Input-Based HPC Run Time Prediction

Kenneth Lamar
Department of Computer Science
University of Central Florida
Orlando, United States
kenneth@knights.ucf.edu

Abstract

Predicting the run time of an application is difficult for humans. Sometimes, it is difficult to understand the relationship between the adjustment of an input parameter and its resulting effect on run time. Other times, users simply want to run their job and do not care enough to provide an accurate run time estimate. In high performance computing (HPC), poor predictions result in suboptimal scheduling, or worse, job termination before application completion. In this work, we present a machine learning model to predict the run time of applications accurately and automatically, novelly using the job inputs as predictive parameters. We propose the use of the random forest regressor, which was found to perform best on average among 180 model variants ran on three proxy apps. Ultimately, we found that random forest regression consistently offered state-of-the-art accuracy, as high as 99% and always competitive.

Index Terms

run time prediction, machine learning, high performance computing

I. Introduction

High performance computing (HPC) is used to solve problems that would be impossibly large on conventional machines. HPC platforms consist of many server-grade machines linked via a high-performance network interconnect, working on large, distributed problems. To manage these resources between tasks, a job management system, such as SLURM [1], is used to schedule jobs. Scheduling behavior is based on expected resource usage, such as node count and run time. Providing accurate predictions improves scheduling quality.

Jobs in SLURM are provided in the form of a submission script. To provide expected resource usage to SLURM, users must specify the expected run time of the job in its submission script. If the user overpredicts the run time, it will result in wasteful scheduling, as the scheduler will have already made scheduling decisions under the assumption that the job needed more time. If the user underpredicts, then the job will be terminated early. This happens because the expected run time is typically used in scheduling as the upper bound on the job's allowed run time. Thus, accurate predictions are necessary in HPC to improve scheduling quality.

In the user-led approach, users are often bad at predicting job run time. Typically, users do not want to take the time to determine an accurate run time prediction, even if they are skilled enough to do so, as this takes away time from the work they are trying to perform on the cluster. Even when they do have an accurate run time, they are incentivized to overpredict, to prevent jobs from being killed early.

An alternative is a machine-based predictor. Machines often make their predictions based on historic data. They typically consider only data common to all jobs at their start, such as job name, username, user-provided run time prediction, and number of requested nodes. This data has been applied both to traditional algorithms as well as machine learning approaches. This approach has significant limitations; it misses the underlying inputs used by the job, which often have dramatic impacts on total job run time.

In this work, we propose a machine learning pipeline that predicts the run time of applications using application-specific input parameters as training data. Data was collected on Voltrino, an HPC testbed. We ran synthetic tests with randomized parameters on three proxy apps, ExaMiniMD [2], SWFFT [3], and NEKbone [4]. We removed unchanging parameters, then fed the training data into a random forest regressor model. When testing with five-fold cross-validation, our model performed well against the baselines in all proxy app scenarios, always providing at least a top 10 result against 180 model variants and often providing the best results across all models.

II. Important Definitions and Problem Statement

A. Important Definitions

Our data consists of job input parameters, the data passed into an application, either from file or from command line arguments. The core intuition is that these parameters are principally responsible for changes to job run time. Parameters include both discrete named data (ex. the name of an execution mode) as well as various numeric types (ex. float, int), depending on the test. We also recorded the node and task count for each job, as this determines how many resources are allocated to each job. Whether or not the job crashed is also provided in the training data.

Our prediction target is job run time. As run time is a continuous value, our task is a regression problem. Each job run time in seconds is recorded in the output of each test performed and linked to its associated job input parameters for use in the training and testing data sets.

All data in our data set is clean and precise data that directly corresponds to job run time. The only uncertainty present in the data is run time variance from external factors, such as network congestion. However, discussions with our Sandia partners suggest that the Voltrino network interconnect is hard to saturate, and thus run times should be relatively unaffected by external factors on this test bed.

B. Problem Statement

For this machine learning project, we want to accurately predict job run times. We are provided a set of job input parameters associated with each run, and the resulting run times corresponding to each test. The objective is to learn the function that relates app input parameters to run times, a regression task.

III. Overview of Proposed Approach

In this work, we propose the use of input parameters for use as machine learning parameters to predict job run time. Traditional approaches use only on data that is standard to all jobs: job name, username, user-provided run time prediction, number of requested nodes, etc. While the traditional approach is applicable to any job submitted to the SLURM scheduler, it can provide only a rough approximation of run times. These approaches tend to rely on other factors, such as recency, predicting that jobs will take a similar amount of time as recent jobs with similar

parameters did. This is typically a reasonable assumption, for instance, when a user keeps running the same job with minor tweaks, but it makes the predictor incapable of predicting major changes because the underlying job inputs are opaque to the predictor. In contrast, job input parameters can capture information that can apply to jobs that have never happened before, without the reliance on recency. Machine learning provides the generality to train to each job and learn the significance of each input parameter on final run time, as well as their interrelationships.

In this work, all aspects of the machine learning pipeline were handled using the scikit-learn [5] framework. At a broad level, our proposed machine learning pipeline uses the random forest regressor with 100 fully expanded trees, requiring minimal pre-processing. Existing works have found random forests to offer state-of-the-art results as well as short training times [6]–[9].

IV. Technical Details of Proposed Approach

Features are extracted using a testing script that saves randomly generated valid input parameter sweeps to a CSV file alongside the job run time. All input parameters have the potential to be important to the training data, and thus all are initially provided to the model. From there, if a parameter is left unchanged in all tests associated with a particular proxy app, it is filtered out.

For pre-processing, all non-numeric features are converted to numeric equivalents using a label encoder. This is required because all training features must ultimately be provided to the model as floating-point numbers for scikit-learn. Finally, all features are standardized, as this was found to improve prediction accuracy for nearly every tested model.

