ABSTRACT - The use of high performance computing, or HPC, clusters is growing among scientific and engineering communities. In general, an HPC cluster is a set of interconnected computers such that their resources can be aggregated for improved application performance. One common issue with maintaining HPC clusters is how to manage and schedule the individual jobs they are running at any given time; this is especially true when power consumption is a concern. This paper examines this problem using machine learning classification techniques to predict running applications on an HPC cluster given performance data and clustering techniques to visualize computing resource usage of various HPC applications. The results of these models are compared and summarized to justify whether this is a meaningful application of data mining.

I. INTRODUCTION

Performance is an important aspect of computer systems; especially in high performance computing, or HPC, clusters. The use of HPC clusters for advanced computation is growing among scientific and engineering communities. An HPC cluster is a set of interconnected computers such that their resources can be aggregated for improved application performance. Any software performance increase via an HPC cluster may be negated if the cluster is poorly managed. Bad performance not only affects the user experience but it also costs companies significant financial resources because it is generalized as higher resource usage. HPC clusters or critical software can be shut down because they are not maintained for performance. Therefore, there has been significant empirical research work focusing on the performance of HPC clusters and how to best manage them. One common issue with maintaining HPC clusters is how to schedule individual jobs that a cluster is running at any given time; this is especially true when power consumption of various jobs is a concern. This paper offers a comparative analysis between machine learning classification and clustering techniques to solve this problem. Specifically, we use a dataset provided by the National Renewable Energy Laboratory (NREL) with power consumption and performance data for HPC jobs [1] and compare the usefulness of k-nearest neighbors, naive bayes, multilayer perceptron, random forest, k-means, and hierarchical clustering using this dataset to address HPC performance issues.

II. RELATED WORK

Despite the relative novelty of machine learning and HPC systems, the need to appropriately manage system resources is nothing new. However, the combination of these two concepts is non-trivial and some light research in this area has been previously done. One study by Ozan Tuncer et al. [2] use machine learning classification models to detect “anomalous” compute nodes within an HPC cluster. In this case, the training and testing dataset was historical performance data. The authors used decision tree, random forest, and AdaBoost algorithms to test their theory and found random forest to perform the best [2]. Based on a 5-fold cross validation of their random forest model, the authors were able to achieve a 98.2% success rate when detecting anomalous compute nodes. This research presents a strong argument towards using random forest classification for HPC performance analysis. However, using random forest is not the only option. Another study by Mark Endrei et al. [3] used an artificial neural network as the prediction interface for a similar problem. The goal of this paper was to predict the relative trade-off between energy efficiency and performance loss in HPC clusters. Their artificial neural network used three hidden layers and 2,000 training steps. Evaluation and application of this model provided a 45% increase in HPC energy efficiency and a 10% decrease in performance with a 90% accuracy rate. This study provides support for the idea that machine learning can be actively used with HPC systems to decrease power consumption.

III. METHODOLOGY

In order to provide a wide scope for comparative analysis, this paper looks at nine different data mining techniques in an attempt to offer insight into how HPC cluster performance can be improved. This analysis comes in the form of classification and clustering techniques. The dataset used for this problem is from NREL’s HPC Energy Research division [1]. The dataset is publicly available and includes 10,000 samples of power consumption and performance data randomly selected from an active HPC cluster. The original dataset contains 27 features however our preprocessing method chose 8 features and an additional column as the class label. The feature reduction was chosen because only 8 features are relevant to HPC job performance and most others were redundant or nominal (id, user_name, group_name, etc). The chosen features include: power, start_time, end_time, nodes_used, processors_used, wallclock_used, cpu_used, mem_used, and app_name. For the purposes of classification, the app_name column is chosen was the class label. To ensure sufficient data quality, any sample with unknown or empty values was removed from the dataset. Additionally, the start_time and end_time features were combined such that we considered only a run_time value for each job. This provides us with a dataset that consists of 9,792 samples of HPC job performance data with 25 unique class labels.

i. Classification

The goal of the classification methodologies in this paper is to predict the app_name label for each HPC job sample. The app_name label is a string that signifies the name of an HPC software program. Based on the dataset features previously mentioned, the prediction comes from power consumption and performance information for each program. This classification problem can be characterized as a multiclass problem because there are 25 unique class labels. The comparative analysis between classification methods was done using k-nearest neighbors, Gaussian naive bayes, multilayer perceptron, and random forest. For each classifier to enumerate each unique class label, the string values were encoded to integer values for classification which was simply based on list indices. Evaluation of each model was completed using a combination of accuracy and F1 score measurements. Due to class imbalance issues within the dataset, the random forest method was enriched using a boosting algorithm from Microsoft called LightGBM [5]. LightGBM is a relatively new approach to gradient boosting for imbalanced classes based on decision tree models. An explanation of the implementation of these classification methods is discussed in section IV.

