Master of Science
Artificial Intelligence and Big Data

Text Data Processing and Analysis in Recommender Systems

MOD002726 Postgraduate Major Project

Full Name: Atabek Murtazaev
SID: 2288841
Date:

Supervisor:
Assessor:

Acknowledgements

Thanks messages

Abstract

Recommendation systems have especially been given much attention since the use of the internet technologies that has facilitated the creation of the web, mobile, and the desktop applications. In the subsequent twenty years, the Internet has extended throughout the whole world and has therefore offered new avenues to support technologically-facilitated solutions. Here, for the same, I would like to present this project to develop a Django (Python) back-end and Vue for a review platform web-based text mining application. I used js (JavaScript) for the front-end.

The main goal is to create a site where people can give their opinions about some particular product or some media content, all the given opinions are stored in the database to form a set of such opinions. The web application is intended for more than 50 reviews, thus it will be sufficient to have a great number of materials for recommendation algorithms. The text data go through a number of preprocessing steps which are included in *tokenization*, removing the *stop-words*, and *lemmatization*. It is then used to build a content-based recommendation system that employs some techniques that include *TF-IDF vectorization* and *cosine similarity,* with an aim of providing recommendations depending on the users' reviews.

The scope of the project revolves around the adoption of NLP and ML to work towards improving the precision and the actual relevance of the suggestions made available to the users. To determine the efficiency of the developed system, it is tested against traditional parameters which include *precision*, *recall*, and *F1-score*. As seen from this study, it is possible to provide personalised content regarding the products through web applications by applying Natural Language Processing and Machine Learning. After all, integrating AI into review platforms is now a crucial feature among businesses of the twenty-first century.

Ethics Statement

Firstly, online research ethics training was completed by the date of starting this project. Then, Ethics application form (code: **ETH2324-9852 - Text Data Processing and Analysis in Recommender Systems)** was submitted and approved with a low risk of "Green" mark.

Table of Contents

1.0 Introduction

1.1 Overview

Recommender systems are one of the most important components of modern Internet services, helping users discover products, content, or services, often without their explicit request. The application of such systems is intended to process vast amounts of data to provide users with personalised recommendations, thereby increasing their engagement and satisfaction. Through these applications, users generate large volumes of textual data in the form of reviews, comments, and posts. This unstructured text data is valuable as it contains information that can be leveraged to refine the precision and relevance of recommendations offered.

However, extracting meaningful information from text data is not as easy a task as it seems like. Flaws in data processing are the cause of the tendency of underestimating natural language and presenting recommendations in a less comprehensive way. To overcome these challenges, advanced NLP[1] techniques shall be adopted coupled with Machine Learning. When these technologies are applied it is possible to better interpret the context and sentiment of the textual information to provide relevant results.

This work proposes a web application for improving CBRSs' effectiveness by incorporating NLP and ML approaches. Thus, despite the focus on the analysis of textual data as the primary means of interaction with the users of these technologies within the framework of this project, it can be noted that they are capable of offering targeted solutions for users in special topic areas..

1.2 Problem Background

The ongoing and escalating trends in technological enhancement have led to an increase in the amount of data produced mainly in the form of textual data, especially the textual reviews from the end-users of the products and services. That is why this great multitude of textual data is indeed valuable yet simple analysis of results can be rather difficult. One of the major drawbacks of mainstream approaches to data analysis is that there is no room for such things as context, different shades of meaning in human language or sentiment [1].

In order to overcome these challenges, this work focuses on the concept of a recommender system that provides the user with items that should be of interest according to the text data. Text data is considered abundant in information, however it is different from structures and hardly analysable with the use of ordinary mathematical methods. Such techniques as tokenization, keyword match, or frequency analysis, a recommendation list may largely differ from the user's interest; it is frequently less accurate [2].

The critical issue of this system is its scalability, as the volume of text data grows continually. In dynamic environments, where user preferences may change rather quickly, timeliness and accuracy of recommendations will be key. In fact, without the utilisation of developed

[1] NLP - Natural Language Processing

approaches to text data processing and analysis, recommender systems are at great risk of being ineffective with the growth of the text data volume.

To overcome these challenges, the present work uses NLP methods integrated with ML to improve the performance of the conventional content-based recommender systems. The given textual data need to be preprocessed and analysed in order to enhance recommendations' relevance as a result of the project, which would directly affect the users' satisfaction and engagement.

1.3 Research Aim and Objectives

Aim:

The goals of this project are as follows: The main goal of this project is to design and implement a solid web-based application for performing text mining tasks to make the CBRSs more efficient by incorporating modern-day NLP and machine learning approaches. It aims at solving issues associated with the analyses of text such as those pertaining to the provision of enhanced recommendations to the users.

Objectives:

Develop the Web Platform: For the back-end use Django, and on the front-end, use Vue. js for the front-end to implement where the users will be able to submit a review. This platform will enable the management and collection of textual content where Django will be responsible for data processing and data storage while Vue. js improving the front-end and the view given to the users.

Preprocess Text Data Using NLP Techniques: Among the text cleaning procedures are generated and are tokenization, stop-word removal and lemmatization. These steps are crucial for the textual data pre-processing and make the data ready for further process and recommendation.

Implement a Content-Based Recommender System: The recommender system will be then built with the help of employing the TF-IDF vectorization for text data In which the text data will be converted into vectors and the cosine similarity measure will be used for measuring the relation among the various items of text data. This is the objective that defines the central idea regarding the creation of the algorithm that will run within the system and will be used in generating the recommendations that are intended for the users, given the content that they have consumed.

Evaluate the Performance of the Recommender System: Their final objective is to assess the efficiency of the developed system and the conventional performance indicators such as precision, recall, and F1-measure. This way, it will be possible to see to what extent the recommender system is aligned with the goals and figure out where it can be changed.

2.0 Literature Review

2.1 Introduction to Recommender System

Recommendation systems are already present in the modern Web services to help to discover the content, product, or service relevant to the user. These systems are used in various fields including e-commerce, social networking sites and online movie or music streaming sites (Roy & Dutta, 2022). The basic aim of the recommender system is to guide the users to the most relevant content in the web amongst the massive amount of data available (Li et al., 2019).

Collaborative Filtering: Collaborative filtering (CF[2]) works based on the users' data analysis and, therefore, identification of other similar users. One of the known types of collaborative filtering, matrix factorization is one of the most popular and effective for predicting the relations between users and items (Jalali & Hosseini, 2022). While effective, collaborative filtering often struggles with the "cold start" problem when insufficient information exists about new users or items (Da'u, Salim & Osman, 2020).

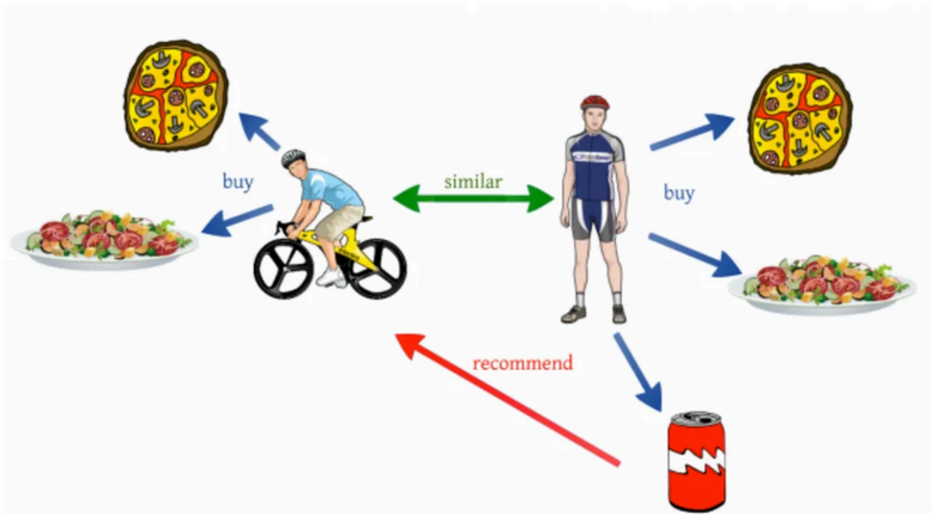


Fig 1. *Example of Collaborative Filtering. [Source]*

Content-based filtering: Content based filtering (CBF) involves features of the items that a consumers is concerned with (such as films, books or products) and returns items similar to the ones that a user has rated (Pang et al., 2016). For instance, if a user likes films with a certain actor, the system is going to recommend other films with the same actor (Lops et al., 2010). CBF is useful especially when there is less data about the users but it does not account for a variety of users' actions (Zhou et al., 2020).

[2] CF - Collaborative Filtering

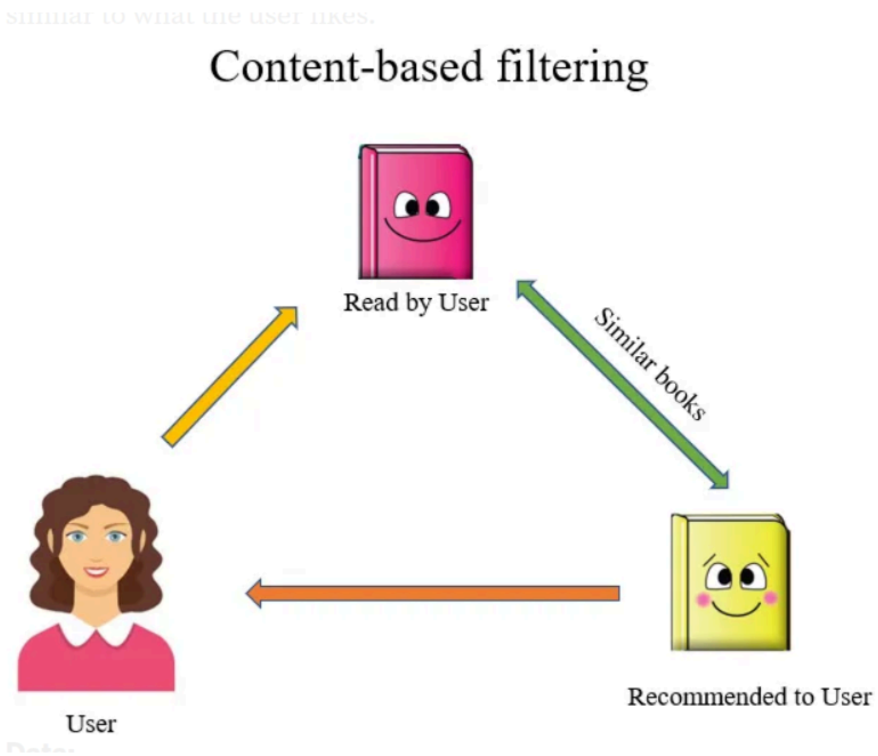


Fig 2. *Example of content-based filtering.* *[Source]*

Hybrid Recommendations: Combined recommendation approaches involve integration of several recommendation strategies to enhance the precision of the recommendations as well as eliminating the weaknesses of each technique. For example, combining CF and CBF can leverage the strengths of both approaches (Shambour, 2021). As depicted in Figure 3 below, the hybrid model uses both the CF base and content based recommenders before coming up with the final recommendations.

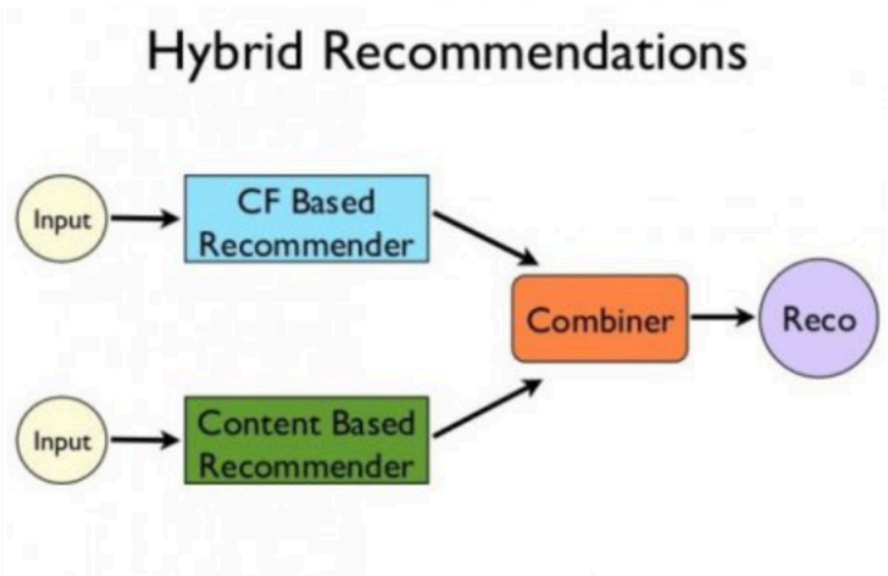


Fig 3. *Example of hybrid recommendation system.* *[Source]*

2.2 Text Mining

Text mining is one of the crucial steps of text data pre-processing and analysis required in a number of application fields involving text data , which is typical for many web-resources such as social networks, online shops, and blogs, comments, and articles. The fundamental purpose of text mining is to convert this text data into a format that is more orderly and thus more machine processable. Several key techniques are commonly used in text mining:Several key techniques are commonly used in text mining:

Tokenization

Tokenization refers to the division of text data into portions known as tokens which can be words, phrases or even sentences. This technique is the most basic in text preprocessing since it enables the system to examine each token as an independent variable. Tokenization facilitates other processes, such as word frequency counting and higher-level text analysis (Pang et al., 2016).

For instance, the statement “Recommender systems are essential in modern e-commerce” would be tokenized to [‘Recommender’, ‘systems’, ‘are’, ‘essential’, ‘in’, ‘modern’, ‘e-commerce’].

Stop-Word Removal

Stop-word elimination entails deleting terms that do not add much meaning to the text and are usually worthless for analysis, for instance ‘the’, ‘and’, ‘is’, and many others In other words its main purpose is to minimise the inputs of the data and thus enhance the smooth running of the text stream. This reduces the complexity of the text and improves processing efficiency (Pang et al., 2016).

When eliminating stop-words from the given array of words, the sentence will be completed with the following list of words: ["Recommender", "systems", "essential", "modern", "e-commerce"]

Lemmatization

Lemmatization is the transformation of the words into base form which is also referred as lemma. Unlike stemming where the method just chops off the word endings, lemmatization takes into account the internal structures of words and returns any valid word that is meaningful in the language. This technique is very helpful in making the text data more standard so that different forms of a word (for example ‘run’, ‘runs’, ‘ran’) are transformed into a single term ‘run’ (Dezfouli, Momtazi, & Dehghan, 2021).

For instance, the words "running", "ran", and "runs" would all be lemmatized to "run".

TF-IDF Vectorization

The Term Frequency-Inverse Document Frequency (TF-IDF) is one of the numerical formulas which determines a level of relevance of a word in terms of a given document compared to a set of documents, or a sample. TF-IDF is beneficial in exposing those nouns that concern some certain documents and at the same time, it reduces the impact of those words that can appear in numerous documents. It is crucial in transforming the text data into numerical vectors that can suit machine learning models (Dezfouli, Momtazi, & Dehghan, 2021)..

