Linear Regression Model of Categorical UW Course Data

Collection, Processing, and Analysis

Introduction:

In undergraduate studies, course selection is often a hot topic for debate. Discussions include challenging courses, the difficulty of different disciplines, and planning the timing of particularly demanding classes. Traditionally, students have relied on anecdotal advice from senior peers, which, while valuable, provides a perspective based on personal experience that may not comprehensively represent the academic environment. This project aims to supplement anecdotal insights with data analysis using a linear regression model machine learning. To provide a more rounded understanding of the factors that influence course difficulty. By analyzing various attributes of courses, such as department, level, and campus, against the historical grade distributions, I seek to identify patterns and trends that can inform students' course planning decisions.

Part One: Collection

The first step in the process is to collect the data. In my case, there are two sources that I will pull from: the UW course catalog and the DawgPath" course database. DawgPath is an internal UW site that contains a GPA distribution and other categorical data. I will start with the course catalog and use it to escape the department keywords for each course prefix. I will use it later to leverage word frequency as an additional metric. Then, I will need to collect each course from the DawgPath database.

Course Catalogue

The UW course catalog offers information about course departments, which is used for gathering keywords associated with each course prefix. These keywords will later be instrumental in analyzing word frequencies and deriving additional course metrics. The UW course catalog is hosted as a simple HTML page, meaning scraping will be relatively straightforward, especially using the BeautifulSoup package, allowing us to sort through the HTML directly by headers, reducing complexity. This function constructs URLs for each department listed in the course catalog for different campuses. I achieved this by pulling the top-level campus course catalog and selecting all the links. I then add any link that matches the Regex pattern for a course URL. This is then returned as a list of URLs. This function constructs URLs for each department listed in the course catalog for different campuses. Next, the departmental HTML content is scraped and stored. I then collect all of the words in the department descriptions. I filter out redundant words and return a list of departments associated with each course prefix.

```
def build_department_urls():
    # Define the URL suffix for each campus
    # Loop through each campus to construct department URLs
        # Loop through each link header to find department URLs
                # Store the full URL for each department
    return urls
```

```
def scrape_and_save_html(urls):
    # Loop through each department URL to scrape its HTML content
        # Store the prettified HTML content
    return html_dict

def process_department_html(html_dict):
    # Process the HTML content to extract department words
        # Extract and clean the department header text
        # Filter out unwanted words
        # Store the words for each department
    return dep_word_list
```

DawgPath Database

DawgPath is an internal University of Washington (UW) website utilized by students and advisors to access historical GPA distributions and other relevant course data. Interestingly, the JSON data for each course, sourced from DawgPath, contains more detailed information than what is visible on the website. The data collection process involves constructing URLs for API requests. Each UW campus has a top-level JSON file listing all courses in its database. The method is iterating through each campus, retrieving course codes, and appending them to the standard JSON URL. Once complete, each course's JSON is requested and added to a new JSON under the concatenation of the department prefix and course number. The resulting JSON is ~10826 entries and ~28MB. This structured approach enables the efficient gathering of course information from DawgPath for my analysis.

```
def get_courses():
    # Define the campuses to fetch course data for
    # Loop through each campus
        # Construct the URL for the API call
        # Evaluate the text response to Python list
    return courses

def build_urls(courses):
    # Loop through each course to construct its URL
        # Extract course prefix and number
        # Store the full URL with the course key
    return urls

def get_jsons(urls):
    # Loop through each URL and fetch the JSON data
        # Parse the JSON response
        # Log output as working to a file in case of interruptions
    return raw_jsons
```

Part 2: Processing

In this second phase of my work, I concentrate on refining the collected data to improve its usability and efficiency for analysis. This involves a series of steps aimed at streamlining the dataset for better performance

in subsequent processing and to work towards a format that my eventual model will understand.

