Project Report

Problem Description

As a student, one of, if not the most important, things constantly on one's mind is how their courses are going and what final grades they will get for a given course. While, obviously, grades are not everything; they are mostly objective and readily accepted measures of a student's performance. Furthermore, whether or not a student passes a course could make the difference between getting that dream job or not. Thus, belonging to the student population ourselves, this group chose to perform our project on predicting whether or not a given student will pass or fail a given course, given attributes such as assignment, midterm, project, and attendance grades, sleep amount, and more. Furthermore, while an employer might see and only care about the final grade, the far more paramount question to the student population is, in addition to pass or fail, what attributes contribute the most to my final grade, and how can I optimize those attributes throughout the semester?

The dataset this project uses is this dataset from Kaggle. This project elects to use this dataset as it has rich information on course performance, such as midterm and final score, and assignment and quiz average. It also has information on the individual, such as sleep hours; however, as will later be seen, the course attributes proved by far the most important in deciding a student's final grade.

So to reiterate, at a high level, this project will proceed as follows. The initial aim will be to predict pass fail using an FFNN. This will be ultimately unsuccessful, however, and thus this project will switch to a perceptron architecture. Finally, through error analysis, a key insight will be discovered. The end result of this project will then be invaluable insights to a student attempting to optimize the paramount final grade.

Code Description

All of the code discussed below is in this file unless otherwise specified: **PassFail.ipynb**

Data Cleaning and Data Loader

The first thing we do is pull in the data and convert everything to numbers so our model can run on it. We found that parents' educational level had no correlation with grades, and a lot of people are missing that column, so we just dropped it. Next, we convert income level into a range from 1 to 3 corresponding to low, medium, and high. Next, we convert the department to a number using an index. Then we convert grades to 0's or 1's corresponding to a passing or failing grade. We also partition the data into training, testing, and dev sets. We use 70% for train, 20% for dev, and 10% for tests. Next, we define a data loader for our dataset. It takes in which column we are using as features and which column we are using as a target.

Perceptron Training and Evaluation

We first define a function called train_perceptron that takes in a data loader and the number of features. Then it goes through the data and updates the perceptron based on the error. It then returns the finished model weights and bias. After that, we use Pearson correlation to determine which columns to use based on how they relate to Grade. We used every point >.01, but we also found in testing that adding attendance improved performance. Then, for each of the times (before midterm, before final, and after final), we train 10 models and average their weights at the end. This is different from the typical average perceptron, but we found it improves performance dramatically. The difference in training between the times is that before midterm does not use the midterm_score, final_score, or project_score columns. Before final adds back in midterm_score, and after final adds back in final_score and project_score. After training the perceptron, we evaluate it on the dev set and print out a confusion matrix, the accuracy, precision, recall, and F1-Score. We used these metrics to tune our model and see how well we were doing. Finally, we evaluated our model trained with all the columns on the test data set and printed out the confusion matrix and the other measures to fully assess the model's performance.

Error analysis

After evaluating the model on the dev set at each time, we did extensive error analysis on the errors that the model was making after the final. We pulled 15 false positives and 10 false negatives from our model and stored them in a CSV so we could do more specific error analysis in our other file, where we did error analysis. We also printed out the weights for each feature to see what the final model had learned.

The rest of the code discussed for Error analysis can be found in this file:
Correlation_Testing.ipynb

We start by loading all the data as we did before and doing the same preprocessing discussed above. We then defined a create_datasets function, which sets up the train, test, and dev x and y data frames for easy use in our decision tree. We then fit 3 trees on the train data set, using the same features used for each corresponding model. We also set the max node to 10 to limit overfitting of the decision tree and to make the tree more interpretable. We then print out each tree to see how it builds out the decisions. We then evaluate each decision tree on the dev set to see how it performed in comparison to the average perceptron. We looked at all the same metrics we did for the average perceptron described above. Finally, we read in the CSVs generated as described above for the TP and FP examples, and passed each example through the final decision tree to see the path it took to help identify where our model was going wrong.

