Overview

Today's education system is based on one task only: improving test scores. Instead of focusing on employment outcomes for their students, schools and teachers are forced to focus on increasing their students' test scores, as local, state, and national school boards have changed educational focuses towards easily quantifiable metrics that can be tested in standardized tests. As a result, a heavy emphasis has been placed on what can affect a student's exam score, and how to best accommodate students such that they receive higher marks. Similarly, this has led to the rise of tests such as the STAR test, Advanced Placement (AP) exams, the SAT, the ACT, and the GRE, alongside traditional exams and finals, which hope to quantify student performance metrics in one (or a few) easy-to-understand scores. As a senior who has spent the past few months studying for, and subsequently taking, the GRE, this topic is personal to me as I hope to better understand how to approach the studying process.

This project investigated the effect of various characteristics and factors of a student's surroundings, environment, and schooling, and how these would affect their exam scores. The goal of this section is to see if it is possible to create a model that can predict the score of a student based on these factors. The data for this study came from a Kaggle dataset with around 6,400 data points and 20 different variables for students and their exam scores. For this analysis, I focused on a smaller sample of variables but kept the rest in the dataset as it was easier to not delete everything.

Dataset Overview

The dataset is taken from Kaggle, and is stored in the final_project subfolder as StudentPerformanceFacIt has a total of 20 different variables, listed below. The dataset is composed of i32 and categorical strings of 2-3 values.

- **Hours_studied** - amount of hours studied, saved as an i32
- **Attendance** - attendance as an i32 from 0-100
- **Parental_involvement** - parental involvement in a student's education level, as a categorical variable of 3 values, Low, Medium, or High
- **Access_to_resources** - access to resources to help a student study, as a categorical variable of 3 values, Low, Medium, or High
- **Extracurricular_activities** - if a student does extracurricular activities, as a categorical variable of 2 values, Yes or No
- **Sleep_hours** - amount of hours slept by a student, as an i32
- **Previous_scores** - previous exam score of the student, as an i32
- **Motivation_level** - motivation level of a student, as a categorical variable of 3 values, Low, Medium, or High
- **Internet_access** - internet access for a student, as a categorical variable of 2 values, Yes or No
- **Tutoring_sessions** - amount of tutoring sessions attended by a student, as an i32
- **Family_income** - the family income of the student's family, as a categorical variable of 3 values, Low, Medium, or High
- **Teacher_quality** - the quality of the student's teacher, as a categorical variable of 3 values, Low, Medium, or High
- **School_type** - type of school the student is enrolled in, as a categorical variable of 2 values, Public or Private
- **Peer_influence** - the influence of peers on the academic performance of the student, as a categorical variable of 3 values, Negative, Neutral, or Positive
- **Physical_activity** - hours of physical activity the student does, as an i32
- **Learning_disabilities** - whether the student has a learning disability, as a categorical variable of 2 values, Yes or No
- **Parental_education_level** - the education level of the students' parents, as a categorical variable of 3 values, High School, College, or Postgraduate
- **Distance_from_home** - distance of a school from the student's home, as a categorical variable of 3 values, Near, Moderate, or Far
- **Gender** - gender of the student, as a categorical variable 2 values, Male or Female
- **Exam_score** - the exam score of the student (the output we are predicting), as an i32

The Kaggle dataset can be found at this link, but is also inside the project folder.

Writing the Code

This section goes over how I wrote the code, specifically by going over each commit and what I did, problems I encountered, and how I overcame challenges while coding.

Commit 1

This is the first commit, made when I did Homework 5, which was based on a completely different project. This is irrelevant to the project, but it is retained for record-keeping purposes.

Commit 2

First, before I wrote my code, I set out a plan inside my Project Proposal document. Seeing the structure I made, I first began by creating the StudentRecord struct which held the data, and then the Graph struct which would hold the node graph I used for analysis. I first started with having an id number inside StudentRecord, but changed it after realizing I could just have the id in the Graph struct. As for the Graph struct, I spent the first part of my project with edges (connections) stored as HashMap<usize, HashSet<usize>>, before changing it to HashMap<(usize, usize), u32> before again changing it to an adjacency list in the format of HashMap<usize, Vec<(usize, u32)>>. The changes would result in several minor changes to the functions that used edges, but nothing that was overly complicated to implement. In this first commit I added under impl Graph a new() function to create new graphs, add_student() to add a student, and add_edge() to add edges between students. I also added a read_csv() function that reads the dataset and puts it into a graph.

Commit 3

Created the README.md. This would later change as the project evolved.

