep into what each of these DAGs do, we created a helper library called emr_utils.py. Basically, it is a general purpose library which is tailored to interact with AWS Elastic Map Reduce service. The script facilitates creating, monitoring, adding steps and termination of an EMR cluster.

cluster.create(emr,name,worker_nodes) :

This function takes in 3 parameters:

- emr: The EMR client object from the boto3 library.
- name: The name of the EMR cluster.
- worker_nodes: Number of core/worker nodes.

The function sends a request to AWS to create an EMR cluster with one master node (m5.xlarge) and worker nodes (m5.xlarge) which are provided to the function through an Airflow variable. The worker nodes use SPOT instances to reduce the cost. It specifies log storage in S3 (s3://dag-emr-logs/), a subnet ID, and an EC2 key pair for SSH access. It also includes a bootstrap action to install 'textstat' (presumably a software or script located in s3://imdb-cs777/scripts/install-textstat.sh). Hadoop and Spark are specified as applications to be installed on the cluster. The function returns the cluster ID after creation.

```python
def create(emr, name, worker_nodes):

    cluster_config = emr.run_job_flow(
    Name=name,
    ReleaseLabel='emr-6.15.0',
    LogUri = "s3://dag-emr-logs/",
    Instances={
        'InstanceGroups': [
            {
                'Name': "Master node",
                'Market': 'ON_DEMAND',
                'InstanceRole': 'MASTER',
                'InstanceType': 'm5.xlarge',
                'InstanceCount': 1,
            },
            {
                'Name': "Core nodes",
                'Market': 'SPOT',
                'InstanceRole': 'CORE',
                'InstanceType': 'm5.xlarge',
                'InstanceCount': worker_nodes,
            }
        ],
        'Ec2SubnetId': 'subnet-00a64607baeea1ad6',
        'Ec2KeyName': 'Khaire',
        'KeepJobFlowAliveWhenNoSteps': True,
        'TerminationProtected': False,
    },
    BootstrapActions=[
        {
            'Name': 'Install textstat',
            'ScriptBootstrapAction': {
                'Path': 's3://imdb-cs777/scripts/install-textstat.sh',
            }
        },
    ],
    Applications=[{'Name': 'Hadoop'}, {'Name': 'Spark'}],
    JobFlowRole='EMR_EC2_DefaultRole',
    ServiceRole='EMR_DefaultRole',
    )

    cluster_id = cluster_config['JobFlowId']
    return cluster_id
```

cluster.wait_launch(id,emr):

This function takes in 2 parameters:

- id: Cluster ID returned by the create function.
- emr: The EMR client object from the boto3 library.

The function basically pokes the cluster every 15 seconds to check if the cluster has launched or not. If the cluster reaches a 'WAITING' or 'RUNNING' state, it returns 200 (indicating success). If the cluster enters a terminating state, it returns 410 (indicating failure or termination).

```python
def wait_launch(id,emr):

    while True:
        response = emr.describe_cluster(ClusterId=id)
        status = response['Cluster']['Status']['State']

        logging.info(f"Cluster status: {status}")

        if status in ['WAITING', 'RUNNING']:
            return 200
        elif status in ['TERMINATING', 'TERMINATED', 'TERMINATED_WITH_ERRORS']:
            return 410
        else:
            time.sleep(15)
```

cluster.step(id,emr,step_name,s3_path,arg):

This function takes in 5 parameters:

- id: Cluster ID returned by the create function
- emr: The EMR client object from the boto3 library.
- step_name: The name of the step/job that needs to be completed on the cluster.
- s3_path: The path of the script in s3.
- arg: Contains any arguments that need to be submitted to the Spark Job.

The function submits a Spark job to the cluster and then monitors its status. If the job completes successfully, it returns 200; if it fails, is cancelled, or is interrupted, it returns 410.

```python
def step(id,emr,step_name,s3_path, arg):

    step_config = {
        'Name': step_name,
        'ActionOnFailure': 'TERMINATE_CLUSTER',
        'HadoopJarStep': {
        'Jar': 'command-runner.jar',
        'Args': ['spark-submit','--deploy-mode', 'cluster', s3_path, arg]
        }
    }

    response = emr.add_job_flow_steps(JobFlowId=id, Steps=[step_config])
    step_id = response['StepIds'][0]
    logging.info(f"Added step: {step_id}")

    while True:
        step_status = emr.describe_step(ClusterId=id, StepId=step_id)
        state = step_status['Step']['Status']['State']

        logging.info(f"Step status: {state}")

        if state in ['COMPLETED']:
            return 200
        elif state in ['CANCELLED', 'FAILED', 'INTERRUPTED']:
            return 410
        else:
            time.sleep(15)
```

cluster.terminate(id,emr):

This function takes in 2 parameters:

- id: Cluster ID returned by the create function.
- emr: The EMR client object from the boto3 library.

The function sends a request to terminate the specified EMR cluster and returns the response from the AWS service.

```python
def terminate(id,emr):

    response = emr.terminate_job_flows(JobFlowIds=[id])
    print(f"Cluster {id} is being terminated.")

    return response
```

