Teaching Assistants Demand Prediction

Andrew Lau

BSc Computer Science

Submission Date: 26th April 2024

Supervisor: Sarah Sanders

*This report is submitted as part requirement for the BSc Degree in Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text.*

*The report may be freely copied and distributed provided the source is explicitly acknowledged.*

**Abstract**

The recruitment of Postgraduate Teaching Assistants (PGTAs) around University College London (UCL) Computer Science Department has been an ambiguous process with no clear metrics that can predict demand accurately. Despite the obvious answers of number of students, workload of the module and the difficulty of the syllabus, there is no linear relation between these metrics and the demand of PGTAs after plotting graphs to look at the results. Experimenting with AI models and different statistical tool has also proved to be inaccurate due to the unpredictability nature of modules. Hence, this project research deeper into what creates demand for PGTAs and how it can be better predicted for future references.

Table of Contents

[1 Introduction 5](#_Toc161835041)

[1.1 Literature Review 5](#_Toc161835042)

[1.2 Aims & Goals 6](#_Toc161835043)

[2 Background 8](#_Toc161835044)

[2.1 Technology Stack 8](#_Toc161835045)

[2.2 System Architecture 9](#_Toc161835046)

[2.2.1 Data Collection 10](#_Toc161835047)

[2.2.2 Data Preprocessing 10](#_Toc161835048)

[2.2.3 Data Visualisation 10](#_Toc161835049)

[2.2.4 Data Analysis 11](#_Toc161835050)

[2.2.5 Prediction Prompt 11](#_Toc161835051)

[3 Data Handling and Database Implementation 12](#_Toc161835052)

[3.1 Data Sources 12](#_Toc161835053)

[3.1.1 Cap and Actual Students Dataset 12](#_Toc161835054)

[3.1.2 Module Assessment Dataset 12](#_Toc161835055)

[3.1.3 New Module Dataset 13](#_Toc161835056)

[3.1.4 PGTA Requested and Recruited Dataset 13](#_Toc161835057)

[3.1.5 Job Description Dataset 13](#_Toc161835058)

[3.2 Data Processing 14](#_Toc161835059)

[3.2.1 Data Processing for Graph Plotting and Linear Regression Model Training 15](#_Toc161835060)

[3.2.2 Data Processing for DataFrame Cleaning 16](#_Toc161835061)

[3.2.3 Data Processing for TF-IDF Vectorisation 17](#_Toc161835062)

[3.3 Database (SQLite) 18](#_Toc161835063)

[3.3.1 Database Comparison 18](#_Toc161835064)

[3.3.2 Database Implementation 18](#_Toc161835065)

[3.3.3 Database Interactions 20](#_Toc161835066)

[4 Graph Analysis 23](#_Toc161835067)

[4.1 PGTAs Requested vs PGTAs Recruited Graph 23](#_Toc161835068)

[4.2 Module History Graph 24](#_Toc161835069)

[4.3 Students Enrolled vs PGTAs Recruited Graph 25](#_Toc161835070)

[4.4 Exam-Coursework Ratio vs PGTAs Recruited Graph 25](#_Toc161835071)

[4.5 Module Delivery Code vs PGTAs Recruited Graph 26](#_Toc161835072)

[4.6 Duties vs PGTA Hours Graph 26](#_Toc161835073)

[4.7 Duties vs Average PGTA Hours Graph 27](#_Toc161835074)

[4.8 Results Evaluation and Conclusion 28](#_Toc161835075)

[5 Machine Learning Models Implementation 29](#_Toc161835076)

[5.1 Regression Models 29](#_Toc161835077)

[5.1.1 Introduction 29](#_Toc161835078)

[5.1.2 Regression Model Training 29](#_Toc161835079)

[5.1.3 Results Evaluation 30](#_Toc161835080)

[5.2 Natural Language Processing 31](#_Toc161835081)

[5.2.1 Prediction Using Term Frequency-Inverse Document Frequency (TF-IDF) 31](#_Toc161835082)

[5.2.2 Prediction Using Feature Engineering 34](#_Toc161835083)

[5.3 Generalised Additive Model 35](#_Toc161835084)

[5.3.1 Introduction 35](#_Toc161835085)

[5.3.2 GAM and Linear Regression Comparison 35](#_Toc161835086)

[5.3.3 Model Training 36](#_Toc161835087)

[5.3.4 Results and Evaluation 36](#_Toc161835088)

[5.4 Prediction Prompts 36](#_Toc161835089)

[5.4.1 Linear Regression Prompt 36](#_Toc161835090)

[5.4.2 Ridge Regression Prompt 37](#_Toc161835091)

[5.4.3 Generalized Additive Model Prompt 39](#_Toc161835092)

[5.4.4 TF-IDF Vectorisation Prompt 40](#_Toc161835093)

[5.4.5 Feature Engineering Prompt 40](#_Toc161835094)

[5.4.6 Architecture Overview 41](#_Toc161835095)

[6 Tests 43](#_Toc161835096)

[6.1 Unit Tests 43](#_Toc161835097)

[6.1.1 Graph Plotting and Linear Regression Model Training Functions 43](#_Toc161835098)

[6.1.2 DataFrame Cleaning Functions 44](#_Toc161835099)

[6.1.3 Text Preprocessing Functions 45](#_Toc161835100)

[6.2 Integration Tests 45](#_Toc161835101)

[6.2.1 Database Operations 46](#_Toc161835102)

[6.2.2 Machine Learning Predictions 46](#_Toc161835103)

[7 Conclusion 47](#_Toc161835104)

[7.1 Achievements 47](#_Toc161835105)

[7.2 Limitations and Challenges 48](#_Toc161835106)

[7.3 Future Work 49](#_Toc161835107)

[8 Appendix 50](#_Toc161835108)

[8.1 References 50](#_Toc161835109)

[8.2 Code Listing 50](#_Toc161835110)

[8.2.1 Data Preprocessing Functions 50](#_Toc161835111)

[8.2.2 Graph Interfaces 54](#_Toc161835112)

[8.2.3 Machine Learning Model Training 60](#_Toc161835113)

[8.2.4 Machine Learning Prompts 63](#_Toc161835114)

[8.2.5 Database Operations 69](#_Toc161835115)

# 

# 1 Introduction

Postgraduate Teaching Assistants (PGTAs) are often recruited in higher education institutions to assist educators as well as facilitate the learning of undergraduate students. Some of their duties include conducting problem classes, grading and providing feedback on assessments, supporting course lecturers, preparing teaching materials and others. Since they play a crucial role in the success of running a university degree, it is important that there is an accurate estimation of demand to provide the best learning experience to undergraduate students.

At University College London (UCL) Computer Science Department, the recruitment of PGTAs is done manually through a recruitment process filled with ambiguity around the demand for PGTAs. For modules that are new, there isn’t a tried-and-tested method or official guideline on how to measure demand for PGTAs and it is usually done based on a review of the module content and intuition. This often leads to inaccurate estimation of workforce needed and inefficient allocation of resources within the department and hence, the intended teaching quality is often not delivered. A successful implementation of this project could help module administators eliminate the ambiguity when hiring PGTAs and ease module management. 1

This paper discusses the analysis of different aspects module data ranging from number of students enrolled to job descriptions of teaching assistants to help educational institutes predict the demand/workload of teaching assistants for modules. The analysis includes the creation of machine learning prediction models such as linear regression and Natural Language Processing to analyse the module structure in detail and discover hidden patterns. This project uses the historical data of module structure collected by module administrators from the past years.

## 1.1 Literature Review

Predictive analytics in the educational sector has been a crucial tool for enhancing student learning experiences and institutional efficiency. Educational institutions are increasingly leveraging data-driven decisions to identify students' strengths and weaknesses, optimize resource allocation, and ultimately improve educational outcomes. By using the latest techniques in machine learning employed on predicting teaching assistants (TA) demand, we believe the University College London (UCL) computer science department can reap the same benefits.

The research conducted by Aaditya Bhusal at the University of Northampton highlights the significant potential of machine learning in forecasting student performance. Bhusal's investigation centers on harnessing data mined from Learning Management Systems (LMS) to discover patterns indicative of students' academic outcomes. By aggregating and analyzing students’ behavioral data, there are indications of early identification of students at risk of underperforming and facilitating timely interventions can improve their academic journey.

A notable aspect of Bhusal's methodological approach is the use of binary encoding to convert non-numerical dataset features into a numerical format conducive for machine learning algorithms. This preprocessing step is crucial for handling diverse data types inherent in educational datasets, ensuring that both binary and nominal features are appropriately quantified. Such encoding techniques resonate with the practices adopted in our project, where binary and one-hot encoding methods have been employed to prepare module information for predictive modeling. Bhusal’s study, through its comprehensive data preprocessing and predictive modeling, underscores the transformative impact of predictive analytics in enhancing educational support mechanisms, motivating us to refine PGTA demand forecasting using machine learning.2

Top of Form

The paper “Predictive analytics in education: a comparison of deep learning” explores the utility and applicability of deep learning for educational data mining and learning analyticsz. The authors compare the predictive accuracy of popular deep learning frameworks/libraries, including Keras, Theano, Tensorflow, fast.ai, and Pytorch and argue that statistical learning techniques should be selected to maximize interpretability and should contribute to our understanding of educational and learning phenomena. The deep learning techniques explored in this paper could potentially be applied to predictive analytics in predicting teaching assistant (TA) demand.

Additionally, the paper’s emphasis on interpretability aligns with the goals of your research. By using interpretable models, the UCL computer science department could gain insights into the factors that drive TA demand. This could help the department make more informed decisions about resource allocation.3

Research within educational resource allocation often revolves around students and their learning outcomes. Despite the extensive studies within the educational setting, the predictive analysis for TA demand remains underexplored.Bottom of Form These studies indicate a research gap in predictive analysis specifically targeting TA demand based on module information. This gap suggests the need for this project to utilize historical data on module structure to develop machine learning models that can better predict TA demand, enhancing the resource allocation process and potentially the quality of education delivery.

## 1.2 Aims & Goals

The project aims to enhance the understanding of Postgraduate Teaching Assistants (PGTAs) demand through the deployment of analytical tools that facilitates data-driven decisions in the recruitment process. By leveraging data analytics, the project aims to bring new insights into PGTA recruitment practices and realise its potential for ensuring that resources are better allocated to improve the learning experiences of students at University College London (UCL). Here are the aims and goals that should be achieved by the end of the project:

**Identify and Understand Recruitment Challenges**

* To explore the underlying challenges in the current PGTA recruitment process.

**Analyze Factors Influencing PGTA Demand**

* To investigate the recruitment process in detail, identifying key factors that influence the demand for PGTAs.

**Facilitate Data Visualization**

* To provide various graphical visualizations that presents patterns and insights within the module information dataset, enabling stakeholders to visualise data relationships and make informed decisions regarding PGTA recruitment.

**Data Processing and Model Training**

* To process and analyse data from historical records of module information, creating an integrated dataset that serves as the foundation for training machine learning models and handling queries.

**Predict PGTA Demand for New Modules**

* To develop predictive models capable of accurately forecasting the number of PGTA hours required for new and previously untaught modules, ensuring efficient and effective TA allocation.

**Develop an Interactive Prediction Interface**

* To create a user-friendly input prompt that allows for the input of module-specific data into the machine learning models and getting real-time predictions of PGTA demand.

**Enable Database Modification by Users**

* To provide the functionality of adding new module data and delete existing entries from the database, keeping the dataset updated through user interaction.

# 2 Background

## 2.1 Technology Stack

Selecting the technology stack for a project can be dependent on specific requirements of the project, such as the complexity of data, real-time processing needs, interactivity level, and the intended user’s technical proficiency. The project’s supervisor will be the intended user after the completion of this project and hence, the technology stack chosen must suit their technical proficiency and preferences.

The main requirements of the technology stack include ease of use, Python based and freely available. In addition to that, the chosen technologies also takes into consideration the best practices in software development and data science to ensure scalability, maintainability, and ease of use. Their strengths and weaknesses in different areas of the project are considered carefully to enable a successful implementation of this project. The selected technology stack for this project comprises a range of various tools and libraries, each contributing to a specific facet of the project, from data storage and manipulation to predictive analytics and user interaction.

**Dash Framework for Interactive Visualization**

* The Dash framework by Plotly is chosen to present the findings and enable user interaction with the dataset. Considering the intended development language, Dash facilitates the creation of web applications using Python, allowing for the development of interactive, dynamic visualizations without the need for JavaScript. This framework supports the project's goal of making analytical insights accessible to module administrators through intuitive graphs and user interfaces, enabling informed decision-making.4

**Dash Bootstrap Components for Front-end Components**

* Dash Bootstrap Components, a library of Bootstrap components for Plotly Dash, provide a collection of styled components and responsive layouts. This selection enhances the appearance and functional aspects of the user interface and allows for the development of visually appealing, responsive web applications.5

**SQLite Database**

* SQLite was selected due to its lightweight, serverless architecture that provides a easy setup with minimal configuration, making it ideal for a small-medium sized project like this. This choice aligns with the project's need for a simple, efficient and self-contained database solution capable of handling basic queries on module records.6

**SQLAlchemy ORM**

* SQLAlchemy, an Object-Relational Mapping (ORM) library for Python, was chosen as a database management tool. SQLAlchemy provides a high-level abstraction to execute SQL operations through Python objects, enabling seamless data manipulation and retrieval. Its ORM capabilities promotes code maintainability by allowing definition of database models in Python. 7

**Pandas for Data Processing**

* Pandas, a data analysis and manipulation library for Python, plays a crucial role in preprocessing the collected historical module information in the form of dataframes. It allows efficient data cleaning, manipulation, and integration of datasets to prepare for machine learning model training.8

**Scikit-learn for Machine Learning Libraries**

* Scikit-learn, a comprehensive machine learning library for Python, was utilized as it offers a wide array of algorithms for regression and classification. The project leverages Scikit-learn's regression models to forecast PGTA demand based on relevant module information and Scikit-learn's classification model (TF-IDF Vecotoriser) to forecast PGTA demand by classifying features within textual data. Its extensive documentation is also crucial in facilitating model development.9

**NLTK for Textual Data Preprocessing**

* The Natural Language Toolkit (NLTK) is chosen for preprocessing textual data within PGTA job descriptions. It supports tokenization, stemming, lemmatization and stopwords removal, crucial for cleaning unstructured text for use in machine learning models.10

**NumPy for Mathematical Computations**

* NumPy, a library for scientific computing in Python, is used for its efficient handling of arrays and matrices. In the context of this project, it calculates the root mean squared error (RMSE) as a metric to evaluate the performance of predictive models.11

**Pickle for Model Serialization**

* Pickle is utilized for serializing and deserializing Python object structures, specifically trained machine learning models. This allows for the convenient saving and loading of models, ensuring that predictive capabilities can be easily preserved and transferred as needed.12

## 2.2 System Architecture

A diagram of data processing

Description automatically generated

The system’s architecture consists of five main components: Data Collection, Data Preprocessing, Data Visualization, Data Analysis and Prediction Prompt. It presents a streamlined process from data collection to actionable predictions.

### 2.2.1 Data Collection

This process is the foundational phase of the project, where data is sourced from Excel sheets that contain raw data on module assessments, TA allocations, and other relevant metrics necessary for analysis.

### 2.2.2 Data Preprocessing

This phase involves converting the raw data into a structured format suitable for analysis. It consists of two subtasks:

* **Processing Functions**: A series of functions are applied to clean and transform the data into the desired format. There are also new DataFrames being created with datapoints from other DataFrames.
* **SQL Database Integration**: The processed data is then migrated into an SQLite database, providing a centralised data management system.

### 2.2.3 Data Visualisation

The processed data are translated into graphical representations that provide insights into trends and patterns. The visualisations include:

* **Module History**: Showcasing the historical data of PGTAs requested vs recruited.
* **Variables vs PGTA Hours**: Showcasing correlations between various factors such as exam-coursework ratio and the number of PGTAs recruited.
* **Duties vs PGTA Hours**: Showcasing correlations between duties of the PGTAs and the number of PGTAs recruited.

### 2.2.4 Data Analysis

In the analysis phase, the preprocessed data feeds into machine learning models that uncover deeper insights and predict outcomes based on historical trends. The ML models include:

* **Linear Regression Model**: Predicts outcomes based on linear relationships between variables.
* **Ridge Regression Model**: Addresses multicollinearity in the data to explore complex relationships and reduces overfitting.
* **Generalized Additive Model:** Captures non-linear relationships between variables through the use of smooth functions.
* **Feature Engineering Model**: Utilises binary encoding of TA duties, creating a feature vector to be fed into a linear regression model.
* **TF-IDF Vectorizer Model**: Analyses text features from job descriptions using the TF-IDF vectoriser feature from the Scikit-Learn library.