Commit 4

For my second code-centric commit, I began by changing the graph to be edge-weighted so that I could take into account the degree of connection (weight). This was done with a calculate_weight() function that tied into read_csv(). I also made some changes to add_edge() since here is when I shifted from HashMap<usize, HashSet<usize>> to HashMap<(usize, usize), u32> since this allowed me to store weights (u32) of a connection (usize, usize) between nodes with the usize being the ids of the node. No other changes of note were made here.

Commit 5

Here, I test-ran the code and encountered some bugs, such as wrong capitalization and some issues in my read_csv() function, that I then fixed. I also added a print() function to my Graph impl since I wanted to be able to see the graph. Given the size of the graph (6,000+ entries), I made it a modified print statement that printed out a set number of entries of the graph based on the inputted parameter.

Commit 6

I fixed a minor dataset error, since I had accidentally messed with the data after downloading it. Nothing else happened in this commit.

Commit 7

Updated README.md to a more accurate project description. This would again be updated later.

Commit 8

Merged the README.md edit and main code branches.

Commit 9

Here, I performed the first data science operation on my graph, creating a degree_centrality() function to compute the degree centrality of each node of the graph. I followed the basic formula, counting the number of edges per node, storing it in a HashMap, and returning the HashMap. I also commented parts of my code here for readability (I would go on later to heavily comment on this code) and fixed some warnings within the code of unused elements in StudentRecord.

Commit 10

I first began this commit by creating clusters() which would find connected components (clusters of nodes) in the graph. However, after running some tests, I instead decided to add parameters to filter search for clusters based on a few characteristics and weights since the graph was very connected. However, in doing so, I realized that a function to properly change strings to the new variables' required attributes was needed. As a result, I created the get_attribute() function under StudentRecord to convert from a string of a varaible name to the value of that variable within StudentRecord. With this tool, I then added filtering on weight and attribute to clusters(), allowing me to narrow down the search of clusters to better understand the data.

Commit 11

I worked on creating the closeness_centrality() function? Within the Graph impl to calculate the closeness centrality of each node in the graph. In doing so, I created a helper function shortest_path() which would find the shortest distance between the nodes. However this helper function would cause me a lot of issues, as it would take too long to compute a single node. This was due to what I later determined was two reasons: the side of the graph, and the usage of the wrong search method (BFS). I did not discover this until a few commits later. However, it seemed that the shortest_path() function was working on smaller cases, so I continued to write the closeness_centrality() function without testing it by following the formula for it. I resolved to bug test the function later.

Commit 12

Given my issues with shortest_path(), I created my first test to figure out if it really worked using a smaller test case. In doing this test, I realized that the current implementation worked, and that my major issue was going to be runtime, as it would take around 5-7 second to pass through each node, and with a few thousand nodes, this would make it impossible to run in a reasonable time, instead taking hours.

Commit 13

After checking lecture notes and performing additional research , I found that Dijkstra's algorithm would be a better way to compute the shortest path. However, in doing so, I realized that it would probably be easier for me to change the Graph to an adjacency list for edges, so I changed Graph to HashMap<usize, Vec<(usize, u32)>> which is a HashMap of (id, Vec(adjacent, weight)) which was much better implementation. Doing so required me to modify a bunch of code that relied on the old implementation of looking at edges, but it was very simple and did not cause any problems. Creating Dijkstra's algorithm was a little complicated, but based on the model I found online and within the Lecture 15 notes, I was able to code an implementation of the section that worked well. I left the old BFS method as a block comment in case I needed it, but otherwise used Dijkstra's algorithm.

The other major change was how I loaded the data. After discussing with Professor Chator, I realized that I could achieve much of the same results while greatly decreasing the runtime by only loading a select amount of the data into the graph. I therefore changed read_csv() to only accept 20% of the inputted lines(increasing to 30% later), and ignoring the rest (later, this ignored data would be converted into test data for a DecisionTree). This commit took a very long time to write given how complex it was and the amount of changes, but this was when I began to see major progress in my project and realized I was nearing a good point.