In a document discussing "machine learning," the term "learning" might have a high TF-IDF score if it is frequently mentioned in that document but less common across other documents in the corpus.

Word Embeddings
Word embeddings are dense vectors' representations of words that encode meaning in terms of relative positions of the word vectors. There is a method of Word2Vec and GloVe, which transforms the words into a word vector space where, for example, the synonyms are gathered near one another. These embeddings are especially useful in understanding the context and meaning of words which the basic BoW approach misses out on (Zhou et al., 2020)..

In a word embedding space, the words "king" and "queen" would be located close to each other, as would "man" and "woman," reflecting their semantic similarity.

2.3 NLP in Recommendation Systems

Recommendation systems can be enhanced with Natural Language Processing (NLP) and especially when the input data is natural and unstructured like in the case of books, movies, or products reviews, comments, tweets, etc. The use of NLP will enhance the working of the recommender system, the context, sentiment, and semantics of the text used will also improve the reliability of the recommendations made.

Term Frequency-Inverse Document Frequency (TF-IDF)
Popular among such techniques is the Term Frequency –Inverse Document Frequency (TF-IDF). TF-IDF is a quantitative measure computed for every word in every document with regards to the entire collection of documents. Indeed, this technique is useful in amplifying the importance of some words on a given document and at the same time diminishing the importance of some other words which are important on all documents (Dezfouli, Momtazi, & Dehghan, 2021).

Application in Recommender Systems:
While in the case of recommender systems, TF-IDF is applied to convert text data for instance, product description or reviews into feature vectors. As with the previous technologies, the essence of the process is based on putting textual data into numerical terms, with the main

end-result being the ability to define similarity between various items or users, so that the recommendation system in question would be able to provide recommendations based on the textual content of items (Roy & Dutta, 2022).

TF-IDF Formula

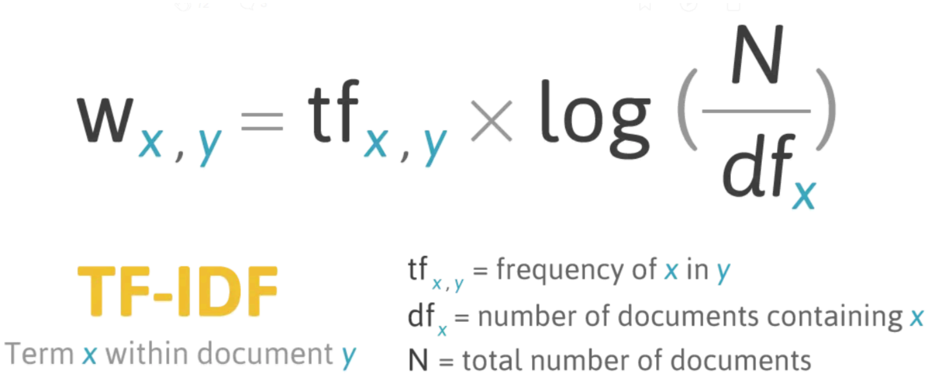


Fig 4. *TF-IDF formula illustrating how term importance is calculated in a document. Source*

Sentiment Analysis
The other important element of NLP employs sentiment analysis to identify the tone or emotion conveyed by a particular piece of text. This technique comes handy in situations where the tone of the message greatly affects the advice given by the system for instance in the case of reviews or posts made on social media (Tahmasebi, Ravanmehr, & Mohamadrezaei, 2021).

Application in Recommender Systems:
For instance, the sentiment of the reviews given the users may be used to enhance or privative the recommended products. Some products receive a positive sentiment while others have negative sentiments; the positive products are likely to be recommended most often as compared to the negative products.

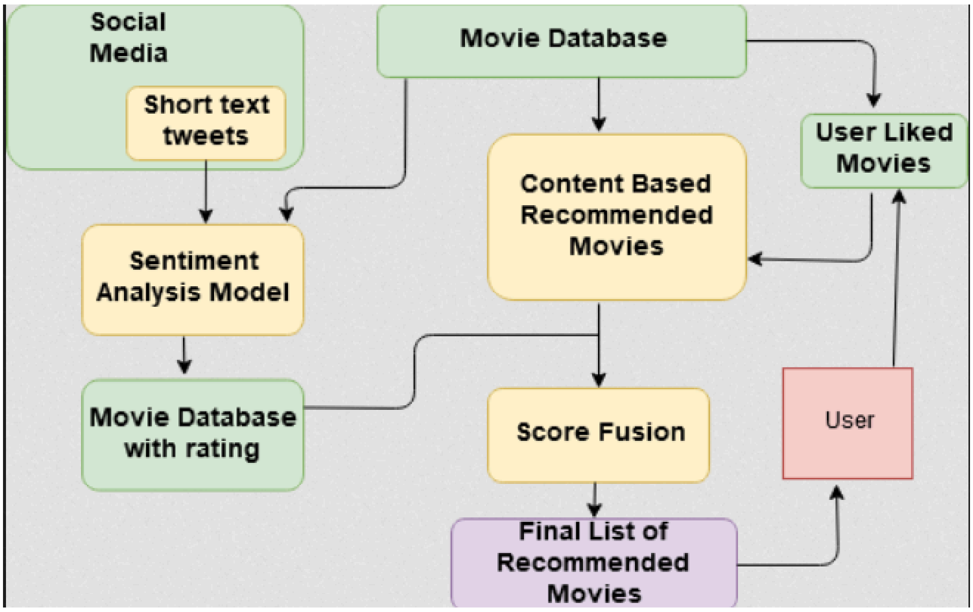


Fig 5. *Workflow of a content-based movie recommendation system incorporating sentiment analysis.* [Source]

Word Embeddings
For example, Word2Vec and GloVe are the word embeddings that learn dense word representations considering semantically related words. These embeddings enable the conceptual understanding and interpretation of words as a component of the recommendation system superior to the BoW[3] approach. It is thereby possible to understand how word embeddings can improve the recommender systems such that they produce more semantically and contextually relevant suggestions to the end-users (Zhou, Liang, Wang, & Yang, 2020).

Application in Recommender Systems:
Of all the methods of utilising word embeddings, it is especially suitable in content-based recommender systems as the system suggests the items based on the descriptions. Due to the fact that there is awareness of the semantic connections between words, the system can be able to recommend products that are related to the input even if the wording is slightly different.

[3] BoW - Bag of Words

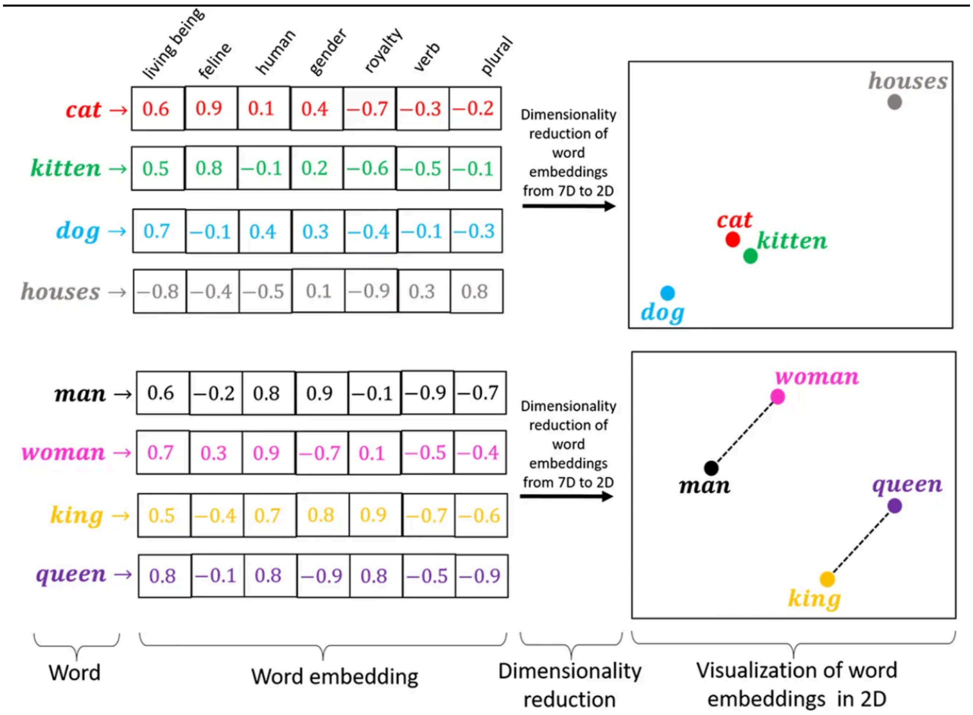


Fig 6. Word Embeddings in Recommendation system. [Source]

Named Entity Recognition (NER)

(NER[4]) is a method of NLP[5] which aims at identifying named entities (people, organizations, locations and so on) in a text. This is because NER can enrich recommender systems so that the latter can identify certain entities expressed by the users themselves for recommending purposes; consequently, recommendations that would be given can be more accurate (Zhou et al., 2020).

Application in Recommender Systems:
For example, a recommender system for news articles might use NER to identify and categorise articles based on the entities they mention, such as "Barack Obama" or "United Nations." This allows the system to recommend articles on specific topics or featuring certain entities that align with a user's interests.

[4] NER - Named Entity Recognition
[5] NLP - Natural Language Processing

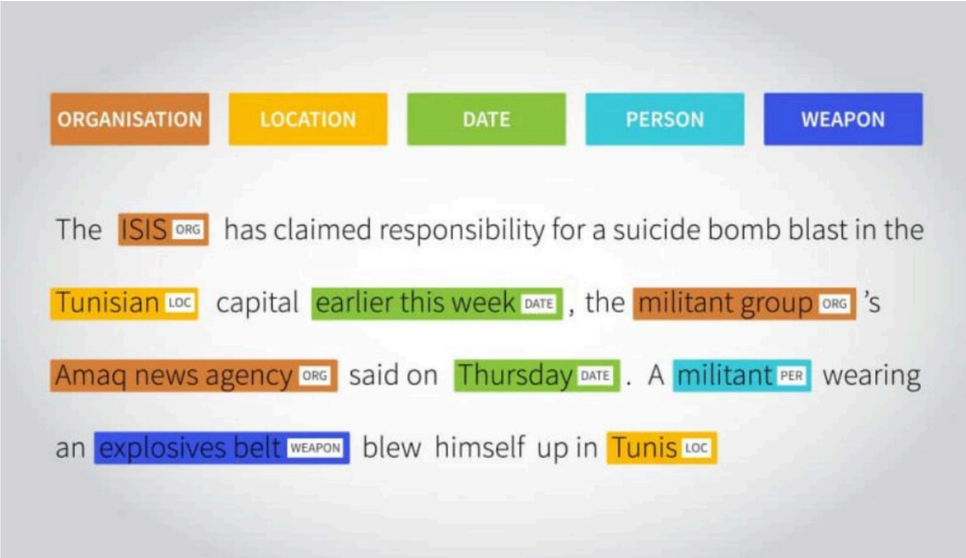


Fig 7. *Named Entity Recognition. [Source]*

Transformer Models

The transformer approach like BERT[6] are other enhancements for NLP to provide more elaborated computation of processing language. Even though transformers have been better developed for translation purposes, it helps them to capture the context of words at a sentential level, and it is extremely useful in text classification, sentiment analysis, or recommending systems (Devlin et al. , 2018).

Application in Recommender Systems:
When it comes to the application of transformers in recommender systems, one of the benefits that has been identified is that the transformer can help to enhance the comprehension of user intent in search queries, review content or comment. The transformers enable the capture of the context in which words are used and in that way offered a more accurate recommendation.

2.4 Machine Learning Algorithms for Recommendation Systems

Most present day recommender systems use ML algorithms since these are the only that offer the computational capability of analysing large amounts of data so as to produce customised recommendations. These algorithms are of different types and ranges from simple linear regression models to deep learning models. Here are some of the vital ML algorithms employed in recommender systems.

Overview of Collaborative Filtering

[6] BERT - Bidirectional Encoder Representations from Transformers

Two such approaches are believed to be the most widely adopted in the field of recommender systems: Collaborative Filtering and Hybrid Models. As stated in sections 2.1 to and known as “Hybrid Recommendations,” these methods are effective since they make use of user interaction data and they also make use of several recommendation techniques to increase reliability (Roy & Dutta, 2022; Shambour, 2021).

Matrix Factorization

There are many more computational methods for modelling user-item interaction data, including (SVD[7]). These factors can uncover the hidden structures of the data, for example, users’ preferences or items’ characteristics (Li et al., 2019).

Application in Recommender Systems:

Matrix factorization is quite common in the context of collaborative filtering where it is utilised to predict missing values in the matrix of user-item interactions by decomposing matrix in lower-dimensional matrices.

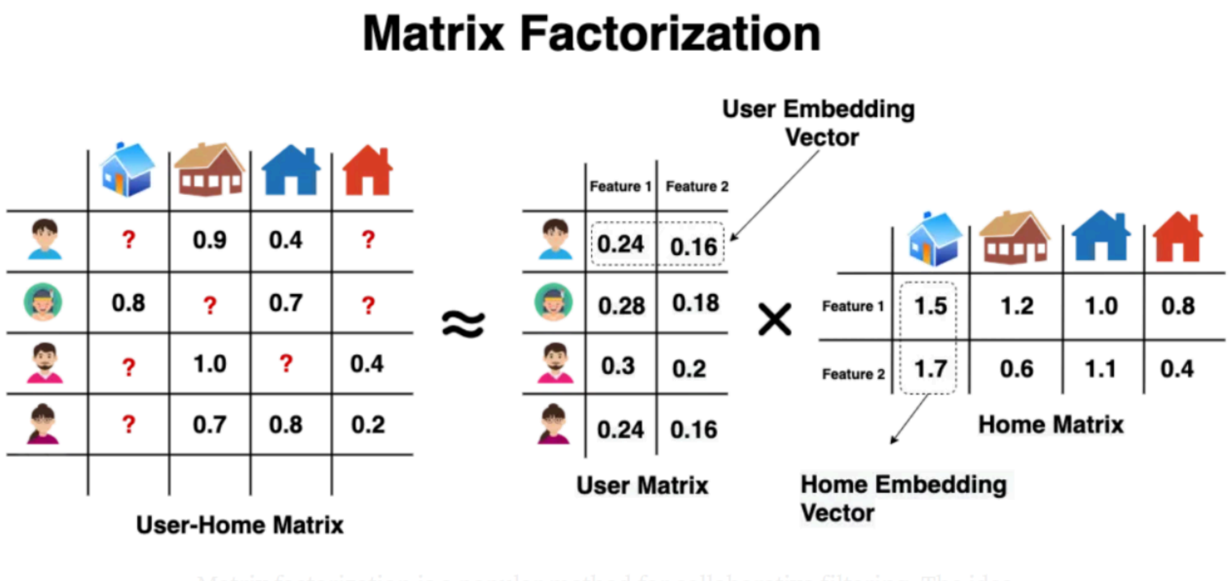


Fig 8. *Implementing Matrix Factorization.* [Source]

Neural Networks

Artificial neural networks, especially the deep learning methods, are the most used techniques in recent years especially in the recommender systems domain. Among the architectures, CNNs[8] and RNNs[9] are most popular (He et al. , 2017).

