Amir Ghari

NexLetter

A Newsletter Recommendation Engine

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

Sep 2025

Abstract

Author(s): First Name Last Name

Title: Title of the Thesis

Number of Pages: xx pages + x appendices

Date: 28 August 2020

Degree: Name of the degree

Degree Programme: Name of the degree programme

Specialisation option: Name of the specialisation option

Instructor(s): First name Last name, Title (e.g., Project Manager)

First name Last name, Title (e.g., Principal Lecturer)

The text section of the abstract is written to fit the space used on the page. The text section uses the **Body Text No Spacing** style.

Keywords: Keyword

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The originality of this thesis has been checked using Turnitin Originality Check service.

**Contents**

[1 Introduction 1](#_Toc204852132)

[2 Theoretical Background 1](#_Toc204852133)

[**2.1** **Recommender Systems: Overview and Importance** 1](#_Toc204852134)

[**2.2** **Types of Recommender Systems** 1](#_Toc204852135)

[2.2.1 Content-Based Filtering 2](#_Toc204852136)

[2.2.2 Collaborative Filtering 2](#_Toc204852137)

[2.2.3 Hybrid Recommender Systems 3](#_Toc204852138)

[2.2.4 The Chosen Method For This Thesis Project 3](#_Toc204852139)

[**2.3** **Evolution and Use Cases of Recommender Systems** 3](#_Toc204852140)

[**2.4** **Mathematical Techniques Behind Recommendation** 5](#_Toc204852141)

[**2.5** **NLP and Semantic Scoring in Modern Systems** 6](#_Toc204852142)

[**2.6** **Chapter Summary** 6](#_Toc204852143)

[3 Implementation 8](#_Toc204852144)

[**3.1** **System Overview** 8](#_Toc204852145)

[3.1.1 Technology Stack 8](#_Toc204852146)

[3.1.2 Rationale For Choosing Python 8](#_Toc204852147)

[3.1.3 Why PostgreSQL? 9](#_Toc204852148)

[3.1.4 Why FastAPI for the Backend? 9](#_Toc204852149)

[**3.2** **Data Collection and Storage** 9](#_Toc204852150)

[3.2.1 Article Data Collection 10](#_Toc204852151)

[3.2.2 User Interaction Logging 10](#_Toc204852152)

[3.2.3 Database Schema Design 11](#_Toc204852153)

[3.2.4 Design Considerations and Benefits 13](#_Toc204852154)

[3.2.5 Article Preprocessing and Refresh Mechanism 14](#_Toc204852155)

[3.2.6 Data Quality Measures 14](#_Toc204852156)

[**3.3** **Recommendation Algorithm Design** 15](#_Toc204852157)

[3.3.1 Development Infrastructure and Tools 15](#_Toc204852158)

[3.3.2 Scoring Formula Overview 15](#_Toc204852159)

[3.3.3 Preference Score Calculation 16](#_Toc204852160)

[3.3.4 Interaction Score Calculation 16](#_Toc204852161)

[3.3.5 NLP-Based Similarity Score 17](#_Toc204852162)

[3.3.6 Ranking and Filtering 18](#_Toc204852163)

[3.3.7 Limitations and Considerations 18](#_Toc204852164)

[**3.4** **System Architecture** 19](#_Toc204852165)

[3.4.1 API Layer 20](#_Toc204852166)

[3.4.2 Database Access Layer 20](#_Toc204852167)

[3.4.3 Database Ingestion Pipeline 20](#_Toc204852168)

[3.4.4 Recommender Logic Layer 21](#_Toc204852169)

[3.4.5 NLP Logic Layer 21](#_Toc204852170)

[**3.5** **Chapter Summary** 22](#_Toc204852171)

[4 Testing and evaluation 23](#_Toc204852172)

[4.1 How to add alternative text to images 23](#_Toc204852173)

[4.2 Subheading 24](#_Toc204852174)

[5 Conclusions 25](#_Toc204852175)

[4.3 Finish the document properties 25](#_Toc204852176)

[4.4 Check the accessibility of your thesis 25](#_Toc204852177)

[4.5 Save the Word document as an accessible PDF 26](#_Toc204852178)

[References 28](#_Toc204852179)

[Appendices 30](#_Toc204852180)

[Title of the Appendix 30](#_Toc204852181)

[Title of the Appendix 32](#_Toc204852182)

# Introduction

Write the introduction of your thesis here. Use line spacing 1.5 throughout the paper. Only the left edge is aligned, and the text is not hyphenated. Leave one blank line between paragraphs (press enter once).

