Stephen Hornyak (1075791723)
Victor Hui (3051128112)
Joshua Williams (8139286920)
12/9/2021

EE 454 Project Phase 3B: Writeup of Methods and Results

Explanation of terms

Classification Model: Given a feature Vector X and a qualitative response Y, the task of a classification model is to build a function C(X) that takes as input the gesture vector X and predict its value for Y

No Free Lunch Theorem: The idea that in Machine Learning, no singular prediction model will achieve the best results across all datasets.

Curse of Dimensionality: A set of phenomena that occurs when a Machine Learning model attempts to analyse data in High dimensional space (more than 3 features). For example, for the K nearest Neighbors algorithm, finding euclidean distance between data points in a high dimensional dataset becomes computationally expensive and the distance between data points will appear further than they actually are.

Class imbalance: A dataset is imbalanced if members of a certain class are much rarer than others. However, the lack of observation does not always imply that the rarer class is less relavent.

Overview

Task scheduling with maximum gain is a NP-Hard problem. Knowing that both Machine Learning and Greedy Algorithms are valid approaches to obtaining close to optimal solutions, we decided to create our own Machine Learning task scheduling algorithm. Designed for a real time operating system, we implemented a KNN classification model that would classify incoming tasks into one of four job types, and approximated their burst time based on the formula (JobType + 1) * 300. Then, we combine the Classification model with a greedy algorithm that would optimize turnaround time. The greedy algorithm would optimize task scheduling based on the predicted burst time provided by the classification model.

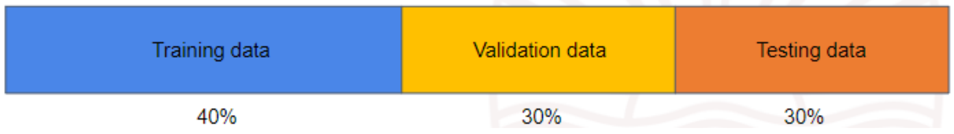
The machine learning classification model experimentation can be found in "**phase3_part2_classification.ipynb"**

The implementation of our machine learning model into our scheduling algorithm code can be found in "**phase3_part2_implementation.ipynb"**

Dataset

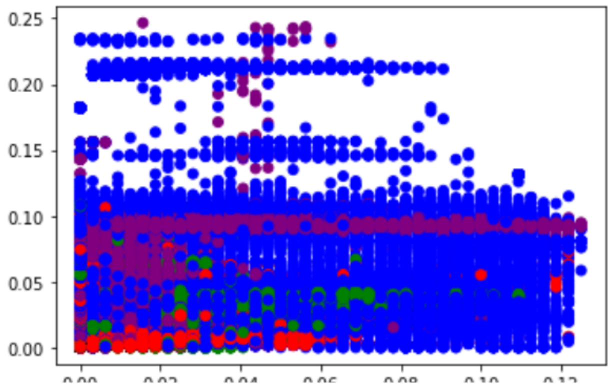
```
Time            ParentID        TaskID          JobType         NrmlTaskCores   NrmlTaskMem
90000           757745334       1488529826      0               0               0.0311296
90000           975992247       1488529821      0               0               0
90000           1468458091      1488529832      1               0.021875        0.00235309
90000           1460281235      1488529840      0               0               0
90000           1164728954      1488529835      0               0.003125        0.0016384
90000           1288997448      1488529848      0               0.003125        0.0049152
90000           1488529845      1488529847      1               0.003125        0.000719232
90000           1263655469      1488529844      2               0 0
```

The google cluster dataset consists of 4 job types (from 0 - 3), and 2 features (NrmlTaskCores and NrmlTaskMem). Since there is an abundance of datapoints (3,535,029 data points), we decided to split the dataset into three subsets. We created a 4:3:3 split between Training data, Validation data, and Testing data. In order to preserve a basic level of class balance within the datasets, we split the data in the following way:

