Towards Practical, Generalizable Machine-Learning Training Pipelines to build Regression Models for Predicting Application Resource Needs on HPC Systems

***Description: Pipeline to generate and create Training Data.***

Automated Data Collection and Model Training: Each target application was incorporated into the framework and treated as a black-box that exposes tunable parameters through which their behavior was modified systematically in order to generate the training data. This training data was then pipelined into the model training. We used the Cheetah Experiment Harness and Campaign Management System. Cheetah is an experiment harness designed as part of the Exascale Computing Project (ECP)’s Co-design Center for Online Data Analysis and Reduction (CODAR) project, that “allows the systematic conducting of parametric studies to explore various aspects of online analysis and reduction of scientific data”. Essentially, Cheetah enables running a target application with different application and system configurations within a single job. A Cheetah campaign file was used to configure the parameters against which the target application was run and its outcomes such as its run time, memory requirements, and application-specific output was analyzed. Cheetah’s runtime module, Savannah converts the meta-data provided in the campaign files into execution configurations and runs them on the target system. Each experiment is run in a designated target folder within a “workspace endpoint”, and then profiled. A post-execution script is used to gather the identified features (that we designated as relevant in predicting execution time).

The Tuning and Analysis Utilities (TAU) profiler was used within Cheetah to capture the runtime (i.e. dynamic) features that capture the performance-related information of a target application. TAU is compatible with and tested against the Linux-based operating systems and used in our experiments to capture profiling information on our target applications which were written in C, JAVA, or Python. Among other information, the TAU profiler captures port or memory-mapped IO read/write bandwidths along with maximum bytes written/read per execution. Pytools was used to capture static profiling information such as CPU cores, memory, and other environment-related features. Figure 2 shows the complete workflow of the framework. As the first step, we identify the different configurations under which we want to profile a target application. Once the execution hyper-parameters are identified, we run the application against all these configurations using Cheetah and profile them using TAU, which results in the generation of our training data. We then ran a correlation analysis between the captured features and the execution time in order to identify and eliminate similar or irrelevant features. Then we applied principal component analysis (PCA) to reduce the feature dimensions. Finally, we developed our models; we developed two classes of models to evaluate, with both being off-the-shelf regression models, viz. A decision tree regressor (DTR) and a simple 1-hidden layer feed-forward neural (1HLNN) network.

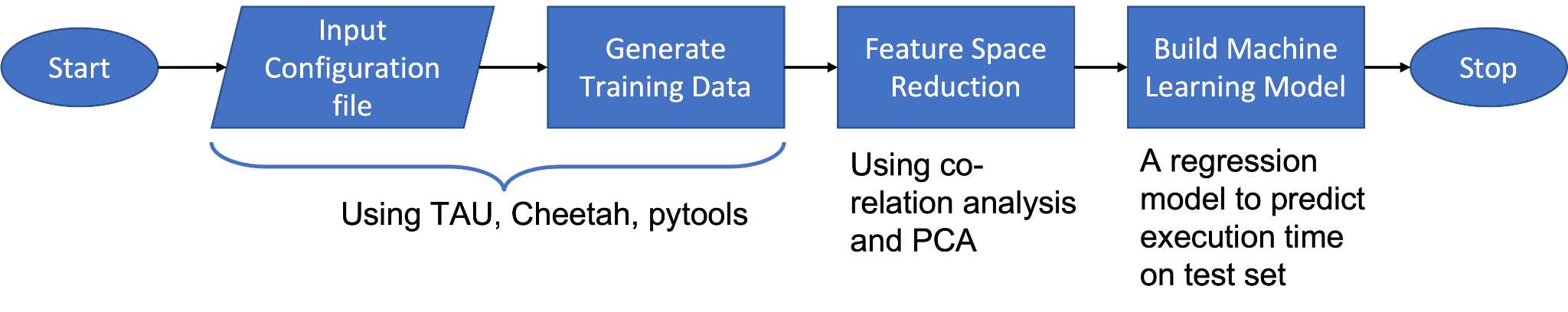


Figure 2: The pipeline that establishes the workflow from generating training data to building the prediction model.

Target Applications: We next describe our three target applications. These applications spanned a range of behavior and had a different portfolio of resource needs.

*Gray Scott Simulator*: This application [REF] is a simulator for chemical diffusion (based on Gray Scott equations [CITE]). Input to this application includes the two chemical substances that react with one another, their densities, feed and kill rate, plot gap, as well as the number of time-steps the reaction is to be simulated. We ran this application in multi-processor mode using *mpirun*. This application is both computationally and I/O-intensive.

