

# **CourseCompass: An Integrated Data Pipeline and Web Application for Academic Course Recommendation**

Gabriel Bangoura, Jerome Hsing, Bhuman Sharma

McKelvey School of Engineering

## Abstract

This report details the design, development, and deployment of an end-to-end data pipeline and interactive web application aimed at enhancing course selection for university students. The project addresses the limitations of conventional Rate My Professors (RMP) platforms by integrating multiple data sources, including PDF files and online professor reviews, through an automated Apache Airflow pipeline. Data is cleaned, enriched with sentiment scores, transformed, and stored in a Snowflake data warehouse. A Flask API and a Streamlit-based web application provide real-time course recommendations built on these data. The report outlines each component of the project, describes the methodology and implementation details, and discusses the implications of our findings on improving student decision-making in course enrollment.

## Introduction

University students face significant challenges when selecting courses for upcoming semesters. Critical information required for informed decision-making is typically scattered across multiple platforms and document formats, creating inefficiencies in the course selection process. This project aims to develop a comprehensive course recommendation system that consolidates disparate data sources, applies analytical techniques to student feedback, and delivers personalized course suggestions based on individual preferences and historical performance metrics.

Our CourseCompass system addresses the limitations of existing platforms by integrating professor ratings from RateMyProfessor (RMP), official course listings, and institutional course evaluations into a unified database. By developing custom data collection methods and implementing an automated pipeline for data processing, we create a foundation for delivering reliable, data-driven course recommendations that enhance student academic planning and satisfaction. Our solution uses a combination of Python-based data processing, Snowflake for scalable data storage, Apache Airflow for pipeline orchestration, and modern web frameworks (Flask and Streamlit) for user interaction.

# **Problem Statement**

The current landscape of course selection tools present several critical challenges for university students. Essential course and professor data exists across multiple disconnected platforms, including RateMyProfessor, university course catalogs, and institutional course evaluations. Students must navigate these separate systems independently to gather comprehensive information. Also, while RateMyProfessor is frequently used for professor evaluations, it often contains incomplete data, particularly for newer professors or specialized courses. Conversely, official university course evaluations contain valuable feedback but are typically stored in structured PDF documents that are difficult to access and analyze at scale. Importantly, no existing system effectively combines quantitative metrics (e.g., professor ratings, course difficulty) with qualitative feedback from multiple sources to provide holistic course recommendations. As a result, these issues create difficulties when students are attempting to navigate course selection.

# **System Architecture Overview**

CourseCompass employs a modular architecture that separates concerns across different components while maintaining a cohesive data flow to compensate for the various data sources that we are drawing from to address these issues that students face. Figure 1 illustrates the system architecture, comprising data collection, data processing, data storage, and user interface layers.

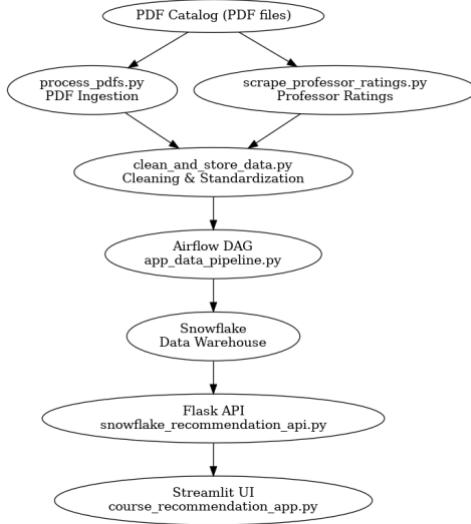


Figure 1: CourseCompass System Architecture

## Data Collection Methodology

To build our comprehensive course recommendation system, we implemented three specialized data collection components designed to handle different source formats and structures.

### Course Information Scraper

The first component of our data collection pipeline extracts course listing information from the university's website. This provides essential baseline data about course offerings, including course codes, titles, and instructor assignments. We developed a Python-based web scraper using BeautifulSoup that systematically extracts course information from the university's departmental pages, which navigates the structure of the university's course listing pages, extracting relevant information including course codes, names, and instructor assignments. The implementation includes robust error handling and alternative extraction paths to account for inconsistencies in the HTML structure.

The scraper specifically handles cases where multiple instructors are listed for a single course by creating separate entries for each instructor-course combination, facilitating later joining with professor rating data.

## **RateMyProfessor Data Scraper**

The second component collects professor ratings and related data from RateMyProfessor, which provides critical qualitative assessment information about instructors. Since RateMyProfessor does not offer a public API, we implemented a custom scraper that extracts professor ratings and review metrics, as well as manually retrieving professors' IDs from the website.

A significant challenge in collecting data from RateMyProfessor is its implementation of anti-scraping protections. To address this, we incorporated several techniques:

1. Browser User-Agent Headers: The scraper uses browser headers to mimic legitimate user requests.
2. Request Rate Limiting: We implemented random delays between requests to avoid triggering rate-limiting mechanisms.
3. Error Handling and Resilience: The implementation includes comprehensive error handling to manage failures gracefully and continue the overall process even if there are individual request failures.

## **For each professor, the scraper extracts several key metrics:**

- Overall quality rating (scale 1-5)
- "Would take again" percentage
- Difficulty rating (scale 1-5)
- Associated tags (e.g., "Clear grading criteria", "Accessible outside class")

## **PDF Processing Module**

The third and most technically challenging component of our data collection system extracts structured information from course evaluation PDFs. These documents contain valuable feedback from previous students but are not readily accessible in a machine-readable format.

### **The extraction process follows these steps:**

1. Load PDF document using pdfplumber
2. Extract text content from each page
3. Apply regular expressions to identify course-related patterns
4. Structure the extracted information into a standardized format
5. Validate and clean the extracted data

The module employs a set of carefully crafted regular expressions to handle variations in document formatting and structure. For example, course codes are identified using patterns that match common formats (e.g., “CS 131”, “SDS 3211”).

Several challenges complicated the PDF extraction process. First, course evaluation PDFs exhibited variable layouts and structures, requiring a flexible extraction approach. Second, important information was often embedded within complex text layouts, necessitating sophisticated regular expression patterns for reliable extraction. Finally, the extraction process needed robust error handling to manage cases where expected patterns were not found, or PDF parsing encountered issues.

To address these challenges, we implemented a multi-pattern matching approach that tried several different regular expressions to locate the same information, increasing the likelihood of successful extraction across varying PDF formats.

## **Data Processing**

## **Data Cleaning and Transformation**

The data cleaning and transformation module standardizes data formats, handles missing values, and resolves inconsistencies across different data sources. Key operations include:

- Standardizing course code formats
- Normalizing professor names and department designations
- Converting text-based ratings to numerical values
- Handling missing or incomplete information
- Resolving conflicts between different data sources

The module employs regular expressions extensively for pattern matching and text transformation. For example, professor names are standardized by extracting first and last name components and applying consistent formatting rules.

## **Workflow Orchestration**

Apache Airflow orchestrates the entire data pipeline, managing dependencies between tasks and ensuring reliable execution. The workflow includes:

1. Scheduled extraction of PDF documents
2. Periodic scraping of professor ratings
3. Data cleaning and transformation processes
4. Loading data into the Snowflake data warehouse
5. Generating derived metrics and aggregations
6. Validating data quality and completeness

The Airflow DAG (Directed Acyclic Graph) defines task dependencies and execution schedules, providing visibility into pipeline operations and facilitating troubleshooting when issues arise.



Figure 2: Airflow DAG diagram

## Data Storage

CourseCompass uses Snowflake as its primary data warehouse, leveraging its scalability, performance, and separation of storage and compute resources. Snowflake's unique architecture and capabilities provide several key benefits for managing large-scale educational data.

### Snowflake Usage in CourseCompass

Snowflake's advanced data warehousing capabilities are integral to the data storage and processing strategy in CourseCompass. Here's how we use Snowflake:

### Centralized Data Storage

All course-related data, including structured data from PDFs and enriched data from web scraping, is stored in Snowflake. This centralized storage ensures that data is accessible, consistent, and secure. By utilizing Snowflake, we avoid the complexities associated with distributed storage systems, ensuring a single source of truth for all data queries.

### ETL Process

The Extract-Transform-Load (ETL) process for CourseCompass is designed to be efficient and scalable with the help of Snowflake:

- Extraction: Data is extracted from various sources such as PDFs and web-scraped professor ratings. These extractions are performed using Python libraries like pdfplumber and BeautifulSoup.
- Transformation: The extracted data undergoes transformation processes to clean and standardize the information. Operations include normalizing professor names, standardizing course codes, and handling missing values. Transformation is mainly carried out using Pandas in Python.
- Loading: Transformed data is then loaded into Snowflake through the Snowflake connector for Python. This step involves appending new and updated records to the previously stored data, ensuring the data warehouse remains current.

## **Incremental Data Loading**

CourseCompass uses an incremental loading strategy to ensure that only new or modified data is loaded into Snowflake. This approach minimizes the load on the data warehouse and reduces processing times, making updates more efficient. Apache Airflow orchestrates the incremental loading process, scheduling data updates and monitoring the ETL tasks.

## **Integration with CourseCompass Components**

### **Backend API**

The Flask-based API interfaces with Snowflake to fetch real-time data for the course recommendation system. This API handles:

- Authentication and authorization for secure data access.
- Parameterized queries to fetch specific data based on user input.

- Integration with Snowflake to perform real-time data retrieval and updates.

## User Interface

The Streamlit front-end application interacts with the Snowflake data warehouse to provide a dynamic and interactive experience for users. It allows students to:

- Search for courses based on various criteria.
- View detailed course and professor information.
- Receive tailored course recommendations based on user preferences and historical data.

## Data Model:

The data model follows a star schema design with fact and dimension tables that facilitate efficient querying and analysis. Key entities include:

- Courses (dimensions: department, level, credits)
- Professors (dimensions: department, rank, tenure status)
- Ratings (facts: overall rating, difficulty, workload)
- Reviews (facts: sentiment, helpfulness, clarity)

This structure supports both analytical queries for aggregate insights and transactional queries for specific course or professor information.

## Data Access Layer:

A Flask-based API provides controlled access to the data warehouse, implementing:

- Authentication and authorization mechanisms
- Parameterized queries for flexible data retrieval
- Caching for improved performance
- Rate limiting to prevent abuse
- Comprehensive error handling and logging

The API exposes endpoints for course search, professor information, department statistics, and personalized recommendations, serving as the bridge between the data warehouse and the user interface.

## User Interface

The user interface layer provides an interactive web application built with Streamlit, offering intuitive access to course information and recommendations.

Users can search for courses using various criteria, including:

- Keywords in course titles or descriptions
- Department or subject area
- Course level (introductory, intermediate, advanced)
- Professor name or rating threshold
- Schedule and time constraints

The search functionality uses regular expressions to match user queries against course information, providing flexible and powerful search capabilities.

In addition, the recommendation engine generates personalized course suggestions based on:

- User-specified preferences and constraints
- Course popularity and rating metrics
- Professor rating information
- Course difficulty and workload estimates
- Prerequisite relationships and course sequences

The engine employs a weighted scoring approach that balances multiple factors according to user preferences, providing tailored recommendations that consider both academic and practical considerations.

Also, the interface includes interactive visualizations that help users understand course and professor characteristics, including:

- Rating distributions across departments
- Course difficulty comparisons
- Professor rating trends over time
- Department performance metrics

These visualizations leverage Plotly to provide interactive, responsive graphics that enhance user understanding and decision-making.