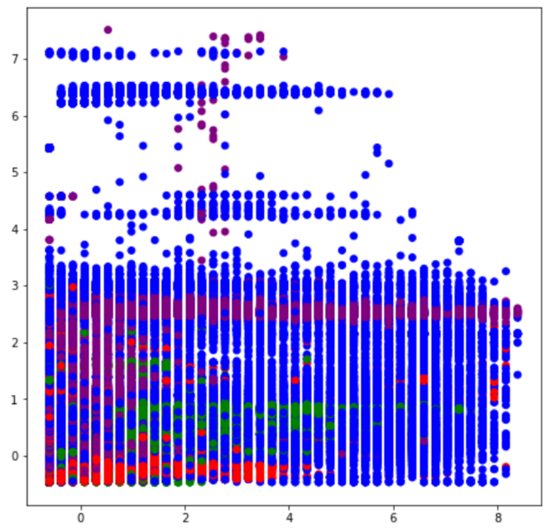


Denote “Job Type 0” as J0, “Job Type 1” as J1, etc. Select the first 40% of J0, 40% of J1, 40% of J2, and 40% of J3. Place the selected data points in the training set. Split the raminder evenly into the validation and testing set.
Note: this does not mean the classes are balanced in the dataset, which we will explore later.

Before we began testing our training data on Multiclass classification models, we first plotted a pairwise interaction plot between our two features to see if any classification model should be considered immediately considered. As we can see from the figure to the right (x axis:NrmlTaskCores, y axis:NrmlTaskMem) (blue: JobType0, green: JobType1, red: JobType 2, purple: Jobtype3), the data points are too densely packed together, which makes it difficult to infer correlation.



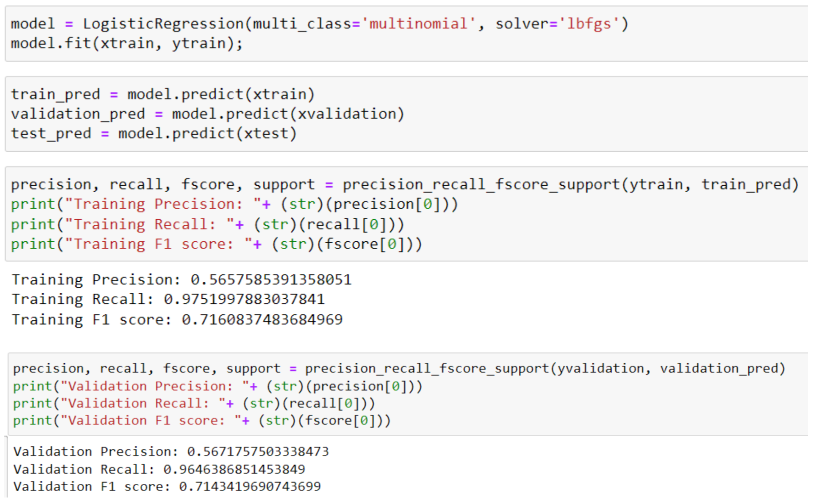
Therefore, to remedy this situation, we normalized the feature set so that there could be more separation between the data points. From the normalized pairwise interaction plot, we can immediately notice that there is no clear linear separation boundary, nor are there any apparent clusters. This shows that classification models that rely solely on linear separation boundaries such as Multinomial Logistic Regression should not be considered. To illustrate this point, we fitted a Multinomial Logistic Regression model to the training set.



Classification Models for Multiclass Classification

In accordance with the No Free Lunch Theorem, we tested four Classification methods, to see which one would perform better.

Multinomial Logistic Regression



```
model = LogisticRegression(multi_class='multinomial', solver='lbfgs')
model.fit(xtrain, ytrain);
```

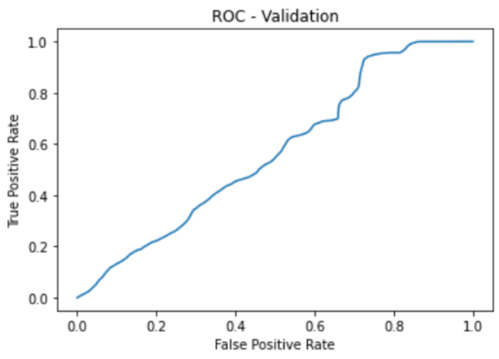
```
train_pred = model.predict(xtrain)
validation_pred = model.predict(xvalidation)
test_pred = model.predict(xtest)
```

```
precision, recall, fscore, support = precision_recall_fscore_support(ytrain, train_pred)
print("Training Precision: "+ (str)(precision[0]))
print("Training Recall: "+ (str)(recall[0]))
print("Training F1 score: "+ (str)(fscore[0]))
```

```
Training Precision: 0.5657585391358051
Training Recall: 0.9751997883037841
Training F1 score: 0.7160837483684969
```

```
precision, recall, fscore, support = precision_recall_fscore_support(yvalidation, validation_pred)
print("Validation Precision: "+ (str)(precision[0]))
print("Validation Recall: "+ (str)(recall[0]))
print("Validation F1 score: "+ (str)(fscore[0]))
```