*Trimmomatic*: This is a software tool used to clean raw Illumina[CITE] sequence raw archives (SRA) reads [CITE] by trimming the “noisy” ends, and removing low-quality reads and the synthetically added adapters. Cleaning is an important step in genome sequencing necessary to achieve a higher quality of assembly, especially from older SRAs that have higher impurities. To give a sense of the scale of this application, note that there could be a few thousand to a few billions of tiny base sequences read from a DNA sample within an SRA. Trimmomatic reads these sequence-quality pairs, trims and crops each pair and writes the trimmed sequence-quality pair to an output file. This application is both computionally and I/O intensive. <Add watervole read sizes>

*DNNs for Image Classification*: This was a family of applications used to classify images by age group. These applications read the entire training data into the memory in order to train the target models on a single node. Each image in the training data has a size of 244x244 pixels and is stored in terms of its RGB values. Hence each image is stored as a 244x244x3 matrix in the dataframe. This family of applications comprised three pre-defined TensorFlow image classification models - VGG16, ResNet50 and InceptionV3. Each of these models have a few thousand learnable parameters. Based on the number of images and batch size, the memory requirements of these models change. The goal of the classification is to assign an age label 0 for ages 0-10 ages, 1 for ages 10-20, and so on to 9 for 90-100 ages) to the target image. All of these models are both computationally and memory intensive.

Model Features: Features that impact the execution time of an application fall into two categories, namely, system and execution environment-specific features. We term these “system-specific” features. These features are collected for all target applications. System-specific features could be static, such as the number of cores on a node, or dynamic, such as the network or I/O bandwidth consumption of the target application. Features could also be “application-specific”. These application-specific features are the tunable parameters, and characteristics of the input data for the application, specifically those characteristics that impact the execution time. Figure 3 shows this classification of features.

Static System-Specific Features describe the execution environment and are independent of the target application. Dynamic features describe the runtime behavior of an application within the execution environment. These features are listed below:

* Static: Number of physical cores (numerical), Total number of cores (Physical cores\* no. of ports) (numerical), Clock speed (numerical), RAM (numerical), Swap size (numerical), Is GPU? (ordinal), Processor type (32 or 64 bit, numerical).
* Dynamic: Maximum read/write rate: (Numerical), Bytes read/written: (numerical).

Each application has its own set of application-specific features that describe the configurable hyperparameters and the input data. These features change with the target application. Application-specific features for each of our target applications are shown below:

|  |  |
| --- | --- |
| **Feature Category** | **Feature: Description [Datatype]** |
|
| Gray-Scott | |
| input | L: Size of the global matrix which is LxLxL cube [Numerical]  Du: Diffusion coefficient of U [Numerical]  Dv: Diffusion coefficient of V [Numerical]  F: Feed rate of U [Numerical]  k: Kill rate of V [Numerical] |
| configuration | dt: the timestep in between two reaction steps [Numerical]  steps: total no. of steps [Numerical]  ​​plotgap: number of steps to take before logging the reaction states [Numerical]  noise: Amount of noise to be induced in the reaction [Numerical] |
| Trimmomatic | |
| input | file type: The SRA could either be the single-end or paired-end reads (the single end is one direction read of the gene sequence; the paired-end is a bi-directional read of the gene sequence in gene) [Categorical]  file\_size: The size of the entire SRA file(s). [Numerical] |
| configuration | adapter: Number of synthetic adapter sequences that are induced as noise  seed mismatches: Number of allowed seed mismatches in the adapter noise clipping/trimming [Numerical]  palindrome clip or simple clip threshold: threshold for paired-end or single-end adapter trimming [Ordinal]  leading or trailing quality: The quality needed for the start and end of each read sequence in SRA. [Ordinal]  minimum size: The minimum size of the read that is needed to keep it in the output file. [Numerical]  maximum information & strictness index: The maximum information in terms of the required quality to maintain the strictness score. [Numerical and Ordinal]  window size & quality: The minimum quality of the bases on the window is needed to retain them in the output. [Numerical and Ordinal] |
| DNN Models | |
| input | Number of images: total number of images in the training data [Numerical]  Image dimensions: the size of the image in terms of pixels and channels [Numerical] |
| configuration | Type of TensorFlow model - We tried three different pre-defined models - VGG16, InceptionV3, ResNet50 [Categorical]  batch size - no. of images to be trained in one training step as a batch. [Numerical]  epochs- no. of times the entire training dataset should be processed during training phase [Numerical]  learning rate- the rate at which the weights are learned per straining step. [Numerical] |