### 2.2.5 Prediction Prompt

This is the interface where users interact with the trained models. It allows the input of relevant parameters to receive instant predictions on various aspects like PGTA hours needed based on the duties and module infomation. The inputs are feeded into the relevant pre-trained ML models to get an output.

# 3 Data Handling and Database Implementation

## 3.1 Data Sources

The main data source of this project is my supervisor. She is involved in the Postgraduate Teaching Assistant (PGTA) recruitment and collects data on module information cruicial for the implementation of this project. The data sheets provided are listed as below and five rows of data in each dataset is displayed for reference:

### 3.1.1 Cap and Actual Students Dataset

* This dataset contains data on the capacity of each module and the number of students who took each module

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Module Code | Module Title | 2023-24 cap | 2022-23 actual students | Notes |
| COMP0002 | Principles of Programming | 170 | 170 | NaN |
| COMP0003 | Theory of Computation | 170 | 171 | NaN |
| COMP0004 | Object-Oriented Programming | 170 | 171 | NaN |
| COMP0005 | Algorithms | 170 | 171 | NaN |
| COMP0007 | Directed Reading | 27 | 18 | NaN |

### 3.1.2 Module Assessment Dataset

* This dataset contains data of each modules on their examinable components such as courseworks, projects, examinations, assessments, etc.

|  |  |  |  |
| --- | --- | --- | --- |
| Module Code | Module Title | Assessment Type Name | Assessment Weight |
| COMP0002 | A4U | Exam (In Person Written) (Centrally Managed) | 90 |
| COMP0002 | A4U | Coursework | 5 |
| COMP0002 | A4U | Coursework | 5 |
| COMP0010 | A5U | Group project | 50 |
| COMP0010 | A5U | Coursework | 50 |

*\* Due to width limitation, columns are simplified and not all columns are included to allow readability \**

### 3.1.3 New Module Dataset

* This dataset contains the list of modules indicating whether each module is a new module in the year 2023. This dataset is not currently being used for analysis or decision-making of any kind but it might be looked upon further.

|  |  |  |
| --- | --- | --- |
| Module Code | Module Title | Is module new in 2023 |
| COMP0008 | Computer Architecture and Concurrency | FALSE |
| COMP0009 | Logic | FALSE |
| COMP0010 | Software Engineering | FALSE |
| COMP0011 | Introductory Mathematics for Computer Science | FALSE |
| COMP0012 | Compilers | TRUE |

*\* Due to width limitation, columns are simplified and not all columns are included to allow readability \**

### 3.1.4 PGTA Requested and Recruited Dataset

* This dataset contains the number of PGTAs requested and recruited for each module from the year 2021 to 2024.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Module Code | 2023-24 requested | 2023-24 recruited | 2022-23 requested | 2022-23 recruited | 2021-22 requested | 2021-22 recruited |
| COMP0002 | 150 | 100 | 150 | 115.5 | 150 | 67 |
| COMP0003 | 140 | 64 | 140 | 124 | 140 | 140 |
| COMP0004 | 360 | 94 | 270 | 166 | 270 | 259 |
| COMP0005 | 180 | 120 | 120 | 134 | 120 | 80 |
| COMP0007 | 70 | 0 | 70 | 70 | 70 | 45 |

*\* Due to width limitation, columns are simplified and not all columns are included to allow readability \**

### 3.1.5 Job Description Dataset

* This dataset contains the job description of modules and the PGTA hours needed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Select module | PGTA hours excl. marking | Marking hours excl. exam | Marking hours for exam | Duties |
| COMP0027 | 120 | 80.0 | 0.0 | Supporting scheduled sessions (computing lab / tutorial / class etc ), Providing student support (e.g. Moodle Q&A, office hours), Preparing lab/tutorial/class activities, Marking - other (e.g. coursework, coding activities, in class tests, formative assessment, etc) |
| COMP0113 | 100 | 60.0 | 0.0 | Supporting scheduled sessions (computing lab / tutorial / class etc ), Providing student support (e.g. Moodle Q&A, office hours), Facilitating student teams (e.g. projects), Preparing lab/tutorial/class activities, Marking - other (e.g. coursework, coding activities, in class tests, formative assessment, etc) |
| COMP0005 | 120 | 60.0 | 0.0 | Supporting scheduled sessions (computing lab / tutorial / class etc ), Marking - other (e.g. coursework, coding activities, in class tests, formative assessment, etc) |
| COMP0147 | 194 | 0.0 | 50.0 | Supporting scheduled sessions (computing lab / tutorial / class etc ), Providing student support (e.g. Moodle Q&A, office hours), Preparing lab/tutorial/class activities, Marking - end of year exam (term 3) |
| COMP0009 | 95 | 0.0 | 0.0 | Supporting scheduled sessions (computing lab / tutorial / class etc ), Providing student support (e.g. Moodle Q&A, office hours), Marking - other (e.g. coursework, coding activities, in class tests, formative assessment, etc) |

*\* Due to width limitation, columns are simplified and not all columns are included to allow readability \**

## 3.2 Data Processing

The provided dataset required extensive preprocessing due to the presence of missing values and data in inconsistent formats. The dataProcessing.py file includes a suite of functions crucial for data preparation, enabling analysis and modeling. Each function plays a role in transforming raw data into a structured format conducive to analytics and predictive modeling, ensuring the usability of the dataset. Here's a detailed look at these functions:

### 3.2.1 Data Processing for Graph Plotting and Linear Regression Model Training

**no\_data\_modules (df, col1, col2)**

* This function identifies modules with 'No data found' entries in specified columns. It returns a list of module codes where either of the two specified columns contains 'No data found'. It is used for listing modules with incomplete data in the PGTAs recruited and requested columns for a selected year.

**split\_coursework\_exam\_ratio\_column (df)**

* Splits the 'Exam:Coursework Ratio' column into two separate columns ('Exam Weight' and 'Coursework Weight') and removes the original ratio column. This is processed so that each of the coursework weights and exam weights are used as features for training the machine learning model.

**handle\_missing\_data (df, columns)**

* Substitutes 'No data found' entries with 0 in specified columns and converts them to numeric format. This ensures data uniformity and prevents computational errors during analysis.

**handle\_nan\_data (df)**

* Fills NaN (not a number) entries with zero, addressing the issue of missing data. This also ensures data uniformity and prevents computational errors during analysis.

**column\_sum (df, column)**

* Calculates and returns the sum of values in a specified column of the DataFrame, providing aggregated data of total students, PGTAs hired and requested for each year.

**column\_average (df, column)**

* Calculates and returns the average of values in a specified column of the DataFrame, providing the average PGTA hours for each duty, allowing direct comparison.

**difference\_calculation (df, selected\_year)**

* Computes the difference between requested and recruited PGTAs for a given year and adds this as a new column in the DataFrame. This feature is pivotal in evaluating recruitment effectiveness and identifying gaps in estimation of PGTA demand for each year.

**set\_colour (df)**

* Assigns a colour based on the difference between requested and recruited PGTAs, with red colour indicating the demand was higher than expected while green colour indicates demand lower than expected, allowing easy data interpretation.

**one\_hot\_encode\_delivery\_code (df)**

* This converts categorical data into numerical format and it is achieved by creating additional columns indicating the presence of each possible value in the original data. After encoding, the original "Delivery Code" column is removed from the DataFrame, and the new one-hot encoded columns are appended.

### 3.2.2 Data Processing for DataFrame Cleaning

**get\_total\_pgta\_hours (df)**

* Calculates the total PGTA hours required for each job description by summing the PGTA hours excluding marking, marking hours excluding end of year exam, and marking hours for the end of year exam. This function modifies the DataFrame with an additional column (PGTA hours) representing the total hours required, including all types of marking hours and removes irrelevant columns to clean the DataFrame.

**split\_module\_code\_and\_name (df)**

* Separates the combined module code and name into two distinct columns: 'Module Code' and 'Module Name' respectively. The module code is extracted by getting the first word separated by a space in the combined string, while the module name is the rest of the string. This transformation makes the data easier to work with when only the module code or module name is needed.

**create\_coursework\_exam\_ratio\_column (df)**

* This function processes a DataFrame to calculate and merge total exam and coursework weights for modules and appends a new column showing these as integer ratio. It also simplifies the DataFrame by removing irrelevant columns and filling missing values. This column is essential to create the ‘Exam:Coursework Ratio vs PGTAs Recruited Graph’.

**create\_combined\_variables\_df (df\_moduleAssessmentData, df\_capVsActualStudents, df\_requestedVsRecruited, df\_jobDescriptionData)**

* Combines data from multiple DataFrames (df\_moduleAssessmentData, df\_capVsActualStudents, df\_requestedVsRecruited, df\_jobDescriptionData) to assemble a new DataFrame based on the 'Module Code', ensuring that only modules present in all source dataframes are included. This new DataFrame includes Module Code, Number of Students, PGTAs Recruited, Exam:Coursework Ratio, Exam Weight, Coursework Weight, Delivery Code and Duties. It is used in plotting the graphs of each individual variables against PGTAs recruited to gain insights on how each variables affect the number of PGTAs recruited.

**create\_df\_average\_pgta\_hours (df, duties)**

* Initializes an empty DataFrame, **df\_averagePGTAHours**, and populates it with a 'Duty' column derived from the input duties list. It then calculates the average PGTA hours for each duty, which is the average of the 'PGTA hours' column from the input df for rows corresponding to that duty, and assigns this list to the 'Average PGTA Hours' column. The function returns a DataFrame mapping each duty to its average PGTA hours, providing a clear view of the workload associated with each duty.

**split\_duties (duty)**

* Splits a string containing multiple duties separated by commas. It uses a stack to keep track of parentheses to avoid incorrect splits for commas within parentheses and returns a list of individual duties.

**get\_set\_of\_duties (job\_desc)**

* Extracts a unique set of duties from the job descriptions, serving as the basis for feature engineering.

**create\_feature\_vector (df, unique\_duties)**

* Transforms the 'Duties' column into a binary feature matrix. Each duty is given its own column, where a 1 indicates the duty's presence in a job description, and a 0 indicates its absence. This is done as part of the feature engineering model training.

**filter\_base\_duty\_in\_duties (df, duty)**

* Filters a DataFrame based on the presence of a specified duty within the 'Duties' column of the DataFrame. It performs a case-insensitive search for the specified duty within each entry of the 'Duties' column, returning a new DataFrame that includes only those rows where the duty is found. The use of ‘*re.escape(duty)’* escapes special characters so that only the actual text of the duty is being matched instead of a regex pattern.

### 3.2.3 Data Processing for TF-IDF Vectorisation

**download\_nltk\_resources ()**

* Downloads the necessary NLTK (Natural Language Toolkit) resources, which include tokenizers, stop words, wordnet, and a POS tagger. It is called before using any NLTK functionality to ensure that the required resources are available.

**tokenize\_text (text)**

* Tokenizes the input text into a list of words or tokens.

**remove\_stopwords (tokens)**

* Removes common words that do not carry significant meaning from a list of tokens.

**get\_wordnet\_pos (word)**

* Maps POS (Part of Speech) tags to the format accepted by the WordNetLemmatizer library for lemmatization.

**lemmatize\_tokens (tokens)**

* Lemmatizes a list of tokens, reducing them to their base or dictionary form.

**preprocess\_text (text)**

* Applies all the preprocessing steps (tokenization, stopword removal, lemmatization) to the input text.

**preprocess\_text\_list (text\_list)**

* Calls the *preprocess\_text* function on a Pandas Series containing text descriptions to be preprocessed.

## 3.3 Database (SQLite)

The implementation of this database is crucial for our predictive models, enabling the analysis of relationships between module characteristics and TA allocation needs. By leveraging a well-defined schema, we ensure that our data is robust and reliable.

### 3.3.1 Database Comparison

SQLite was chosen as the database management system for this project due to its serverless architecture, which provides ease of use and convenience for smaller-scale applications. As a self-contained, file-based database, SQLite provides a lightweight solution without the need for a separate server process, minimising setup and administrative tasks. This also results in low-latency access to the data, as read/write operations do not involve network communication or complex protocols associated with client-server Database Management Systems (DBMS).12

SQLite has limitations in handling concurrent write operations, user management or large-scale data processing needs. However, it shouldn’t be a concern as the requirements of this project is mainly a database that supports simple CRUD operations and this web application is intended to be used by one user only. Hence, SQLite is an excellent choice.6

Traditional SQL databases like PostgreSQL and MySQL are designed for a different set of requirements. They excel in environments where large-scale data handling, complex transactions, and high concurrency are common. They offer extensive features for user management, data security, and advanced query optimizations. However, these systems require more resources in terms of hardware and maintenance. They may introduce more latency due to the client-server model, which can impact the speed of operations.

### 3.3.2 Database Implementation

The SQLite relational database is implemented composing of five tables, each capturing different aspects of module information. The PRIMARY KEY in each table ensures data integrity and provides a unique identifier for each record. These tables and their properties are defined as follows:

|  |  |  |
| --- | --- | --- |
| Job Descriptions | Number of Students | requested\_vs\_recruited |
| PRIMARY KEY (id)  module\_code VARCHAR  module\_title VARCHAR  number\_of\_TA INTEGER  duties VARCHAR  total\_hours INTEGER | PRIMARY KEY (id)  module\_code VARCHAR  module\_title VARCHAR  cap\_23\_24 INTEGER  actual\_22\_23 INTEGER  notes TEXT | PRIMARY KEY (id)  module\_code VARCHAR  module\_title VARCHAR  requested\_23\_24 INTEGER  recruited\_23\_24 INTEGER  requested\_22\_23 INTEGER  recruited\_22\_23 INTEGER  requested\_21\_22 INTEGER  recruited\_21\_22 INTEGER |

|  |  |  |
| --- | --- | --- |
| module\_assessment | average\_pgta\_hours | combined\_variables |
| PRIMARY KEY (id)  module\_code VARCHAR  delivery\_code VARCHAR  module\_delivery\_period\_code VARCHAR  exam\_weight INTEGER  coursework\_weight INTEGER  exam\_coursework\_ratio VARCHAR | PRIMARY KEY (id)  duties VARCHAR  average\_hours | PRIMARY KEY (id)  module\_code VARCHAR  module\_title VARCHAR  number\_of\_students INTEGER  pgtas\_recruited INTEGER  exam\_coursework\_ratio VARCHAR  exam\_weight INTEGER  coursework\_weight INTEGER |

#### 3.3.2.1 SQLAlchemy

SQLAlchemy is a Python SQL toolkit and Object-Relational Mapping (ORM) library, responsible for managing interactions with the SQLite database. It allows developers to work with Python objects rather than SQL queries, enhancing code readability and maintainability.7

By integrating SQLAlchemy with SQLite, the project benefits from the ease of database handling and manipulation using SQLAlchemy's ORM feature, coupled with the efficiency and simplicity of a serverless database system, making it a great setup for Python-based applications with moderate data management requirements.

#### 3.3.2.2 Database Initialisation

This section discusses the database workflow for initialisation.

* **Model Declaration:** SQLAlchemy's ORM feature is used to initiate models such as JobDescription, RequestedVsRecruited, ModuleAssessment, etc. which define the structure of the tables within the SQLite database.
* **Engine Creation:** A database engine is initiated through SQLAlchemy's *create\_engine* method, which establishes the connection to the SQLite database file. The *init\_db* function initialises the database where Base.metadata.create\_all(engine) creates all the defined tables.
* **Session Management:** The session, an instance of the Session class created by the sessionmaker bound to the engine, manages interactions with the database by acting as a staging zone for all objects loaded into the database session.
* **Data Manipulation:** CSV data is directly loaded into the database tables in the *load\_csv\_to\_database* function. This function iterates over a pandas DataFrame, creating instances of the corresponding model populated with data from each row, and adds these instances to the SQLAlchemy session.
* **Session Commitment:** The *session.commit()* method is called to persist all staged changes to the database, translating the object state operations into corresponding SQL statements.
* **Session Closure:** Upon committing the changes, the session is terminated via *session.close()* to free resources and maintain database integrity.

### 3.3.3 Database Interactions

This section shows the interactions between different components within the system architecture when a user adds or deletes a module through the Dash Web Application interface.

A diagram of a machine learning process

Description automatically generated

The figure above illustrates the system overview of module addition and deletion operations

#### 3.3.3.1 Add Module

This operation allows users to add modules into the database, allowing the machine learning models to train on the latest dataset to potentially yield higher accuracy. The processes are documented as below:

**Front End:** The user fills out a form with module details such as code, name, student numbers, and other relevant data. Upon form submission, a HTTP request is generated and sent to the Flask Server at the Back End.