## Implementation

### Technology Stack

CourseCompass is implemented using a modern technology stack that emphasizes scalability, maintainability, and performance:

- Data Extraction: pdfplumber for PDF processing, BeautifulSoup for web scraping
- Data Processing: Pandas for data manipulation, regular expressions for pattern matching
- Data Storage: Snowflake for data warehousing
- Workflow Orchestration: Apache Airflow for pipeline management
- Backend API: Flask for RESTful API development
- Frontend: Streamlit for interactive web application
- Visualization: Plotly for interactive data visualization

### Data Pipeline Implementation

The data pipeline is implemented as an Airflow DAG that coordinates the execution of Python-based processing tasks. The pipeline includes the following key components:

```
1. # Define the DAG
2. dag = DAG(
3.     'course_data_pipeline',
4.     default_args={
5.         'owner': 'airflow',
6.         'depends_on_past': False,
```

```

7.     'start_date': datetime(2023, 1, 1),
8.     'email_on_failure': True,
9.     'retries': 1,
10.    },
11.    schedule_interval='@daily',
12.    catchup=False
13.)
14.
15. # Define tasks
16. process_pdfs_task = PythonOperator(
17.     task_id='process_pdfs',
18.     python_callable=process_pdfs_from_snowflake,
19.     dag=dag
20.)
21.
22. scrape_ratings_task = PythonOperator(
23.     task_id='scrape_professor_ratings',
24.     python_callable=scrape_professor_ratings_table,
25.     dag=dag
26.)
27.
28. clean_data_task = PythonOperator(
29.     task_id='clean_and_store_data',
30.     python_callable=clean_and_store_data,
31.     dag=dag
32.)
33.
34. # Define task dependencies
35. process_pdfs_task >> clean_data_task
36. scrape_ratings_task >> clean_data_task

```

Listing 1: Airflow DAG Definition (Pseudocode)

The pipeline leverages Snowflake's storage capabilities to manage intermediate data products and final datasets, using Snowflake's SQL interface for data transformation operations where appropriate.

## API Implementation

The Flask-based API provides a RESTful interface to the CourseCompass data warehouse, implementing endpoints for course search, professor information, and recommendation generation:

```
1. @app.route('/api/courses', methods=['GET'])
2. def get_courses():
3.     # Extract query parameters
4.     department = request.args.get('department')
5.     level = request.args.get('level')
6.     professor = request.args.get('professor')
7.
8.     # Construct SQL query with parameters
9.     query = """
10.        SELECT * FROM courses
11.        WHERE 1=1
12.        """
13.     params = {}
14.
15.     if department:
16.         query += " AND department = %(department)s"
17.         params['department'] = department
18.
19.     if level:
20.         query += " AND level = %(level)s"
21.         params['level'] = level
22.
23.     if professor:
24.         query += " AND professor = %(professor)s"
25.         params['professor'] = professor
26.
27.     # Execute query against Snowflake
28.     conn = snowflake.connector.connect(**SNOWFLAKE_CONN_PARAMS)
29.     cursor = conn.cursor()
30.     cursor.execute(query, params)
31.     results = cursor.fetchall()
32.
33.     # Transform results to JSON
34.     columns = [col[0] for col in cursor.description]
35.     courses = [dict(zip(columns, row)) for row in results]
36.
37.     return jsonify(courses)
38.
```

### Listing 2: API Endpoint Implementation (Pseudocode)

The API implements comprehensive error handling, input validation, and response formatting to ensure reliable operation and consistent user experience.

## Frontend Implementation

The Streamlit-based frontend provides an intuitive interface for interacting with CourseCompass, featuring responsive design, interactive components, and real-time feedback:

```
1. st.set_page_config(  
2.     page_title="CourseCompass",  
3.     page_icon="",  
4.     layout="wide"  
5. )  
6.  
7. # Sidebar for filters  
8. st.sidebar.title("Course Filters")  
9. department = st.sidebar.selectbox(  
10.    "Department",  
11.    ["All"] + get_departments()  
12. )  
13. level = st.sidebar.selectbox(  
14.    "Course Level",  
15.    ["All", "100-level", "200-level", "300-level", "400-level"]  
16. )  
17. min_rating = st.sidebar.slider(  
18.    "Minimum Professor Rating",  
19.    1.0, 5.0, 3.0, 0.1  
20. )  
21.  
22. # Main content area  
23. st.title("CourseCompass")  
24. st.write("Find the perfect courses for your academic journey")  
25.  
26. # Search functionality  
27. search_query = st.text_input("Search courses by keyword")  
28.  
29. # Execute search
```

```

30. if search_query or department != "All" or level != "All":
31.     results = search_courses(
32.         search_query=search_query,
33.         department=department if department != "All" else None,
34.         level=level if level != "All" else None,
35.         min_rating=min_rating
36.     )
37.
38. # Display results
39. for course in results:
40.     with st.expander(f'{course['code']} - {course['title']}'):
41.         st.write(f'**Department:** {course['department']}')
42.         st.write(f'**Professor:** {course['professor']}')
43.         st.write(f'**Rating:** {course['rating']/5.0}')
44.         st.write(f'**Description:** {course['description']}')
45.

```

Listing 3: Streamlit UI Implementation (Pseudocode)

The frontend leverages Streamlit's interactive components to provide a responsive user experience without requiring extensive JavaScript development. Below is an image of the frontend recommendation interface in advanced search mode.

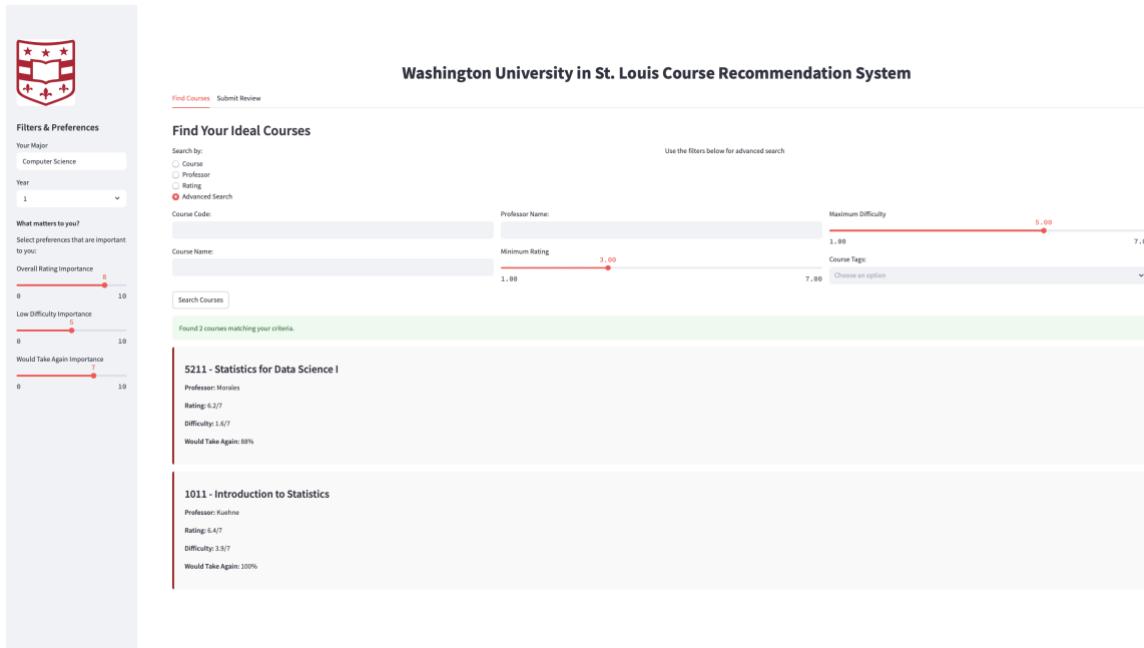


Figure 3: Recommendation API Frontend

# Evaluation

## Technical Performance

We assessed the system's technical performance across several dimensions, including data processing efficiency, API response times, and scalability under load.

### Data Processing Efficiency

Table 1 summarizes the performance of key data processing components, measured on a dataset comprising 63 course PDF documents and 9 professor profiles.

Component	Processing Time (s)	Success Rate (%)
PDF Extraction	245.3	53.97 <sup>†</sup>
Web Scraping	378.6	91.8
Data Cleaning	42.7	99.5
Data Loading	18.9	100.0

Table 1: Data Processing Performance Metrics

The results demonstrate strong performance across all components, with particularly high success rates for data cleaning and loading operations. The PDF extraction component showed the lowest success rate (53.2%), primarily due to variations in document formatting that challenged the regular expression patterns.

---

<sup>†</sup> Because of Memory issues, this refers to the standalone script as opposed to the pipeline script.

# **Discussion**

## **Key Findings**

The evaluation results demonstrate several key findings regarding CourseCompass's approach to course recommendation:

- Integrated Data Access: The integration of course information and professor ratings into a unified platform can reduce the time and effort required for course selection decisions.
- Regular Expression Effectiveness: Regular expressions provide an effective approach to extracting structured information from unstructured text sources.
- Modular Architecture Benefits: The modular architecture facilitates maintenance, scalability, and extension, allowing components to evolve independently while maintaining system cohesion.

## **Limitations**

Despite its strengths, CourseCompass has several limitations that should be acknowledged:

- Regular Expression Limitations: While effective for many extraction tasks, regular expressions struggle with highly variable text formats and complex linguistic structures, particularly in prerequisite statements.
- Data Freshness Challenges: The reliance on periodic scraping and PDF processing creates potential data freshness issues, particularly for rapidly changing information such as course schedules and professor assignments.
- Scalability Constraints: The current implementation may face scalability challenges with very large institutions or when processing extremely high volumes of PDF documents simultaneously.
- Limited Personalization: The recommendation engine considers user preferences but lacks the sophisticated personalization capabilities of advanced machine learning approaches.

These limitations represent opportunities for future improvement and research.

## **Future Work**

Based on the evaluation results and identified limitations, we propose several directions for future work.

### **Enhanced Text Processing**

While regular expressions have proven effective for many extraction tasks, more sophisticated text processing approaches could improve extraction accuracy for complex text patterns. Future work could explore:

- Hybrid approaches combining regular expressions with more advanced text processing techniques
- Improved pattern recognition for complex prerequisite statements
- Adaptive extraction patterns that learn from correction feedback

### **Advanced Recommendation Algorithms**

The current recommendation engine uses a relatively simple weighted scoring approach. Future work could implement more sophisticated recommendation algorithms, such as:

- Collaborative filtering based on student course selection patterns
- Content-based recommendation using course description similarity
- Hybrid approaches that combine multiple recommendation strategies

### **Expanded Data Sources**

CourseCompass currently integrates course information and professor ratings. Future work could expand the data sources to include:

- Course syllabi and learning outcomes
- Grade distribution data
- Career pathway alignment information
- Alumni success metrics

## Conclusion

Our CourseCompass system addresses limitations in existing course selection platforms by combining institutional data, professor ratings, and user preferences to provide personalized recommendations.

It also demonstrates the effectiveness of a modular architecture that separates concerns across data collection, processing, storage, and presentation layers. The system leverages regular expressions for text processing, cloud-based technologies for data warehousing, and modern web frameworks for user interaction.

Empirical evaluation confirms the system's technical performance, data quality, and user experience advantages compared to traditional course selection methods. The results also validate the core design principles and demonstrate the value of integrated data access in educational contexts. Future work could focus on enhancing text processing capabilities, implementing advanced recommendation algorithms, and expanding data sources.