Our Architecture

As mentioned before, our initial attempt at this problem was building a FFNN between the features found in the dataset and the student's grade we went with an FFNN after looking at the diversity of features that were available in the data set (attendance, assignments , quizzes, sleep information etc). We decided to build a FFNN to account for our hypothesis that these different features must have non linear interactions with each other and the grade feature. The dataset is tabulated and not ordered so we did not consider RNNs or Transformers as those models are designed for data that is ordered/sequenced, we implemented a simple FFNN instead which seemed like the right choice at the time (non linearity hypothesis).

The FFNN had one hidden layer with a ReLU activation and an output layer that produced the scores for Pass (1) or Fail (0), during the training we kept an eye on both the training and development loss and accuracy. When tested however, our results were less than satisfactory; the FFNN was extremely unstable and unreliable.We believe this is due to the relatively small size of the dataset (5000 rows). The overfitting was obvious when we saw the loss and accuracy plots and the difference between the training and dev accuracy. We attempted to address these problems by tuning the hyperparameters, using SMOTE to balance the heavy class imbalance found in the dataset without much improvement to performance.

After these issues with performance (low accuracy, poor recall, poor F1 scores) we decided to implement a simpler linear classifier, an average perceptron instead this would bring down the risk of overfitting. The implementation had 10 perceptron models and we averaged their learned weights and bias which is slightly different from the typical average perceptron however it improved stability in our testing. It was also hypothesised that since we are trying to predict pass/fail, the decision boundary must be somewhat linear from the features due to the poor performance from the FFNN, no hyperparameter tuning was required with this approach so we had more space to focus on the feature selection using pearson correlation coefficients between the features for our error analysis .

The average perceptron approach made it easier to debug, easier to interpret and we expected it to be more stable than the FFNN. Along with our changes to the times when we were predicting the grade (Before Midterms, After Midterms and After Finals) different features became available at different points in the "semester" more information gets added in the features as these intervals progress, we expected to see much better performance compared to our initial attempt.

Results

Performance metrics summary:

As we mentioned above, we evaluated our averaged perceptron classifier at three different points in time: before the midterm, after the midterm but before the final and then after the final. At each of these points in time our model was predicting whether a student would pass (receive a grade of A, B or C) or fail (receive a grade of D or F) based on the features the model was trained on. In the below table the comparison of our model's performance can be seen over the three time periods.

Performance Metrics Summary

Metric	Before Midterm	Before Final	After Final
Accuracy	61.67%	63.67%	70.11%
Precision	0.6167	0.6432	0.7061
Recall	1.0000	0.9225	0.8829
F1-score	0.7629	0.7580	0.7846

Confusion Matrix

Time Moment	True Negative	False positive	False NEgative	True Positive
Before Midterm	0	345	0	555
Before Final	61	284	43	512
After Final	141	204	65	490

Before the midterm the model achieves perfect recall (1.0) but poor precision (0.6167) due to the fact that it predicts all students as "Pass". This indicates that without the exam scores and the project score the remaining features are not sufficient in order to distinguish students who fail from students who pass the course. Moving onto before the final, here the midterm feature is included in the training and testing of our averaged perceptron and it begins to be able to identify when students fail (now identifying successfully that 61 students failed) but it still retains a high false positive rate. The precision improves slightly going from 0.6167 to 0.6432 but in

doing so the recall does get worse going down to 0.9225. In the final time period we evaluate, after the final, the final exam score and the project score are included. By including these features the performances increase to 70.11% overall accuracy and has the highest F1-score of the three time periods (F1-score = 0.7846). The introduction of these two features drastically improves the model's ability to correctly identify failing students (reaching 141 true negatives) which increases the precision to 0.7061. In doing so the recall does drop to 0.8829 but the balance of greatly increased precision in expense of recall does cause the overall model's performance to improve greatly through the introduction of the midterm score, project score and final score. These results demonstrate that the predictive accuracy of the model improves as more academic performance data becomes available, with exam scores along with the project score being highly important features for distinguishing whether students pass or fail.