**Back End:** The Flask Server receives the request and performs server-side logic which involves invoking the *insertModule* function. It constructs a *CombinedVariables* object with the provided module details and commits this object to the session, effectively adding it to the SQLite database.

**Database Update**: Following a successful insert operation, the SQLite database is updated to reflect the changes on the Front End, where the newly added module can be seen upon refreshing the tabs of the application.

**Response:** Finally, the Flask Server sends a response back to the Front End in the form of an alert message indicating the success of module addition.

A diagram of a diagram

Description automatically generated

#### 3.3.3.1 Delete Module

This operation allows users to delete modules from the database, often used for data clean-ups or when there is an error in module addition. The processes are documented as below:

**Front End:** The user fills out a form with the module code of the module to be removed. Upon form submission, a HTTP request is generated and sent to the Flask Server at the Back End.

**Back End:** The Flask Server receives the request and performs server-side logic which involves invoking the *deleteModule* function. It retrieves the module from the database using the unique identifier id and removes it from the SQLite database if the id is found. Otherwise, there will be no changes to the database.

**Database Update**: Following a successful delete operation, the SQLite database is updated to reflect the changes on the Front End, where the deleted module can no longer be seen upon refreshing the tabs of the application.

**Response:** Finally, the Flask Server sends a response back to the Front End in the form of an alert message indicating the success or failure of module deletion.

A diagram of a diagram

Description automatically generated

# 4 Graph Analysis

This section discusses the dynamics between the demand for Postgraduate Teaching Assistants (PGTAs) through a series of carefully plotted visualizations. The core components being analysed are the discrepancies between predicted and actual TA needs, the impact of student enrollment numbers, assessment structures, the correlation between module delivery codes, specific duties, and allocated PGTA hours. Each graph not only offers a standalone insight into specific facets of TA needs but also collectively provides a comprehensive overview. This analysis aims to explore the underlying patterns and anomalies within TA recruitment processes, offering a data-driven foundation for enhancing future predictive models.

## 4.1 PGTAs Requested vs PGTAs Recruited Graph

This graph aims to provide insights into which modules have inaccurate estimation of PGTA demand and its scale.

A screenshot of a graph

Description automatically generated

.

.

.

A screenshot of a graph

Description automatically generated

**Analysis**

The bar graph illustrates the comparison between the number of PGTAs requested by modules and the actual number recruited. This visualization highlights modules with significant discrepancies, which suggests a potential misalignment between anticipated and actual TA needs. Modules with a higher number of recruited PGTAs than requested may indicate either an underestimation of TA requirements or an adjustment to increased student demand or course complexity.

## 4.2 Module History Graph

Building upon the previous graph, this visualization presents an alternative view of the history of PGTAs requested vs PGTAs recruited for a selected module from the dropdown menu, allowing direct comparison between different modules. It also shows the total number of PGTAs recruited and requested across all modules for the past three years.

A screenshot of a graph

Description automatically generated

## 4.3 Students Enrolled vs PGTAs Recruited Graph

This graph aims to determine whether the number of students enrolled affect the number of PGTAs recruited for the academic year 2022-2023 (only data for one year can be found)

A graph with blue dots

Description automatically generated

**Analysis**

The scatter plot suggests no clear linear relationship between the number of students enrolled in a module and the number of PGTAs recruited, indicating that student numbers are not the sole determinant of TA recruitment. Several outliers suggest that other factors such as module complexity or administrative decisions may play a role.

## 4.4 Exam-Coursework Ratio vs PGTAs Recruited Graph

This graph aims to determine whether the weight of coursework and exams affect the number of PGTAs recruited.

A graph with numbers and text

Description automatically generated

**Analysis**

This graph shows a distribution of PGTAs recruited across different exam-coursework ratios. A higher concentration of points towards extreme ratios (e.g., 100:0 or 0:100) may suggest that modules with a singular assessment focus either on exams or coursework might have more predictable TA needs. However, the spread across the spectrum indicates that the ratio alone does not dictate TA recruitment patterns.

## 4.5 Module Delivery Code vs PGTAs Recruited Graph

This graph aims to determine whether module delivery code (modules taken by students in different years of study. In other words, difficulty of the module) affect the number of PGTAs recruited.

A white box with blue dots

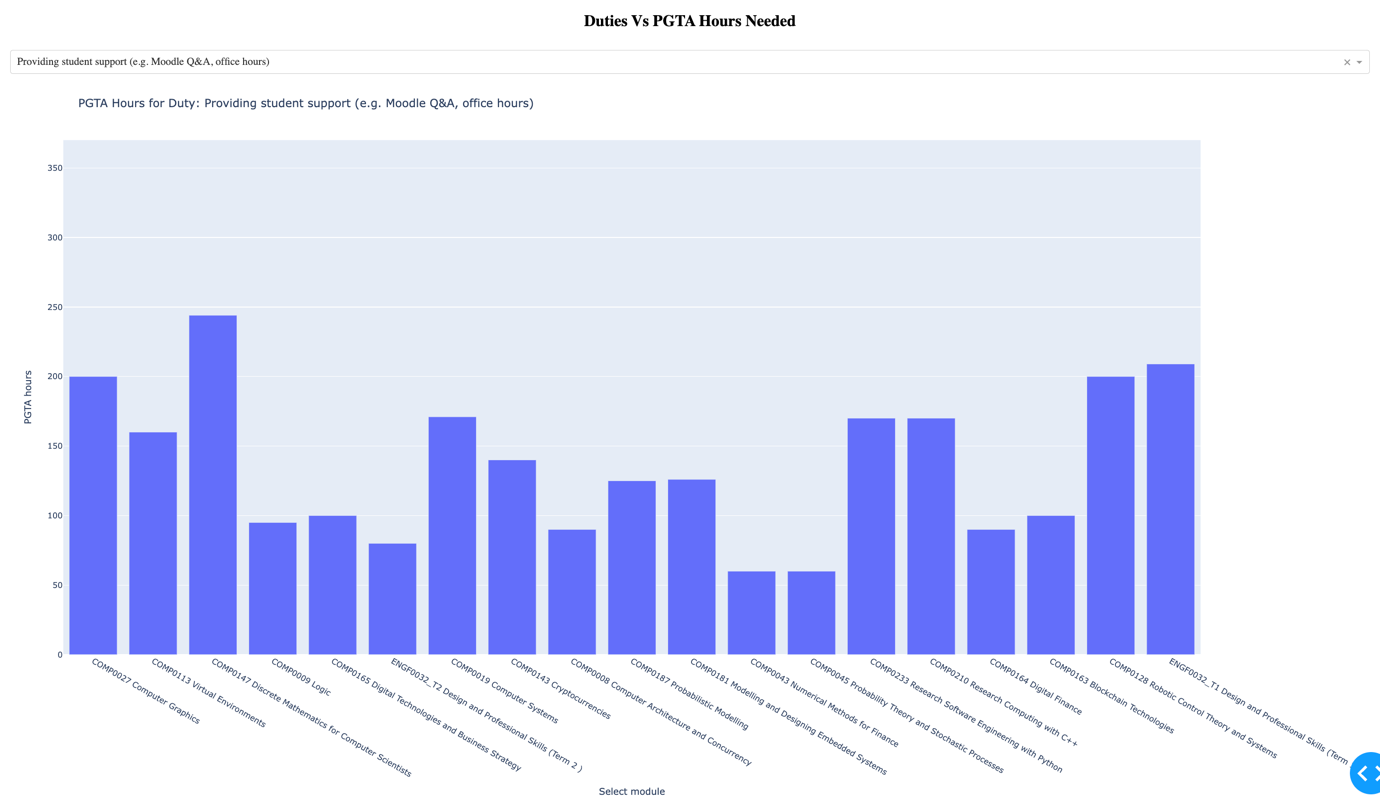
Description automatically generated

**Analysis**

The final graph examines the relationship between module delivery codes and the number of PGTAs recruited. Variability across delivery codes could suggest that the nature of the module, potentially its complexity or the year of study it's catered to, might influence TA recruitment. However, the data presents a diverse pattern, suggesting a more intricate interplay of factors beyond just the delivery code.

## 4.6 Duties vs PGTA Hours Graph

This graph aims to determine the relationship between specific duties assigned within modules and the corresponding PGTA hours allocated.



**Analysis**

The 'Duties' column encompasses a mix of six base duties:

*1. Providing student support (e.g. Moodle Q&A, office hours)*

*2. Facilitating student teams (e.g. projects)*

*3. Marking - other (e.g. coursework, coding activities, in class tests, formative assessment, etc)*

*4. Marking - end of year exam (term 3)*

*5. Preparing lab/tutorial/class activities*

*6. Supporting scheduled sessions (computing lab / tutorial / class etc )*

For each selected duty from the dropdown menu, a bar graph shows the PGTA hours for all modules containing the selected duty in its job description. This visualization helps identify which duties demand more or less PGTA time, reflecting on the potential intensity or complexity of these duties. However, the variation in PGTA hours for modules sharing common duties indicates that direct comparison of duty intensity is not straightforward. To address this, an aggregated view is plotted in the subsequent graph.

## 4.7 Duties vs Average PGTA Hours Graph

Building upon the previous graph, this visualization presents the average PGTA hours for modules that contain each base duty, streamlining direct comparisons across duties.

A graph of blue rectangular shapes

Description automatically generated with medium confidence

**Analysis**

The graph reveals that the duty 'Supporting scheduled sessions (computing lab / tutorial / class etc.)' requires the most PGTA hours on average, while 'Facilitating student teams (e.g., projects)' demands the least. This insight is pivotal for refining our prediction model; understanding the differential demands of various duties allows for more accurate forecasts of PGTA hour requirements.

## 4.8 Results Evaluation and Conclusion

Across all visualizations, it is evident that predicting TA demand is multifaceted and complicated, with no single variable providing a complete understanding. The disparities in requested versus recruited PGTAs highlight the need for refined predictive models that can accommodate a range of influencing factors. The lack of a strong correlation in some graphs suggests that further qualitative data, such as module content analysis or instructor feedback, might be necessary to fully understand TA recruitment needs. These analyses underscore the complexity of academic resource planning and the potential for data-driven approaches to enhance decision-making processes.

The visualizations provide valuable insights into the factors influencing TA recruitment. However, they also highlight the complexities in academic administration and the potential benefits of adopting comprehensive analytic strategies to improve accuracy in PGTA demand.

# 5 Machine Learning Models Implementation

This section discusses the machine learning models implemented to conduct analysis with documented results and evaluation.

## 5.1 Regression Models

### 5.1.1 Introduction

**Linear Regression** is a statistical method that models the relationship between a scalar dependent variable y and one or more independent variables (predictors) X. The core idea is to obtain a linear equation that predicts y as accurately as possible from X. It is based on the ordinary least squares (OLS) estimator and assumes a linear relationship between the variables.14

**Ridge Regression** is an extension of linear regression where the loss function is modified to minimize the complexity of the model. This is achieved by adding a penalty term to the OLS loss function, which is proportional to the square of the magnitude of the coefficients. Ridge Regression aims to prevent overfitting by penalizing large coefficients.15

### 5.1.2 Regression Model Training

The regression models aims to predict the number of PGTAs recruited based on features like the number of students, exam weight, coursework weight, and the one-hot encoded delivery codes.

**One-Hot Encoding**

One-hot encoding is a process that converts categorical data variables into a binary vector representation to be used in machine learning algorithms. Before training, the *one\_hot\_encode\_delivery\_code* function converts categorical data into a numerical format that can be fed into a regression model by adding multiple columns representing different delivery codes (e.g., 'delivery\_code\_A4U', 'delivery\_code\_A5U') and setting them to 1 or 0 based on the presence of the code in the original row.17

**Model Training**

The linear regression model is instantiated and trained using the scikit-learn **LinearRegression** class while the ridge regression model is instantiated and trained using the scikit-learn **Ridge** class. Training involves fitting the models to the feature set **X** and target variable **y**, where **X** includes the features mentioned above and **y** is the 'pgtas\_recruited' column from the DataFrame.

**Model Saving**

After training and evaluation, the trained regression models are saved to disk as **.pkl** files using the **save\_model** function. This enables the model to be easily loaded and reused for predictions without the need for retraining.

### 5.1.3 Results Evaluation

**Cross-Validation**

A model usually creates a prediction function from the same set of data, which might lead to a bias towards that dataset. This might produce inaccuracies when prompted with un-seen data (a situation called overfitting). As the datasets provided in this project are small, they are more prone to overfitting and necessary measures are needed to prevent that. Cross-validation (CV) solves this problem by using the K-fold strategy. The dataset is split into k smaller datasets and the model is trained and tested k times on k-1 folds and 1-fold respectively. E.g. a CV with 5 folds running on a dataset of 100 data will produce 20 data per fold. Cross-validation is computationally done using the cross\_val\_score helper function on the dataset by splitting the data, fitting a model, and computing the score for k number of consecutive times (with different splits each time).

One way to assess how well a regression model fits a dataset is to calculate the root mean square error (RMSE), which is a metric that tells us the average distance between the predicted values from the model and the actual values in the dataset. The lower the RMSE, the better a given model is able to “fit” a dataset22 whereas the Standard deviation (SD) is the measure of dispersion, or how spread out values are, in a dataset.23

The RMSE and SD are used as performance metrices, which provides a measure of the model's prediction error. The results of each of the selected number of folds are documented below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| K-folds | Linear Regression | | Ridge Regression | |
| Mean RSME | Standard Deviation | Mean RSME | Standard Deviation |
| 3 | 84.05 | 8.41 | 73.90 | 9.62 |
| 5 | 85.7 | 22.35 | 72.67 | 9.56 |
| 10 | 75.47 | 32.61 | 69.90 | 28.97 |
| 15 | 63.84 | 42.42 | 60.31 | 39.53 |

A low Mean RMSE value indicates that the model, on average, has a low prediction error, directly relates to the model's accuracy. A low SD value suggests that the model's performance is consistent across different subsets of the dataset, crucial for ensuring that the model is reliable and not just performing well on specific types of data. As the number of folds increased from 3 to 15, both models exhibited a decrease in mean RMSE but an increase in standard deviation, indicating improved prediction accuracy but worse consistency with more cross-validation folds.

The cross-validation folds for each models are chosen by considering the trade-off between accuracy and consistency as it is crucial to achive reliable model performance while maintaining robustness across diverse data samples. Hence, the Ridge Regression model is set to train with 5 folds at mean RMSE of 72.67 and standard deviation of 9.56 while the Linear Regression model is set to train with 3 folds at mean RMSE of 84.05 and standard deviation of 8.41.

## 5.2 Natural Language Processing

**Introduction**

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and humans through natural language. One of the applications of NLP in educational settings involves analysing the job descriptions of Postgraduate Teaching Assistants (PGTAs) to gauge how their duties might influence its demand. It uses techniques that extract key features and entities from textual data to enable predictive capabilities. This model aims to accurately forecast the required number of teaching assistant hours for a module over the period of a university year.

The implementation of NLP models can be a complex process and a successful implementation requires the careful selection of data, the choice of modeling techniques, and the adoption of appropriate preprocessing methods. Given the diversity and complexity of textual data encountered in PGTA job descriptions, linguistic analysis is needed before feeding data into a machine learning model. The project leverages Natural Language Toolkit (NLTK) is used for comprehensive text analysis and manipulation as it excels in providing tools for detailed text processing and the scikit-learn library for the subsequent model development and evaluation phases.

### 5.2.1 Prediction Using Term Frequency-Inverse Document Frequency (TF-IDF)

#### 5.2.1.1 Introduction

TF-IDF is a statistical measure used to evaluate the importance of a word to a document in a collection or corpus. It is often used in text mining and information retrieval to weigh the frequency of words by how common they are across documents, thus helping to adjust for the fact that some words appear more frequently in general.24

**Term Frequency (TF):** This measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization:

**Inverse Document Frequency (IDF):** This measures how important a term is. While computing TF, all terms are considered equally important. However, certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scaling up the rare ones with the logarithm scale used to dampen the effect of IDF. If a term appears in all documents, its IDF value becomes 0 (since log(1) = 0), showing that the term is not unique or important. This is implemented by computing the following:

The TF-IDF value is simply the multiplication of TF and IDF:

This value increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words are generally more common than others.16

#### 5.2.1.2 Data Preprocessing

This step involves applying text preprocessing to the "Duties" column, which includes tokenization, stopwords removal, stemming/lemmatization, and vectorization. This process converts the text into a numerical format that can be used by machine learning models.

**Tokenization19**

* Tokenization is the process of splitting text into units called tokens, which can be words, sentences, or subwords. This is the first step in text preprocessing and is crucial for understanding the structure of the text. This is done using nltk.tokenize. word\_tokenize.