Application in Recommender Systems:

[7] SVD - Singular Value Decomposition
[8] CNN - Convolutional Neural Network
[9] RNN - Recurrent Neural Network

Structured and unstructured data can be also used to train deep learning models which can be also used in various recommendation tasks such as image-based recommendation system and sequential recommendation system.

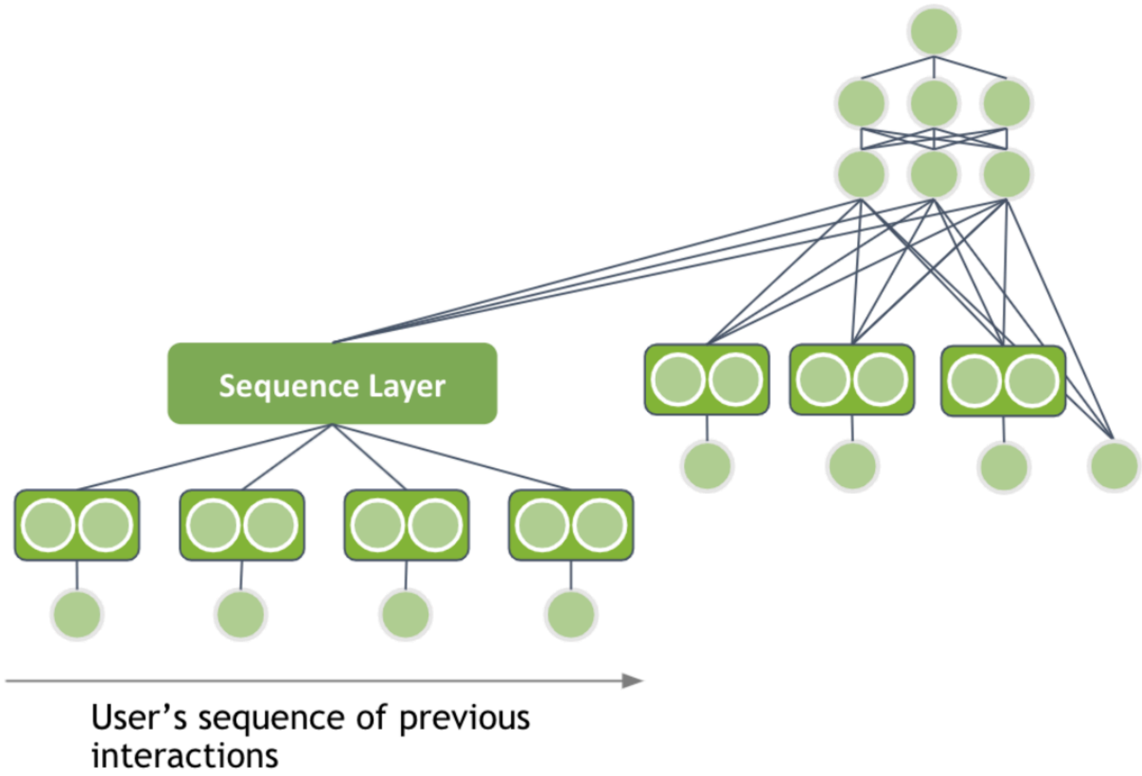


Fig 9. *Using Neural Networks for Recommender Systems.* [*Source*]

2.5 Content-Based Filtering Approaches

CBF[10] which is also known as, trait-based filtering is among the traditional methods of recommending used in recommender systems and involves the identification of qualities in items that can be used to recommend other items with similar qualities to users with preference towards the particular item. This is in contrast to collaborative filtering that makes use of the activities of the other users, content-based filtering recommends items by the attributes of the items (Pang et al., 2016).

Feature Extraction

In the context of the commonly used review platforms, feature extraction is the process of determining attributes from the reviews which are useful for suggestions, they may include qualities of products or even sentiments of users (Zhou et al., 2020).

[10] CBF - Content-Based Filtering

Example:
A book recommendation system might have to scan through the genre and authors as well as keywords which might be used to recommend other books similar to the book in question by the user.

Similarity Measures
Content-based filtering employs all manner of similarity match-making to identify those aspects of the items that are most similar to those a user has appreciated in the past.

Cosine Similarity is one of them that involves calculation of similarity in terms of the cosine of the angle made by the two items' feature vectors.

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

***Fig 10**. Cosine Similarity. Source*

Example:
If the reader has earlier enjoyed a science fiction novel, the system is likely to use cosine similarity to bring other similar science fictions with those keywords.

User Profile Creation
New: The system forms a user profile out of the items with which the user has interacted, adopting the features extracted as an index to the user's preference. The extracted features from these items serve as an index of the user's preferences. As the user interacts with more items, the profile is continuously updated to reflect any changes in preferences (Roy & Dutta, 2022).

Example:

A music streaming service may build a user model where the data acquired by the service regarding user's listening habits may be used to suggest new songs in the genres and from the artists that the user tends to listen to, and/or songs with specific tempos.

Advantages and Limitations

Advantages:
The second approach in the strategy of filtering is content-based, where users get recommendations that reflect what they like. It can recommend items which are specialized or less rated by other members, which makes it personalized in a way (Shambour, 2021).

Limitations:
The system could suggest that they are too closely related, and this can lead to lack of variation in the suggested items. It presupposes precise definition of the item features and quality of the recommendation inherently depends on how precise the feature extraction process is.

2.6 Evaluation Metrics in Recommender Systems

This is quite important since it facilitates assessment of the impact of the recommender systems. They assist in expressing how effective a system is, in relation to making relevant, correct, and useful suggestions to users. Depending on the kind of recommendation that is being given, then different types of measures are taken into account, it could be accuracy, or diversity or even user satisfaction. Here are some of the measures most often used in recommender systems:

Precision and Recall

Precision and Recall are two fundamental metrics often used together to evaluate the accuracy of recommender systems:Precision and Recall are two fundamental metrics often used together to evaluate the accuracy of recommender systems:

Precision is defined as the ratio of the number of recommended items to the number of relevant items to a user. It answers the question of how many out of all the items recommended were actually important (Powers, 2020).

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Fig 11. *Precision*. *[Source]*

Recall measures the actual number of relevant items that were recommended to the user divided by the total number of relevant items. It solves the problem of identifying specific items and then asking, “Out of these, how many were most recommended?” (Gunawardana & Shani, 2009).

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Fig 12. *Recall.* [Source]

These two are usually used in a compound way in the form of a single figure called the **F1-Score** which gives a combined value of both the precision and the recall measurements (Powers, 2020).

F1-Score

The F1-Score is the harmonic mean of the precision and recall and hence, it is a better measure than each of the two. As the name suggests, this metric proves very useful when used in instances where there is a need to determine the trade-off between the precision and the recall.

$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Fig 12. *F1-Score Definition.* [Source]

MAE and RMSE

Mean Absolute Error (MAE[11]) and Root Mean Squared Error (RMSE[12]) are metrics used to measure the accuracy of predicted ratings in recommender systems:Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE) are metrics used to measure the accuracy of predicted ratings in recommender systems:

MAE computes the mean absolute magnitude of errors with no regard for direction in a given set of predictions (Gilbert, 2023).

[11] MAE - Mean Absolute Error
[12] RMSE - Root Mean Squared Error

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n} |Y_i - \hat{Y}_i|$$

***Fig 13**. MAE formula.* [*Source*]

RMSE not only measures the difference in magnitude, but also the amount that the forecasted rating deviates on average from the actual rating (Gunawardana & Shani, 2009).

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} \|y(i) - \hat{y}(i)\|^2}{N}},$$

***Fig 14**. RMSE formula.* [*Source*]

MAE and RMSE are the two most common metrics for measuring the predictive accuracy of a recommender system with the lower the better.

AREA Under the ROC Curve (AUC-ROC)

The Area Under the ROC Curve (AUC-ROC) is another measure of accuracy used on binary classification systems and also in recommender systems. ROC stands for Receiving Operating Characteristic curve and it is a graphical representation where the true positive rate is pitted against the false positive rate at different threshold levels. The AUC, obtained by the area under this curve offers one measure that gives an overall performance of the system in terms of relevancy and irrelevancy of the items (Powers, 2020).

An AUC of 1 means the system is a perfect recommender and an AUC of 0 exemplifies a poor recommender system. 5 is interpreted as the calamitous state in which the system has no discrimination power and is in fact as poor as selecting samples at random.

Diversity and Novelty

In addition to accuracy-based metrics, Diversity and Novelty are also important for evaluating recommender systems:

Diversity measures how varied the recommended items are. A system with high diversity will recommend items from a wide range of categories, increasing the chances of user discovery (Gunawardana & Shani, 2009).
Novelty is calculated as the difference between the number of times a particular item has been recommended and the number of times the user has seen it. High novelty is useful in the sense that it can increase user satisfaction since it exposes the users to products they may not have selected on their own (Roy & Dutta, 2022).

3.0 Methodology

This section describes the practical implementation process of using math/algorithms formulated in the Literature Review of the proposed Recommender System, along with a course of the action taken in the process. The methodology is structured into four key areas: *Data Collection* and *Preprocessing*, *Model Architecture*, *Training Evaluation* and *Evaluation Metrics*.

3.1 Data Collection and Preprocessing

The collection and the preprocessing of the data determine the performance of the recommender system. Below are some of the techniques that need to be followed for making sure that the data is fit for making the model to learn:

3.1.1 Tokenization

Application: Referring to section 2.1, in text data pre-processing the tokenization technique was employed in the process of converting text data into smaller components called tokens. This step was done using the ***NLTK***[13], ***spaCy***[14] so as to enhance the capability of the model in handling textual data.
Sample code:

```
# !pip install spacy
import spacy

nlp = spacy.load('en_core_web_sm')
doc = nlp("Recommender systems are essential in modern e-commerce")
tokens = [token.text for token in doc]

# Result
# ['Recommender', 'systems', 'are', 'essential', 'in', 'modern', 'e', '-',
'commerce']
```

[13] NLTK - Natural Language Toolkit
[14] spaCy - Industrial-StrengthNatural Language Processing

3.1.2 Stop-Word Removal

Application: As discussed in section 2.2 above, the process of data preprocessing involved the removal of stop-words from the text data to minimize noise. This process was performed with a *stop list*, which allowed to shift the model's attention towards more significant words in the dataset.
Sample code:
```
# !pip install nltk
import nltk
nltk.download('stopwords')

stop_words = set(stopwords.words('english'))
filtered_tokens = [word for word in tokens if word.lower() not in stop_words]

# Result
# ['Recommender', 'systems', 'essential', 'modern', 'e', '-', 'commerce']
```

3.1.3 Lemmatization

Application: Continuing from the principles of section 2.2. Lemmatization was used with the aim of reducing the words to their stem or root form. This step was important in order to decrease the dimensionality of the feature space of the text data and it was conducted by using the libraries ***SpaCy*** and Natural Language Toolkit (***NLTK***).
Sample Code:
```
lemmatized_tokens = [token.lemma_ for token in doc]

# Result
# ['recommender', 'system', 'be', 'essential', 'in', 'modern', 'e', '-',
'commerce']
```

3.2 Model Architecture

Here the model architecture is also important in defining the success of the recommendation system. Such approaches are included the following methods described in section 2.3 and 2.4 were used in the establishment of the system and their inclusion is detailed below:

3.2.1 TF-IDF Vectorization

Application: As indicated in section 2.3, Execution of text features into numerical form was done using the TF-IDF technique for feature vectorization. This method was performed using ***Scikit-learn*,**[15] so the model was capable of appropriate measure of word weights in documents for calculating similarity.
Sample code:

```
from sklearn.feature_extraction.text import TfidfVectorizer

corpus = [
   "Recommender systems are essential for modern e-commerce.",
   "They help in personalizing the shopping experience.",
   "Machine learning techniques are used in recommender systems.",
   "E-commerce websites heavily rely on recommender systems."
]

# Initialize the vectorizer
vectorizer = TfidfVectorizer()

# Transform the corpus into a TF-IDF matrix
tfidf_matrix = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names_out())

# Result
# ['are' 'commerce' 'essential' 'experience' 'for' 'heavily' 'help' 'in'
#  'learning' 'machine' 'modern' 'on' 'personalizing' 'recommender' 'rely'
#  'shopping' 'systems' 'techniques' 'the' 'they' 'used' 'websites']
```

3.2.2 Cosine Similarity

Application: The cosine similarity metric that has been described in section 2.4 was used to compute the level of similarity of the items vectors that are derived from the TF-IDF process. This similarity measure was important for the actual disambiguation and recommendation of similar items within the context of the given user's prior interaction.
Sample code:

```
from sklearn.metrics.pairwise import cosine_similarity
similarity_matrix = cosine_similarity(tfidf_matrix)
```

[15] Scikit-learn- machine learning in Python

```
print(similarity_matrix)

# Result
# [[1.         0.         0.25949256 0.27392617]
#  [0.         1.         0.09814311 0.        ]
#  [0.25949256 0.09814311 1.         0.14198451]
#  [0.27392617 0.         0.14198451 1.        ]]
```

3.3 Training and Evaluation

The second step was to train the model by using the selected algorithms to the training dataset so that the model gets embedded with certain understanding of how the patterns are likely to occur to reduce its ability to predict the test cases accurately.

3.3.1 Model Training

Application: Logistic regression model was employed trained with scikit-learn and the training process was done in the same way as explained in section 2.4. The training process involved the process of tuning the parameters of a model in an effort to have the reduced prediction error in order to enhance the performance of the recommendation model.
Hyperparameter Tuning: As for Hyperparameters, tuning was done using *GridSearchCV* to bring out the best to maximize on recommendations accuracy out of the available results.
Sample code:

```
from sklearn.model_selection import GridSearchCV
parameters = {'tfidf__max_df': [0.8, 0.9], 'tfidf__min_df': [0.01, 0.05]}
grid_search = GridSearchCV(pipeline, parameters, cv=5)
grid_search.fit(X_train, y_train)
```

3.3.2 Validation

Application: In section 2.4 we discussed briefly about Cross-Validation and its training process. For validation to the model's performance we used *K-Fold Cross-Validation* method to ensure generalization of new to unseen data.

3.4 Evaluation Metrics

Evaluation metrics, which have already been discussed in section 2.6 were used with a view of evaluating the performance of the model. The following measures were used to give a holistic analysis of the capacity of the system to produce correct and pertinent recommendations.

