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.

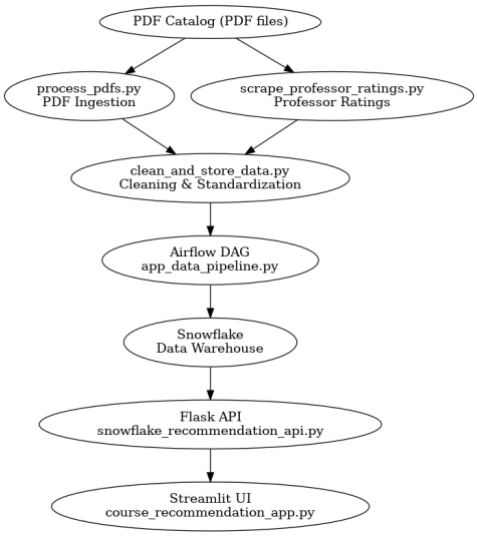

Figure 1: CourseCompass System Architecture

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:

```
# Define the DAG
dag = DAG(
    'course_data_pipeline',
    default_args={
        'owner': 'airflow',
        'depends_on_past': False,
```

```
    'start_date': datetime(2023, 1, 1),
    'email_on_failure': True,
    'retries': 1,
},
schedule_interval='@daily',
catchup=False
)

# Define tasks
process_pdfs_task = PythonOperator(
    task_id='process_pdfs',
    python_callable=process_pdfs_from_snowflake,
    dag=dag
)

scrape_ratings_task = PythonOperator(
    task_id='scrape_professor_ratings',
    python_callable=scrape_professor_ratings_table,
    dag=dag
)

clean_data_task = PythonOperator(
    task_id='clean_and_store_data',
    python_callable=clean_and_store_data,
    dag=dag
)

# Define task dependencies
process_pdfs_task >> clean_data_task
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:

```
@app.route('/api/courses', methods=['GET'])
def get_courses():
    # Extract query parameters
    department = request.args.get('department')
    level = request.args.get('level')
    professor = request.args.get('professor')

    # Construct SQL query with parameters
    query = """
    SELECT * FROM courses
    WHERE 1=1
    """
    params = {}

    if department:
        query += " AND department = %(department)s"
        params['department'] = department

    if level:
        query += " AND level = %(level)s"
        params['level'] = level

    if professor:
        query += " AND professor = %(professor)s"
        params['professor'] = professor

    # Execute query against Snowflake
    conn = snowflake.connector.connect(**SNOWFLAKE_CONN_PARAMS)
    cursor = conn.cursor()
    cursor.execute(query, params)
    results = cursor.fetchall()

    # Transform results to JSON
    columns = [col[0] for col in cursor.description]
    courses = [dict(zip(columns, row)) for row in results]

    return jsonify(courses)

```

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
```

```
if search_query or department != "All" or level != "All":
    results = search_courses(
        search_query=search_query,
        department=department if department != "All" else None,
        level=level if level != "All" else None,
        min_rating=min_rating
    )

    # Display results
    for course in results:
        with st.expander(f"{course['code']} - {course['title']}"):
            st.write(f"**Department:** {course['department']}")
            st.write(f"**Professor:** {course['professor']}")
            st.write(f"**Rating:** {course['rating']}/5.0")
            st.write(f"**Description:** {course['description']}")

```

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.

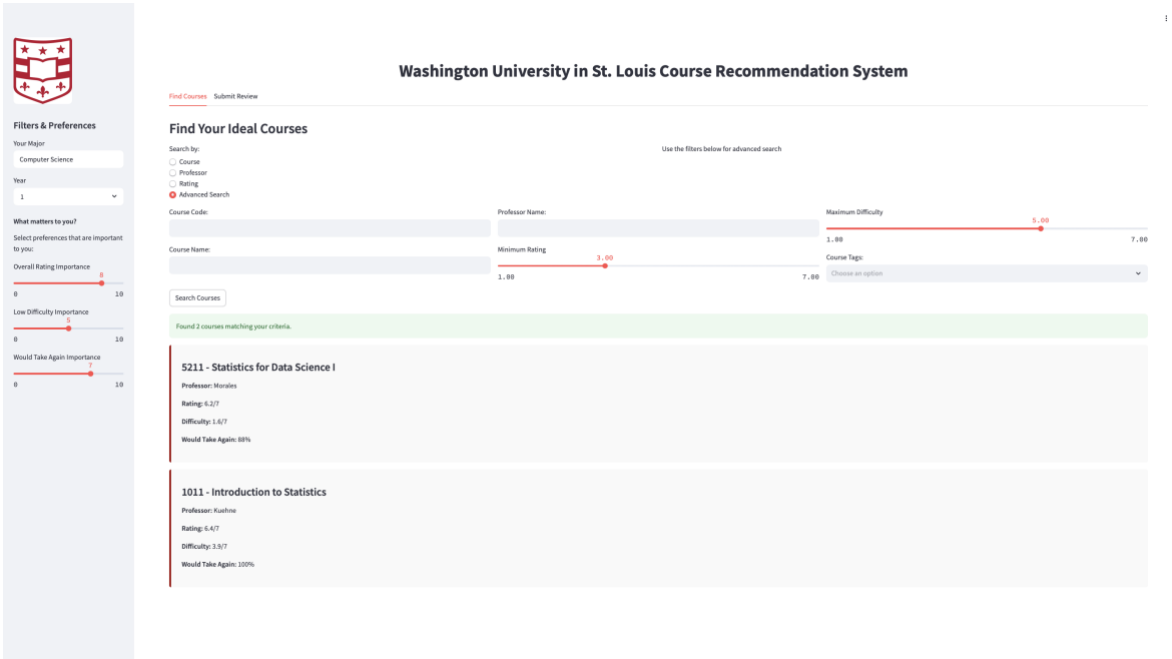

Figure 3: Recommendation API Frontend

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[†]
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.

