

Stochastic HPC Queue Wait Time Predictions

Gordon Gibb

March 9, 2021

Contents

1	Introduction	1
1.1	Aims	1
1.2	Predicting Job Wait Time	2
1.3	Random effects	2
2	Generating Random Job Walltimes	2
2.1	Distribution of walltimes	2
2.2	Generating a Random Number from an Arbitrary Distribution	2
3	Models	4
3.1	K-Nearest Neighbours	4
3.1.1	Importance of Picking Good Test and Training Data	4
3.2	Slurm	6
4	Results So Far	6
4.1	KNN	6
4.2	Randomly Generated Queue Statistics	7
4.3	Predictions with Errors	7
5	Codes and their Usage	7
5.1	Getting Queue Data	11
5.2	Putting the Queue Data Into a Database	11
5.3	Generating Walltime Distributions	11
5.4	Generating Test/Training Data for the KNN	11
5.5	Predicting with the KNN	11

1 Introduction

1.1 Aims

We wish to predict the wait time for a job to be submitted to a HPC machine, such that we can use this information to decide which machine out of a pool of machines to submit a job to get the quickest turnaround. Ideally we also want the predicted wait time and a measure of confidence around this prediction (e.g. error bars).

1.2 Predicting Job Wait Time

When a job is submitted to a HPC machine, it is put into a queue, and will be run when resources are available and when the scheduler deems it suitable for this job to run. The start time is based on a number of criteria that may or may not be known to the submitter. Often the scheduler is optimised to maximise job throughput whilst minimising idle nodes, although it may have a bias towards smaller or larger jobs etc. This means that the queue does not typically obey a first-on first-off rule (FIFO).

When submitting a job, the submitter requests the amount of resources their job will use, typically the number of nodes/CPU's and the maximum amount of time their job will run for (the maximum walltime). They may also request a certain queue that their job is to be submitted to, e.g. a short/debug queue for small jobs that will only run for a few minutes, or a special queue for large jobs. This helps the scheduler decide when to run this job, and ensures the job has the resources it needs.

Typically the queue will contain many jobs. Some will be running, and some will be waiting until resources are available. This list of jobs in the queue (plus the user submitted job) represents the state of the queue, and ideally given this we should be able to predict the wait time of a job.

1.3 Random effects

Predicting the wait time is unfortunately complicated by a number of factors outwith our control. As mentioned previously, we may not know the full set of criteria the scheduler uses to determine when jobs will run, although with a suitable model we can account for this. Additionally, as the queue is not FIFO, jobs submitted after our job is submitted may affect our job's wait time. A further complication is that for each job in the queue, we know its maximum walltime, but in reality each job will have a walltime less than or equal to this. We therefore do not know exactly how long each job will take, and therefore we do not know the total work in the queue.

We can use this last problem to our advantage, however, as we can use this uncertainty in the amount of work in the queue to quantify the uncertainty on our predicted wait time. We can use a model trained on the actual walltimes of jobs as our prediction model. When we make a prediction, we generate a large number of possible queue states based on randomly chosen walltimes for the queued and running jobs, and for each possible state, run it through our model. We can then use the resulting distribution of predicted wait times to determine the expected (e.g. mean) wait time, and the error on this (standard deviation or similar).

At present, we have not considered jobs that could be submitted after our job is submitted!

2 Generating Random Job Walltimes

2.1 Distribution of walltimes

In order for our stochastic prediction method to work, we need to obtain a set of random walltimes for queued/running jobs, ideally following the same distribution of walltimes as jobs previously submitted to the queue have had. In order to do this, we consider the distribution of actual walltime to requested walltime for historical jobs. In the most simple sense, we can consider the distribution over all jobs. This is illustrated in the top panel of Figure 1. We can go further by determining the distribution for jobs within certain node ranges, as shown in the bottom panel of Figure 1. Beyond this (although this has not been done yet) we could also consider the distribution of actual to requested walltimes within certain walltime ranges (e.g. for jobs with requested walltimes $< 1\text{h}$). This would give a more accurate possible distribution for each given job, although for some edge cases there may not be a lot of jobs to draw a distribution from.

