

**Applied Analytics Assignment**

Diploma in Cybersecurity & Digital Forensics

Diploma in Data Science

Diploma in Information Technology

Year 2/3 (2025), Semester 3/5

**~~TEAM~~/INDIVIDUAL REPORT**

\*delete where applicable

(40% of AA Module)

**Deadline for Submission:**

**Presentation Slides: 10th August 2025 (Sunday),23:59hrs**

**Report & Code: 10th August 2025 (Sunday),23:59hrs**

|  |  |  |  |
| --- | --- | --- | --- |
| Tutorial Group | : | T01 | |
| Team Number | : |  | |
| Tutor | : | Dr Wang Siqi | |
| Members | : | Student No. | Student Name |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Penalty for late submission:**

10% of the marks will be deducted every day after the deadline.

**NO** submission will be accepted after 15th August 2025, 23:59 hrs.

1. Introduction

Text Classification is a foundational task in Natural Language Processing, with applications ranging from spam detection to sentiment analysis and content recommendation. For this assignment, the aim is to explore text classification techniques by building a model that can predict a movie’s genre based on its text description – focusing on applying text analysis techniques in Python to process, understand and model movie descriptions extracted from IMDb.

## Problem Understanding

The core objective is to classify movies into one of the five genres – action, comedy, documentary, drama or thriller – using the movie’s description. This is a multi-class classification problem where each movie belongs to one of the genres mentioned. The dataset consists of 5000 movie entries, evenly distributed across the five genres, with 1000 in each. Each entry consists of three main fields: Title, Genre and Description. As the focus is on the textual data, the “Description” field will be the primary input feature used for training my models. The text data to understand movie descriptions is often unstructured, noisy and may contain domain specific language, slang or idiomatic expressions. Additionally, genre boundaries can overlap – whereby some action movies may contain thriller elements, thereby adding further complexity to this classification.

Therefore, it is crucial to clean and preprocess the textual data so that it can be understood that it is suitable for to be used by the Machine Learning algorithms.

* 1. Approach

# Text Data Preprocessing

## Load and Cleanse Text Data

### 2.1.1 Packages & Modules

To start our Text Data Preprocessing, I first installed all the required Python packages using “pip”. Using “%” before the command allows us to run shell commands directly inside the notebook cell. These would be the packages that I require moving forward in the assignment.

A screen shot of a computer

AI-generated content may be incorrect.

Next, I import all required modules for the features that I will be using.

A screen shot of a computer program

AI-generated content may be incorrect.

### 2.1.2 Data Loading & Initial Exploration

To begin, I read the CSV file that I will be using.



Before starting with the preprocessing, I did some initial exploration of the file provided to us. Firstly, I looked at the first few lines of the CSV file given to us using .head(). This provided me with an idea of what the CSV file contained.

A black screen with white text

AI-generated content may be incorrect.

Next, I began with checking if there were any duplicates in the “Description”. While there can be 2 movie titles of the same name, there would unlikely be any movies with the exact same description word for word. Using .duplicated().sum(), it found 4 duplicated descriptions. With this, I used .drop\_duplicates to help remove any of the duplicates yet keeping the first instance of each. Whilst there were not many duplicates, there was still a chance that these keywords could affect the later parts of this assignment; hence it was good to remove them still. Thereafter, there were no more duplicates.

A computer screen with a black background

AI-generated content may be incorrect.



A screen shot of a computer

AI-generated content may be incorrect.

Another check I did was to look for any empty values within the cells. Using .isnull(), I did not find any empty cells.

A screen shot of a computer

AI-generated content may be incorrect.

After these checks, I used .value\_counts() to see how many rows of data I was left with. Due to removing the duplicates, the “Drama” genre now only has 996 entries, whereas the rest of the genres have 1000 entries.

A screenshot of a computer

AI-generated content may be incorrect.