[†] 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

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:

```
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from airflow.providers.common.sql.sensors.sql import SqlSensor
from airflow.operators.python import PythonOperator
from airflow import DAG
from airflow.decorators import dag
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());
```

```
        """,
    )
    check_app_stg_signal = SqlSensor(
        task_id='check_app_stg_signal',
        conn_id='SnowFlakeConn',
        sql="SELECT 1 FROM SIGNAL_TABLE WHERE TABLE_NAME = 'APP_STG' AND STATUS =
        ↪  'success'",
        timeout=600,
        poke_interval=30,
        mode='reschedule'
    )
    create_target_table = SQLExecuteQueryOperator(
        task_id='create_target_table',
        conn_id='SnowFlakeConn',
        sql="""
            CREATE OR REPLACE TABLE MY_WASHU_COURSES_WITH_RATINGS (
                FILENAME VARCHAR,
                PROFESSOR VARCHAR,
                SEMESTER VARCHAR,
                COURSE_CODE VARCHAR,
                SECTION VARCHAR,
                COURSE_NAME VARCHAR,
                OVERALL_RATING FLOAT,
                RMP_ID NUMBER(38,0),
                SCRAPED_OVERALL_RATING FLOAT,
                WOULD_TAKE_AGAIN FLOAT,
                DIFFICULTY FLOAT,
                TAGS VARCHAR
            );
        """,
    )
    def run_process_pdfs():
        stage_name = "APP_STG"
        df = process_pdfs_from_snowflake(stage_name)
        if df.empty:
            print("No PDF data extracted; skipping further steps.")
            return
        print("Processed PDF data stored in table APP_PROCESSED_PDFS")
    process_pdfs_task = PythonOperator(
        task_id='process_pdfs',
        python_callable=run_process_pdfs,
    )
    def run_scrape_rmp():
        input_table = "APP_PROCESSED_PDFS"
        output_table = "RMP_ENRICHED_DATA"
        scrape_professor_ratings_table(input_table, output_table, sf_conn_params)
        print("Scraped RMP data stored in table RMP_ENRICHED_DATA")
```

```
    scrape_rmp_task = PythonOperator(
        task_id='scrape_rmp',
        python_callable=run_scrape_rmp,
    )
    def run_clean_and_store():
        clean_and_store_data("RMP_ENRICHED_DATA", "MY_WASHU_COURSES_WITH_RATINGS",
        ↪  sf_conn_params)
        print("Enriched data loaded into table MY_WASHU_COURSES_WITH_RATINGS")
    load_enriched_data_task = PythonOperator(
        task_id='load_enriched_data',
        python_callable=run_clean_and_store,
    )
    standardization_sql = """
    CREATE OR REPLACE TABLE WASHU_COURSES_STANDARDIZED AS
    WITH normalized_data AS (
        SELECT
            TRIM(PROFESSOR) AS ORIGINAL_PROFESSOR_NAME,
            SPLIT_PART(TRIM(PROFESSOR), ' ', -1) AS PROFESSOR_NAME,
            TRIM(COURSE_CODE) AS COURSE_CODE,
            TRIM(COURSE_NAME) AS COURSE_NAME,
            TRIM(SEMESTER) AS SEMESTER,
            TRIM(SECTION) AS SECTION,
            RMP_ID,
            TO_NUMBER(OVERALL_RATING) AS OVERALL_RATING_NUM,
            TO_NUMBER(SCRAPED_OVERALL_RATING) AS SCRAPED_OVERALL_RATING_NUM,
            CASE
                WHEN TO_NUMBER(WOULD_TAKE_AGAIN) > 1 THEN
                ↪  TO_NUMBER(WOULD_TAKE_AGAIN)/100
                ELSE TO_NUMBER(WOULD_TAKE_AGAIN)
            END AS WOULD_TAKE_AGAIN_NUM,
            TO_NUMBER(DIFFICULTY) AS DIFFICULTY_NUM,
            TRIM(TAGS) AS TAGS,
            ROW_NUMBER() OVER (
                PARTITION BY
                    SPLIT_PART(TRIM(PROFESSOR), ' ', -1),
                    TRIM(COURSE_CODE),
                    TRIM(SEMESTER),
                    TRIM(SECTION)
                ORDER BY
                    CASE WHEN TO_NUMBER(OVERALL_RATING) IS NOT NULL THEN 0 ELSE 1
                    ↪  END,
                    CASE WHEN TO_NUMBER(SCRAPED_OVERALL_RATING) IS NOT NULL THEN
                    ↪  0 ELSE 1 END,
                    CASE WHEN TRIM(COURSE_NAME) IS NOT NULL THEN 0 ELSE 1 END
            ) AS ROW_RANK
        FROM MY_WASHU_COURSES_WITH_RATINGS
    )
```

```
    SELECT
        PROFESSOR_NAME,
        ORIGINAL_PROFESSOR_NAME AS FULL_PROFESSOR_NAME,
        COURSE_CODE,
        COURSE_NAME,
        SEMESTER,
        SECTION,
        RMP_ID,
        OVERALL_RATING_NUM AS OVERALL_RATING,
        SCRAPED_OVERALL_RATING_NUM AS SCRAPED_OVERALL_RATING,
        WOULD_TAKE_AGAIN_NUM AS WOULD_TAKE_AGAIN,
        DIFFICULTY_NUM AS DIFFICULTY,
        TAGS,
        CASE
            WHEN SCRAPED_OVERALL_RATING_NUM IS NOT NULL THEN
            ↪  (SCRAPED_OVERALL_RATING_NUM - 1) * (6/4) + 1
            ELSE NULL
        END AS NORMALIZED_RMP_RATING
    FROM normalized_data
    WHERE ROW_RANK = 1;

    CREATE OR REPLACE VIEW V_WASHU_COURSES AS
    SELECT * FROM WASHU_COURSES_STANDARDIZED;
    """
    standardization_task = SQLExecuteQueryOperator(
        task_id='run_standardization',
        conn_id='SnowFlakeConn',
        sql=standardization_sql,
    )
    insert_signal_final = SQLExecuteQueryOperator(
        task_id='insert_signal_final',
        conn_id='SnowFlakeConn',
        sql="""
            INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
            VALUES ('WASHU_COURSES_STANDARDIZED', 'success',
            ↪  CURRENT_TIMESTAMP());
        """,
    )
    create_signal_table >> insert_dummy_signal >> check_app_stg_signal >>
    ↪  process_pdfs_task
    process_pdfs_task >> scrape_rmp_task >> create_target_table >>
    ↪  load_enriched_data_task >> standardization_task >> insert_signal_final

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

```
import requests
from bs4 import BeautifulSoup
import csv
import re

def scrape_courses(url):
    """
    Scrape course information from the given URL

    Args:
        url (str): URL of the course listing page
```

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

```
            # Check if there's an "Instructors:" label
            if "Instructors:" in instructor_text:
                instructors = instructor_text.split("Instructors:")[1].strip()

        # Alternative method for instructor extraction
        if instructors == "N/A":
            details_div = article.find('div', class_='details')
            if details_div:
                instructor_info = details_div.find(text=re.compile('INSTRUCTORS:',
                ↪  re.IGNORECASE))
                if instructor_info:
                    instructor_parent = instructor_info.parent
                    if instructor_parent:
                        instructors =
                        ↪  instructor_parent.text.replace('INSTRUCTORS:',
                        ↪  '').strip()

        # Split instructors if there are commas and create separate entries
        if instructors != "N/A" and "," in instructors:
            instructor_list = [instr.strip() for instr in instructors.split(",")]
            for instructor in instructor_list:
                courses.append({
                    'course_code': course_code,
                    'course_name': course_name,
                    'instructors': instructor
                })
        else:
            courses.append({
                'course_code': course_code,
                'course_name': course_name,
                'instructors': instructors
            })

    return courses

def save_to_csv(courses, filename):
    """
    Save course information to a CSV file

    Args:
        courses (list): List of dictionaries containing course information
        filename (str): Name of the CSV file
    """
    with open(filename, 'w', newline='', encoding='utf-8') as csvfile:
        fieldnames = ['course_code', 'course_name', 'instructors']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

```

```
        writer.writeheader()
        for course in courses:
            writer.writerow(course)

    print(f"Data saved to {filename}")

def main():
    # URL of the course listing page
    url = "https://sds.wustl.edu/course_listing"  # Replace with actual URL

    # Scrape course information
    courses = scrape_courses(url)

    # Save to CSV
    save_to_csv(courses, "washu_data_science_courses.csv")

    print(f"Scraped {len(courses)} courses")

if __name__ == "__main__":
    main()


```

6.2 Course Evaluation PDF Extractor

This notebook extracts course information from course evaluation PDFs.