**Stopwords removal20**

* Stopwords are common words like "and", "the", "is", etc., that are usually removed because they appear frequently in the text but don't carry significant meaning for analysis or modeling. This is done using nltk.corpus.stopwords to get a list of stopwords for English.

**Stemming and lemmatization21**

Both processes aim to reduce words to their base or root form, but they do so differently:

* Stemming chops off word prefixes and suffixes indiscriminately, which might result in non-existent word forms but reduces the complexity of the textual data.
* Lemmatization considers the morphological analysis of the words, aiming to remove inflectional endings only and return the base or dictionary form of a word, known as the lemma.
* This is done using nltk.stem.PorterStemmer and nltk.stem.WordNetLemmatizer

**Vectorization21**

Vectorization is the process of converting text into numerical data (vectors) so that machine learning algorithms can understand it. The TF-IDF method is used by importing the The sklearn.feature\_extraction.text.TfidfVectorizer library because it fits the dataset.

#### 5.2.1.3 Implementation

**Model Training**

Model training is done by first applying all of the preprocessing steps above. The data is then divided into training and test sets with cross-validation to ensure the model’s performance is robust across different subsets of the data. The model is trained using the vectorized "Duties" as input features and "PGTA hours excluding marking" as the target variable.

**Model Saving**

After training and evaluation, the trained model is saved to disk as a **.pkl** file using the **save\_model** function. This enables the model to be easily loaded and reused for predictions without the need for retraining.

**5.2.1.4 Results Evaluation**

A comparative analysis of both regression models is conducted to predict the required PGTA hours based on job descriptions. The objective is to determine which model and number of cross-validation folds would provide the most accurate predictions as measured by the Root Mean Squared Error (RMSE) and Standard Deviation (SD) across a different folds of cross-validation. The results are documented as below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| K-folds Cross Validation | Linear Regression | | Ridge Regression | |
| Mean RSME | Standard Deviation | Mean RSME | Standard Deviation |
| 3 | 84.30 | 16.72 | 67.58 | 3.59 |
| 5 | 66.37 | 20.36 | 57.89 | 21.25 |
| 10 | 61.76 | 31.45 | 55.76 | 28.48 |
| 15 | 55.88 | 41.86 | 51.67 | 34.77 |

The analysis employed Root Mean Squared Error (RMSE) as a measure of model accuracy, alongside its standard deviation to gauge consistency. As the number of folds increased from 3 to 15, both models exhibited a decrease in mean RMSE but an increase in standard deviation, indicating improved prediction accuracy but worse consistency with more cross-validation folds. Ridge Regression consistently outperformed Linear Regression across all fold partitions, demonstrating lower mean RMSEs and standard deviation ranges. This suggests that Ridge Regression's regularisation techniques are effective in handling multicollinearity within text-based features.

The Ridge Regression model with 5 folds as the training strategy yields the best combination of mean RMSE (57.89) and standard deviation (21.25) when considering the trade-off between accuracy and consistency. This is a balanced model in terms of achieving reliable model performance and maintaining robustness across diverse data samples.

### 5.2.2 Prediction Using Feature Engineering

#### 5.2.2.1 Introduction

Feature Engineering is a pivotal aspect of the predictive modeling process, especially when dealing with natural language data such as job descriptions. Binary encoding is a form of feature engineering where textual data is converted into a numerical format that machine learning models can interpret and learn from. Binary encoding is particularly suited for this context because the job descriptions may involve a wide variety of base duties, but each description mentions only a subset. Binary encoding efficiently represents this sparse data. Additionally, it allows for straightforward comparisons across job descriptions, enabling analysis of which duties commonly lead to higher or lower PGTA hours.

#### 5.2.2.2 Implementation

**Creating Feature Vector**

This step involves applying text preprocessing to the "Duties" column. The *create\_feature\_vector* function transforms the list of duties into a binary vector, indicating the presence (1) or absence (0) of each duty in job descriptions. It converts the text into a numerical format that can be used by machine learning models.

**Model Training**

With the features predefined, the chosen model is fitted to the feature set **X** and target variable **y**, where **X** is the list of duties and **y** is the 'pgtas\_recruited' column from the DataFrame.

**Model Saving**

After training and evaluation, the trained model is saved to disk as a **.pkl** file using the **save\_model** function. This enables the model to be easily loaded and reused for predictions without the need for retraining.

#### 5.2.2.3 Results Evaluation

A comparative analysis of both regression models is conducted to predict the required PGTA hours based on job descriptions. The objective is to determine which model and number of cross-validation folds would provide the most accurate predictions as measured by the Root Mean Squared Error (RMSE) and Standard Deviation (SD) across a different folds of cross-validation. The results are documented as below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| K-folds Cross Validation | Linear Regression | | Ridge Regression | |
| Mean RSME | Standard Deviation | Mean RSME | Standard Deviation |
| 3 | 84.30 | 16.72 | 68.93 | 4.07 |
| 5 | 66.37 | 20.36 | 58.32 | 21.91 |
| 10 | 61.76 | 31.45 | 56.26 | 30.51 |
| 15 | 55.88 | 41.86 | 51.27 | 40.32 |

The analysis employed Root Mean Squared Error (RMSE) as a measure of model accuracy, alongside its standard deviation to gauge consistency. As the number of folds increased from 3 to 15, both models exhibited a decrease in mean RMSE but an increase in standard deviation, indicating improved prediction accuracy but worse consistency with more cross-validation folds. Ridge Regression consistently outperformed Linear Regression across all fold partitions, demonstrating lower mean RMSEs and similar standard deviation ranges.

The Ridge Regression model with 5 folds as the training strategy yields the best combination of mean RMSE (58.32) and standard deviation (21.91) when considering the trade-off between accuracy and consistency. This is a balanced model in terms of achieving reliable model performance and maintaining robustness across diverse data samples.

## 5.3 Generalised Additive Model

### 5.3.1 Introduction

This section dives into another method of PGTA recruitment predictions by deploying a Generalized Additive Model (GAM). GAMs offer a more flexible approach to modeling complex, non-linear relationships in data compared to linear regression, which is more straightforward but less adaptable to non-linearity. These models excel in handling the intricacies of data by allowing for the inclusion of non-linear effects of variables through smooth functions.

### 5.3.2 GAM and Linear Regression Comparison

The comparison between GAMs and Linear Regression models is crucial in understanding the flexibility and applicability of these statistical techniques in various data analysis scenarios. Here's a detailed examination of their differences, along with evaluations:

|  |  |  |
| --- | --- | --- |
|  | GAM | Linear Regression |
| Flexibilty and Complexity | Offers the flexibility to model complex, non-linear relationships between predictors and the response variable.GAM manages complexity via smoothing functions and regularization | A structurally simpler model that assumes a linear relationship between these elements​​, which can be a limiting factor for accuracy if the true relationship is non-linear. |
| Regularization | Can inherently include regularization techniques to control model complexity | Requires external regularization methods like Ridge or Lasso |
| Sample Size Requirements | Often require larger sample sizes to capture non-linear trends effectively | Less demanding due to its simplicity​​ |
| Visualizations | Visualization of smooth functions in GAMs can be particularly helpful in understanding the relationships within the data | Difficult to be represented in graphs or other visual representations |

### 5.3.3 Model Training

A GAM is constructed to predict PGTA requirements, with input features of student numbers, exam weight, coursework weight and module delivery codes. The dataset was partitioned into training and testing subsets, maintaining a test size of 20%. The GAM is then fitted on the training data, leveraging spline functions to capture non-linear relationships.

### 5.3.4 Results and Evaluation

The analysis revealed a RMSE score of 45.76, lower than the mean RMSE of both linear and ridge regression models across various k-folds cross-validation setups. While linear and ridge regression models provide valuable insights, their assumption of linearity restricts their adaptability to complex data dynamics. This improvement in prediction accuracy demonstrates GAM's superior capability to mold itself around the data's structure, offering a more accurate analysis of the factors influencing PGTA recruitment needs. This result also highlights the importance of choosing models that align with the data's characteristics (non-linear data relationships).

## 5.4 Prediction Prompts

This section documents the implementation of a user-friendly interface that allows for user input to get an estimate of the PGTA hours required by feeding the input into pre-trained machine learning models.

### 5.4.1 Linear Regression Prompt

This interface utilizes a trained Linear Regression model for prediction. The prompt is part of a web application built using the Dash framework, enabling interactive and user-friendly prediction based on input variables related to a module. The prompt provides fields for users to input relevant predictors such as the number of students enrolled in a module, the weight of exams, the weight of coursework, and the delivery code associated with the module.

A diagram of a graph

Description automatically generated

**A screenshot of a survey

Description automatically generated**

When the predict button within the Dash interface is clicked (indicated by n\_clicks > 0), the *linearRegressionPredictor* function proceeds to create a DataFrame with the provided inputs. The Delivery Code input is one-hot encoded to match the format used during model training. The function also ensures that all feature columns expected by the model are present in the input DataFrame, adding any missing columns with a default value of 0. The columns are reordered to match the training data's order. The Linear Regression model loaded from the saved .pkl file is then used to predict the number of PGTAs based on the input data.

### 5.4.2 Ridge Regression Prompt

This interface utilizes a trained Ridge Regression model, which is a variant of linear regression with regularization to prevent overfitting. The prompt is part of a web application built using the Dash framework, enabling interactive and user-friendly prediction based on input variables related to a module. The prompt provides fields for users to input relevant predictors such as the number of students enrolled in a module, the weight of exams, the weight of coursework, and the delivery code associated with the module.

A diagram of a computer

Description automatically generated

**A screenshot of a quiz

Description automatically generated**

When the predict button within the Dash interface is clicked (indicated by n\_clicks > 0), the *ridgeRegressionPredictor* function proceeds to create a DataFrame with the provided inputs. The Delivery Code input is one-hot encoded to match the format used during model training. The function also ensures that all feature columns expected by the model are present in the input DataFrame, adding any missing columns with a default value of 0. The columns are reordered to match the training data's order. The Ridge Regression model loaded from the saved .pkl file is then used to predict the number of PGTAs based on the input data.

### 5.4.3 Generalized Additive Model Prompt

This interface utilizes a trained Generalized Additive Model (GAM) for prediction. The prompt is part of a web application built using the Dash framework, enabling interactive and user-friendly prediction based on input variables related to a module. The prompt provides fields for users to input relevant predictors such as the number of students enrolled in a module, the weight of exams, the weight of coursework, and the delivery code associated with the module.

A diagram of a diagram

Description automatically generated

A white background with black text

Description automatically generated

When the predict button within the Dash interface is clicked (indicated by n\_clicks > 0), the *gamPredictor* function proceeds to create a DataFrame with the provided inputs. The Delivery Code input is one-hot encoded to match the format used during model training. The function also ensures that all feature columns expected by the model are present in the input DataFrame, adding any missing columns with a default value of 0. The GAM model loaded from the saved .pkl file is then used to predict the number of PGTAs based on the input data.

### 5.4.4 TF-IDF Vectorisation Prompt

TF-IDF vectorization can extract semantic features from job descriptions, which are effective in forecasting outcomes like the estimated hours required for a task. The *vectoriserPredictor* function is designed to handle user interactions with a web application built with the Dash framework. The function aims to predict the number of Postgraduate Teaching Assistants (PGTAs) hours required based on a set of selected duties using a pre-trained machine learning regression model.

A diagram of a person's work flow

Description automatically generated

A black text on a white background

Description automatically generated

When the predict button within the Dash interface is clicked (indicated by n\_clicks > 0), the *vectoriserPredictor* function concatenates the user-selected duties into a single string. This string undergoes preprocessing where it is tokenized, cleaned of stopwords, lemmatized, and then reconstructed into a processed string. This preprocessed text is then vectorized using a TF-IDF vectorizer, a critical step that translates the text into a numerical form, necessary for the regression model's input.

### 5.4.5 Feature Engineering Prompt

The feature engineering process begins with the collection of duties described in job descriptions for PGTAs. Each duty is encoded as a binary feature indicating its presence or absence in the description. A predictive linear model is then applied to this binary vector to forecast the required PGTA hours.

The *featureEngineeringPredictor* function is designed to handle user interactions with a web application built with the Dash framework. The function aims to predict the number of Postgraduate Teaching Assistants (PGTAs) hours required based on a set of selected duties using a pre-trained machine learning regression model.

A diagram of a diagram

Description automatically generated

A white background with black text

Description automatically generated

When the predict button within the Dash interface is pressed (indicated by n\_clicks > 0), the function initialises a dictionary with all possible duties set to 0. It then updates this dictionary to set the selected duties to 1, creating a binary representation of the duties. A pandas DataFrame (input\_df) is created from the dictionary, which serves as the input for the machine learning model. The function ensures that the input DataFrame contains all the features that the model was trained on. If any are missing, it adds these features to the DataFrame and sets them to 0, indicating the absence of those duties in the current prediction context. The columns in the input DataFrame are reordered to match the order expected by the trained model, which is necessary for the model to make an accurate prediction. The model's predict method is called with the prepared DataFrame, and the first prediction is accessed. This prediction indicates the estimated number of PGTAs required and is displayed in the Dash application.

### 5.4.6 Architecture Overview

A diagram of data processing

Description automatically generated

The diagram illustrates the architecture of machine learning prediction prompts. The Front End displays the input fields for input features and a ‘predict’ button for submission. The ‘predict’ buttons on Front End triggers HTTP requests to the Back End, where raw data is processed and fetch the pre-trained and saved machine learning models. The result of the prediction is then displayed back to the Front End

# 6 Tests

The implementation of both integration and unit tests plays a pivotal role in the development lifecycle of our application. These tests facilitate the identification and correction of bugs during the development process. By asserting expected outcomes against actual results, these tests also play a crucial role in the scalability and maintainability of the application, ensuring reliability and consistency in the system's data-driven insights and predictions. The testing framework supports the continuous enhancement of the application by ensuring new features and modifications do not compromise existing functionalities.

## 6.1 Unit Tests

Unit tests are implemented to verify the functionality and correctness of individual components and utilities within the application. These tests cover all data processing functions.25

### 6.1.1 Graph Plotting and Linear Regression Model Training Functions

**test\_no\_data\_modules**

* Verifies that the function *no\_data\_modules* correctly identifies modules with missing data in specified columns within a DataFrame.

**test\_split\_coursework\_exam\_ratio\_column**

* Verifies that the function *split\_coursework\_exam\_ratio\_column* correctly splits an "Exam:Coursework Ratio" column into separate "Exam Weight" and "Coursework Weight" columns.

**test\_handle\_missing\_data**

* Verifies that the *handle\_missing\_data* function correctly replaces instances of "No data found" with zeros in specified columns.

**test\_handle\_nan\_data**

* Verifies that the *handle\_nan\_data* function correctly replaces NaN values within a DataFrame with zeros.

**test\_column\_sum**

* Verifies that the *column\_sum* function correctly calculates the sum of values for specified columns within a DataFrame.

**test\_column\_average**

* Verifies that the *column\_average* function correctly computes the average of numerical values in specified columns.

**test\_difference\_calculation**

* Verifies that the *difference\_calculation* function correctly computes the difference between two specified columns.

**test\_set\_colour**

* Verifies that the *set\_colour* function correctly assigns color based on the sign of the values in the "Difference" column.

**test\_load\_regression\_data**

* Verifies that the *one\_hot\_encode\_delivery\_code* function correctly loads and prepares data for regression analysis.

### 6.1.2 DataFrame Cleaning Functions

**test\_get\_total\_pgta\_hours**

* Verifies that the *get\_total\_pgta\_hours* function correctly computes total PGTA hours by summing individual hour columns in a DataFrame.

**test\_split\_module\_code\_and\_name**

* Verifies that the *split\_module\_code\_and\_name* function correctly separates combined module code and name strings into individual columns.

**test\_create\_coursework\_exam\_ratio\_column**

* Verifies that the *create\_coursework\_exam\_ratio\_column* function correctly generates a combined ratio column from separate assessment weights.

**test\_create\_combined\_variables\_df**

* Verifies that the *create\_combined\_variables\_df* function correctly combines multiple DataFrames into one, with all expected columns correctly populated.

**test\_create\_df\_average\_pgta\_hours**

* Verifies that the *create\_df\_average\_pgta\_hours* function correctly creates the Average PGTA Hours DataFrame, with all expected values correctly populated.

**test\_split\_duties**

* Verifies that the *split\_duties* function correctly splits concatenated duties separated by commas into individual list items.

**test\_get\_set\_of\_duties**

* Verifies that the *get\_set\_of\_duties* function correctly extracts the unique set of duties from a DataFrame column.