We propose the random forest regression model. Existing work has found that this approach provides the best results [6]–[9]. Random forests are valued for their flexibility, requiring minimal pre-processing to get a good result. As a result, little tuning was required. We set our forest size to 100 trees. Each tree was set up to grow fully, with no impurity threshold or depth limits.

V. Experimental Results

A. Data Description

The data set consists of numerous runs of three proxy apps representative of common HPC workloads, ExaMinMD, SWFFT, and NEKbone. This data set was generated on Voltrino, a 24 node Knights Landing (KNL) cluster at Sandia National Labs. Voltrino is a testbed based on the Cray XC40 design used in the Trinity cluster [10]. Runs were synthetically generated and tested using parameters randomly chosen between sane input ranges. The number of nodes was varied from 1 to 4 and the number of tasks was varied from 1 to 32 to evaluate the impact of adjusting the number of available resources on run times.

ExaMiniMD [2] is a modular application used in the simulation of molecular dynamics. This application's input is contained entirely within an input file. Valid input files are LAMMPS [11] input scripts, with a subset of supported operations. Tests on this proxy app consisted of two major use cases, a SNAP simulation and a 3D Lennard-Jones melt. Within both cases, the lattice size, time step size, and number of steps was varied at random for each test. Our final ExaMiniMD data set comprised 1,562 test runs and 21 features.

SWFFT [3] is a standalone version of the distributed 3D Fast Fourier Transform (FFT) application used by HACC. Application input parameters for this app consist entirely of command line arguments. All available parameters were adjusted, including the grid size on each axis (X, Y, and Z) and the number of repetitions. Our final SWFFT data set comprised 1,618 test runs and 7 features.

NEKbone [4] is a Helmholtz equation solver for physics calculations, pared down from functionality in Nek5000 software. All available parameters were varied between sane defaults. Input parameters were passed in via an input file. Unlike the other proxy apps tested, NEKbone was constrained to test only on one node with 10 tasks, as the version used was compiled to run on exactly 10 machine cores. Our final NEKbone data set comprised 1,579 test runs and 12 features.

B. Evaluation Metrics

Prediction quality was measured using five-fold cross-validation of the correlation coefficient, R^2, as well as the root-mean-square error (RMSE). R^2 is a standard way to measure prediction accuracy in regression tasks. With this metric, an R^2 of 1 is perfect correlation between the predicted and actual run time, 0 means the predictions are of equivalent quality to always predicting the average run time, and negative values of R^2 are for anything worse than that average. R^2 results are reported in Fig. 1. RMSE reports the square root of the sum of the squares of each feature error. It is a standard risk metric used to report loss. RMSE results are reported in Fig. 2. For all metrics, five-fold cross-validation is used as an easy way to split the training data from the testing data, with the average of those runs reported in the results.

C. Baseline Models

We compared our proposed model against seven alternatives, varying major model parameters to tune for best results. We also tried each model against three slightly adjusted pre-processing techniques, since some models benefit from adjustments to the input data set. This resulted in a total of 180 model variants, with each of these variants tested against our three proxy apps.

As baselines, we compared against the Bayesian ridge regressor, the linear stochastic gradient descent (SGD) regressor, and the decision tree regressor, all with default parameters in scikit-learn. We also compared against the support vector regressor with RBF; sigmoid; and first, second, and third degree polynomial kernels. We tested the k-nn regressor with k varied from 1 to 7. We tested PLS regression, varying the number of components to keep from 1 to 4. Finally, we tested against a variety of multi-layer perceptron (MLP) approaches, varying from 1 to 10 layers, with 100 hidden neurons per layer. We evaluated several MLP perceptron types, including linear no-op (identity), logistic sigmoid (logistic), hyperbolic tangent (tanh), and the rectified linear unit (relu) perceptron functions. All MLP approaches used the ”adam” solver, a stochastic gradient-based optimizer. Alternative solvers were attempted but offered poor results.

D. Performance Results

Full performance results are reported in Fig. 1 and Fig. 2. In both figures, rows contain unique model variants and columns contain unique pre-processing variants. Below each pre-processing variant’s name, the shape of the