```
# Uncomment and run this cell if packages need to be installed
# !pip install pdfplumber pandas tqdm matplotlib seaborn

import pdfplumber
import re
import os
import pandas as pd
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
```

6.2.1 Define Functions

```
# Function to extract course information from a PDF
def extract_course_info(pdf_path):
    """
    Extract course information from a course evaluation PDF.
```

```
    Args:
        pdf_path (str): Path to the PDF file
    Returns:
        dict: Dictionary containing course information
    """
    try:
        # Extract basic info from filename
        filename = os.path.basename(pdf_path)
        # Extract professor name
        professor_match = re.search(r'\(([^)]+)\)', filename)
        professor_name = professor_match.group(1) if professor_match else
        ↪  "Unknown"

        # Extract semester and year
        semester_match = re.search(r'^(SP|FL)(\d{4})', filename)
        semester = f"{semester_match.group(1)} {semester_match.group(2)}" if
        ↪  semester_match else "Unknown"

        # Extract course code and section
        course_match = re.search(r'(L24|E81)[._](\d+\w*)[._](\d+)', filename)
        if course_match:
            course_code = course_match.group(2)
            section = course_match.group(3)
        else:
            # Try another pattern
            course_match = re.search(r'(L24|E81)\s+(\d+\w*)\s+(\d+)', filename)
            if course_match:
                course_code = course_match.group(2)
                section = course_match.group(3)
            else:
                course_code = "Unknown"
                section = "Unknown"

        # Extract text from PDF
        text = ""
        with pdfplumber.open(pdf_path) as pdf:
            for page in pdf.pages:
                page_text = page.extract_text()
                if page_text:
                    text += page_text + "\n"

        # Extract course name
        course_name = "Unknown"
        course_name_match = re.search(r'Reports for .*?- (.*?)\(', text)
        if course_name_match:
            course_name = course_name_match.group(1).strip()

```

```
        # Look for Learning section which contains overall course rating
        overall_rating = None
        patterns = [
            r'Learning\s+Competency Statistics\s+Value\s+Mean\s+(\d+\.\d+)',
            r'Learning.*?\nCompetency Statistics\s+Value\s+Mean\s+(\d+\.\d+)',
            r'Learning.*?Mean\s+(\d+\.\d+)',
            r'Learning.*?Mean\s*(\d+\.\d+)',
        ]
        for pattern in patterns:
            learning_match = re.search(pattern, text, re.DOTALL)
            if learning_match:
                overall_rating = learning_match.group(1)
                break

        return {
            'Filename': filename,
            'Professor': professor_name,
            'Semester': semester,
            'Course Code': course_code,
            'Section': section,
            'Course Name': course_name,
            'Overall Rating': overall_rating
        }
    except Exception as e:
        print(f"Error processing {pdf_path}: {e}")
        return {
            'Filename': os.path.basename(pdf_path),
            'Professor': "Error",
            'Semester': "Error",
            'Course Code': "Error",
            'Section': "Error",
            'Course Name': "Error",
            'Overall Rating': None
        }
```

```
# Function to process all PDFs in a directory
def process_pdfs(directory):
    """
    Process all PDFs in a directory.
    Args:
        directory (str): Directory containing PDF files
    Returns:
        pd.DataFrame: DataFrame containing course information
    """
    # Get all PDF from the directory
    pdf_files = [os.path.join(directory, f) for f in os.listdir(directory) if
    ↪  f.endswith('.pdf') and (f.startswith('SP') or f.startswith('FL'))]
```

```
    print(f"Found {len(pdf_files)} PDF files to process")

    # Process each PDF
    results = []
    for pdf_file in tqdm(pdf_files, desc="Processing PDFs"):
        result = extract_course_info(pdf_file)
        results.append(result)

    # Create DataFrame
    df = pd.DataFrame(results)

    # Convert Overall Rating to numeric
    df['Overall Rating'] = pd.to_numeric(df['Overall Rating'], errors='coerce')
    return df
```

6.2.2 Process PDFs

```
# Set the directory containing the PDF files
pdf_directory = os.path.expanduser("~/Documents/course_evaluations")

# Process the PDFs
df = process_pdfs(pdf_directory)
```

6.2.3 Analyze Results

```
# Display basic statistics
print(f"Total files processed: {len(df)}")
print(f"Files with ratings found: {df['Overall Rating'].notna().sum()}")
print(f"Success rate: {df['Overall Rating'].notna().sum() / len(df) * 100:.2f}%")

# Display the professors and their ratings
ratings_df = df[df['Overall Rating'].notna()].sort_values('Overall Rating',
↪  ascending=False)
print("\nProfessors and their ratings:")
display(ratings_df[['Professor', 'Course Code', 'Course Name', 'Overall
↪  Rating']])
```

6.2.4 Visualize Results

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

```
plt.title('Professor Ratings', fontsize=16)
plt.xlabel('Overall Rating', fontsize=14)
plt.ylabel('Professor', fontsize=14)
plt.tight_layout()
plt.show()
```

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

6.2.5 Additional Analysis

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

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

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

6.3 process_pdfs.py

```
import os
import re
import shutil
```

```python
import tempfile
import snowflake.connector
import pdfplumber
import pandas as pd
from io import BytesIO
from tqdm import tqdm
from snowflake.connector.pandas_tools import write_pandas

def extract_course_info_from_bytes(file_bytes, filename):
    filename = filename.strip()
    # Extract professor name via $$...$$ delimiters.
    prof_m = re.search(r'\$\$(.*?)\$\$', filename)
    professor = prof_m.group(1) if prof_m else "Unknown"
    # Extract semester (e.g., SP2025 or FL2025).
    sem_m = re.search(r'^(SP|FL)(\d{4})', filename)
    semester = f"{sem_m.group(1)} {sem_m.group(2)}" if sem_m else "Unknown"
    # Extract course code and section from filename.
    crs_m = re.search(r'(L24|E81)[._\s]+(\w+)[._\s]+(\d+)', filename)
    course_code, section = (crs_m.group(2), crs_m.group(3)) if crs_m else
    ↪  ("Unknown", "Unknown")

    text = ""
    pdf_stream = BytesIO(file_bytes)
    try:
        with pdfplumber.open(pdf_stream) as pdf:
            for page in pdf.pages:
                page_text = page.extract_text()
                if page_text:
                    text += page_text + " "
    except Exception as e:
        print(f"Error extracting text from {filename}: {e}")
        text = ""

    name_m = re.search(r'Reports for .*?- (.*?)\(', text)
    course_name = name_m.group(1).strip() if name_m else "Unknown"
    rate_m = re.search(r'Overall Rating:\s*(\d+\.\d+)', text)
    overall_rating = float(rate_m.group(1)) if rate_m else None

    return {
        "filename": filename,
        "professor": professor,
        "semester": semester,
        "course_code": course_code,
        "section": section,
        "course_name": course_name,
        "overall_rating": overall_rating
    }
```