Commit 14

In this commit, I created a decision_tree() function to build a decision tree that would later be used to predict a person's exam score. The creation of the function was easy and did not take much time. Significant strife came, however, when I was creating the model, as I was experiencing major errors with DecisionTree::params().fit(). First I tried to create separate features and labels and inputting both, which encountered errors in dim size and number of inputs. So I followed this up by trying to create a Dataset from linfa to take in the data. However, this also encountered dim errors. After a lot of debugging, I began to understand the error was due to my dimensionality of the data, so wrote a series of different test and other things to determine how best to fix it, before landing on a rather annoying set of code that created the targets and records and then recast their shape into the dataset twice, and then using that dataset to build the model. In the process of debugging, I also downgraded ndarray from 0.16.1 to 0.15.6 (which I saw online would help, and seems to have without adversely affecting my data). This resulted in a completed DecisionTree, however, I was unable to test its implementation until later as in this commit I had not built the prediction() function. I also fixed the print() function implementation because of errors.

Commit 15

Here, I wrote the prediction() function to predict exam scores based on the model for a single input student. Doing this was incredibly easy; it involved copying some code from decision_tree() and adding the prediction interpretation code that DecisionTrees had built-in. I also added an accuracy() function outside the Graph impl to calculate the accuracy of the model I did not immediately realize when I made the function that my calculator was slightly wrong, but did not significantly change the accuracy (which droppe from 98% to 97% accuracy). I also took the 80% of the data that was not loaded and turned it into the test model, which required some changes to read_csv(). I then ran all the code at once to see the data and outputs.

Commit 16

Here, I wrote 3 more tests to test my code (all working), cleaned up my code, and fixed an error in the calculator of the accuracy of my model (wrong format for percent difference). I also added a ton more code in main{} to run when the whole file runs, to better analyze the data. All major code writing was now completed, save for commenting/cleaning up the code and some minor changes to main{} as I further analyze the data.

Commit 17

Updated README.md to explain the project more in-depth, explain the files in the GitHub, and where the dataset is from.

Commit 18

This is the last major commit. I moved all the code into modules to simply final changes and make the program easier to work with. I should have done this earlier, but alas, it's better late than never. This helped fix some issues and made code easier to read. Two modules are created, *graph.rs* that runs all of the graph functions (except *decision_tree()*) and *tree.rs* which runs all the DecisionTree functions. Therefore *graph.decision_tree()* became *decision_tree()*.

I also moved prediction() outside of the Graph impl since it did not use any Graph implementations so it made no sense to be there. I also changed read_csv() to take in a value between 0.0-1.0, as well as changing the amount of data going into train_graph from 20% to 30%, thus reducing the data in test_graph from 80% to 70%. Some minor changes to main{} made the code more readable and gave me better outputs for a few functions using println!() statements. Demarcated sections in main{}. Fixed an error in Dijkstra's algorithm as I wasn't accounting for weights when I should have been. I also added *feature_importance()* and *altered_graph()* to help with analysis of the decision tree model. Added the project writeup to the project folder.

Each Function

This section is organized through each module. A brief explanation of each module is provided at the top of each section. The purpose of this section is to understand each function, how they work, and the organization of the code.

graph.rs

This module is focused on all functions relating to the graph, modifying it, and creating analysis from them. The majority of the code is here, alongside the implementation of the StudentRecord and Graph structs and their impls. All functions are public.

StudentRecord impl functions

fn get_attribute(&self, a: &str) -> Option<String>

This is used as a helper function for *graph.clusters()* to convert between Option<Vec<&str>> and the respective value of a student for that specific variable within StudentRecord. It takes in one parameter, a string (&str) that is supposed to match the exact name of a variable within StudentRecord. It returns a string (Option<String>) of the value of that variable. It is never used outside that function.

Graph impl functions

fn new()-> Self, also written as Graph::new()

Creates a new graph with empty nodes and adjacency list. Takes in no parameters. Can be used to clear all data from an existing graph by going graph.new(), or create a new blank graph using Graph::new().

fn add_student(&mut self, student: StudentRecord, id: usize)

Adds a new student with an attached StudentRecord and id to the graph. It takes in two parameters, a student (a StudentRecord struct) and an id (usize). It then adds to the HashMap the student and their id to the node, as well as adding an entry to the adjacency_list with the id, with no connections.

fn add_edge(&mut self, id1: usize, id2: usize, weight: u32)

Creates an edge for the graph. Takes in id1 (usize), id2 (usize), and a weight (u32). It adds to both id1 and id2's adjacency_list a connection between the two, with the set weight given in the function call.

fn print(&self, mut lines1: i32, mut lines2: i32)

Graph.print() is a basic function implemented to print out information on the nodes and connections between nodes (edges) of a graph. It takes in two parameters, lines1 (i32) which prints lines1 amount of nodes and their attached StudentRecord files, and lines2 (i32) which prints out all the connections (edges) between lines2 amount of nodes. Due to the size of the graph, I recommend keeping the value of lines2 low.