Execution environments: Our experiments were run on the following systems:

* Linux VM - 1 Node: Processor: 2.3 GHz 8-Core Intel Core i9, Memory: 16 GB 2400 MHz DDR4. VM: Cores-1, 2, 4 Memory - 2GB, 4GB
* HPC:
  + **Owens (CPU):** Dell PowerEdge C6320 two-socket servers with Intel Xeon E5-2680 v4 (Broadwell, 14 cores, 2.40GHz) 28 processors, 128GB memory
  + **Owens (GPU):** Dual Intel Xeon 6148s Skylakes 40 cores per node @ 2.4GHz, 192GB memory
  + **Pitzer (CPU):** Dual Intel Xeon 8268s Cascade Lakes 48 cores per node @ 2.9GHz 192GB memory
  + **Pitzer (GPU):** Dual Intel Xeon 8268s - Dual NVIDIA Volta V100 w/32GB GPU memory 48 cores per node @ 2.9GHz, 384GB memory

Dataset Generation: Our goal with respect to data generation was to explore the tractability of the model building process. To motivate this need, note that if (for example) training a deep neural network model such as VGG16 on 10,000 images on two different environments, it would take an estimated 9 hours on the Owens-CPU node at OSC. In order to generate just a 100 training samples for a CPU as well as a GPU node on Owens for each set of application-specific values (such as batch-size or epochs), it would take an estimated 950 hours. Thus, we sought to assess to what extent a model built from data collected from a small set of runs of a single application on an easily accessible, simple environment (such as a workstation) could then be extrapolated or generalized. To this end, we generated our “scaled-down” (SD) models on workstations using a limited set of training samples generated from smaller inputs (for example, to train the scaled-down VGG16 model we used just 200 images). Next, we noted that as an application is used in production, more runtime data becomes available. To mimic this, we then expanded SD by adding training instances generated from running the application on production-scale data on the target environment. We call this extended dataset the *full-scale* (FS) dataset. Essentially, the goal is to see how a regression model can learn to scale both before and as “real” data becomes available.

We generated two sets of data (SD and FS) for each application. These data sets were produced as follows:

* Gray-Scott Simulator: The entire dataset was generated on a Linux workstation. For scaled-down samples, we kept the matrix size < 100 and step-size < 1000. For full-scale training data generation, we kept the matrix dimension at 128 with a step-size of 1000 (these values are closer to real simulations).
* Trimmomatic: The scaled-down (SD) dataset was generated on a Linux VM as well as a OSC Owens CPU node. The full-scale (FS) dataset was generated on OSC Owens CPU node. For the scaled-down dataset, we took a random set of 1K, 10K, 100K, and 1M sequence samples from one SRA. For the full-scale dataset, we used 10 complete SRAs (of size?). We ran both kinds of inputs through several trimming procedures, with a range of trimming hyperparameters.
* Deep Neural Networks for Image Classification: We ran both scaled-down and full-scale training generation on Owens CPU and Owens GPU nodes, since the models were too large to fit in the memory of a Linux VM on a workstation. We ran the three DNN models on 100, 200, 400, 1K, 2K, and 5K images for the scaled-down (SD) datasets. Full-scale (FS) training data consisted of 10K images.

Details of these datasets are in the table below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Application** | **Total Training Data** | **SD** | **SD Generation Time** | **FS** | **FS Generation time** | **Total Features** |
| Gray-Scott | 8486 | 8088 | 297223  37 | 398(~5%) | 90058  226 | 17 |
| Trimmomatic | 3362 | 3234 | 20085  6 | 128(~4%) | 78338  612 | 22 |
| Deep Neural Networks – Resnet50, InceptionV3, VGG16 - ALL | 1523 | 1313 | 166942  127 | 210(~14%) | 176660  841 | 15 |
| Deep Neural Networks – Resnet50, InceptionV3, VGG16 - EACH | 510 | 440 |  | 70(~14%) |  | 14 |

Regression models: We used two different regression models, as follows:

* One Hidden Layer Neural Network (1HLNN) built with the Adam optimizer, using mean square and mean absolute metrics for optimization using TensorFlow. This model was trained for 300 epochs with 20 as batch size, 0.01 learning rate, and 20% validation split (best-identified configuration).
* Decision Tree Regressor (DTR): We used the Scikit-learn [REF] off-the-shelf decision tree regressor model with default configurations.