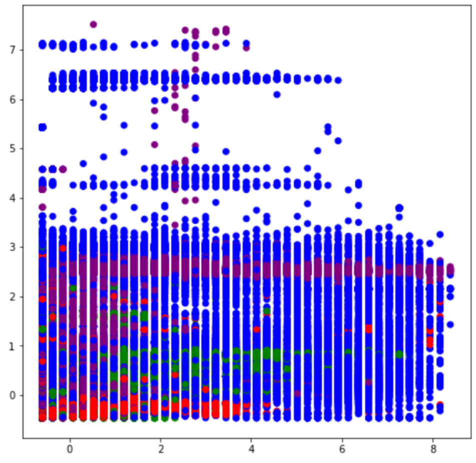
```
Validation Precision: 0.5671757503338473
Validation Recall: 0.9646386851453849
Validation F1 score: 0.7143419690743699
```

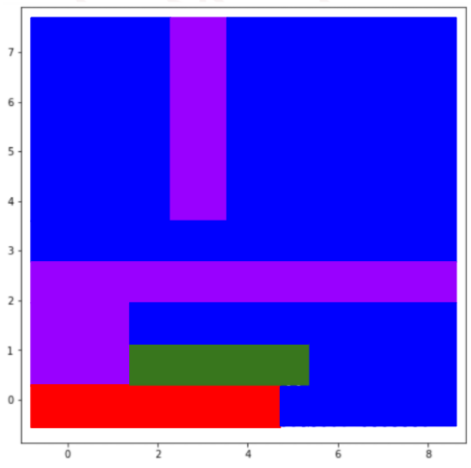


As we can see for the Receiver Operating Characteristic (ROC) curve, as expected, multinomial logistic regression shows no discrimination capactiy to distinguish between the different classes.

Decision Tree Classification

Returning to the pairwise interaction plot, a classification method that immediately stood out to us was the decision tree classification. By simply eye-balling, we could easily stratify the predictor space into a number of simple regions. And by segmenting the predictor space, a decision tree can be summarised.



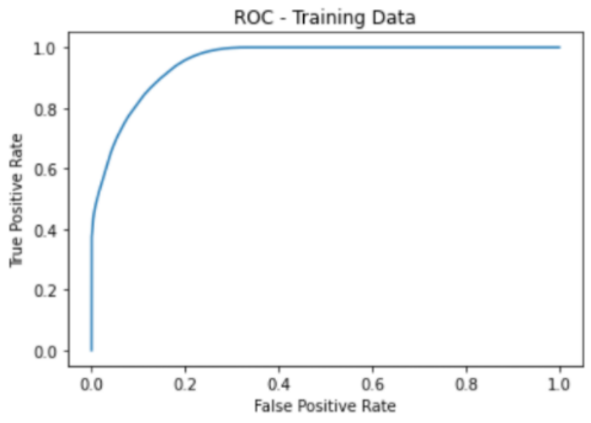


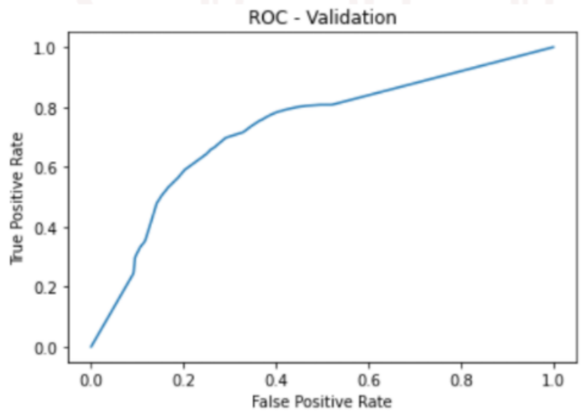
```
precision, recall, fscore, support = precision_recall_fscore_support(ytrain, train_pred)
print("Training Precision: "+ (str)(precision[0]))
print("Training Recall: "+ (str)(recall[0]))
print("Training F1 score: "+ (str)(fscore[0]))
```

```
Training Precision: 0.7279415397689655
Training Recall: 0.8104104555605672
Training F1 score: 0.7669654755955356
```

```
precision, recall, fscore, support = precision_recall_fscore_support(yvalidation, validation_pred)
print("Validation Precision: "+ (str)(precision[0]))
print("Validation Recall: "+ (str)(recall[0]))
print("Validation F1 score: "+ (str)(fscore[0]))
```

```
Validation Precision: 0.7200754546423223
Validation Recall: 0.619927862030589
Validation F1 score: 0.6662593018134777
```