2.2 Generating a Random Number from an Arbitrary Distribution

Now we have distributions for the actual/requested walltimes, we want to be able to generate a random number that obeys this distribution. This can be achieved using the cumulative distribution function of the

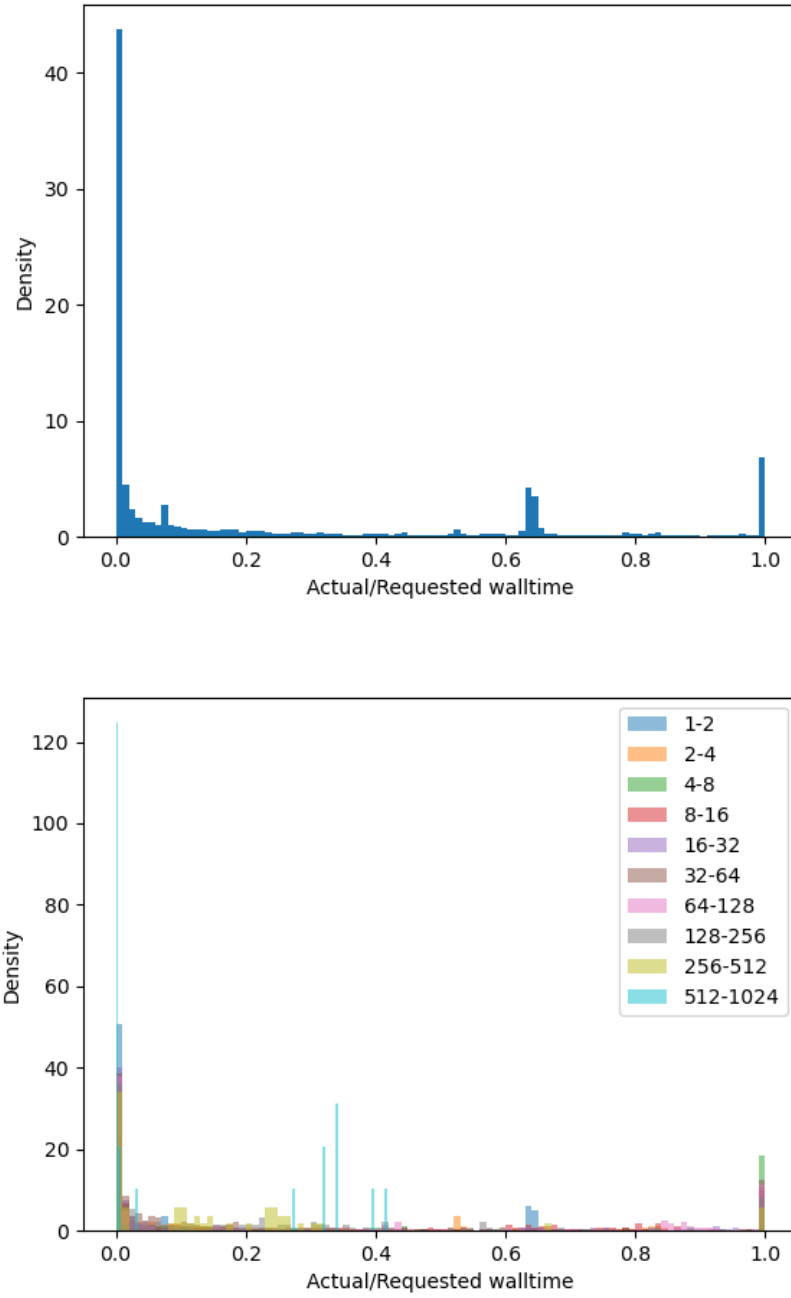


Figure 1: The distribution of actual/requested walltime for all jobs (top) and stratified by node count (bottom) submitted to ARCHER2 in Feb 2021.