4. Experiments and Results

To begin with, we used the Cheetah harness (see above) to generate dynamic system- and application-specific training data for a systematic sweep of application parameters and for two execution environments per application. We then split our generated datasets into training, validation and test sets. Next, we noted that under-estimated jobs incur early-termination and hence resubmission costs. Overestimated jobs (only) incur higher wait-times in job queues, but run till execution (at least on public scientific cyber infrastructures such as OSC). Therefore, we analyzed the distribution of under-predictions on the validation dataset to estimate a simple multiplier applied to the predictions would significantly mitigate under-predictions. We then applied this adjustment when we ran our models on the test set; that is, our results on the test set adjusted results. Specifically, we made predictions on the validation dataset to analyze the percentage error range of under-predictions. We then calculate a multiplier that when applied to the predicted value of each job will correct at least 50% of the underpredictions. This multiplier is our adjustment value for the test set. We call this the Underprediction Adjustment Multiplier (UPAM). In other words, each prediction on the test set is increased by UPAM times. This is the final prediction estimate of our model.

**Baselines**: We proposed two baselines to evaluate the scalability of the model with respect to the application input size and target environment. These baseline models differ in the type of training and test data chosen per application. We adjust each baseline prediction with respect to the validation set. This is our adjusted baseline (A<Baseline-Model>).

Baseline 1: Here we train, validate and test the model on each application’s scaled-down dataset (i.e. SD). In order to keep the scale relative, we use the same size of test set used in Baseline2. The baseline and adjusted baseline models are BaselineSD and ABaselineSD respectively.

Baseline 2: Here we train, validate and test the model on the appropriate full-scale dataset (i.e. FS). We split FS into a 50:25:25 training, validation, and test split. We denote the baseline and adjusted baseline models as BaselineFS and ABaselineFS respectively.

Metrics: We use the following measures to evaluate the results: <I will include the formula>

1. Mean Absolute Error (MAE): The mean of all the absolute differences between the predicted and actual execution times.
2. Percentage Error (PE): The difference in percentage between the predicted and actual execution times.
3. Under Prediction Percentage (UPP): The number of predictions that were under-estimated w.r.t to the actual execution time.
4. Mean Absolute Percentage Error for Under Predictions (MUPP): The mean of all the absolute difference percentages between the predicted and actual execution times.

We could not directly use regression metrics like Adjusted R-square or Root Mean Square Errors as they do not account for the under-prediction or over-prediction cost trade-offs.

**Model Selection:** When comparing two regression models, we selected the better-performing model based on the following heuristic criteria:

1. Pick the model with the lowest UPP and MAE, or
2. Pick the model whose distribution of under-prediction errors is relatively higher over lower error windows. Pick a model that satisfies the following constraint:

If Model A has the lowest MAE and Model B has the lowest UPP, then

1. Select Model A if the MAE of Model A is decreasing at least by the rate at which UPP increases w.r.t Model B
2. else choose Model B Example

The table below shows …

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 Layer Neural Network (1LNN) | | | Decision Tree Regressor  (DTR) | | |
| Type of Application | Baseline | MAE (secs) | UPP (%) | MUPP | MAE (secs) | UPP (%) | MUPP |
| Gray-Scott | BaselineSD | 8.08 | 43.56 | 16.19 | 10.08 | 21.78 | 39.87 |
| ABaselineSD | UPAP = 20 | | | UPAP = 10 | | |
| 11.48 | **16.83** | 17.08 | 13.84 | **15.84** | 48.54 |
| BaselineFS | 255.30 | 47.52 | 10.82 | 35.33 | 19.80 | 18.45 |
| ABaselineFS | UPAP = 20 | | | UPAP = 10 | | |
| 317.2 | **9.90** | 14.31 | 47.23 | **7.92** | 34.51 |
| Trimmomatic | BaselineSD | 1.84 | 51.28 | 19.51 | 1.53 | 51.28 | 29.11 |
| ABaselineSD | UPAP = 20 | | | UPAP = 30 | | |
| 2.24 | **17.94** | 25.07 | 2.27 | **17.94** | 47.83 |
| BaselineFS | 15.49 | 66.66 | 3.97 | 12.30 | 43.58 | 3.98 |
| ABaselineFS | UPAP = 10 | | | UPAP = 10 | | |
| 44.57 | **7.69** | 8.26 | 48.32 | **2.56** | 13.53 |
| Family of DNN | BaselineSD | 49.95 | 53.70 | 27.95 | 82.18 | 25.92 | 17.75 |
| ABaselineSD | UPAP = 30 | | | UPAP = 10 | | |
| 96.08 | **20.37** | 35.42 | 96.09 | **11.11** | 31.04 |
| BaselineFS | 223.02 | 55.55 | 22.34 | 178.17 | 35.18 | 19.37 |
| ABaselineFS | UPAP = 40 | | | UPAP = 10 | | |
| 337.37 | **27.77** | 14.17 | 226.90 | **14.81** | 33.73 |