### 2.1.3 Text Preprocessing Pipeline

The first function of my text preprocessing pipeline is to retrieve the stopwords from “stopwords.txt”. Before retrieval, to ensure consistency, using .strip() and .lower(), it strips any whitespace and convert all the words to lowercase respectively. The function returns a frozenset of the stopwords – whereby it is immutable and more efficient for faster checks during filtering.

A computer screen with colorful text

AI-generated content may be incorrect.

The next function is to clean and normalize text from the CSV file. The goal is to ensure consistency across the dataset, which is crucial for accurate text processing. To begin, using unidecode(), it removes any accented characters and converts them into their closest ASCII equivalent, standardizing words across different languages and formats. Next, using .lower(), it converts all characters to lowercase so that all previously capitalized words were treated uniformly like its non-capitalized versions. Lastly, using re.sub(), it removes any HTML/XML tags, eliminates all non-alphabetic characters, removes any extra spaces and replaces them with a singular space.

A computer screen with text and symbols

AI-generated content may be incorrect.

The next function is to tokenize the text. Using word\_tokenize(), it tokenizes each description into a list of word tokens.



The next function is to tag each token with its corresponding Part-Of-Speech tag. Using these tags, it can help the lemmatizer distinguish between the different grammatical uses of a word – through identifying whether the word is a noun/verb/adjective/adverb. If unable to distinguish, it will default to tagging it as a noun. It will then return it in the format expected by the WordNet lemmatizer.

A computer screen with text

AI-generated content may be incorrect.

Since the input tokens have already been tagged with their POS labels, the lemmatization process is context-aware and therefore more accurate. Lemmatization reduces each token to its base or dictionary form, maintaining consistency across the text by removing variations of the same word. The resulting lemmatized words are stored in a list and returned as the output of the function. For this assignment, I chose lemmatization over stemming because lemmatization returns valid dictionary words, whereas stemming only removes word endings, producing non-real words. As real words are important for the analysis, lemmatization would be the more appropriate choice.

A screen shot of a computer program

AI-generated content may be incorrect.

This is my full pipeline for the text preprocessing. To start off, I retrieve and create a frozenlist of the stopwords using my get\_stop\_words() function – parsing through “stopwords.txt”. The main function, full\_text\_pipeline() is then ran and applied to each movie description. In sequence, it cleans text, tokenizes it into individual words, POS-tagged, lemmatized for context-aware normalization and stopwords are removed. I chose to apply lemmatization before removing stopwords in my preprocessing pipeline based on my research. Lemmatizing first ensures that all stopwords are removed, as during lemmatization, many words are reduced to the base or dictionary form, which may end up being listed as stopwords too. If stopwords were removed before lemmatization, some words that were not initially recognized as stopwords could potentially be converted into their base forms that are stopwords. This would cause those words to remain in the text and potentially skew the analysis. Through lemmatization first, the pipeline achieves a more thorough and accurate removal of stopwords.

The function returns two outputs – whereby 1 is a list of the filtered tokens and another as a cleaned string where the tokens are joined by spaces. These outputs are then stored as new columns in the DataFrame. Using .len() for my description\_tokens, it calculates the number of tokens in each list.

A computer screen shot of text

AI-generated content may be incorrect.

Using description\_tokens, I calculated the average number of tokens left after preprocessing. There was an average of 68 tokens per description.

A screen shot of a computer

AI-generated content may be incorrect.

### 2.1.4 Bag of Words

Bag of Words is a fundamental technique for representing text data in a numerical format. It works by creating a vocabulary of all unique words across the dataset and then representing each document as a vector based on the frequency of these words. Each position in the vector corresponds to a specific word, and the value represents how many times that word appears in the document.

For Bag of Words, using CountVectorizer, to construct the Bag of Words representation. The vectorizer is initialized with 2 key parameters whereby the max\_df = 0.16 and max\_features = 5000. The max\_df parameter ensures that any word appearing in more than 16% of all documents is ignored, as these high-frequency words are often not informative. I chose this value as through testing, I figured that this number provided a good balance of words. The max\_features parameter limits the vocabulary to the 5000 most frequent terms which meet the criteria, thereby reducing dimensionality and focusing on the most relevant words. The vectorizer is then fitted using fit\_transform() which first scans through the entire corpus to learn the vocabulary – which match the specified parameters, and assigns each word a unique index based on its frequency. Secondly, it uses that learnt vocabulary to convert each description into a numerical vector that represents the frequency of each word in the document. The result is a sparse matrix, where each row corresponds to a movie description and each column corresponds to a word from the vocabulary. The values in the matrix represent the number of times each word appears in each description.