Results from the ROC shows that the Decision Tree Classifier outperformed the Multinomial Logistic Regression Model but shows that validation ROC is still undesirable. This could be due to Decision Tree Classifier’s tendency to overgeneralize, leading to inferior predictive accuracy compared to other classification approaches.

K Nearest Neighbors (K = 3)

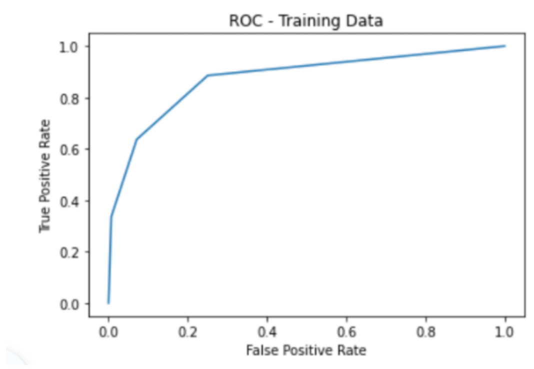
For our next approach, we decided to use K Nearest Neighbors, luckily, since we only have 2 features in our dataset, we did not need to worry about the curse of dimensionality. First, we fitted a K Nearest Neighbors classifier with K = 3 to the training data.

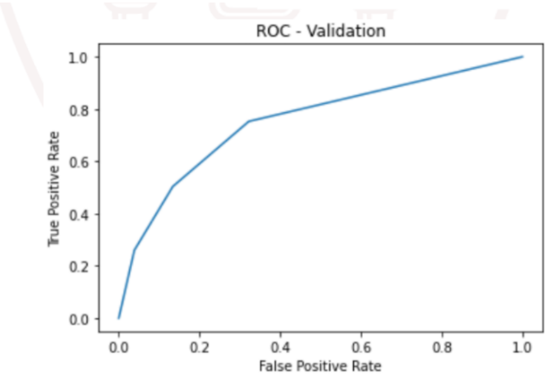
```
precision, recall, fscore, support = precision_recall_fscore_support(ytrain, train_pred)
print("Training Precision: "+ (str)(precision[0]))
print("Training Recall: "+ (str)(recall[0]))
print("Training F1 score: "+ (str)(fscore[0]))
```

```
Training Precision: 0.7566481543628395
Training Recall: 0.9068748346123313
Training F1 score: 0.824978283312161
```

```
precision, recall, fscore, support = precision_recall_fscore_support(yvalidation, validation_pred)
print("Validation Precision: "+ (str)(precision[0]))
print("Validation Recall: "+ (str)(recall[0]))
print("Validation F1 score: "+ (str)(fscore[0]))
```

```
Validation Precision: 0.6757883716773485
Validation Recall: 0.80038972245632
Validation F1 score: 0.7328303670072797
```





The results show that with K = 3, performance of K Nearest Neighbors is comparable to Decision Tree Classifier, with a slight improvement on Validation data.