**Observations:**

From figure X, we can see that more than 50% of the under-predicted jobs made by Baseline-1 (Baseline-SD and BaselineFS) models fall in this [-40, 0] percentage error for all the applications. So we choose to over-predict the execution time by a maximum of 40% UPAP based on the validation data distribution per application and baseline. From Table A, we observe that adjusting the predictions is reducing the no. of under-predictions on the test set.

All underpredictions within the UPAP adjustment window shift towards the over-predictions. For example, in Figure Y, for Trimmomatic with UPAP as 10%, we can see that all the (-10, 0] percentage error predictions add to the (0, 10] percentage errors. Since the UPAP is model and training data specific, and generated with the help of validation data, we don't see a linear shift of the predictions over the percentage error ranges. Instead, we notice that most of the pre-adjusted over-predictions with lower MAE of (0, 10] still remain in the same window.

We notice that the distribution of percentage errors is similar in both the baselines and adjusted baselines. So we can use the scaled-down down with/without a few samples from full-scale data to train the model that can scale w.r.t. application data and execution environment.

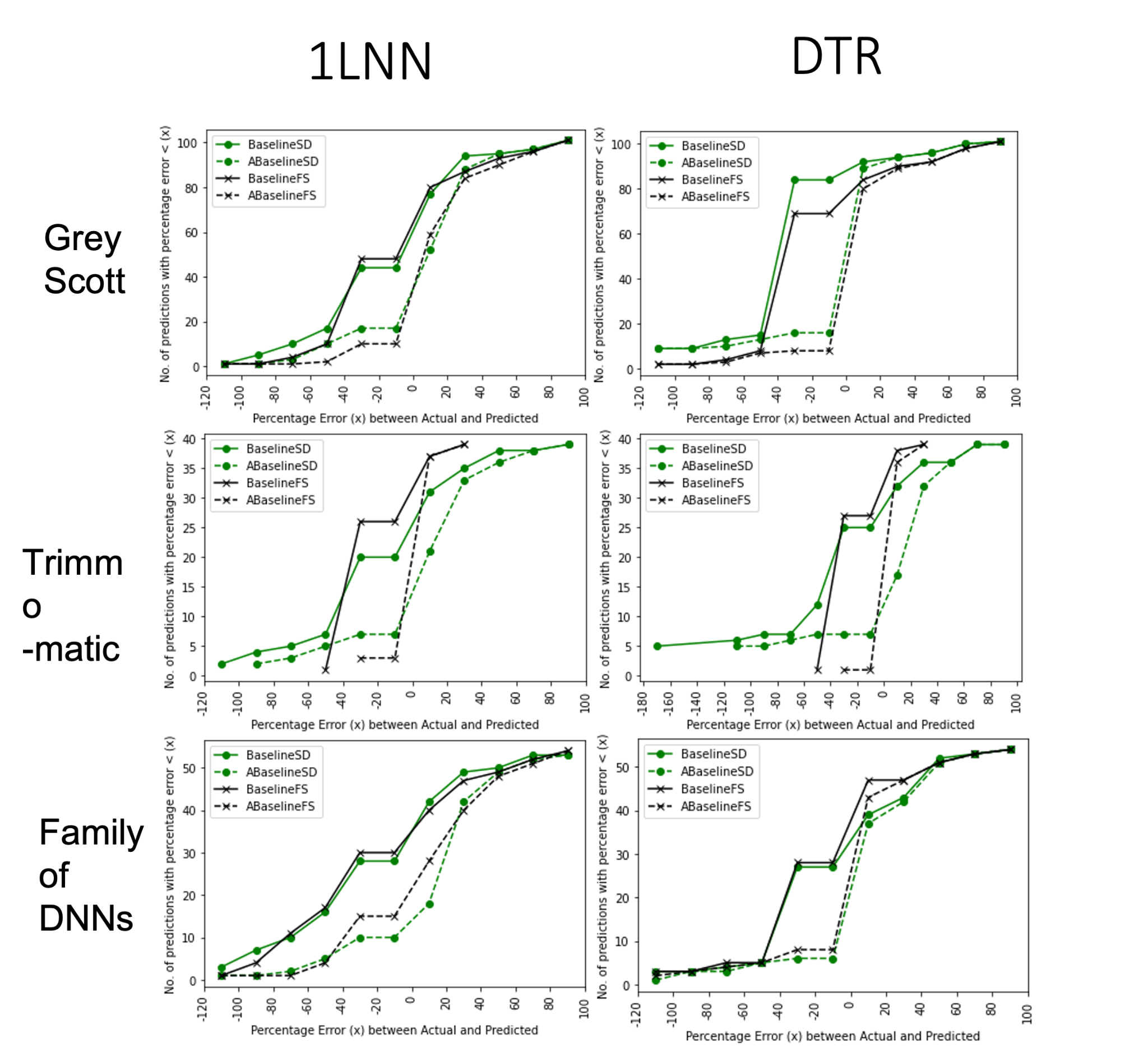


Figure Y: DTR: - Full-scale



Experiment 1: experiment to show the scalability of a trained model w.r.t. full-scale data.

We start training the model on the scaled-down data set and test it on full-scale data to see if the model is able to scale the predictions w.r.t. change in the target environment and increase in the size of application-specific input data. We call this model NoFS model as we do not include any full-scale data in training the model. We gradually augment the scaled-down data with samples from full-scale data and train new models. We build two other models after including different no. of samples from full-scale data. We test these models on left-out full-scale data after making the adjustment on left-out validation full-scale data. We call these models 25%FS and 50%FS as we include 25% and 50% of full-scale samples in the scaled-down training data.

