Final Report on Advanced Book Search & Recommender APIs

Names

May 15, 2024

Abstract

This report presents a detailed development summary of the Advanced Book Search and Recommendations system. The system was created with the aim of providing fast, relevant, and personalized recommendations and search algorithms to all manner of bookstores, libraries, and other facilities. This report details the research, development, testing, and deployment stages of the project.

Acknowledgement

We owe great thanks to many people who helped and supported us during the designing and completing our project as well as the writing of this report also. First and foremost, the gratitude goes to our project supervisor Dr. S. Kirushanth, who encouraged, guided, and supported us to complete each milestone of our project from the beginning to the end. And also, we thank our friends, who have shared their knowledge and experiences for our project development. Apart from the efforts of the project team, the success of the project depends on the encouragement and guidelines given by many others. We also thank our seniors, colleagues, and instructors of the Faculty of Applied Science. Finally, we thank everyone who guided us towards the success.

Thank you.

Declaration

We hereby declare that this report submitted for evaluation of the course module IT3162 leading to the award of Bachelor of Information Technology is entirely our own work and the contents taken from the work of others has been cited and acknowledged within the text. This report has not been submitted for any degree in this university or any other institution.

Reg.No	Name	Signature
2019/ICT/24	Mithini Rathnayake	
2019/ICT/35	M.A.Ihsana	
2019/ICT/46	S.G.Seyone	
2019/ICT/64	Charitha Dilman	
2019/ICT/66	Tharushi Nimnadi	
2019/ICT/85	Ashma Sandeepa	
2019/ICT/98	Fathima Zeena	
2019/ICT/108	Sonali Kalpani	

I hereby certified that I have supervised this project.

Dr.S.Kirushanth
Supervisor
Senior Lecturer
Head of the department
Department of Physical Science,
Faculty of Applied Sciences.

Date

Contents

1 Introduction

The digital age has revolutionized the way we interact with literature, providing unprecedented access to a vast array of books through online platforms. However, despite the convenience of digital bookstores, users often encounter challenges in navigating the plethora of available titles, finding books that match their interests, and ensuring the authenticity of transactions, particularly in the realm of used book sales. Addressing these challenges, our project focuses on the development of a used book sales system powered by blockchain technology. By integrating blockchain, we aim to enhance transparency, security, and trust in the used book marketplace, mitigating concerns related to counterfeit books and fraudulent transactions. Furthermore, by incorporating content-based search algorithms, our system offers personalized recommendations to users, guiding them towards books that align with their tastes and preferences. This report provides a comprehensive overview of our project, detailing the objectives, methodology, design, development, testing, deployment, and future prospects of our innovative solution. Through this endeavor, we seek to revolutionize the used book marketplace, fostering a seamless and trustworthy experience for book enthusiasts worldwide.

2 Problem statement

The domain of online bookstores, encompassing both new and used books, constitutes a significant segment of the e-commerce industry. However, the current landscape of online book sales platforms, particularly in the realm of used books, presents several challenges that hinder user experience and trustworthiness. Existing solutions often lack robust mechanisms for facilitating efficient search and discovery processes, resulting in user frustration and suboptimal outcomes. Users encounter difficulties in navigating through the vast array of available titles, and concerns regarding the authenticity and provenance of books persist, casting doubt on the reliability of transactions.

Inadequate search algorithms exacerbate the problem, failing to provide personalized recommendations tailored to individual user preferences. As a result, users struggle to find relevant titles amidst the extensive catalog, hampering their overall experience. Furthermore, the management of bookstores faces challenges in efficiently handling inventory and ensuring the quality and authenticity of used books.

To address these challenges, our project proposes a comprehensive solution that leverages blockchain technology and advanced search algorithms. By integrating blockchain, we aim to enhance transparency, security, and trust in the used book marketplace. Blockchain ensures immutable records of transactions, mitigating concerns related to counterfeit books and fraudulent activities. Additionally, our solution focuses on improving the performance of search algorithms to provide users with personalized recommendations, facilitating smoother transactions and enhancing user satisfaction.