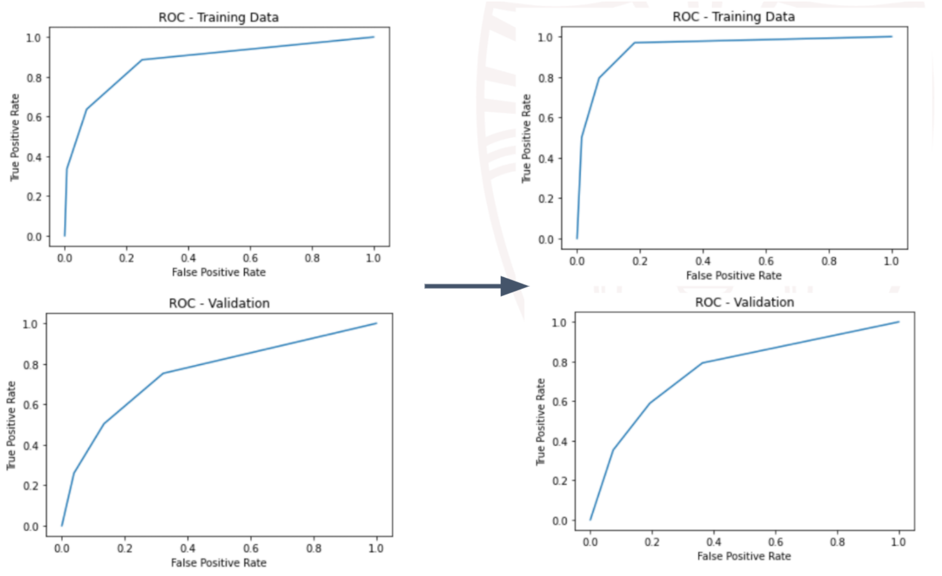
Resampling

Looking to improve K Nearest Neighbors, we return to focusing on the dataset. As we mentioned above, classes in the training set are imbalanced as 0.4 J0 is significantly larger than the other classes. We can attempt to resample the data by either downsampling or upsampling. Here, we downsampled J0, J1, and J3 to the size of J2/2 (which has the lowest number of samples). Since the size of the dataset is so large to begin with, J2 itself has also been downsized to J2/2 to avoid overfitting.

```
samplesize = (int)(len(Jobtype2)/2) #job 2 has the lowest number of samples
trainingset_J0 = train[train["JobType"]==0]
new_trainingset_J0 = resample(trainingset_J0, replace = True, n_samples = samplesize, random_state=42)
trainingset_J1 = train[train["JobType"]==1]
new_trainingset_J1 = resample(trainingset_J1, replace = True, n_samples = samplesize, random_state=42)
trainingset_J2 = train[train["JobType"]==2]
new_trainingset_J2 = resample(trainingset_J2, replace = True, n_samples = samplesize, random_state=42)
trainingset_J3 = train[train["JobType"]==3]
new_trainingset_J3 = resample(trainingset_J3, replace = True, n_samples = samplesize, random_state=42)
```

```
new_train = new_trainingset_J0.append(new_trainingset_J1)
new_train = new_train.append(new_trainingset_J2)
new_train = new_train.append(new_trainingset_J3)
```

Performance of KNN (K = 3) after downsampling



From the figure above, we can see that after downsampling, both training and validation data yield better results compared to the original dataset.

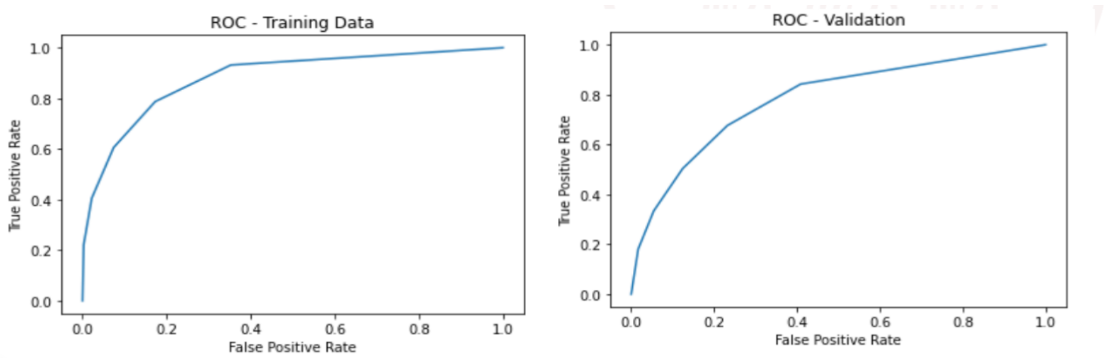
K Nearest Neighbors (K = 5)

```
precision, recall, fscore, support = precision_recall_fscore_support(ytrain, train_pred)
print("Training Precision: "+ (str)(precision[0]))
print("Training Recall: "+ (str)(recall[0]))
print("Training F1 score: "+ (str)(fscore[0]))
```

```
Training Precision: 0.7477928685227403
Training Recall: 0.8992431860280498
Training F1 score: 0.8165548528478883
```

```
precision, recall, fscore, support = precision_recall_fscore_support(yvalidation, validation_pred)
print("Validation Precision: "+ (str)(precision[0]))
print("Validation Recall: "+ (str)(recall[0]))
print("Validation F1 score: "+ (str)(fscore[0]))
```

```
Validation Precision: 0.6818534021784296
Validation Recall: 0.8072527087021214
Validation F1 score: 0.739273046862739
```