Remove Extra Data

Now that the data is collected, I must turn it into something usable. To start, I need to reduce the size of the dataset. Removing the extra data upfront will reduce the time cost of the later functions. First, I removed all of the courses that returned errors. It only eliminates ~9 classes, meaning the data set is pretty clean, and the scraping process worked as intended. I also removed extra information I didn't need from the prereq data field. The most significant removal is the extraction of only courses with GPA distribution, which eliminates ~4100 entries. Since my course data will be compared to the grades, I don't want any classes that don't give grades.

```
def remove_errors(data):
    # Remove courses from the dataset that contain a specific error pattern.
    # Searches for courses with an error message matching the pattern
'.*Course.*'.
    return data

def remove_options(data):
    # If 'prereq_graph' is present, deletes the 'options' key from it.
    return data

def get_gpa_courses(data):
    # Identifies courses 'gpa_distro' is empty or has a total count of 0.
    # Iterates through the dataset and deletes such courses.
    return data
```

Clean Data

The data cleaning and feature extraction for the course dataset involved steps to enhance analysis. Initially, I calculated and added 'percent_mastered' to the DataFrame, representing the percentage of grades above a GPA of 30. I then standardized the course levels by rounding each course's identifier to the nearest hundred.

```
def percent_mastered(data):
    # Calculate the percentage of high grades (GPA >= 3.0)
    return data

def add_level(data):
    # Deduce course level from course_id and add as new column
    return data
```

Subsequently, I flattened the 'coi_data' from each course, creating separate columns like 'course_coi', 'course_level_coi', 'curric_coi', and 'percent_in_range'. These are UW's internal metrics for course difficulty; they are incredibly sparse and are not a good metric for analysis, but including the data helps increase the reliability of the model and increases the amount of variance accounted. The 'concurrent_courses' data was processed to identify common course pairings and student course load, and I added columns indicating prerequisites and subsequent courses to understand curriculum progression.

```
def flatten_coi_data(data):
    # Extract complexity of information (COI) data and add as new columns
    # Initialize lists for each new column
        # Append COI data, or None if missing
    # Add new columns to DataFrame
    return data

def flatten_concurrent_courses(data):
    # Create a set of concurrent courses, ensuring each key is space-free
        # Create a set of courses without spaces, or None if missing
    return data

def flatten_prereq(data):
    # Add prerequisite relationship data as new columns
        # Generate sets for courses that are prerequisites or have this course as
a prerequisite
    return data
```

I then tackled the 'course_offered' data for each course. I added a new column, 'course_offered', to the DataFrame by extracting the quarters in which each course is available. This column holds a set of quarters specific to each course. I also accommodated special cases, such as jointly offered courses and entries split by ';'. Where 'course_offered' data was unavailable or lacked specific quarter information, I assigned None.

```
def flatten_course_offered(data):
    # Extract quarters when courses are offered and add as new column
        # Map abbreviation to full quarter name, add None if missing
    return data
```

Next, I refined the 'course_description' for each course. I removed prepositions, short words, and numeric terms and created a new 'course_description' column in the DataFrame with clean and concise descriptions. This involved breaking down each description into individual words and filtering them based on length as an easy way to ensure a focus on substantial, course-relevant content.

```
def flatten_description(data):
    # Clean and process course descriptions
    # Remove punctuation, split into words, and filter based on length and type
    return data
```

Furthermore, I mapped each course to its respective departments using a pre-constructed dictionary derived from scraping the course catalog. Using the 'course_abbrev' column, I added a set of department strings to each course entry. Lastly, I removed all the columns I had extracted data and no longer needed. This approach of continuous data trimming helps in reducing runtime and minimizing errors.

```
def add_departments(data):
    # Map courses to their departments
```

```
    # Use department dictionary to map courses, add None if not found
    return data

def remove_extra_columns(data):
    # Remove columns that are no longer needed from DataFrame
    # Delete column if it exists
    return data
```

All the data in my data frame is now organized and easy to manipulate. I need to do one more cleaning phase to format the data in a format relevant to the regression model.