To identify the most frequently occurring words in the dataset, I calculated the total word frequencies across all movie descriptions using the Bag of Words matrix. By summing the matrix along the document axis, I obtained the total count of each word across the entire corpus. Using the vocabulary generated by the CountVectorizer to map each word to its corresponding index in the matrix, I created a list of tuples, where each tuple contains a word and its total frequency. Finally, sorting this list in descending order of frequency to highlight the most common terms in the dataset.

A screenshot of a computer program

AI-generated content may be incorrect.

To make Bag of Words more interpretable, I created a new DataFrame, using the actual vocabulary terms as column headers by passing “get\_feature\_names\_out()” to the columns parameter. This made the DataFrame much more interpretable, with each column representing a word from the vocabulary and each row corresponding to a specific movie description. The values in the DataFrame represent the frequency of each word with respect to the column.



### 2.1.5 TF-IDF

TF-IDF is a weighting scheme that reflects how important a word is to a particular document in a collection – assigning higher weights to words that are frequent within a document but not common across all documents, allowing more relevant and distinctive terms to stand out. Using TfidfTransformer, with the parameters use\_idf = True to enable inverse document frequency scaling, and smooth\_idf = True to apply smoothing, which avoids division by zero and stabilizes the weights for rare items. Using fit\_transform(), the transformer calculated the IDF values across the dataset and then applied the full tf-idf transformation to the word count matrix. The resulting matrix contains weighted values for each term in each description, providing a numerical representation that better captures the relative importance of words across the entire dataset.



To make TFIDF more interpretable, I created a new DataFrame using the resulting sparse matrix using .toarray(). In this format, where each row corresponds to a document and each column corresponds to a term from the learned vocabulary. The values in the DataFrame represents the TF-IDF weights, indicating the relative importance of each term within a specific document.



To examine the TF-IDF further, I identified the words with the lowest and highest TF-IDF scores across all descriptions. I first computed the maximum TF-IDF value for each term across the entire dataset before sorting the terms based on these values. After sorting, I displayed the top and bottom 20 terms which provided valuable insight into which words contributed most and least to distinguishing content within the dataset.

A computer screen with colorful text

AI-generated content may be incorrect.

# Text Data Understanding

## Keywords Extraction

Before starting the keyword extraction for each individual movie description, I created a new list to store the extracted results and decided on the number of keywords to retrieve per description. I chose to extract the top 40 keywords for each entry. This number was based on the average number of tokens per description, which was approximately 68. Selecting 40 words captures around 60% of each description’s content, which I found to be a good balance as it retained enough meaningful words to summarize the description effectively while filtering out the less important ones.



To extract the most relevant keywords from each movie description, I used a for loop to iterate over all descriptions and retrieved their corresponding TF-IDF vectors. For each document, I created a temporary table that listed the indices of the words that appeared in the description, along with their TF-IDF scores. I then sorted this table in descending order to get the words with the highest importance. From this sorted list, I selected the top 40 keywords as decided on earlier, and matched each index back to the actual word using the vocabulary. I stored both the word its corresponding TF-IDF score, rounded to three decimal places, in a dictionary. This dictionary represents the most important terms for that specific description and was appended to a list. Once all descriptions have been processed, I added the full list of keyword dictionaries as a new column named “keywords” in the original DataFrame.

A computer screen with many colorful text

AI-generated content may be incorrect.