```
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_pdfs_from_snowflake(stage_name="APP_STG"):
62      conn = snowflake.connector.connect(
63          account="SFEDU02-IOB55870",
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'):
```

```
                continue
            tmp_dir = tempfile.mkdtemp()
            try:
                stage_file = f"@{stage_name}/{filename}"
                get_query = f"GET '{stage_file}' file://{tmp_dir}/"
                try:
                    cursor.execute(get_query)
                except Exception as e:
                    print(f"Failed GET for {filename}: {e}")
                    continue
                local_path = os.path.join(tmp_dir, filename)
                if not os.path.exists(local_path):
                    print(f"File not found locally: {local_path}")
                    continue
                try:
                    with open(local_path, 'rb') as f:
                        file_bytes = f.read()
                    info = extract_course_info_from_bytes(file_bytes, filename)
                    results.append(info)
                except Exception as e:
                    print(f"Failed processing {filename}: {e}")
            finally:
                shutil.rmtree(tmp_dir, ignore_errors=True)
        if not results:
            print("No PDF files were successfully processed")
            return pd.DataFrame()
        df = pd.DataFrame(results)
        cleaned_df = clean_and_store_data(df)
        success, nchunks, nrows, _ = write_pandas(
            conn=conn,
            df=cleaned_df,
            table_name='APP_PROCESSED_PDFS',
            schema='AIRFLOW',
            database='BISON_DB'
        )
        if success:
            print(f"Successfully wrote {nrows} rows to APP_PROCESSED_PDFS.")
        else:
            print("Failed to write PDF data to Snowflake.")
        return cleaned_df
    except Exception as e:
        print(f"Error in process_pdfs_from_snowflake: {e}")
        return pd.DataFrame()
    finally:
        cursor.close()
        conn.close()

```

```python
if __name__ == "__main__":
    df = process_pdfs_from_snowflake()
    if not df.empty:
        print("\nProcessed Data Sample:")
        print(df.head())
```

6.4 scrape_professor_ratings.py

```python
import os
import re
import shutil
import tempfile
import time
import random
import pandas as pd
import requests
from bs4 import BeautifulSoup
import snowflake.connector

def scrape_professor_rating(rmp_id):
    if not rmp_id or pd.isnull(rmp_id):
        return None
    print(f"Scraping professor: {rmp_id}")
    url = f"https://www.ratemyprofessors.com/professor/{rmp_id}"
    headers = {
        'User-Agent': ('Mozilla/5.0 (Windows NT 10.0; Win64; x64) '
                       'AppleWebKit/537.36 (KHTML, like Gecko) '
                       'Chrome/91.0.4472.124 Safari/537.36'),
        'Accept-Language': 'en-US,en;q=0.9',
        'Referer': 'https://www.ratemyprofessors.com/'
    }
    try:
        response = requests.get(url, headers=headers)
        if response.status_code != 200:
            print(f"Failed to retrieve page for professor ID {rmp_id}:
            ↪ {response.status_code}")
            return None
        soup = BeautifulSoup(response.text, 'html.parser')
        rating_info = {
            'rmp_id': rmp_id,
            'scraped_overall_rating': None,
            'would_take_again': None,
            'difficulty': None,
            'tags': []
        }
```

```
        rating_element = soup.find('div',
        ↪  class_=re.compile(r'RatingValue__Numerator-.*'))
        if rating_element:
            rating_info['scraped_overall_rating'] = rating_element.text.strip()
        feedback_numbers = soup.find_all('div',
        ↪  class_=re.compile(r'FeedbackItem__FeedbackNumber-.*'))
        if len(feedback_numbers) >= 2:
            wt_again = feedback_numbers[0].text.strip()
            difficulty = feedback_numbers[1].text.strip()
            if '%' in wt_again:
                wt_again = wt_again.replace('%', '')
            rating_info['would_take_again'] = wt_again
            rating_info['difficulty'] = difficulty
        tag_elements = soup.find_all('span', class_=re.compile(r'Tag-.*'))
        if tag_elements:
            tags = set([tag.text.lower().strip() for tag in tag_elements])
            rating_info['tags'] = tags
        print(rating_info)
        return rating_info
    except Exception as e:
        print(f"Error scraping professor ID {rmp_id}: {e}")
        return None

def load_input_table(cursor):
    query = "SELECT * FROM APP_INPUT"
    cursor.execute(query)
    df = cursor.fetch_pandas_all()
    if df is not None and not df.empty:
        df.columns = [col.lower().strip() for col in df.columns]
    return df

def scrape_professor_ratings_table(input_table, output_table, sf_conn_params):
    conn = snowflake.connector.connect(**sf_conn_params)
    cursor = conn.cursor()
    try:
        query = f"SELECT * FROM {input_table}"
        cursor.execute(query)
        df = cursor.fetch_pandas_all()
        if df is None or df.empty:
            print("No data found in input table.")
            cursor.close()
            conn.close()
            return
        df.columns = [col.lower().strip() for col in df.columns]
        input_df = load_input_table(cursor)
        if input_df.empty:
            print("APP_INPUT table is empty. Exiting.")
```

```
            cursor.close()
            conn.close()
            return
        df['course_code'] = df['course_code'].astype(str).str.strip()
        input_df['course_code'] = input_df['course_code'].astype(str).str.strip()
        try:
            df = pd.merge(df, input_df[['course_code', 'rmp_id']],
            ↪  on='course_code', how='left', suffixes=("", "_input"))
            if 'rmp_id_input' in df.columns:
                df['rmp_id'] = df['rmp_id_input'].combine_first(df.get('rmp_id'))
                df.drop(columns=['rmp_id_input'], inplace=True)
        except Exception as e:
            print(f"Error during merge with APP_INPUT: {e}")
        df.columns = [col.lower().strip() for col in df.columns]
        if 'rmp_id' not in df.columns or df['rmp_id'].dropna().empty:
            print("No RMP IDs found after merging. Exiting.")
            cursor.close()
            conn.close()
            return
        df['scraped_overall_rating'] = None
        df['would_take_again'] = None
        df['difficulty'] = None
        df['tags'] = None
        unique_ids = df['rmp_id'].dropna().unique()
        print(f"Found {len(unique_ids)} unique professors with RMP IDs")
        ratings_by_id = {}
        processed = 0
        for rmp_id in unique_ids:
            try:
                rmp_id_clean = str(int(float(rmp_id))).strip()
            except Exception:
                rmp_id_clean = str(rmp_id)
            rating_info = scrape_professor_rating(rmp_id_clean)
            if rating_info:
                ratings_by_id[rmp_id_clean] = rating_info
            processed += 1
            print(f"Processed {processed}/{len(unique_ids)} professors")
            time.sleep(random.uniform(1, 3))
        for idx, row in df.iterrows():
            if pd.notna(row.get('rmp_id')):
                try:
                    rmp_id_clean = str(int(float(row['rmp_id']))).strip()
                except Exception:
                    rmp_id_clean = str(row['rmp_id'])
                if rmp_id_clean in ratings_by_id:
                    info = ratings_by_id[rmp_id_clean]
                    df.at[idx, 'scraped_overall_rating'] =
                    ↪  info.get('scraped_overall_rating')
```