Begin a new paragraph at the left margin, that is, do not indent the first line.

# Theoretical Background

## **Recommender Systems: Overview and Importance**

Recommender systems have become integral components of modern digital platforms, enabling personalized content delivery across domains such as e-commerce, news, entertainment, and education. Their primary purpose is to filter and rank large volumes of data in a way that aligns with individual user preferences, thereby improving user engagement and satisfaction.

At a high level, a recommender system analyzes user data (e.g., past interactions, preferences) and item characteristics (e.g., categories, keywords, metadata) to generate ranked lists of suggestions. These systems reduce information overload and help users discover relevant items more efficiently.

In this thesis, we focus on building a **newsletter and article recommender system** that leverages user preferences, reading behavior, and text similarity techniques to deliver more personalized news content.

## **Types of Recommender Systems**

Recommender systems are generally categorized based on the kind of data they use and how they process this data to generate recommendations. The three primary types are **content-based filtering**, **collaborative filtering**, and **hybrid recommender systems**. Each method has distinct strengths and limitations, and they are often combined to maximize recommendation accuracy and personalization.

### Content-Based Filtering

Content-based filtering relies on the properties or features of items to recommend similar ones to a user. For instance, if a user has previously engaged with articles categorized as "technology" or from the "United States," a content-based system will suggest new articles with similar tags. This approach typically represents items and user profiles using keywords, tags, or metadata and applies techniques like term frequency or cosine similarity for matching.

The main advantage of content-based filtering is that it does not require data from other users. However, it can lead to a phenomenon known as **over-specialization**, where users are only recommended items similar to what they’ve already seen, reducing diversity.

### Collaborative Filtering

Collaborative filtering focuses on user behavior rather than item attributes. It assumes that if two users have reacted similarly to the same set of items, they will continue to do so. There are two subtypes:

* **User-based collaborative filtering**: recommends items that similar users have liked.
* **Item-based collaborative filtering**: recommends items that have received similar user interactions.

This method works well when a large amount of user interaction data is available. However, it suffers from the **cold-start problem** (when new users or items have insufficient data) and **sparsity** (when user interactions are too few or scattered).

### Hybrid Recommender Systems

Hybrid recommender systems combine two or more recommendation strategies to benefit from their complementary strengths. For instance, a system can merge content-based filtering with collaborative filtering or integrate behavioral signals (like time spent on an article) with preference-based models.

Hybrid systems are often more accurate and robust, addressing many of the limitations associated with standalone approaches. For example, they can help mitigate the cold-start problem by using content features and user preferences until enough behavioral data is collected.

### The Chosen Method For This Thesis Project

The recommender system developed in this project follows a **hybrid approach**. It blends:

* User-defined **preferences** (preferred countries and categories)
* User **interaction signals** (liked articles and time spent)
* **NLP-based similarity** scoring (based on cosine similarity between liked article titles and new titles)

This layered structure aims to maximize personalization while adapting to evolving user behavior. By incorporating both explicit preferences and implicit behavioral signals, the system produces more relevant and contextually aware article suggestions.

## **Evolution and Use Cases of Recommender Systems**

Since their early use in information retrieval and online retail in the late 1990s, recommender systems have evolved to become central components of the digital user experience. Their development has closely mirrored advances in machine learning, data availability, and user modeling. Initially based on simple heuristics and keyword matching, modern recommenders now incorporate sophisticated algorithms capable of dynamic adaptation and personalization [1].

Recommender systems are now deployed across a broad range of industries and digital products:

* **E-commerce**: Platforms like Amazon use recommendation systems to suggest products based on browsing history, purchase patterns, and similar customer behavior [2].
* **Media and Entertainment**: Streaming services such as Netflix and Spotify leverage user preferences, time-based consumption, and deep content metadata to tailor viewing and listening experiences [3].
* **Social Media**: Platforms like Facebook, X (formerly Twitter), and TikTok utilize recommender systems to prioritize content in user feeds, drawing on interaction metrics such as likes, shares, and watch time [4].
* **News Aggregation**: Personalized news platforms recommend articles based on user interests, regional preferences, and reading history.
* **Education**: E-learning environments use recommender systems to suggest courses, study materials, and learning paths based on user goals and engagement [5].