	ExaMiniMD			SWFFT			NEKbone		
	base	norm	poly	base	norm	poly	base	norm	poly
	(1562, 21)	(1562, 21)	(1562, 253)	(1618, 7)	(1618, 7)	(1618, 36)	(1579, 12)	(1579, 12)	(1579, 91)
Random Forest Regressor	**0.732664193**	**0.793603084**	0.366890571	**0.773893103**	0.466321999	**0.768365766**	**0.987550752**	**0.961175425**	**0.983676451**
Bayesian Ridge	0.307756732	0.403984823	0.500059676	0.116693646	0.104908667	0.241554132	0.753037023	0.792200928	0.928062273
SVR RBF	-0.135880024	-0.135686792	-0.142561288	-0.017999624	-0.018214607	-0.016919815	0.697806037	0.673379979	0.349620153
SVR poly 1	-0.126490846	-0.125976458	-0.142927947	-0.015667768	-0.018503565	-0.003018618	0.700084911	0.74004849	0.506908346
SVR poly 2	-0.14445748	-0.143431834	-0.143149637	0.004299923	-0.015586346	0.053558902	0.134835615	0.139800799	0.298368932
SVR poly 3	-0.129263498	-0.129190228	-0.143565874	0.039700812	-0.018019803	0.168092353	0.523825521	0.586308741	0.192477663
SVR sigmoid	-0.13000108	-0.129527472	-0.143423841	-0.019544249	-0.018731357	-0.016706519	0.672104169	0.726383397	0.490134864
Linear SGD Regressor	0.221632858	0.359253608	-5.40035E+14	0.116253053	0.112473612	0.221376314	0.752966947	0.792282003	0.927943305
1-NN Regressor	0.367107084	0.304131568	0.395624225	-0.146776915	-0.362223465	-0.253639875	0.691180236	0.678619129	0.69151092
2-NN Regressor	0.609903365	0.525537524	0.593242722	0.222837234	0.052260528	0.264000531	0.784803476	0.772570749	0.759549548
3-NN Regressor	0.671833323	0.618137749	0.640664003	0.275830152	0.116816051	0.323749123	0.81258348	0.812460521	0.794122366
4-NN Regressor	0.700854864	0.656493371	0.671178626	0.267897322	0.159843328	0.279488046	0.827723074	0.832492201	0.802000385
5-NN Regressor	**0.707915126**	0.680321377	0.666687502	0.316067235	0.204969371	0.371750835	0.827509393	0.835130115	0.803734156
6-NN Regressor	**0.710922018**	0.692181235	0.670626618	0.31873803	0.264421641	0.3391312	0.832325442	0.837826199	0.801657841
7-NN Regressor	**0.705068932**	0.695910513	0.665738518	0.319817192	0.252547872	0.344394446	0.839435085	0.842470356	0.797520222
1 PLS Regression	0.240744462	0.245383016	0.127370886	0.11818297	0.108804241	0.196858163	0.688530285	0.754428332	0.685547515
2 PLS Regression	0.375592003	0.383252959	0.152992308	0.116590962	0.109471845	0.233939369	0.739719973	0.787792059	0.863759533
3 PLS Regression	0.284778196	0.387041444	0.190119149	0.116477713	0.109498948	0.228810849	0.752306383	0.792045991	0.913120579
4 PLS Regression	0.242763526	0.364700751	0.231752265	0.116504176	0.10950024	0.229221337	0.75299082	0.792178295	0.924910383
Decision Tree Regressor	0.465568922	0.586909588	0.036563739	**0.706744772**	-0.002902105	**0.653430439**	**0.982973318**	0.932128581	**0.977391578**
1 MLP Regressor relu	0.36016759	0.038285367	0.360396233	0.131939728	0.04697183	0.252832297	0.943922423	0.928821728	0.940640684
1 MLP Regressor identity	0.141517218	0.065186375	0.325163783	0.115226455	0.046076042	0.24401984	0.752793106	0.792293061	0.927976906
1 MLP Regressor logistic	-0.307721818	-0.301958435	-0.309733148	-0.002870644	-0.00331181	-0.002462966	0.861801076	0.852595789	0.950414287
1 MLP Regressor tanh	-0.314342737	-0.298735969	-0.303211819	-0.002725814	-0.002014774	-0.002797012	**0.960217312**	0.92040598	0.930030764
2 MLP Regressor relu	0.670730938	0.616675788	0.2852946	0.350777046	0.231654069	0.513012555	0.947754064	0.939572785	0.92192882
2 MLP Regressor identity	0.188442077	0.395497888	0.379647481	0.116526427	0.108929624	0.232535155	0.752478888	0.792487633	0.92725614
2 MLP Regressor logistic	-0.309037559	-0.306756485	-0.311393304	-0.003964092	-0.003665409	-0.005731755	0.948540176	0.904152315	0.916486297
2 MLP Regressor tanh	-0.307939686	-0.302698889	-0.303243399	-0.004155649	-0.002508126	-0.006532709	**0.959878515**	**0.961924671**	0.918969823
3 MLP Regressor relu	0.657077043	**0.747587443**	0.307125095	**0.574850136**	0.232826148	0.507547721	0.937344636	0.929437046	0.917727918
3 MLP Regressor identity	0.219022702	0.399394624	0.402910755	0.118301803	0.108437011	0.232715174	0.751427749	0.791779638	0.924459492