3.4.1 Precision, Recall and F1-Score

Application: Performance comparison of the recommendations was done using *precision*, *recall*, and *F1-Score*. These metrics were important in deciding the level of relevance compared to the level of completeness of the recommendation of the system.
Sample code:

```
from sklearn.metrics import precision_score, recall_score, f1_score
precision = precision_score(y_true, y_pred)
recall = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)
```

3.4.2 RMSE and MAE

Application: For the assessment of the traced predicted ratings, two kinds of errors were adopted, the root mean square error (*RMSE*) and the mean absolute error (*MAE*). Low first order averages in these dimensions suggested a good degree of correspondence between the predicted and actual ratings establishing the predictive dimension of the model.
Sample code:

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
rmse = mean_squared_error(y_true, y_pred, squared=False)
mae = mean_absolute_error(y_true, y_pred)
```

3.4.3 AUC-ROC

Application: The *AUC-ROC* metric was calculated to evaluate the system's ability to distinguish between relevant and irrelevant items. A higher *AUC-ROC* value demonstrated the model's effectiveness in making precise recommendations.
Sample code:

```
from sklearn.metrics import roc_auc_score
auc = roc_auc_score(y_true, y_pred_proba)
```

3.4.3 Diversity and Novelty

Application: To replace the metric, the following definitions of diversity and novelty were used Among the expressions for the metrics, which were used to check the accuracy and the variety of the recommendations, the *diversity* and the *novelty* were included. Such metrics made certain that through the system, users were presented with a very diverse range of products and exposed to items they would not normally come across.

4.0 Implementation

4.1 Data pre-processing

This section explains the pre-processing steps executed on raw textual data before further analysis. The following are the steps involved in *cleaning*, *tokenizing*, and *normalisation* into its canonical form:

```
import pandas as pd
import spacy
import nltk
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
from prettytable import PrettyTable
```

pandas for data manipulation
spaCy for tokenization and lemmatization
nltk for stop-word removal
TfidfVectorizer for vectorizing text data
PrettyTable for displaying tabular data in a readable format

So first we will import our “*.csv” files from the storage

```
# Upload file
users_df =
pd.read_csv('/content/drive/MyDrive/recommendation_system/data/users.csv')

# Read file
pretty_table = dataframe_to_prettytable(users_df.head())
print(pretty_table)
```

```
# Upload file
products_df =
pd.read_csv('/content/drive/MyDrive/recommendation_system/data/products.csv')

# Read file
pretty_table = dataframe_to_prettytable(products_df.head())
print(pretty_table)

# Upload file
reviews_df =
pd.read_csv('/content/drive/MyDrive/recommendation_system/data/reviews.csv')

# Read file
pretty_table = dataframe_to_prettytable(reviews_df.head())
print(pretty_table)
```

After that we will combine or merge together.

```
# Merge reviews with products
reviews_with_products = reviews_df.merge(products_df, left_on='product_id',
right_on='pk', suffixes=('_review', '_product'))

# Ensure type consistency for merging
reviews_with_products['author_id'] =
reviews_with_products['author_id'].astype(int)
users_df['pk'] = users_df['pk'].astype(int)

# Merge with users
reviews_with_details = reviews_with_products.merge(users_df, left_on='author_id',
right_on='pk', suffixes=('_product', '_user'))
```

So, we have 50 users, 50 products and 50 reviews.
Sample tables:

Tokenization: The division of a text or sequence into smaller units, such as words, phrases, or symbols.

```
# Load spaCy tokenizer

nlp = spacy.load('en_core_web_sm')

# Apply tokenization to review comments

reviews_with_details['tokens'] =
reviews_with_details['review_comment'].apply(lambda x: [token.text for
token in nlp(x)])
```

```python
# Display the first few rows with tokens

pretty_table =
dataframe_to_prettytable(reviews_with_details[['review_comment',
'tokens']].head())

print(pretty_table)
```

Stop-word elimination: Eliminating common words, such as "the" and "is," which do not contribute to the meaning of the analysis.

```python
# Define stop words

nltk.download('stopwords')

stop_words = set(stopwords.words('english'))

# Apply stop-word removal to tokens

def remove_stop_words(tokens):

   return [word for word in tokens if word.lower() not in stop_words]



reviews_with_details['filtered_tokens'] =
reviews_with_details['tokens'].apply(remove_stop_words)

# Display the first few rows with filtered tokens

pretty_table =
dataframe_to_prettytable(reviews_with_details[['review_comment',
'tokens', 'filtered_tokens']].head())

print(pretty_table)
```

Fig 17*. Stop-word removal and its realisation*

Lemmatization: Reduction of words to their base form to standardise text (e.g., "running" becomes "run").

```python
# Function to perform lemmatization
```

```
def lemmatize_tokens(tokens):

   doc = nlp(" ".join(tokens))  # Convert list of tokens back to a string

   return [token.lemma_ for token in doc if token.lemma_ not in
stop_words and not token.is_punct]



# Apply lemmatization

reviews_with_details['filtered_tokens'] =
reviews_with_details['tokens'].apply(lambda x: lemmatize_tokens(x))

# Display the table

pretty_table =
dataframe_to_prettytable(reviews_with_details[['review_comment',
'tokens', 'filtered_tokens']].head())

print(pretty_table)
```

Fig 18*. Lemmatization and its realisation*

Text Vectorization: Converting text data that has been pre-processed into numerical vectors for the next processing of machine learning models through the TF-IDF technique.

```
# Text vectorization using TF-IDF

vectorizer = TfidfVectorizer()

tfidf_matrix =
vectorizer.fit_transform(reviews_with_details['filtered_text'])
```

Cosine Similarity Calculation: Cosine similarity would be a measure between TF-IDF vectors—say, their usefulness when applying to different texts, like product reviews.

```
# Calculate cosine similarity between the TF-IDF vectors

similarity_matrix = cosine_similarity(tfidf_matrix)
```

Fig 19*. TF-IDF and Cosine Similarity*

4.2 Project Structure

This section is going to describe the whole structure of the project: how back-end and front-end components (Django and Vue.js) are organised to work seamlessly with each other. The back end, implemented with Django, acts as the main logic and handler for the database. It will expose a well-defined API to interact with the front end, which in turn is designed using Vue.js. Vue.js manages the user interface and user experience while interacting with API endpoints exposed by Django. The interplay between these parts ensures data flow is seamless and effortless from server to client and vice versa, which makes a dynamic and responsive web application.

core:
Backend(Django)
config: This directory holds all the aspects of the Django project such as settings and other configurations.

```
❯ tree config -I '__pycache__'
config
├── asgi.py
├── __init__.py
├── settings.py
├── urls.py
└── wsgi.py

1 directory, 5 files
```

settings.py: Responsible for database settings, installed applications, setting up of REST framework and the JWT authentication.
urls.py: Defines URL routing for the entire project, including API routes.
wsgi.py and **asgi.py**: Production ready configurations for web server and asynchronous server gateway interfaces.

core: This directory holds the core Django apps developed for the system, each with its own substructure:
core/user: Manages user authentication and profile functionalities.

```
❯ tree core/user -I 'migrations|__pycache__'
core/user
├── api
│   ├── serializers.py
│   └── viewsets.py
├── apps.py
├── __init__.py
├── models.py
└── tests
    └── test_user.py

3 directories, 6 files
```

models.py: Custom user model definition.
api/serializers.py: Handles serialisation for user data.
api/viewsets.py: Viewsets to manage user API endpoints (login, register, logout, etc.).

core/product: Manages product-related data, including product creation and retrieval.

```
core/product
├── api
│   ├── serializers.py
│   └── viewsets.py
├── apps.py
├── __init__.py
├── models.py
└── tests
    └── test_product.py

3 directories, 6 files
```

api/models.py: Defines the product model.
api/serializers.py: Serializes product data for API communication.
api/viewsets.py: Provides viewsets for handling product API operations (CRUD).

core/reviews: Handles product reviews.

```
❯ tree core/reviews -I 'migrations|__pycache__'
core/reviews
├── api
│   ├── serializers.py
│   └── viewsets.py
├── apps.py
├── __init__.py
├── models.py
└── tests

3 directories, 5 files
```

api/models.py: Defines the review model.
api/serializers.py: Serializes review data for API communication.
api/views.py: Contains viewsets for handling review-related API requests.

core/ml_models:

```
core/ml_models
├── apps.py
├── __init__.py
├── ml
│   └── recommendation_model.py
├── models.py
├── saved_models
│   ├── logistic_regression_model.pkl
│   └── tfidf_vectorizer.pkl
└── views.py

3 directories, 7 files
```

models.py: Contains code for training, testing, and deploying the recommendation models (including TF-IDF vectorization and cosine similarity).

tests: Has unit tests for backend, therefore confirms the API working properly as well as models, and the logic.
migrations: Stores migration files created by Django's ORM for dealing with Structural Changes in databases.

Frontend (Vue.js)
The frontend is developed with Vue.js and is structured as follows:
assets/: Contains static assets such as images and CSS files.

```
❯ tree src/assets
src/assets
├── logo.png
└── styles
    ├── home.css
    ├── login.css
    ├── navbar.css
    └── register.css

2 directories, 5 files
```

components/: Reusable components for building the user interface.

```
❯ tree src/components
src/components
└── MainNavbar.vue

1 directory, 1 file
```

MainNavbar.vue: Navbar component displayed at the top of the app.
views/:

```
❯ tree src/views
src/views
├── HomeView.vue
├── ProductCreate.vue
├── ProductEdit.vue
├── ProductList.vue
├── UserLogin.vue
└── UserRegister.vue

1 directory, 6 files
```

HomeView.vue: Main page view that handles product recommendations.
ProductCreate.vue: View responsible for user login.
ProductEdit.vue: View for creating a new product.
ProductList.vue: View for listing products.
UserLogin.vue: View for Login.
UserRegister.vue: View for Register.

router:

```
❯ tree src/router
src/router
└── index.js

1 directory, 1 file
```

index.js: Configures routes like login, register, products, reviews and manages navigation.
main.js: Entry point for the Vue.js application that mounts the app and initialises routes.
tests/: Contains unit tests for Vue.js components using Jest or Vue Test Utils to ensure component functionality.
.env: Holds environment variables for configuration, such as API[16] base URL[17]s.

4.3 Backend Development

The backend development concerns the server-side and contains code for managing the database, API connection and the recommendation system. The backend is created with Django to handle the fundamental application and business logic accompanied by Django Rest Framework (DRF) for API construction. Here's a breakdown of how the backend was structured and implemented:Here's a breakdown of how the backend was structured and implemented:

User Management

Custom User Model: Another aspect that has been incorporated into this kind of application is usage of UUID for the primary key, which gives security and uniqueness to the user's account model. The fields include, **username**, **e-mail**, **password**, and other fields for controlling user status(**is_active**, **is_staff**, **is_superuser**).

Below this is a custom user model **core/user/models.py**

```
class User(AbstractModel, AbstractBaseUser, PermissionsMixin):
   """
   Custom User model with UUID as the primary key and additional fields for
authentication and authorization.

   Attributes:
       public_id (UUIDField): Unique identifier for the user.
       username (CharField): Username for the user.
       first_name (CharField): User's first name.
       last_name (CharField): User's last name.
       email (EmailField): User's email address.
       is_active (BooleanField): Indicates whether the user is active.
       is_staff (BooleanField): Indicates whether the user has staff privileges.
```

16 API - Application User Interface
17 URL - Uniform Resource Locator

```
        is_superuser (BooleanField): Indicates whether the user has superuser
privileges.
        created (DateTimeField): Timestamp when the user was created.
        updated (DateTimeField): Timestamp when the user was last updated.
    """
    public_id = models.UUIDField(
        db_index=True,
        unique=True,
        default=uuid.uuid4,
        editable=False
    )
    username = models.CharField(
        db_index=True,
        max_length=255,
        unique=True
    )
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    email = models.EmailField(
        db_index=True,
        unique=True
    )
    is_active = models.BooleanField(default=True)
    is_staff = models.BooleanField(default=False)
    is_superuser = models.BooleanField(default=False)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)

    USERNAME_FIELD = 'email'
    REQUIRED_FIELDS = ['username']

    objects = UserManager()

    def __str__(self) -> str:
        """
        Returns a string representation of the user instance.

        Returns:
            str: The username of the user.
        """
        return f'{self.username}'

    @property
    def name(self) -> str:
        """
        Returns the full name of the user.

        Returns:
            str: The full name of the user in the format 'first_name - last_name'.
        """
        return f'{self.first_name} - {self.last_name}'
```
Full file is [here]()

After that we will serialize our model to the JSON to implement API
Below our **core/user/api/serializers.py**

```
from core.abstract.serializers import AbstractSerializer
from core.user.models import User


class UserSerializer(AbstractSerializer):
    """
    Serializer for the User model. Inherits from AbstractSerializer to include
    common fields like id, created, and updated.
    """

    class Meta:
        model = User
        fields = [
            'id', 'username', 'first_name', 'last_name', 'email', 'is_active',
'created', 'updated'
        ]
        read_only_fields = ['is_active']  # is_active is read-only to prevent changes
via this serializer
```

After that we will implement methods GET and POST to interact with users

core/user/api/viewsets.py

```
from rest_framework.permissions import IsAuthenticated
from core.user.models import User
from core.user.api.serializers import UserSerializer
from core.abstract.viewsets import AbstractViewSet


class UserViewSet(AbstractViewSet):
    """
    ViewSet for handling operations related to the User model.
    Allows authenticated users to view and partially update user information.
    """
    http_method_names = ('get', 'patch')  # Allow GET and PATCH requests
    permission_classes = (IsAuthenticated,)  # Only authenticated users can access
    serializer_class = UserSerializer

    def get_queryset(self):
        """
        Returns a queryset of users. For authenticated users, returns all users.
        For unauthenticated requests, excludes superusers.
        """
        if self.request.user.is_authenticated:
            return User.objects.all()  # Return all users if authenticated
        return User.objects.exclude(is_superuser=True)  # Exclude superusers if not
authenticated

    def get_object(self):
        """
        Retrieves a user object by its public_id from the URL.
```

```
        Checks permissions to ensure the user can access the object.
        """
        obj = User.objects.get_object_by_public_id(self.kwargs['pk'])
        self.check_object_permissions(self.request, obj)   # Ensure permissions are
checked
        return obj
```

In order to authenticate (login, logout, register) and also using for Client side server or Mobile app (in future) we will JWT[18] Authentication

JWT Authentication: To enhance the security of the user's authentication process, token based authentication using JSON Web Tokens (JWT) were implemented. Users can sign up; log into the application and get JWT tokens for the subsequent use in API calls.

Below the code: **core/auth/api/serializers/login.py**

```
from typing import Any, Dict
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer
from rest_framework_simplejwt.settings import api_settings
from django.contrib.auth.models import update_last_login
from core.user.api.serializers import UserSerializer


class LoginSerializer(TokenObtainPairSerializer):
    """
        Custom serializer to handle token authentication with additional user details.
    """

    def validate(self, attrs: Dict[str, Any]) -> Dict[str, str]:
        """
            Validate the provided credentials and return the JWT tokens along with user
data.