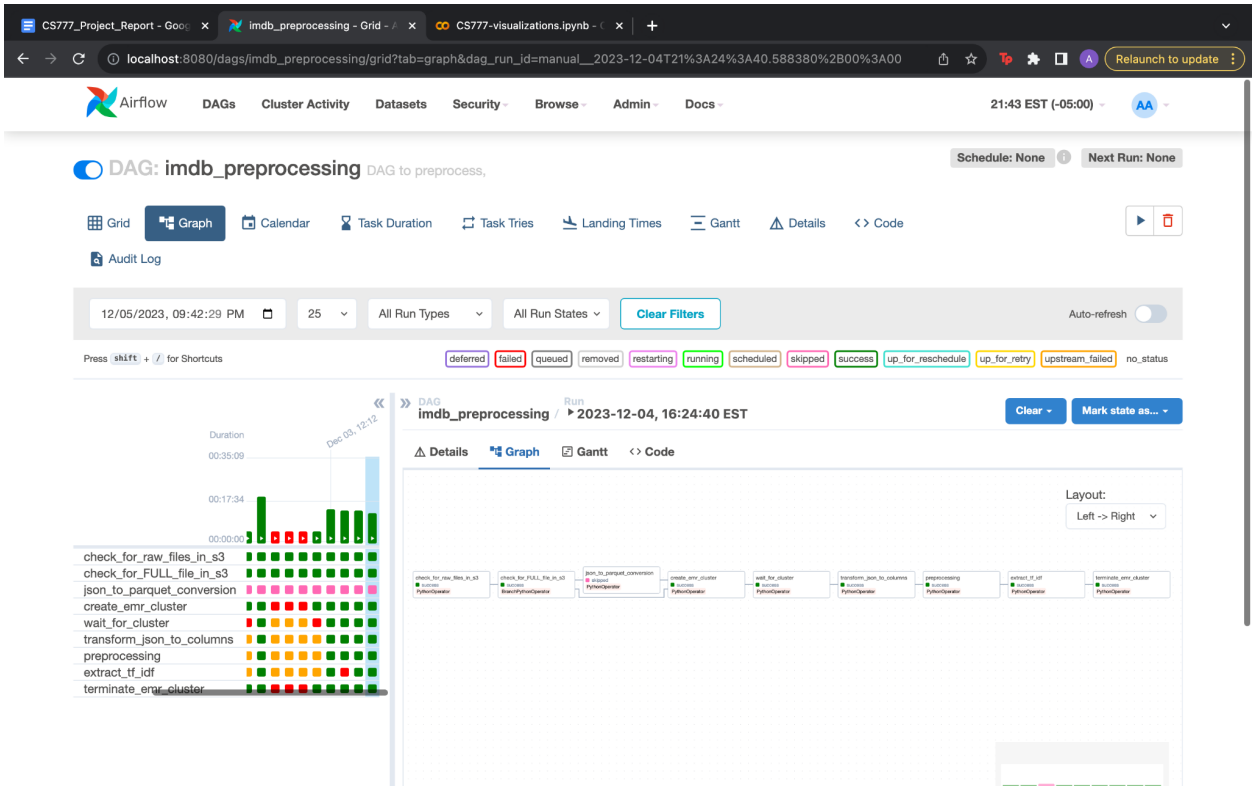
In addition to this code, we implemented the use of config yaml files. So for example -

```yaml
cluster:
  name: IMDB-Preprocessing
  emr-version: emr-6.15.0
  log-uri: s3://dag-emr-logs/
  instances:
    instance-groups:
      master:
        name: Master Node
        market: ON_DEMAND
        instance-role: MASTER
        instance-type: m5.xlarge
        instance-count: 1
      core:
        name: Core Nodes
        market: ON_DEMAND
        instance-role: CORE
        instance-type: m5.xlarge
        instance-count: 2
      task:
        name: Task Nodes
        market: ON_DEMAND
        instance-role: TASK
        instance-type: m5.xlarge
        instance-count: 6
    ec2-subnet-id: subnet-00a64607baeea1ad6
    ec2-key-name: Khaire
    keep-job-flow-alive-when-no-steps: True
    termination-protected: False
  bootstrap-actions:
    name: install textstat
    path: s3://imdb-cs777/scripts/install-textstat.sh
  applications: spark,hadoop
  job-flow-role: EMR_EC2_DefaultRole
  service-role: EMR_DefaultRole
```

The above config file is for preprocessing DAG. By using the above, we can change the EMR cluster configuration whenever we want and with simplicity. We use the PyYAML library to read the configs.

imdb_preprocessing DAG -

Kindly refer to the code file named imdb_preprocessing.py for the DAG architecture. This DAG is responsible for all the tasks that need to be completed in order to feed the data to the models. This involves, converting json to parquet, checking if the files already exist in S3, creating an emr cluster, transforming json to columns, text preprocessing and extracting tf-idf features.



First, we import all the necessary libraries. This includes all the airflow packages and boto3. Boto3 is an SDK for all AWS services. This is the library through which we upload files to S3, create clusters, run spark steps etc.
Next we import the airflow variables. These are the variables that are set in the Airflow UI.

RUN_MODE: Specifies the run mode of the DAG. If set to ‘full’, all the raw files in the S3 bucket will be read. If set to ‘select’, only the files specified in the FILE_LIST variable are read. This variable is useful when we just have to check if the code is working on EMR for a small dataset.

AWS_ACCESS_KEY: Your AWS access key that is required by the boto3 functions to work.

AWS_SECRET_KEY: Your AWS secret key which serves the same purpose as AWS_ACCESS_KEY.

BUCKET: The bucket name where all the data files, results, temporary results, etc. will be stored.

RAW_FOLDER: The folder name where the DAG will look for the raw files.

Next, we declare two boto3 clients, s3 and emr. S3 handles operations related to S3 and emr does the same for EMR.

Tasks:

Check_for_raw_files -

This task first generates a list of all the json files contained in the raw bucket. If the RUN_MODE is full, it just checks if the list is not empty. If the RUN_MODE is select, only those files listed in the FILE_LIST variable will be checked if they are present in the folder in S3. After the necessary conditions are fulfilled, the task will be marked success.

check_for_FULL_file_in_s3 -

This task is very important in order to reduce the time required to complete the DAG. This task essentially checks if a previous run had already generated a processed full raw file, so that the time required to convert json to parquet is reduced which normally takes anywhere between 30-45 mins. If this task finds the processed raw parquet files, it skips the json_to_parquet_conversion tasks and goes straight to the create_emr_cluster task.
Json_to_parquet_conversion -

This task is responsible for converting json files to parquet. The individual json blobs get stacked on top of each other in separate rows in a Pandas dataframe and this dataframe is then further converted to parquet file so that the spark steps can easily read them.

Create_emr_cluster -

This task creates an AWS EMR cluster without going manually to the interface and creating one. This task uses the emr_utils library that we created which relies on Boto3 to create a cluster. This task passes the path of the config file named preprocessing.yml to the create function and this function then reads the config file and parses it to create a cluster according to our specifications. After the cluster is created, it returns the cluster id which can then be used by other tasks.

Wait_for_cluster -

After a cluster is created, it takes some time to properly launch it. This task pokes the cluster every 15 seconds to check if the cluster has started and once it gets the status as ‘Waiting’, this task gets marked as success and the control goes to the next task.

Transform_json_to_columns -

This task converts the json blobs to columns. For example -

{
9 items
"review_id":
string"rw5704482"
"reviewer":
string"raeldor-96879"
"movie":
string"After Life (2019–)"
"rating":
string"9"
"review_summary":
string"Very Strong Season 2"
"review_date":
string"3 May 2020"
"spoi
ler_tag":
int0
"review_detai
l":
string"I enjoyed the first season, but I must say I think season 2 is even stronger. Ricky does a great job as both writer, actor and director and brings out the best in a superb supporting cast. If there was one thing I'd change, I'd like to hear him talk about himself less with other people and speak more in the third person, but other than that it's pretty hard to fault this funny yet emotional comedy."
"helpful":[
2 items
0
:
string"1"
1
:
string"1"
]

The above json blob gets converted to -

review_id	reviewer	movie	rating	review_summary	review_date	spoiler_tag	review_detail	helpful

Through this task, the spark script stored locally at ‘dags/spark_scripts/transform.py’ gets passed to the step function in emr_utils. This script is then uploaded to s3 and is then read by the cluster and added as a step. This way, we don't have to keep constantly uploading files to s3 whenever we need to make changes to the code.