3 MLP Regressor logistic	-0.313593043	-0.313573747	-0.313118712	-0.007956199	-0.007942338	-0.007607194	**0.957651743**	0.902657262	0.905089446
3 MLP Regressor tanh	-0.305475402	-0.303159011	-0.303135536	-0.00609808	-0.004732506	-0.004474352	0.945039777	**0.959728858**	0.920001908
4 MLP Regressor relu	0.660902472	**0.733524815**	0.305694608	**0.587395591**	0.180372777	0.540498616	0.929023526	0.926091211	0.917804436
4 MLP Regressor identity	0.244919296	0.400679277	0.391428336	0.115247294	0.109580637	0.230717528	0.75140987	0.792381052	0.926612709
4 MLP Regressor logistic	-0.314412205	-0.314410783	-0.312049389	-0.00896257	-0.008960441	-0.008217155	0.572931175	-0.003248922	0.917501817
4 MLP Regressor tanh	-0.31316086	-0.303192246	-0.303287163	-0.006890548	-0.003560948	-0.006516782	0.943835376	0.949973488	0.92531092
5 MLP Regressor relu	0.661044698	0.688208932	0.356655583	**0.613499161**	0.198310042	0.456287197	0.92974628	0.92505743	0.918711316
5 MLP Regressor identity	0.247392674	0.401802767	0.38269591	0.117978713	0.109282225	0.225320145	0.750561757	0.791407014	0.927255536
5 MLP Regressor logistic	-0.314123267	-0.314122718	-0.311917623	-0.00815554	-0.008155465	-0.007940113	-0.002906833	-0.002907061	-0.003191628
5 MLP Regressor tanh	-0.313966338	-0.303232349	-0.303304571	-0.005738436	-0.005743009	-0.005549735	0.947825714	0.946275149	0.925498262
6 MLP Regressor relu	0.613428274	0.672639948	0.324908173	**0.663531518**	0.261978768	0.499473414	0.930585198	0.925324263	0.915875112
6 MLP Regressor identity	0.296030441	0.400481181	0.388943333	0.110215401	0.106712979	0.234739514	0.75206996	0.788754232	0.923944051
6 MLP Regressor logistic	-0.31443382	-0.314433774	-0.313740696	-0.00816883	-0.008168829	-0.007933491	-0.002894507	-0.002894522	-0.00303339
6 MLP Regressor tanh	-0.307008812	-0.303073084	-0.303180037	-0.005746587	-0.005749519	-0.005730032	0.94177433	-0.003085608	0.92967725
7 MLP Regressor relu	**0.721817927**	0.703173623	0.45865547	**0.688547515**	0.122370456	0.482713602	0.931410474	0.924212497	0.914034857
7 MLP Regressor identity	0.297576354	0.400210995	0.463587083	0.1173259	0.110405329	0.220594931	0.74583252	0.791928864	0.921952445
7 MLP Regressor logistic	-0.314324193	-0.31432419	-0.314667097	-0.008415364	-0.008415363	-0.008081911	-0.003216815	-0.003216817	-0.003128295
7 MLP Regressor tanh	-0.311494424	-0.303207903	-0.303218486	-0.005753421	-0.005756942	-0.005767619	0.943603021	-0.003144193	0.927283586
8 MLP Regressor relu	0.633618262	**0.715461133**	0.45311566	0.545606624	0.180382637	0.508354661	0.933713931	0.919469891	0.916357953
8 MLP Regressor identity	0.278230753	0.397626729	0.408632904	0.118428457	0.106614093	0.230356021	0.74779763	0.788615303	0.924072382
8 MLP Regressor logistic	-0.314596686	-0.314596686	-0.312952078	-0.008283131	-0.00828313	-0.00849232	-0.002840383	-0.002840383	-0.003109842
8 MLP Regressor tanh	-0.306986244	-0.303362573	-0.303176553	-0.005715079	-0.00571617	-0.005714283	0.94160496	-0.003187306	0.9316586
9 MLP Regressor relu	0.695899895	0.675906718	0.407981449	0.571388871	0.270346834	0.382705235	0.933040149	0.929443529	0.91930292
9 MLP Regressor identity	0.277142611	0.405559063	0.415426692	0.115179568	0.104931341	0.239383747	0.74510536	0.77952027	0.91716362
9 MLP Regressor logistic	-0.313099923	-0.313099922	-0.316341175	-0.008612067	-0.008612067	-0.007855291	-0.003159196	-0.003159196	-0.002967624
9 MLP Regressor tanh	-0.303301562	-0.303301325	-0.303179768	-0.005750321	-0.005752884	-0.005789497	0.940689812	-0.003214355	0.556713816
10 MLP Regressor relu	0.697750441	**0.724049516**	0.392120052	**0.667368104**	0.141418761	0.464959168	0.933138197	0.928039941	0.916655366
10 MLP Regressor identity	0.275876747	0.406803013	0.393678148	0.114063477	0.102655397	0.23150956	0.743397833	0.775902013	0.922837546
10 MLP Regressor logistic	-0.31222043	-0.31222043	-0.314481521	-0.008751273	-0.008751273	-0.008595119	-0.00316001	-0.00316001	-0.003126391
10 MLP Regressor tanh	-0.308011171	-0.303171343	-0.303258251	-0.005791864	-0.005794156	-0.005725702	0.93848226	-0.003010219	0.371413844