Dev vs Test Comparison Table

Metric	Dev Set (n = 900)	Test Set (n = 500)
Accuracy	70.11%	70.80%
Precision	0.7061	0.7100
Recall	0.8829	0.8704
F1-Score	0.7846	0.7821

Test Set Confusion Matrix

	Predicted Fail	Predicted Pass
Actual Fail	92 (TN)	107 (FP)
Actual Pass	39 (FN)	262 (TP)

The results mentioned in the previous paragraph were all evaluated on the development partition of the data (n = 900 data points). In order to obtain an unbiased estimate of our model's performance we performed a final evaluation on the separate test partition (n = 500 data points) using the model trained with the "after final" features. The test set contained around 60% of students who passed and the remaining 40% was students who failed. On the test set, our averaged perceptron achieved an overall accuracy of 70.80% with a precision of 0.7100, recall of 0.87604 and an F1-score of 0.7821. The confusion matrix above illustrates that the model correctly identified 92 failing students and 262 passing students while incorrectly predicting 107 failing students as passing and 39 passing students as failing. Comparing these results to the development set performance we can see all the metrics are extremely similar with the test accuracy being ever so slightly better at 70.80% compared to 70.11% on the development set. This consistency between development and test set performance indicates our model is able to generalize to unseen data and is not being overfitted to the development partition.

In-Depth Error Analysis

The code used to do this error analysis is described above, but we will now go into detail about what was learned by doing that error analysis. The first thing we will discuss is the features that the average perceptron learned. Here are the features:

```
=== Averaged Perceptron Weights by Feature ===

Attendance (%)              : -6233.673986
Extracurricular_Activities  : -55.700000
Midterm_Score               : 597.349064
Final_Score                 : 5958.020999
Assignments_Avg             : -475.011991
Quizzes_Avg                 : -2485.471943
Participation_Score         : 1239.589938
Projects_Score              : 3926.331010
Stress_Level (1-10)         : -519.700000
Sleep_Hours_per_Night       : -580.430005

Bias (b_avg): -99.200000
```

Looking at the features, there are a couple of things that stand out. One, attendance has the highest feature weight, and it is very negative. Additionally, Final_Score and Project_Score are the only other features that are even close to it in weight. This observation gives a lot of insight into the issues with our model. For some reason, it lets attendance dominate over everything else, such that for almost all the other features, their value contributes almost nothing to the final decision. This indicates that the model is overfitting to the training data because having high attendance should make the model more likely to predict that they pass, not fail. If we look at the 10 examples of FN, we can see that they all have really high attendance scores. This identifies the main reason why the model is making a lot of its mistakes.

```
 Attendance (%)
          92.88
          89.28
          98.37
          99.74
          94.61
          83.61
          94.06
          81.75
          91.93
          95.08
```

If we look at the FP, we see a similar trend. Almost all of them have pretty low attendance scores.

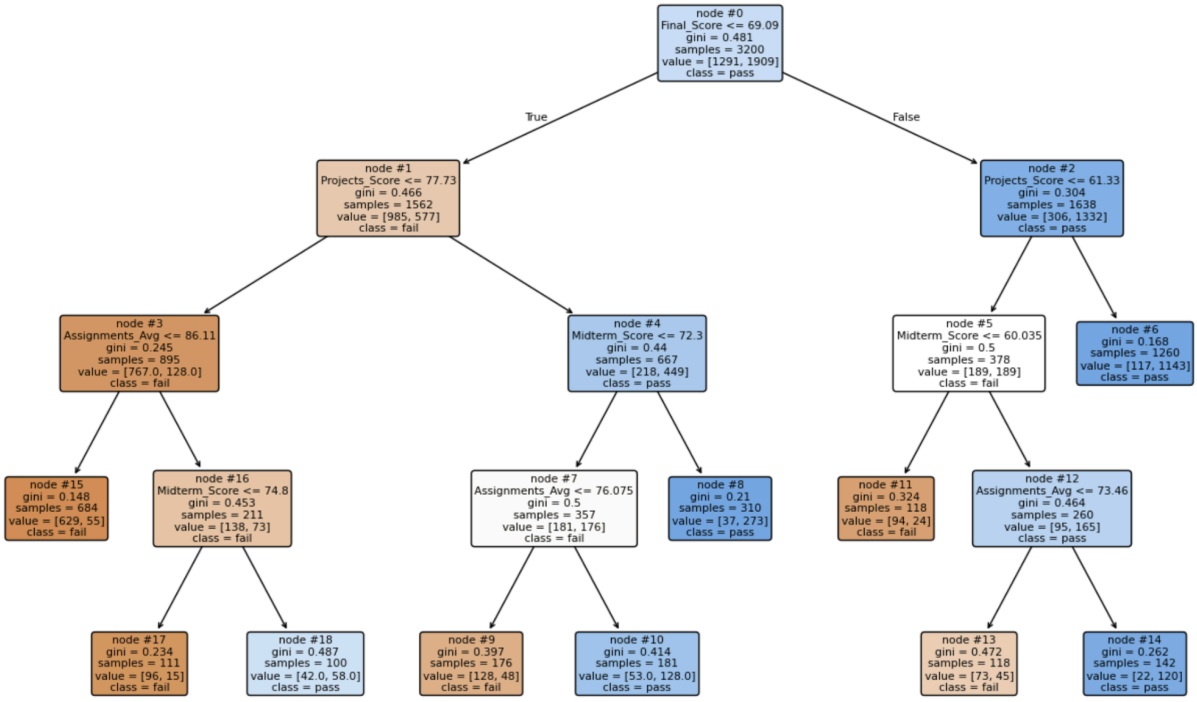
Attendance (%)
92.68
70.32
52.96
68.14
66.87
68.47
57.46
65.60
56.27
60.67
61.57
71.90
54.93
82.05
63.44