ii. Centroid Clustering

We sought to determine if k-means clustering was a viable mining technique for HPC performance data. How accurate is k-means with performance data of HPC jobs? Is the data too job-dependent for us to cluster confidently? Also, regardless of k-means viability with this dataset: what dimension count and specific features will render the best results? The k-means code was tested with dimensions of 7, 5 and 2, excluding app_name. 7 dimensions were initially tested in order to determine what the baseline centroid results were when all meaningful continuous data values were considered from the data set. We then tested 5 dimensions, excluding memory and run_time. This was because a majority of the 'mem_used' values were 0, which led us to believe that there was some faulty data evident within that data column. We also excluded run_time, because we noticed that run_time and wallclock_used was the same data, so to avoid valuing that aspect of dimension twice as much as the other dimensions, we excluded the run_time. Finally, we tested our data set with a myriad of two-dimension combinations, determining one particular combination rendered the best results, discussed further in implementation. We wanted to try and narrow our scope to two features that seem dependent upon each other, as to see if we could draw any meaningful data from them. This was because we did not want an irrelevant or overvalued dimension to weaken our Sum of Squared Error value, resulting in any bloated influence to the location of the clusters.

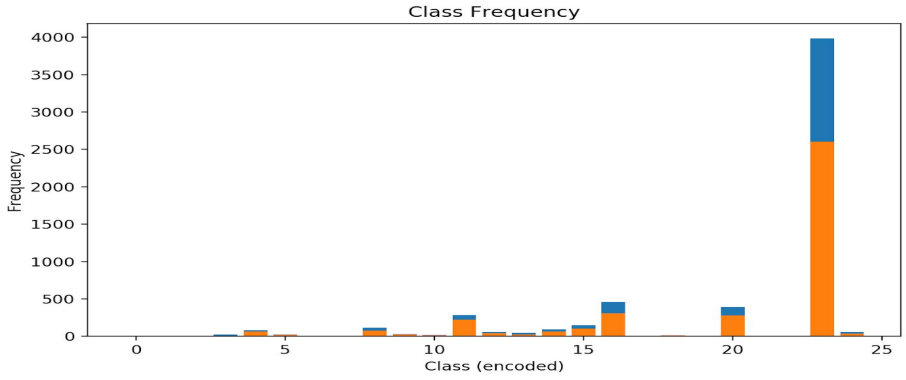
iii. Hierarchical Clustering

We implemented hierarchical clustering in order to create clusters using the dataset, such that each cluster would correspond to each app_name. Agglomerative clustering was implemented because this is the most common type of hierarchical clustering. All 7 dimensions previously mentioned were used for clustering. The implementation includes choosing group average, minimum (single link) and max (complete link) as the proximity measures.

IV. IMPLEMENTATION

i. Classification

The implementation for each classification algorithm was completed with Python 3 and the machine learning algorithms came from the Python scikit-learn library and the Python LightGBM implementation. At this point, assume the NREL dataset has already been imported, preprocessed, and encoded appropriately. The dataset contains 9,792 usable samples and these were split into training and testing sets. Using random selection, 40% of the data was selected as the test set and 60% was selected as the training set. The graph below shows the class frequency distribution among the training (blue) and testing (orange) sets.



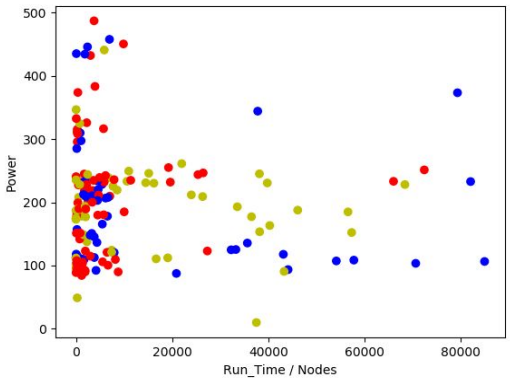
As can be seen, this dataset suffers from a class imbalance problem. The effect of this problem on classification changes based on the algorithm being used. Initially, the class imbalance problem was not addressed because the first task is to find the best classification algorithm for this dataset among KNN, naive bayes, multilayer perceptron, and random forest. Once the concrete algorithm is chosen, the class imbalance is addressed in an attempt to further increase the accuracy of the model. Each model was evaluated using accuracy, F1 score, and the number of classes that were never predicted. Due to the low frequency of some of the classes, the models made assumptions such that they were never predicted in the test set despite their presence in the training set. It was assumed that random forest would perform the best due to the previous research mentioned earlier, but we also needed to select others for comparison. The parameters used for each algorithm are shown in the table below.