**test\_create\_feature\_vector**

* Verifies that the *create\_feature\_vector* function correctly creates a binary feature vector for categorical duties.

**test\_filter\_base\_duty\_in\_duties**

* Verifies that the *filter\_base\_duty\_in\_duties* function correctly filters duties based on a specified base duty.

### 6.1.3 Text Preprocessing Functions

**test\_tokenize\_text**

* Verifies that the *tokenize\_text* function correctly implements the tokenization process for converting strings of text into tokens.

**test\_remove\_stopwords**

* Verifies that the *remove\_stopwords* function correctly removes common stopwords from a list of tokens.

**test\_get\_wordnet\_pos**

* Verifies that the *get\_wordnet\_pos* function correctly assigns the correct WordNet POS tag to words.

**test\_lemmatize\_tokens**

* Verifies that the *lemmatize\_tokens* function correctly lemmatizes of a list of tokens to their base or dictionary forms.

**test\_preprocess\_text**

* Verifies that the *preprocess\_text* function correctly implementes the complete text preprocessing pipeline, ensuring that text is tokenized, stopwords are removed, and tokens are lemmatized correctly.

## 6.2 Integration Tests

Integration tests are implemented to validate the interaction between different components of the application, ensuring that they work together as expected. These tests include database operations and machine learning predictions.26

### 6.2.1 Database Operations

Tests were implemented to validate the functionality of database operations, including the insertion, deletion, and fetching of module data. This ensures the integrity of data management within the application.

**test\_insert\_module**

* Validates the insertion of a new module into the database.

**test\_delete\_module**

* Validates the deletion of a module from the database.

**test\_fetch\_data**

* Validates the retrieval of module data from the database.

### 6.2.2 Machine Learning Predictions

Tests were implemented to assess the accuracy and reliability of the machine learning models integrated into the application, tests were conducted for each model's prediction capabilities.

**test\_linear\_regression\_prompt**

* Validates the ridge regression model's functionality and ensures the model produces predictions within the expected bounds.

**test\_ridge\_regression\_prompt**

* Validates the ridge regression model's functionality and ensures the model produces predictions within the expected bounds.

**test\_generalized\_additive\_model\_prompt**

* Validates the generalized additive model's functionality and ensures the model produces predictions within the expected bounds.

**test\_feature\_engineering\_prompt**

* Validates the feature engineering model's functionality and ensures the model produces predictions within the expected bounds.

**test\_vectoriser\_prompt**

* Validates the text vectorizer model's functionality and ensures the model produces predictions within the expected bounds.

# 7 Conclusion

The conclusion involves reflecting on the objectives set out at the beginning, evaluating the outcomes achieved, and considering the implications for future work. This project was set out to utilise machine learning techniques and data analytics to predict the demand for Postgraduate Teaching Assistants (PGTAs) in a university setting, specifically within the Computer Science Department of University College London (UCL). The goal was to bring clarity and efficiency to the PGTA recruitment process, enhancing the educational experience for both undergraduate students and educators.

## 7.1 Achievements

**Achievements**

1. **Interactive User Interface**: Developed a user-friendly web interface with Dash, enabling database manipulation, interface switching (changing tabs and dropdowns for graph visualisations) and alerts.
2. **Offline Functionality**: Enables offline useage with all necessary resources stored locally, making it always available and accessible.
3. **CRUD Operations**: Successfully implemented Create, Read, Update, and Delete (CRUD) operations, allowing users to manage the dataset directly from the browser.
4. **Implementation of Machine Learning Models**: Integrated multiple machine learning models, including Linear Regression, Ridge Regression, feature engineering and TF-IDF Vectorizer for predictive analytics.
5. **Implementation of Statistical Model**: Integrated a Generalized Additive Model (GAM) for predictive analytics.
6. **Data Preprocessing Pipeline**: Established a comprehensive data preprocessing pipeline that prepares data for graph plotting, machine learning model training and text pre-processing.
7. **Automated Testing**: Created a suite of unit tests and integration tests that ensure the integrity and accuracy of data processing functions and functionality of the application.
8. **Real-Time Data Processing**: Enabled real-time prediction capabilities, allowing for immediate insights and predictions based on user input.
9. **Database Connectivity**: Implemented seamless SQLite database integration for persistent data storage and retrieval.

**Non-Achievements**

1. **Cloud Integration**: The application does not use any cloud-based services due to the project's offline usability requirement and for simplicity.
2. **Live Data Feeds**: The application does not support live data feeds or real-time data streaming from external sources as it is currently intended for one user only and database syncing is not required.
3. **User Account Management**: The app lacks a user account management system for personalised interfaces and operations. This can be implemented in the future
4. **Mobile Responsiveness**: The application is not supported on mobile devices, preventing usability on smartphones and tablets. This
5. **Advanced Analytics Features**: Advanced features like predictive model tuning, ensemble methods, or deep learning were not incorporated.
6. **Dynamic Model Retraining**: The system does not support dynamic retraining of machine learning models with new data without manual intervention.
7. **Scalability**: Scalability to handle very large datasets or to serve a large number of concurrent users is not addressed at this stage of development as it is not required.
8. **Deployment**: The project is not deployed to a server as it is not required. Updates and maintenance requires manual processes.

## 7.2 Limitations and Challenges

Despite achieveing many of it objectives, the project also faced a few constraints, particularly in data limitations and the complex nature of predicting PGTA demand solely based on module information.

One notable limitation is the feature engineering model's inaccurate estimation of PGTA hours due its simplicity. Having an input of six duties resulted in lower predicted PGTA hours compared to four duties or lesser. This is contrarary to expectations as the model is trained predominantly on data with three or four duties, so it struggles to appropriately scale predictions for six duties. Implementing a count vector that factors in the quantity of duties to refine the predictions as part of the model training processes could address this issue.

Additionally, while the models successfully predicted the pgta demand, its accuracy is not the best due to the limitation in data collection. Most of the historical data for modules within the Univeristy College London Computer Science Department is unstructured and non-existent and hence, the model has way lesser data to work on than originally intended. However, the insert module operation aims to solve this problem as the models will get more accurate as the database scales.

## 7.3 Future Work

There are huge potential to extend the impact of this application which includes incorporating additional data sources (student feedback, module timetable structure) to xallow for more sophisticated machine learning models and increase their prediction capabilities. Collaborations with other educational researchers or module administrators outside of the University College London’s Computer Science Department could enhance data collection to provide a larger integrated database to allow for better accuracy. This would require implementation of account login systems and proper access controls. Deploying this application and scaling it to be used as part of universities’ administrative systems could offer great value in educational settings.

The application has significant potential for further development. Future enhancements may include integrating additional datasets, such as student feedback and module scheduling details, to allow for more sophisticated machine learning algorithms and improve predictive accuracy. Extending collaboration to include educational researchers or department administrators beyond the University College London's Computer Science Department could improve data collection, creating a more expansive and varied database. This expansion would require the implementation of user accounts and robust access controls. Deploying and scaling this application across university administrative systems could provide substantial benefits within educational management contexts.

# 8 Appendix

## 8.1 References

1. “UCL Human Resources, ‘Postgraduate Teaching Assistant Code of Practice,’ UCL, [Online]. Available at: <https://www.ucl.ac.uk/human-resources/postgraduate-teaching-assistant-code-practice>. [Accessed: Dec 12, 2024].
2. Bhusal, A.( 2021). Predicting Student’s Performance Through Data Mining, University of Northampton, Available at: [arXiv.org](https://arxiv.org/pdf/2112.01247.pdf). [Accessed Dec 12, 2024].
3. T. Doleck, D. J. Lemay, R. B. Basnet, and P. Bazelais, (2020), “Predictive analytics in education: a comparison of deep learning frameworks,” Educ Inf Technol, vol. 25, no. 5, pp. 1951-1963. Available at: <https://doi.org/10.1007/s10639-019-10068-4>. [Accessed Dec 12, 2024].
4. Harris, C. R., Millman, K. J., van der Walt, S. J., et al. (2020). Array programming with NumPy. Nature, 585(7825), 357–362. Available at: <https://doi.org/10.1038/s41586-020-2649-2>. [Accessed Dec 14, 2024].
5. McKinney, W. (2010). Data Structures for Statistical Computing in Python. Proceedings of the 9th Python in Science Conference, 445, 51–56. Available at: [(PDF) Data Structures for Statistical Computing in Python (researchgate.net)](https://www.researchgate.net/publication/265001241_Data_Structures_for_Statistical_Computing_in_Python) .[Accessed Dec 20, 2024].
6. Hipp, D. R. (2000). SQLite: An Embeddable SQL Database Engine. Proceedings of the USENIX Annual Technical Conference, FREENIX Track, 317–330. Available at: [Embedding an SQL Database with SQLite | Linux Journal](https://www.linuxjournal.com/article/6650). [Accessed Dec 20, 2024].
7. Bayer, M. (2006). SQLAlchemy: The Python SQL Toolkit and Object Relational Mapper. Available at: <http://www.sqlalchemy.org/>. [Accessed Dec 21, 2024].
8. McKinney, W. (2011). pandas: a Foundational Python Library for Data Analysis and Statistics. Python for High Performance and Scientific Computing, 14. Available at: [(PDF) pandas: a Foundational Python Library for Data Analysis and Statistics (researchgate.net)](https://www.researchgate.net/publication/265194455_pandas_a_Foundational_Python_Library_for_Data_Analysis_and_Statistics). [Accessed Dec 21, 2024].
9. Pedregosa, F., Varoquaux, G., Gramfort, A., et al. (2011). Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research, 12, 2825–2830. Available at: [[1201.0490] Scikit-learn: Machine Learning in Python (arxiv.org)](https://arxiv.org/abs/1201.0490). [Accessed Dec 23, 2024].
10. Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O’Reilly Media. Available at: [NLTK Book](https://www.nltk.org/book/). [Accessed Dec 23, 2024].
11. van der Walt, S., Colbert, S. C., & Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science & Engineering, 13(2), 22–30. Available at: [[1102.1523] The NumPy array: a structure for efficient numerical computation (arxiv.org)](https://arxiv.org/abs/1102.1523). [Accessed Dec 23, 2024].
12. Python Software Foundation. (2020). pickle — Python object serialization. Available at: [https://docs.python.org/3/library/pickle.html.](1.%09https:/docs.python.org/3/library/pickle.html) [Accessed Dec 23, 2024].
13. SQLite. SQLite Is Serverless. Available at: [SQLite Is Serverless](https://www.sqlite.org/serverless.html) [Accessed Dec 23, 2024].
14. P. D. Grünwald, (2007), "Linear Regression," in The Minimum Description Length Principle, MIT Press. pp. 335-368. doi: 10.7551/mitpress/9780262256247.003.0012. Available at: [Linear Regression | part of The Minimum Description Length Principle | MIT Press books | IEEE Xplore](https://ieeexplore.ieee.org/document/6282057). [Accessed Jan 4, 2024].
15. K. Chaudhuri, C. Monteleoni, and A. D. Sarwate, (2013), “Privacy-Preserving Ridge Regression on Hundreds of Millions of Records,” in *2013 IEEE Symposium on Security and Privacy*, pp. 334-348. doi: 10.1109/SP.2013.29. Available at [Privacy-Preserving Ridge Regression on Hundreds of Millions of Records | IEEE Conference Publication | IEEE Xplore](https://ieeexplore.ieee.org/abstract/document/6547119). [Accessed Jan 4, 2024].
16. S. K. Saha, A. K. Jha, and S. K. Singh (2014), “Pattern and Cluster Mining on Text Data,” Fourth International Conference on Communication Systems and Network Technologies, 2014,.Available at: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6821432> [Accessed Feb 3, 2024].
17. M. Alaradi, L. Alaradi, and S. Al-Maadeed, (2021), "A Deep-Learned Embedding Technique for Categorical Features Encoding," IEEE Access, vol. 9, pp. 114381-114391. doi: 10.1109/ACCESS.2021.3104512. Available at [A Deep-Learned Embedding Technique for Categorical Features Encoding | IEEE Journals & Magazine | IEEE Xplore](https://ieeexplore.ieee.org/document/9512057?denied=). [Accessed Feb 5, 2024].
18. Scikit-learn developers. (n.d.). 3.1. Cross-validation: evaluating estimator performance. Scikit-learn. Available at: <https://scikit-learn.org/stable/modules/cross_validation.html> [Accessed Feb 7, 2024].
19. P. Prakrankamanant and E. Chuangsuwanich, (2022), “Tokenization-based data augmentation for text classification,” in 2022 19th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 1-6. doi: 10.1109/JCSSE54890.2022.98362681. Available at [Tokenization-based data augmentation for text classification | IEEE Conference Publication | IEEE Xplore](https://ieeexplore.ieee.org/document/9836268/citations#citations). [Accessed Feb 21, 2024].
20. K. V. Ghag and K. Shah, “Comparative analysis of effect of stopwords removal on sentiment classification,” in 2015 International Conference on Computer, Communication and Control (IC4), Indore, India, 2015, pp. 1-4. doi: 10.1109/IC4.2015.7375527. Available at [Comparative analysis of effect of stopwords removal on sentiment classification | IEEE Conference Publication | IEEE Xplore](https://ieeexplore.ieee.org/document/7375527/metrics#metrics). [Accessed Feb 21, 2024].
21. A. Dashti et al., (2023), “A Survey of Vectorization Methods in Topological Data Analysis,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 45, no. 12, pp. 14069-14080. doi: 10.1109/TPAMI.2023.3308391. Available at [A Survey of Vectorization Methods in Topological Data Analysis | IEEE Journals & Magazine | IEEE Xplore](https://ieeexplore.ieee.org/document/10235748/figures#figures). [Accessed Feb 21, 2024].
22. Statology, (2021), "How to Interpret Root Mean Square Error (RMSE),". Available at [How to Interpret Root Mean Square Error (RMSE) (statology.org)](https://www.statology.org/how-to-interpret-rmse/). [Accessed Feb 5, 2024].
23. DeepAI, "Standard Deviation,". Available at [Standard Deviation Definition | DeepAI](https://deepai.org/machine-learning-glossary-and-terms/standard-deviation). [Accessed Feb 5, 2024].
24. Anonymous, “Understanding TF-IDF (Term Frequency-Inverse Document Frequency),” GeeksforGeeks, Available at . [Accessed Feb 23, 2024].
25. M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, (2017), "Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects," in 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, pp. 167-177. doi: 10.1109/ICST.2017.26. Available at [Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects | IEEE Conference Publication | IEEE Xplore](https://ieeexplore.ieee.org/abstract/document/7927976). [Accessed Mar 1, 2024].
26. ISO/IEC/IEEE 29119-2:2021, (2021), "Software and systems engineering - Software testing - Part 2: Test processes," Available at [ISO/IEC/IEEE 29119-2:2021(en), Software and systems engineering — Software testing — Part 2: Test processes](https://www.iso.org/obp/ui/en/#iso:std:iso-iec-ieee:29119:-2:ed-2:v1:en). [Accessed Mar 1, 2024].
27. Available at . [Accessed Feb 14, 2024].

## 8.2 Code Listing

The code listing included in this section represents a selection from the larger codebase of the project, which focuses on developing the predictive analytics part of the application.

The core functionalities such as data preprocessing, machine learning model implementation, user interface design and database interactions have been chosen as part of code listing due to their central role in the application's operation, design patterns and relevance to the primary objectives of the project as discussed at the start of this report.