This would seem to indicate that the model is not using attendance properly, and when we look at the Pearson correlation scores, we can see that it has a pretty low correlation with the grade.

	Grade	Grade_Corr_Abs
Grade	1.000000	1.000000
Total_Score	0.801708	0.801708
Final_Score	0.481276	0.481276
Projects_Score	0.473152	0.473152
Assignments_Avg	0.265017	0.265017
Midterm_Score	0.251470	0.251470
Quizzes_Avg	0.156222	0.156222
Participation_Score	0.150405	0.150405
Stress_Level (1-10)	0.020894	0.020894
Sleep_Hours_per_Night	0.010893	0.010893
Extracurricular_Activities	-0.010261	0.010261
Internet_Access_at_Home	0.009488	0.009488
Attendance (%)	-0.009264	0.009264
Study_Hours_per_Week	0.007204	0.007204
department index	-0.005741	0.005741
Family_Income_Level	0.004704	0.004704
Gender	0.002685	0.002685
Age	-0.001905	0.001905

Another potential issue we identified is that the features are not regularized. Final_Score, Projects_Score, Attendance, and others are from 0-100. Others, like Stress_Level, are from 1-10, which may lead to the model putting more weight on things like Attendance and Projects_Score, more so than it should. We attempted to fix this by implementing a function called shave columns that gives all the features with an absolute correlation >=.01, excluding columns we wanted to explicitly exclude, such as Total_Score and things like Final_Score for the first 2 models. We also added in the preprocessing a line that divides each column by the max of that column to make sure every feature is between 0 and 1 to try and fix the weights. These seemed like glaring issues, and that fixing them would improve our performance, but in reality, the performance for every model on the dev went down, and when we evaluated the final

model on the test data set, the performance was worse as well. Our best guess as to why this is the case is that we have a very small dataset, so it just happened that making these mistakes allowed a slightly better model in this instance. However, we would expect that, in general, this newer model should perform better given more data. The code for this update can be found in the file called **AttemptedCorrelationColumnShrinking.ipynb**. This file is almost identical to the earlier average perceptron discussed above, with just those minor tweaks to the feature selection and data preprocessing. In addition to looking at the patterns in the data and the feature weights, we used a decision tree as a helpful visualization to further analyze the mistakes our model was making. As discussed in the coding section above, we fit three decision trees to match the 3 different time periods we have for our models and looked at the performance of the decision tree. The decision tree performed better than our model in every case, which was helpful because that meant we might be able to apply some of the logic learned by this decision tree to our model. The decision tree we focused the most on is the one that corresponded to the model trained after the final, as that was our best-performing model, so we wanted to try and improve that one. We fed each of our example mistakes into the decision tree and looked at the path through the tree it took.



Looking at the decision tree, it appears that it is making decisions on a non-linear boundary. This indicates to us that it is possible that a linear classifier simply won't be able to properly classify this problem. If we had more time, we would have further explored this issue by attempting a voting perceptron. We originally tried an FFNN because we assumed the issue was non-linear, but the performance was abysmal, so we tried switching to a simpler model. That is why we would try a voting perceptron given more time, as it is still simpler than an FFNN, but it can have a non-linear decision boundary.