Fig. 1: R^2 for each test. Color scales are used, where green indicates a good result, red indicates a poor result, and yellow indicates an average result. The top 10 best results for each proxy app are highlighted in **bold**.

	ExaMiniMD			SWFFT			NEKbone		
	base (1562, 21)	norm (1562, 21)	poly (1562, 253)	base (1618, 7)	norm (1618, 7)	poly (1618, 36)	base (1579, 12)	norm (1579, 12)	poly (1579, 91)
Random Forest Regressor	**4764.0635**	**4344.310531**	6216.032985	**1984.463122**	2934.427639	**1950.449106**	**3.044853225**	**5.567193378**	**3.601639548**
Bayesian Ridge	7730.456744	7335.208279	6482.042066	4108.664267	4133.159498	3823.764683	13.91073033	12.77640106	7.500978111
SVR RBF	10120.01668	10119.17998	10148.71726	4368.249249	4368.669888	4366.061623	15.45483292	16.06738243	22.67029422
SVR poly 1	10078.91811	10076.3341	10150.29825	4364.522991	4369.286557	4342.563724	15.38666495	14.3280767	19.73967553
SVR poly 2	10157.10917	10152.80042	10151.30319	4329.654969	4363.462208	4244.064148	26.14556249	26.07264023	23.54561252
SVR poly 3	10091.59019	10090.78828	10153.04162	4269.752789	4368.340657	4032.488027	19.39532429	18.08152828	25.25771899
SVR sigmoid	10093.92853	10091.83707	10152.72053	4371.166088	4369.697273	4365.703857	16.08637268	14.70008479	20.07135868
Linear SGD Regressor	7927.628557	7488.17857	1.88807E+11	4112.130851	4131.58221	3876.717579	13.91938001	12.77738418	7.492680585
1NN Regressor	7468.294059	7705.558648	7280.469043	4613.103499	4896.980068	4749.619263	15.57436879	15.8905247	15.58813591
2NN Regressor	5948.171084	6482.735557	6044.297695	3832.246768	4235.656757	3690.293745	12.99773857	13.36825479	13.76908096
3NN Regressor	5458.098362	5841.793638	5675.6854	3670.077754	4054.454614	3540.377414	12.14990148	12.13905918	12.74128691
4NN Regressor	5208.57866	5552.29751	5431.693803	3719.987708	4003.825544	3661.034185	11.63936022	11.47894145	12.47266208
5NN Regressor	5152.027705	5379.906484	5467.067627	3640.755414	3909.405461	3458.508781	11.63953895	11.39414831	12.40888042
6NN Regressor	5127.556554	5284.267226	5431.507417	3628.158765	3764.559547	3552.046522	11.48507653	11.31048168	12.47938819
7NN Regressor	5179.451219	5249.574956	5465.172992	3615.848061	3777.844413	3544.018091	11.2424553	11.14222375	12.62152581
1 PLS Regression	8284.079663	8261.872841	8850.901526	4105.025942	4126.412677	3955.213893	15.63751258	13.89744037	15.75137981
2 PLS Regression	7508.58113	7472.08404	8496.058787	4108.507607	4126.257046	3836.64187	14.28339654	12.9111078	10.35366502
3 PLS Regression	7834.019319	7422.996532	7833.375707	4109.298424	4126.262593	3841.813525	13.93152899	12.7809775	8.255324176
4 PLS Regression	7977.663264	7526.007376	7403.047017	4109.25824	4126.263668	3842.33465	13.9110034	12.77649626	7.667895026
Decision Tree Regressor	6624.213927	6001.071861	8435.889115	**2320.195573**	3887.837932	**2122.509291**	**3.639024101**	7.557621643	**4.116250758**
1 MLP Regressor relu	7620.639923	9318.766969	6949.325099	4066.161382	4233.700716	3806.6849	6.607609646	7.476429498	6.802067508
1 MLP Regressor identity	8653.227927	9175.145963	7322.207498	4107.017751	4234.959925	3822.219383	13.91681649	12.77341891	7.502869386
1 MLP Regressor logistic	10842.08469	10821.93386	10853.70541	4338.270046	4339.042088	4337.601266	10.42096377	10.76678111	6.239098649
1 MLP Regressor tanh	10862.99289	10808.74715	10827.17304	4338.045305	4336.549787	4338.335149	**5.58313105**	7.902744896	7.409365629
2 MLP Regressor relu	5354.910336	5869.897705	6988.295187	3506.686653	3827.847763	2941.955442	6.381990634	6.892106524	7.807019547
2 MLP Regressor identity	8141.459571	7381.462806	6885.83083	4107.910929	4126.647308	3837.446434	13.92643386	12.76628715	7.542652368
2 MLP Regressor logistic	10850.70998	10841.5804	10860.48521	4340.435522	4339.866395	4343.986237	6.370689002	8.683666747	8.094524548
2 MLP Regressor tanh	10842.89539	10825.08915	10827.32742	4340.838193	4337.747468	4345.534466	**5.622828168**	**5.464777967**	7.951266986
3 MLP Regressor relu	5363.516327	**4723.886702**	6876.83982	**2705.189722**	3789.370489	2969.813025	6.952794781	7.449227394	8.022188246
3 MLP Regressor identity	8053.537416	7360.482815	6811.765731	4105.268507	4127.732332	3834.984429	13.95566632	12.79138106	7.692975548
3 MLP Regressor logistic	10869.42044	10869.34228	10867.4922	4348.238244	4348.209766	4347.516253	**5.774487471**	8.756217346	8.63026788
3 MLP Regressor tanh	10835.02308	10827.05044	10826.89322	4344.438815	4342.764441	4342.192106	6.555705236	**5.626514635**	7.901808663
4 MLP Regressor relu	5232.735121	**4732.432145**	6794.991558	**2681.016954**	3968.28712	2897.018906	7.435002378	7.610236295	7.99134921
4 MLP Regressor identity	7959.269829	7357.028798	6840.169282	4110.732405	4125.746512	3841.219239	13.95944308	12.76953866	7.579606692
4 MLP Regressor logistic	10872.74207	10872.73636	10863.14535	4350.246699	4350.242039	4348.738555	14.72643685	28.1411383	8.011969165
4 MLP Regressor tanh	10860.92201	10827.12487	10827.50273	4346.600734	4339.422125	4345.103987	6.65522009	6.264609967	7.645729265
5 MLP Regressor relu	5215.688006	4992.889247	6742.728684	**2620.119918**	3914.893174	3115.139606	7.387467066	7.663536281	7.983688085
5 MLP Regressor identity	7973.701989	7345.685686	6890.874314	4107.354789	4126.852924	3851.163037	13.97752331	12.80318731	7.543686137
5 MLP Regressor logistic	10871.56673	10871.56454	10862.6004	4348.638784	4348.638628	4348.195261	28.13648566	28.13648885	28.14040896
5 MLP Regressor tanh	10863.63122	10827.27362	10827.57146	4343.751542	4343.761025	4343.204459	6.416864475	6.504703309	7.649365095
6 MLP Regressor relu	5388.342481	5109.840875	6938.095456	**2409.364742**	3771.556276	2938.101031	7.343932148	7.664252551	8.113798167
6 MLP Regressor identity	7788.520727	7358.112557	6940.477783	4117.932656	4129.955645	3837.050683	13.94166818	12.88769125	7.72646213
6 MLP Regressor logistic	10872.82066	10872.82048	10870.01951	4348.653288	4348.653286	4348.174469	28.13634664	28.13634686	28.13806817
6 MLP Regressor tanh	10840.03763	10826.62143	10827.07469	4343.756518	4343.762599	4343.714501	6.780900385	28.1388945	7.411563135
7 MLP Regressor relu	**4779.591288**	**4831.015113**	6216.042166	**2321.53799**	4017.298189	3015.966069	7.292516691	7.684824171	8.204169949
7 MLP Regressor identity	7786.528762	7366.829509	6607.891996	4108.059277	4124.055217	3857.326368	14.11469274	12.78228204	7.830427656
7 MLP Regressor logistic	10872.37906	10872.37905	10873.76037	4349.153497	4349.153496	4348.492411	28.14075616	28.14075618	28.13942279
7 MLP Regressor tanh	10855.32458	10827.17614	10827.20895	4343.777717	4343.784931	4343.805887	6.666939314	28.13969762	7.534844913
8 MLP Regressor relu	5432.26437	**4967.139186**	6358.503543	2833.495348	3885.64122	2934.864033	7.194507264	7.917560365	8.108161572
8 MLP Regressor identity	7867.600593	7380.998025	6860.340604	4106.444333	4132.172416	3840.613251	14.06935825	12.8952392	7.704947334
8 MLP Regressor logistic	10873.48925	10873.48925	10866.8069	4348.872151	4348.872151	4349.290924	28.13550591	28.13550592	28.13931057
8 MLP Regressor tanh	10840.17295	10827.81361	10827.05284	4343.674138	4343.676436	4343.68271	6.788920068	28.14051234	7.314400007
9 MLP Regressor relu	**4951.77647**	5042.253521	6529.845995	2818.068644	3767.071421	3210.409634	7.216739822	7.411463759	7.965710543
9 MLP Regressor identity	7862.061538	7331.874215	6759.100495	4112.415417	4134.810681	3824.733924	14.12241519	13.16915458	8.030733945
9 MLP Regressor logistic	10867.4133	10867.41329	10880.54318	4349.548795	4349.548795	4348.029646	28.13997793	28.13997793	28.13733572
9 MLP Regressor tanh	10827.55505	10827.5601	10827.06958	4343.76029	4343.765419	4343.853388	6.844997652	28.14047022	15.35213705
10 MLP Regressor relu	**4982.965002**	**4804.556992**	6576.40138	**2445.247125**	4007.414082	3053.139028	7.217723945	7.487983314	8.083747214
10 MLP Regressor identity	7865.473031	7327.436524	6879.293441	4116.437759	4138.081036	3844.324224	14.18818844	13.27826937	7.769067759
10 MLP Regressor logistic	10863.8445	10863.8445	10873.00779	4349.81699	4349.81699	4349.520785	28.13993313	28.13993313	28.13938126
10 MLP Regressor tanh	10843.52706	10827.02851	10827.37766	4343.839795	4343.844507	4343.719106	6.962819567	28.13776351	19.61608228