distribution we wish to use. Given the probability density function (PDF), in our case the histogram, we can determine the cumulative distribution function (CDF) from

$$\text{CDF}(x) = \int_{x_{\min}}^x \text{PDF}(x')dx', \quad (1)$$

where the CDF ranges from $[0, 1]$. Using this, if we pick a uniform random number, x , between $[0, 1]$ then we can obtain a random number y that follows the PDF by using

$$y = \text{CDF}^{-1}(x). \quad (2)$$

We are thus able to generate random numbers that have the same distributions as the histograms shown in Figure 1. Using these random numbers, we can determine random walltimes by calculating

$$\text{Estimated walltime} = y \times \text{Requested walltime} \quad (3)$$

3 Models

3.1 K-Nearest Neighbours

Previous work looking at predicting queue wait times have found success using K-nearest neighbours (KNN) [1, 2, 3, 4]. To do this need to represent the state of the queue and the submitted job’s properties in the form of a vector of features. We then compare this vector to the vector of all historical jobs, and take the mean of the k nearest (based on some distance metric) vectors’ actual wait times to be the predicted wait time. The values within the vector we chose to use are displayed in Table 1. For the histogram features, the bins are chosen to that over all jobs in the data, each bin contains approximately the same number of jobs.

Once the feature vectors have been determined, we can normalise each element, such that they all have approximately equal range. For each element, we calculate the mean and standard deviation for this element for all jobs, and adjust it such that

$$f_i \rightarrow \frac{f_i - \langle f_i \rangle}{\text{STD}(f_i)}, \quad (4)$$

e.g. it is centered around zero, with a standard deviation of 1.

We can then optionally add weights to each vector to increase/decrease its importance. The weight of each element can be arbitrarily chosen, or can be the absolute value of the correlation coefficient of the feature with the wait time. Given a weight, W_i , for each feature, f_i , we apply the weight by setting

$$f_i \rightarrow \frac{f_i}{W_i}. \quad (5)$$

To test the model’s accuracy, we first use test data queue features (Table 1) derived from the jobs’ actual walltimes, as this is what the training data uses. Once we have done this, and seen how the model works for the “correct” queue features, we can then run the model using test data with queue features generated from random walltimes.

3.1.1 Importance of Picking Good Test and Training Data

Due to the nature of the queues, whereby from submission to submission the queue state has not changed much, if we were to split the test and training data at random (e.g. every n^{th} job is a test job) then we get an artificially good prediction, as this job’s neighbours (with near identical queue features and likely a similar wait time) are in the training set and thus are used for the prediction. Instead we need to ensure that test and training data are separated in time. We have done this by using jobs submitted every n^{th} day as a test jobs. In this case, overlap of the queue state between test and training data is only found at around midnight. The affects of test-train choice are show in Figure 2.