Prepare Data

Now, I need to prepare the data frame for analysis in a machine-learning context. I need to turn any field with categorical data into new columns of booleans. This is only sometimes possible as some features have thousands of categories, like in the case of 'course_description'. For 'course_description' and 'course_title', I can count the frequency of each word and then a variable amount of 'top_words' and see what works best. For 'concur_courses', I used the mean level of all the concurrent courses. It will also be easier to compare to the 'course_level' data. It begins with calculate_mean_level, a function that computes the average level of concurrent courses by extracting and averaging the last three digits of each course code, rounding to the nearest hundred. The get_words function then analyzes the 'course_description' and 'course_title' columns to count word frequencies and identify the top x most frequent words, creating a new DataFrame indicating these words' presence in course descriptions and titles.

```
def calculate_mean_level(concurrent_courses):
    # Calculate the average level of the concurrent_courses and round to the
nearest 100
    return mean_level

def get_words(x):
    # Count the frequency of each word
    # Identify the top 10 most frequent words
    # Add a new boolean column per top word for each course
    return pd.DataFrame(top_word_columns)
```

Next, I converted the numerical values from 'course_credits' to the smallest number of credits possible and filtered 'course_title' to retain words longer than four characters. Courses are categorized into STEM and Humanities based on department affiliations that we scraped from the course catalog. I turned the 'has_prereq' column into a boolean by checking for any prerequisites. Lastly, I convert the 'course_offered' and 'course_campus' fields to individual boolean columns.

Part 3: Analysis

I begin by splitting the dataset into training and testing sets, using a 25% test size and setting a random state for reproducibility. This division is used for evaluating the model's performance on unseen data. After splitting, I create a pipeline that combines the StandardScaler for feature scaling and the linear regression

model. The StandardScaler standardizes the features (independent variables) by removing the mean and scaling them to have unit variance before use.

Once the pipeline is set up, I fit it with the training data, which involves learning the linear relationship between the features (X) and the target variable (Y). The process of 'training' or 'fitting' the model is accomplished by finding the values of coefficients that minimize the difference between the observed values and the values predicted by the model. This is usually done using Ordinary Least Squares (OLS), which aims to minimize the sum of the squares of the residuals (the differences between observed and predicted values). The model is then used to predict the target variable on the testing set.

```
# Split the data into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25,
random_state=0)

# Create a pipeline for scaling and linear regression
pipeline = make_pipeline(StandardScaler(), LinearRegression())
pipeline.fit(x_train, y_train)

# Predict using the model
y_pred = pipeline.predict(x_test)
```

The model's performance is evaluated using several metrics. Mean Squared Error (MSE) and Root Mean Squared Error (RMSE) give an idea of the error magnitude. The MSE measures the average squared difference between the actual and predicted values. Squaring the errors gives more weight to larger errors. The RMSE is the square root of MSE. It brings the error metric back to the same scale as the dependent variable, making it more interpretable. Mean Absolute Error (MAE) is the average of the absolute differences between predicted and actual values. Unlike MSE, it treats all errors equally, regardless of their size. It provides a straightforward interpretation of the average prediction error. The coefficient of Determination, or R^2 score, indicates the proportion of the variance in the dependent variable that is predictable from the independent variables. A score of one means perfect prediction, while a score of zero indicates that the model is no better than a model that predicts the mean of the dependent variable for all observations.

```
# Evaluate the model's performance
mse = mean_squared_error(y_test, y_pred)
rmse = math.sqrt(mse)
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

Results

After some trial and error and tweaking, I could reliably produce these results.

```
Mean Squared Error (MSE): 0.01
Root Mean Squared Error (RMSE): 0.11
Mean Absolute Error (MAE): 0.08
Coefficient of Determination (R² score): 0.31
```

The MSE, RMSE, and MAE might seem good initially, but the R^2 score shows the true picture. The model only explains about 31% of the change in the percentage of students who mastered the topic. In addition to that 31%, the model is wrong by about 11%. Our results are arguably meaningless, with most features representing less than 2.0%. However, for this project, I will continue as if they are and, in the end, see if, despite the errors, there is any insight to glean from the data.