After extracting the top keywords for each description, I converted these into a transaction style format keeping only the words and ignoring their TF-IDF values. Each document’s keywords were stored as a list, representing a transaction of items. Then, I expanded these lists into a DataFrame where each row corresponds to a document and each column holds one of the top keywords for that entry. This format was saved to a CSV file named “transactions.csv” which will be used later for association rule mining.

A screen shot of a computer code

AI-generated content may be incorrect.

## 3.2 Association Rule Mining based on keywords

To start my Association Rule Mining, using the previously saved CSV file “Transactions.csv”, I read the file.



Moving on, I extracted the keywords from each document, ignoring the TF-IDF scores using .apply() to convert them into a list of words. To convert this list of transactions into a suitable format for Association Rule Mining algorithms like Apriori, using TransactionEncoder(), helps to transform the list of transactions into a one hot encoded matrix, where each row corresponds to a movie and each column corresponds to a keyword. The resulting array was then converted into a DataFrame.

A screen shot of a computer program

AI-generated content may be incorrect.

Next, I filtered the encoded dataset to retain only the most frequently occurring keywords to reduce dimensionalities during Association Rule Mining. This was done by summing each column in the binary matrix to calculate the total number of descriptions in which each keyword appears. After sorting the keywords by frequency in descending order, I selected the top 100 keywords to focus on the most representative and meaningful terms across all descriptions. The original one hot encoded DataFrame was then filtered to include only these top keywords.

A computer code on a black background

AI-generated content may be incorrect.

To determine an appropriate minimum support threshold, I used a for loop to run through the Apriori algorithm multiple times using different support values ranging from 0.05 to 0.002. For each support value, the algorithm scanned the filtered dataset of the top 100 keywords to identify the number of itemsets generated for each threshold. This allowed me to view what would be a good threshold for my support level.



After testing with the various support thresholds, I decided to use a minimum support value of 0.002 to generate my frequent itemsets. Although a support of 0.003 produced around 2320 itemsets, there was a significant increase in the number of itemsets to 3939 when dropping to 0.002, which indicated the presence of many potentially interesting keyword combinations that would be missed out with a higher slightly threshold. While 0.002 may be a relatively low support value, I found it to be a good balance whereby it was low enough to capture a wider range of itemsets, yet high enough to ensure that these itemsets were not too rare. To generate these itemsets, I applied the Apriori algorithm to the one hot encoded keyword matrix, using column names to maintain keyword readability.



After generating the frequent itemsets, I used the association\_rules() function to derive meaningful relationships between keywords with lift as the evaluation metric and setting the minimum threshold to 1 to exclude any negatively associated rules. Using this, I managed to obtain 6956 different association rules and it was hard to determine which were interesting if based only off the lift level of 1.

A screenshot of a computer

AI-generated content may be incorrect.

Hence, I applied filtering to retain only those with a lift greater than 5 and a confidence higher than 0.3. These thresholds ensure that the rules not only represent strong associations but are also reasonably reliable. I chose a minimum lift of 5 as after running different iterations of my minimum lift threshold, I found that 5 provided me with interesting rules but not too many that I could not interpret. I also chose a minimum confidence level of 0.3 for similar reasons. I then sorted the rules by lift as lift would help me discover the most interesting associations whereby I do not expect these words to appear together. I then exported these to a CSV file for better interpretability.

A screenshot of a computer

AI-generated content may be incorrect.

Likewise, I also did the same for confidence, whereby I sorted the rules by confidence as confidence can help me prioritize predictive strength where one word leads to another. This could potentially give me some insights into the words used that are often together, which could make for some interesting rules too. I also exported these to another CSV file for better interpretability.



Another reason why I decided to go with my minimum lift and confidence levels was because I used a boxplot to plot out the lift and confidence values for all generated association rules. Looking at both boxplots, I observed that the median is around 1 and 0.7 for lift and confidence values respectively. Choosing values outside of the interquartile ranges and from the outliers, would help me in extracting more strong, reliable rules while removing majority of the less useful ones.

A screen shot of a graph

AI-generated content may be incorrect.

A screen shot of a graph

AI-generated content may be incorrect.

4. Text Data Understanding