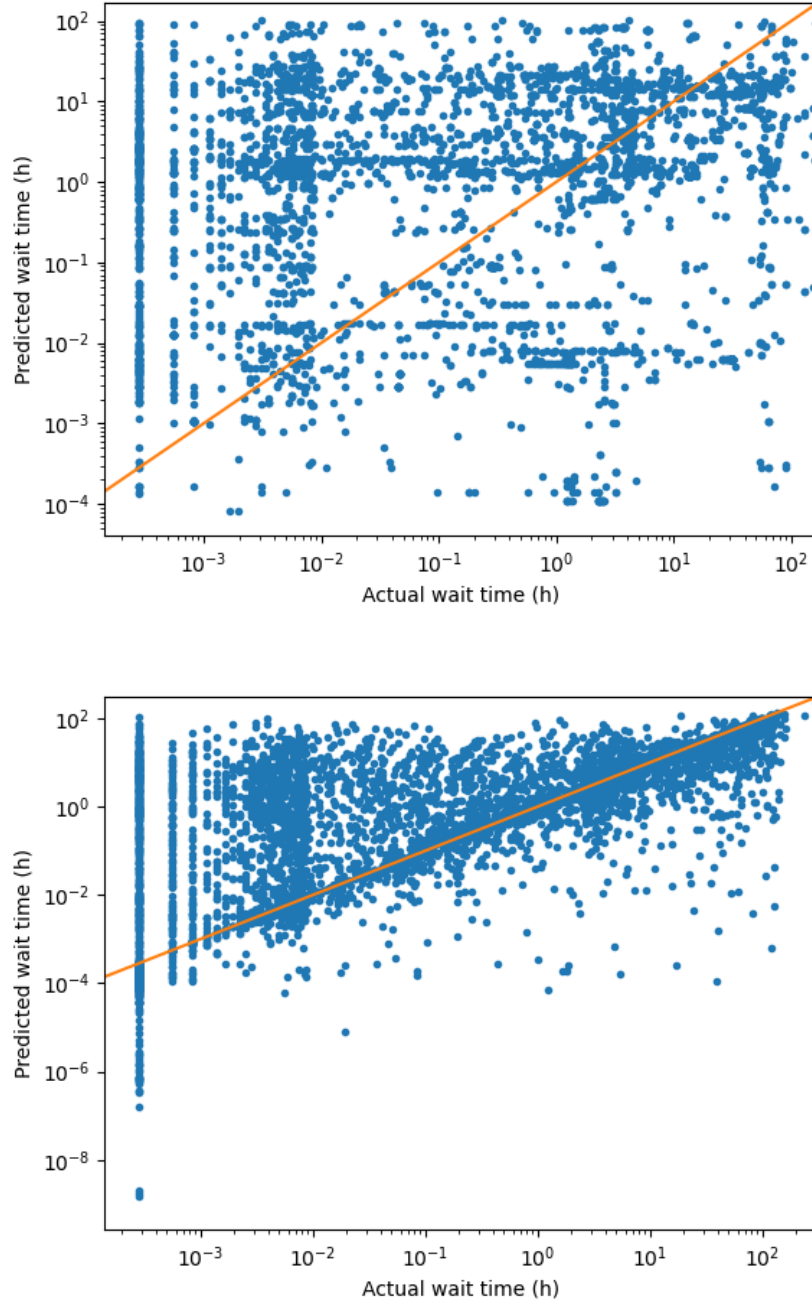


Figure 2: Splitting test/train data up by day (top) and randomly (bottom). Splitting test and training data randomly gives better predictions, but the accuracy is a result of test and training jobs that are close to each other in time having near-identical queue states. This accuracy is therefore due to bad choice of test/training data, and over-emphasises the accuracy of the model.

Name	Description
nodes_req	Number of nodes requested by the job
req_wtime	The requested walltime (hours)
day	Day of the week (0-6)
hour	Hour of the day (0-23)
s_q_jobs	The number of queued jobs
s_q_nodes	The total number of nodes requested by the queued jobs
s_q_work*	The total work (nodes \times walltime) of the queued jobs
m_q_wait	The median time queued jobs have been waiting for to run (hours)
d_q_nodes[0-7]	Histogram of the nodes requested by queued jobs (8 values)
d_q_work[0-7]*	Histogram of work requested by queued jobs (8 values)
d_q_wait[0-7]	Histogram of wait times for queued jobs (8 values)
s_r_jobs	The number of running jobs
s_r_nodes	The total number of nodes requested by the running jobs
s_r_work*	The total remaining work (nodes \times remaining time) of the running jobs
d_r_nodes[0-7]	Histogram of the nodes requested by running jobs (8 values)
d_r_work[0-7]*	Histogram of remaining work of running jobs (8 values)
d_r_remain[0-7]*	Histogram of remaining times for running jobs (8 values)

Table 1: The features used in the KNN model. Features with asterisks are calculated using actual walltimes for historical data and for model testing. For prediction, randomly drawn walltimes (Section 2) are used to calculate these.

3.2 Slurm

An idea I have had is to use the slurm scheduler in the place of the prediction algorithm. We somehow feed the lists of jobs in the queue into this and it can “schedule” all the jobs in the queue and hence predict the run time.

4 Results So Far

4.1 KNN

Unfortunately the KNN method does not seem to produce very good predictions. The top panel of Figure 2 shows the actual wait times (x -axis) against the predicted wait times (y -axis) for ARCHER2. I have tried weighting the features by their correlations with wait time, removing some features from the model, and several different numbers of neighbours with which to make the prediction, yet still the model is poor at making these predictions.