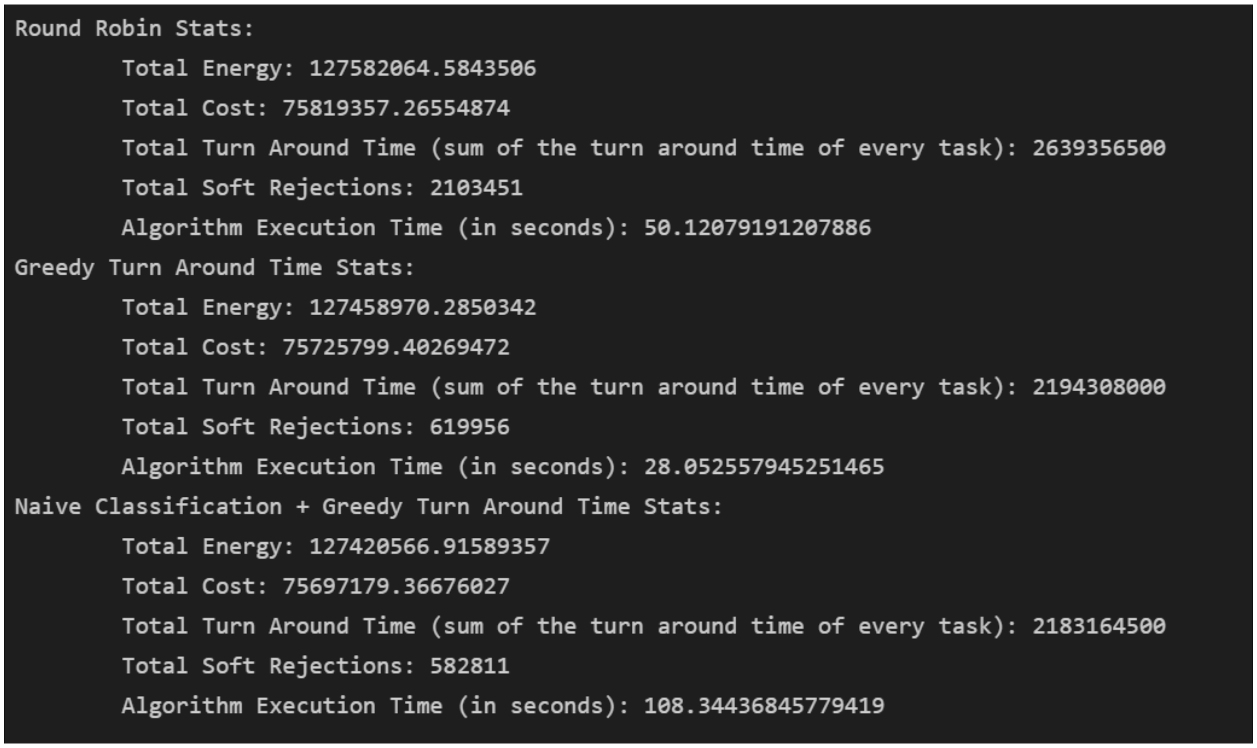


Finally, after performing 3 folds cross validation with different K numbers, we decided to proceed with K=5 Nearest Neighbors because of its superior validation performance compared to the other classification models.

Combining the classification model to Greedy Algorithm
After deciding on using K = 5 Nearest Neighbors as our classification metric, we moved on to combine our classification model with our Greedy Algorithm for turnaround time. In order to combine the models, we needed to make a few changes to the code. First, we modified the burst time calculation from (TaskID % 10 + 1) * 300 to (JobType + 1) * 300. Second, we changed the resource allocation for the VMs. We chose to use soft rejection because soft rejection is what is needed to put a task into the waiting state. On the other hand, hard rejection discards the task instead of putting it into the waiting state. Since we wanted to measure turnaround time, which is equal to the execution time plus the waiting time, we selected soft rejections. To best compare our ML algorithm with RR and Greedy, we created two setups, one with a small number of soft rejections and another with a large number of soft rejections. Through trial and error, we discovered CPU and memory unit settings for the VMs that achieved the desired soft rejections. In order to utilize the predicted burst time as a metric for the greedy algorithm, we also had to modify the turnaround time greedy algorithm to give priority based on predicted burst time remaining instead of actual burst time remaining.