```
                    df.at[idx, 'would_take_again'] = info.get('would_take_again')
                    df.at[idx, 'difficulty'] = info.get('difficulty')
                    df.at[idx, 'tags'] = ','.join(info.get('tags', []))
        def clean_cell(x):
            if pd.isnull(x):
                return None
            if isinstance(x, str) and x.strip().lower() == "nan":
                return None
            return str(x).strip()
        df = df.applymap(clean_cell)
        columns = list(df.columns)
        create_sql = f"CREATE OR REPLACE TABLE {output_table} ("
        create_sql += ", ".join([f"\"{col.strip().upper().replace(' ', '_')}\" STRING" for col in columns])
        create_sql += ")"
        print("DDL:", create_sql)
        cursor.execute(create_sql)
        placeholders = ", ".join(["%s"] * len(columns))
        insert_sql = f"INSERT INTO {output_table} VALUES({placeholders})"
        data = [tuple(row) for row in df.values]
        cursor.executemany(insert_sql, data)
        conn.commit()
        cursor.close()
        conn.close()
        print(f"Scraped professor ratings stored in table {output_table}")
    except Exception as e:
        print(f"Error in scrape_professor_ratings_table: {e}")
        cursor.close()
        conn.close()

if __name__ == "__main__":
    sf_conn_params = {
        "account": "SFEDU02-IOB55870",
        "user": "BISON",
        "password": "voshim-bopcyt-3virBe",
        "role": "TRAINING_ROLE",
        "warehouse": "BISON_WH",
        "database": "BISON_DB",
        "schema": "AIRFLOW",
    }
    scrape_professor_ratings_table("APP_PROCESSED_PDFS", "RMP_ENRICHED_DATA", sf_conn_params)
```

6.5 clean_and_store_data.py

```python
import os
import pandas as pd
import numpy as np
import snowflake.connector
from snowflake.connector.pandas_tools import write_pandas

def clean_and_store_data(input_table, target_table, sf_conn_params=None):
    """
    Clean the DataFrame and store it in Snowflake.
    Reads from input_table (e.g. RMP_ENRICHED_DATA), cleans the data,
    and writes the result to target_table.
    """
    if sf_conn_params is None:
        sf_conn_params = {
            "account": "SFEDU02-IOB55870",
            "user": "BISON",
            "password": "voshim-bopcyt-3virBe",
            "role": "TRAINING_ROLE",
            "warehouse": "BISON_WH",
            "database": "BISON_DB",
            "schema": "AIRFLOW",
        }
    conn = snowflake.connector.connect(**sf_conn_params)
    cursor = conn.cursor()
    try:
        query = f"SELECT * FROM {input_table}"
        cursor.execute(query)
        df = cursor.fetch_pandas_all()
        if df.empty:
            print(f"No data found in table {input_table}.")
            return df
        df = df.replace([np.nan, 'NAN', 'nan', 'NaN', '', None], None)
        df.columns = [col.upper() for col in df.columns]
        if "OVERALL_RATING" in df.columns:
            df["OVERALL_RATING"] = pd.to_numeric(df["OVERALL_RATING"],
            ↪  errors="coerce")
        if "RMP_RATING" in df.columns:
            df["RMP_RATING"] = pd.to_numeric(df["RMP_RATING"], errors="coerce")
        if "DIFFICULTY" in df.columns:
            df["DIFFICULTY"] = pd.to_numeric(df["DIFFICULTY"], errors="coerce")
        if "WOULD TAKE AGAIN" in df.columns:
            df["WOULD TAKE AGAIN"] = pd.to_numeric(df["WOULD TAKE AGAIN"],
            ↪  errors="coerce")
            df["WOULD TAKE AGAIN"] = df["WOULD TAKE AGAIN"].apply(lambda x: x /
            ↪  100 if pd.notna(x) and x > 1 else x)
```

```
        print("Cleaned data sample:")
        print(df.head())
        success, nchunks, nrows, _ = write_pandas(conn, df, target_table,
        ↪  quote_identifiers=True)
        if success:
            print(f"Loaded {nrows} rows into table {target_table}")
        else:
            print("Failed to write data to Snowflake.")
    finally:
        cursor.close()
        conn.close()
    return df

if __name__ == "__main__":
    sf_conn_params = {
        "account": "SFEDU02-IOB55870",
        "user": "BISON",
        "password": "voshim-bopcyt-3virBe",
        "role": "TRAINING_ROLE",
        "warehouse": "BISON_WH",
        "database": "BISON_DB",
        "schema": "AIRFLOW",
    }
    clean_and_store_data("RMP_ENRICHED_DATA", "MY_WASHU_COURSES_WITH_RATINGS",
    ↪  sf_conn_params)
```

6.6 app_data_pipeline.py (DAG)

```
from airflow.providers.common.sql.operators.sql import SQLExecuteQueryOperator
from airflow.providers.common.sql.sensors.sql import SqlSensor
from airflow.operators.python import PythonOperator
from airflow import DAG
from airflow.decorators import dag
from datetime import datetime, timedelta

from process_pdfs import process_pdfs_from_snowflake
from scrape_professor_ratings import scrape_professor_ratings_table
from clean_and_store_data import clean_and_store_data

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime.now() - timedelta(days=1),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
```

```
}

sf_conn_params = {
    "account": "SFEDU02-IOB55870",
    "user": "BISON",
    "password": "voshim-bopcyt-3virBe",
    "role": "TRAINING_ROLE",
    "warehouse": "BISON_WH",
    "database": "BISON_DB",
    "schema": "AIRFLOW",
}

@dag(
    default_args=default_args,
    schedule='@daily',
    catchup=False,
    dag_id="app_data_pipeline"
)
def app_data_pipeline():
    create_signal_table = SQLExecuteQueryOperator(
        task_id='create_signal_table',
        conn_id='SnowFlakeConn',
        sql="""
            CREATE OR REPLACE TABLE SIGNAL_TABLE (
                TABLE_NAME STRING,
                STATUS STRING,
                TIMESTAMP TIMESTAMP_LTZ
            );
        """,
    )
    insert_dummy_signal = SQLExecuteQueryOperator(
        task_id='insert_dummy_signal',
        conn_id='SnowFlakeConn',
        sql="""
            INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
            VALUES ('APP_STG', 'success', CURRENT_TIMESTAMP());
        """,
    )
    check_app_stg_signal = SqlSensor(
        task_id='check_app_stg_signal',
        conn_id='SnowFlakeConn',
        sql="SELECT 1 FROM SIGNAL_TABLE WHERE TABLE_NAME = 'APP_STG' AND STATUS =
        ↪  'success'",
        timeout=600,
        poke_interval=30,
        mode='reschedule'
    )
```