There are two possibilities (that I can think of) for why the model fails:

- The features used (Table 1) do not adequately represent the queue state
- There is insufficient data, and therefore the queue state of a job to be predicted is not close enough to the queue state of any of the training jobs

The first possibility, whilst seeming likely, is contradicted by the various other KNN-based models that claim success using similar features to those I use. None of these works outline how they split their test and training data up, so perhaps they have split this data up poorly.

The second possibility provides a compelling argument for ARCHER2, for which there is around only one month of useful data. Cirrus, however, has well over 6 months’ data, and its wait times are equally badly predicted. On Cirrus however, most jobs request the maximum walltime (4 days), a single node, and typically only last for a few seconds. This means that the queue state is not very well differentiated

Confidence Interval	Proportion of jobs
1 hour	33%
2 hours	47%
6 hours	63%
12 hours	73%
24 hours	86%

Table 2: Proportion of predicted jobs correct to within given confidence intervals.

between different jobs, and hence the KNN method may be unable to select appropriate jobs for an accurate prediction.

4.2 Randomly Generated Queue Statistics

Comparing the features based on actual walltimes and those based on randomly generated ones, there is some discrepancy. Figure 3 shows the actual vs mean (based on 100 random samples) for the sum of queued work. Similarly, Figure 4 shows the same but for the remaining work in the queue. Both figures show that although the actual and randomly generated are correlated, the randomly generated values consistently underestimate the actual value by up to several standard deviations. Table 2 displays the proportion of jobs whose prediction is accurate to within given confidence intervals. As can be seen, fewer than half of the jobs are predicted accurately to within 2 hours. To put this in context, 65% of the jobs’ actual wait times are less than 1 hour.

One possible reason for this that I can think of is that when we determined the distributions to draw from, we only used different distributions for different numbers of nodes requested. We did not consider the requested walltime. Given that a lot of users select the maximum allowed walltime regardless of how long they believe they job will run for, this means that jobs that request the maximum walltime likely have a large range of possible actual walltimes. Meanwhile, users who request something other than this likely have a good idea of how long their job will actually run for, and therefore the range of actual walltimes is likely more constrained. Going forward, it may be an idea to further categorise jobs by requested node and requested walltime when picking a distribution to draw a random walltime from.

4.3 Predictions with Errors

Despite the model not giving accurate predictions, and the randomly generated queue features not matching the actual queue features, we could go ahead and predict queue wait times for all the jobs with randomly selected queue features, and from them generate the mean predicted wait time, and the error on this. To do this we used 100 randomly generated queue features for each job.

Unsurprisingly, given the poor results from the predictions on actual queue data and the failures of the randomly generated queue features, the predictions using this method are bad. Figure 5 shows the scatter plot of actual wait times against the mean predicted wait time for the 100 versions of each job’s queue state. Table 3 shows the proportions of correct results for varying confidence intervals.

5 Codes and their Usage

I’ll now outline the codes written to do the calculations. The requirements are python3, numpy, matplotlib, pony (object relational matter) and scikit learn. The queue data is stored in a sqlite database, which is accessed in python through the pony object relational mapper. The database structure is described in `jobs.py`.