Algorithm	**Parameters**
K-Nearest Neighbors	k=5, uniform class weights, kd tree
Naive Bayes	Gaussian classifier
Multilayer Perceptron	200 max iterations, relu activation, lbfgs solver
Random Forest	100 trees, GINI criterion, balanced/unbalanced class weights

ii. Centroid Clustering

Similar to the classification implementation, the k-means techniques were written in Python 3 with the help of the Python scikit-learn library. Rather than sorting all 25 languages into clusters, we decided to narrow it down to 3 languages that had a similar sample count. We sampled the first 1000 usable samples, excluding samples where the 'app_name' = "Unknown" and the 'cpu_used' = 0. We created two arrays of linked lists, one of which ignored the app_name feature for each sample and one that did not. This is what allowed us to check the accuracy of our clusters after we ran the k-means algorithm. We then converted these arrays into dataframes, and utilized the built in kmeans() function within the scikit-learn library to determine our centroids.

Our dataset contained 176 total samples; 73 python, 50 gaussian, and 53 mono app_names. After trial and error we determined that our best results for 2 dimensions were when power and (wallclock_used/nodes) were considered. This rendered the best results due to the fact that power and execution duration are two properties that correlate to the characteristics of a language. For example, a language typically executed in parallel would have a high power, and a low duration of execution, where as a sequential program would have the opposite. Thus, we created a 2-dimension graph for all of our results, in order to compare each dimension size on the same scale. Below is the graph, with each color sorted by its proper app_name.



Run_Time/Nodes refers to the execution duration/node used, not the run_time feature

Immediately we noticed an issue at hand. Over 70% of the samples had an (execution duration/nodes) < 20000, and less than 30% had 2000 < (execution duration/nodes) < 85000. Regardless, we wanted to continue with our analysis, and see if we could deduce any valuable clusters, prior to finding a way of normalizing the dataset.

iii. Hierarchical Clustering

Hierarchical clustering was implemented from scratch, without the use of sklearn library. As discussed earlier, the implementation includes choosing group average, single link and complete link as proximity measures.

App_name labels were separated from the dataset in order to calculate the accuracy of the clusters returned by the clustering algorithm. The remaining data contains the sample points to run the clustering algorithm on. Number of clusters are determined by the number of unique app_names in the dataset. When the algorithm is left with the required number of clusters, it stops.

The original data was preprocessed. First, the samples where values of any dimension were 0 were removed since these samples would incorrectly represent the app (It does not make sense that 0 memory or cpu or nodes were used when the apps were run). Second, the app_names that had a frequency of less than 10 in the dataset were removed, where many of these removed samples had frequency of just 1. The reason to do this filtering is that we have samples that can be clustered properly by the algorithm rather than having 1 or very few points in a cluster.