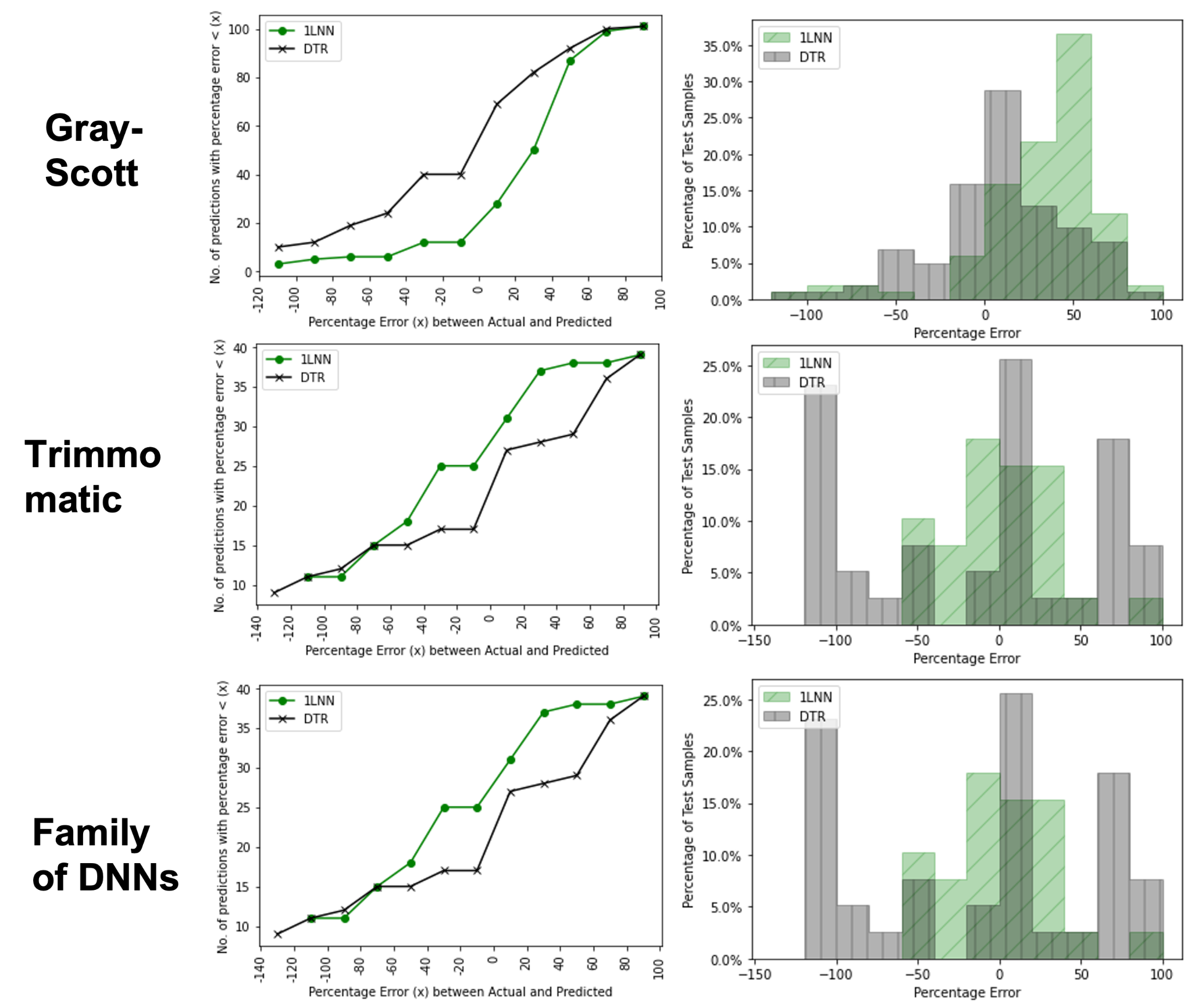
|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | 1 Layer Neural Network (1LNN) | | | Decision Tree Regressor  (DTR) | | |
| Application | Type of training data | MAE (secs) | UPP | MUPP | MAE (secs) | UPP | MUPP |
| Gray-Scott | Baseline | 317.2 | 9.90 | 14.31 | **47.23** | **7.92** | **34.51** |
| NoFS | 94.16 | 11.88 | 25.68 | 58.89 | 39.60 | 29.39 |
| 25%FS | **82.78** | **17.82** | **17.28** | 413.05 | 30.69 | 35.92 |
| 50%FS\* | **67.61** | **3.96** | **27.12** | 267.78 | 16.83 | 52.87 |
| Trimmomatic | Baseline | 44.57 | 7.69 | 8.26 | **48.32** | **2.56** | **13.53** |
| NoFS | 225.64 | 64.10 | 41.61 | 1125.13 | 43.58 | 42.85 |
| 25%FS | 305.07 | 64.10 | 59.94 | **61.77** | **7.69** | **40.81** |
| 50%FS\* | 435.88 | 61.53 | 50.98 | **46.86** | **5.12** | **12.69** |
| Family of DNN | Baseline | 337.37 | 27.77 | 14.17 | **226.90** | **14.81** | **33.73** |
| NoFS | 708.43 | 1.85 | 4.76 | 922.70 | 12.96 | 24.24 |
| 25%FS\* | **292.42** | **11.11** | **22.57** | 1811.05 | 9.25 | 50.68 |
| 50%FS | **511.30** | **5.55** | **19.07** | 599.84 | 14.81 | 56.94 |

[[add 1 and 2 here]