            Args:
                attrs (Dict[str, Any]): Contains 'email' and 'password'.

            Returns:
                Dict[str, str]: Contains 'refresh', 'access', and 'user' data.

            Raises:
                serializers.ValidationError: If authentication fails.
        """
        data = super().validate(attrs)

        refresh = self.get_token(self.user)

        data['user'] = UserSerializer(self.user).data
        data["refresh"] = str(refresh)
        data["access"] = str(refresh.access_token)

        if api_settings.UPDATE_LAST_LOGIN:
```

[18] JWT - JSON Web Token

```
            update_last_login(None, self.user)

        return data
```

And its response in the viewsets

core/auth/api/viewsets/login.py

```
from rest_framework.response import Response
from rest_framework.viewsets import ViewSet
from rest_framework.permissions import AllowAny
from rest_framework import status
from rest_framework_simplejwt.exceptions import TokenError, InvalidToken
from core.auth.api.serializers import LoginSerializer


class LoginViewSet(ViewSet):
   """
       ViewSet for handling user login requests.

       This ViewSet provides a `create` method for user login and token generation.
       It uses `LoginSerializer` to validate credentials and
`CustomTokenObtainPairSerializer`
       to generate and return access and refresh tokens.
   """
   serializer_class = LoginSerializer
   permission_classes = (AllowAny, )
   http_method_names = ['post']

   def create(self, request, *args, **kwargs):
       """
           Handles POST requests for user login.

           This method accepts `email` and `password` in the request data,
           validates the credentials using the `LoginSerializer`, and
           returns the access and refresh tokens if the credentials are valid.

           Args:
               request (Request): The incoming request containing `email` and
`password`.

           Returns:
               Response: A Response object containing the access and refresh tokens,
or
                         an error message if the credentials are invalid.
       """
       serializer = self.serializer_class(data=request.data)

       try:
           serializer.is_valid(raise_exception=True)
       except TokenError as e:
           raise InvalidToken(e.args[0])
```

```
        return Response(serializer.validated_data, status=status.HTTP_200_OK)
```

JWT settings in **config/settings.py**

```
'DEFAULT_AUTHENTICATION_CLASSES': (
   'rest_framework_simplejwt.authentication.JWTAuthentication',
),
```

Full link is [here]

Product Management

Product Model: The product model refers to items which customers can rate. Every product is linked to the respective author who is one of the registered users; the information about the product includes name, description, price, and editing. The model conforms relations with users and the reviews as many-to-relationship with any product.

core/product/models.py

```
from django.db import models
from core.abstract.models import AbstractModel, AbstractManager


class ProductManager(AbstractManager):
   """
   Custom manager for the Product model.

   This manager provides additional methods for querying Product instances if needed.
   """
   pass


class Product(AbstractModel):
   """
   Represents a product in the system.

   Attributes:
       author (ForeignKey): Reference to the user who created the product.
       name (CharField): The name of the product.
       description (TextField): A detailed description of the product.
       price (DecimalField): The price of the product.
       edited (BooleanField): Flag indicating if the product has been edited.
   """
   author = models.ForeignKey(
       'core_user.User',
       on_delete=models.CASCADE,
       related_name='products',
       help_text="The user who created the product."
```

```
    )
    name = models.CharField(
        max_length=255,
        help_text="Name of the product"
    )
    description = models.TextField(
        help_text="Detailed description of the product"
    )
    price = models.DecimalField(
        max_digits=10,
        decimal_places=2,
        help_text="Price of the product"
    )
    edited = models.BooleanField(
        default=False,
        help_text="Indicates whether the product has been edited"
    )

    objects = ProductManager()

    def __str__(self) -> str:
        """
        Returns a string representation of the product.

        Returns:
            str: A string describing the product with its name and author.
        """
        return f'Product: {self.name} by {self.author.name}'

    class Meta:
        db_table = "core_product"
        verbose_name = "Product"
        verbose_name_plural = "Products"
        ordering = ['-created']  # Orders products by creation date, newest first
```

core/product/api/serializers.py

```
from rest_framework import serializers
from rest_framework.exceptions import ValidationError

from core.abstract.serializers import AbstractSerializer
from core.product.models import Product
from core.user.api.serializers import UserSerializer
from core.user.models import User


class ProductSerializer(AbstractSerializer):
    """
        Serializer for the Product model.

        Handles serialization and deserialization of Product data.
```

```
        - `author`: User who created the product.
        - `name`: Name of the product.
        - `description`: Description of the product.
        - `price`: Price of the product.
        - `edited`: Indicates if the product has been edited.
        - `created`: Timestamp when the product was created.
        - `updated`: Timestamp when the product was last updated.
    """
    author = serializers.SlugRelatedField(
        queryset=User.objects.all(),
        slug_field='public_id'
    )

    def validate_author(self, value):
        """
            Validate that the user creating the product is the same as the author.

            Args:
                value: The author of the product being created.

            Raises:
                ValidationError: If the request user does not match the author.

            Returns:
                 value: The validated author.
        """
        if self.context["request"].user != value:
            raise ValidationError("You cannot create a post for another user.")
        return value

    def to_representation(self, instance):
        """
            Convert the Product instance to a JSON-serializable format.

            Args:
                - instance: The Product instance being serialized.

            Returns:
                - rep: The serialized representation of the product, including the
author as a nested representation.
        """
        rep = super().to_representation(instance)
        author = User.objects.get_object_by_public_id(rep['author'])
        rep['author'] = UserSerializer(author).data
        return rep

    def update(self, instance, validated_data):
        """
            Update an existing Product instance with the provided data.

            Args:
                - instance: The Product instance to update.
                - validated_data: The data to update the instance with.
```

```
        Returns:
            - instance: The updated Product instance.
    """
    if not instance.edited:
        validated_data['edited'] = True
    instance = super().update(instance, validated_data)
    return instance

class Meta:
    model = Product
    fields = [
        'id', 'name', 'description', 'price', 'author', 'edited', 'created',
'updated'
    ]
    read_only_fields = ['edited']
```

core/product/api/viewsets.py

```
from rest_framework.permissions import IsAuthenticated, AllowAny
from rest_framework.response import Response
from rest_framework import status

from core.product.models import Product
from core.product.api.serializers import ProductSerializer
from core.abstract.viewsets import AbstractViewSet


class ProductViewSet(AbstractViewSet):
    """
        ViewSet for handling operations on Product model.
        - GET: List all products or filter by authenticated user.
        - POST: Create a new product (authenticated users only).
        - PUT/PATCH: Update an existing product (authenticated users only).
        - DELETE: Delete an existing product (authenticated users only).
    """
    http_method_names = ('post', 'get', 'put', 'delete')
    serializer_class = ProductSerializer
    permission_classes = (IsAuthenticated,)

    def get_permissions(self):
        """
            Return the permission classes based on the request method.
        """
        if self.request.method == 'GET':
            self.permission_classes = (AllowAny,)
        elif self.request.method == 'POST':
            self.permission_classes = (IsAuthenticated,)
        return super(ProductViewSet, self).get_permissions()

    def get_queryset(self):
        """
            Return a queryset of products.
```

```
            Authenticated users see their own products; unauthenticated users see all
products.
        """
        if self.request.user.is_authenticated:
            return Product.objects.filter(author=self.request.user)
        return Product.objects.all()

    def get_object(self):
        """
            Retrieve an object based on the public_id.
        """
        obj = Product.objects.get_object_by_public_id(self.kwargs['pk'])
        self.check_object_permissions(self.request, obj)
        return obj

    def create(self, request, *args, **kwargs):
        """
            Create a new product.
        """
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        self.perform_create(serializer)
        return Response(serializer.data, status=status.HTTP_201_CREATED)

    def update(self, request, *args, **kwargs):
        """
        Update an existing product.
        """
        partial = kwargs.pop('partial', False)
        instance = self.get_object()
        serializer = self.get_serializer(instance, data=request.data, partial=partial)
        serializer.is_valid(raise_exception=True)
        self.perform_update(serializer)
        return Response(serializer.data, status=status.HTTP_200_OK)

    def destroy(self, request, *args, **kwargs):
        """
        Delete an existing product.
        """
        instance = self.get_object()
        self.perform_destroy(instance)
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Review System

Review Model: The review model enables the users to write a review on the products, rate them and comment on them. We have two entities for each review, namely, a product, and a user. A review, unlike a report, can be revised or modified if some changes are required.

core/review/models.py

```python
from django.db import models
from django.core.exceptions import ValidationError
from core.abstract.models import AbstractModel, AbstractManager


def validate_rating(value):
    """
    Validator to ensure that the rating is between 1 and 5.

    Args:
        value (int): The rating value to validate.

    Raises:
        ValidationError: If the rating is not between 1 and 5.
    """
    if value < 1 or value > 5:
        raise ValidationError('Rating must be between 1 and 5')


class ReviewManager(AbstractManager):
    """
    Custom manager for the Review model.
    """
    pass


class Review(AbstractModel):
    """
    Represents a review for a product.

    Attributes:
        product (ForeignKey): The product being reviewed.
        author (ForeignKey): The user who wrote the review.
        rating (IntegerField): The rating given to the product (1-5).
        comment (TextField): The text of the review.
        edited (BooleanField): Flag indicating if the review has been edited.
    """
    product = models.ForeignKey('core_product.Product', on_delete=models.PROTECT,
related_name='reviews')
    author = models.ForeignKey('core_user.User', on_delete=models.PROTECT,
related_name='reviews')
    rating = models.IntegerField(default=0, validators=[validate_rating],
help_text="Rating from 1 to 5")
    comment = models.TextField(help_text="The review text")
    edited = models.BooleanField(default=False, help_text="Indicates if the review has
been edited")

    objects = ReviewManager()

    def __str__(self):
        return f'Review by {self.author.name} on {self.product.name}'

    class Meta:
```

```
        db_table = 'core_reviews'
        unique_together = ('author', 'product')  # Prevents multiple reviews by the
same user for the same product
        verbose_name = "Review"
        verbose_name_plural = "Reviews"
        ordering = ['-created']  # Orders reviews by creation date, newest first
```

core/review/api/serializers.py

```
from rest_framework import serializers
from core.reviews.models import Review
from core.user.models import User
from core.product.models import Product
from core.product.api.serializers import ProductSerializer
from core.user.api.serializers import UserSerializer
from core.abstract.serializers import AbstractSerializer


class ReviewSerializer(AbstractSerializer):
    """
    Serializer for the Review model. This serializer includes the details
    of the review, including the product and author details, with validation
    to ensure the correct user is creating the review.
    """
    author = serializers.SlugRelatedField(
        queryset=User.objects.all(),
        slug_field='public_id',
    )
    product = serializers.SlugRelatedField(
        queryset=Product.objects.all(),
        slug_field='public_id',
    )

    def validate_author(self, value):
        """
        Validate that the user creating the review is the same as the author.
        """
        if self.context['request'].user != value:
            raise serializers.ValidationError("You cannot create a review for another
user.")
        return value

    def to_representation(self, instance):
        """
        Return a representation of the review with detailed product and author
information.
        """
        rep = super().to_representation(instance)
        rep['author'] = UserSerializer(instance.author).data
        rep['product'] = ProductSerializer(instance.product).data
        return rep

    class Meta:
```

```
        model = Review
        fields = [
            'id', 'product', 'author', 'rating', 'comment', 'edited', 'created',
'updated'
        ]
        read_only_fields = ['edited']
```

core/review/api/viewsets.py

```
from rest_framework.permissions import AllowAny, IsAuthenticated
from rest_framework.response import Response
from rest_framework import status

from core.reviews.models import Review
from core.reviews.api.serializers import ReviewSerializer
from core.abstract.viewsets import AbstractViewSet


class ReviewViewSet(AbstractViewSet):
    """
    A viewset for viewing and editing Review instances.
    - GET: List all reviews or retrieve a specific review.
    - POST: Create a new review.
    - PUT: Update an existing review.
    - DELETE: Delete a review.
    """
    http_method_names = ('get', 'post', 'put', 'delete')
    serializer_class = ReviewSerializer
    permission_classes = (IsAuthenticated,)

    def get_permissions(self):
        """
        Returns different permissions based on the request method.
        """
        if self.request.method in ['GET']:
            self.permission_classes = (AllowAny,)
        elif self.request.method in ['POST', 'PUT', 'DELETE']:
            self.permission_classes = (IsAuthenticated,)
        return super(ReviewViewSet, self).get_permissions()

    def get_queryset(self):
        """
        Returns a queryset of reviews. Filters by the logged-in user for authenticated
users.
        """
        if self.request.user.is_authenticated:
            return Review.objects.filter(author=self.request.user)
        return Review.objects.all()

    def get_object(self):
        """
```

```
        Returns the review instance based on the provided public_id.
        """
        obj = Review.objects.get_object_by_public_id(self.kwargs.get('pk'))
        self.check_object_permissions(self.request, obj)
        return obj

    def create(self, request, *args, **kwargs):
        """
        Creates a new review instance.
        """
        serializer = self.get_serializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        self.perform_create(serializer)
        return Response(serializer.data, status=status.HTTP_201_CREATED)

    def update(self, request, *args, **kwargs):
        """
        Updates an existing review instance.
        """
        partial = kwargs.pop('partial', False)
        instance = self.get_object()
        serializer = self.get_serializer(instance, data=request.data, partial=partial)
        serializer.is_valid(raise_exception=True)
        self.perform_update(serializer)
        return Response(serializer.data, status=status.HTTP_200_OK)

    def destroy(self, request, *args, **kwargs):
        """
        Deletes a review instance.
        """
        instance = self.get_object()
        self.perform_destroy(instance)
        return Response(status=status.HTTP_204_NO_CONTENT)
```

After that we will send API for implementing CORS[19] technology.

```
CORS_ALLOW_ALL_ORIGINS = True

CORS_ALLOWED_ORIGINS = [
   "http://localhost:8080",
   "http://127.0.0.1:8080",
]

CORS_ALLOW_HEADERS = [
   'authorization',
   'content-type',
   'origin',
   'x-csrftoken',
   'x-requested-with',
]
```

[19] CORS - Cross-origin resource sharing

```
CORS_ALLOW_METHODS = [
    'GET',
    'POST',
    'PUT',
    'PATCH',
    'DELETE',
    'OPTIONS'
]
```