After its done converting, the transformed reviews are saved in s3 so that the consequent tasks and even other DAGs can work on it.

Preprocessing -

The way this task works is similar to how all the spark steps work. The spark script does the following -

- Creates a UDF(User defined function) to classify the ratings into categories. (>6.66 - 3, >3.33 - 2, <3.33 - 1).
- Filtering the records which have non null ratings
- Converting ratings to float
- Removing all special characters, spaces, escape keys from review_detail
- Change all characters to lower case
- Filtering all the records, where cleaned review’s length is greater than 1000 to only get meaningful reviews

Extract_tf_idf -

The spark script does the following -

- Tokenizing the review_detail column contents
- Removing the stop words like articles, conjunctions etc.
- Creating a term frequency column containing the frequencies of each word in the review

- Creating an inverse document frequency column containing the frequencies of each word in a document throughout all reviews.
- Creating a pipeline to do all of the above.
- Fitting the pipeline on the data.
- Selecting only relevant columns for the model.(features, review_id, rating_category)

Terminate_emr_cluster -

This task uses the emr_utils to terminate the cluster by passing the cluster id previously supplied by the create_emr_cluster task.

After all the task methods have been declared, we define the task configurations which actually run the python functions. All the spark tasks are given a trigger_rule as none_failed so that a single task failure will automatically redirect to the termination task in order to reduce the costs of cluster idle time.

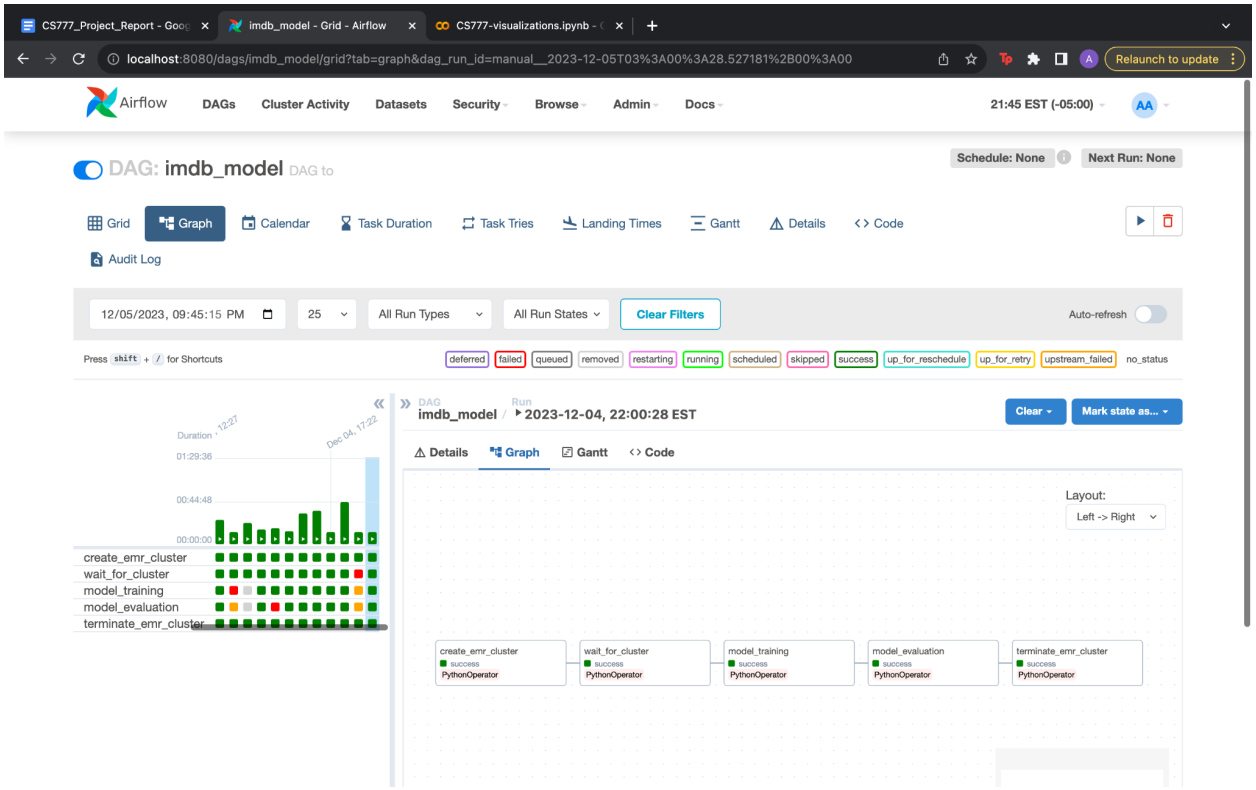
Finally we define the task workflow, so that tasks are executed in the order that we intend them to.

```
check_for_raw_files_task >> check_for_FULL_file_in_s3_task

check_for_FULL_file_in_s3_task >> json_to_parquet_conversion_task
check_for_FULL_file_in_s3_task >> create_emr_cluster_task

json_to_parquet_conversion_task >> create_emr_cluster_task
create_emr_cluster_task >> wait_for_cluster_task >> transform_json_to_columns_task
transform_json_to_columns_task >> preprocessing_task >> extract_tf_idf_task >>
terminate_cluster_task
```

Imdb_model DAG -

This DAG is pretty straightforward. It is created to train the model of our choice and evaluate as well. Kindly refer the imdb_model.py file to understand the DAG architecture. This DAG has the same create_emr_cluster and wait_for_cluster tasks as the previous DAG.

We decided to separate the preprocessing and the model operations, so that while testing we don't have to go through preprocessing all over again when there are changes made only to the model.



First, we import all the necessary libraries. Then we get the airflow variables. The most important variable here is the MODEL variable. Now, in this project we are running multiple models to get the best accuracy for semantic classification. To run a specific model, we will just change the MODEL variable to lets say 'LogisticRegression' etc.

The code will automatically locate the spark script pertaining to the model.