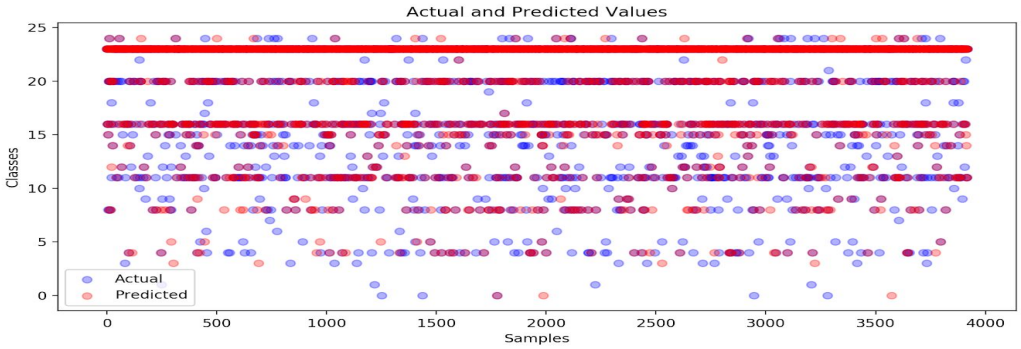
V. EXPERIMENTAL RESULTS

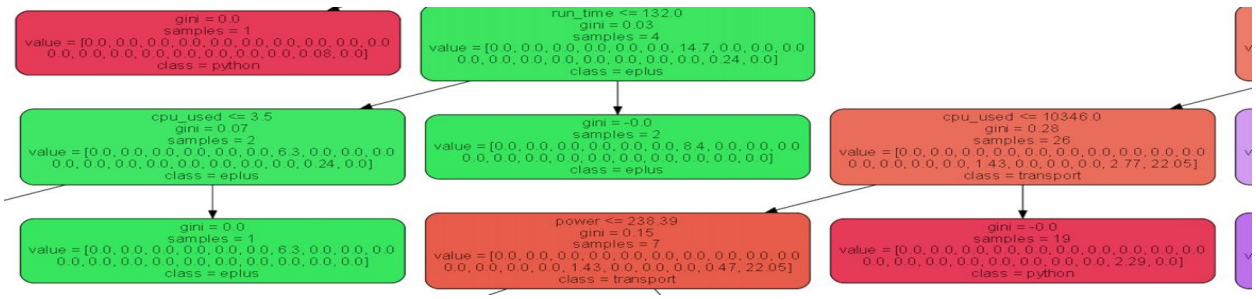
i. Classification

Classification evaluation is summarized as a comparison between each tested algorithm and the measurements showing their accuracy. The table below depicts the measurements.

Algorithm	**Accuracy**	**F1 Score**	**Classes Not Predicted**
K-Nearest Neighbor	0.7146	0.6965	4
Naive Bayes	0.0853	0.1530	8
Multilayer Perceptron	0.4297	0.4864	10
Random Forest	0.8121	0.7973	6

As shown, the random forest classifier performed the best across accuracy and F1 score measurements. However, it did still have 6 of 25 classes that it never predicted. The worst performing method by far was naive bayes; this is likely due to the fact that the dataset is not following a Gaussian distribution or any distribution that is suitable for naive bayes. The following two images visualize the random forest classifier.





The scatter plot shows the accuracy via actual and predicted values. As expected, the class with the highest frequency has the highest predicted success rate and the classes with lower frequency have few or zero predictions. The tree chart is an example of one of the trees in the random forest model. The accuracy and F1 score measure are from a random forest model using balanced class weights based on inverse frequencies [4]. Since the random forest model initially performed the best, 5-fold cross validation was also performed with accuracies of [0.8163 0.8215 0.8237 0.8091 0.8327]; these results appropriately reflect the high quality of the random forest model.

Another method that was used to improve random forest accuracy with the class imbalance issue was LightGBM [5]. LightGBM is a Microsoft project that provides an efficient framework for gradient boosting with decision tree algorithms which hopefully improves model accuracy [6]. However, as the below table shows, the accuracy significantly decreased but there was an improvement where all classes were predicted. The lower accuracy is most likely caused by the lower frequency classes being used in predictions, but not well.

Algorithm	**Accuracy**	**F1 Score**	**Classes Not Predicted**
LightGBM Random Forest	0.6847	0.7197	0

ii. Centroid Clustering

Clustering results for datasets of 7, 5, and 2 dimensions are depicted in the tables below.

7 dimensions

Cluster	**samples**	**cpu_used**	**mem_used**	**nodes**	**power**	**processors**	**run_time**	**wallclock**
0	18	25,632.83	355,307.38	1.08	190.12	14.58	42,003.81	12,430.65
1	149	44,120.78	19,573,418.7	1.11	223.05	24.89	154,418	20,018
2	9	47,574.16	4,718,592	1.28	241.81	28.89	8,080	8,080

5 dimensions

Cluster	**samples**	**cpu_used**	**nodes**	**power**	**processors**	**wallclock**
0	50	63,994.32	1.08	211.17	18.66	8,506.6
1	105	10,343.66	1.09	192.94	14.79	4,548.33
2	21	37,473.09	1.23	184.32	20.52	60,707.85

2 dimensions

Cluster	**samples**	**power**	**wallclock/nodes**
0	140	201.67	3,238.64
1	25	174.95	32,378.53
2	11	189.23	68,106.73

Prior to any statistics on the accuracy of these clusters, the 5 dimension dataset separates the quantity of samples minimally better than the other two. They all still have one cluster with over double the samples as compared to the other clusters, recognizing the fact that there is some factor within our dimensions that is impacting the cluster more/less than it should be. However, once the percentage of each app_name within each cluster is understood, it is evident that sole fact that 5 dimensions has a closer-to-even spread, does not make it the best option.