```
Campus Features:
  seattle: 2.05%
  tacoma: -1.54%
```

Next was the campus features. You might notice that despite scraping all three campuses, only two have been used in the model. The reason is that the columns are supposed to be independent of each other, and by including all three campuses, we would have added a completely dependent system where every value would have been one of the three and never more than one. It causes the regression model to collapse and renders those columns useless. As the data shows, the Seattle campus is associated with a ~2% increase, and the Tacoma campus with ~ -1.5%.

```
Season Features:
  offered_winter: 0.01%
  offered_summer: -0.40%
  offered_spring: 0.34%
  offered_autumn: -0.04%

Bottleneck and Gateway Features:
  is_bottleneck: 0.25%
  is_gateway: -0.76%
  has_prereq: 0.28%
```

The season features and the degree map features produced no significant impact. The only interesting bit is that 'has_prereq' was consistently associated with a positive increase, and 'is_gateway' was consistently negative. Considering that, by definition, gateway courses are prerequisites for other classes; the data seems to suggest that the prerequisite classes are harder than the actual classes they are preparing you for.

```
Course Level Features:
  course_level: 2.04%
  mean_concur_level: 0.62%

Discipline Features:
  is_humanities: 1.21%
  is_stem: -0.52%
```

The last of any meaningful categories were the course level and discipline features. The course-level features support the same idea as the degree map features. Namely, the farther in the degree, the better students do,

despite the assumedly harder coursework.

```
Top 5 'word_' Features and their Impact:
  word_minimum: -2.70%
  word_grade: 1.29%
  word_three: 0.88%
  word_either: -0.77%
  word_health: 0.66%
```

Finally, the word features. The features are hard to classify because the words represent multiple ideas in different classes. For example, the most impactful word was 'minimum,' but there is no way to distinguish between a concept of prerequisites or math and could potentially include courses from another. A similar problem exists with most of the words on the list. Thus, words mostly representative of one idea are the only ones where conclusions can be drawn. Of, the theory/theories are the only ones above 1% of absolute change. This indicates that classes heavy in concepts and perhaps light on application are associated with a negative impact.

Conclusion

Although the model never produced any statistically significant results, the project was an excellent exercise and learning experience for Python, web scraping, data processing, machine learning applications, and data analytics. One metric wasn't included because it overlaps with the metric we are using for our y values, the number of people who received a zero or dropped the class. If I add a couple of lines to the code and one more category:

```
df['number_of_drops'] = df['gpa_distro'].apply(lambda distro: sum(grade['count']
    for grade in distro if int(grade['gpa']) == 0) / sum(grade['count']
    for grade in distro))
```

Then exclude 0 GPA from our percent_mastered:

```
df['percent_mastered'] = df['gpa_distro'].apply(lambda distro: sum(grade['count']
    for grade in distro if int(grade['gpa']) >= 30) / sum(grade['count']
    for grade in distro if grade['gpa'] != '0'))
```

I am left with the single most impactful feature of the whole project despite the rest of the model remaining relatively the same, which wasn't initially included because the data was for personal use. While I can't predict the future, I don't plan to drop any courses and thus wasn't looking in that direction:

```
number_of_drops: -3.01%
```

This is although the students with zero are no longer a factor in the percentage. While you may assume this is because it represents classes that are more heavily < 3.0, the number of zeros is more independent from the

rest of the grades than expected. The raw GPA distro showed that unlike the top quartile, where there is a standard distribution of grades, the bottom quartile is almost exclusively 0. Other impactful features like the course level features and the bottleneck, gateway, and prereq features support this. The farther along the degree you get, the better the students do. Mainly because fewer of them drop classes as they are more invested and would only be able to register for the higher classes if they had shown a history of not dropping classes. It also explains the difference in campus. It's not that the Seattle campus is easier; the prestige of the Seattle campus allows them to be more selective about who they admit and thus only admit students with a track record of already doing well.

So, to answer the question, what classes are the easiest? The classes you show up to.