<u>Tasks:</u>

Model_training -

This task first reads the Airflow variable MODEL. Based on the model name, it will locate the script in the local system and upload it to s3. This will remove any human intervention needed to update the code. Once it's uploaded, the script path in s3 will be passed to emr_utils step function and the spark job will be triggered. Based on the type of model, different task times will be incurred

Model_evaluation -

Once the model is done training, the previous task will store the predictions in a file which will be read by this script. Then using pyspark.ml evaluation libraries, it will perform the necessary evaluations.

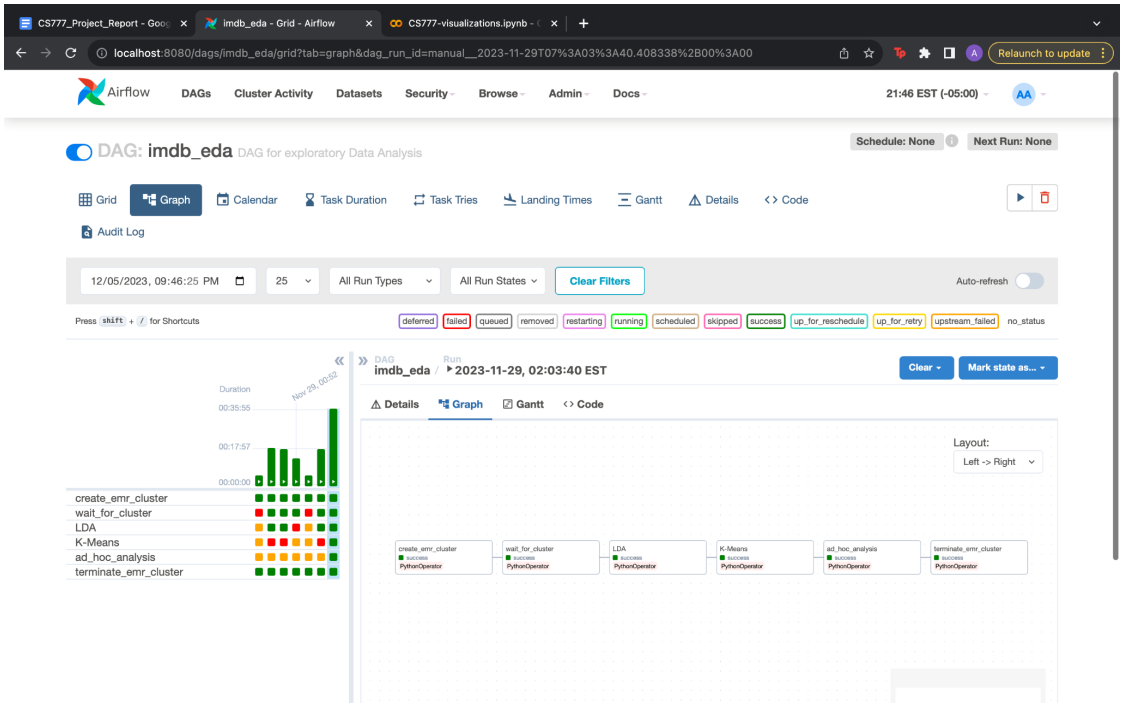
If the model is Linear Regression, it will calculate RMSE using the RegressionEvaluator and RMSE will be calculated both for the training and test set.

If it's any other model, it will calculate the accuracy for both training and test set using the MultiClassClassificationEvaluator.

As for the models, descriptions about that have already been explained before.

Imdb_eda DAG -

This DAG is designed to do some analysis on the data (both preprocessed and raw). We created this DAG to gain more insight into the data we are working on, see the trends and patterns in the data. This DAG is for 3 things - Latent Dirichlet Allocation for topic modeling, Kmeans clustering to group similar reviews so that we can understand how the ratings are distributed in terms of review content and some ad hoc analysis like readability scores, sentiment scores, statistical analysis, correlation etc.



Tasks:

LDA -

This task groups reviews that belong to a certain topic. It is important to note that the LDA algorithm does not give us a topic name. We ourselves have to infer from the top 10 most frequent words the topic contains and therefore we have written the code that saves those top 10 words for 10 topics. LDA.py does the following:

- Tokenize the reviews
- Remove the stop words
- Generating a count of each and every words occurrence in the review
- Generating a count of each and every words occurrence in all the reviews
- Fitting the LDA model (10 topics and 10 iterations)
- Generating a list of top 10 terms in all the topics
- Saving this to S3

Kmeans -

This task is responsible for another type of grouping. We use the K-means clustering algorithm which groups similar reviews together in the hope that the reviews with same rating or rating_category get grouped together. This task does the following -

- Merges the review_length and ratings column together into a vector column
- Standardizes the vector column by eliminating the mean and perform scaling to a unit variance
- Fitting the Kmeans model using the scaledfeatures column
- Getting the cluster centers and the cluster counts.