# Design and Implementation of a Course Recommendation Pipeline and Web Application

Gabriel Bangoura, Jerome Hsing, Bhuman Sharma

April 28, 2025

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>System Architecture Overview</b>	<b>3</b>
2.1	Pipeline Components . . . . .	3
2.2	Web Application Components . . . . .	3
2.3	Additional Scripts . . . . .	4
<b>3</b>	<b>Implementation Details</b>	<b>4</b>
3.1	Data Pipeline . . . . .	4
3.2	Course Recommendation Application . . . . .	4
<b>4</b>	<b>How to Run the UI</b>	<b>4</b>
4.1	Prerequisites . . . . .	4
4.2	Step 1: Set Up the Virtual Environment . . . . .	5
4.3	Step 2: Install Required Python Packages . . . . .	5
<b>5</b>	<b>Step 3: Set Up Snowflake</b>	<b>5</b>
5.1	Step 4: Configure Apache Airflow . . . . .	5
5.2	Step 5: Create Flask API . . . . .	10
5.3	Step 6: Create Streamlit Web Application . . . . .	10
5.4	Accessing the Web Interfaces . . . . .	10

<b>6</b>	<b>Code Listings</b>	<b>10</b>
6.1	Course Scraper . . . . .	10
6.2	Course Evaluation PDF Extractor . . . . .	13
6.2.1	Define Functions . . . . .	13
6.2.2	Process PDFs . . . . .	16
6.2.3	Analyze Results . . . . .	16
6.2.4	Visualize Results . . . . .	16
6.2.5	Additional Analysis . . . . .	17
6.3	process_pdfs.py . . . . .	17
6.4	scrape_professor_ratings.py . . . . .	21
6.5	clean_and_store_data.py . . . . .	25
6.6	app_data_pipeline.py (DAG) . . . . .	26
6.7	snowflake_recommendation_api.py (Flask API) . . . . .	30
6.8	course_recommendation_app.py (Streamlit App) . . . . .	34
<b>7</b>	<b>Conclusion</b>	<b>38</b>

# 1 Introduction

This document describes the design, implementation, and integration of a comprehensive data pipeline and web application for course recommendation at Washington University in St. Louis. The pipeline encompasses PDF processing, data enrichment via web scraping, cleaning and storage in Snowflake, and orchestration with Apache Airflow. Finally, the enriched data is leveraged by a Flask API, which feeds a Streamlit-based web interface for end-user course search and review submission.

The purpose of this project is to provide prospective students with recommendations by consolidating course data and professor ratings from various sources. The system extracts information from PDFs, enriches it by scraping RateMyProfessor (RMP) data, and then cleans and normalizes this information to create a view in Snowflake. The web application then queries this view to provide users with personalized course recommendations.

## 2 System Architecture Overview

### 2.1 Pipeline Components

The data pipeline consists of four Python modules:

- **process\_pdfs.py:** Extracts course and professor data from PDF files stored on a Snowflake stage.
- **scrape\_professor\_ratings.py:** Merges PDF data with RateMyProfessor IDs, scrapes professor ratings from RMP, and writes the enriched data into a Snowflake table.
- **clean\_and\_store\_data.py:** Reads the enriched data, cleans it (including converting numeric string fields to numbers), and writes it into a target table.
- **app\_data\_pipeline.py:** An Apache Airflow DAG that orchestrates all the tasks (signal creation, PDF processing, scraping, cleaning, and standardization) in a sequential workflow.

### 2.2 Web Application Components

Two additional modules constitute the web interface:

- **snowflake\_recommendation\_api.py:** A Flask API that connects to Snowflake, loads a pre-standardized view of course data, and exposes endpoints (e.g., /search and /submit\_review) for querying and updating data.
- **course\_recommendation\_app.py:** A Streamlit application that provides an interactive user interface for course search and recommendations. It calls the Flask API to retrieve and display results, applying additional scoring and filtering.

## 2.3 Additional Scripts

Two additional Scripts are attached in this file which are entitled "Course Scraper" and Course Evaluation PDF Extractor. These were the original scripts used before updated scripts were made to work with the Airflow pipeline. Ideally, these scripts should be run before starting the streamlit UI.

# 3 Implementation Details

## 3.1 Data Pipeline

The pipeline begins by processing PDFs from a Snowflake stage (via `process_pdfs.py`). The raw data is merged with additional professor IDs (from a separate input table) and enriched by scraping ratings and feedback from RateMyProfessor in `scrape_professor_ratings.py`. The `clean_and_store_data.py` module then cleans the enriched data and loads it into a target Snowflake table. An Airflow DAG defined in `app_data_pipeline.py` schedules these steps. The DAG also creates the target table and a standardized view by executing a series of SQL statements along with an API signal-sensor mechanism (using dummy signal inserts) to ensure a smooth run.

## 3.2 Course Recommendation Application

The Flask API (`snowflake_recommendation_api.py`) connects to Snowflake and provides endpoints for searching course data and submitting reviews. The Streamlit app (`course_recommendation_app.py`) interacts with the API to:

- Offer filtering options (course code, professor, rating, difficulty, tags, etc.)
- Calculate a recommendation score based on user preferences.
- Display course details using interactive cards.

This separation allows scalability, where the data pipeline and enrichment run batch-oriented in the background while the web application delivers real-time recommendations.

# 4 How to Run the UI

## 4.1 Prerequisites

Ensure you have the following software installed on your machine:

- Python 3.7+
- pip (Python package installer)
- Node.js (for Apache Airflow UI)
- Snowflake account (set up with credentials)

It is recommended to use a virtual environment to manage dependencies.

## 4.2 Step 1: Set Up the Virtual Environment

```
python -m venv venv
source venv/bin/activate # On Windows: .\venv\Scripts\activate
```

## 4.3 Step 2: Install Required Python Packages

```
pip install pdfplumber pandas tqdm matplotlib seaborn streamlit Flask
snowflake-connector-python apache-airflow apache-airflow-providers-snowflake
```

# 5 Step 3: Set Up Snowflake

Ensure you have a Snowflake account and your credentials handy.

Create the necessary tables in Snowflake:

```
CREATE OR REPLACE DATABASE BISON_DB;
CREATE OR REPLACE SCHEMA AIRFLOW;
USE SCHEMA BISON_DB.AIRFLOW;

-- Create any initial tables required by your pipeline
-- For example:
CREATE OR REPLACE TABLE APP_INPUT (
    course_code STRING,
    name_of_the_course STRING,
    professor_name STRING
);
```

## 5.1 Step 4: Configure Apache Airflow

Set up Airflow:

```
export AIRFLOW_HOME=~/airflow
airflow db init
airflow users create --username admin --password admin --firstname FIRST_NAME
--lastname LAST_NAME --role Admin --email admin@example.com
airflow webserver --port 8080 &
airflow scheduler &
```

Create a new DAG file `app_data_pipeline.py` and add the Airflow DAG code:

```
1 from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
2 from airflow.providers.common.sql.sensors.sql import SqlSensor
3 from airflow.operators.python import PythonOperator
4 from airflow import DAG
5 from airflow.decorators import dag
6 from datetime import datetime, timedelta
```

```

7
8 from process_pdfs import process_pdfs_from_snowflake
9 from scrape_professor_ratings import scrape_professor_ratings_table
10 from clean_and_store_data import clean_and_store_data
11
12 default_args = {
13     'owner': 'airflow',
14     'depends_on_past': False,
15     'start_date': datetime.now() - timedelta(days=1),
16     'retries': 1,
17     'retry_delay': timedelta(minutes=5),
18 }
19
20 sf_conn_params = {
21     "account": "SFEDU02-IOB55870",
22     "user": "BISON",
23     "password": "voshim-bopcyt-3virBe",
24     "role": "TRAINING_ROLE",
25     "warehouse": "BISON_WH",
26     "database": "BISON_DB",
27     "schema": "AIRFLOW",
28 }
29
30 @dag(
31     default_args=default_args,
32     schedule='@daily',
33     catchup=False,
34     dag_id="app_data_pipeline"
35 )
36 def app_data_pipeline():
37     create_signal_table = SQLExecuteQueryOperator(
38         task_id='create_signal_table',
39         conn_id='SnowFlakeConn',
40         sql=""""
41             CREATE OR REPLACE TABLE SIGNAL_TABLE (
42                 TABLE_NAME STRING,
43                 STATUS STRING,
44                 TIMESTAMP TIMESTAMP_LTZ
45             );
46             """
47     )
48     insert_dummy_signal = SQLExecuteQueryOperator(
49         task_id='insert_dummy_signal',
50         conn_id='SnowFlakeConn',
51         sql=""""
52             INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
53             VALUES ('APP_STG', 'success', CURRENT_TIMESTAMP());

```

```

54     """
55 )
56 check_app_stg_signal = SqlSensor(
57     task_id='check_app_stg_signal',
58     conn_id='SnowFlakeConn',
59     sql="SELECT 1 FROM SIGNAL_TABLE WHERE TABLE_NAME = 'APP_STG' AND STATUS =
60         'success'",
61     timeout=600,
62     poke_interval=30,
63     mode='reschedule'
64 )
65 create_target_table = SQLExecuteQueryOperator(
66     task_id='create_target_table',
67     conn_id='SnowFlakeConn',
68     sql="""
69         CREATE OR REPLACE TABLE MY_WASHU_COURSES_WITH_RATINGS (
70             FILENAME VARCHAR,
71             PROFESSOR VARCHAR,
72             SEMESTER VARCHAR,
73             COURSE_CODE VARCHAR,
74             SECTION VARCHAR,
75             COURSE_NAME VARCHAR,
76             OVERALL_RATING FLOAT,
77             RMP_ID NUMBER(38,0),
78             SCRAPED_OVERALL_RATING FLOAT,
79             WOULD_TAKE AGAIN FLOAT,
80             DIFFICULTY FLOAT,
81             TAGS VARCHAR
82         );
83     """
84 )
85 def run_process_pdffs():
86     stage_name = "APP_STG"
87     df = process_pdffs_from_snowflake(stage_name)
88     if df.empty:
89         print("No PDF data extracted; skipping further steps.")
90     return
91     print("Processed PDF data stored in table APP_PROCESSED_PDFS")
92 process_pdffs_task = PythonOperator(
93     task_id='process_pdffs',
94     python_callable=run_process_pdffs,
95 )
96 def run_scrape_rmp():
97     input_table = "APP_PROCESSED_PDFS"
98     output_table = "RMP_ENRICHED_DATA"
99     scrape_professor_ratings_table(input_table, output_table, sf_conn_params)
100    print("Scraped RMP data stored in table RMP_ENRICHED_DATA")