```
    create_target_table = SQLExecuteQueryOperator(
        task_id='create_target_table',
        conn_id='SnowFlakeConn',
        sql="""
            CREATE OR REPLACE TABLE MY_WASHU_COURSES_WITH_RATINGS (
                FILENAME VARCHAR,
                PROFESSOR VARCHAR,
                SEMESTER VARCHAR,
                COURSE_CODE VARCHAR,
                SECTION VARCHAR,
                COURSE_NAME VARCHAR,
                OVERALL_RATING FLOAT,
                RMP_ID NUMBER(38,0),
                SCRAPED_OVERALL_RATING FLOAT,
                WOULD_TAKE_AGAIN FLOAT,
                DIFFICULTY FLOAT,
                TAGS VARCHAR
            );
        """,
    )
    def run_process_pdfs():
        stage_name = "APP_STG"
        df = process_pdfs_from_snowflake(stage_name)
        if df.empty:
            print("No PDF data extracted; skipping further steps.")
            return
        print("Processed PDF data stored in table APP_PROCESSED_PDFS")
    process_pdfs_task = PythonOperator(
        task_id='process_pdfs',
        python_callable=run_process_pdfs,
    )
    def run_scrape_rmp():
        input_table = "APP_PROCESSED_PDFS"
        output_table = "RMP_ENRICHED_DATA"
        scrape_professor_ratings_table(input_table, output_table, sf_conn_params)
        print("Scraped RMP data stored in table RMP_ENRICHED_DATA")
    scrape_rmp_task = PythonOperator(
        task_id='scrape_rmp',
        python_callable=run_scrape_rmp,
    )
    def run_clean_and_store():
        clean_and_store_data("RMP_ENRICHED_DATA", "MY_WASHU_COURSES_WITH_RATINGS",
        ↪  sf_conn_params)
        print("Enriched data loaded into table MY_WASHU_COURSES_WITH_RATINGS")
    load_enriched_data_task = PythonOperator(
        task_id='load_enriched_data',
        python_callable=run_clean_and_store,
```

```
    )
    standardization_sql = """
    CREATE OR REPLACE TABLE WASHU_COURSES_STANDARDIZED AS
    WITH normalized_data AS (
        SELECT
            TRIM(PROFESSOR) AS ORIGINAL_PROFESSOR_NAME,
            SPLIT_PART(TRIM(PROFESSOR), ' ', -1) AS PROFESSOR_NAME,
            TRIM(COURSE_CODE) AS COURSE_CODE,
            TRIM(COURSE_NAME) AS COURSE_NAME,
            TRIM(SEMESTER) AS SEMESTER,
            TRIM(SECTION) AS SECTION,
            RMP_ID,
            TO_NUMBER(OVERALL_RATING) AS OVERALL_RATING_NUM,
            TO_NUMBER(SCRAPED_OVERALL_RATING) AS SCRAPED_OVERALL_RATING_NUM,
            CASE
                WHEN TO_NUMBER(WOULD_TAKE_AGAIN) > 1 THEN
                ↪  TO_NUMBER(WOULD_TAKE_AGAIN)/100
                ELSE TO_NUMBER(WOULD_TAKE_AGAIN)
            END AS WOULD_TAKE_AGAIN_NUM,
            TO_NUMBER(DIFFICULTY) AS DIFFICULTY_NUM,
            TRIM(TAGS) AS TAGS,
            ROW_NUMBER() OVER (
                PARTITION BY
                    SPLIT_PART(TRIM(PROFESSOR), ' ', -1),
                    TRIM(COURSE_CODE),
                    TRIM(SEMESTER),
                    TRIM(SECTION)
                ORDER BY
                    CASE WHEN TO_NUMBER(OVERALL_RATING) IS NOT NULL THEN 0 ELSE 1
                    ↪  END,
                    CASE WHEN TO_NUMBER(SCRAPED_OVERALL_RATING) IS NOT NULL THEN
                    ↪  0 ELSE 1 END,
                    CASE WHEN TRIM(COURSE_NAME) IS NOT NULL THEN 0 ELSE 1 END
            ) AS ROW_RANK
        FROM MY_WASHU_COURSES_WITH_RATINGS
    )
    SELECT
        PROFESSOR_NAME,
        ORIGINAL_PROFESSOR_NAME AS FULL_PROFESSOR_NAME,
        COURSE_CODE,
        COURSE_NAME,
        SEMESTER,
        SECTION,
        RMP_ID,
        OVERALL_RATING_NUM AS OVERALL_RATING,
        SCRAPED_OVERALL_RATING_NUM AS SCRAPED_OVERALL_RATING,
        WOULD_TAKE_AGAIN_NUM AS WOULD_TAKE_AGAIN,
```

```
        DIFFICULTY_NUM AS DIFFICULTY,
        TAGS,
        CASE
            WHEN SCRAPED_OVERALL_RATING_NUM IS NOT NULL THEN
            ↪  (SCRAPED_OVERALL_RATING_NUM - 1) * (6/4) + 1
            ELSE NULL
        END AS NORMALIZED_RMP_RATING
    FROM normalized_data
    WHERE ROW_RANK = 1;

    CREATE OR REPLACE VIEW V_WASHU_COURSES AS
    SELECT * FROM WASHU_COURSES_STANDARDIZED;
    """
    standardization_task = SQLExecuteQueryOperator(
        task_id='run_standardization',
        conn_id='SnowFlakeConn',
        sql=standardization_sql,
    )
    insert_signal_final = SQLExecuteQueryOperator(
        task_id='insert_signal_final',
        conn_id='SnowFlakeConn',
        sql="""
            INSERT INTO SIGNAL_TABLE (TABLE_NAME, STATUS, TIMESTAMP)
            VALUES ('WASHU_COURSES_STANDARDIZED', 'success',
            ↪  CURRENT_TIMESTAMP());
        """,
    )
    create_signal_table >> insert_dummy_signal >> check_app_stg_signal >>
    ↪  process_pdfs_task
    process_pdfs_task >> scrape_rmp_task >> create_target_table >>
    ↪  load_enriched_data_task >> standardization_task >> insert_signal_final

app_data_pipeline_dag = app_data_pipeline()
```

6.7 snowflake_recommendation_api.py (Flask API)

```
from flask import Flask, request, jsonify
import pandas as pd
import numpy as np
import json
import re
import snowflake.connector
from datetime import datetime
import os
import csv

```

```python
app = Flask(__name__)

def safe_float(value, default=float('nan')):
    if value is None or value == '':
        return default
    try:
        return float(value)
    except (ValueError, TypeError):
        return default