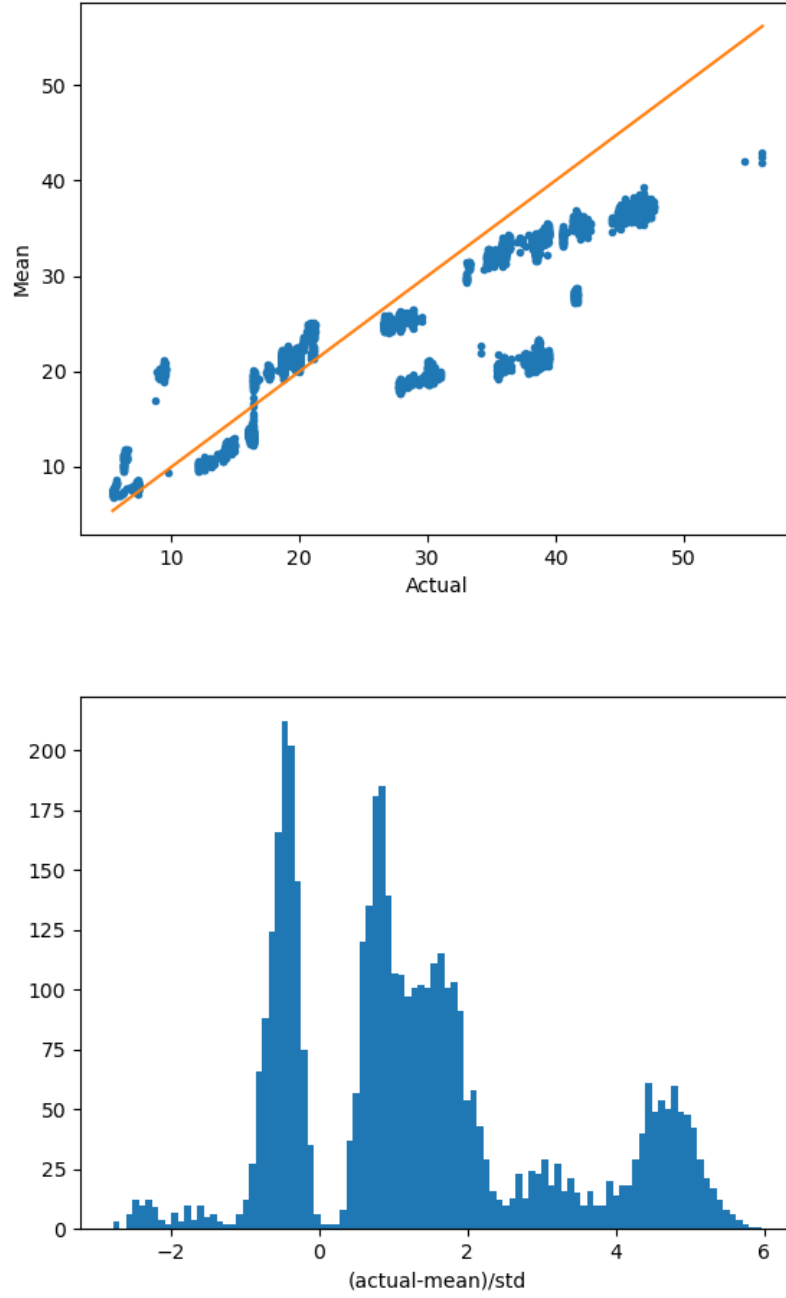


Figure 3: Actual vs randomly generated total work for queued jobs. Top: The actual values plotted against the mean of 100 randomly generated values. Bottom: The fractional error, $(\text{actual} - \text{mean}) / \text{std}$, of the same values.