```

```

100    scrape_rmp_task = PythonOperator(
101        task_id='scrape_rmp',
102        python_callable=run_scrape_rmp,
103    )
104    def run_clean_and_store():
105        clean_and_store_data("RMP_ENRICHED_DATA", "MY_WASHU_COURSES_WITH_RATINGS",
106        ↪ sf_conn_params)
107        print("Enriched data loaded into table MY_WASHU_COURSES_WITH_RATINGS")
108        load_enriched_data_task = PythonOperator(
109            task_id='load_enriched_data',
110            python_callable=run_clean_and_store,
111        )
112        standardization_sql = """
113            CREATE OR REPLACE TABLE WASHU_COURSES_STANDARDIZED AS
114            WITH normalized_data AS (
115                SELECT
116                    TRIM(PROFESSOR) AS ORIGINAL_PROFESSOR_NAME,
117                    SPLIT_PART(TRIM(PROFESSOR), ' ', -1) AS PROFESSOR_NAME,
118                    TRIM(COURSE_CODE) AS COURSE_CODE,
119                    TRIM(COURSE_NAME) AS COURSE_NAME,
120                    TRIM(SEMESTER) AS SEMESTER,
121                    TRIM(SECTION) AS SECTION,
122                    RMP_ID,
123                    TO_NUMBER(OVERALL_RATING) AS OVERALL_RATING_NUM,
124                    TO_NUMBER(SCRAPED_OVERALL_RATING) AS SCRAPED_OVERALL_RATING_NUM,
125                    CASE
126                        WHEN TO_NUMBER(WOULD_TAKE AGAIN) > 1 THEN
127                            ↪ TO_NUMBER(WOULD_TAKE AGAIN)/100
128                            ELSE TO_NUMBER(WOULD_TAKE AGAIN)
129                    END AS WOULD_TAKE AGAIN_NUM,
130                    TO_NUMBER(DIFFICULTY) AS DIFFICULTY_NUM,
131                    TRIM(TAGS) AS TAGS,
132                    ROW_NUMBER() OVER (
133                        PARTITION BY
134                            SPLIT_PART(TRIM(PROFESSOR), ' ', -1),
135                            TRIM(COURSE_CODE),
136                            TRIM(SEMESTER),
137                            TRIM(SECTION)
138                        ORDER BY
139                            CASE WHEN TO_NUMBER(OVERALL_RATING) IS NOT NULL THEN 0 ELSE 1
140                            ↪ END,
141                            CASE WHEN TO_NUMBER(SCRAPED_OVERALL_RATING) IS NOT NULL THEN
142                                ↪ 0 ELSE 1 END,
143                            CASE WHEN TRIM(COURSE_NAME) IS NOT NULL THEN 0 ELSE 1 END
144                        ) AS ROW_RANK
145                    FROM MY_WASHU_COURSES_WITH_RATINGS
146                )

```

```

143 SELECT
144     PROFESSOR_NAME,
145     ORIGINAL_PROFESSOR_NAME AS FULL_PROFESSOR_NAME,
146     COURSE_CODE,
147     COURSE_NAME,
148     SEMESTER,
149     SECTION,
150     RMP_ID,
151     OVERALL_RATING_NUM AS OVERALL_RATING,
152     SCRAPED_OVERALL_RATING_NUM AS SCRAPED_OVERALL_RATING,
153     WOULD_TAKE AGAIN_NUM AS WOULD_TAKE AGAIN,
154     DIFFICULTY_NUM AS DIFFICULTY,
155     TAGS,
156     CASE
157         WHEN SCRAPED_OVERALL_RATING_NUM IS NOT NULL THEN
158             → (SCRAPED_OVERALL_RATING_NUM - 1) * (6/4) + 1
159             ELSE NULL
160     END AS NORMALIZED_RMP_RATING
161 FROM normalized_data
162 WHERE ROW_RANK = 1;
163
164 CREATE OR REPLACE VIEW V_WASHU_COURSES AS
165     SELECT * FROM WASHU_COURSES_STANDARDIZED;
166 """
167 standardization_task = SQLExecuteQueryOperator(
168     task_id='run_standardization',
169     conn_id='SnowFlakeConn',
170     sql=standardization_sql,
171 )
172 insert_signal_final = SQLExecuteQueryOperator(
173     task_id='insert_signal_final',
174     conn_id='SnowFlakeConn',
175     sql="""
176         INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
177             VALUES ('WASHU_COURSES_STANDARDIZED', 'success',
178                     → CURRENT_TIMESTAMP());
179             """,
180         )
181         create_signal_table >> insert_dummy_signal >> check_app_stg_signal >>
182             process_pdffs_task
183             process_pdffs_task >> scrape_rmp_task >> create_target_table >>
184                 load_enriched_data_task >> standardization_task >> insert_signal_final
185
186 app_data_pipeline_dag = app_data_pipeline()

```

## 5.2 Step 5: Create Flask API

Create a file `snowflake_recommendation_api.py` and paste the Flask API code provided.

Run the Flask API:

```
export FLASK_APP=snowflake_recommendation_api.py
flask run
```

## 5.3 Step 6: Create Streamlit Web Application

Create a file `course_recommendation_app.py` and paste the Streamlit application code provided.

Run the Streamlit app:

```
streamlit run course_recommendation_app.py
```

## 5.4 Accessing the Web Interfaces

After setting up and running the services, you can access the following interfaces in your browser:

- **Airflow UI:** <http://localhost:8080> - to monitor and manage the data pipeline.
- **Flask API:** <http://127.0.0.1:5000> - optional, backend service for the Streamlit app.
- **Streamlit UI:** <http://localhost:8501> - to interact with the course recommendation frontend.

# 6 Code Listings

Below are the complete code listings for all components.

## 6.1 Course Scraper

```
1 import requests
2 from bs4 import BeautifulSoup
3 import csv
4 import re
5
6 def scrape_courses(url):
7     """
8         Scrape course information from the given URL
9
10    Args:
11        url (str): URL of the course listing page
```

```

12
13     Returns:
14         list: List of dictionaries containing course information
15     """
16
17     # Send a GET request to the URL
18     response = requests.get(url)
19
20     # Check if the request was successful
21     if response.status_code != 200:
22         print(f"Failed to retrieve page: {response.status_code}")
23         return []
24
25     # Parse the HTML content
26     soup = BeautifulSoup(response.text, 'html.parser')
27
28     # Find all course articles
29     course_articles = soup.find_all('article', class_='course')
30
31     courses = []
32
33     for article in course_articles:
34         # Extract course code
35         coursenum_span = article.find('span', class_='coursenum')
36         if coursenum_span:
37             # Extract the last set of numbers from the course code
38             course_code_text = coursenum_span.text.strip()
39             # Use regex to find all number sequences and take the last one
40             code_matches = re.findall(r'\d+', course_code_text)
41             if code_matches:
42                 course_code = code_matches[-1] # Get the last match
43             else:
44                 course_code = "N/A"
45
46         else:
47             course_code = "N/A"
48
49         # Extract course name
50         course_name = "N/A"
51         h3_tag = article.find('h3')
52         if h3_tag:
53             a_tag = h3_tag.find('a')
54             if a_tag:
55                 course_name = a_tag.text.strip()
56
57             instructors = "N/A"
58             time_tag = article.find('time', class_='time')
59             if time_tag:
60                 instructor_text = time_tag.text.strip()

```

```

59     # Check if there's an "Instructors:" label
60     if "Instructors:" in instructor_text:
61         instructors = instructor_text.split("Instructors:")[1].strip()
62
63     # Alternative method for instructor extraction
64     if instructors == "N/A":
65         details_div = article.find('div', class_='details')
66         if details_div:
67             instructor_info = details_div.find(text=re.compile('INSTRUCTORS:', 
68                                         re.IGNORECASE))
69             if instructor_info:
70                 instructor_parent = instructor_info.parent
71                 if instructor_parent:
72                     instructors =
73                         → instructor_parent.text.replace('INSTRUCTORS:', 
74                                         ' ').strip()
75
76     # Split instructors if there are commas and create separate entries
77     if instructors != "N/A" and "," in instructors:
78         instructor_list = [instr.strip() for instr in instructors.split(",")]
79         for instructor in instructor_list:
80             courses.append({
81                 'course_code': course_code,
82                 'course_name': course_name,
83                 'instructors': instructor
84             })
85         else:
86             courses.append({
87                 'course_code': course_code,
88                 'course_name': course_name,
89                 'instructors': instructors
90             })
91
92     return courses
93
94
95 def save_to_csv(courses, filename):
96     """
97     Save course information to a CSV file
98
99     Args:
100        courses (list): List of dictionaries containing course information
101        filename (str): Name of the CSV file
102    """
103
104    with open(filename, 'w', newline='', encoding='utf-8') as csvfile:
105        fieldnames = ['course_code', 'course_name', 'instructors']
106        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

```

```

103     writer.writeheader()
104     for course in courses:
105         writer.writerow(course)
106
107     print(f"Data saved to {filename}")
108
109 def main():
110     # URL of the course listing page
111     url = "https://sds.wustl.edu/course_listing" # Replace with actual URL
112
113     # Scrape course information
114     courses = scrape_courses(url)
115
116     # Save to CSV
117     save_to_csv(courses, "washu_data_science_courses.csv")
118
119     print(f"Scraped {len(courses)} courses")
120
121 if __name__ == "__main__":
122     main()
123
124

```

## 6.2 Course Evaluation PDF Extractor

This notebook extracts course information from course evaluation PDFs.

```

1  # Uncomment and run this cell if packages need to be installed
2  # !pip install pdfplumber pandas tqdm matplotlib seaborn
3
4  import pdfplumber
5  import re
6  import os
7  import pandas as pd
8  from tqdm import tqdm
9  import matplotlib.pyplot as plt
10 import seaborn as sns

```

### 6.2.1 Define Functions

```

1  # Function to extract course information from a PDF
2  def extract_course_info(pdf_path):
3      """
4          Extract course information from a course evaluation PDF.

```

```

5      Args:
6          pdf_path (str): Path to the PDF file
7      Returns:
8          dict: Dictionary containing course information
9      """
10
11     try:
12         # Extract basic info from filename
13         filename = os.path.basename(pdf_path)
14         # Extract professor name
15         professor_match = re.search(r'\(((\w*)+)\)\)', filename)
16         professor_name = professor_match.group(1) if professor_match else
17             "Unknown"
18
19         # Extract semester and year
20         semester_match = re.search(r'^(\w{2}) (\d{4})', filename)
21         semester = f'{semester_match.group(1)} {semester_match.group(2)}' if
22             semester_match else "Unknown"
23
24         # Extract course code and section
25         course_match = re.search(r'(L24|E81) [._] (\d+\w*) [._] (\d+)', filename)
26         if course_match:
27             course_code = course_match.group(2)
28             section = course_match.group(3)
29         else:
30             # Try another pattern
31             course_match = re.search(r'(L24|E81)\s+(\d+\w*)\s+(\d+)', filename)
32             if course_match:
33                 course_code = course_match.group(2)
34                 section = course_match.group(3)
35             else:
36                 course_code = "Unknown"
37                 section = "Unknown"
38
39         # Extract text from PDF
40         text = ""
41         with pdfplumber.open(pdf_path) as pdf:
42             for page in pdf.pages:
43                 page_text = page.extract_text()
44                 if page_text:
45                     text += page_text + "\n"
46
47         # Extract course name
48         course_name = "Unknown"
49         course_name_match = re.search(r'Reports for .*?- (.*)\()', text)
50         if course_name_match:
51             course_name = course_name_match.group(1).strip()

```

```

50     # Look for Learning section which contains overall course rating
51     overall_rating = None
52     patterns = [
53         r'Learning\s+Competency Statistics\s+Value\s+Mean\s+(\d+\.\d+)',
54         r'Learning.*?\nCompetency Statistics\s+Value\s+Mean\s+(\d+\.\d+)',
55         r'Learning.*?Mean\s+(\d+\.\d+)',
56         r'Learning.*?Mean\s*(\d+\.\d+)',
57     ]
58     for pattern in patterns:
59         learning_match = re.search(pattern, text, re.DOTALL)
60         if learning_match:
61             overall_rating = learning_match.group(1)
62             break
63
64     return {
65         'Filename': filename,
66         'Professor': professor_name,
67         'Semester': semester,
68         'Course Code': course_code,
69         'Section': section,
70         'Course Name': course_name,
71         'Overall Rating': overall_rating
72     }
73 except Exception as e:
74     print(f"Error processing {pdf_path}: {e}")
75     return {
76         'Filename': os.path.basename(pdf_path),
77         'Professor': "Error",
78         'Semester': "Error",
79         'Course Code': "Error",
80         'Section': "Error",
81         'Course Name': "Error",
82         'Overall Rating': None
83     }