A striking indicator of the importance of recommender systems is their impact on business outcomes in large technology companies. For instance:

* **Netflix** has publicly stated that more than 80% of the content watched on its platform is driven by recommendations. Their personalized home screen and row-based organization are entirely curated by algorithmic suggestions [3].
* **YouTube** reports that over 70% of user watch time comes from recommended videos. These recommendations are powered by complex deep learning models that factor in watch history, user preferences, device type, and more [6].
* **Amazon** generates a significant portion of its revenue from recommended products shown through “Customers who bought this also bought” and other dynamic modules [2].

These examples underscore the transformative role that personalized recommendation has played in enhancing user engagement, retention, and monetization, turning recommender systems into strategic assets rather than optional features.

## **Mathematical Techniques Behind Recommendation**

Modern recommender systems rely on a range of mathematical foundations to process user behavior and item data, generate profiles, and rank suggestions. Some of the most common techniques include:

* **Vector Space Models**: Items and user profiles are represented as vectors in a high-dimensional space, allowing similarity to be measured using distance or angle metrics (e.g., cosine similarity) [7].
* **Matrix Factorization**: Often used in collaborative filtering, this method decomposes a user-item interaction matrix into latent factors that represent underlying user preferences and item characteristics [8].
* **TF-IDF (Term Frequency–Inverse Document Frequency)**: Widely used in content-based filtering and natural language processing, TF-IDF assigns weights to words in a document relative to their frequency across a corpus, helping capture the importance of keywords in item descriptions or titles [9].
* **Clustering and Classification**: Unsupervised and supervised learning methods (e.g., k-means, decision trees) are used to group similar users or items and to predict preferences.
* **Regression and Ranking Models**: Algorithms such as logistic regression or learning-to-rank models are used to estimate the likelihood of a user engaging with an item and to order recommendations accordingly [10].

These mathematical tools, often embedded within broader algorithmic strategies, allow recommender systems to scale to millions of users and items while maintaining efficiency and relevance.

## **NLP and Semantic Scoring in Modern Systems**

Natural Language Processing (NLP) has added a new layer of semantic understanding to recommender systems. Traditional systems often rely on categorical metadata or interaction patterns, but NLP enables deeper insight into textual data such as article titles, product descriptions, or user-generated content [11].

Techniques such as TF-IDF vectorization and cosine similarity help quantify textual similarity between items, making it possible to recommend items based not only on explicit categories but also on the semantic overlap of their titles or content [9].

More advanced models, such as word embeddings (e.g., Word2Vec, GloVe) or transformer-based models (e.g., BERT), allow for richer representations of text by capturing context and nuanced meanings [12]. These models are increasingly used in real-time recommendation engines to match user interests with newly published content.

In the context of this project, NLP-based similarity scoring is used to match the titles of articles or newsletters a user has previously liked with the titles of new items. This enables the system to capture subtle, thematic consistencies beyond just category or location tags, improving recommendation quality—especially for new or niche content that might otherwise be overlooked.

## **Chapter Summary**

This chapter explored the theoretical underpinnings of recommender systems, starting from their types and evolution to the mathematical and linguistic foundations that drive their decision-making. We examined real-world business use cases demonstrating their significance, and how modern recommendation engines combine explicit user data with advanced NLP techniques for contextual awareness.

In the next chapter, we shift from theory to practice by examining the implementation of our hybrid recommender system. This includes the design of the backend architecture, database schema, recommendation pipeline, and scoring mechanisms that bring the system to life.

# Implementation

## **System Overview**

This chapter describes the architecture and inner workings of the implemented newsletter recommendation system, with a focus on how different modules collaborate to process user data and generate personalized article suggestions. The design was guided by a strong emphasis on modularity, performance, and ease of future integration.

### Technology Stack

The system leverages a modern and efficient technology stack consisting of:

* **Backend Language**: Python 3.12
* **Database**: PostgreSQL
* **Web Framework**: FastAPI
* **Data Handling and NLP**: scikit-learn, NumPy, psycopg2
* **Testing Framework**: pytest
* **API Testing and Documentation**: FastAPI Swagger UI
* **Containerization (Planned for Deployment)**: Docker
* **Frontend (Planned)**: Next js, React.js

This selection of tools supports rapid development, strong community support, and performance suitability for recommendation applications.

### Rationale For Choosing Python

Python is widely adopted in data-driven projects and is particularly favored for its rich ecosystem of machine learning, natural language processing (NLP), and data manipulation libraries. Its clean syntax and expressive design also make it easier to implement complex algorithms and experiment with different recommendation strategies. The use of libraries like scikit-learn allows for seamless integration of techniques such as TF-IDF and cosine similarity, which are central to our scoring approach [9][7].