SNOWFLAKE_USER = "BISON"
SNOWFLAKE_PASSWORD = "voshim-bopcyt-3virBe"
SNOWFLAKE_ACCOUNT = "SFEDU02-IOB55870"
SNOWFLAKE_WAREHOUSE = "BISON_WH"
SNOWFLAKE_DATABASE = "BISON_DB"
SNOWFLAKE_SCHEMA = "PUBLIC"

def connect_to_snowflake():
    try:
        conn = snowflake.connector.connect(
            user=SNOWFLAKE_USER,
            password=SNOWFLAKE_PASSWORD,
            account=SNOWFLAKE_ACCOUNT,
            warehouse=SNOWFLAKE_WAREHOUSE,
            database=SNOWFLAKE_DATABASE,
            schema=SNOWFLAKE_SCHEMA
        )
        return conn
    except Exception as e:
        print(f"Error connecting to Snowflake: {str(e)}")
        return None

def load_data_from_snowflake():
    conn = connect_to_snowflake()
    if conn:
        try:
            cursor = conn.cursor()
            cursor.execute("SELECT * FROM V_WASHU_COURSES")
            column_names = [desc[0] for desc in cursor.description]
            rows = cursor.fetchall()
            df = pd.DataFrame(rows, columns=column_names)
            cursor.close()
            conn.close()
            numeric_cols = ['STANDARDIZED_RATING', 'STANDARDIZED_DIFFICULTY',
            ↪  'STANDARDIZED_WOULD_TAKE_AGAIN']
            for col in numeric_cols:
                if col in df.columns:
                    df[col] = pd.to_numeric(df[col], errors='coerce')
            return df
        except Exception as e:
            print(f"Error loading data from Snowflake: {str(e)}")
            return load_data_from_csv()
    else:
```

```
        return load_data_from_csv()

def load_data_from_csv():
    try:
        df = pd.read_csv("DataBase-2.csv")
        numeric_cols = ['STANDARDIZED_RATING', 'STANDARDIZED_DIFFICULTY',
        ↪  'STANDARDIZED_WOULD_TAKE_AGAIN']
        for col in numeric_cols:
            if col in df.columns:
                df[col] = pd.to_numeric(df[col], errors='coerce')
        return df
    except Exception as e:
        print(f"Error loading data from CSV: {str(e)}")
        return pd.DataFrame()

df = load_data_from_snowflake()

@app.route('/search', methods=['GET'])
def search():
    try:
        course_code = request.args.get('course_code', '')
        course_name = request.args.get('course_name', '')
        professor_name = request.args.get('professor_name', '')
        min_rating = request.args.get('min_rating', '')
        max_difficulty = request.args.get('max_difficulty', '')
        tags = request.args.get('tags', '')

        filtered_df = df.copy()
        if course_code:
            filtered_df =
            ↪  filtered_df[filtered_df['COURSE_CODE'].str.contains(course_code,
            ↪  case=False, na=False)]
        if course_name:
            filtered_df =
            ↪  filtered_df[filtered_df['COURSE_NAME'].str.contains(course_name,
            ↪  case=False, na=False)]
        if professor_name:
            filtered_df =
            ↪  filtered_df[filtered_df['PROFESSOR_NAME'].str.contains(professor_name,
            ↪  case=False, na=False)]
        if min_rating:
            try:
                min_rating_float = safe_float(min_rating)
                if not np.isnan(min_rating_float):
                    filtered_df = filtered_df[filtered_df['STANDARDIZED_RATING'] >=
                    ↪  min_rating_float]
            except (ValueError, TypeError):
                pass
        if max_difficulty:
            try:
                max_difficulty_float = safe_float(max_difficulty)
                if not np.isnan(max_difficulty_float):
                    filtered_df = filtered_df[filtered_df['STANDARDIZED_DIFFICULTY'] <=
                    ↪  max_difficulty_float]
```

```
            except (ValueError, TypeError):
                pass
        if tags:
            tag_list = tags.split(',')
            for tag in tag_list:
                filtered_df = filtered_df[filtered_df['TAGS'].str.contains(tag.strip(),
                ↪  case=False, na=False)]

        results = filtered_df.fillna('').to_dict(orient='records')
        return jsonify({
            "status": "success",
            "results": results
        })
    except Exception as e:
        return jsonify({
            "status": "error",
            "error": str(e)
        })

@app.route('/submit_review', methods=['POST'])
def submit_review():
    try:
        review_data = request.json
        if not review_data:
            return jsonify({"status": "error", "error": "No review data provided."})
        required_fields = ['course_code', 'professor_name']
        for field in required_fields:
            if field not in review_data or not review_data[field]:
                return jsonify({"status": "error", "error": f"{field} is required."})
        try:
            if 'rating' in review_data:
                try:
                    review_data['rating'] = safe_float(review_data['rating'])
                except ValueError:
                    review_data['rating'] = None
            if 'difficulty' in review_data:
                review_data['difficulty'] = safe_float(review_data['difficulty'])
            if 'would_take_again' in review_data:
                review_data['would_take_again'] =
                ↪  safe_float(review_data['would_take_again'])
        except Exception as e:
            return jsonify({"status": "error", "error": f"Invalid numeric value:
            ↪  {str(e)}"})
        review_data['timestamp'] = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        reviews_file = "reviews.csv"
        headers = ['course_code', 'course_name', 'professor_name', 'semester',
        ↪  'section',
                   'rating', 'difficulty', 'would_take_again', 'tags', 'comments',
                   ↪  'timestamp']
        file_exists = os.path.exists(reviews_file)
        with open(reviews_file, 'a', newline='', encoding='utf-8') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=headers)
            if not file_exists:
```

```
                writer.writeheader()
            sanitized_data = {
                key: str(review_data.get(key, '')).replace(',', ';') for key in headers
            }
            writer.writerow(sanitized_data)
        return jsonify({"status": "success", "message": "Review submitted
        ↪  successfully."})
    except Exception as e:
        return jsonify({"status": "error", "error": str(e)})

if __name__ == '__main__':
    app.run(debug=True)
```

6.8 course_recommendation_app.py (Streamlit App)

```
import streamlit as st
import pandas as pd
import numpy as np
import requests
import json
import plotly.graph_objects as go
from datetime import datetime

def safe_float(value, default=float('nan')):
    if value is None or value == '':
        return default
    try:
        return float(value)
    except (ValueError, TypeError):
        return default

API_ENDPOINT = "http://127.0.0.1:5000"

st.set_page_config(
    page_title="Course Recommendation System",
    page_icon="",
    layout="wide",
    initial_sidebar_state="expanded"
)

st.markdown('''
<style>
    .main-header {
        font-size: 2.5rem;
        color: #A51417;
        text-align: center;
        margin-bottom: 1rem;
    }
    .sub-header {
        font-size: 1.5rem;
```