fn degree_centrality(&self) -> HashMap<&usize, i32>

Calculates the degree centrality of each node in the graph. This takes in no inputs and outputs a HashMap<&usize, i32> which has the id of a node (usize) and its associated degree centrality value (i32).

fn clusters(&self, weight: u32, filter: Option<Vec<&str>>) -> Vec<Vec<usize>>

Calculates the connected components of the graph to find clusters of nodes. Given that the data is heavily clustered, I have decided to allow for parameters to refine the search for connected components, a weight (u32) threshold, and a filter (Option<Vec<&str>>) of attributes within the StudentRecord of each student. To call the function, do graph.clusters(weight value, Some(vec![string(s) of attribute names"]). Make sure the attribute names are an exact match for those within the StudentRecord struct, otherwise the filter for that attribute will not work. It will output a Vec<Vec<usize>> which will be a vector containing each cluster of nodes, with the sub-vector having each id that is connected. This function uses the *studentrecord.get_attributes()* helper function to aid in converting from the filter Option<Vec<&str>> into a readable parameter for a comparison.

fn shortest_path(&self, id1: usize) -> HashMap<usize, u32>

Implemented as a helper function for closeness centrality, this function calculates the shortest distance between two nodes using Dijkstra's algorithm. It takes in an id (usize) and outputs a HashMap<usize, u32> of the distance (u32) to every other node on the graph (usize). Any node that is not connected to has a distance of u32::MAX (4,294,967,295).

fn closeness_centrality(&self) -> HashMap<usize, f64>

This function calculates the closeness centrality value of each node in the graph. It takes in no parameters and outputs a HashMap<usize, f64>, which is the id of a node (usize) and the corresponding closeness centrality value (f64). This function uses the helper function *graph.shortest_path()* to compute the closeness centrality.

Generic Functions

fn read_csv(path: &str, graph1: &mut Graph, graph2: &mut Graph, percent: f64) -> Result<(), Box<dyn Error>>

Before inputting a file, you need to create two graphs: a test graph and a train graph. This can be done using Graph::new(). As explained previously, due to computational limits on my computer, I had to limit the amount of data going into the train graph to just 20-30% of all data. However, you can easily change this with a parameter, of a value between 0.0-1.0. To then read a file, you need to do read_csv(file path as a string, train graph, test graph, percent of data into train_graph (f64 between 0.0-1.0)), which will read the file path, and then write the data to the test and train graph. The function does not return a graph, instead modifying the two inputted graphs, therefore the result of this function can be reutnred to a dummy variable.

fn calc_weight(student1: &StudentRecord, student2: &StudentRecord) -> u32

This function is used as a helper function for read_csv() to calculate the weights between two nodes and then attack that output to the adjacency_list of the Graph. It takes in two parameters, student1 (StudentRecord) and student2 (StudentRecord) and returns the weight of their connection (u32).

tree.rs

This module is the smaller of the two, and focuses on all of the code relating to the creation and use of the DecisionTree module for this project. It imports the StudentRecord and Graph struct from *graph.rs*, and does not create any structs, instead is full of only public functions.

fn decision_tree(graph: &Graph) -> Result<DecisionTree<f64, usize>, Box<dyn Error>>

This function creates a decision tree based on the input data to calculate the exam score of a student. It takes in no parameters. It uses a total of 9 variables in the decision tree: school type, family income, peer influence, motivation levels, and learning disability (all categorical variables), alongside hours studied, attendance, previous exam score, and amount of tutoring sessions attended (all quantitative). The function outputs a DecisionTree<f64, usize> if it is able to create the decision tree, otherwise it outputs Box<dyn Error>. This does not use the decision tree, instead just makes it for future use.

fn prediction(model: &DecisionTree<f64, usize>, student: &StudentRecord) -> usize

This function predicts a student's exam score using the DecisionTree created in *decision_tree()*. It takes in two inputs, DecisionTree<f64, usize> alongside a student (StudentRecord), and returns a usize result of a person's predicted exam score. The function was previously in the Graph impl before being moved out. This function is used as a helper function for *accuracy()* to predict the accuracy of the model as well.

fn accuracy(graph: &Graph, model: DecisionTree<f64, usize>) -> f64

Calculates the accuracy of a given DecisionTree model over a graph as a value between 0.0 (not predictive at all) and 1.0 (fully predictive, no errors). It takes in two parameters, a graph to test the decision tree (Graph) and the model being used (DecisionTree<f64, usize>). It uses a helper function of *prediction()* to calculate the predicted value of a student in the Graph and then calculates how much error there is over all possible scores, returning the accuracy as an f64 value between 0.0 and 1.0.