### Why PostgreSQL?

PostgreSQL was selected for its robustness, support for complex queries, and native handling of JSON and array types. It is highly suitable for storing semi-structured metadata such as article categories and multilingual content. Its indexing capabilities and ACID compliance make it an ideal choice for managing user preferences, interactions, and newsletter data at scale [11].

### Why FastAPI for the Backend?

FastAPI was chosen over traditional frameworks like Flask or Django for several reasons:

* **Performance**: Built on Starlette and Pydantic, FastAPI offers high performance comparable to Node.js or Go.
* **Type Safety**: It uses Python type hints to enforce request/response models, minimizing runtime errors and improving code maintainability.
* **Auto-Generated Documentation**: FastAPI automatically generates Swagger and ReDoc-based API documentation, streamlining testing and third-party integration.
* **Asynchronous Capabilities**: As the system grows, asynchronous endpoints will allow it to scale more efficiently without major architectural overhauls.

## **Data Collection and Storage**

The foundation of this recommender system lies in structured data collection, focusing on both user interactions and up-to-date news content. To power personalized article suggestions, the system collects and stores news articles via a third-party API and logs key behavioral signals from user activity. PostgreSQL serves as the core data storage solution, ensuring reliability and performance for structured queries.

### Article Data Collection

The article dataset used in this project is retrieved via the [Newsdata.io](https://newsdata.io) API [13], which provides structured JSON feeds of news content aggregated from a wide variety of global and regional sources. Newsdata.io offers categorized news across countries and topics, making it a reliable source for building geographically and topically personalized feeds.

A Python script using the requests library is responsible for fetching the data. The API is queried every 24 hours using a simple scheduler to ensure a continuously updated pool of articles. Fields received include the article’s title, content snippet, description, publication date, link, source, country, language, category, and image URL. This raw data is directly inserted into the PostgreSQL database, and no additional text normalization or HTML cleaning is performed, as the API returns already structured and readable content [13].

### User Interaction Logging

To personalize recommendations beyond explicit preferences, the system captures behavioral signals based on user interaction. These include:

* **Liked articles** (manually marked by the user),
* **Time spent on each article** (automatically tracked).

This dual-signal approach mirrors methodologies used by large-scale platforms like Netflix and YouTube, where a mix of implicit and explicit feedback strengthens the accuracy of hybrid recommendation models [3][4][6].

Interaction data is logged in a dedicated interactions table, which maintains relationships between users and articles. This schema enables the system to compute scores reflecting user engagement — a key component in behavioral and adaptive recommender models [1][11].

### Database Schema Design

The database schema was designed with simplicity and scalability in mind. The core structure revolves around the articles, users, and interactions tables. The articles table captures key metadata needed for recommendation, such as categories (stored as arrays), language, and publication timestamps, which enable efficient filtering and scoring operations.

Particular attention was given to data types and indexing strategies. For instance, array types allow efficient storage and querying of multi-label categories, while TIMESTAMP fields facilitate time-based ranking and freshness checks. The schema avoids unnecessary joins by denormalizing where appropriate — for example, storing source and country directly with each article record — which helps reduce query complexity and speeds up real-time recommendation tasks.

This schema serves as the backbone for both recommendation logic and analytics, ensuring fast retrieval and flexible experimentation as new features are added.

The core schema is defined through four tables, structured as follows:

**Articles Table**

CREATE TABLE IF NOT EXISTS articles (

id SERIAL PRIMARY KEY,

title TEXT,

content TEXT,

link TEXT,

pub\_date TIMESTAMP,

source TEXT,

description TEXT, country TEXT,

category TEXT[], language TEXT, image\_url TEXT

);

The articles table stores the metadata fetched from Newsdata.io. Fields like category, country, and pub\_date enable content filtering, freshness ranking, and location-aware recommendations.

**Users Table**

CREATE TABLE IF NOT EXISTS users (

id SERIAL PRIMARY KEY,

email TEXT UNIQUE,

preferred\_countries TEXT[],

preferred\_categories TEXT[]

);

The users table captures user-defined preferences for countries and categories. These fields are used as filters and weights in the recommender’s scoring logic.