In summary, our project seeks to revolutionize the way users interact with online bookstores by addressing the inefficiencies and challenges prevalent in the current landscape. By providing a robust and user-friendly platform for buying and selling used books, we aim to foster a seamless and trustworthy experience for both book enthusiasts and bookstore management.Used references are BookSwap.lk and UsedBooks.lk

3 Literature Review

4 Methodology

Prior to proposal presentation, we created system block diagrams for how the various systems are to function within the system.

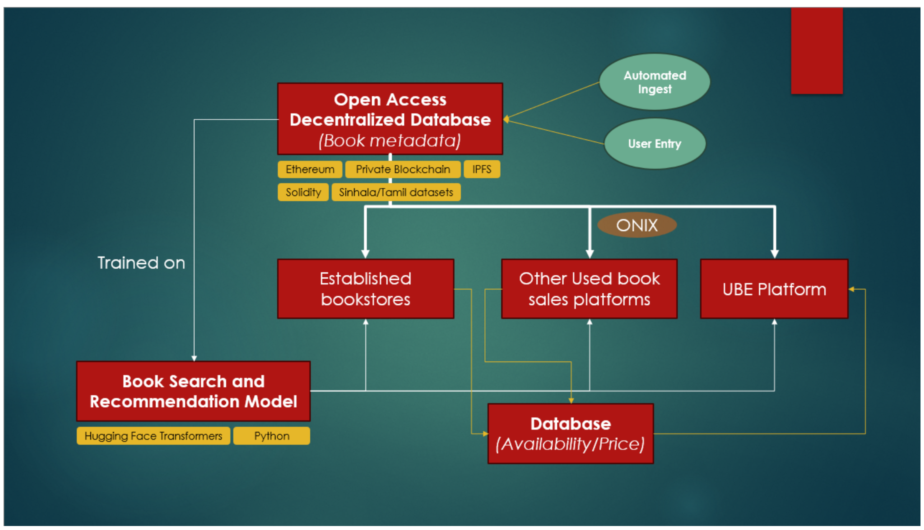

Figure 1: Block Diagram for Proposed System

As per our proposal, we identified 4 key deliverables necessary to succeed in our project, namely:

- Book Data Database
- Recommendation System APIs
- Search APIs
- Demo applications

4.1 Development

Development was split into three phases. Namely,

- API Creation
- Demo Application Creation
- Testing & Integration

4.1.1 API Creation

We chose to run our API on Flask, for it's performance and ease of use. Since we simply needed to deploy the model and have a few routes to describe the functionalities of the API, and to serve the models, we felt that a micro-framework like Flask would be the best.

The API contains the following routes.

`/predict`

`/models`

`/predict` is a HTTP POST endpoint, and is used to serve the Recommender models themselves. This accepts two parameters, `title`, and `model`. `Title` is the name of the book for which we want recommendations. `Model` gives shorthand for the various models able to be served by this API. For now, we can serve `distilbert`, `distilbert_v2`, `bert`, `tf_idf`, and `word2vec` models.

`/models` is a HTTP GET endpoint, and simply returns a list of available models that can be used by the client to request recommendations.

Recommender Models

We tried various approaches in creating the recommender models. First, we tried traditional ML techniques to create a model that can recommend books based on simiarities in content to other books. Our first model, `tf_idf`, ended up being our most stable and useful model.

During development of the models, we faced various setbacks. One pitfall we encountered was due to us using Google Colaboratory to test our models. As we exported the model, and imported the code to our API program, the dumped model (using joblib) would not run. We later figured out that the dumping has to be done within the same package as will be used in execution, and that the dumping has to be done from a seperate python file (we created a file called dump_model), else it will not create the __main__ function properly.

TF-IDF

`tf_idf` utilizes Term Frequency - Inverse Document Frequency to generate vector embeddings for the various fields used to compare books. Term Frequency-Inverse Document Frequency (TF-IDF) is a numerical statistic used in information retrieval and text mining to reflect the importance of a term in a document relative to a collection of documents (corpus). It is commonly used as a weighting factor in various text analysis tasks such as document classification, information retrieval, and text similarity calculation. In this case, we utilized TF-IDF to generate embeddings for the book titles and descriptions. This model also processes language codes, genres, and author data, but utilizes different methodologies to convert this data into embeddings. For genres, we use Multi-Label Binarization, and a hashing algorithm to compute embeddings for the author field. The libraries implementing these algorithtms were taken from `sklearn`.