Fig. 2: RMSE for each test. Color scales are used, where green indicates a good result, red indicates a poor result, and yellow indicates an average result. The top 10 best results for each proxy app are highlighted in **bold**.

data set is provided, listing the number of data points followed by the number of features per data point. Color scales are used to provide at-a-glance insights on which models work best, where green indicates a good result, red indicates a poor result, and yellow indicates an average result. The top 10 best results for each proxy app are highlighted in bold to provide context for which results are best relative to the whole test suite. Recall that all tests report the average result of five-fold cross-validation, to separate the training and testing data set.

We start by discussing the three compared pre-processing types. Base applies the base pre-processing used by all tests; features that never change are removed from the data set, and all data is standardized. The rationale is that features that never change never contribute to the data set and thus can be safely removed. This would happen, for instance, in NEKbone testing, where the number of processors and nodes are fixed and thus those features do not contribute. Standardization was found in previous testing to have little to no adverse effects on any models while greatly enhancing results for some models. As a result, it is applied across the board in all tests. Norm applies the base pre-processing and additionally normalizes the data. This change generally led to a poor result, but occasionally improved MLP regressor results. Poly applies the base pre-processing and additionally adds second degree polynomial versions of all feature combinations to the feature set. This pre-processing had only minor impacts, occasionally bolstering certain MLP regressor results.