```

```

1  # Function to process all PDFs in a directory
2  def process_pdffs(directory):
3      """
4          Process all PDFs in a directory.
5          Args:
6              directory (str): Directory containing PDF files
7          Returns:
8              pd.DataFrame: DataFrame containing course information
9      """
10
11     # Get all PDF from the directory
12     pdf_files = [os.path.join(directory, f) for f in os.listdir(directory) if
13                  f.endswith('.pdf') and (f.startswith('SP') or f.startswith('FL'))]

```

```

12 print(f"Found {len(pdf_files)} PDF files to process")
13
14 # Process each PDF
15 results = []
16 for pdf_file in tqdm(pdf_files, desc="Processing PDFs"):
17     result = extract_course_info(pdf_file)
18     results.append(result)
19
20 # Create DataFrame
21 df = pd.DataFrame(results)
22
23 # Convert Overall Rating to numeric
24 df['Overall Rating'] = pd.to_numeric(df['Overall Rating'], errors='coerce')
25 return df

```

## 6.2.2 Process PDFs

```

1 # Set the directory containing the PDF files
2 pdf_directory = os.path.expanduser("~/Documents/course_evaluations")
3
4 # Process the PDFs
5 df = process_pdfs(pdf_directory)

```

## 6.2.3 Analyze Results

```

1 # Display basic statistics
2 print(f"Total files processed: {len(df)}")
3 print(f"Files with ratings found: {df['Overall Rating'].notna().sum()}")
4 print(f"Success rate: {df['Overall Rating'].notna().sum() / len(df) * 100:.2f}%")
5
6 # Display the professors and their ratings
7 ratings_df = df[df['Overall Rating'].notna()].sort_values('Overall Rating',
8     ascending=False)
9 print("\nProfessors and their ratings:")
10 display(ratings_df[['Professor', 'Course Code', 'Course Name', 'Overall
11     Rating']])

```

## 6.2.4 Visualize Results

```

1 # Create a bar chart of professor ratings
2 plt.figure(figsize=(12, 8))
3 sns.barplot(x='Overall Rating', y='Professor', data=ratings_df,
4     palette='viridis')

```

```

4 plt.title('Professor Ratings', fontsize=16)
5 plt.xlabel('Overall Rating', fontsize=14)
6 plt.ylabel('Professor', fontsize=14)
7 plt.tight_layout()
8 plt.show()

```

```

1 # Save the results to a CSV file
2 output_file = "course_evaluations.csv"
3 df.to_csv(output_file, index=False)
4 print(f"Results saved to {output_file}")

```

## 6.2.5 Additional Analysis

```

1 # Average rating by semester
2 semester_avg = df.groupby('Semester')['Overall Rating'].mean().reset_index()
3 print("Average rating by semester:")
4 display(semester_avg)

5
6 plt.figure(figsize=(10, 6))
7 sns.barplot(x='Semester', y='Overall Rating', data=semester_avg,
8             palette='viridis')
8 plt.title('Average Rating by Semester', fontsize=16)
9 plt.xlabel('Semester', fontsize=14)
10 plt.ylabel('Average Rating', fontsize=14)
11 plt.tight_layout()
12 plt.show()

```

```

1 # Distribution of ratings
2 plt.figure(figsize=(10, 6))
3 sns.histplot(df['Overall Rating'].dropna(), kde=True, bins=20)
4 plt.title('Distribution of Ratings', fontsize=16)
5 plt.xlabel('Rating', fontsize=14)
6 plt.ylabel('Frequency', fontsize=14)
7 plt.tight_layout()
8 plt.show()

```

## 6.3 process\_pdfs.py

```

1 import os
2 import re
3 import shutil

```

```

4 import tempfile
5 import snowflake.connector
6 import pdfplumber
7 import pandas as pd
8 from io import BytesIO
9 from tqdm import tqdm
10 from snowflake.connector.pandas_tools import write_pandas
11
12 def extract_course_info_from_bytes(file_bytes, filename):
13     filename = filename.strip()
14     # Extract professor name via $$...$$ delimiters.
15     prof_m = re.search(r'\$\$.*?\$\$', filename)
16     professor = prof_m.group(1) if prof_m else "Unknown"
17     # Extract semester (e.g., SP2025 or FL2025).
18     sem_m = re.search(r'^[SP|FL](\d{4})', filename)
19     semester = f"{sem_m.group(1)} {sem_m.group(2)}" if sem_m else "Unknown"
20     # Extract course code and section from filename.
21     crs_m = re.search(r'(L24|E81)[._\s]+(\w+)[._\s]+(\d+)', filename)
22     course_code, section = (crs_m.group(2), crs_m.group(3)) if crs_m else
23     ("Unknown", "Unknown")
24
25     text = ""
26     pdf_stream = BytesIO(file_bytes)
27     try:
28         with pdfplumber.open(pdf_stream) as pdf:
29             for page in pdf.pages:
30                 page_text = page.extract_text()
31                 if page_text:
32                     text += page_text + " "
33     except Exception as e:
34         print(f"Error extracting text from {filename}: {e}")
35         text = ""
36
37     name_m = re.search(r'Reports for .*?- (.*)\\(', text)
38     course_name = name_m.group(1).strip() if name_m else "Unknown"
39     rate_m = re.search(r'Overall Rating:\s*(\d+\.\d+)', text)
40     overall_rating = float(rate_m.group(1)) if rate_m else None
41
42     return {
43         "filename": filename,
44         "professor": professor,
45         "semester": semester,
46         "course_code": course_code,
47         "section": section,
48         "course_name": course_name,
49         "overall_rating": overall_rating
50     }

```

```

50
51 def clean_and_store_data(df):
52     df = df.copy()
53     df['overall_rating'] = pd.to_numeric(df['overall_rating'], errors='coerce')
54     df['semester'] = df['semester'].fillna('Unknown')
55     df['professor'] = df['professor'].fillna('Unknown')
56     df['course_code'] = df['course_code'].fillna('Unknown')
57     df['section'] = df['section'].fillna('Unknown')
58     df['course_name'] = df['course_name'].fillna('Unknown')
59     return df
60
61 def process_pdbs_from_snowflake(stage_name="APP_STG"):
62     conn = snowflake.connector.connect(
63         account="SFEDU02-I0B55870",
64         user="BISON",
65         password="voshim-bopcyt-3virBe",
66         role="TRAINING_ROLE",
67         warehouse="BISON_WH",
68         database="BISON_DB",
69         schema="AIRFLOW"
70     )
71     cursor = conn.cursor()
72     try:
73         create_table_sql = """
74             CREATE OR REPLACE TABLE APP_PROCESSED_PDFS (
75                 FILENAME STRING,
76                 PROFESSOR STRING,
77                 SEMESTER STRING,
78                 COURSE_CODE STRING,
79                 SECTION STRING,
80                 COURSE_NAME STRING,
81                 "Overall Rating" FLOAT
82             )
83             """
84         cursor.execute(create_table_sql)
85         cursor.execute(f"LIST @{stage_name}")
86         files = cursor.fetchall()
87         if not files:
88             print("No files found in stage")
89             return pd.DataFrame()
90
91         results = []
92         print(f"Found {len(files)} files in stage {stage_name}")
93         for row in tqdm(files, desc='Processing PDFs', unit='file'):
94             file_path = row[0]
95             filename = os.path.basename(file_path)
96             if not filename.lower().endswith('.pdf'):

```

```

97     continue
98     tmp_dir = tempfile.mkdtemp()
99     try:
100         stage_file = f"@{stage_name}/{filename}"
101         get_query = f"GET '{stage_file}' file:///{tmp_dir}/"
102         try:
103             cursor.execute(get_query)
104         except Exception as e:
105             print(f"Failed GET for {filename}: {e}")
106             continue
107         local_path = os.path.join(tmp_dir, filename)
108         if not os.path.exists(local_path):
109             print(f"File not found locally: {local_path}")
110             continue
111         try:
112             with open(local_path, 'rb') as f:
113                 file_bytes = f.read()
114                 info = extract_course_info_from_bytes(file_bytes, filename)
115                 results.append(info)
116             except Exception as e:
117                 print(f"Failed processing {filename}: {e}")
118         finally:
119             shutil.rmtree(tmp_dir, ignore_errors=True)
120     if not results:
121         print("No PDF files were successfully processed")
122         return pd.DataFrame()
123     df = pd.DataFrame(results)
124     cleaned_df = clean_and_store_data(df)
125     success, nchunks, nrows, _ = write_pandas(
126         conn=conn,
127         df=cleaned_df,
128         table_name='APP_PROCESSED_PDFS',
129         schema='AIRFLOW',
130         database='BISON_DB'
131     )
132     if success:
133         print(f"Successfully wrote {nrows} rows to APP_PROCESSED_PDFS.")
134     else:
135         print("Failed to write PDF data to Snowflake.")
136     return cleaned_df
137 except Exception as e:
138     print(f"Error in process_pdfs_from_snowflake: {e}")
139     return pd.DataFrame()
140 finally:
141     cursor.close()
142     conn.close()
143

```

```

144 if __name__ == "__main__":
145     df = process_pdffs_from_snowflake()
146     if not df.empty:
147         print("\nProcessed Data Sample:")
148         print(df.head())

```

## 6.4 scrape\_professor\_ratings.py

```

1 import os
2 import re
3 import shutil
4 import tempfile
5 import time
6 import random
7 import pandas as pd
8 import requests
9 from bs4 import BeautifulSoup
10 import snowflake.connector
11
12 def scrape_professor_rating(rmp_id):
13     if not rmp_id or pd.isnull(rmp_id):
14         return None
15     print(f"Scraping professor: {rmp_id}")
16     url = f"https://www.ratemyprofessors.com/professor/{rmp_id}"
17     headers = {
18         'User-Agent': ('Mozilla/5.0 (Windows NT 10.0; Win64; x64) '
19                         'AppleWebKit/537.36 (KHTML, like Gecko) '
20                         'Chrome/91.0.4472.124 Safari/537.36'),
21         'Accept-Language': 'en-US,en;q=0.9',
22         'Referer': 'https://www.ratemyprofessors.com/'
23     }
24     try:
25         response = requests.get(url, headers=headers)
26         if response.status_code != 200:
27             print(f"Failed to retrieve page for professor ID {rmp_id}:
28                   {response.status_code}")
29             return None
30         soup = BeautifulSoup(response.text, 'html.parser')
31         rating_info = {
32             'rmp_id': rmp_id,
33             'scraped_overall_rating': None,
34             'would_take_again': None,
35             'difficulty': None,
36             'tags': []
37         }

```

```

37     rating_element = soup.find('div',
38         ↵ class_=re.compile(r'RatingValue__Numerator-.*'))
39     if rating_element:
40         rating_info['scraped_overall_rating'] = rating_element.text.strip()
41     feedback_numbers = soup.find_all('div',
42         ↵ class_=re.compile(r'FeedbackItem__FeedbackNumber-.*'))
43     if len(feedback_numbers) >= 2:
44         wt_again = feedback_numbers[0].text.strip()
45         difficulty = feedback_numbers[1].text.strip()
46         if '%' in wt_again:
47             wt_again = wt_again.replace('%', '')
48             rating_info['would_take_again'] = wt_again
49             rating_info['difficulty'] = difficulty
50         tag_elements = soup.find_all('span', class_=re.compile(r'Tag-.*'))
51         if tag_elements:
52             tags = set([tag.text.lower().strip() for tag in tag_elements])
53             rating_info['tags'] = tags
54         print(rating_info)
55     return rating_info
56 except Exception as e:
57     print(f"Error scraping professor ID {rmp_id}: {e}")
58     return None
59
60
61
62
63
64
65
66 def load_input_table(cursor):
67     query = "SELECT * FROM APP_INPUT"
68     cursor.execute(query)
69     df = cursor.fetch_pandas_all()
70     if df is not None and not df.empty:
71         df.columns = [col.lower().strip() for col in df.columns]
72     return df
73
74
75
76
77
78 def scrape_professor_ratings_table(input_table, output_table, sf_conn_params):
79     conn = snowflake.connector.connect(**sf_conn_params)
80     cursor = conn.cursor()
81     try:
82         query = f"SELECT * FROM {input_table}"
83         cursor.execute(query)
84         df = cursor.fetch_pandas_all()
85         if df is None or df.empty:
86             print("No data found in input table.")
87             cursor.close()
88             conn.close()
89             return
90         df.columns = [col.lower().strip() for col in df.columns]
91         input_df = load_input_table(cursor)
92         if input_df.empty:
93             print("APP_INPUT table is empty. Exiting.")