For this model, we utilized Cosine Similarity as our measure of similarity, as it provided the best recommendations as per (limited) user testing. Given below is the feature extraction portion of the TF-IDF Model.

Listing 1: Feature Extraction using traditional ML

```
class FeatureExtractor:
def extract_features(self, books_df_processed):
    vectorizer = TfidfVectorizer()

    # Count of unique authors
    count_of_unique_authors = books_df_processed['
        author'].nunique()

    hasher = FeatureHasher(n_features=
        count_of_unique_authors, input_type='string')
    mlb = MultiLabelBinarizer()

    # Hash the authors
    # author_features = hasher.transform(
        books_df_subset['author'])

    # Binarize the genres column
    binarized_genres = mlb.fit_transform(
        books_df_processed['genres'])

    # One-hot encode the language_code
    books_df_subset = pd.get_dummies(
        books_df_processed, columns=['language_code'])

    # Vectorize the title column
    title_features = vectorizer.fit_transform(
        books_df_subset['title'])

    # Vectorize the description column
    description_features = vectorizer.fit_transform(
        books_df_subset['description'])

    # Composite feature Vector
```

```
    composite_feature_vector = hstack([
        binarized_genres, title_features,
        description_features])

    return composite_feature_vector
```

* Note that the Author embeddings are suppressed in this code, and it is supressed from the model because the data in the original dataset is not clean enough, which resulted in improper recommendations when using that column.

Fuzzy TF-IDF

This model is a simple modification of the TF-IDF model, which adds a fuzzy logic layer to the algorithm. This ensures that the closest match to a given book title is found using fuzzy logic, and that book title is processed in the recommender algorithm. The reason we decided to go this way is because in the event the user makes a typo, or does not know the name of the book in the exact format that is in the database (for instance, `The Smell of Death: Death #3 (2003), the user might just type The Smell of Death or The Seell of Dearth`)

Bert & DistilBert

BERT is a language model based on the transformer architecture, released in 2018 by researchers at Google. We intended to use BERT for generating embeddings for the data, as transformer based pre-trained models provide significant advantages over traditional ML methodologies for extracting richer relationships between text tokens.

DistilBert is a newer model released by Google that is 40% smaller than Bert, yet provides 95% of it's functionality. We were able to get a rudimentary model based on DistilBert running once, but without all the parameters it was ultimately useless.

While this was the idea in theory, we were unable to actually train any of the BERT based models, as the hardware available to us was not sufficient to train the model in any reasonable length of time. The best machine available at our disposal was a laptop equipped with an AMD Rzyen 9 6900HX processor, 16 Gigabytes of RAM, and a AMD Radeon 6700S GPU. BERT / DISTILIBERT is not optimized for the AMD arrchitecture, and caused massive issues during training.

Listing 2: Feature Extraction using DistilBert

```
class FeatureExtractor:
def __init__(self, model_name="distilbert-base-
    uncased"):
     self.tokenizer = DistilBertTokenizer.
         from_pretrained(model_name)
```

```
        self.model = TFDistilBertModel.from_pretrained(
            model_name)

    def extract_features(self, books_df_processed):
        document_embeddings = []
        for author, title, desc, genres in zip(
            books_df_processed['author'],
            books_df_processed['title'],
        books_df_processed['description'],
            books_df_processed['genres']):
            # Concatenate author, title, and description
            input_text = author + ' ' + title + ' ' +
                desc
            genre_text = ' '.join(genres)
            input_text = input_text + ' ' + genre_text

            # Tokenize input text
            inputs = self.tokenizer(input_text, padding=
                True, truncation=True, return_tensors="tf"
                )

            # Forward pass through BERT model
            outputs = self.model(inputs)

            # Extract embeddings
            last_hidden_states = outputs.
                last_hidden_state
            # You can choose to use the embedding of the
                [CLS] token or pool the embeddings to get
                a single vector
            pooled_embedding = tf.reduce_mean(
                last_hidden_states, axis=1)
            document_embeddings.append(pooled_embedding.
                numpy())

        # Combine document embeddings with other features
        ##language_features = pd.get_dummies(
            books_df_processed['language_code']).values
        composite_feature_vector = np.vstack([
            document_embeddings])

        return composite_feature_vector
```

Note that distilbert and distilbert_v2 are two seperate models. In v2, we are capturing the embeddings seperately for each field, and then combining them together during model formation. In the original model, we concatenate all the fields into one string, and use that to train the model. We feel that v2 should have better performance across the board, but are unable to test it due to hardware limitations.