Ad_hoc_analysis -

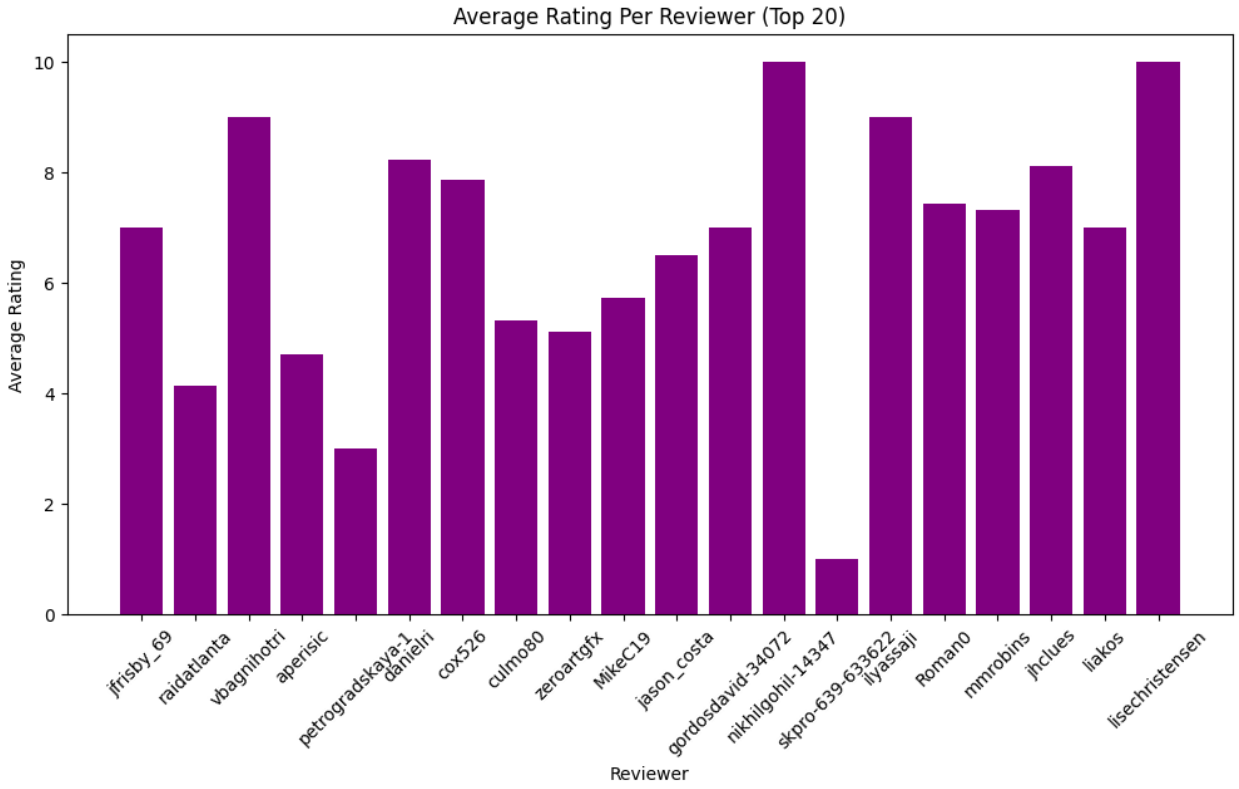
This task performs the following -

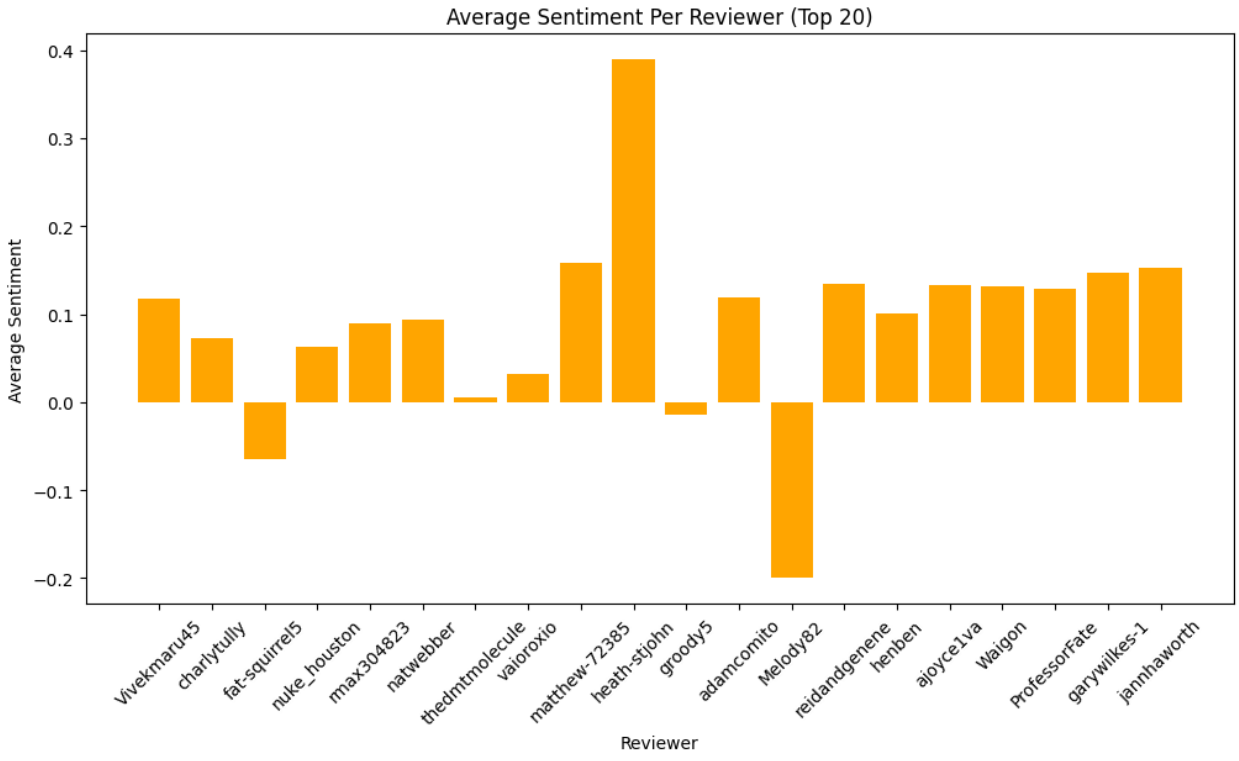
- Generating a list of top 20 most frequent words throughout all reviews with their counts.
- Average Rating per movie
- Readability Score
- Sentiment Score
- Review count per reviewer
- Average rating per reviewer