**Interaction Table**

CREATE TABLE IF NOT EXISTS interactions (

id SERIAL PRIMARY KEY,

user\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

article\_id INTEGER REFERENCES articles(id) ON DELETE CASCADE,

liked BOOLEAN,

time\_spent\_seconds INTEGER,

timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

The interactions table logs behavioral data. The liked field records explicit feedback, while time\_spent\_seconds provides implicit engagement. This dual-signal structure is inspired by large-scale hybrid systems like Netflix and YouTube [3][4].

**Liked\_Titles Table**

CREATE TABLE IF NOT EXISTS liked\_titles (

id SERIAL PRIMARY KEY,

user\_id INTEGER REFERENCES users(id) ON DELETE CASCADE,

title TEXT

);

The liked\_titles table is dedicated to storing the titles of articles each user has explicitly liked. These titles are later compared to new article titles using TF-IDF and cosine similarity for personalized recommendations. This separation makes the NLP pipeline more modular and keeps the interactions table lean.

### Design Considerations and Benefits

This schema is optimized for:

* **Simplicity**: Tables are normalized, but not overcomplicated. Core data entities are separated for clarity.
* **Scalability**: Indexed fields and well-defined relationships support fast querying even as data volume grows.
* **Modularity**: Storing liked titles separately allows the NLP module to function independently of the rest of the app logic.
* **Hybrid Scoring**: Combining preferred\_\*, interactions, and liked\_titles enables personalized, multi-signal recommendations.

Together, these tables serve as the foundation for both real-time recommendations and analytics, while allowing future extensions like comment tracking, multilingual filtering, or advanced behavioral modeling.

### Article Preprocessing and Refresh Mechanism

To maintain content freshness and relevance, the system includes a scheduled update pipeline that fetches new articles once every 24 hours from Newsdata.io [13]. This process is executed through a script (main\_fetch.py) tied to a cron-based job runner.

Each fetch cycle performs the following steps:

* Queries predefined countries and categories from the API,
* Checks for duplication based on article title and pub\_date,
* Inserts new articles into the database if they are not already present.

No HTML parsing or text cleaning is performed; the API returns clean, structured fields. The refresh process ensures that users receive recent and relevant content tailored to their interests and geographic region.

### Data Quality Measures

Although minimal preprocessing is required due to the structure of Newsdata.io’s responses, basic validation steps are applied to ensure consistency:

* Duplicate articles are avoided by checking for collisions in title and publication date,
* Country and category fields are stored in lowercase to support uniform filtering,
* Articles in unsupported languages are either tagged or omitted based on user language preferences.

These measures, although lightweight, help maintain data quality and ensure the effectiveness of the scoring algorithms used in the recommender engine [9][11].

## **Recommendation Algorithm Design**

The recommendation logic in this project follows a **hybrid approach** that combines explicit user preferences, interaction-based scoring, and NLP-based similarity analysis. This section outlines the design choices made during the development of the algorithm, focusing on the scoring mechanism and how recommendations are ranked in real time.

### Development Infrastructure and Tools

The goal of the system is to offer a lightweight, interpretable recommendation mechanism that is effective without the need for large-scale collaborative filtering models or opaque machine learning pipelines. Given the limited user base and interaction volume during early phases, matrix factorization or deep learning-based recommenders are avoided in favor of rule-based and statistically grounded heuristics. This decision was influenced by practices in cold-start or sparsity-constrained environments, as discussed in [1][8][11].

### Scoring Formula Overview

Each article receives a **composite score** based on the following key inputs:

* **User Preferences (countries and categories)**
* **User Interactions (liked articles, time spent)**
* **NLP-based Similarity (title comparison)**

These inputs are combined using weighted scoring:

Weights are currently set through manual tuning (e.g., w1 = 0.4, w2 = 0.3, w3 = 0.3) based on experimental feedback. The structure allows easy adjustment in future iterations.

### Preference Score Calculation

Users can select preferred **countries** and **categories** during onboarding. Each article is tagged with metadata from Newsdata.io [13], including these fields.

preference\_score = 1.0 if country/category matches, else 0.0

Multiple category matches (e.g., “Technology”, “AI”) increase the score proportionally. No match results in a zero contribution. This method aligns with typical filtering strategies in content-based recommenders [1].

### Interaction Score Calculation

The system tracks two user behaviors:

* Articles **explicitly liked** (binary signal)
* **Time spent** on articles (implicit engagement)

The score is computed based on how frequently a user interacts with content from specific countries or categories.