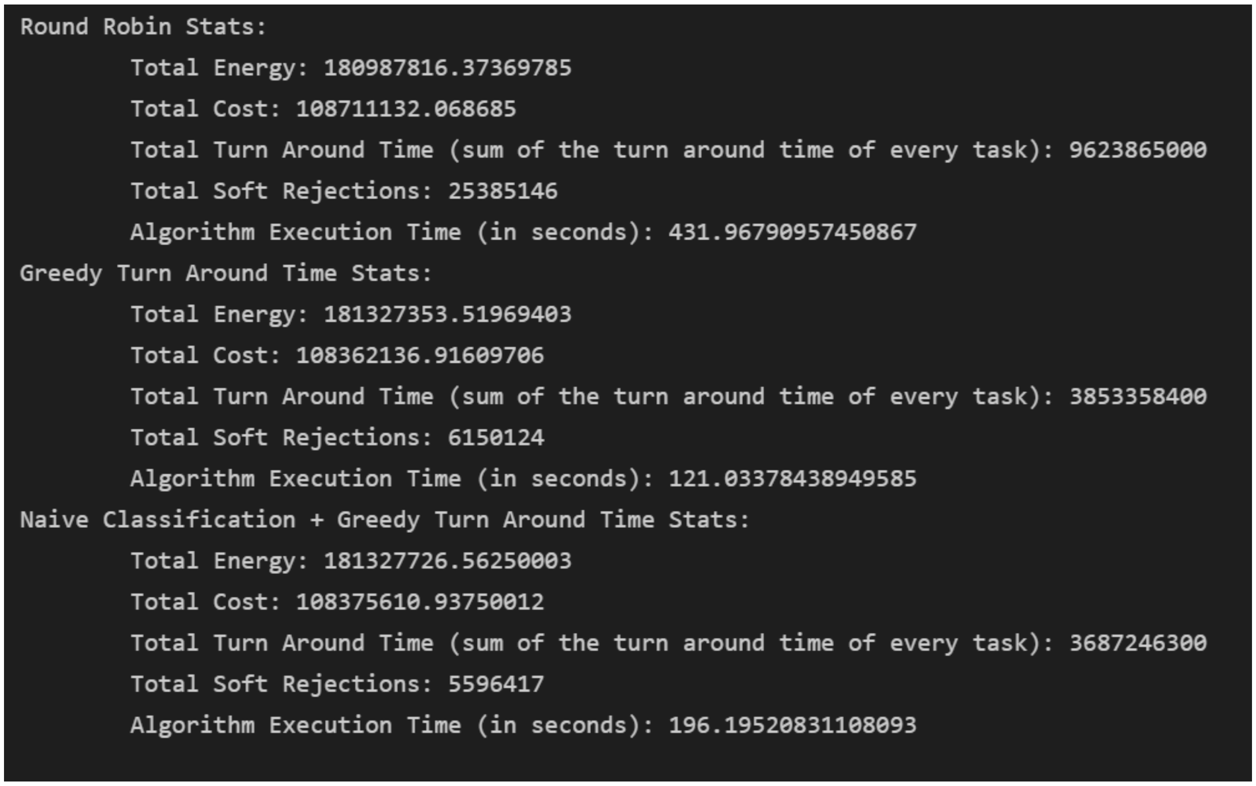
Results:

Setup 1: Small Number of Soft Rejections (CPU = 16, MEM = 22)



```
Round Robin Stats:
        Total Energy: 127582064.5843506
        Total Cost: 75819357.26554874
        Total Turn Around Time (sum of the turn around time of every task): 2639356500
        Total Soft Rejections: 2103451
        Algorithm Execution Time (in seconds): 50.12079191207886
Greedy Turn Around Time Stats:
        Total Energy: 127458970.2850342
        Total Cost: 75725799.40269472
        Total Turn Around Time (sum of the turn around time of every task): 2194308000
        Total Soft Rejections: 619956
        Algorithm Execution Time (in seconds): 28.052557945251465
Naive Classification + Greedy Turn Around Time Stats:
        Total Energy: 127420566.91589357
        Total Cost: 75697179.36676027
        Total Turn Around Time (sum of the turn around time of every task): 2183164500
        Total Soft Rejections: 582811
        Algorithm Execution Time (in seconds): 108.34436845779419
```