Word2Vec

Word2Vec is another traditional ML approach to generating embeddings. We used Word2Vec on all the available fields (title, description, author, genres), and the result was not satisfactory. The resultant model produced matches that simply matched strings within the other titles, and thus did not capture any semantic meaning behind the words themselves.

We left the model in the system, for testing and evaluation purposes.

Listing 3: Feature Extraction using traditional Word2Vec

```
class FeatureExtractor:
def __init__(self):
    self.word2vec_model = None

def train_word2vec_model(self, books_df_processed):
    # Tokenize text (title and description) into
        words
    tokenized_text = [word_tokenize(title + ' ' +
        desc) for title, desc in
                        zip(books_df_processed['title'
                            ], books_df_processed['
                            description'])]

    # Train Word2Vec model
    self.word2vec_model = Word2Vec(sentences=
        tokenized_text, vector_size=100, window=5,
        min_count=1, workers=4)

def extract_features(self, books_df_processed):
    if self.word2vec_model is None:
         self.train_word2vec_model(books_df_processed)

    # Generate document embeddings using Word2Vec
    document_embeddings = []
    for title, desc in zip(books_df_processed['title'
        ], books_df_processed['description']):
         tokenized_title = word_tokenize(title)
         tokenized_desc = word_tokenize(desc)
         title_embedding = np.mean(
              [self.word2vec_model.wv.get_vector(word)
                  for word in tokenized_title if word in
                   self.word2vec_model.wv],
```

```
            axis=0)
        desc_embedding = np.mean(
            [self.word2vec_model.wv.get_vector(word)
                for word in tokenized_desc if word in
                self.word2vec_model.wv],
            axis=0)
        document_embeddings.append(title_embedding)

    # Binarize the genres column
    mlb = MultiLabelBinarizer()
    binarized_genres = mlb.fit_transform(
        books_df_processed['genres'])

    # One-hot encode the language_code
    books_df_subset = pd.get_dummies(
        books_df_processed, columns=['language_code'])

    # Combine document embeddings with other features
    composite_feature_vector = np.hstack([
        binarized_genres, document_embeddings,
    books_df_subset.drop(columns=['genres', 'title',
        'description']).values])

    return composite_feature_vector
```

4.1.2 Demo Application Creation

The demo application is needed to demonstrate the functionalities of the API, and also gather user feedback for the performance of the models. The application should be easily accessible, and not weigh down the user's device much, if at all. With these considerations in mind, we chose to create a web-app to demonstrate the model technology.

Our team was familiar with many web solution stacks. But overall, we wanted to keep the development within NodeJS, as our entire team was very well acquanite with JavaScript, having had a wonderful instructor to teach us. The next problem was the choice of framework. Multiple choices came up, with frontline choices being NextJS and VueJS. Given below are some notable differences between the two fullstack frameworks.

As the team member leading the development of the web application was more familiar with NextJS, we decided to choose NextJS. This coincidentally also lended very well into the nature of the application itself, as interactivity wasn't much of a priority with this application, and we wanted to make everything fit into a simple, mostly SSR (server side rendered, for increased performance) application.