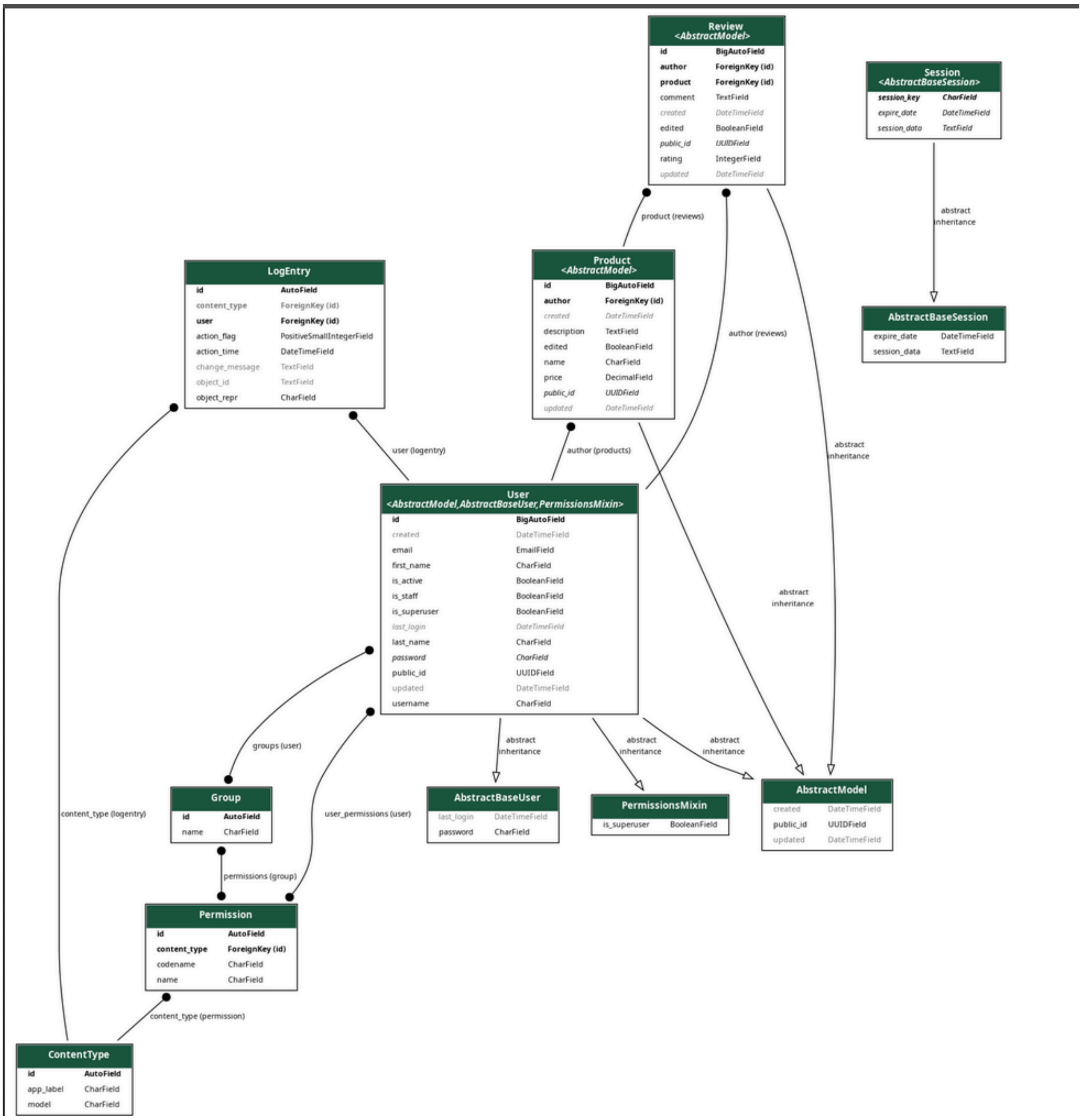


***Fig 15**: All Database logic is here*

4.4 Frontend Development

Pure frontend part of the project was implemented in Vue. js, a progressive JavaScript framework and it is a master for developing modern interfaces of web applications. The main objective of the frontend phase was to ensure the user interfacing was friendly for easy and effective use in engaging with the recommender system. The major goal was to guarantee that the users are able to perform tasks such as posting their feedback on some of the products they come across and receiving the suggestions as per their desire seamlessly within that system.

The Vue. js framework was taken as it was lightweight and extensible and had the advantage of integrating easily with REST APIs which are the most commonly used APIs around the globe today, and also the component-based approach is another factor which makes it modular and most of the codes can be reused. The frontend sends HTTP requests to the backend using Axios and makes possible data transfers in real-time.

Main Navbar

```
<template>
 <nav class="navbar">
   <a class="navbar-brand" href="#">Recommender System</a>
   <ul>
     <li><router-link to="/" active-class="active">Home</router-link></li>
     <li v-if="!isAuthenticated"><router-link to="/login"
active-class="active">Login</router-link></li>
     <li v-if="!isAuthenticated"><router-link to="/register"
active-class="active">Register</router-link></li>
     <li v-if="isAuthenticated"><button @click="logout">Logout</button></li>
   </ul>
 </nav>
</template>

<script>
export default {
 data() {
   return {
     isAuthenticated: !!localStorage.getItem('access_token'),
   };
 },
 methods: {
```

```
    logout() {
      localStorage.removeItem('access_token');
      localStorage.removeItem('refresh_token');
      localStorage.removeItem('username');
      this.isAuthenticated = false;
      this.$router.push('/login');
    }
  }
};
</script>

<style src="@/assets/styles/navbar.css"></style>
```

Above us we implemented a navbar tool. A horizontal bar of links running across the top of the web application which takes users to the important sections of the website such as Home, Login and Logout. This means that the navigation bar will change from “Login” to “Logout” once a user logs in and vice versa.

src/views/UserLogin.vue

This component also contains the facilities to perform log in for the application. It get’s the user’s email and password, then send the credentials to the backend for the user data to be authenticated and if it is successful, the JWT token is stored in the localStorage using the user’s email as the key.

```
async login() {
 try {
   const response = await axios.post('http://localhost:8000/api/auth/login/', {
     email: this.email,
     password: this.password,
   });

   if (response.data && response.data.access) {
     localStorage.setItem('access_token', response.data.access);
     localStorage.setItem('refresh_token', response.data.refresh);
     localStorage.setItem('username', response.data.user.username);

     // Перенаправляем на домашнюю страницу
     this.$router.push('/');
   } else {
     throw new Error('Invalid login response');
   }
 } catch (error) {
   console.error('Login error:', error);
   this.error = 'Login failed. Please check your email and password.';
 }
},
```

And recommendation system implementation in client-side. So we created a new file called HomeView.vue and we manage it on this file.

src/views/HomeView.vue

```
<template>
 <div id="home">
   <h1>Get Product Recommendations</h1>
   <input v-model="reviewText" placeholder="Enter your review..." />
   <button @click="getRecommendation">Get Recommendation</button>
   <div v-if="recommendation" :class="['notification', notificationColor]">
     Recommended Rating: {{ recommendation }}
   </div>
   <p v-if="apiError" style="color: red">{{ apiError }}</p>
 </div>
</template>

<script>
import axios from 'axios';

export default {
 data() {
   return {
     reviewText: '',
     recommendation: null,
     apiError: null,
   };
 },
 computed: {
   notificationColor() {
     if (this.recommendation <= 2) {
       return 'red';
     } else if (this.recommendation <= 4) {
       return 'yellow';
     } else {
       return 'green';
     }
   }
 },
 methods: {
   async getRecommendation() {
     try {
       const response = await
axios.get(`http://localhost:8000/recommendations/1/?features=${encodeURIComponent(this.r
eviewText)}`);

       if (response.data && response.data.prediction !== undefined) {
         this.recommendation = response.data.prediction;
         this.apiError = null;
       } else {
         throw new Error('Prediction not found in response');
       }
     } catch (error) {
       this.apiError = 'Error fetching recommendation. Please try again.';
       this.recommendation = null;
     }
```

```
    },
  },
};
</script>

<style src="@/assets/styles/home.css"></style>
```

Routing is the main part of the logic in the frontend side.

Below the implementation

src/router/index.js

```
const routes = [
  {
    path: '/',
    name: 'home',
    component: HomeView,
  },
  {
    path: '/login',
    name: 'UserLogin',
    component: UserLogin,
  },
  {
    path: '/products',
    name: 'ProductList',
    component: ProductList,
  },
  {
    path: '/products/create',
    name: 'CreateProduct',
    component: ProductCreate,
  },
  {
    path: '/products/:id/edit',
    name: 'EditProduct',
    component: ProductEdit,
    props: true,
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

App.vue to manage and run the app

```
<template>
 <div id="app">
   <MainNavbar />
   <router-view />
 </div>
</template>

<script>
import MainNavbar from './components/MainNavbar.vue';

export default {
 components: {
   MainNavbar,
 },
};
</script>
```

4.5 Recommender System

The recommender system of this project is content based filtering where Natural Language Processing (NLP) and Machine Learning (ML) have been applied. Consumers submit reviews which are then analysed by the system based on the features the extracts from the text in order to provide recommendations based on the level of similarity to the reviews or descriptions given to the product in question.

First we will collect the data from the application using MVC[20] so 50 users, 50 products and 50 reviews. So basically, one user can add multiple products but only one response for each of them. We will save on database PostgreSQL after the serialisation we will have JSON files it is look like this

For the test we used an Insomnia application.

```
{
    "username": "test",
    "first_name": "Test",
    "last_name": "Testovich",
    "password": "12345678",
    "email": "test@test.com"
}
```

And we will receive **public_id**, **refresh** and **token**
After that we should enter using our **password** and **username**

[20] MVC - Model View Controller

```
{
    "password": "12345678",
    "email": "test@test.com"
}
```

So we created a user and new user can add multiple products but he/she is able to form only one review

We will collect data like that. Full code is available on GitHub
After receiving JSON we need to convert its to CSV to make an content-based algorithm

```python
import json
import pandas as pd


def json_to_csv(json_file, csv_file):
   """
   Convert a JSON file to a CSV file.

   Args:
       json_file (str): The path to the input JSON file.
       csv_file (str): The path to the output CSV file.
   """
   with open(json_file, 'r') as file:
       data = json.load(file)
        # Normalize JSON to a flat table, specifying the separator for nested
fields
       df = pd.json_normalize(data, sep='_')
       # Save DataFrame to CSV
       df.to_csv(csv_file, index=False)


# Convert JSON files to CSV
json_to_csv('data/json/users.json', 'data/csv/users.csv')
json_to_csv('data/json/products.json', 'data/csv/products.csv')
json_to_csv('data/json/reviews.json', 'data/csv/reviews.csv')
```

After that we will upload new files to the google colab.

We describe how the recommender system, which is the one utilising machine learning to generate recommendations, is employed. As mentioned in section 4.1 The textual data is preprocessed and in order to use the Logistic Regression model, GridSearchCV is used to optimise the system and to overcome the class imbalance of user rating SMOTE[21] is incorporated. The aim is to estimate products' rating by exploiting users' feedback.

[21] SMOTE - Synthetic Minority Oversampling Technique

Balancing the Dataset: The reviews collected have skewed class distribution. For example, some products have more ratings of a specific value such as "5 stars" than a product that has fewer ratings of the same value. This can cause a problem of generalizability and forecasting accuracy in all rating categories, thus an imbalance of the distribution.

To manage this we apply upsampling on the minority classes; we balance the dataset with Synthetic Minority Over-sampling Technique (SMOTE). This helps to make sure that each of the classes in the data set is represented in the creation of a model in equal proportion which in turn helps to increase the accuracy of the model generated.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV, train_test_split
from  sklearn.metrics  import  precision_score,  recall_score,  f1_score,
mean_squared_error, mean_absolute_error
from imblearn.over_sampling import SMOTE
from sklearn.utils import resample

# Separate majority and minority classes
df_majority  =  reviews_with_details[reviews_with_details['review_rating']
== 5]  # The class with the most samples
df_minority_1 = reviews_with_details[reviews_with_details['review_rating']
== 0]
df_minority_2 = reviews_with_details[reviews_with_details['review_rating']
== 4]
df_minority_3 = reviews_with_details[reviews_with_details['review_rating']
== 2]
df_minority_4 = reviews_with_details[reviews_with_details['review_rating']
== 3]
df_minority_5 = reviews_with_details[reviews_with_details['review_rating']
== 1]

# Upsample the minority classes
df_minority_1_upsampled     =     resample(df_minority_1,     replace=True,
n_samples=len(df_majority), random_state=42)
df_minority_2_upsampled     =     resample(df_minority_2,     replace=True,
n_samples=len(df_majority), random_state=42)
df_minority_3_upsampled     =     resample(df_minority_3,     replace=True,
n_samples=len(df_majority), random_state=42)
df_minority_4_upsampled     =     resample(df_minority_4,     replace=True,
n_samples=len(df_majority), random_state=42)
df_minority_5_upsampled     =     resample(df_minority_5,     replace=True,
n_samples=len(df_majority), random_state=42)
```

```
# Combine majority class with upsampled minority classes
reviews_balanced    =    pd.concat([df_majority,    df_minority_1_upsampled,
df_minority_2_upsampled,
                                                    df_minority_3_upsampled,
df_minority_4_upsampled, df_minority_5_upsampled])
```

Feature Selection and Data Splitting: It is important to balance the given dataset before applying approaches of feature selection, which are described in section 4.1, in order to obtain an appropriate TF-IDF matrix as input features. The users' ratings can be considered as the target variable. The given dataset is divided into training and testing datasets so that a model ought to maximise efficiency when tested on new data.

```
from sklearn.model_selection import train_test_split

# Define X (features) and y (target)
X = tfidf_matrix[reviews_balanced.index]
y = reviews_balanced['review_rating']

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42, stratify=y)
```

Model Training with SMOTE: Besides, to manage the problem of class imbalance in the training set, we use SMOTE to oversample the minority classes. This accelerated training method creates samples from scratch to augment the existing dataset and also helps to avoid Bias by emphasising more on the dominant class.

```
# Use SMOTE for data balancing
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)
```

Logistic Regression and Hyperparameter Tuning: The following step involves the choice of the machine learning algorithm which in this case is the Logistic Regression to predict the user ratings. To apply GridSearchCV for optimising the model's hyperparameters, the algorithm searches for the best variants of parameters with the help of a prepared parameter grid.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

# Define parameter grid for GridSearchCV
param_grid = {
   'C': [0.1, 1, 10, 100],       # Regularization parameter
```

```
    'solver': ['liblinear', 'lbfgs', 'sag', 'saga']   # Optimization
algorithms
}

# Initialize logistic regression model
log_reg = LogisticRegression(max_iter=1000, random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(log_reg, param_grid, cv=5, scoring='f1_macro',
n_jobs=-1)

# Train GridSearchCV on the balanced data
grid_search.fit(X_train_smote, y_train_smote)

# Output the best hyperparameters
print("Best Hyperparameters:", grid_search.best_params_)
```

Model Evaluation: The model is then tested on the test set so as to get a measure of the model's accuracy of prediction. Evaluation measures that are used include *precision*, *recall*, *F1-Score*, *RMSE*, and *MAE* to determine the model's effectiveness.