### 8.2.1 Data Preprocessing Functions

|  |
| --- |
| import pandas as pd  import re  from sklearn.preprocessing import OneHotEncoder  import nltk  from sklearn.feature\_extraction.text import TfidfVectorizer  from nltk.corpus import stopwords  from nltk.tokenize import word\_tokenize  from nltk.stem import WordNetLemmatizer  from nltk.corpus import wordnet  ​  *# =======================================================================*  *# HANDLES DATA PROCESSING FOR GRAPH PLOTTING AND LINEAR REGRESSION MODELS*  *# =======================================================================*  ​  *# list modules with 'no data found' in their respective years*  def no\_data\_modules(df, col1, col2):     return df.loc[(df[col1] == 'No data found') | (df[col2] == 'No data found'), 'Module Code'].tolist()  ​  def split\_coursework\_exam\_ratio\_column(df):     df[['Exam Weight', 'Coursework Weight']] = df['Exam:Coursework Ratio'].str.split(':', expand=True).astype(int)     return df.drop('Exam:Coursework Ratio', axis=1)  ​  *# Replace 'No data found' with 0 in the specified columns*  def handle\_missing\_data(df, columns):     for col in columns:         df[col] = df[col].replace('No data found', 0)     df[columns] = df[columns].apply(pd.to\_numeric, errors='coerce')     return df  ​  def handle\_nan\_data(df):     return df.fillna(0)  ​  *# get the sum of the column*  def column\_sum(df, column):     return df[column].sum()  ​  *# get the average of the column*  def column\_average(df, column):     return df[column].mean()  *# calculate the difference between pgtas requested and recruited*  def difference\_calculation(df, selected\_year):     df['Difference'] = df[f'requested\_{selected\_year}'] - df[f'recruited\_{selected\_year}']     return df  ​  *# red is shown for PGTAs recruited > requested, signalling demand higher than expected*  def set\_colour(df):     colours = []     for diff in df['Difference']:         if diff < 0:             colours.append('red')         else:             colours.append('green')     return colours  ​  def one\_hot\_encode\_delivery\_code(df):  *# One-hot encode the 'Delivery Code' column*     encoder = OneHotEncoder(sparse=False)     encoded\_delivery\_code = encoder.fit\_transform(df[['delivery\_code']])  ​  *# Create a DataFrame from the encoded array*     encoded\_delivery\_code\_df = pd.DataFrame(encoded\_delivery\_code, columns=encoder.get\_feature\_names\_out(['delivery\_code']))  ​  *# Drop the original 'Delivery Code' column and concatenate the encoded columns*     df = df.drop('delivery\_code', axis=1)     df = pd.concat([df, encoded\_delivery\_code\_df], axis=1)     return df  ​  ​  *# ==============================================*  *# HANDLES DATA PROCESSING FOR DATAFRAME CLEANING*  *# ==============================================*  ​  *# Obtain the total PGTA hours needed including marking*  def get\_total\_pgta\_hours(df):     df = handle\_nan\_data(df)     df['PGTA hours'] = df['PGTA hours excluding marking'] + df['Marking hours excluding end of year exam (if required)'] + df['Marking hours for end of year exam (if required)']     df = df.drop(['PGTA hours excluding marking', 'Marking hours excluding end of year exam (if required)',                   'Marking hours for end of year exam (if required)', 'Timestamp',                   'Enter text to be used in the advert for your module', 'Select up to 3 categories that best fit the role',                   'When is the module taught/delivered?'                  ], axis=1)     return df  ​  def split\_module\_code\_and\_name(df):  *# splits module code and name into two separate columns*     df.rename(columns={'Select module': 'Module Code'}, inplace=True)     df['Module Name'] = df['Module Code'].apply(lambda x: (' ').join(x.split(' ')[1:]))     df['Module Code'] = df['Module Code'].apply(lambda x: x.split(' ')[0])     return df  ​  def create\_coursework\_exam\_ratio\_column(df):  *# group weightage of exams and courseworks for each module*     exam\_type\_assessment = df[df['Assessment Type Name'].str.contains('Exam')]     coursework\_type\_assessment = df[~df['Assessment Type Name'].str.contains('Exam')]     total\_exam\_weights = exam\_type\_assessment.groupby('Module Code')['Assessment Weight'].sum().reset\_index()     total\_coursework\_weights = coursework\_type\_assessment.groupby('Module Code')['Assessment Weight'].sum().reset\_index()  ​  *# merge the exam and coursework weights above into the dataframe and create the Exam:Coursework Ratio column*     df = df.drop\_duplicates(subset='Module Code')     df = df.merge(total\_exam\_weights, on='Module Code', how='left')     df = df.merge(total\_coursework\_weights, on='Module Code', how='left')     df.rename(columns={'Assessment Weight\_y':'Exam Weight', 'Assessment Weight':'Coursework Weight'}, inplace=True)     df['Exam Weight'].fillna(0, inplace=True)     df['Coursework Weight'].fillna(0, inplace=True)     df['Exam:Coursework Ratio'] = df.apply(lambda row: f"{int(row['Exam Weight'])}:{int(row['Coursework Weight'])}", axis=1)     df = df.drop(['Assessment Weight\_x', 'Assessment Sequence Number', 'Assessment Type Name', 'Assessment Title', 'Assessment Type Code'], axis=1)  ​     return df  ​  *# Define a new DataFrame consisting of the following data: Module Code, Number of Students, PGTAs Recruited, Exam:Coursework Ratio, Delivery Code, PGTA Hours and Duties*  def create\_combined\_variables\_df(df\_moduleAssessmentData, df\_capVsActualStudents, df\_requestedVsRecruited, df\_jobDescriptionData):     combined\_data\_list = []       for module\_code in df\_jobDescriptionData['Module Code'].unique():  ​  *# the module code column in df\_moduleAssessmentData and df\_requestedVsRecruited are different. Only take the pgta data if the module from*  *# df\_requestedVsRecruited exists in df\_moduleAssessmentData*         if module\_code in df\_capVsActualStudents['Module Code'].values and module\_code in df\_requestedVsRecruited['Module Code'].values and module\_code in df\_moduleAssessmentData['Module Code'].values:  *# extract data from their respective dataframes*             module\_name = df\_jobDescriptionData[df\_jobDescriptionData['Module Code'] == module\_code]['Module Name'].iloc[0]             students\_2223 = df\_capVsActualStudents[df\_capVsActualStudents['Module Code'] == module\_code]['2022-23 actual students'].iloc[0]             recruited\_2223 = df\_requestedVsRecruited[df\_requestedVsRecruited['Module Code'] == module\_code]['2022-23 recruited'].iloc[0]             exam\_coursework\_ratio = df\_moduleAssessmentData[df\_moduleAssessmentData['Module Code'] == module\_code]['Exam:Coursework Ratio'].iloc[0]             exam\_weight = df\_moduleAssessmentData[df\_moduleAssessmentData['Module Code'] == module\_code]['Exam Weight'].iloc[0]             coursework\_weight = df\_moduleAssessmentData[df\_moduleAssessmentData['Module Code'] == module\_code]['Coursework Weight'].iloc[0]             delivery\_code = df\_moduleAssessmentData[df\_moduleAssessmentData['Module Code'] == module\_code]['Delivery Code'].iloc[0]             duties = df\_jobDescriptionData[df\_jobDescriptionData['Module Code'] == module\_code]['duties'].iloc[0]  ​             row\_data = {                 'Module Code': module\_code,                 'Module Name': module\_name,                 'Number of Students': students\_2223,                 'PGTAs Recruited': recruited\_2223,                 'Exam:Coursework Ratio': exam\_coursework\_ratio,                 'Exam Weight': exam\_weight,                 'Coursework Weight': coursework\_weight,                 'Delivery Code': delivery\_code,                 'Duties': duties            }  ​             combined\_data\_list.append(row\_data)  ​  *# converts list into dataframe*     df\_combined = pd.DataFrame(combined\_data\_list)  ​  *# replace 'No data found' values with 0 to prevent complexities in plotting*     df\_combined = handle\_nan\_data(df\_combined)  ​     return df\_combined  ​  def create\_df\_average\_pgta\_hours(df, duties):     df\_averagePGTAHours = pd.DataFrame()     df\_averagePGTAHours['duties'] = duties     average\_pgta\_hours = []     for duty in duties:         average\_pgta\_hours.append(column\_average(filter\_base\_duty\_in\_duties(df, duty), 'PGTA hours'))     df\_averagePGTAHours['Average PGTA Hours'] = average\_pgta\_hours     return df\_averagePGTAHours  ​  *# Split the duties into a list of individual duties*  def split\_duties(duty):     if duty == 'No data found' or len(duty) == 0:         return []     result = []     stack = []     temp = ''     for c in duty:         if c == '(':             stack.append(c)         if c == ')':             stack.pop()         if c == ',' and stack == []:             if temp[0] == ' ' and temp[-1] == ' ':                 result.append(temp[1:-1])             elif temp[0] == ' ':                 result.append(temp[1:])             elif temp[-1] == ' ':                 result.append(temp[:-1])             else:                 result.append(temp)             temp = ''         else:             temp += c     if temp[0] == ' ' and temp[-1] == ' ':         result.append(temp[1:-1])     elif temp[0] == ' ':         result.append(temp[1:])     elif temp[-1] == ' ':         result.append(temp[:-1])     else:         result.append(temp)     return result  ​  *# Obtain the set of base duties from the job description*  def get\_set\_of\_duties(job\_desc):     base\_duties = []     for duties\_combination in job\_desc:         duty = split\_duties(duties\_combination)         for d in duty:             base\_duties.append(d)     return set(base\_duties)  ​  def create\_feature\_vector(df, unique\_duties):     for duty in unique\_duties:         df[duty] = 0     for index, row in df.iterrows():         for duty in unique\_duties:             if duty in row['duties']:                 df.at[index, duty] = 1     return df  ​  def filter\_base\_duty\_in\_duties(df, duty):     return df[df['duties'].str.contains(re.escape(duty), case=False, na=False)]  ​  ​  *# ==========================*  *# HANDLES TEXT PREPROCESSING*  *# ==========================*  ​  *# Ensure necessary NLTK resources are downloaded*  def download\_nltk\_resources():     nltk.download('punkt')     nltk.download('stopwords')     nltk.download('wordnet')     nltk.download('averaged\_perceptron\_tagger')  ​  *# Tokenize the text*  def tokenize\_text(text):     return word\_tokenize(text)  ​  *# Remove stopwords from a list of tokens*  def remove\_stopwords(tokens):     stop\_words = set(stopwords.words('english'))     return [word for word in tokens if word not in stop\_words]  ​  *# Mapping from POS tag to wordnet tag*  def get\_wordnet\_pos(word):     """Map POS tag to the format accepted by WordNetLemmatizer"""     pos\_tag = nltk.pos\_tag([word])[0][1]  *# Get the POS tag for the word*  ​  *# Define a mapping from the POS tag to the format accepted by WordNetLemmatizer*     tag\_dict = {         'NN': wordnet.NOUN, 'NNS': wordnet.NOUN, 'NNP': wordnet.NOUN, 'NNPS': wordnet.NOUN,         'VB': wordnet.VERB, 'VBD': wordnet.VERB, 'VBG': wordnet.VERB, 'VBN': wordnet.VERB, 'VBP': wordnet.VERB, 'VBZ': wordnet.VERB,         'JJ': wordnet.ADJ, 'JJR': wordnet.ADJ, 'JJS': wordnet.ADJ,         'RB': wordnet.ADV, 'RBR': wordnet.ADV, 'RBS': wordnet.ADV    }     return tag\_dict.get(pos\_tag, wordnet.NOUN)  *# Default to NOUN if not found*  ​  *# Lemmatize a list of tokens*  def lemmatize\_tokens(tokens):     lemmatizer = WordNetLemmatizer()     return [lemmatizer.lemmatize(word, get\_wordnet\_pos(word)) for word in tokens]  ​  *# Preprocess a single description*  def preprocess\_text(text):     tokens = tokenize\_text(text)     tokens = remove\_stopwords(tokens)     lemmatized\_tokens = lemmatize\_tokens(tokens)     return ' '.join(lemmatized\_tokens)  ​  *# Preprocess all descriptions in a list and return a transformed list*  def preprocess\_text\_list(text\_list):     return text\_list.apply(preprocess\_text)  ​ |

### 8.2.2 Graph Interfaces

#### 8.2.2.1 PGTAs Requested vs PGTAs Recruited Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.graph\_objects as go  from dash import html, dcc  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataProcessing import difference\_calculation, set\_colour  from data\_processing.statsLayout import stats\_layout  from database.models import RequestedVsRecruited  ​  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query = session.query(RequestedVsRecruited)  df\_requestedVsRecruited = pd.read\_sql(query.statement, engine)  ​  def requestedVsRecruitedGraphLayout():     options = [{'label': year, 'value': year} for year in ['23\_24', '22\_23', '21\_22']]       return html.Div([         dcc.Dropdown(             options=options,             value='23\_24',             id='requestedVsRecruitedGraphDropdown'        ),         dcc.Graph(figure={}, id='requestedVsRecruitedGraph'),         stats\_layout,    ])  ​  ​  def requestedVsRecruitedGraph(selected\_year):     layout = go.Layout(         title=f'Comparison of Requested vs Recruited PGTAs for {selected\_year}',         barmode='group',         xaxis\_title='Count',         yaxis\_title='Module Code',         hovermode='closest',         height=5000,         width=1700    )  ​     df = difference\_calculation(df\_requestedVsRecruited, selected\_year)     colours = set\_colour(df)  ​  *# plot the bar for pgtas recruited*     trace\_recruited = go.Bar(           x=df[f'recruited\_{selected\_year}'],         y=df['module\_code'],         name='Recruited',         text=['Diff: ' + str(diff) for diff in df['Difference']],         marker\_color=colours,         orientation='h'    )  *# plot the bar for pgtas requested*     trace\_requested = go.Bar(         x=df[f'requested\_{selected\_year}'],         y=df['module\_code'],         name='Requested',         text=['Diff: ' + str(diff) for diff in df['Difference']],         marker\_color=colours,         orientation='h'    )     return go.Figure(data=[trace\_recruited, trace\_requested], layout=layout)  ​  *(app.py)*  *# Callback for requestedVsRecruitedGraph*  *@app*.callback(     Output(component\_id='requestedVsRecruitedGraph', component\_property='figure'),     Input(component\_id='requestedVsRecruitedGraphDropdown' , component\_property='value')  )  ​  def update\_requestedVsRecruitedGraph(selected\_year):     return requestedVsRecruitedGraph(selected\_year) |

#### 8.2.2.2 Module History Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.graph\_objects as go  from dash import html, dcc  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataProcessing import difference\_calculation, set\_colour  from data\_processing.statsLayout import stats\_layout  from database.models import RequestedVsRecruited  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query = session.query(RequestedVsRecruited)  df\_requestedVsRecruited = pd.read\_sql(query.statement, engine)  ​  def moduleHistoryGraphLayout():  *# Get a list of unique modules from the dataframe*     modules = df\_requestedVsRecruited['module\_code'].unique().tolist()     options = [{'label': module, 'value': module} for module in modules]       return html.Div([         dcc.Dropdown(             options=options,             value=modules[0],  *# Default value set to the first module in the list*             id='moduleHistoryGraphDropdown'        ),         dcc.Graph(figure={}, id='moduleHistoryGraph'),         stats\_layout,  *# html.Div(id='moduleStudentsDisplay')*    ])  ​  def moduleHistoryGraph(selected\_module):  *# Filter the dataframe for the selected module*     module\_data = df\_requestedVsRecruited[df\_requestedVsRecruited['module\_code'] == selected\_module]  ​     traces = []     for year in ['21\_22', '22\_23', '23\_24']:  *# Create traces for each year*         df = difference\_calculation(df\_requestedVsRecruited, year)         colours = set\_colour(df)  ​         trace\_recruited = go.Bar(             x=[year],             y=[module\_data[f'recruited\_{year}'].values[0]],             name=f'Recruited {year}',             text=['Diff: ' + str(diff) for diff in df['Difference']],             marker\_color=colours,        )         trace\_requested = go.Bar(             x=[year],             y=[module\_data[f'requested\_{year}'].values[0]],             name=f'Requested {year}',             text=['Diff: ' + str(diff) for diff in df['Difference']],             marker\_color=colours,        )         traces.extend([trace\_recruited, trace\_requested])  ​     layout = go.Layout(         title=f'Comparison of Requested vs Recruited PGTAs for {selected\_module}',         barmode='group',         xaxis\_title='Year',         yaxis\_title='Count',         hovermode='closest',         height=700,         width=1200    )     return go.Figure(data=traces, layout=layout)  ​  session.close()  ​  *(app.py)*  ​  *# Callback for moduleHistoryGraph*  *@app*.callback(     Output(component\_id='moduleHistoryGraph', component\_property='figure'),     Input(component\_id='moduleHistoryGraphDropdown' , component\_property='value')  )  ​  def update\_moduleHistoryGraph(selected\_module):     return moduleHistoryGraph(selected\_module) |

#### 8.2.2.3 Students Enrolled vs PGTAs Recruited Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.express as px  from dash import html, dcc  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataframeCleaning import df\_combined\_variables  from database.models import CombinedVariables  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query = session.query(CombinedVariables)  df\_combined\_variables = pd.read\_sql(query.statement, engine)  ​  def studentsVsRecruitedGraphLayout():     return html.Div([         dcc.Graph(figure=studentsVsRecruitedGraph(), id='studentsVsRecruitedGraph')    ])  ​  def studentsVsRecruitedGraph():     fig = px.scatter(         df\_combined\_variables,         x='number\_of\_students',         y='pgtas\_recruited',         hover\_name='module\_code',    )     fig.update\_traces(textposition='top center')  ​     return fig |