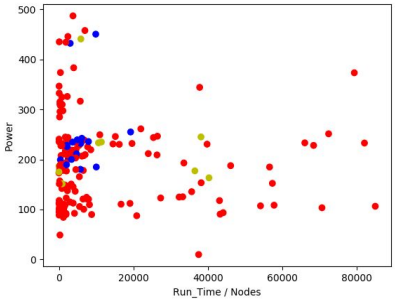
7 Dimensions:
Cluster 0: **100% python**
Cluster 1: **35.6% python**, 35.6% mono, 28.8% gaussian
Cluster 2: **77.8% gaussian**, 22.2% python
5 Dimensions:
Cluster 0: **46% python,** 30% gaussian, 24% mono
Cluster 1: **44.8% python**, 32.4% mono, 22.8% gaussian
Cluster 2: **52.4% gaussian**, 33.3% mono, 14.3% python
2 Dimensions:
Cluster 0: **47.2% python,** 28.5% mono, 24.3% gaussian
Cluster 1: **52% gaussian,** 28% mono, 20% python

Cluster 2: **54% mono,** 27.3% gaussian, 18.2% python

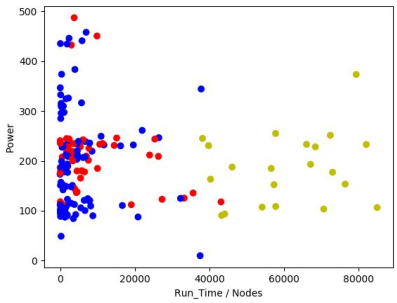
While clusters 0 and 1 for 7-dimension makes it seem like it produces the best output, two very important aspects need to be considered. First, clusters 0 and 1 make up a total of 27 out of the 176 samples. Also, cluster 2, containing 149 out of 176 samples, is barely better at predicting the language than a 3-sided coin flip. The fact that multiple clusters have the same language making up a majority of the samples (0 = python, 2 = python) corroborates the notion that the clusters are not actually clustering the samples accurately.

We believe that the two most important aspects of proper clustering within this data set are: does each language make up the majority of exactly one cluster, and which cluster renders a better prediction between each datasets largest cluster (7D - cluster 1 (149), 5D- cluster 1 (105), 2D- cluster 0 (140))? To both of these points, 2 dimensional data reigns king. The 2 dimensional dataset is the only set that accurately represents one language per cluster, with the next highest language never higher than 28%. Scatter plots in order for side by side comparison of these dimension-counts are depicted below.

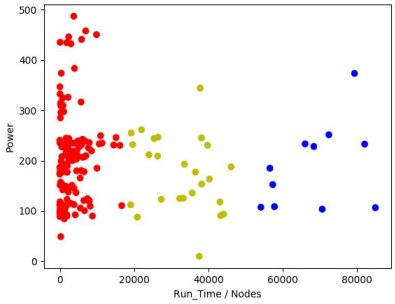
7 dimension



5 dimension



2 dimension



In all three scatter plots, a majority of the samples lie on the X axis between 0 and 7500. To determine if we could possibly get better clusters with a more normalized sample set across the board, we only considered samples within that specified range with 2 dimensions. Our results were as follows:

Cluster	samples	power	wallclock/nodes
0	16	201.87	5,858
1	39	182.48	431
2	20	234.76	2,415.71

Cluster 0: 43.75% Mono, 31.25% Gaussian, 25% Python
Cluster 1: 38.46% Gaussian, 33.3% Python, 28.2% Mono
Cluster 2: 45% Python, 30% Gaussian, 25% Mono

With 26 samples of python, and gaussian and 23 samples of mono, both of our prior concerns were addressed in this trial. We had a similar amount of each language, rather than having 20 more Python samples, and we narrowed the range from 0-85,000 to 0-7,500. It rendered no better results, but arguably worse.

iii. Hierarchical Clustering

Class imbalance is the major problem in the dataset. The clustering algorithm does not seem to work well with all the points we have in the dataset (after the preprocessing). The initial results with all the app_names show that the app_names with more samples in the dataset seem to be contained in almost all the clusters. For example, when the algorithm ran for all the app_names in the dataset with group average as proximity measure, vasp (which has the highest sample number) was present in 9 out of 10 clusters.