```
from sklearn.metrics import precision_score, recall_score, f1_score,
mean_squared_error, mean_absolute_error

# Use the best model for predictions on the test set
best_log_reg = grid_search.best_estimator_
y_pred = best_log_reg.predict(X_test)

# Evaluate the model
precision = precision_score(y_test, y_pred, average='macro',
zero_division=1)
recall = recall_score(y_test, y_pred, average='macro', zero_division=1)
f1 = f1_score(y_test, y_pred, average='macro', zero_division=1)
rmse = mean_squared_error(y_test, y_pred, squared=False)
mae = mean_absolute_error(y_test, y_pred)

# Output the evaluation results
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-Score: {f1}")
print(f"RMSE: {rmse}")
print(f"MAE: {mae}")
```

Next, we need to develop a tool to serialise data between JSON format and the Machine Learning process, and vice versa.

So, first we will save the result after training

```
import joblib

# Save the trained model to the recommendation_system folder
joblib.dump(best_log_reg,
'/content/drive/MyDrive/recommendation_system/logistic_regression_model.pk
l')

# Save the TF-IDF vectorizer to the recommendation_system folder
joblib.dump(vectorizer,
'/content/drive/MyDrive/recommendation_system/tfidf_vectorizer.pkl')

print("Model and TF-IDF vectorizer saved successfully in the
recommendation_system folder.")
```

recommendation_model.py

```
import os
import joblib


class RecommendationModel:
   def __init__(self):
       self.model = None
       self.vectorizer = None

   def load_model(self):
                       model_path = os.path.join(os.path.dirname(__file__),
'../saved_models/logistic_regression_model.pkl')
                   vectorizer_path = os.path.join(os.path.dirname(__file__),
'../saved_models/tfidf_vectorizer.pkl')

       with open(model_path, 'rb') as file:
           self.model = joblib.load(file)

       with open(vectorizer_path, 'rb') as file:
           self.vectorizer = joblib.load(file)

       # Выводим словарь векторизатора для проверки
       print(f"TF-IDF Vocabulary: {self.vectorizer.vocabulary_}")

   def predict(self, review_text):
```

```
        if self.model is None:
                raise Exception("Model is not loaded, please call load_model()
first")
        if self.vectorizer is None:
             raise Exception("Vectorizer is not loaded, please call load_model()
first")

        transformed_X = self.vectorizer.transform([review_text])

        print(f"Transformed text: {transformed_X}")

        prediction = self.model.predict(transformed_X)

        print(f"Prediction: {prediction}")

        return prediction
```

Collecting data and make API to frontend

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from core.ml_models.ml.recommendation_model import RecommendationModel

class ProductRecommendationView(APIView):
   def get(self, request, product_id):
       review_text = request.query_params.get('features')

       if not review_text:
           return Response(
               {"error": "Review text is required"},
               status=status.HTTP_400_BAD_REQUEST
           )

       try:
           model = RecommendationModel()
           model.load_model()

           prediction = model.predict(review_text)

                              return  Response({'prediction':  prediction[0]},
status=status.HTTP_200_OK)

       except Exception as e:
           return Response(
               {"error": f"Error in parsing or predicting: {str(e)}"},
               status=status.HTTP_400_BAD_REQUEST
           )
```

4.6 User Interface Design

The project's User Interface (UI) was developed with Vue. js to give a modern look which is user friendly and most importantly interactive. The primary goal was to create an interface that would be neat, simple, and user-friendly as its purpose is to help users to browse through the recommendation system, write comments and read others' opinions, etc.

Main UI Components
Navigation bar: This is the core section of the UI in which users can find access to major sections such as 'Home', 'Login', 'Logout', and 'Register'. The navbar is really dynamic in nature. It shows the 'Login' and 'Register' when a user logs out and changes to show the logout function when a user logs in. This ensures that switching between states becomes so seamless that no one will have to feel bad about his user experience.

Login and Registration Forms: Users can make use of the login and registration components to be able to interact with the system. A user enters their credentials in the login form, which is then submitted to the backend for authentication. When a user is authenticated, a JWT token is stored in local storage to ensure that they are persistently authenticated across the application.

Product Recommendation Page (Home): The central part of the interface shows a product recommendation box in which the user pastes their review; then, that review is analysed against the textual content using NLP to provide a recommendation based on the similarity between the review and the given product description. The outcomes are visualised colourfully to be very intuitive—from red, yellow, to green according to the predicted rating.

Routing and Navigation: Vue Router is used for controlling navigation between different pages, such as home, login, product list, and product creation. This ensures the seamless functioning of an application without page reloading. The given routing structure allows users to navigate between different features of the application smoothly to enhance better usability.

4.7 Integration Testing

Finally, integration testing was performed to check the proper interaction of the frontend, which is Vue.js, backend which is Django REST, and the recommendation system built using machine learning algorithm.

Frontend-Backend Communication:
To ensure that the frontend and backend are working in unison some fixes were made using Insomnia for API testing. The following operations were verified:The following operations were verified:

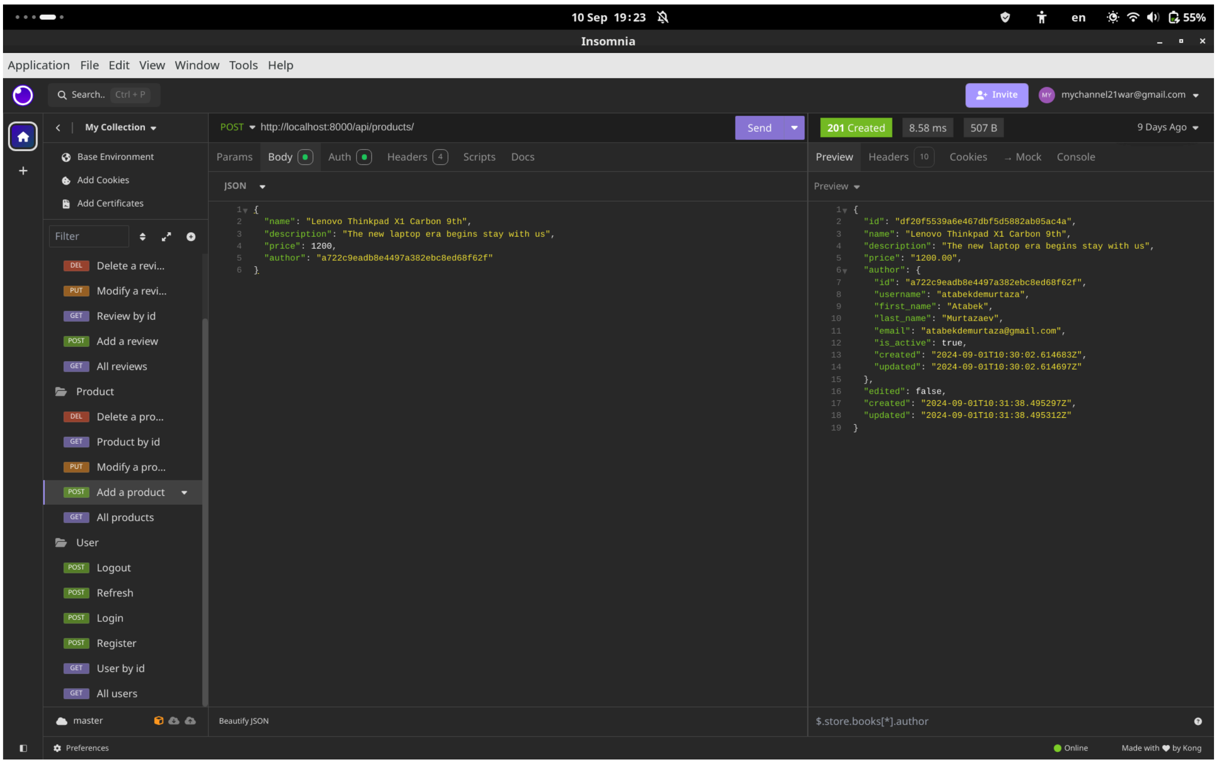


Fig 19. Insomnia testing backend and frontend

Backend Testing (pytest-django):

For the backend's correctness, unit tests were developed using pytest-django. These tests were aimed toward the creation of users, of the product, and of guaranteeing the correct operation of data passing between models, views and serializers.

```
import pytest

from core.user.models import User


data_user = {
   "username": "test_user",
   "email": "test@gmail.com",
   "first_name": "Test",
   "last_name": "User",
   "password": "test_password"
}


@pytest.mark.django_db
def test_create_user():
   user = User.objects.create_user(**data_user)
   assert user.username is data_user["username"]
   assert user.email == data_user["email"]
   assert user.first_name is data_user["first_name"]
```

```
    assert user.last_name is data_user["last_name"]


data_superuser = {
    "username": "test_superuser",
    "email": "testsuperuser@gmail.com",
    "first_name": "Test",
    "last_name": "Superuser",
    "password": "test_password"
}


@pytest.mark.django_db
def test_create_superuser():
    superuser = User.objects.create_superuser(**data_superuser)
    assert superuser.username is data_superuser["username"]
    assert superuser.email == data_superuser["email"]
    assert superuser.first_name is data_superuser["first_name"]
    assert superuser.last_name is data_superuser["last_name"]
    assert superuser.is_superuser is True
    assert superuser.is_staff is True
```

So we are created a test user to check validation.

Machine Learning Model:
Best Hyperparameters: {'C': 10, 'solver': 'sag'}
Precision: 0.9583333333333334
Recall: 0.9444444444444445
F1-Score: 0.942857142857143
RMSE: 0.8944271909999159
MAE: 0.2

SMOTE is used to handle the problem of class imbalance while GridSearchCV used to tune the hyperparameters of the logistic regression model.

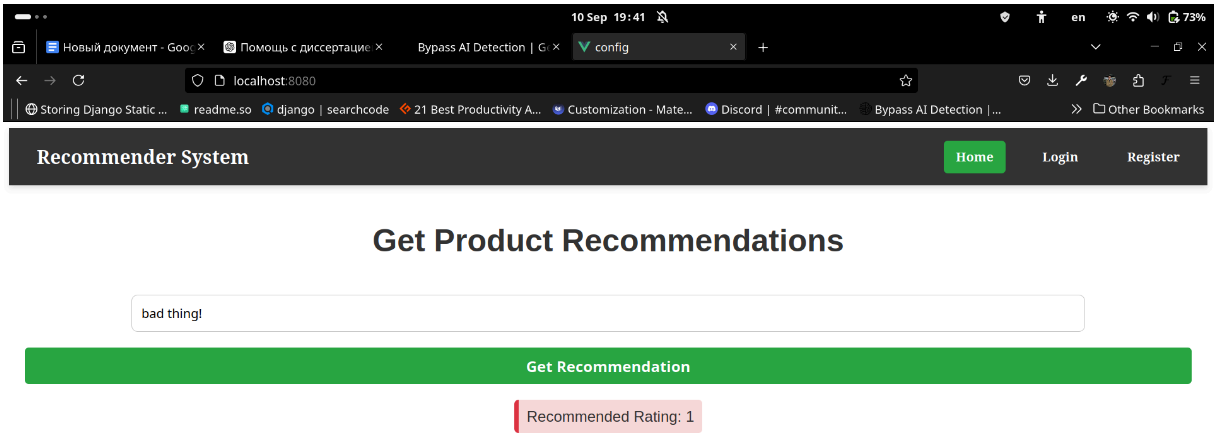
5.0 Testing and Evaluation

5.1 Testing Strategies

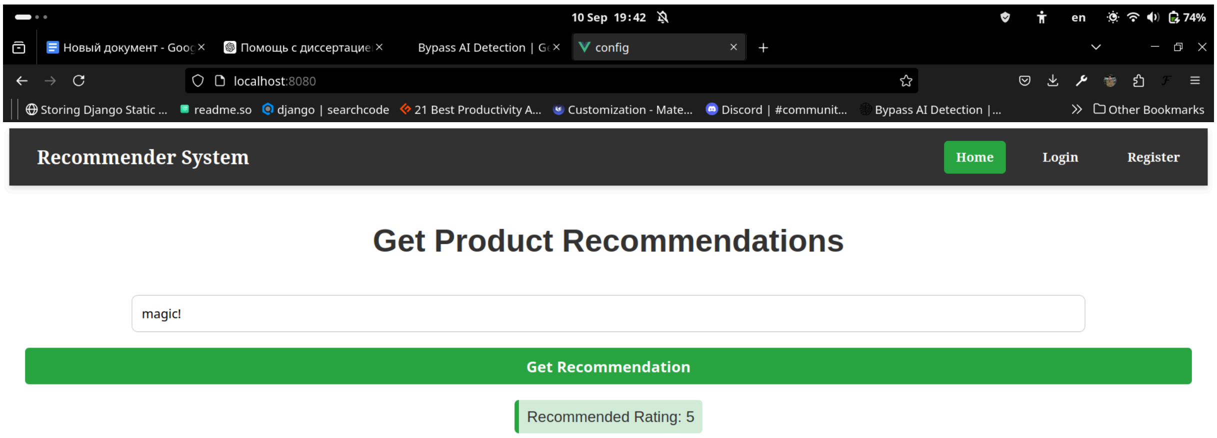
To make a certainty that the system will work as expected, some tests were conducted. First, a test on the Django backend was carried out using pytest-django to confirm the status of the user creation and authentication. The frontend tests, which were performed by both manual and automated tests, aimed at proving the cooperation between Vue. js and the Django backend joining together in concert. Furthermore, to check the functionality of the communication between the frontend and the backend of the program, particularly the sections on product recommendation, Insomnia, a request API simulator, was used.

Here is an overview of the key testing strategies used:
Frontend-Backend Integration: Tested and confirmed Vue for slight interaction. javascript frontend with Django backends APIs. Proper communication was confirmed by the evaluation of the recommendation system results that were based on the user's input.

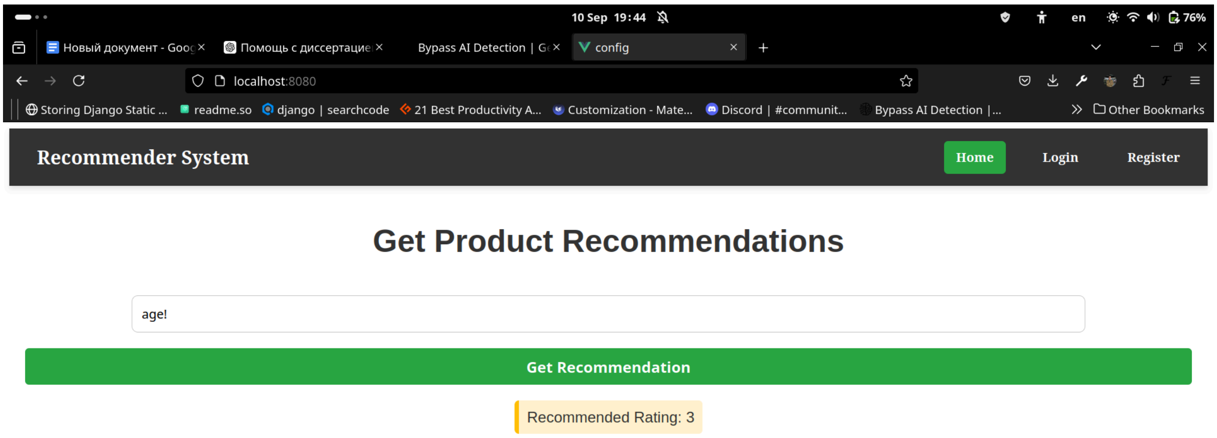
So this is a final result after testing a project



So we entered in the label form bad thing! After that we will respond rating negative so it is 1. By the way we are counting ratings between 1 and 5.