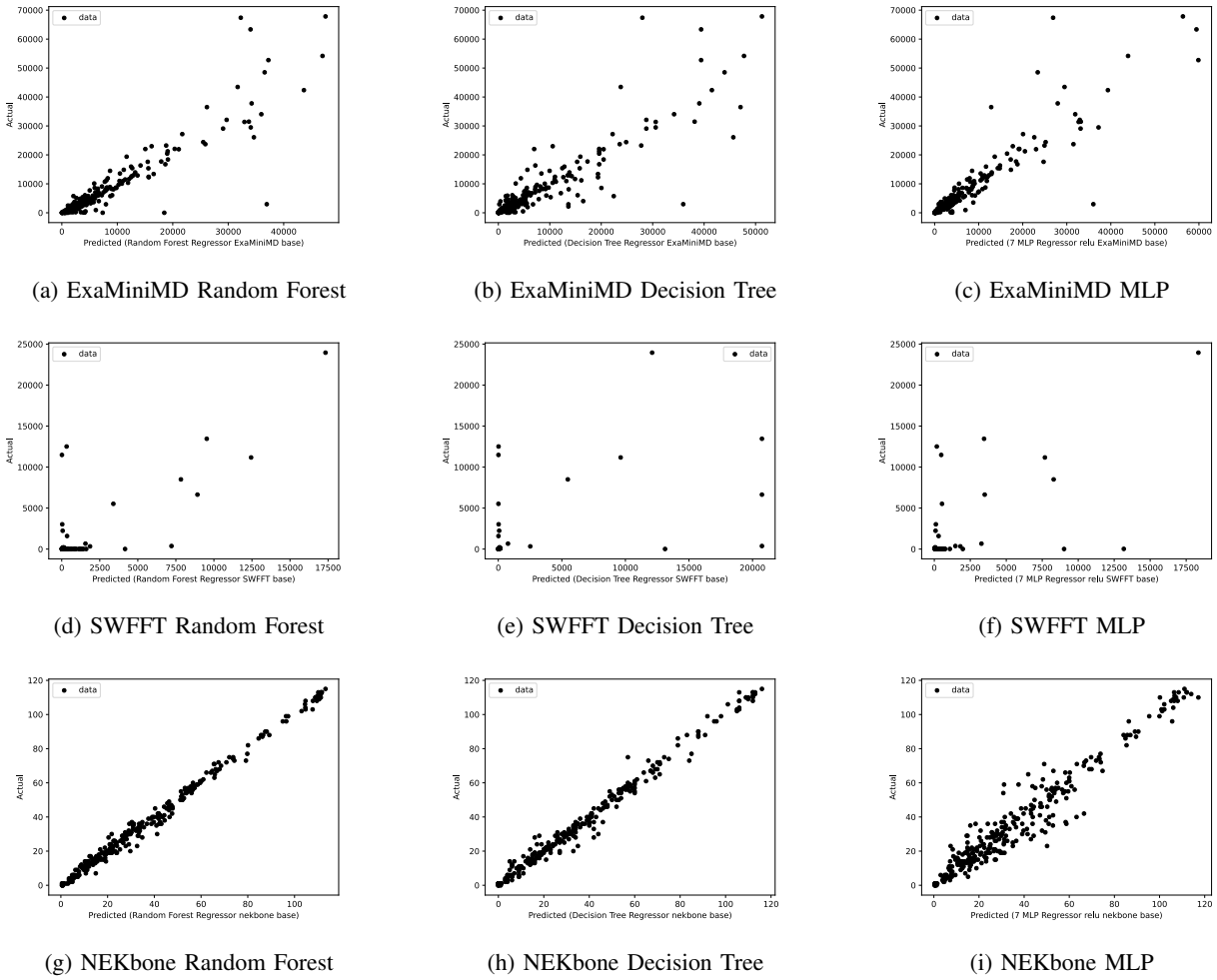
Now we can discuss the results of each machine learning model. Most existing works find that the random forest regressor offers the most accurate model while training quickly [6]–[9]. Our results agree with these findings. We find that base pre-processing with the random forest regressor, our proposed approach, is the only method that achieves top 10 in all three proxy app scenarios. In fact, the base random forest regressor is the best evaluated approach for SWFFT and NEKbone, while being fourth best, losing by a small margin, for ExaMiniMD. This is accomplished while requiring minimal tuning or pre-processing. Even the standardization performed on all tests could be removed and would likely improve its results marginally.

Other promising approaches include the decision tree regressor and the MLP regressor with the rectified linear unit perceptron (relu). These models often provided competitive results against the random forest regressor but lacked the needed consistency, as a model that happened to work well in one set of proxy app tests often performed poorly in another. We found that 3 or 7 layers of the MLP with the relu perceptron performed best when tuning parameters. All other perceptron types generally performed poorly.

Other approaches worth mentioning include SVR and KNN. We used a radial basis kernel function (RBF) in SVR with little success, though other works have found it to perform well [12]. KNN tended to perform well on average and has historically performed reasonably well in related works. a K value around 6 seemed to be ideal for parameter tuning.

We noticed that NEKbone provides significantly better accuracy results for all tests. We attribute this to the relatively short runs of each test and narrow range of valid parameters when compared with ExaMiniMD and SWFFT, allowing for a narrower selection of prediction choices and many more tests with similar run times.

The correlation graphs in Fig. 3 visualize some of the best performing machine learning models against each of the tested proxy apps. The X-axis represents the predicted run time in seconds. The Y-axis represents the actual run time in seconds. Each point on the graph is a test data point. Each graph is the visualization of 20% of all data,



(a) ExaMiniMD Random Forest (b) ExaMiniMD Decision Tree (c) ExaMiniMD MLP

(d) SWFFT Random Forest (e) SWFFT Decision Tree (f) SWFFT MLP

(g) NEKbone Random Forest (h) NEKbone Decision Tree (i) NEKbone MLP

Fig. 3: Correlation graphs of three major machine learning models.

with the other 80% used as training, comparable to a single run of five-fold cross-validation. Using these graphs, we can see which run lengths are most responsible for inaccurate predictions. In the case of ExaMiniMD, long test runs are generally difficult to predict because fewer tests were performed in that range. SWFFT is dependent on very specific input for a valid run; the random testing resulted in many invalid tests that ran for a very short duration. As a result, there are many points clumped at short run time, and a handful of poorly predicted points with long run times. It is evident from this visualization that SWFFT results would be improved if random inputs were validated or constrained before running, to ensure tests that run to completion. NEKbone has excellent correlation, just as you would expect from the 99% accuracy seen in the R^2 measurement.

VI. Related Works

Run time prediction is a popular problem with a large body of existing work.

A. Traditional Prediction

Smith et al. proposed a system that categorized workloads based on standard job parameters, then making predictions based on historic runtimes associated with each workload class [13]. Galleguillos et al. likewise used workload categories [14]. Jobs were split into per-user categories, then predicted the run time to be equal to the most recent job with the longest matching substring. This work found that this heuristic was better than machine learning on the same data.

Rather than using job parameters or input parameters, several approaches use static analysis of the application itself to predict job run time [15], [16]. This is an excellent approach for application configurations that have never been seen before.

B. Machine Learning Prediction

Matsunaga et al. proposed one of the earliest attempts to use machine learning, specifically Support Vector Machine and k-nearest neighbors, to predict job run times from standard workload data [17].

Hutter et al. performed a thorough evaluation of existing uses of machine learning models to predict run times [6]. They evaluated 11 algorithms and compared their effectiveness against 35 domains. In their work, they found that the random forest regressor provided the best results in terms of prediction quality and training times in nearly all cases.

Emeras et al. used supervised learning techniques to predict several aspects of HPC resource usage metrics, including job run time [18]. Much like other works, they constrained themselves to standard job data, rather than input parameters. Their approach used confusion matrices to bin applications into major resource intensiveness categories rather than specific run time predictions.