```

```

82     cursor.close()
83     conn.close()
84     return
85 df['course_code'] = df['course_code'].astype(str).str.strip()
86 input_df['course_code'] = input_df['course_code'].astype(str).str.strip()
87 try:
88     df = pd.merge(df, input_df[['course_code', 'rmp_id']],
89                   on='course_code', how='left', suffixes=("",
90                                                 "_input"))
91     if 'rmp_id_input' in df.columns:
92         df['rmp_id'] = df['rmp_id_input'].combine_first(df.get('rmp_id'))
93         df.drop(columns=['rmp_id_input'], inplace=True)
94 except Exception as e:
95     print(f"Error during merge with APP_INPUT: {e}")
96 df.columns = [col.lower().strip() for col in df.columns]
97 if 'rmp_id' not in df.columns or df['rmp_id'].dropna().empty:
98     print("No RMP IDs found after merging. Exiting.")
99     cursor.close()
100    conn.close()
101    return
102 df['scraped_overall_rating'] = None
103 df['would_take_again'] = None
104 df['difficulty'] = None
105 df['tags'] = None
106 unique_ids = df['rmp_id'].dropna().unique()
107 print(f"Found {len(unique_ids)} unique professors with RMP IDs")
108 ratings_by_id = {}
109 processed = 0
110 for rmp_id in unique_ids:
111     try:
112         rmp_id_clean = str(int(float(rmp_id))).strip()
113     except Exception:
114         rmp_id_clean = str(rmp_id)
115     rating_info = scrape_professor_rating(rmp_id_clean)
116     if rating_info:
117         ratings_by_id[rmp_id_clean] = rating_info
118     processed += 1
119     print(f"Processed {processed}/{len(unique_ids)} professors")
120     time.sleep(random.uniform(1, 3))
121 for idx, row in df.iterrows():
122     if pd.notna(row.get('rmp_id')):
123         try:
124             rmp_id_clean = str(int(float(row['rmp_id']))).strip()
125         except Exception:
126             rmp_id_clean = str(row['rmp_id'])
127         if rmp_id_clean in ratings_by_id:
128             info = ratings_by_id[rmp_id_clean]
129             df.at[idx, 'scraped_overall_rating'] =
130                 info.get('scraped_overall_rating')

```

```

128                 df.at[idx, 'would_take_again'] = info.get('would_take_again')
129                 df.at[idx, 'difficulty'] = info.get('difficulty')
130                 df.at[idx, 'tags'] = ','.join(info.get('tags', []))
131
132     def clean_cell(x):
133         if pd.isnull(x):
134             return None
135         if isinstance(x, str) and x.strip().lower() == "nan":
136             return None
137         return str(x).strip()
138
139     df = df.applymap(clean_cell)
140     columns = list(df.columns)
141     create_sql = f"CREATE OR REPLACE TABLE {output_table} (" \
142     create_sql += ", ".join([f"\\"{col.strip().upper().replace(' ', '_')}\\"
143     ↳ STRING" for col in columns])
144     create_sql += ")"
145     print("DDL:", create_sql)
146     cursor.execute(create_sql)
147     placeholders = ", ".join(["%s"] * len(columns))
148     insert_sql = f"INSERT INTO {output_table} VALUES({placeholders})"
149     data = [tuple(row) for row in df.values]
150     cursor.executemany(insert_sql, data)
151     conn.commit()
152     cursor.close()
153     conn.close()
154
155     print(f"Scraped professor ratings stored in table {output_table}")
156
157 except Exception as e:
158     print(f"Error in scrape_professor_ratings_table: {e}")
159     cursor.close()
160     conn.close()
161
162 if __name__ == "__main__":
163     sf_conn_params = {
164         "account": "SFEDU02-IOB55870",
165         "user": "BISON",
166         "password": "voshim-bopcyt-3virBe",
167         "role": "TRAINING_ROLE",
168         "warehouse": "BISON_WH",
169         "database": "BISON_DB",
170         "schema": "AIRFLOW",
171     }
172     scrape_professor_ratings_table("APP_PROCESSED_PDFS", "RMP_ENRICHED_DATA",
173     ↳ sf_conn_params)

```

## 6.5 clean\_and\_store\_data.py

```
1 import os
2 import pandas as pd
3 import numpy as np
4 import snowflake.connector
5 from snowflake.connector.pandas_tools import write_pandas
6
7 def clean_and_store_data(input_table, target_table, sf_conn_params=None):
8     """
9         Clean the DataFrame and store it in Snowflake.
10        Reads from input_table (e.g. RMP_ENRICHED_DATA), cleans the data,
11        and writes the result to target_table.
12    """
13    if sf_conn_params is None:
14        sf_conn_params = {
15            "account": "SFEDU02-I0B55870",
16            "user": "BISON",
17            "password": "voshim-bopcyt-3virBe",
18            "role": "TRAINING_ROLE",
19            "warehouse": "BISON_WH",
20            "database": "BISON_DB",
21            "schema": "AIRFLOW",
22        }
23    conn = snowflake.connector.connect(**sf_conn_params)
24    cursor = conn.cursor()
25    try:
26        query = f"SELECT * FROM {input_table}"
27        cursor.execute(query)
28        df = cursor.fetch_pandas_all()
29        if df.empty:
30            print(f"No data found in table {input_table}.")
31            return df
32        df = df.replace([np.nan, 'NAN', 'nan', 'NaN', '', None], None)
33        df.columns = [col.upper() for col in df.columns]
34        if "OVERALL_RATING" in df.columns:
35            df["OVERALL_RATING"] = pd.to_numeric(df["OVERALL_RATING"],
36                                                errors="coerce")
37        if "RMP_RATING" in df.columns:
38            df["RMP_RATING"] = pd.to_numeric(df["RMP_RATING"], errors="coerce")
39        if "DIFFICULTY" in df.columns:
40            df["DIFFICULTY"] = pd.to_numeric(df["DIFFICULTY"], errors="coerce")
41        if "WOULD TAKE AGAIN" in df.columns:
42            df["WOULD TAKE AGAIN"] = pd.to_numeric(df["WOULD TAKE AGAIN"],
43                                                    errors="coerce")
44            df["WOULD TAKE AGAIN"] = df["WOULD TAKE AGAIN"].apply(lambda x: x /
45                                                    100 if pd.notna(x) and x > 1 else x)
```

```

43     print("Cleaned data sample:")
44     print(df.head())
45     success, nchunks, nrows, _ = write_pandas(conn, df, target_table,
46         ↳ quote_identifiers=True)
47     if success:
48         print(f"Loaded {nrows} rows into table {target_table}")
49     else:
50         print("Failed to write data to Snowflake.")
51     finally:
52         cursor.close()
53         conn.close()
54     return df
55
56 if __name__ == "__main__":
57     sf_conn_params = {
58         "account": "SFEDU02-IOB55870",
59         "user": "BISON",
60         "password": "voshim-bopcyt-3virBe",
61         "role": "TRAINING_ROLE",
62         "warehouse": "BISON_WH",
63         "database": "BISON_DB",
64         "schema": "AIRFLOW",
65     }
66     clean_and_store_data("RMP_ENRICHED_DATA", "MY_WASHU_COURSES_WITH_RATINGS",
67         ↳ sf_conn_params)

```

## 6.6 app\_data\_pipeline.py (DAG)

```

1  from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
2  from airflow.providers.common.sql.sensors.sql import SqlSensor
3  from airflow.operators.python import PythonOperator
4  from airflow import DAG
5  from airflow.decorators import dag
6  from datetime import datetime, timedelta
7
8  from process_pdfs import process_pdfs_from_snowflake
9  from scrape_professor_ratings import scrape_professor_ratings_table
10 from clean_and_store_data import clean_and_store_data
11
12 default_args = {
13     'owner': 'airflow',
14     'depends_on_past': False,
15     'start_date': datetime.now() - timedelta(days=1),
16     'retries': 1,
17     'retry_delay': timedelta(minutes=5),

```

```

18 }
19
20 sf_conn_params = {
21     "account": "SFEDU02-I0B55870",
22     "user": "BISON",
23     "password": "voshim-bopcyt-3virBe",
24     "role": "TRAINING_ROLE",
25     "warehouse": "BISON_WH",
26     "database": "BISON_DB",
27     "schema": "AIRFLOW",
28 }
29
30 @dag(
31     default_args=default_args,
32     schedule='@daily',
33     catchup=False,
34     dag_id="app_data_pipeline"
35 )
36 def app_data_pipeline():
37     create_signal_table = SQLExecuteQueryOperator(
38         task_id='create_signal_table',
39         conn_id='SnowFlakeConn',
40         sql="""  

41             CREATE OR REPLACE TABLE SIGNAL_TABLE (
42                 TABLE_NAME STRING,
43                 STATUS STRING,
44                 TIMESTAMP TIMESTAMP_LTZ
45             );
46             """  

47     )
48     insert_dummy_signal = SQLExecuteQueryOperator(
49         task_id='insert_dummy_signal',
50         conn_id='SnowFlakeConn',
51         sql="""  

52             INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
53             VALUES ('APP_STG', 'success', CURRENT_TIMESTAMP());
54             """  

55     )
56     check_app_stg_signal = SqlSensor(
57         task_id='check_app_stg_signal',
58         conn_id='SnowFlakeConn',
59         sql="SELECT 1 FROM SIGNAL_TABLE WHERE TABLE_NAME = 'APP_STG' AND STATUS =
60             'success'",
61         timeout=600,
62         poke_interval=30,
63         mode='reschedule'
64     )