#### 8.2.2.4 Exam-Coursework Ratio vs PGTAs Recruited Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.express as px  from dash import html, dcc  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataframeCleaning import df\_combined\_variables  from database.models import CombinedVariables  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query = session.query(CombinedVariables)  df\_combined\_variables = pd.read\_sql(query.statement, engine)  ​  def examWeightsVsRecruitedGraphLayout():     return html.Div([         dcc.Graph(figure=examWeightsVsRecruitedGraph(), id='examWeightsVsRecruitedGraph')    ])  ​  def examWeightsVsRecruitedGraph():     fig = px.scatter(         df\_combined\_variables,         x='exam\_coursework\_ratio',         y='pgtas\_recruited',         hover\_name='module\_code',    )     fig.update\_traces(textposition='top center')  ​     return fig |

#### 8.2.2.5 Module Delivery Code vs PGTAs Recruited Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.express as px  from dash import html, dcc  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataframeCleaning import df\_combined\_variables  from database.models import CombinedVariables  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query = session.query(CombinedVariables)  df\_combined\_variables = pd.read\_sql(query.statement, engine)  ​  def deliveryCodeVsRecruitedGraphLayout():     return html.Div([         dcc.Graph(figure=deliveryCodeVsRecruitedGraph(), id='deliveryCodeVsRecruitedGraph')    ])  ​  def deliveryCodeVsRecruitedGraph():     fig = px.scatter(         df\_combined\_variables,         x='delivery\_code',         y='pgtas\_recruited',         hover\_name='module\_code',    )     fig.update\_traces(textposition='top center')  ​     return fig |

#### 8.2.2.6 Duties vs PGTA Hours Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.express as px  from dash import html, dcc  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataProcessing import filter\_base\_duty\_in\_duties  from data\_processing.dataframeCleaning import duties  from database.models import JobDescription, AveragePGTAHours  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query\_JobDescription = session.query(JobDescription)  query\_AveragePGTAHours = session.query(AveragePGTAHours)  df\_jobDescriptionData = pd.read\_sql(query\_JobDescription.statement, engine)  df\_averagePGTAHours = pd.read\_sql(query\_AveragePGTAHours.statement, engine)  ​  *# plot the graph of duties vs pgta hours where duty in duties is present in the dataframe*  def dutiesVsPGTAHoursGraphLayout():     return html.Div([         dcc.Dropdown(             id='dutiesVsPGTAHoursGraphDropdown',             options=[{'label': duty, 'value': duty} for duty in duties],             value=duties[0]        ),         dcc.Graph(figure=dutiesVsPGTAHoursGraph(duties[0]), id='dutiesVsPGTAHoursGraph'),    ])  ​  def dutiesVsPGTAHoursGraph(duty):     df = filter\_base\_duty\_in\_duties(df\_jobDescriptionData, duty)     fig = px.bar(         df,         x='module\_code',         y='total\_hours',         title=f'PGTA Hours for Duty: {duty}'    )     max\_hours = df\_jobDescriptionData['total\_hours'].max()     min\_hours = 0     fig.update\_layout(         yaxis=dict(             range=[min\_hours, max\_hours + 10]  *# Adding a buffer to the maximum for better visualization*        ),         height=1100    )     return fig  ​​  session.close()  *(app.py)*  ​  *# Callback for Duties Vs PGTA Hours Graph*  *@app*.callback(     Output(component\_id='dutiesVsPGTAHoursGraph', component\_property='figure'),     Input(component\_id='dutiesVsPGTAHoursGraphDropdown' , component\_property='value')  )  ​  def update\_dutiesVsPGTAHoursGraph(selected\_duty):     return dutiesVsPGTAHoursGraph(selected\_duty) |

#### 8.2.2.6 Duties vs Average PGTA Hours Graph

|  |
| --- |
| import sys  import os  import pandas as pd  import plotly.express as px  from dash import html, dcc  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataProcessing import filter\_base\_duty\_in\_duties  from data\_processing.dataframeCleaning import duties  from database.models import JobDescription, AveragePGTAHours  from sqlalchemy import create\_engine  from sqlalchemy.orm import sessionmaker  ​  *# import data from database*  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  query\_JobDescription = session.query(JobDescription)  query\_AveragePGTAHours = session.query(AveragePGTAHours)  df\_jobDescriptionData = pd.read\_sql(query\_JobDescription.statement, engine)  df\_averagePGTAHours = pd.read\_sql(query\_AveragePGTAHours.statement, engine)  ​  *# graph showing the average pgta hours for each duty*  def dutiesVsPGTAHoursAverageGraphLayout():     return html.Div([         dcc.Graph(figure=dutiesVsPGTAHoursAverageGraph(), id='dutiesVsPGTAHoursAverageGraph'),    ])  ​  def dutiesVsPGTAHoursAverageGraph():     fig = px.bar(         df\_averagePGTAHours,         x='duties',         y='average\_hours',         title='Average PGTA Hours for Each Duty',         height=1100    )     return fig  ​  session.close() |

### 8.2.3 Machine Learning Model Training

#### 8.2.3.1 Linear Regression Model Training​

|  |
| --- |
| import sys  import os  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../..')))  from data\_processing.dataProcessing import one\_hot\_encode\_delivery\_code  from sklearn.model\_selection import cross\_val\_score, KFold  from sklearn.linear\_model import LinearRegression  import numpy as np  from ml\_models.modelSaving import save\_model  ​  def train\_linear\_regression\_model(df):  *# Prepare the features and target variables*     df = one\_hot\_encode\_delivery\_code(df)  ​     X = df[['number\_of\_students', 'exam\_weight', 'coursework\_weight', 'delivery\_code\_A4U', 'delivery\_code\_A5U', 'delivery\_code\_A6U', 'delivery\_code\_A7U', 'delivery\_code\_A7P']]     y = df['pgtas\_recruited']  ​  *# Train the model*     model = LinearRegression()     model.fit(X, y)  ​  *# Initialize the K-Fold cross-validator*     kf = KFold(n\_splits=3, shuffle=True, random\_state=42)  ​  *# Perform cross-validation and compute the scores*     scores = cross\_val\_score(model, X, y, cv=kf, scoring='neg\_mean\_squared\_error')  ​  *# Convert the scores to root mean squared error (RMSE)*     rmse\_scores = np.sqrt(-scores)  ​     print("Linear Regression Mean RMSE:", rmse\_scores.mean())     print("Linear Regression Standard deviation:", rmse\_scores.std())  ​  *# Save the trained model*     save\_model(model, 'linear\_model.pkl')     print("Linear Regression Model trained and saved as linear\_model.pkl")     print('-----------------------------------------------------------------------------')  ​​ |

#### 8.2.3.2 Ridge Regression Model Training

|  |
| --- |
| import sys  import os  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../..')))  from data\_processing.dataProcessing import one\_hot\_encode\_delivery\_code  from sklearn.model\_selection import cross\_val\_score, KFold  from sklearn.linear\_model import Ridge  import numpy as np  from ml\_models.modelSaving import save\_model  ​  def train\_ridge\_regression\_model(df):  *# Prepare the features and target variables*     df = one\_hot\_encode\_delivery\_code(df)  ​     X = df[['number\_of\_students', 'exam\_weight', 'coursework\_weight', 'delivery\_code\_A4U', 'delivery\_code\_A5U', 'delivery\_code\_A6U', 'delivery\_code\_A7U', 'delivery\_code\_A7P']]     y = df['pgtas\_recruited']  ​  *# Train the model*     model = Ridge(alpha=1.0, random\_state=42)     model.fit(X, y)  ​  ​  *# Initialize the K-Fold cross-validator*     kf = KFold(n\_splits=5, shuffle=True, random\_state=42)  ​  *# Perform cross-validation and compute the scores*     scores = cross\_val\_score(model, X, y, cv=kf, scoring='neg\_mean\_squared\_error')  ​  *# Convert the scores to root mean squared error (RMSE)*     rmse\_scores = np.sqrt(-scores)  ​     print("Ridge Regression Mean RMSE:", rmse\_scores.mean())     print("Ridge Regression Standard deviation:", rmse\_scores.std())  ​  *# Save the trained model*     save\_model(model, 'ridge\_model.pkl')     print("Ridge Regression Model trained and saved as ridge\_model.pkl")     print('---------------------------------------------------------------------------- ') |

#### 8.2.3.3 Generalized Additive Model Training

|  |
| --- |
| import numpy as np  import sys  import os  from sklearn.model\_selection import train\_test\_split  from sklearn.metrics import mean\_squared\_error  from pygam import LinearGAM, s, f  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataProcessing import one\_hot\_encode\_delivery\_code  from ml\_models.modelSaving import save\_model  ​  def train\_generalised\_additive\_model(df):  *# Prepare the features and target variables*     df = one\_hot\_encode\_delivery\_code(df)  ​     X = df[['number\_of\_students', 'exam\_weight', 'coursework\_weight', 'delivery\_code\_A4U', 'delivery\_code\_A5U', 'delivery\_code\_A6U', 'delivery\_code\_A7U', 'delivery\_code\_A7P']]     y = df['pgtas\_recruited']  ​  *# Setting up the GAM model with spline terms for continuous features and factor terms for categorical*     terms = s(0) + s(1) + s(2) *# Spline terms for the continuous features*     for i in range(3, X.shape[1]):  *# f() for encoded categorical features starting from index 3 onwards*         terms += f(i)  ​     X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)  ​     gam = LinearGAM(terms)     gam.fit(X\_train, y\_train)  ​     y\_pred = gam.predict(X\_test)  ​     rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))  ​     print("Generalised Additive Model RMSE:", rmse)  ​  *# Save the trained model*     save\_model(gam, 'gam\_model.pkl')     print("Generalised Additive Model trained and saved as gam\_model.pkl")     print('-----------------------------------------------------------------------------')  ​ |

#### 8.2.3.4 Feature Engineering Model Training

|  |
| --- |
| import os  import sys  from sklearn.model\_selection import KFold, cross\_val\_score  from sklearn.linear\_model import Ridge  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../..')))  import numpy as np  from ml\_models.modelSaving import save\_model  from data\_processing.dataProcessing import create\_feature\_vector  ​  def train\_feature\_engineering\_model(df, duties):     df = create\_feature\_vector(df, duties)     X = df[duties]     y = df['pgtas\_recruited']  ​     model = Ridge(alpha=1.0, random\_state=42)     model = model.fit(X, y)  ​  *# Evaluate the model*     kf = KFold(n\_splits=3, shuffle=True, random\_state=42)     scores = cross\_val\_score(model, X, y, cv=kf, scoring='neg\_mean\_squared\_error')     rmse\_scores = np.sqrt(-scores)  ​     print("Feature Engineering Mean RMSE:", rmse\_scores.mean())     print("Feature Engineering Standard deviation:", rmse\_scores.std())  ​     save\_model(model, 'feature\_engineering\_model.pkl')     print("Feature Engineering Model trained and saved as feature\_engineering\_model.pkl")     print('-----------------------------------------------------------------------------')  ​ |

#### 8.2.3.5 TF-IDF Vectorizer Model Training

|  |
| --- |
| import numpy as np  import os  import sys  from sklearn.model\_selection import KFold, cross\_val\_score  from sklearn.feature\_extraction.text import TfidfVectorizer  from sklearn.linear\_model import Ridge  from sklearn.pipeline import Pipeline  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../..')))  from data\_processing.dataProcessing import preprocess\_text\_list  from ml\_models.modelSaving import save\_model  ​  def train\_tf\_idf\_model(df):  ​     X = df['duties']     y = df['pgtas\_recruited']  ​  *# Preprocess the 'Duties' text data*     X\_preprocessed = preprocess\_text\_list(X)  ​  *# Create a pipeline with TF-IDF Vectorization and Linear Regression*     pipeline = Pipeline([        ('tfidf', TfidfVectorizer()),        ('regressor', Ridge())    ])  ​  *# Fit the pipeline to the data*     pipeline.fit(X\_preprocessed, y)  ​     kf = KFold(n\_splits=5, shuffle=True, random\_state=42)     scores = cross\_val\_score(pipeline, X\_preprocessed, y, cv=kf, scoring='neg\_mean\_squared\_error')     rmse\_scores = np.sqrt(-scores)  ​     print("TF-IDF Vectorization Mean RMSE:", rmse\_scores.mean())     print("TF-IDF Vectorization Standard deviation:", rmse\_scores.std())  ​     save\_model(pipeline, 'TF-IDF\_model.pkl')     print("TF-IDF Vectorization Model trained and saved as TF-IDF\_model.pkl")     print('-----------------------------------------------------------------------------') |

### 8.2.4 Machine Learning Prompts

#### 8.2.4.1 Linear Regression Model Prompt​

|  |
| --- |
| import pandas as pd  from dash import html  import os  import sys  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../')))  from ml\_models.modelLoading import load\_model  from data\_processing.dataProcessing import one\_hot\_encode\_delivery\_code  import dash\_bootstrap\_components as dbc  ​  ​  def linearRegressionPredictorLayout():     return html.Div([         html.H1("PGTAs Recruitment Predictor with Linear Regression"),         html.Div(        [             html.Br(),             dbc.Input(                 id='number-of-students-linear',                 type='number',                 placeholder='Number of Students',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='exam-weight-linear',                 type='number',                 placeholder='Exam Weight',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='coursework-weight-linear',                 type='number',                 placeholder='Coursework Weight',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='delivery-code-linear',                 type='text',                 placeholder='Delivery Code', style={'width': '15%'}            ),             html.Br(),        ]),         dbc.Button(             'Predict',             color="secondary",             id='linear-regression-prediction-button',             n\_clicks=0        ),         html.Br(),         html.Br(),         html.Hr(),         html.Div(id='linear-regression-prediction-output')    ])  ​  def linearRegressionPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code):     if n\_clicks > 0:  *# Prepare the input data in the format expected by the model*         input\_data = pd.DataFrame([{             'number\_of\_students': number\_of\_students,             'exam\_weight': exam\_weight,             'coursework\_weight': coursework\_weight,             'delivery\_code': delivery\_code        }])           input\_data = one\_hot\_encode\_delivery\_code(input\_data)  ​         model = load\_model('linear\_model.pkl')  ​  *# Ensure all columns from training data are present in input data (fill missing columns with 0s)*         missing\_cols = set(model.feature\_names\_in\_) - set(input\_data.columns)         for col in missing\_cols:             input\_data[col] = 0  *# Reorder columns to match the training data*         input\_data = input\_data[model.feature\_names\_in\_]  ​  *# Make prediction*         prediction = model.predict(input\_data)[0]         return f"Predicted PGTA Hours: {prediction}"     return ""  ​  ​  *(app.py)*  ​  *# Callback for Linear Regression Predictor Prompt*  *@app*.callback(     Output('linear-regression-prediction-output', 'children'),    [Input('linear-regression-prediction-button', 'n\_clicks')],    [State('number-of-students-linear', 'value'),     State('exam-weight-linear', 'value'),     State('coursework-weight-linear', 'value'),     State('delivery-code-linear', 'value')]  )  ​  def update\_linearRegressionPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code):     return linearRegressionPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code)  ​ |

#### 8.2.4.2 Ridge Regression Model Prompt

|  |
| --- |
| import pandas as pd  from dash import html  import os  import sys  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../')))  from ml\_models.modelLoading import load\_model  from data\_processing.dataProcessing import one\_hot\_encode\_delivery\_code  import dash\_bootstrap\_components as dbc  ​  ​  def ridgeRegressionPredictorLayout():     return html.Div([         html.H1("PGTAs Recruitment Predictor with Ridge Regression"),         html.Div(        [             html.Br(),             dbc.Input(                 id='number-of-students-ridge',                 type='number',                 placeholder='Number of Students',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='exam-weight-ridge',                 type='number',                 placeholder='Exam Weight',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='coursework-weight-ridge',                 type='number',                 placeholder='Coursework Weight',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='delivery-code-ridge',                 type='text',                 placeholder='Delivery Code',                 style={'width': '15%'}            ),             html.Br(),        ]),         dbc.Button(             'Predict',             color="secondary",             id='ridge-regression-prediction-button',             n\_clicks=0        ),         html.Br(),         html.Br(),         html.Hr(),         html.Div(id='ridge-regression-prediction-output')    ])  ​  def ridgeRegressionPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code):     if n\_clicks > 0:  *# Prepare the input data in the format expected by the model*         input\_data = pd.DataFrame([{             'number\_of\_students': number\_of\_students,             'exam\_weight': exam\_weight,             'coursework\_weight': coursework\_weight,             'delivery\_code': delivery\_code        }])  ​         input\_data = one\_hot\_encode\_delivery\_code(input\_data)  ​         model = load\_model('ridge\_model.pkl')  ​  *# Ensure all columns from training data are present in input data (fill missing columns with 0s)*         missing\_cols = set(model.feature\_names\_in\_) - set(input\_data.columns)         for col in missing\_cols:             input\_data[col] = 0  *# Reorder columns to match the training data*         input\_data = input\_data[model.feature\_names\_in\_]  ​  *# Make prediction*         prediction = model.predict(input\_data)[0]         return f"Predicted PGTA Hours: {prediction}"     return ""  ​  ​  *(app.py)*  ​  *# Callback for Ridge Regression Predictor Prompt*  *@app*.callback(     Output('ridge-regression-prediction-output', 'children'),    [Input('ridge-regression-prediction-button', 'n\_clicks')],    [State('number-of-students-ridge', 'value'),     State('exam-weight-ridge', 'value'),     State('coursework-weight-ridge', 'value'),     State('delivery-code-ridge', 'value')]  )  ​  def update\_ridgeRegressionPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code):     return ridgeRegressionPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code) |