McKenna et al. used decision trees, random forests, and K-nearest neighbors to predict run time, IO, and other HPC aspects using job script contents as input for testing [7]. When permitting up to a 10-minute error tolerance, predictions were accurate 73% of the time in their testing. This approach was designed for use in production, so it also covered aspects like retraining rate and the optimal amount of recent data to use during retraining.

Aaziz et al. used non-invasive data collection techniques to predict run time, even mid-run [19]. They introduced the idea of a problem size metric, suggesting that it could be provided by the programmer, or identified among job input parameters. These initial ideas were applied to a linear regression model, for simplicity.

Wyatt et al. proposed PRIONN, a deep learning technique where a neural network is fed the entire job script via word2vec to infer job run time [8]. This is an excellent generalized approach that overcomes the limits of reading job input parameters and parsing by hand, but it only considers the job script itself, ignoring the surrounding input files that many applications use.

Wang et al. is the most closely related work to our own [12]. Like our work, they identify a common weakness, where most techniques ignore the value in input files and parameters. They provide results on a single application, VASP, for run time prediction. Their approach uses an RBF network for training.

VII. Conclusions

In this work, we proposed the use of input parameters as training inputs to a random forest regressor model to predict job run times. We found that this model consistently provides the highest accuracies when compared against seven alternative machine learning baselines with varied parameters. In all, the random forest regressor with minimal pre-processing was found to perform best on average among 180 distinct model variants ran on three proxy apps.

In future work, we plan to test additional proxy apps for a more diverse set of results. We are also interested in predicting other factors that are dependent on input parameters, such as memory, network, and disk use, as well as crash detection. We would also like to propose a certainty measure, to not only predict a run time, but also how likely the value is to be accurate. Via cautious overprediction on less certain results, this measure could reduce the possibility of underprediction followed by early job termination. We have also seen interesting work using deep learning to predict job run time by feeding in entire job script contents [8]; we hope to expand on that work to additionally cover input files. Finally, all tests in this work were synthetic. We are interested in integrating this work as a tool in a real HPC environment where the model is learned from real-world data and used for predictions in a scheduler.

Project files can be found at https://github.com/KennethLamar/CAP5610/tree/main/Project/Final. It contains the raw data; the script used to generate, run, and parse tests as well as run the machine learning pipeline; and the paper source and PDF.

References

[1] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60.

[2] C. design center for Particle Applications, "Examinimd," https://github.com/ECP-copa/ExaMiniMD, 2017.

[3] A. Pope, "Swfft," https://git.cels.anl.gov/hacc/SWFFT, 2017.

[4] P. Fischer and K. Heisey, "Nekbone," https://github.com/ECP-copa/ExaMiniMD, 2014.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[6] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370213001082

[7] R. McKenna, S. Herbein, A. Moody, T. Gamblin, and M. Taufer, "Machine learning predictions of runtime and io traffic on high-end clusters," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 255–258.

[8] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, and M. Taufer, "Prionn: Predicting runtime and io using neural networks," in *Proceedings of the 47th International Conference on Parallel Processing*, ser. ICPP 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3225058.3225091

[9] G. Ozer, S. Garg, N. Davoudi, G. Poerwawinata, M. Maiterth, A. Netti, and D. Tafani, "Towards a predictive energy model for hpc runtime systems using supervised learning," in *Euro-Par 2019: Parallel Processing Workshops*, U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, and S. L. Scott, Eds. Cham: Springer International Publishing, 2020, pp. 626–638.

[10] A. M. Agelastos, M. Rajan, N. Wichmann, R. Baker, S. P. Domino, E. W. Draeger, S. Anderson, J. Balma, S. Behling, M. Berry, P. Carrier, M. Davis, K. McMahon, D. Sandness, K. Thomas, S. Warren, and T. Zhu, "Performance on trinity phase 2 (a cray xc40 utilizing intel xeon phi processors) with acceptance applications and benchmarks." 5 2017. [Online]. Available: https://www.osti.gov/biblio/1457905

[11] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465521002836

[12] Q. Wang, J. Li, S. Wang, and G. Wu, "A novel two-step job runtime estimation method based on input parameters in hpc system," in *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, April 2019, pp. 311–316.

[13] W. Smith, I. Foster, and V. Taylor, "Predicting application run times with historical information," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1007–1016, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731504000991

[14] C. Galleguillos, A. Sîrbu, Z. Kiziltan, O. Babaoglu, A. Borghesi, and T. Bridi, "Data-driven job dispatching in hpc systems," in *International Workshop on Machine Learning, Optimization, and Big Data*. Springer, 2017, pp. 449–461.

[15] M. A. Obaida, J. Liu, G. Chennupati, N. Santhi, and S. Eidenbenz, "Parallel application performance prediction using analysis based models and hpc simulations," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 49–59. [Online]. Available: https://doi.org/10.1145/3200921.3200937

[16] H. Cheon, J. Ryu, C. Y. Park, and Y.-S. Han, "Sw runtime estimation using automata theory and deep learning on hpc," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, Aug 2020, pp. 7–12.

[17] A. Matsunaga and J. A. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, May 2010, pp. 495–504.

[18] J. Emeras, S. Varrette, M. Guzek, and P. Bouvry, "Evalix: Classification and prediction of job resource consumption on hpc platforms," in *Job Scheduling Strategies for Parallel Processing*, N. Desai and W. Cirne, Eds. Cham: Springer International Publishing, 2017, pp. 102–122.

[19] O. Aaziz, J. Cook, and M. Tanash, "Modeling expected application runtime for characterizing and assessing job performance," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2018, pp. 543–551.