Setup 2: Large Number of Soft Rejections (CPU = 12, MEM = 18)



```
Round Robin Stats:
        Total Energy: 180987816.37369785
        Total Cost: 108711132.068685
        Total Turn Around Time (sum of the turn around time of every task): 9623865000
        Total Soft Rejections: 25385146
        Algorithm Execution Time (in seconds): 431.96790957450867
Greedy Turn Around Time Stats:
        Total Energy: 181327353.51969403
        Total Cost: 108362136.91609706
        Total Turn Around Time (sum of the turn around time of every task): 3853358400
        Total Soft Rejections: 6150124
        Algorithm Execution Time (in seconds): 121.03378438949585
Naive Classification + Greedy Turn Around Time Stats:
        Total Energy: 181327726.56250003
        Total Cost: 108375610.93750012
        Total Turn Around Time (sum of the turn around time of every task): 3687246300
        Total Soft Rejections: 5596417
        Algorithm Execution Time (in seconds): 196.19520831108093
```

Summary: Key Results of the ML Algorithm and Improvement over RR and Pure Greedy

	Setup 1	Setup 2
CPU Units	16	12
VM Memory Units	22	18
Total Turnaround Time	2183164500	3687246300
Total Soft Rejections	582811	5596417
Execution Time	107.001	188.721
TTT Improvement over RR	17.28%	61.69%
Soft Rejection Improvement over RR	72.29%	77.95%
TTT Improvement over Greedy	0.51%	4.31%
Soft Rejection Improvement over Greedy	5.99%	9.00%

The above figures show our results and comparison between Round Robin, Greedy Turn Around Time, and Naive Classification + Greedy Turn Around Time. Because the main focuses of our algorithms were soft rejections, total turnaround time, and execution time, only these metrics will be discussed and the others will be ignored. Since total turnaround time and number of soft rejections are directly related (a higher number of soft rejections leads to a higher total turnaround time), only the number of soft rejections will be compared.

The Machine learning algorithm took longer to execute than both RR and Greedy in the first setup. This is due to the time it takes to apply the KNN classification. In setup 2, ML executed slower than pure Greedy but faster than RR. This is due to the large number of soft rejections that RR had, causing it to loop through the VMs looking for availability before soft rejecting the task and requiring a lot of time to execute. Since ML had less rejections, it was able to execute faster despite the time that it took to classify the inputs. However, ML is still slower in execution in setup 2 than pure Greedy because they have a very similar number of soft rejections and the ML classification adds time.

For soft rejections, our ML algorithm substantially decreased the number of rejections when compared to RR in both setups. As discussed in class, giving priority to the shortest remaining time first provides the optimal solution to minimizing turnaround time. The difference between Greedy and RR can be clearly seen in this regard as the Greedy algorithm significantly cuts down on the number of soft rejections. This is due to the nature of how the task queue is implemented in RR. The task queue is formed by placing new tasks for the time quantum at the front of the queue, while tasks that ran in the previous time quantum are placed in the back of the task queue. Typically, tasks at the front of the task queue are successfully assigned while tasks at the end of the task queue have a higher probability of being rejected. This is because the VMs fill up as the task queue is emptied, so they have less and less resources left as the queue nears the back. Since tasks at the back of the RR task queue are the tasks that have already had some burst time, they are more likely to be soft rejected and their wait time will increase. Since there is no priority, the short tasks will not be finished first, so they can be rejected even while the long tasks have not completed their full burst time. Overall, this leads to more rejections than the Greedy algorithm. The Greedy algorithm's priority leads to an optimal reduction in the size of the task queue at every iteration, so tasks are less likely to be soft rejected. Surprisingly, our ML task scheduling algorithm actually outperformed pure Greedy in soft rejections and Turnaround time. We suspect that, by luck, some tasks being misclassified and thus executing first actually benefited the turnaround time.

Future work

The next step for this project would be to try implementing a DRL approach, one approach to maximize long-term task satisfaction [1] stood out to us, the problem would be formulated as a partially observable MDP, and a deep recurrent Q-network would be applied to approximate the optimal value function.

References:
[1] Sheng, S.; Chen, P.; Chen, Z.; Wu, L.; Yao, Y. Deep Reinforcement Learning-Based Task Scheduling in IoT Edge Computing. Sensors 2021, 21, 1666. https://doi.org/10.3390/s21051666