Components:

* **Time Bonus**:
  + +5 if time spent > 15 minutes
  + +2 if time spent > 10 minutes
* **Preferred Category Boost**: +5 if the article matches any of the user’s preferred categories.
* **Preferred Country Boost**: +5 if the article matches the user’s preferred country.
* **Liked Category Boost**: Based on how frequently a user has liked articles from a specific category (added as an integer).
* **Liked Country Boost**: Similarly, the more a user has liked content from a country, the higher the boost.

These values are stored and retrieved from the interactions table and the computed user profile, which aggregates behavior such as liked categories and reading durations. During each scoring round, this logic is applied to every article, ensuring recommendations reflect both long-term preferences and recent behavioral trends. This lightweight approach is suitable for systems with limited user activity or in early-phase deployments, and aligns with strategies seen in sparse-data environments [1][11].

### NLP-Based Similarity Score

To enhance semantic understanding in recommendations, the system uses **TF-IDF** (Term Frequency–Inverse Document Frequency) to convert article titles into numerical vectors. TF-IDF assigns higher weights to terms that are frequent in a specific document but rare across the overall corpus, emphasizing words that best represent a title’s unique content [7][9].

Once vectorized, the titles of previously liked articles are compared to candidate article titles using **cosine similarity**, which measures the angle between two vectors in multi-dimensional space. Unlike raw keyword overlap, cosine similarity accounts for context and direction, making it suitable for identifying semantically similar content regardless of length or word count [9][12].

The NLP similarity score is computed as:

similarity\_score=max(cosine\_similarity(liked\_title,new\_article\_title))

The score reflects how semantically related new articles are to the user’s reading history. Unlike category tags, this captures more subtle content affinities, especially useful for under-tagged or niche articles [9][12].

### Ranking and Filtering

After calculating the three component scores — interaction score, preference match boosts, and NLP-based similarity — the system ranks all available articles to generate personalized recommendations.

The recommend\_articles() function performs the following steps:

* Retrieves the user’s profile and historical interaction data.
* Scores each article based on combined heuristics.
* **Filters out articles the user has already seen**, ensuring novelty in the feed.
* Sorts the remaining articles by descending total score.
* Returns the top N results (e.g., N = 10) for recommendation.

This approach ensures that recommendations are not only **timely and relevant**, but also **interpretable and easily tunable**. Unlike black-box models, the logic behind each recommendation can be traced and adjusted without the need for retraining. This makes it particularly suitable for low-data environments or early-stage deployments.

### Limitations and Considerations

While the implemented recommendation system provides lightweight and interpretable personalization, it has several current limitations. Notably, collaborative filtering methods have been intentionally excluded due to the sparse user base and the absence of shared interaction patterns across users, a challenge often encountered in cold-start environments [1][2]. Additionally, the NLP component focuses solely on article titles, rather than full-text content, in order to keep processing times low and avoid unnecessary computational overhead. This limits the semantic depth of similarity comparisons, particularly for articles with ambiguous or generic titles. Furthermore, recommendation scores are computed in real time at the moment of each user request. While this approach simplifies the architecture and keeps recommendations fresh, it may become inefficient as the dataset grows. To address this, future iterations could incorporate caching mechanisms or scheduled precomputation pipelines to balance performance and scalability.

## **System Architecture**

The newsletter recommendation system is implemented using a modular Python-based architecture with clear separation of concerns across different components. The backend is structured into several directories, each handling a specific responsibility: database access, data ingestion, recommendation logic, and NLP similarity scoring.

The high-level directory structure is shown in **Figure 1**. This modular organization supports maintainability and extensibility, allowing future integration of additional features such as collaborative filtering, analytics, or user feedback loops.

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**Figure 1.** Project directory structure of the Nexletter backend application.

### API Layer

The api/ directory contains the FastAPI entry point (main.py), route definitions (routes.py), and data models (models.py). This layer exposes RESTful endpoints for interacting with the application and serves as the interface for user actions such as liking articles or retrieving recommendations.

### Database Access Layer

Located in the db/ directory, this layer handles all direct interactions with the PostgreSQL database. The directory includes repository modules such as article\_repo.py, user\_repo.py, and interaction\_repo.py, each of which encapsulates raw SQL queries to fetch or update data. The connection.py module manages database connectivity, and init\_db.py is responsible for schema creation. This separation of responsibilities ensures clean access patterns and better testability.

### Database Ingestion Pipeline

The data ingestion logic resides in the fetcher/ directory, comprising main\_fetch.py, newsdata\_client.py, and save\_articles.py. Together, these scripts automate the retrieval of fresh article data from Newsdata.io [13] and save it into the PostgreSQL database. A scheduled job using cron now triggers main\_fetch.py once every 24 hours, enabling regular updates with newly published content. This ensures that the recommendation engine operates on up-to-date articles, improving the relevance and freshness of suggestions.