fn feature_importance(graph: &Graph, model: &DecisionTree<f64, usize>) -> Result<HashMap<String, f64>, Box<dyn Error>>

This function calculates the importance of each feature, returning a HashMap<String, f64> of a features name, and its importance (as an f64). It calculates it as a difference between the accuracy score of the original model and an altered model. It uses the helper function of *altered_graph()* to help me do this computation by altering the graph per variable to computer differences in accuracy.

fn altered_graph(graph: &Graph, feature: usize) -> Graph

This is a helper function used to compute *feature_importance()* to help with the computation of which feature is useful. The function takes in a graph and a feature id from 0-18 that augments a graph based on the index value, shifting all values for that one feature to the same level. This is not useful on its own, but is useful in the context of the *feature_importance()* function.

main.rs

In this module, all of the functions in the previous two modules are run and interpreted. It is also home to all the testing functions.

main{}

Given that there are no functions in *main{}*, this portion will be broken up by section, marked by block comments such as */* SECTION [NAME/NUMBER] */* within *main{}*. Each section is also separated by *println!("\n\n\n\n\n");* so that print statements in the terminal can be easily read without confusing one sector for another.

/* BUILDING THE GRAPHS */

This section builds the graphs used for all analysis. It creates a train_graph and test_graph that follow the Graph struct. It then uses *read_csv()* to add information to the files with 30% going to train, 70% to test. It also has a *graph.print()* on train_graph to understand the data of the graph.

/* DEGREE CENTRALITY */

This section computes the degree_centrality of train_graph. It calculates the degree centrality of each node, outputting the HashMap from *graph.degree_centrality()*, then calculates the average degree centrality value for the whole graph, alongside showcasing what that means. In this case, it shows that the data is highly related with a very high degree of centrality, with an average of each node being connected to 89% of the total nodes.

/* CLUSTER NODES */

This section computes the connected components of train_graph, outputting the clusters, separating each cluster (connected component) and outputting the final amount of connected components. The filters used are a weight threshold of 3 and 2 attribute filters of school_type and family_income, however, this can be easily changed by changing the code. There is also a commented out section of code here that computes the shortest path of a random node to all other nodes, however this is ignored, as this is not needed in the final analysis.

/* CLOSENESS CENTRALITY */

This section computes the closeness centrality of train_graph. This is the section that has the longest computer time. If added more nodes to the graph, it would greatly increase the runtime of this section. Alongside outputting the closeness centrality value for each node, it also computes the average closeness centrality of the entire network.

/* DECISION TREE */

This is the shortest section. It computes the decision tree using train_graph.

/* MODEL ACCURACY AND TESTING */

This section tests the previously made decision tree and computes its accuracy. It first runs a test over 5 (although this number can be changed by changing test_amount) students in test_graph, outputting the student information, the predicted score, actual score, and offset of the predicted value. Following this the section computes the accuracy of the model using *accuracy()* and outputs the value as a percent.

mod tests{}

I would have liked to implement a test for the prediction based on a decision tree, but given the inherent randomness of a decision tree, it was not feasible. Instead, I chose to do these tests instead. However, after manual testing of prediction() based on test cases in *main{}*, I found that the decision tree was accurate and worked as intended. While the majority of the code in the final implementation is run on train_graph, it was also run on test_graph as well, but was omitted from here due to runtime considerations.

fn test_graph_new()

Tests the *graph.new()* function to ensure it creates an empty graph. No errors.

fn test_degree_centrality()

Tests *graph.degree_centrality()* to ensure that its calculations are right. It uses a pre-built graph using three students that is also used for *test_closeness_centrality()*. No errors.

<u>fn test_closeness_centrality()</u>

Tests *graph.closeness_centrality()* to ensure that its calculations are right. It uses the same pre-built graph as *test_degree_centrality()*. No errors.

<u>fn test_shortest_path()</u>

Tests *graph.shortest_path()* to ensure that its calculations for the shortest path is correct. This was the first test built when *graph.shortest_path()* still used BFS, but it works using Dijkstra's algorithm as well. No errors.

<u>Results</u>

To look at the results, I will split this into two sections: one on graph analysis, and a second looking at the DecisionTree model made to predict exam scores. Both offer valuable insights into the data and aid in data analysis.