Modularity and reusability was a primary focus in the actual code itself. As we wanted the components built in this project to be used in various live

Feature	**Next.js**	**Vue.js**
Framework	React-based framework for building server-side rendered (SSR) and static websites.	Progressive JavaScript framework for building interactive web interfaces.
Routing	Built-in routing capabilities using file-based routing and the ‘pages‘ directory structure.	Vue Router is the official routing solution for Vue.js applications.
State Management	Supports various state management solutions including React Context API, Redux, and MobX.	Supports Vuex, a state management library inspired by Flux and Redux.
Server-side Rendering (SSR)	Built-in support for SSR with server-side rendering of React components.	SSR is possible using frameworks like Nuxt.js, which is built on top of Vue.js.
Community	Active community support and ecosystem with extensive documentation and resources.	Large and vibrant community with a rich ecosystem of libraries and plugins.
Learning Curve	Requires knowledge of React and JavaScript ecosystem.	Requires knowledge of Vue.js and its ecosystem.

Table 1: Comparison between Next.js and Vue.js

projects as individual components, we felt that it was imperative that the code be clean, developer friendly, and easy to modify and extend. To this extent, we followed all the standard best practices when it comes to NextJS development. This application was built on NextJS 14, which uses a newer routing system called the App Router. There was a bit of a learning curve in understanding and the changes from older NextJS versions, but the benefits to using the newer version were enormous, especially on code organization, and on performance.

The application architecture is divvied up as follows. `page.js` contains the main SPA code, and is generated client side. NextJS by default renders everything on server side, but we had to use client side rendering in order to implement React Hooks to do state management of the data obtained from the server. We were not too concerned with performance drawbacks due to client side rendering here, as the computations performed are minimal, and deal only with computing and displaying the API data. `recommender.api.services` is the service that interacts with the API endpoint. An HTTP POST request is sent to the server, (currently hardcoded to `http://localhost:5000/predict`), and will fetch the predictions for a given book title. By default, this function requests data using the `fuzzy-tf-idf` model.

As of writing, no significant effort was made into the CSS of the web application, as we expect every actual implementation of this system to be different, and use their own individual branding. We have incorporated `react-bootstrap` to simplify the app layout, and make it look more aesthetic and user friendly for demonstration purposes.

4.2 Testing

Testing was performed mostly manually, and divided up into three parts. First, the API and models were tested independently, then, the Demo app's endpoints were tested, and finally, the API and the Demo App were tested together.

Stage 1 - API Testing

The API was tested rigorously using Postman to perform testing. A new workspace was created, and the local testing deployment on port 5000 was tested. Primarily, the behaviour of the API when as it handles expected and unexpected input data was analyzed.

Test Case ID:	TC001
Title:	Validate
Precondition:	The book dataset is loaded and the recommendation model is deployed.
Assumption:	User provides a valid book title available in the dataset.

Number of Test Cases	**Sub Test Case**	**Description and Expected Results**
1	TC001-1	Input a valid book title and expect the system to return the top 10 recommended books.
2	TC001-2	Input a book title with slight typos and expect the system to return the top 10 recommended books, handling the typo gracefully.
3	TC001-3	Input a book title not present in the dataset and expect the system to handle it with an appropriate error message.

Expected Result:	• TC001-1: The system should return a list of 10 recommended books related to the input title. • TC001-2: The system should return a list of 10 recommended books, accounting for the typo in the input. • TC001-3: The system should display an error message indicating the book title is not found.

Actual Result:	• TC001-1: The system returned the correct list of 10 recommended books. • TC001-2: The system handled the typo and returned relevant recommendations. • TC001-3: The system displayed an appropriate error message.
Actual Output:	• TC001-1: [List of 10 recommended books] • TC001-2: [List of 10 recommended books with typo handling] • TC001-3: "Error: Book title not found."
Pass/Fail:	• TC001-1: Pass • TC001-2: Pass • TC001-3: Pass

Test Case ID:	TC002
Title:	User Interaction
Precondition:	The recommendation model is deployed.
Assumption:	User selects a book within the app.

Number of Test Cases	**Sub Test Case**	**Description and Expected Results**
1	TC002-1	User selects a book and provides recommendation.
2	TC002-2	App updates the interaction matrix.

Expected Result:	• The app should give correct results. • Subsequent recommendations consider this new interaction.

Actual Result:	• The app provided correct recommendations. • Subsequent recommendations considered the new interaction.
Actual Output:	• Correct recommendations were displayed. • Interaction matrix was successfully updated.
Pass/Fail:	• TC002-1: Pass • TC002-2: Pass