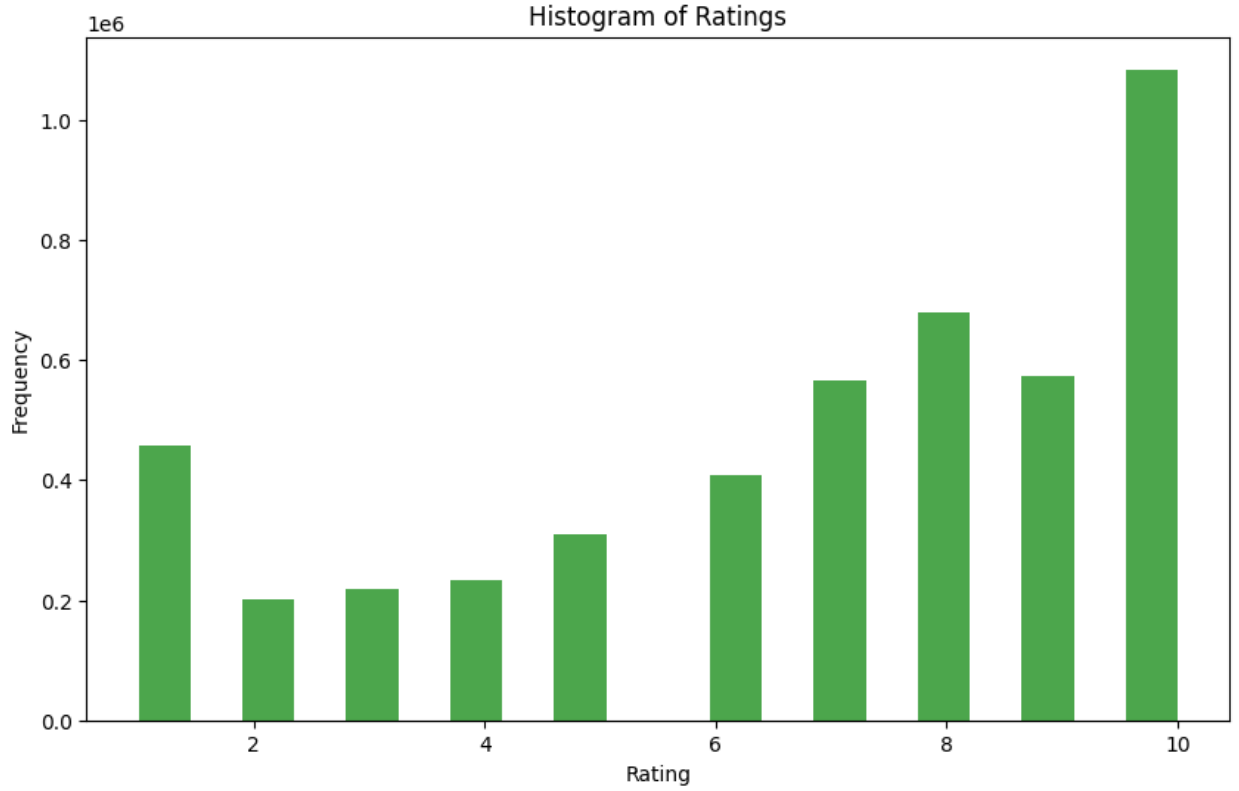
- Average sentiment per reviewer
- Correlation Analysis

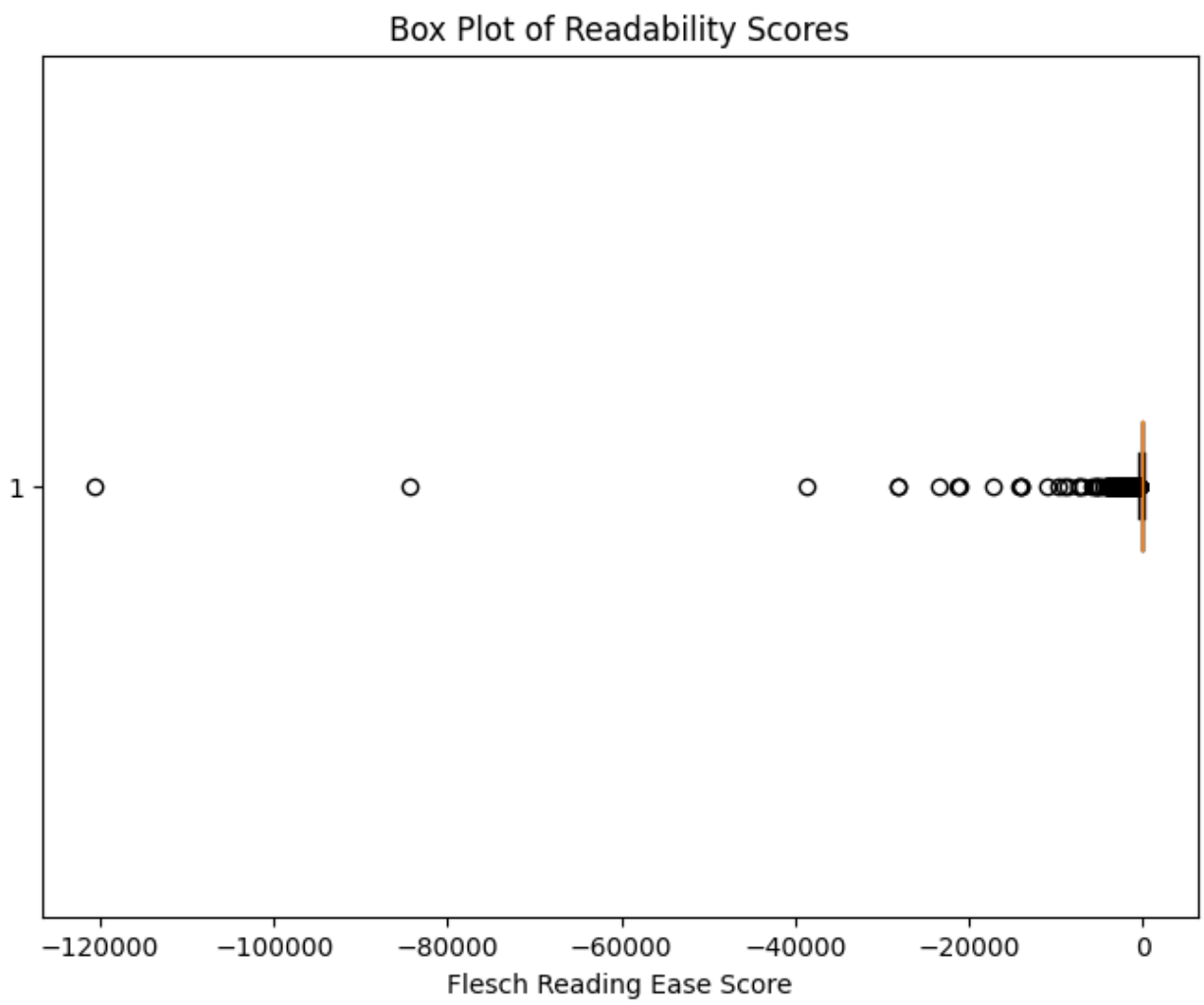
5. Analysis:

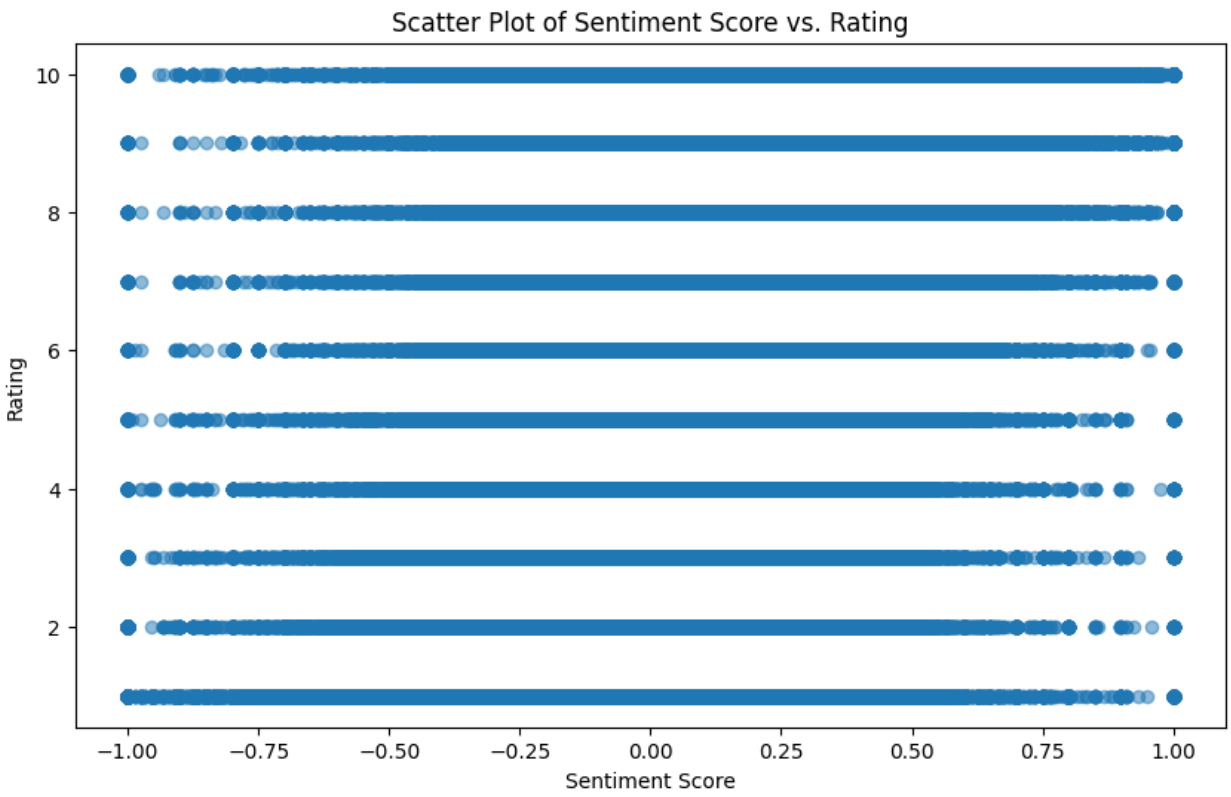
For understanding the key characteristics of the data, we used the following visualizations:

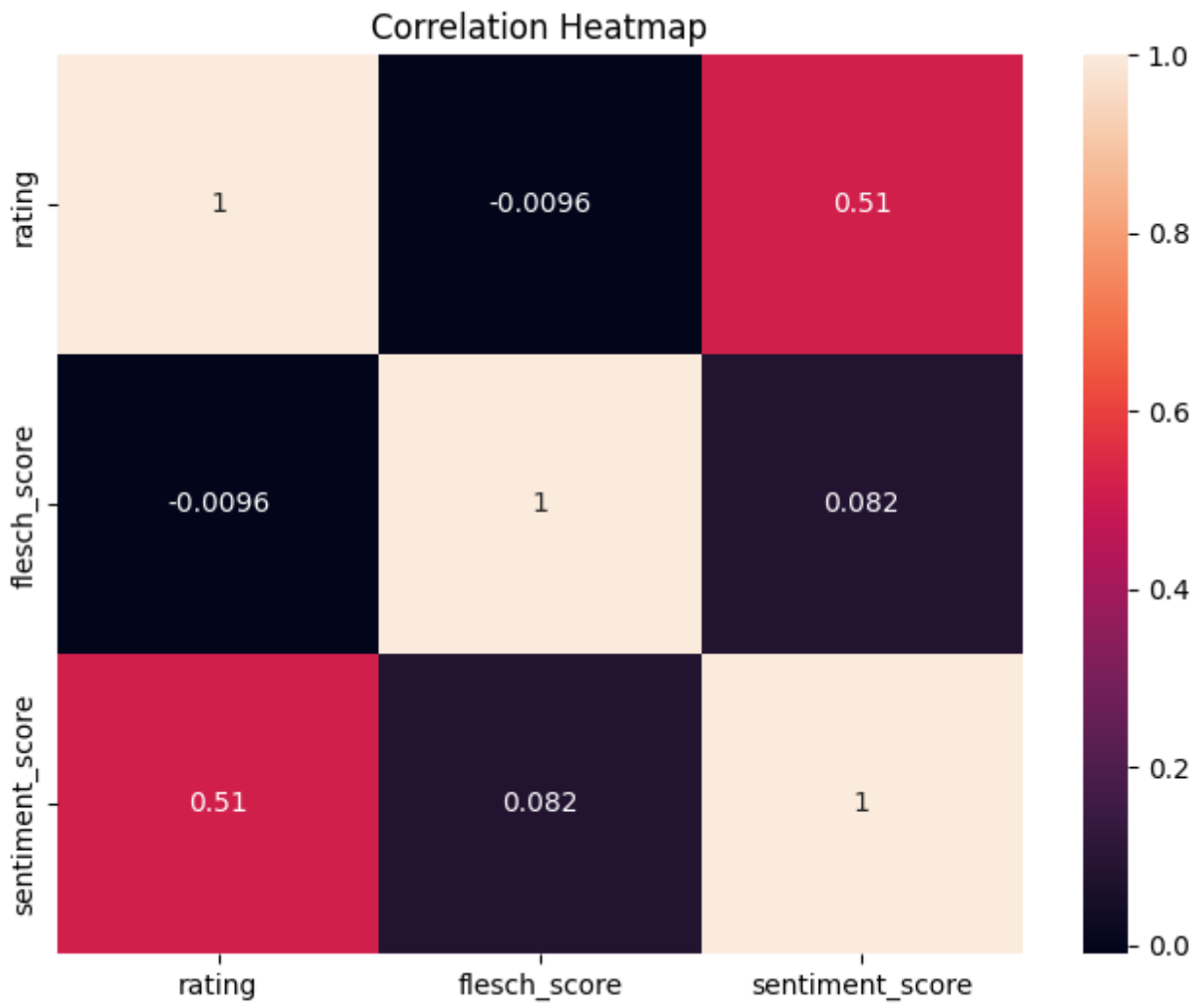


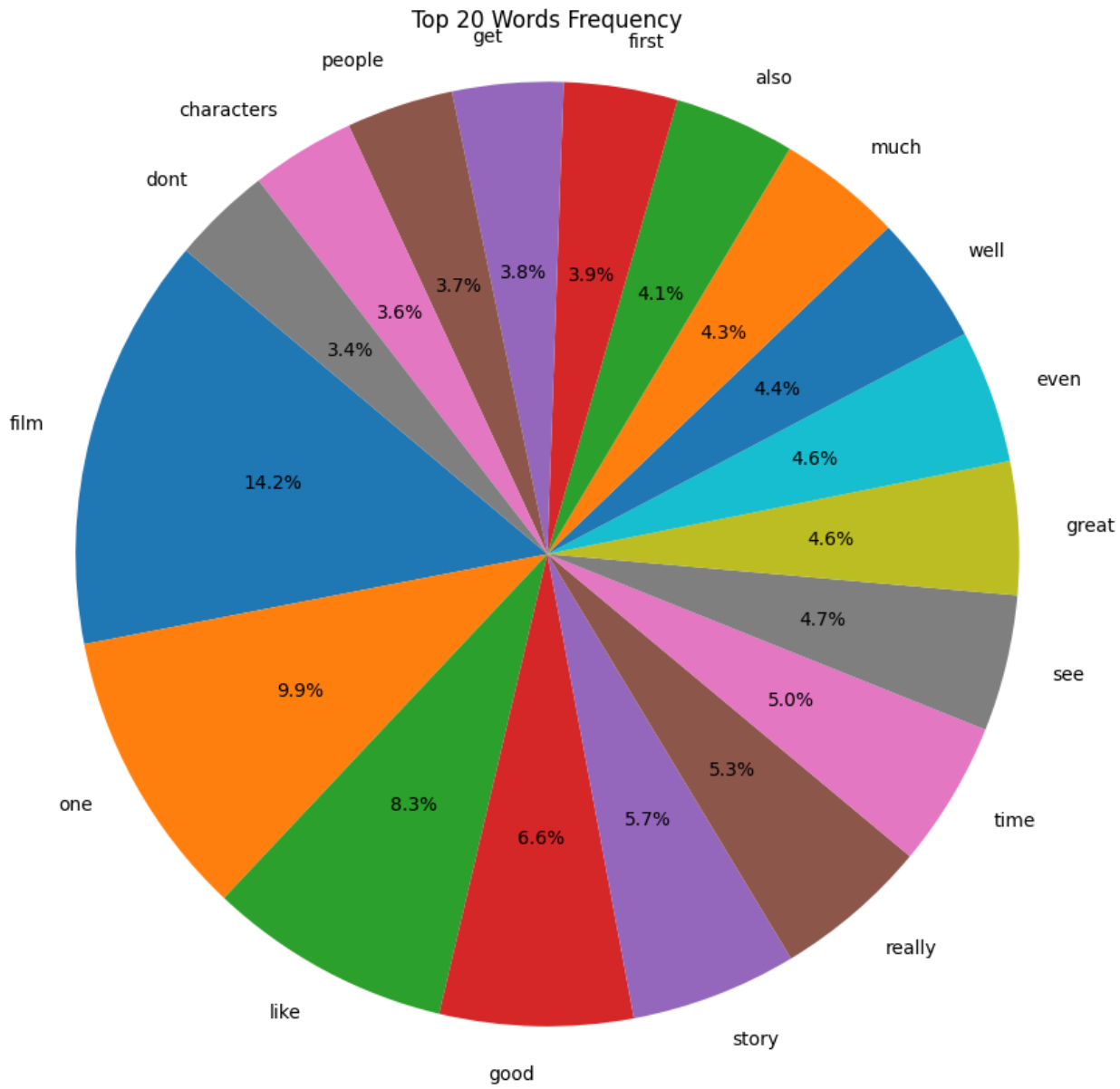

Average Sentiment Per Reviewer (Top 20)
Average Sentiment
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
Vivekmaru45
charlytully
fat-squirrel5
nuke_houston
max304823
natwebber
thedmtmolecule
vaioroxio
matthew-72385
heath-stjohn
groody5
adamcomito
Melody82
reidandgenene
henben
ajoyce1va
Waigon
ProfessorFate
garywilkes-1
jannhaworth
Reviewer


Histogram of Ratings
1e6
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
Rating


Box Plot of Readability Scores
1
−120000
−100000
−80000
−60000
−40000
−20000
0
Flesch Reading Ease Score


Scatter Plot of Sentiment Score vs. Rating
Rating
10
8
6
4
2
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Sentiment Score


Correlation Heatmap
rating
flesch_score
sentiment_score
1
-0.0096
0.51
-0.0096
1
0.082
0.51
0.082
1
rating
flesch_score
sentiment_score
1.0
0.8
0.6
0.4
0.2
0.0


Top 20 Words Frequency
get
first
people
also
characters
much
dont
well
even
film
great
see
one
time
really
like
story
good
14.2%
9.9%
8.3%
6.6%
5.7%
5.3%
5.0%
4.7%
4.6%
4.6%
4.4%
4.3%
4.1%
3.9%
3.8%
3.7%
3.6%
3.4%

6. Results:

Classification models:

Model	Training Accuracy (in %)	Testing Accuracy (in %)
Logistic Regression	69.3	68.9
Random Forest Classification	92.7	91.4
Naive Bayes	15.1	9
Decision Tree Classifier	65.5	50.6

Regression models:

Model	Training RMSE	Testing RMSE
Linear Regression	0.952	1.066
Random Forest Regression	0.505	0.678
Gradient Boosted Tree Regression	0.7	0.65
Decision Tree Regression	0.798	0.791

7. Conclusion:

For classification models, Naive Bayes had the worst testing accuracy of 9%, while random forest classification had the best testing accuracy of 91.4%. This might be because Naive Bayes assumes independence among features For regression models, Linear Regression model had the worst RMSE of 1.066 while Gradient Boosted Tree Regression model had the best RMSE of 0.65. From our analysis, we found classification models were better suited for our dataset and subsequent tasks.

8. References:

[1] Pouransari, H. and Ghili, S., 2014. Deep learning for sentiment analysis of movie reviews. CS224N Proj, pp.1-8.
[2] Shaukat, Z., Zulfiqar, A.A., Xiao, C., Azeem, M. and Mahmood, T., 2020. Sentiment analysis on IMDB using lexicon and neural networks. SN Applied Sciences, 2, pp.1-10.
[3] Enam Biswas. (2021). IMDb Review Dataset - ebD [Data set]. Kaggle. https://doi.org/10.34740/KAGGLE/DSV/1836923