Cluster 0: 85% vasp, Cluster 1: 34%, Cluster 2: 87%,
Cluster 4, 5, 7, 8: 100%, Cluster 6: 54%

After removing vasp, python started to exhibit the similar pattern as python is the second most frequent app_name. However, after removing python app_name samples from the dataset, the clusters begin to cluster the data-points a bit better. Hence, we had different runs for the algorithm by considering different app_names each time. We

removed some of the more frequent app_names in each run so that these app_names do not distort the results of the clusters formed.
Here, yellow highlight means that that specific app_name has the highest number of samples in that certain cluster, among all the clusters. Green highlight means that the specific app_name has highest percentage of samples in that certain cluster, among percentages of that app_name in all clusters (if yellow highlight is not already the highest percentage for that app_name). For example, Cluster 3 contains 173 samples of mono, which is the highest number of samples of mono among any cluster, hence yellow highlight. But cluster 0 has 100% mono samples i.e. highest number of samples of mono in any cluster, hence green highlight.

1. Removed vasp and python:
   a. **Group average:**

Cluster 0: 100% mono (4 samples) ; Cluster 1: 100% matlab (1 sample)
Cluster 2: 50% matlab (3 samples), 17% gamess: (1 sample), 17% gaussian (1 sample), 17% ls-dyna (1 sample)
Cluster 3: 69% mono (173), 16% amber (40), 7% mfix: (18), 3% gaussian (10), 1% gamess (3 samples), 1% ls-dyna (3 samples), < 1% fast: (2), < 1% matlab (2)
Cluster 4: 74% gaussian (23), 12% ls-dyna (4), 6% gamess: (2), 6% matlab (2)
Cluster 5: 92% mono (11), 8% gaussian (1)
Cluster 6: 41% fast (17), 17% gaussian (7), 15% gamess (6), 12% amber: (5), 10% mono (4), 5% matlab: (2) ; Cluster 7: 56% gaussian (5), 22% matlab (2), 22% ls-dyna (2)

   b. **Single link (Min)**

Cluster 0: 100% matlab (1) ; Cluster 1: 100% gaussian (1)
Cluster 2: 50% gaussian (1), 50% matlab (1) ; Cluster 3: 100% mono (6)
Cluster 4: 83% mono (5), 16% gaussian (1)
Cluster 5: 6% mfix (18), 1% ls-dyna (3), 15% amber (45), 6% fast (19), 1% matlab (4), 60% mono (177), 3% gamess (9), 5% gaussian (17) ; Cluster 6: 100% mono (4)
Cluster 7: 13% matlab (6), 6% gamess (3), 16% ls-dyna (7), 62% gaussian (27)

   c. **Complete link (Max)**

Cluster 0: 100% mono (6) ; Cluster 1: 100% matlab (1)
Cluster 2: 16% ls-dyna (1), 50% matlab (3), 16% gamess (1), 16% gaussian (1)
Cluster 3: 15% ls-dyna (4), 7% matlab (2), 76% gaussian (20)
Cluster 4: 68% mono (173), 1% ls-dyna (3), < 1% matlab (2), 7% mfix (18), < 1% fast (2), 15% amber (40), 1% gamess (3), 3% gaussian (10)
Cluster 5: 90% mono (9), 10% gaussian (1)
Cluster 6: 9% mono (4), 4% matlab (2), 41% fast (17), 12% amber (5), 14% gamess (6), 17% gaussian (7)
Cluster 7: 14% ls-dyna (2), 14% matlab (2), 14% gamess (2), 57% gaussian (8)

2. Removed vasp, python and mono:
   a. **Group average:**

Cluster 0: 100% matlab (1) ;
Cluster 1: 100% gaussian (1)
Cluster 2: 9% amber (4), 7% ls-dyna (3), 7% gamess (3), 4% fast (2), 21% gaussian (9), 4% matlab (2), 43% mfix (18)
Cluster 3: 12% ls-dyna (4), 74% gaussian (23), 6% matlab (2), 6% gamess (2)
Cluster 4: 7% gaussian (4), 77% amber (41), 3% matlab (2), 11% gamess (6)
Cluster 5: 11% matlab (3), 18% gaussian (5), 3% ls-dyna (1), 3% gamess (1), 62% fast (17) ;
Cluster 6: 22% ls-dyna (2), 55% gaussian (5), 22% matlab (2)

   b. **Single link (Min)**:

Cluster 0: 100% matlab (1) ;
Cluster 1: 100% gaussian (1),
Cluster 2: 50% gaussian (1), 50% matlab (1),

Cluster 3: 19% gaussian (7), 5% matlab (2), 47% fast (17), 13% gamess (5), 13% amber (5)
Cluster 4: 4% matlab (2), 4% fast (2), 6% gamess (3), 40% mfix (18), 15% amber (7), 22% gaussian (10), 6% ls-dyna (3)
Cluster 5: 2% gamess (1), 97% amber (33)
Cluster 6: 63% gaussian (28), 13% matlab (6), 15% ls-dyna (7), 6% gamess (3)

c. Complete link (Max):

Cluster 0: 100% matlab (1) ;
Cluster 1: 100% gaussian (1)
Cluster 2: 16% gaussian (1), 16% ls-dyna (1), 50% matlab (3), 16% gamess (1)
Cluster 3: 7% matlab (2), 76% gaussian (20), 15% ls-dyna (4)
Cluster 4: 3% ls-dyna (3), 12% gaussian (10), 2% fast (2), 3% gamess (3), 2% matlab (2), 23% mfix (18), 51% amber (40)
Cluster 5: 5% matlab (2), 18% gaussian (7), 45% fast (17), 16% gamess (6), 13% amber (5)
Cluster 6: 14% matlab (2), 57% gaussian (8), 14% gamess (2), 14% ls-dyna (2)

These results seem too much to process at first, but there are some specific things we would like to mention. In some cases, a cluster contained 100% of one app_name, however that usually was the case when that was the only one sample in that cluster. In some cases, the algorithm was able to separate out maximum samples of any app_name, if not all, in one cluster. For example, after removing vasp, python and mono, by using single link proximity, the algorithm was able to separate out 33 samples of amber in cluster 5 (which is 97% of that cluster), with only 1 sample of other app_name. For the same configuration, all mfix samples (18) were separated out in cluster 4, which is 40% mfix. However, this cluster has more mix with other app_names, but majority of the samples are mfix. By controlling which app_names to be included in the clustering, we were able to get better results than initially considering all samples in the dataset. There is no clear winner in case of hierarchical clustering since there are little improvements by tweaking the data but no clear difference. However, when we removed vasp and python, single link proximity did a good job of separating out all samples of mfix, amber and fast in one cluster (cluster 5). This shows that these app_names were properly separated but cluster structure needs to improve by having them in separate clusters.

VI. CONCLUSION

In our research, we compared classification techniques such as k-nearest neighbors, Gaussian naive bayes, multilayer perceptron, and random forest; k-means clustering dimension lengths such as 7 dimension, 5 dimension, 2 dimension, and a more normalized 2 dimension; and agglomerative hierarchical clustering. The goal was to deduce which method would render the best results at identifying which job is running based off of performance data from an HPC. We concluded that the random forest classification method produced the best results, comparatively, given the fact that it worked with 81% accuracy, with the next closest classification technique not within 10% of that. Also, for this problem with this dataset, clustering performed subpar. With k-means clustering, we were able to get each of 3 clusters represented primarily by samples of one specific language, but it was never more than 54% of the holistic cluster, which is not the mark of an efficient clustering. Of the four, the best clustering algorithm was with the 2 dimensional data, allowing us to avoid features that seemed less relevant to the problem at hand. With hierarchical clustering, even though we were able to get some good results by removing more frequent apps (hence trying to balance the classes), it still does not compare with classification where random forest achieved an accuracy of 81%.

VII. REFERENCES

[1] NREL HPC Energy Research Data: https://cscdata.nrel.gov/#/datasets/d332818f-ef57-4189-ba1d-beea291886eb
[2] Ozan Tuncer et al. Online Diagnosis of Performance Variation in HPC Systems Using Machine Learning. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, April 2019.
[3] Mark Endrei et al. Statistical and Machine Learning Models for Optimizing Energy in Parallel Applications. In *International Journal of High Performance Computing Applications*, 2018.
[4] Python scikit-learn: https://scikit-learn.org/stable/modules/classes.html
[5] Microsoft LightGBM: https://github.com/Microsoft/LightGBM
[6] Guolin Ke et al. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, 2017.