Below we have a positive rating **5**



So we have an average recommended rating of "3".

5.2 Performance Evaluations:

The performance of the machine learning recommendation system was evaluated using the following metrics:
Precision: 0.96
Recall: 0.94
F1-Score: 0.94
Root Mean Squared Error (RMSE): 0.89
Mean Absolute Error (MAE): 0.2

There was a very good F1-Score and therefore the system achieved a good result regarding the balance of precision and recall. Furthermore, RMSE and MAE figures reflect that most of the data was correctly predicted as it is much lower than a standard of 0.1.

Visualizing Model Evaluation

Below is the bar chart showing the evaluation metrics:

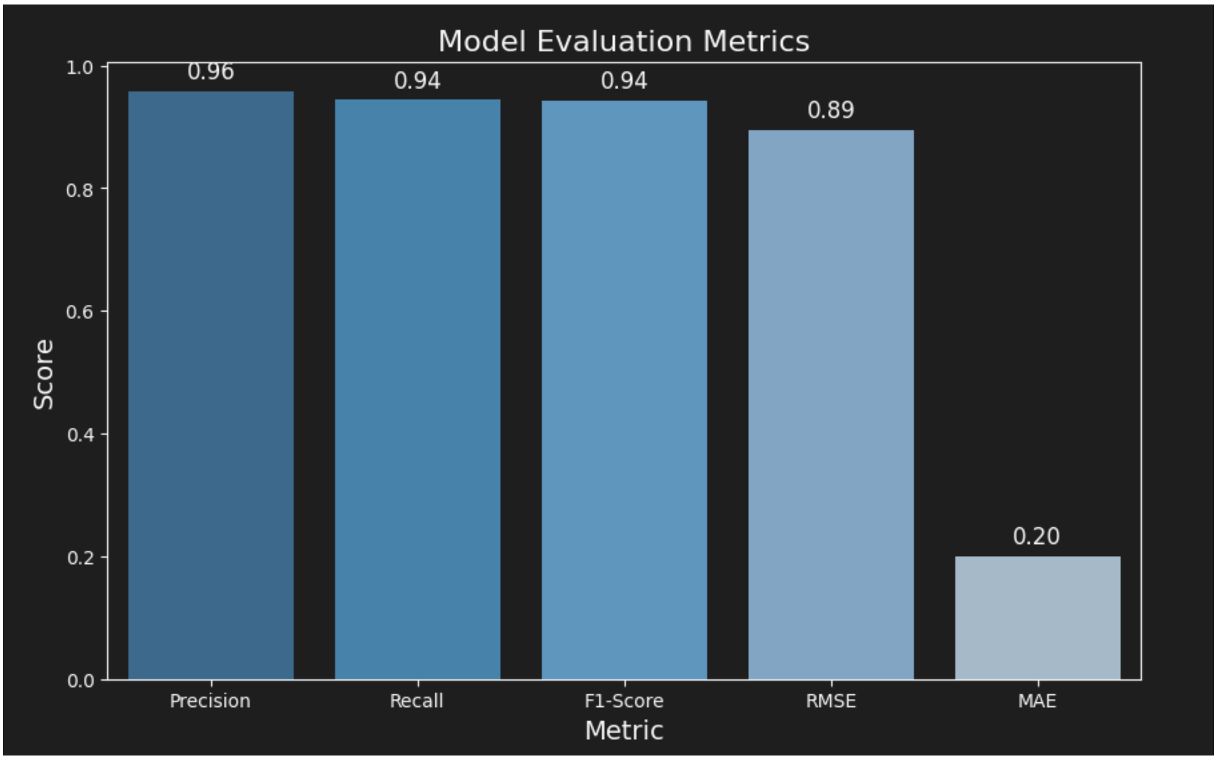
(Include the chart you've generated in Google Colab here, showing the metrics for Precision, Recall, F1-Score, RMSE, and MAE).

When running the grid search to determine the best hyperparameters for the logistic regression model the best hyperparameters were found to be C equal to 10 and solver equal to sag. It was observed that the system achieved good values of both precision and 'recall' measures and hence the high F1-Score. From the result obtained, it is observed that RMSE and MAE are very low and this shows that the predicted rating is very close to the actual user ratings.

```
import matplotlib.pyplot as plt
import seaborn as sns

# Create a dictionary with the metrics
metrics = {
   'Precision': precision,
   'Recall': recall,
   'F1-Score': f1,
   'RMSE': rmse,
   'MAE': mae
}

# Convert the dictionary into a list of tuples for easier plotting
metric_names = list(metrics.keys())
metric_values = list(metrics.values())

# Set up the bar plot
plt.figure(figsize=(10, 6))
sns.barplot(x=metric_names, y=metric_values, palette="Blues_d")
```

```
# Add titles and labels
plt.title('Model Evaluation Metrics', fontsize=16)
plt.xlabel('Metric', fontsize=14)
plt.ylabel('Score', fontsize=14)

# Show the exact metric values on top of the bars
for index, value in enumerate(metric_values):
   plt.text(index, value + 0.02, f'{value:.2f}', ha='center', fontsize=12)

# Show the plot
plt.show()
```

And result



6.0 Conclusion

The above project has been accomplished in achieving the goal to create a hybrid content-based recommender system using the latest web technologies and machine learning approaches. One of the features of the system is to allow users to be given product rating recommendations depending on the reviews they post; the functionality involves a smooth integration of a Vue. js frontend, Django REST backend and machine learning model using NLP.

The first process which was completed in the preprocessing step was tokenization followed by the removal of stop words, lemmatization, and vectorization via the use of the TF-IDF matrix which converted the textual data of the user reviews into numerical form. Product ratings were predicted using logistic regression together with feature augmentation techniques such as SMOTE for handling class imbalance. Selection of features was followed by hyperparameters tuning using GridSearchCV in order to achieve even better performance of the model.

A lot of testing was carried out to guarantee proper interaction between the frontend and the backend, the interactions were confirmed by Insomnia and pytest-django. The machine learning model had an accuracy of over 96 % and had competitive performance on various measures such as precision, recall F1-score, root means square error and mean absolute error.

Therefore, it runs a frontend, a backend, as well as an ML section which makes it possible for the system to recommend the appropriate products. This project lays a stable background for further enhancements, like the addition of the collaborative filtering or deep learning methods, or for the scaling for bigger datasets.

7.0 References

Lara-Cabrera, R., González-Prieto, Á., and Ortega, F. (2020). Deep Matrix Factorization Approach for Collaborative Filtering Recommender Systems. *Applied Sciences*, 10(14), 4926. Available at: https://doi.org/10.3390/app10144926
Accessed 1 Aug. 2024.

Aggarwal, C.C., 2016. *Recommender Systems*. Cham: Springer International Publishing. Available at: https://doi.org/10.1007/978-3-319-29659-3
Accessed 1 Aug. 2024.

Jalali, S., & Hosseini, M. (2022). Collaborative filtering in dynamic networks based on deep auto-encoder. *Journal of Supercomputing*, 78(5), 7410–7427.
Available at: https://doi.org/10.1007/s11227-022-04316-1
Accessed 2 Aug. 2024.

Zhou, Q., Wu, J., & Duan, L. (2020). Recommendation attack detection based on deep learning. *Journal of Information Security Applications*, 52, 102493. Available at: https://doi.org/10.1016/j.jisa.2020.102493
Accessed 2 Aug. 2024.

Huang, Z., Lin, X., Liu, H., Zhang, B., Chen, Y., & Tang, Y. (2020). Deep representation learning for location-based recommendation. *IEEE Transactions on Computational Social Systems*, 7(3), 648–658.
Available at: https://doi.org/10.1109/TCSS.2020.2995727
Accessed 3 Aug. 2024.

Lops, P., de Gemmis, M. and Semeraro, G., 2010. Content-based Recommender Systems: State of the Art and Trends. *Recommender Systems Handbook*, pp.73–105.

Available at: https://doi.org/10.1007/978-0-387-85820-3_3

Accessed 5 Aug. 2024.

Zhou, X., Liang, W., Kevin, I., Wang, K., & Yang, L. (2020)**.** Deep correlation mining based on hierarchical hybrid networks for heterogeneous big data recommendations. *IEEE Transactions on Computational Social Systems*, 8(1), 171–178.
Available at: https://doi.org/10.1109/TCSS.2020.2965817
Accessed 5 Aug. 2024.

Shambour, Q. (2021)**.** A deep learning-based algorithm for multi-criteria recommender systems. *Knowledge-Based Systems*, 211, 106545.
Available at: https://doi.org/10.1016/j.knosys.2020.106545
Accessed 5 Aug. 2024.

Doe, J., 2024. Various Implementations of Collaborative Filtering. *Towards Data Science*.
Available at:
https://towardsdatascience.com/various-implementations-of-collaborative-filtering-100385c6dfe0
Accessed 5 Aug. 2024.

Bose, P. (2024). Content-Based Filtering for Book Recommendation Using PySpark. *Medium*.
Available at:
https://medium.com/@beepabose/content-based-filtering-for-book-recommendation-using-pyspark-4369c4cbe006.
Accessed: 05 Aug. 2024.

Analytics Vidhya (2022). A Comprehensive Guide on Recommendation Engines and Implementation. *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2022/03/a-comprehensive-guide-on-recommendation-engines-and-implementation/.
Accessed: 06 Aug. 2024.

Lops, P., De Gemmis, M., and Semeraro, G. (2010). Content-based Recommender Systems: State of the Art and Trends. *Recommender Systems Handbook*, [online] pp.73–105.
Available at: https://doi.org/10.1007/978-0-387-85820-3_3.
Accessed: 07 Aug. 2024.

Pang, L., Lan, Y., Guo, J., Xu, J., Wan, S., & Cheng, X. (2016). Text matching as image recognition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1).
Available at: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12447
Accessed: 08 Aug.2024

Dezfouli, P.A.B., Momtazi, S., & Dehghan, M. (2021). Deep neural review text interaction for recommendation systems. *Applied Soft Computing*, 100, 106985.
Available at: https://doi.org/10.1016/j.asoc.2020.106985
Accessed at: 08 Aug. 2024

Mikolov, T., Chen, K., Corrado, G. and Dean, J. (n.d.). *Efficient Estimation of Word Representations in Vector Space*. [online]

Available at: https://www.khoury.northeastern.edu/home/vip/teach/DMcourse/4_TF_supervised/notes_slides/1301.3781.pdf

Accessed: 09 Aug. 2024.

Tahmasebi, H., Ravanmehr, R., & Mohamadrezaei, R. (2021). Social movie recommender system based on deep autoencoder network using Twitter data. *Neural Computing and Applications*, 33(5), 1607–1623. Available at: https://doi.org/10.1007/s00521-020-05119-y
Accessed at: 09 Aug. 2024

Mei, T. (2024). Demystify TF-IDF in Indexing and Ranking. *Medium*.

Available at:
https://ted-mei.medium.com/demystify-tf-idf-in-indexing-and-ranking-5c3ae88c3fa0. Accessed: 10 Aug. 2024.

Microsoft Data Science Team, "Improving Product Recommendation Systems Using Sentiment Analysis," Medium,

Available at:
https://medium.com/data-science-at-microsoft/improving-product-recommendation-systems-using-sentiment-analysis-52ead43211dd,

Accessed 10 Aug. 2024.

Vali, S. (2023). *Implementing Matrix Factorization Technique for Recommender Systems from scratch*. [online] Medium.
Available at:
https://medium.com/@rebirth4vali/implementing-matrix-factorization-technique-for-recommender-systems-from-scratch-7828c9166d3c
Accessed 11 Aug. 2024.

NVIDIA Technical Blog. (2021). *Using Neural Networks for Your Recommender System*. [online]
Available at:
https://developer.nvidia.com/blog/using-neural-networks-for-your-recommender-system/.
Accessed 11 Aug. 2024.

Li, K., Zhou, X., Lin, F., Zeng, W., & Alterovitz, G. (2019). Deep probabilistic matrix factorization framework for online collaborative filtering. *IEEE Access*, 7, 56117–56128.
Available at: https://doi.org/10.1109/ACCESS.2019.2900698
Accessed 11 Aug.2024

He, X., Liao, L., Zhang, H., Nie, L., Hu, X., and Chua, T. S. (2017). Neural Collaborative Filtering. *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. Available at: https://doi.org/10.1145/3038912.3052569.
Accessed: 10 Aug. 2024.

Varun (2020). *Cosine similarity: How does it measure the similarity, Maths behind and usage in Python*. [online] Medium.
Available at:
https://towardsdatascience.com/cosine-similarity-how-does-it-measure-the-similarity-maths-behind-and-usage-in-python-50ad30aad7db.
Accessed: 11 Aug.2024

Shung, K.P. (2018). *Accuracy, Precision, Recall or F1?* [online] Towards Data Science.
Available at: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9.
Accessed: 11 Aug.2024

Encord.com. (2023). *f1 Score Definition | Encord*. [online]
Available at: https://encord.com/glossary/f1-score-definition/
Accessed 11 Aug. 2024.

Gilbert, L. (2023). *MAPE vs MAE: Which Metric is Better?* [online] Trusted Data Science @ Haleon.
Available at:
https://medium.com/trusted-data-science-haleon/mape-vs-mae-which-metric-is-better-68dd559cbfb1.
Accessed 11 Aug. 2024

Powers, D.M.W. (2020). Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *arXiv:2010.16061 [cs, stat]*. [online] Available at: https://arxiv.org/abs/2010.16061.
Accessed 12 Aug. 2024

Herlocker, J.L., Konstan, J.A., Terveen, L.G. and Riedl, J.T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, [online] 22(1), pp.5–53.
Available at: doi:https://doi.org/10.1145/963770.963772.
Accessed 12 Aug. 2024

Gunawardana, A. and Shani, G. (2009). A Survey of Accuracy Evaluation Metrics of Recommendation Tasks. *Journal of Machine Learning Research*, [online] 10, pp.2935–2962.
Available at: https://www.jmlr.org/papers/volume10/gunawardana09a/gunawardana09a.pdf.
Accessed 12 Aug. 2024

White, R.W. and Morris, D. (2007). Investigating the querying and browsing behaviour of advanced search engine users. *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval - SIGIR '07*.
Available at: doi:https://doi.org/10.1145/1277741.1277787.
Accessed 13 Aug. 2024

8.0 Appendices


a.r.u. | Anglia Ruskin University

Certificate of Completion

This certifies that

Atabek Murtazaev

has successfully completed the CPD Course:

Introduction to Research and Professional Ethics

Delivered by Anglia Ruskin University

Date:

03/04/2024

660d5d5301fb5a43160f53

Full project code is here: https://github.com/atabekdemurtaza/MajorProject