A diagram of a diagram

AI-generated content may be incorrect.

**Figure 2.** Daily Article Ingestion pipeline

### Recommender Logic Layer

The core logic for generating article recommendations is implemented in the recommender/ directory. This layer includes:

* recommender.py: Orchestrates the full recommendation pipeline, pulling user profiles, fetching candidate articles, calculating scores, and returning top results.
* scorer.py: Contains the calculate\_score() function, which computes an article’s relevance based on interaction behavior and content similarity.
* utils.py: Includes helper functions like country normalization and category checks that improve data consistency.

This modular separation enables focused development, easy testing, and future expandability (e.g., adding more heuristics or hybrid logic). The recommender layer is tightly coupled with both the db/ and nlp/ directories, making it the integrative core of the system.

### NLP Logic Layer

The nlp/ directory introduces semantic enrichment to the recommendation process by computing the similarity between titles of articles a user has liked and those in the current pool. It includes:

* liked\_title\_repo.py: Retrieves all titles the user has previously liked.
* similarity.py: Uses TF-IDF and cosine similarity to compare titles and assign a semantic score.

This score supplements the behavioral scoring pipeline and enables more nuanced recommendations — especially valuable when categorical metadata is missing or overly generic. While lightweight, the NLP layer plays a key role in diversifying and improving the relevance of article suggestions.

## **Chapter Summary**

This chapter detailed the architectural foundation and key implementation choices of the newsletter recommendation system. From the use of PostgreSQL and FastAPI to the design of the scoring algorithm that combines user preferences, behavioral interactions, and NLP-based similarity, each component was designed with modularity and scalability in mind. Article ingestion is automated through scheduled API calls, while user interactions are persistently logged to refine recommendations over time. The system relies on straightforward and interpretable logic, balancing performance with explainability.

Although the platform is still under active development, the core pipeline is in place, and early-stage functionality has been successfully validated. Collaborative filtering, frontend integration, and containerized deployment via Docker are among the planned future extensions.

A diagram of a user profile builder

AI-generated content may be incorrect.

The diagram in **Figure 3** below summarizes the full flow of data and logic across the system — from article ingestion and storage to scoring and final recommendation generation. This marks the conclusion of the implementation chapter and sets the stage for the next phase: **testing** and evaluating the robustness and correctness of each system component.

# Testing and evaluation

According to accessibility requirements, images must have alternative text. Alternative text is not the same thing as a caption. Alternative text is a description of the content of an image read aloud by screen readers used by the visually impaired. It is not advisable to repeat the caption in the alternative text because screen readers read both contents.

When writing alternative text, think about what information you will not receive if you do not see the image. Use short sentences and plain language. Tell the essential about the picture - you don't have to explain everything.

Alt text can be blank for a decorative image. Decorative images are images that do not convey any information or that have been added for layout.



Figure 2. Long-tailed jaeger is common in Finnmark's mountain plateau in northern Norway. Its main wintering site is in the South Atlantic west of Africa.

## How to add alternative text to images

An alternative text is given to an image in a Word document as follows (Office 365 version):

1. Move the cursor over the image and right-click.
2. Select “Edit Alt Text…” to open the Alternative Text window.
3. Write a brief explanation of the essential content of the image.
4. If the image is purely decorative or the caption contains all the relevant information, mark the image as decorative.

Graphical user interface, text, application

Description automatically generated

The Microsoft Office 2016 version of Word is slightly different:

1. Move the cursor over the image and right-click.
2. Select “Format Picture…”
3. In the “Format Picture” window, select the third icon “Layout and Properties”.
4. Select “Alt Text” and enter a description of the image content in “Description”. Do not write anything under “Title”.

## Subheading

There must always be text or a new subheading below each heading.

Graphical user interface, text, application

Description automatically generated

# 5 Conclusions

## Finish the document properties

Once the content of your thesis is in order, finalise the document by specifying its properties. It is essential to ensure that the pdf file is accessible when you convert a Word file to PDF format. Type a title for the document in the File menu, under Info. Enter the title of your thesis as the title.

Graphical user interface, text, application

Description automatically generated

## Check the accessibility of your thesis

Word has a feature that lets you check the accessibility of a document.