```
        color: #A51417;
        margin-top: 2rem;
        margin-bottom: 1rem;
    }
    .card {
        border-radius: 5px;
        background-color: #f9f9f9;
        padding: 1.5rem;
        margin-bottom: 1rem;
        border-left: 5px solid #A51417;
    }
    .sentiment-tag {
        display: inline-block;
        background-color: #E5E7EB;
        color: #1F2937;
        padding: 0.25rem 0.5rem;
        border-radius: 9999px;
        margin-right: 0.5rem;
        margin-bottom: 0.5rem;
        font-size: 0.875rem;
    }
    .rating-container {
        display: flex;
        align-items: center;
        margin-bottom: 0.5rem;
    }
    .rating-label {
        min-width: 150px;
        font-weight: bold;
    }
    .rating-value {
        margin-left: 1rem;
    }
</style>
''', unsafe_allow_html=True)

st.markdown('<h1 class="main-header">Washington University in St. Louis Course
↪  Recommendation System</h1>', unsafe_allow_html=True)

st.sidebar.image("https://marcomm.washu.edu/app/uploads/2024/02/WashU-SHIELD-Red_RGB.jp
↪  g",
↪  width=150)
st.sidebar.title("Filters & Preferences")
major = st.sidebar.text_input("Your Major", "Computer Science")
year = st.sidebar.selectbox("Year", [1, 2, 3, 4])
st.sidebar.markdown("### What matters to you?")
st.sidebar.markdown("Select preferences that are important to you:")
importance_rating = st.sidebar.slider("Overall Rating Importance", 0, 10, 8)
importance_difficulty = st.sidebar.slider("Low Difficulty Importance", 0, 10, 5)
importance_would_take_again = st.sidebar.slider("Would Take Again Importance", 0, 10, 7)
sentiment_options = [
    "accessible outside class", "amazing lectures", "caring",
    "clear grading criteria", "extra credit", "get ready to read",
```

```
    "gives good feedback", "graded by few things", "group projects",
    "helpful", "inspirational", "lecture heavy", "lots of homework",
    "participation matters", "respected", "skip class? you won't pass",
    "so many papers", "test heavy", "tough grader"
]

tab1, tab2 = st.tabs(["Find Courses", "Submit Review"])

with tab1:
    st.markdown('<h2 class="sub-header">Find Your Ideal Courses</h2>',
    ↪  unsafe_allow_html=True)
    search_col1, search_col2 = st.columns(2)
    with search_col1:
        search_by = st.radio("Search by:", ["Course", "Professor", "Rating", "Advanced
        ↪  Search"])
    with search_col2:
        if search_by == "Course":
            search_term = st.text_input("Enter Course Code or Name:")
        elif search_by == "Professor":
            search_term = st.text_input("Enter Professor Name:")
        elif search_by == "Rating":
            min_rating = st.slider("Minimum Rating", 1.0, 7.0, 4.0, 0.1)
            search_term = str(min_rating)
        else:
            st.write("Use the filters below for advanced search")
            search_term = ""
    if search_by == "Advanced Search":
        adv_col1, adv_col2, adv_col3 = st.columns(3)
        with adv_col1:
            course_code = st.text_input("Course Code:")
            course_name = st.text_input("Course Name:")
        with adv_col2:
            professor_name = st.text_input("Professor Name:")
            min_rating = st.slider("Minimum Rating", 1.0, 7.0, 3.0, 0.1)
        with adv_col3:
            max_difficulty = st.slider("Maximum Difficulty", 1.0, 7.0, 5.0, 0.1)
            selected_tags = st.multiselect("Course Tags:", sentiment_options)
    if st.button("Search Courses"):
        try:
            if search_by == "Course":
                if search_term.isdigit() or (len(search_term) <= 5 and any(c.isdigit()
                ↪  for c in search_term)):
                    response = requests.get(f"{API_ENDPOINT}/search",
                    ↪  params={"course_code": search_term})
                else:
                    response = requests.get(f"{API_ENDPOINT}/search",
                    ↪  params={"course_name": search_term})
            elif search_by == "Professor":
                response = requests.get(f"{API_ENDPOINT}/search",
                ↪  params={"professor_name": search_term})
            elif search_by == "Rating":
                response = requests.get(f"{API_ENDPOINT}/search", params={"min_rating":
                ↪  search_term})
```

```
            else:
                params = {}
                if course_code:
                    params["course_code"] = course_code
                if course_name:
                    params["course_name"] = course_name
                if professor_name:
                    params["professor_name"] = professor_name
                if min_rating:
                    params["min_rating"] = str(min_rating)
                if max_difficulty:
                    params["max_difficulty"] = str(max_difficulty)
                if selected_tags:
                    params["tags"] = ",".join(selected_tags)
                response = requests.get(f"{API_ENDPOINT}/search", params=params)
            if response.status_code == 200:
                result = response.json()
                if result.get("status") == "success":
                    results = result.get("results", [])
                    if not results:
                        st.warning("No courses found matching your criteria.")
                    else:
                        st.success(f"Found {len(results)} courses matching your
                        ↪  criteria.")
                        for course in results:
                            rating_raw = course.get("STANDARDIZED_RATING", '0')
                            difficulty_raw = course.get("STANDARDIZED_DIFFICULTY",
                            ↪  '3.5')
                            would_take_again_raw =
                            ↪  course.get("STANDARDIZED_WOULD_TAKE_AGAIN", '0.5')
                            rating = safe_float(rating_raw) if rating_raw not in [None,
                            ↪  ''] else 0.0
                            difficulty = safe_float(difficulty_raw) if difficulty_raw
                            ↪  not in [None, ''] else 3.5
                            would_take_again = safe_float(would_take_again_raw) if
                            ↪  would_take_again_raw not in [None, ''] else 0.5
                            rating_score = rating * importance_rating / 10
                            difficulty_score = (7 - difficulty) * importance_difficulty
                            ↪  / 10
                            would_take_again_score = would_take_again * 7 *
                            ↪  importance_would_take_again / 10
                            course["RECOMMENDATION_SCORE"] = rating_score +
                            ↪  difficulty_score + would_take_again_score
                        results.sort(key=lambda x: x.get("RECOMMENDATION_SCORE", 0),
                        ↪  reverse=True)
                        for i, course in enumerate(results):
                            with st.container():
                                st.markdown(f'''
                                    <div class="card">
                                        <h3>{course.get('COURSE_CODE', 'N/A')} -
                                        ↪  {course.get('COURSE_NAME', 'N/A')}</h3>
                                        <p><strong>Professor:</strong>
                                        ↪  {course.get('PROFESSOR_NAME', 'N/A')}</p>
```

```
                                        <p><strong>Rating:</strong> {safe_float(course.get('STANDARDIZED_RATING', ''), 0):.1f}/7</p>
                                        <p><strong>Difficulty:</strong> {safe_float(course.get('STANDARDIZED_DIFFICULTY', ''), 0):.1f}/7</p>
                                        <p><strong>Would Take Again:</strong> {safe_float(course.get('STANDARDIZED_WOULD_TAKE_AGAIN', ''), 0.0) * 100:.0f}%</p>
                                    </div>
                                ''', unsafe_allow_html=True)
                    else:
                        st.error("Failed to fetch data from server.")
                else:
                    st.error("Server error. Please check backend.")
            except Exception as e:
                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.