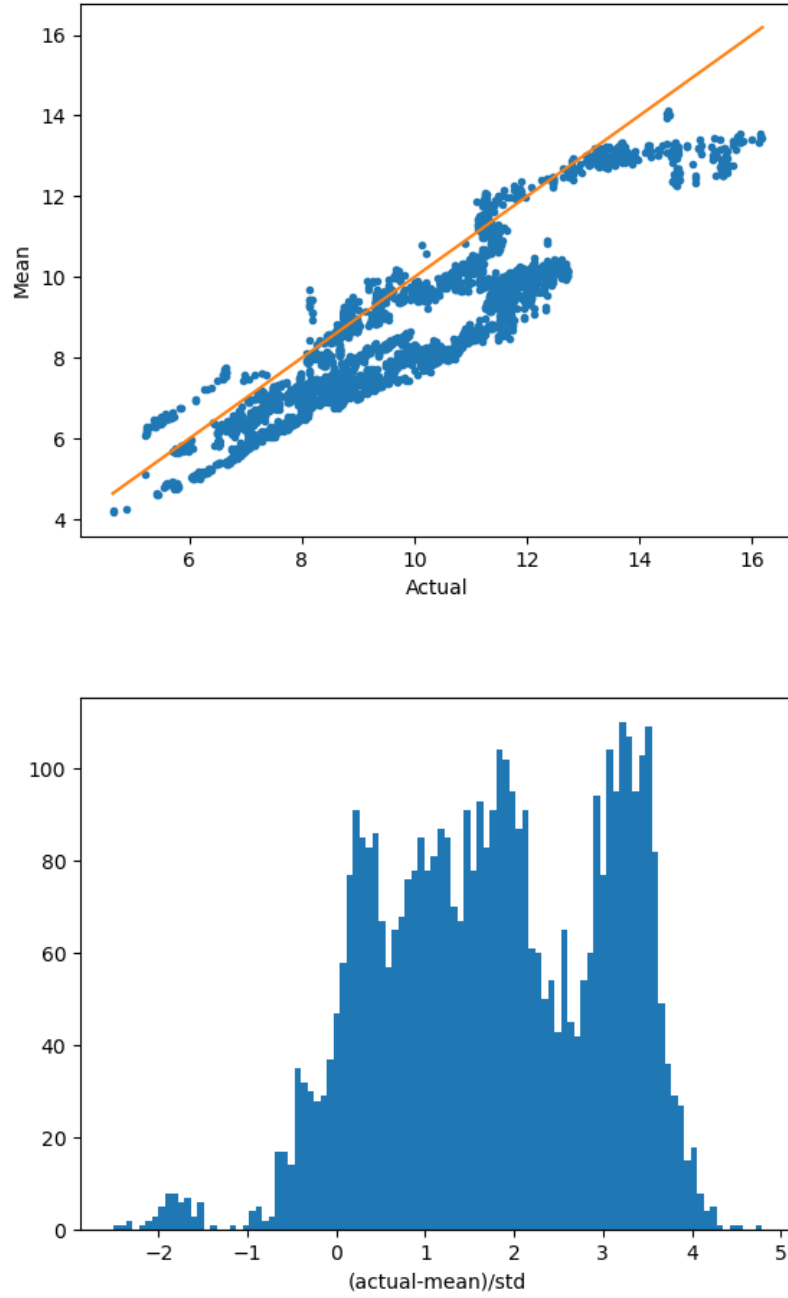


Figure 4: Actual vs randomly generated remaining work for running jobs. Top: The actual values plotted against the mean of 100 randomly generated values. Bottom: The fractional error, $(\text{actual}-\text{mean})/\text{standard deviation}$, of the same values.

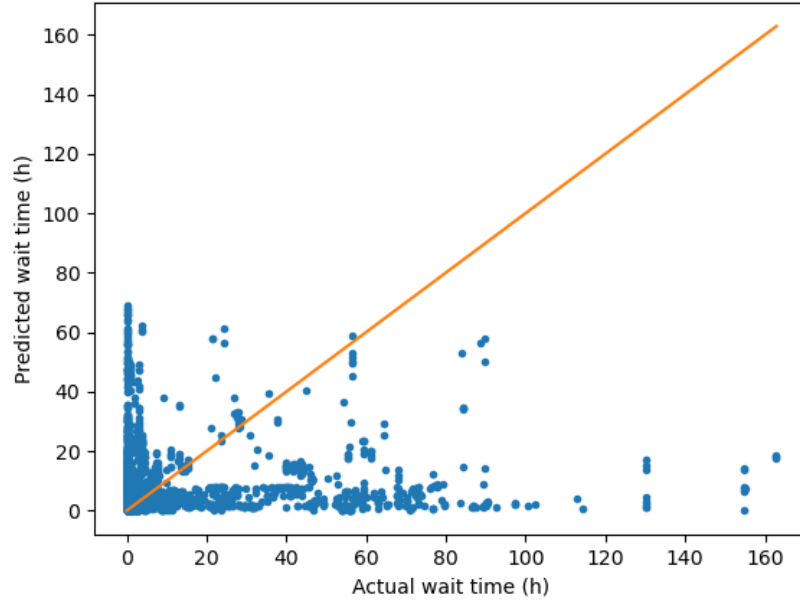


Figure 5: Actual wait times vs mean predicted wait times using 100 randomly generated queue states per job.