1. On the File menu, click Info.
2. Then click Check for Issues.
3. Click Check Accessibility.

Graphical user interface, application, Word

Description automatically generated

The Accessibility Check window will then appear on the right side of the Word. The results of the scan show possible errors and warnings. For more information about results, click the item name in the results list. Word also tells you the reason for the repair, as well as give repair instructions. At least fix any errors.

## Save the Word document as an accessible PDF

Once you have checked your thesis for accessibility, convert it into an accessible PDF document.

1. Create a PDF file using either the Export function (Create PDF) or the Save As function.
2. In the save options, select **Document properties** and **Document structure tags for accessibility**.
3. Click Create bookmarks using Headings.

Graphical user interface, text, application, email

Description automatically generated

Do not use the Print to PDF function because the result is not accessible PDF.

References

1. Ricci, F., Rokach, L., & Shapira, B. (2015). *Recommender Systems Handbook*. Springer.
2. Linden, G., Smith, B., & York, J. (2003). Amazon.com recommendations: Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1), 76–80. <https://doi.org/10.1109/MIC.2003.1167344>
3. Gómez-Uribe, C. A., & Hunt, N. (2016). The Netflix Recommender System: Algorithms, Business Value, and Innovation. *ACM Transactions on Management Information Systems*, 6(4), 13. <https://doi.org/10.1145/2843948>
4. Covington, P., Adams, J., & Sargin, E. (2016). Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys ’16)*. <https://dl.acm.org/doi/10.1145/2959100.2959190>
5. Manouselis, N., Drachsler, H., Verbert, K., & Duval, E. (2012). *Recommender Systems for Learning*. Springer. <https://doi.org/10.1007/978-1-4614-4361-2>
6. YouTube Official Blog. (2018). *How YouTube’s Recommendations Work*. Retrieved from <https://blog.youtube/news-and-events/how-youtubes-recommendations-work/>
7. Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11), 613–620. <https://doi.org/10.1145/361219.361220>
8. Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8), 30–37. <https://doi.org/10.1109/MC.2009.263>
9. Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press. <https://nlp.stanford.edu/IR-book/>
10. Burges, C. J. C. (2010). From RankNet to LambdaRank to LambdaMART: An Overview. *Microsoft Research Technical Report*. <https://www.microsoft.com/en-us/research/publication/from-ranknet-to-lambdarank-to-lambdamart-an-overview/>
11. Beel, J., Gipp, B., Langer, S., & Breitinger, C. (2016). Research-paper recommender systems: A literature survey. *International Journal on Digital Libraries*, 17(4), 305–338. <https://doi.org/10.1007/s00799-015-0156-0>
12. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL-HLT*. <https://arxiv.org/abs/1810.04805>
13. Newsdata.io. (2024). *Free News API for Live News Data Feeds*. Retrieved from [https://newsdata.io](https://newsdata.io/)

Appendices

Title of the Appendix

The content of the appendix is ​​written here. Below are instructions for adding and removing attachments so that the headers remain the correct.

1. Instructions for adding a new attachment:
2. Move the cursor to the end of the last existing attachment page.
3. Choose the Page Layout tab. From the Page Break ribbon select Next Page under Section Breaks. This completes the printing of the new attachment, but the number in its header is not correct.
4. Double tap the header of the new attachment page with the wrong attachment number. If the “Link to previous” option is selected in the ribbon, press that button so that the option is no longer selected.
5. Please correct the attachment number.

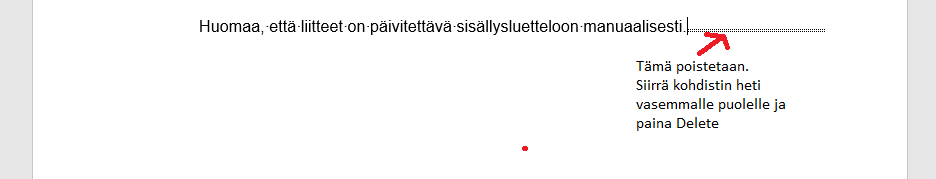
Instructions for removing an unnecessary attachment:

1. First select the entire attached page and press Delete to delete its contents.
2. When you are at the beginning of the attachment page you have emptied (see figure), double tap the header of the blank attachment page and press the Link to Previous button on the ribbon. The following dialogue box appears:



Answer Yes.

1. From the Home tab, toggle hidden characters if they are not visible:Piilomerkki painikkeen kuvake.
2. Remove the section break before the unnecessary attachment (see figure below).



Title of the Appendix

Content of the appendix is placed here