#### 8.2.4.3 Generalized Additive Model Prompt

|  |
| --- |
| import pandas as pd  from dash import html  import os  import sys  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../')))  from ml\_models.modelLoading import load\_model  from data\_processing.dataProcessing import one\_hot\_encode\_delivery\_code  import dash\_bootstrap\_components as dbc  ​  ​  def gamPredictorLayout():     return html.Div([         html.H1("PGTAs Recruitment Predictor with Generalized Additive Model"),         html.Div(        [             html.Br(),             dbc.Input(                 id='number-of-students-gam',                 type='number',                 placeholder='Number of Students',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='exam-weight-gam',                 type='number',                 placeholder='Exam Weight',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='coursework-weight-gam',                 type='number',                 placeholder='Coursework Weight',                 style={'width': '15%'}            ),             html.Br(),             dbc.Input(                 id='delivery-code-gam',                 type='text',                 placeholder='Delivery Code',                 style={'width': '15%'}            ),             html.Br(),        ]),         dbc.Button(             'Predict',             color="secondary",             id='gam-prediction-button',             n\_clicks=0        ),         html.Br(),         html.Br(),         html.Hr(),         html.Div(id='gam-prediction-output')    ])  ​  def gamPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code):     if n\_clicks > 0:         input\_data = pd.DataFrame([{             'number\_of\_students': number\_of\_students,             'exam\_weight': exam\_weight,             'coursework\_weight': coursework\_weight,             'delivery\_code': delivery\_code        }])           input\_data = one\_hot\_encode\_delivery\_code(input\_data)  ​         feature\_names = [             'number\_of\_students',             'exam\_weight',             'coursework\_weight',             'delivery\_code\_A4U',             'delivery\_code\_A5U',             'delivery\_code\_A6U',             'delivery\_code\_A7U',             'delivery\_code\_A7P'        ]  *# Ensure all expected features are present in the input data, filling missing columns with 0s*         for feature in feature\_names:             if feature not in input\_data.columns:                 input\_data[feature] = 0    *# Reorder input\_data columns to match the training feature order*         input\_data = input\_data[feature\_names]  ​         gam = load\_model('gam\_model.pkl')         prediction = gam.predict(input\_data)         return f"Predicted PGTA Hours with GAM: {prediction[0]}"     return ""  ​  ​  *(app.py)*  ​  *# Callback for Generalized Additive Model Predictor Prompt*  *@app*.callback(     Output('gam-prediction-output', 'children'),    [Input('gam-prediction-button', 'n\_clicks')],    [State('number-of-students-gam', 'value'),     State('exam-weight-gam', 'value'),     State('coursework-weight-gam', 'value'),     State('delivery-code-gam', 'value')]  )  ​  def update\_gamPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code):     return gamPredictor(n\_clicks, number\_of\_students, exam\_weight, coursework\_weight, delivery\_code) |

#### 8.2.4.4 Feature Engineering Model Prompt

|  |
| --- |
| import pandas as pd  import os  import sys  from dash import html  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../')))  from ml\_models.modelLoading import load\_model  from data\_processing.dataframeCleaning import duties  import dash\_bootstrap\_components as dbc  ​  ​  def featureEngineeringPredictorLayout():     return html.Div([         html.H1("PGTAs Recruitment Predictor with Feature Engineering"),         html.Div([             dbc.Checklist(             id="feature-base-duties-checklist",             value=[],             options=[{'label': duty, 'value': duty} for duty in duties],            ),        ]),         html.Br(),         dbc.Button(             'Predict',             color="secondary",             id='feature-prediction-button',             n\_clicks=0        ),         html.Hr(),         html.Br(),         html.Div(id='feature-prediction-output')    ])  ​  def featureEngineeringPredictor(n\_clicks, selected\_duties):     if n\_clicks > 0:  *# prepare the input data in the format expected by the model*         data = {duty: 0 for duty in duties}         for duty in selected\_duties:             data[duty] = 1           input\_data = pd.DataFrame([data])           model = load\_model('feature\_engineering\_model.pkl')    *# fill missing columns with 0s*         missing\_cols = set(model.feature\_names\_in\_) - set(input\_data.columns)         for col in missing\_cols:             input\_data[col] = 0  ​  *# reorder columns to match the training data*         input\_data = input\_data[model.feature\_names\_in\_]    *# make prediction*         prediction = model.predict(input\_data)[0]         return f"Predicted PGTA Hours: {prediction}"     return ""  ​  ​  *(app.py)*  ​  *# Callback for Feature Engineering Predictor Prompt*  *@app*.callback(     Output('feature-prediction-output', 'children'),    [Input('feature-prediction-button', 'n\_clicks')],    [State('feature-base-duties-checklist', 'value')]  )  ​  def update\_featureEngineeringPredictor(n\_clicks, selected\_duties):     return featureEngineeringPredictor(n\_clicks, selected\_duties) |

#### 8.2.4.5 TF-IDF Vectorizer Model Prompt

|  |
| --- |
| from dash import html  import os  import sys  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '../')))  from ml\_models.modelLoading import load\_model  from data\_processing.dataProcessing import preprocess\_text  from data\_processing.dataframeCleaning import duties  import dash\_bootstrap\_components as dbc  ​  ​  def vectoriserPredictorLayout():     return html.Div([         html.H1("PGTAs Recruitment Predictor with TF-IDF Vectoriser"),         html.Div([             dbc.Checklist(                 id="vectoriser-base-duties-checklist",                 value=[],                 options=[{'label': duty, 'value': duty} for duty in duties],            ),        ]),         html.Br(),         dbc.Button(             'Predict',             color="secondary",             id='vectoriser-prediction-button',             n\_clicks=0        ),         html.Hr(),         html.Br(),         html.Div(id='vectoriser-prediction-output')    ])  ​  def vectoriserPredictor(n\_clicks, selected\_duties):     if n\_clicks > 0:  *# take the selected duties, join them together by a comma and feed into the model*         input\_data = ', '.join(selected\_duties)           preprocessed\_input\_data = preprocess\_text(input\_data)           model = load\_model('TF-IDF\_model.pkl')  ​           prediction = model.predict([preprocessed\_input\_data])[0]         return f"Predicted PGTA Hours: {prediction}"     return ""  ​  ​  *(app.py)*  ​  *# Callback for Vectoriser Predictor Prompt*  *@app*.callback(     Output('vectoriser-prediction-output', 'children'),    [Input('vectoriser-prediction-button', 'n\_clicks')],    [State('vectoriser-base-duties-checklist', 'value')]  )  ​  def update\_vectoriserPredictor(n\_clicks, selected\_duties):     return vectoriserPredictor(n\_clicks, selected\_duties) |

### 8.2.5 Database Operations

#### 8.2.5.1 Database Model Definition

|  |
| --- |
| from sqlalchemy.ext.declarative import declarative\_base  from sqlalchemy import Column, Integer, String, Text, Float  ​  Base = declarative\_base()  ​  class JobDescription(Base):     \_\_tablename\_\_ = 'job\_descriptions'     id = Column(Integer, primary\_key=True)     module\_code = Column(String)     number\_of\_TA = Column(Integer)     duties = Column(String)     total\_hours = Column(Integer)     module\_title = Column(String)  ​  class RequestedVsRecruited(Base):     \_\_tablename\_\_ = 'requested\_vs\_recruited'     id = Column(Integer, primary\_key=True)     module\_code = Column(String)     module\_title = Column(String)     variant = Column(String)     module\_code\_and\_title = Column(String)     is\_module\_new = Column(String)     requested\_23\_24 = Column(Integer)     recruited\_23\_24 = Column(Integer)     requested\_22\_23 = Column(Integer)     recruited\_22\_23 = Column(Integer)     requested\_21\_22 = Column(Integer)     recruited\_21\_22 = Column(Integer)     notes = Column(Text)  ​  class CapVsActualStudents(Base):     \_\_tablename\_\_ = 'cap\_vs\_actual\_students'     id = Column(Integer, primary\_key=True)     module\_code = Column(String)     module\_title = Column(String)     cap\_23\_24 = Column(Integer)     actual\_22\_23 = Column(Integer)     notes = Column(Text)  ​  class ModuleAssessment(Base):     \_\_tablename\_\_ = 'module\_assessment'     id = Column(Integer, primary\_key=True)     module\_code = Column(String)     delivery\_code = Column(String)     module\_delivery\_period\_code = Column(String)     exam\_weight = Column(Integer)     coursework\_weight = Column(Integer)     exam\_coursework\_ratio = Column(String)  ​  class CombinedVariables(Base):     \_\_tablename\_\_ = 'combined\_variables'     id = Column(Integer, primary\_key=True)     module\_code = Column(String)     module\_name = Column(String)     number\_of\_students = Column(Integer)     pgtas\_recruited = Column(Integer)     exam\_coursework\_ratio = Column(String)     exam\_weight = Column(Integer)     coursework\_weight = Column(Integer)     delivery\_code = Column(String)     duties = Column(String)  ​  class AveragePGTAHours(Base):     \_\_tablename\_\_ = 'average\_pgta\_hours'     id = Column(Integer, primary\_key=True)     duties = Column(String)     average\_hours = Column(Float) |

#### 8.2.5.2 Database Operations

|  |
| --- |
| from dash import html, dcc  from sqlalchemy import create\_engine, MetaData, inspect  from sqlalchemy.orm import sessionmaker  import os  import sys  sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(\_\_file\_\_), '..')))  from data\_processing.dataframeCleaning import duties  import dash\_bootstrap\_components as dbc  import pandas as pd  from database.models import CombinedVariables  from data\_processing.dataframeCleaning import duties  ​  DATABASE\_URI = 'sqlite:///app\_database.db'  engine = create\_engine(DATABASE\_URI)  Session = sessionmaker(bind=engine)  session = Session()  meta = MetaData()  meta.reflect(bind=engine)  table\_names = list(meta.tables.keys())  ​  def displayTableLayout():     return dbc.Container([         dbc.Tabs(             id="database-tabs",             children=[dbc.Tab(                 label=table\_name,                 tab\_id=table\_name            ) for table\_name in table\_names],             active\_tab=table\_names[0] if table\_names else None,        ),         html.Div(id="table-content")    ], fluid=True)  ​  def displayTable(active\_tab):     if active\_tab is None:         return "No table selected"    *# Query the database for the selected table*     df = pd.read\_sql\_table(active\_tab, con=engine)    *# Create the table header*     table\_header = [         html.Thead(html.Tr([html.Th(col) for col in df.columns]))    ]    *# Create the table body*     table\_body = [         html.Tbody([             html.Tr([                 html.Td(df.iloc[i][col]) for col in df.columns            ]) for i in range(len(df))        ])    ]    *# Combine header and body in a dbc.Table*     table = dbc.Table(table\_header + table\_body, bordered=True, dark=True, hover=True, responsive=True, striped=True)       return table  ​  def insertModuleLayout():     return dbc.Container([         dbc.Row([             dbc.Col([                 dbc.Input(                     id="module-code-insert",                     type="text",                     placeholder="Module Code"                ),                 html.Br(),                 dbc.Input(                     id="module-name-insert",                     type="text",                     placeholder="Module Name"                ),                 html.Br(),                 dbc.Input(                     id="number-of-students-insert",                     type="number",                     placeholder="Number of Students"                ),                 html.Br(),                 dbc.Input(                     id="pgtas-recruited-insert",                     type="number",                     placeholder="PGTAs Recruited"                ),                 html.Br(),                 dbc.Input(                     id="exam-weight-insert",                     type="number",                     placeholder="Exam Weight"                ),                 html.Br(),                 dbc.Input(                     id="coursework-weight-insert",                     type="number",                     placeholder="Coursework Weight"                ),                 html.Br(),                 dbc.Input(                     id="delivery-code-insert",                     type="text",                     placeholder="Delivery Code"                ),                 html.Br(),                 html.Div("Select duties", style={"font-weight": "bold"}),                 dbc.Checklist(                     id="base-duties-checklist-insert",                     value=[],                     options=[{'label': duty, 'value': duty} for duty in duties],                ),                 html.Br(),                 dbc.Button(                     "Insert Module",                     id="insert-module-button",                     color="primary"                ),                 html.Br(),            ], width=6),        ]),         html.Br(),         dbc.Row([             dbc.Col([                 html.Div(id="insert-module-alert")            ], width=4)        ])    ], fluid=True)  ​  def insertModule(n\_clicks, module\_code, module\_name, number\_of\_students, pgtas\_recruited, exam\_weight, coursework\_weight, delivery\_code, duties):     if n\_clicks is not None and n\_clicks > 0:         if all([module\_code, module\_name, number\_of\_students, pgtas\_recruited, exam\_weight, coursework\_weight, delivery\_code, duties]):  *# Insert data into database*             new\_module = CombinedVariables(                 module\_code=module\_code,                 module\_name=module\_name,                 number\_of\_students=number\_of\_students,                 pgtas\_recruited=pgtas\_recruited,                 exam\_weight=exam\_weight,                 coursework\_weight=coursework\_weight,                 exam\_coursework\_ratio=f'{exam\_weight}:{coursework\_weight}',                 delivery\_code=delivery\_code,                 duties=(', ').join(duties),            )             session.add(new\_module)             session.commit()             session.close()             return dbc.Alert(                 "Module successfully inserted!",                 color="success",                 duration=4000            )         else:             return dbc.Alert(                 "All fields must be filled to insert a module!",                 color="danger",                 duration=4000            )     return ""  ​  def deleteModuleLayout():     return dbc.Container([         dbc.Row([             dbc.Col([                 dbc.Input(                     id="module-id-delete",                     type="text",                     placeholder="id of Module"                ),                 html.Br(),                 dbc.Button(                     "Delete Module",                     id="delete-module-button",                     color="danger"                )            ], width=6)        ]),         html.Br(),         dbc.Row([             dbc.Col([                 html.Div(id="delete-module-alert")            ], width=4)        ])    ], fluid=True)  ​  def deleteModule(n\_clicks, id):     if n\_clicks is not None and n\_clicks > 0:  *# Delete data from database*         if not id:             return dbc.Alert(                 "Fill in the module's id!",                 color="danger",                 duration=4000            )         module = session.query(CombinedVariables).filter(CombinedVariables.id == id).first()         if not module:             return dbc.Alert(                 "Module not found!",                 color="danger"            )         session.delete(module)         session.commit()         session.close()         return dbc.Alert(             "Module successfully deleted!",             color="success",             duration=4000        )     return ""  ​  ​  *(app.py)*  ​  *# Callback for Table Display*  *@app*.callback(     Output("table-content", "children"),    [Input("database-tabs", "active\_tab")]  )  ​  def update\_displayTable(active\_tab):     return displayTable(active\_tab)  ​  *# Callback for Insert Module*  *@app*.callback(     Output("insert-module-alert", "children"),    [Input("insert-module-button", "n\_clicks")],    [State("module-code-insert", "value"),     State("module-name-insert", "value"),     State("number-of-students-insert", "value"),     State("pgtas-recruited-insert", "value"),     State("exam-weight-insert", "value"),     State("coursework-weight-insert", "value"),     State("delivery-code-insert", "value"),     State("base-duties-checklist-insert", "value")]  )  ​  def update\_insertModule(n\_clicks, module\_code, module\_name, number\_of\_students, pgtas\_recruited, exam\_weight, coursework\_weight, delivery\_code, duties):     return insertModule(n\_clicks, module\_code, module\_name, number\_of\_students, pgtas\_recruited, exam\_weight, coursework\_weight, delivery\_code, duties)  ​  *# Callback for Delete Module*  *@app*.callback(     Output("delete-module-alert", "children"),    [Input("delete-module-button", "n\_clicks")],    [State("module-id-delete", "value")]  )  ​  def update\_deleteModule(n\_clicks, module\_code):     return deleteModule(n\_clicks, module\_code)  ​ |