```

```

64     create_target_table = SQLExecuteQueryOperator(
65         task_id='create_target_table',
66         conn_id='SnowFlakeConn',
67         sql="""
68             CREATE OR REPLACE TABLE MY_WASHU_COURSES_WITH_RATINGS (
69                 FILENAME VARCHAR,
70                 PROFESSOR VARCHAR,
71                 SEMESTER VARCHAR,
72                 COURSE_CODE VARCHAR,
73                 SECTION VARCHAR,
74                 COURSE_NAME VARCHAR,
75                 OVERALL_RATING FLOAT,
76                 RMP_ID NUMBER(38,0),
77                 SCRAPED_OVERALL_RATING FLOAT,
78                 WOULD_TAKE AGAIN FLOAT,
79                 DIFFICULTY FLOAT,
80                 TAGS VARCHAR
81             );
82             """
83     )
84     def run_process_pdfs():
85         stage_name = "APP_STG"
86         df = process_pdfs_from_snowflake(stage_name)
87         if df.empty:
88             print("No PDF data extracted; skipping further steps.")
89             return
90         print("Processed PDF data stored in table APP_PROCESSED_PDFS")
91     process_pdfs_task = PythonOperator(
92         task_id='process_pdfs',
93         python_callable=run_process_pdfs,
94     )
95     def run_scrape_rmp():
96         input_table = "APP_PROCESSED_PDFS"
97         output_table = "RMP_ENRICHED_DATA"
98         scrape_professor_ratings_table(input_table, output_table, sf_conn_params)
99         print("Scraped RMP data stored in table RMP_ENRICHED_DATA")
100    scrape_rmp_task = PythonOperator(
101        task_id='scrape_rmp',
102        python_callable=run_scrape_rmp,
103    )
104    def run_clean_and_store():
105        clean_and_store_data("RMP_ENRICHED_DATA", "MY_WASHU_COURSES_WITH_RATINGS",
106                             sf_conn_params)
107        print("Enriched data loaded into table MY_WASHU_COURSES_WITH_RATINGS")
108    load_enriched_data_task = PythonOperator(
109        task_id='load_enriched_data',
110        python_callable=run_clean_and_store,

```

```

110 )
111 standardization_sql = """
112 CREATE OR REPLACE TABLE WASHU_COURSES_STANDARDIZED AS
113 WITH normalized_data AS (
114     SELECT
115         TRIM(PROFESSOR) AS ORIGINAL_PROFESSOR_NAME,
116         SPLIT_PART(TRIM(PROFESSOR), ' ', -1) AS PROFESSOR_NAME,
117         TRIM(COURSE_CODE) AS COURSE_CODE,
118         TRIM(COURSE_NAME) AS COURSE_NAME,
119         TRIM(SEMESTER) AS SEMESTER,
120         TRIM(SECTION) AS SECTION,
121         RMP_ID,
122         TO_NUMBER(OVERALL_RATING) AS OVERALL_RATING_NUM,
123         TO_NUMBER(SCRAPED_OVERALL_RATING) AS SCRAPED_OVERALL_RATING_NUM,
124         CASE
125             WHEN TO_NUMBER(WOULD_TAKE AGAIN) > 1 THEN
126                 ↪ TO_NUMBER(WOULD_TAKE AGAIN)/100
127             ELSE TO_NUMBER(WOULD_TAKE AGAIN)
128         END AS WOULD_TAKE AGAIN_NUM,
129         TO_NUMBER(DIFFICULTY) AS DIFFICULTY_NUM,
130         TRIM(TAGS) AS TAGS,
131         ROW_NUMBER() OVER (
132             PARTITION BY
133                 SPLIT_PART(TRIM(PROFESSOR), ' ', -1),
134                 TRIM(COURSE_CODE),
135                 TRIM(SEMESTER),
136                 TRIM(SECTION)
137             ORDER BY
138                 CASE WHEN TO_NUMBER(OVERALL_RATING) IS NOT NULL THEN 0 ELSE 1
139                     ↪ END,
140                 CASE WHEN TO_NUMBER(SCRAPED_OVERALL_RATING) IS NOT NULL THEN
141                     ↪ 0 ELSE 1 END,
142                 CASE WHEN TRIM(COURSE_NAME) IS NOT NULL THEN 0 ELSE 1 END
143             ) AS ROW_RANK
144     FROM MY_WASHU_COURSES_WITH_RATINGS
145 )
146     SELECT
147         PROFESSOR_NAME,
148         ORIGINAL_PROFESSOR_NAME AS FULL_PROFESSOR_NAME,
149         COURSE_CODE,
150         COURSE_NAME,
151         SEMESTER,
152         SECTION,
153         RMP_ID,
154         OVERALL_RATING_NUM AS OVERALL_RATING,
155         SCRAPED_OVERALL_RATING_NUM AS SCRAPED_OVERALL_RATING,
156         WOULD_TAKE AGAIN_NUM AS WOULD_TAKE AGAIN,

```

```

154     DIFFICULTY_NUM AS DIFFICULTY,
155     TAGS,
156     CASE
157         WHEN SCRAPED_OVERALL_RATING_NUM IS NOT NULL THEN
158             → (SCRAPED_OVERALL_RATING_NUM - 1) * (6/4) + 1
159         ELSE NULL
160     END AS NORMALIZED_RMP_RATING
161 FROM normalized_data
162 WHERE ROW_RANK = 1;
163
164 CREATE OR REPLACE VIEW V_WASHU_COURSES AS
165 SELECT * FROM WASHU_COURSES_STANDARDIZED;
166 """
167 standardization_task = SQLExecuteQueryOperator(
168     task_id='run_standardization',
169     conn_id='SnowFlakeConn',
170     sql=standardization_sql,
171 )
172 insert_signal_final = SQLExecuteQueryOperator(
173     task_id='insert_signal_final',
174     conn_id='SnowFlakeConn',
175     sql="""
176         INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
177         VALUES ('WASHU_COURSES_STANDARDIZED', 'success',
178             → CURRENT_TIMESTAMP());
179         """,
180         )
181         create_signal_table >> insert_dummy_signal >> check_app_stg_signal >>
182         process_pdffs_task
183         process_pdffs_task >> scrape_rmp_task >> create_target_table >>
184         load_enriched_data_task >> standardization_task >> insert_signal_final
185
186 app_data_pipeline_dag = app_data_pipeline()

```

## 6.7 snowflake\_recommendation\_api.py (Flask API)

```

1 from flask import Flask, request, jsonify
2 import pandas as pd
3 import numpy as np
4 import json
5 import re
6 import snowflake.connector
7 from datetime import datetime
8 import os
9 import csv
10

```

```

11 app = Flask(__name__)
12
13 def safe_float(value, default=float('nan')):
14     if value is None or value == '':
15         return default
16     try:
17         return float(value)
18     except (ValueError, TypeError):
19         return default
20
21 SNOWFLAKE_USER = "BISON"
22 SNOWFLAKE_PASSWORD = "voshim-bopcyt-3virBe"
23 SNOWFLAKE_ACCOUNT = "SFEDU02-IOB55870"
24 SNOWFLAKE_WAREHOUSE = "BISON_WH"
25 SNOWFLAKE_DATABASE = "BISON_DB"
26 SNOWFLAKE_SCHEMA = "PUBLIC"
27
28 def connect_to_snowflake():
29     try:
30         conn = snowflake.connector.connect(
31             user=SNOWFLAKE_USER,
32             password=SNOWFLAKE_PASSWORD,
33             account=SNOWFLAKE_ACCOUNT,
34             warehouse=SNOWFLAKE_WAREHOUSE,
35             database=SNOWFLAKE_DATABASE,
36             schema=SNOWFLAKE_SCHEMA
37         )
38         return conn
39     except Exception as e:
40         print(f"Error connecting to Snowflake: {str(e)}")
41         return None
42
43 def load_data_from_snowflake():
44     conn = connect_to_snowflake()
45     if conn:
46         try:
47             cursor = conn.cursor()
48             cursor.execute("SELECT * FROM V_WASHU_COURSES")
49             column_names = [desc[0] for desc in cursor.description]
50             rows = cursor.fetchall()
51             df = pd.DataFrame(rows, columns=column_names)
52             cursor.close()
53             conn.close()
54             numeric_cols = ['STANDARDIZED_RATING', 'STANDARDIZED_DIFFICULTY',
55                             'STANDARDIZED_WOULD_TAKE AGAIN']
56             for col in numeric_cols:
57                 if col in df.columns:
58                     df[col] = pd.to_numeric(df[col], errors='coerce')
59             return df
60         except Exception as e:
61             print(f"Error loading data from Snowflake: {str(e)}")
62             return load_data_from_csv()
63     else:

```

```

63     return load_data_from_csv()
64
65 def load_data_from_csv():
66     try:
67         df = pd.read_csv("DataBase-2.csv")
68         numeric_cols = ['STANDARDIZED_RATING', 'STANDARDIZED_DIFFICULTY',
69                         'STANDARDIZED_WOULD_TAKE AGAIN']
70         for col in numeric_cols:
71             if col in df.columns:
72                 df[col] = pd.to_numeric(df[col], errors='coerce')
73         return df
74     except Exception as e:
75         print(f"Error loading data from CSV: {str(e)}")
76         return pd.DataFrame()
77
78 df = load_data_from_snowflake()
79
80 @app.route('/search', methods=['GET'])
81 def search():
82     try:
83         course_code = request.args.get('course_code', '')
84         course_name = request.args.get('course_name', '')
85         professor_name = request.args.get('professor_name', '')
86         min_rating = request.args.get('min_rating', '')
87         max_difficulty = request.args.get('max_difficulty', '')
88         tags = request.args.get('tags', '')
89
90         filtered_df = df.copy()
91         if course_code:
92             filtered_df =
93                 filtered_df[filtered_df['COURSE_CODE'].str.contains(course_code,
94                             case=False, na=False)]
95         if course_name:
96             filtered_df =
97                 filtered_df[filtered_df['COURSE_NAME'].str.contains(course_name,
98                             case=False, na=False)]
99         if professor_name:
100             filtered_df =
101                 filtered_df[filtered_df['PROFESSOR_NAME'].str.contains(professor_name,
102                             case=False, na=False)]
103         if min_rating:
104             try:
105                 min_rating_float = safe_float(min_rating)
106                 if not np.isnan(min_rating_float):
107                     filtered_df = filtered_df[filtered_df['STANDARDIZED_RATING'] >=
108                                     min_rating_float]
109             except (ValueError, TypeError):
110                 pass
111         if max_difficulty:
112             try:
113                 max_difficulty_float = safe_float(max_difficulty)
114                 if not np.isnan(max_difficulty_float):
115                     filtered_df = filtered_df[filtered_df['STANDARDIZED_DIFFICULTY'] <=
116                                     max_difficulty_float]
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1288
1289
1290
1291
1292
1293
1294
1295
1296
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1394
1395
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1588
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1688
1689
1690
1691
1692
1693
1694
1695
1695
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1795
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1888
1889
1890
1891
1892
1893
1894
1895
1895
1896
1897
1898
1899
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1988
1989
1990
1991
1992
1993
1994
1995
1995
1996
1997
1998
1999
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2088
2089
2090
2091
2092
2093
2094
2095
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2188
2189
2190
2191
2192
2193
2194
2195
2195
2196
2197
2198
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2278
2279
2280
2281
2
```

```

108         except (ValueError, TypeError):
109             pass
110     if tags:
111         tag_list = tags.split(',')
112         for tag in tag_list:
113             filtered_df = filtered_df[filtered_df['TAGS'].str.contains(tag.strip(),
114             ↳ case=False, na=False)]
115
116     results = filtered_df.fillna('').to_dict(orient='records')
117     return jsonify({
118         "status": "success",
119         "results": results
120     })
121 except Exception as e:
122     return jsonify({
123         "status": "error",
124         "error": str(e)
125     })
126
127 @app.route('/submit_review', methods=['POST'])
128 def submit_review():
129     try:
130         review_data = request.json
131         if not review_data:
132             return jsonify({"status": "error", "error": "No review data provided."})
133         required_fields = ['course_code', 'professor_name']
134         for field in required_fields:
135             if field not in review_data or not review_data[field]:
136                 return jsonify({"status": "error", "error": f"{field} is required."})
137         try:
138             if 'rating' in review_data:
139                 try:
140                     review_data['rating'] = safe_float(review_data['rating'])
141                 except ValueError:
142                     review_data['rating'] = None
143             if 'difficulty' in review_data:
144                 review_data['difficulty'] = safe_float(review_data['difficulty'])
145             if 'would_take_again' in review_data:
146                 review_data['would_take_again'] =
147                     ↳ safe_float(review_data['would_take_again'])
148         except Exception as e:
149             return jsonify({"status": "error", "error": f"Invalid numeric value:
150             ↳ {str(e)}"})
151         review_data['timestamp'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
152         reviews_file = "reviews.csv"
153         headers = ['course_code', 'course_name', 'professor_name', 'semester',
154             ↳ 'section',
155                 'rating', 'difficulty', 'would_take_again', 'tags', 'comments',
156                 ↳ 'timestamp']
157         file_exists = os.path.exists(reviews_file)
158         with open(reviews_file, 'a', newline='', encoding='utf-8') as csvfile:
159             writer = csv.DictWriter(csvfile, fieldnames=headers)
160             if not file_exists:

```

```

156         writer.writeheader()
157         sanitized_data = {
158             key: str(review_data.get(key, '')).replace(',', ';') for key in headers
159         }
160         writer.writerow(sanitized_data)
161     return jsonify({"status": "success", "message": "Review submitted
162     ↵ successfully."})
163 except Exception as e:
164     return jsonify({"status": "error", "error": str(e)})
165
166 if __name__ == '__main__':
167     app.run(debug=True)

```

## 6.8 course\_recommendation\_app.py (Streamlit App)

```

1 import streamlit as st
2 import pandas as pd
3 import numpy as np
4 import requests
5 import json
6 import plotly.graph_objects as go
7 from datetime import datetime
8
9 def safe_float(value, default=float('nan')):
10     if value is None or value == '':
11         return default
12     try:
13         return float(value)
14     except (ValueError, TypeError):
15         return default
16
17 API_ENDPOINT = "http://127.0.0.1:5000"
18
19 st.set_page_config(
20     page_title="Course Recommendation System",
21     page_icon="",
22     layout="wide",
23     initial_sidebar_state="expanded"
24 )
25
26 st.markdown('''
27 <style>
28     .main-header {
29         font-size: 2.5rem;
30         color: #A51417;
31         text-align: center;
32         margin-bottom: 1rem;
33     }
34     .sub-header {
35         font-size: 1.5rem;

```

```

36     color: #A51417;
37     margin-top: 2rem;
38     margin-bottom: 1rem;
39   }
40   .card {
41     border-radius: 5px;
42     background-color: #f9f9f9;
43     padding: 1.5rem;
44     margin-bottom: 1rem;
45     border-left: 5px solid #A51417;
46   }
47   .sentiment-tag {
48     display: inline-block;
49     background-color: #E5E7EB;
50     color: #1F2937;
51     padding: 0.25rem 0.5rem;
52     border-radius: 9999px;
53     margin-right: 0.5rem;
54     margin-bottom: 0.5rem;
55     font-size: 0.875rem;
56   }
57   .rating-container {
58     display: flex;
59     align-items: center;
60     margin-bottom: 0.5rem;
61   }
62   .rating-label {
63     min-width: 150px;
64     font-weight: bold;
65   }
66   .rating-value {
67     margin-left: 1rem;
68   }
69 </style>
70 ''', unsafe_allow_html=True)
71
72 st.markdown('<h1 class="main-header">Washington University in St. Louis Course  

73   ↪ Recommendation System</h1>', unsafe_allow_html=True)
74
75 st.sidebar.image("https://marcomm.washu.edu/app/uploads/2024/02/WashU-SHIELD-Red_RGB.jpg",
76   ↪ g",
77   ↪ width=150)
78 st.sidebar.title("Filters & Preferences")
79 major = st.sidebar.text_input("Your Major", "Computer Science")
80 year = st.sidebar.selectbox("Year", [1, 2, 3, 4])
81 st.sidebar.markdown("### What matters to you?")
82 st.sidebar.markdown("Select preferences that are important to you:")
83 importance_rating = st.sidebar.slider("Overall Rating Importance", 0, 10, 8)
84 importance_difficulty = st.sidebar.slider("Low Difficulty Importance", 0, 10, 5)
85 importance_would_take_again = st.sidebar.slider("Would Take Again Importance", 0, 10, 7)
86 sentiment_options =
87   ["accessible outside class", "amazing lectures", "caring",
88    "clear grading criteria", "extra credit", "get ready to read",

```

```

86     "gives good feedback", "graded by few things", "group projects",
87     "helpful", "inspirational", "lecture heavy", "lots of homework",
88     "participation matters", "respected", "skip class? you won't pass",
89     "so many papers", "test heavy", "tough grader"
90 ]
91
92 tab1, tab2 = st.tabs(["Find Courses", "Submit Review"])
93
94 with tab1:
95     st.markdown('<h2 class="sub-header">Find Your Ideal Courses</h2>',
96                  unsafe_allow_html=True)
97     search_col1, search_col2 = st.columns(2)
98     with search_col1:
99         search_by = st.radio("Search by:", ["Course", "Professor", "Rating", "Advanced
100             Search"])
101     with search_col2:
102         if search_by == "Course":
103             search_term = st.text_input("Enter Course Code or Name:")
104         elif search_by == "Professor":
105             search_term = st.text_input("Enter Professor Name:")
106         elif search_by == "Rating":
107             min_rating = st.slider("Minimum Rating", 1.0, 7.0, 4.0, 0.1)
108             search_term = str(min_rating)
109         else:
110             st.write("Use the filters below for advanced search")
111             search_term = ""
112     if search_by == "Advanced Search":
113         adv_col1, adv_col2, adv_col3 = st.columns(3)
114         with adv_col1:
115             course_code = st.text_input("Course Code:")
116             course_name = st.text_input("Course Name:")
117         with adv_col2:
118             professor_name = st.text_input("Professor Name:")
119             min_rating = st.slider("Minimum Rating", 1.0, 7.0, 3.0, 0.1)
120         with adv_col3:
121             max_difficulty = st.slider("Maximum Difficulty", 1.0, 7.0, 5.0, 0.1)
122             selected_tags = st.multiselect("Course Tags:", sentiment_options)
123     if st.button("Search Courses"):
124         try:
125             if search_by == "Course":
126                 if search_term.isdigit() or (len(search_term) <= 5 and any(c.isdigit()
127                     for c in search_term)):
128                     response = requests.get(f"{API_ENDPOINT}/search",
129                     params={"course_code": search_term})
130                 else:
131                     response = requests.get(f"{API_ENDPOINT}/search",
132                     params={"course_name": search_term})
133             elif search_by == "Professor":
134                 response = requests.get(f"{API_ENDPOINT}/search",
135                     params={"professor_name": search_term})
136             elif search_by == "Rating":
137                 response = requests.get(f"{API_ENDPOINT}/search", params={"min_rating":'
138                     search_term})

```

```

132
133     else:
134         params = {}
135         if course_code:
136             params["course_code"] = course_code
137         if course_name:
138             params["course_name"] = course_name
139         if professor_name:
140             params["professor_name"] = professor_name
141         if min_rating:
142             params["min_rating"] = str(min_rating)
143         if max_difficulty:
144             params["max_difficulty"] = str(max_difficulty)
145         if selected_tags:
146             params["tags"] = ",".join(selected_tags)
147         response = requests.get(f"{API_ENDPOINT}/search", params=params)
148     if response.status_code == 200:
149         result = response.json()
150         if result.get("status") == "success":
151             results = result.get("results", [])
152             if not results:
153                 st.warning("No courses found matching your criteria.")
154             else:
155                 st.success(f"Found {len(results)} courses matching your
156                             ↪ criteria.")
157                 for course in results:
158                     rating_raw = course.get("STANDARDIZED_RATING", '0')
159                     difficulty_raw = course.get("STANDARDIZED_DIFFICULTY",
160                                         ↪ '3.5')
161                     would_take_again_raw =
162                         ↪ course.get("STANDARDIZED_WOULD_TAKE AGAIN", '0.5')
163                     rating = safe_float(rating_raw) if rating_raw not in [None,
164                                         ↪ '' ] else 0.0
165                     difficulty = safe_float(difficulty_raw) if difficulty_raw
166                                         ↪ not in [None, '' ] else 3.5
167                     would_take_again = safe_float(would_take_again_raw) if
168                         ↪ would_take_again_raw not in [None, '' ] else 0.5
169                     rating_score = rating * importance_rating / 10
170                     difficulty_score = (7 - difficulty) * importance_difficulty
171                         ↪ / 10
172                     would_take_again_score = would_take_again * 7 *
173                         ↪ importance_would_take_again / 10
174                     course["RECOMMENDATION_SCORE"] = rating_score +
175                         ↪ difficulty_score + would_take_again_score
176                     results.sort(key=lambda x: x.get("RECOMMENDATION_SCORE", 0),
177                         ↪ reverse=True)
178                     for i, course in enumerate(results):
179                         with st.container():
180                             st.markdown(f'''
181                                 <div class="card">
182                                     <h3>{course.get('COURSE_CODE', 'N/A')} -
183                                         ↪ {course.get('COURSE_NAME', 'N/A')}</h3>
184                                     <p><strong>Professor:</strong>
185                                         ↪ {course.get('PROFESSOR_NAME', 'N/A')}</p>

```

```

173                                     <p><strong>Rating:</strong> {safe_float(course. ]
174                                         ← get('STANDARDIZED_RATING', ''),
175                                         ← 0) : .1f} / 7</p>
176                                     <p><strong>Difficulty:</strong> {safe_float(cou
177                                         ← rse.get('STANDARDIZED_DIFFICULTY', ''),
178                                         ← 0) : .1f} / 7</p>
179                                     <p><strong>Would Take Again:</strong>
180                                         ← {safe_float(course.get('STANDARDIZED_WOULD_]
181                                         ← TAKE AGAIN', ''), 0.0) *
182                                         ← 100 : .0f} %</p>
183                                     </div>
184                                 '''', unsafe_allow_html=True)
185
186             else:
187                 st.error("Failed to fetch data from server.")
188
189         else:
190             st.error("Server error. Please check backend.")
191     except Exception as e:
192         st.error(f"Error during search: {e}")

```

## 7 Conclusion

This document describes a fully integrated system for course recommendation that consists of a data pipeline (implemented in Python using Snowflake and Apache Airflow) and a web application (built with Flask and Streamlit). The pipeline extracts data from PDFs, enriches it with professor ratings scraped from RateMyProfessor, cleans and stores the data in a standardized schema in Snowflake, and creates a view for querying. The Flask API exposes endpoints that the Streamlit app uses for interactive course search and recommendation. This modular design ensures both robust data processing and real-time user interaction.

Future work may include additional data manipulation, improved validation, and more sophisticated recommendation algorithms as well as full integration of review submission directly into the web app.