Edge-Node Graph Analysis

In this project, I made two edge-node graphs, a train_graph and test_graph. Train_graph received 30% of the data while test_graph received 70%. This was done because adding more data to train_graph significantly reduced run time, and after talking to Prof. Chator, this was a workaround that would give me the same results while aiding in reading run time. All numbers given on the graph are subject to some variability since there is randomness on which data is loaded into each graph. However, as I have run this code many times I noticed that all percentages remained roughly the same (within the same percent but decimal percentage differences), so all results are within 1% degree of accuracy and are generalizable over the entire dataset, but I will provide error bars whenever possible or necessary.

In my analysis, I found that these graphs are highly linked, even when considering only 5 categorical variables. This was seen in both the degree centrality calculations, which demonstrated that each node was, on average, connected to 89% (±0.5%) of the other nodes. This is also seen with closeness centrality of 10% (± 1%) meaning that the data is highly linked as well. The high level of interlinking on the data is also confirmed when running *graph.clusters()*, my implementation of connected components, and finding the same result of highly interlinked data and few clusters, even when limiting parameters are added. Having highly linked data, alongside very similar results for connected components, closeness centrality, and degree centrality, suggests that any model that is built from this data will likely be sufficiently accurate to have a strong prediction of the data since there are high levels of linkages between data points.

Model Analysis

Upon seeing this result, I built the decision tree model and tested it. The following are the results from the data. The model was executed with varying amounts of data in the test and train functions, iterating from 10% to 90% of the data on each side of the model. This resulted in a minor change in accuracy of, no more than, 2% (95% for the worst, 97% for the best) when iterated through the best model parameters I found. I iterated through a series of different variables in my testing, before falling on the variables that I used now.

The resulting model created a highly accurate DecisionTree that found that the model was significantly accurate at predicting a student's exam score based on the 9 parameters given (school type, family income, peer pressure/motivation, self-motivation, learning disability, previous exam score, hours studied, attendance to classes, and number of tutoring sessions attended). For example, for a student with the following information:

hours_studied	attendance	parental_involvement	access_to_resources	extracurricular_activities	sleep_hours	previous_scores	motivation_level	internet_access	tutoring_sessions
23	84	Low	High	No	7	73	Low	Yes	0

family_income	teacher_quality	school_type	peer_influence	physical_activity	learning_disabilities	parental_education_level	distance_from_home	gender	exam_score
Low	Medium	Public	Positive	3	No	High School	Near	Male	67

From the created model was a score of in a confidence interval of 65-70, which is very close to their actual score of 67.

When testing the model using *feature_importance()*, I found that the most important feature for the graph was attendance and hours studied, by 1-2 orders of magnitude, with attendance being the greatest predictor. This is supported from research I conducted prior to starting this project, as class attendance and hours studied are typically correlated with incraeased exam scores as it exposes the student to more of the content for longer periods of time in a learning or studying environment. The least important predictor that I test, was whether a student had a learning disability or not. For generality, I also tested every other variable within the model, but all were insignificant.

What Does This Mean?

The resulting data from my project proves that it is possible to predict a student's exam score without them even taking the exam, just solely based on their characteristics. More importantly, the most important stats for anyone when studying for an exam is the quite obvious stats of going to class and studying. The more a person studied and the more classes they attended, the higher their predicted exam score was. At the same time, from my graph analysis, it seems that according to the data, students are a lot more alike than they would think, as there is a high degree of connection between students. This symbolizes that, at least on a macro level, students are far more itnerconnected in terms of their education and relationships to other students, than they would perceive, as metrics for each student are heavily interlinked. From this, I have concluded that it is possible to predict a student's exam scores with an accuracy of ±3 points. As a result, finding out how to maximize a student's score would be as simple as trying to maximize the indicator statistics, ideally increasing attendance and hours studied as the largest indicators, and smaller efforts on the more insignificant variables.

Reflection

This project took around 15-20 hours to complete, which while a long time, was great at helping me better understand how to work Rust code and understand the programming language. Doing this project has greatly helped my understanding of how Rust works and how to use it to solve data science problems while maneuvering around various errors. I got a greater understanding of the course content and helped me connect previous course content within the data science program. As a data science minor, I don't have to take a lot of courses in this field, but this is very helpful in gaining knowledge in data science that I hope to leverage in my majors (international relations and political science). While I did this project a little too late to cross apply over to my GRE exam I took earlier this month, I do find this useful for the rest of my undergraduate career (only one semester left sadly) and very interesting. This was a great project to do although looking back I would probably start this a lot earlier than I did.