Confidence Interval	Proportion of jobs
1- σ	47%
2- σ	70%
3- σ	78%
4- σ	82%
5- σ	85%
1 hour	24%
2 hours	40%
6 hours	68%
12 hours	81%
24 hours	88%

Table 3: Proportion of predicted jobs correct to within given confidence intervals.

5.1 Getting Queue Data

Queue data is obtained directly from Slurm. The script `get_slurm_data.py` is run on the login node of the HPC machine, and runs a Slurm, command to collect all the jobs in the scheduler’s history. You can specify the start and end date for querying the queue records. It’s set up to generate one text file per month so as to reduce the size of each query Slurm has to do under the hood. This command works on ARCHER2 and Cirrus, but some machines may restrict this data. The files produced are of the form `[machine]_Queue_[start_date]_[stop_date].dat`, e.g. `ARCHER2_Queue_2021-02-01_2021-03-01.dat`. *This script requires python3. On ARCHER2 (which still uses python2 by default... no comment...) remember to run it with python3!*

5.2 Putting the Queue Data Into a Database

Once the queue data has been extracted from Slurm, copy the files onto the machine to be used for prediction. Edit some settings at the top of `to_db.py` to specify the name of the database we wish to create, the number of CPUs per node of the machine, and the names of the queue files produced above. Then run this script to generate the database for this machine’s queues.

5.3 Generating Walltime Distributions

The script `CDFs.py` contains the object `CDF` that can be used to generate random walltimes. The database it uses, along with the machine queue to select jobs from is set at the top of the file. If this script is run by itself it generates the cumulative distribution functions it needs to generate the random numbers, saving them to `[machine]_[queue_name]_CDF.npz`. If it is imported by another script, `CDF` can be used to generate the random numbers.

5.4 Generating Test/Training Data for the KNN

The script `generate_testtrain.py` generates the test and training data. The bins used for generating the histogram queue features need to be specified at the top of this file. `get_bins.py` can be run to determine values for the bin edges.

For each job in the selected queue, `generate_testtrain.py` constructs a list of the running and queued jobs in the queue at its submit time, and generates the queue features for it as shown in Table 1. For jobs selected as training jobs, these are written to `train_all.csv`. For test jobs, the queue features generated using the actual walltimes are put into `train_all.csv`. For the training jobs 100 sets of queue features generated with random walltimes are generated and written to `test_all_100.csv`. *Note: This script can take a few hours to run*

5.5 Predicting with the KNN

Prediction is done with `knn.py`. This selects the features wanted, normalises them, weights them according to the Spearman correlation coefficient, then predicts using sklearn’s `KNeighborsRegressor`. Firstly it predicts for the test jobs using the actual walltimes (from `train_all.csv`). This typically takes a few seconds to predict. It then predicts for the test jobs with randomly generated queue features (from `test_all_100.csv`). This takes around one hundred times longer. It then produces graphs like those shown in Figure 2

References

- [1] Renato L.F. Cunha, Eduardo R. Rodrigues, Leonardo P. Tizzei, and Marco A.S. Netto. Job placement advisor based on turnaround predictions for hpc hybrid clouds. *Future Generation Computer Systems*, 67:35–46, 2017.

- [2] Rajath Kumar and Sathish Vadhiyar. Identifying quick starters: Towards an integrated framework for efficient predictions of queue waiting times of batch parallel jobs. In Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 196–215, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [3] Rajath Kumar and Sathish Vadhiyar. Prediction of queue waiting times for metascheduling on parallel batch systems. pages 108–128, 05 2014.
- [4] Prakash Murali and Sathish Vadhiyar. Qespera: an adaptive framework for prediction of queue waiting times in supercomputer systems. *Concurrency and Computation: Practice and Experience*, 28(9):2685–2710, 2016.