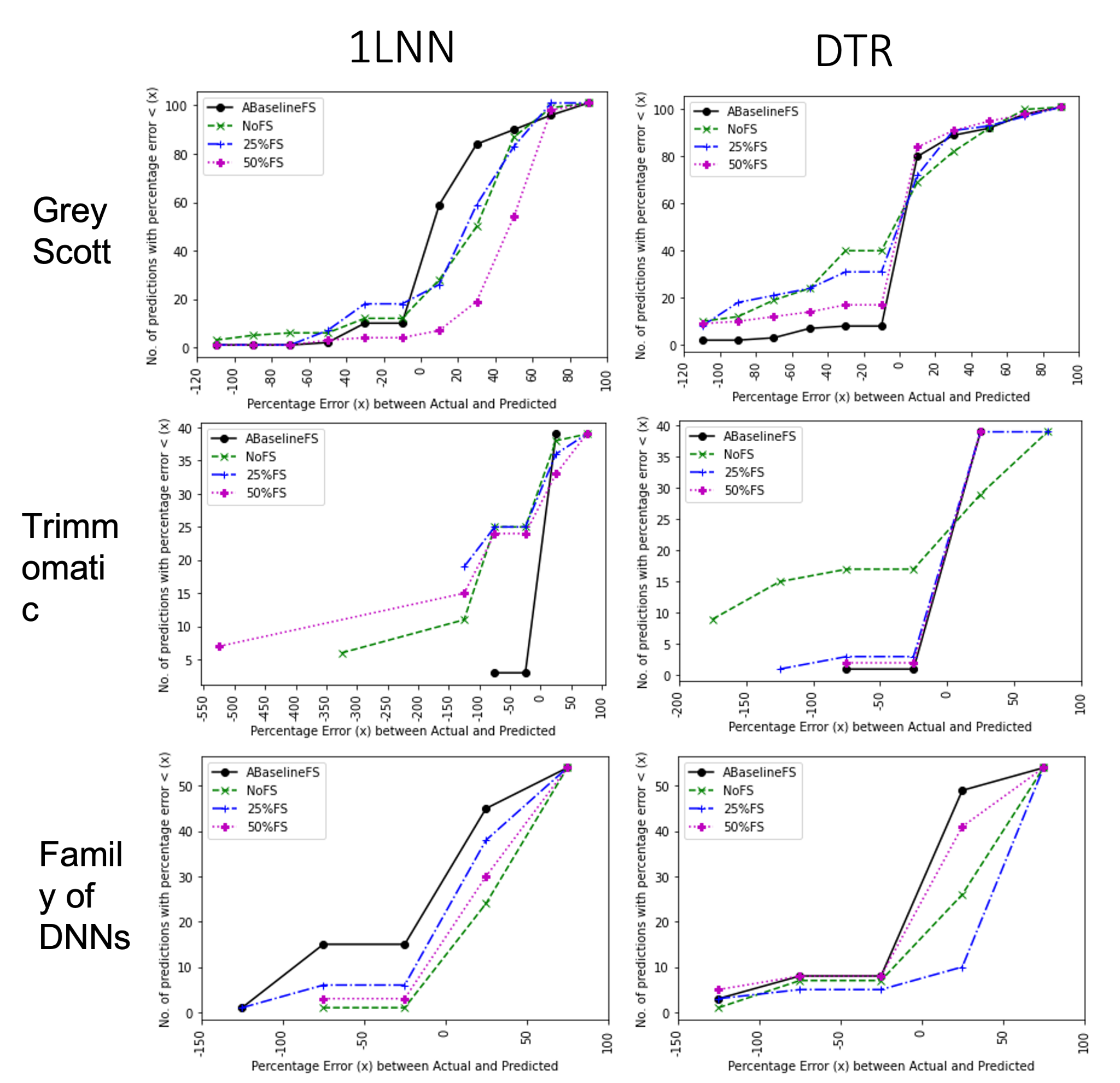
Observations:

The choice of the predictive model depends on the availability of training data.

1. *If generating training data for a “specific target execution environment” is inexpensive and quick, then building a baseline model for that target application and the environment could be efficient*. In the case of Trimmomatic, the baseline system with 1LNN works better than DTR. But the cost of generating even a few samples of full-scale training data for Trimmomatic is expensive (takes an average of 100 times more time to generate one full-scale sample than its scaled-down equivalent)
2. *If generating the target environment-specific training data is expensive (or takes more time), we want to use scaled-down data to train a predictive model*. From figure XX, we can see that 1LNN scales better than DTR for all three applications based on the model selection criteria. <Since DTR needs a few samples from the full-scale to make a more precise prediction on the test set, it fails to perform in these situations>.



1. *If the cost of under-prediction is too high, then we can gradually add a few full-scale samples generated over a period of time to scaled-down data.* We append the full-scale data to the scaled-down data with 25% and 50% full-scale data and train the models. DTR performs better with the baseline (ABaselineFS) for all applications. But with training data 25%FS and 50%FS, DTR doesn't perform better for all applications. While Trimmoatic still performs better with DTR, Gray-Scott and Family of DNNs scale better with 1LNN. Similarly, each application’s needs for the amount of additional full-scale data to scale better differ w.r.t. regression model. 1LNN model scales better at 50%FS for Gray-Scott but scales better at 25%FS only for Family of DNNs. DTR scales better at 50%FS for Trimmomatic.



Experiment 2: Experiment to show the transferability of a model trained on one or more applications of a family w.r.t. A new application in this family. The three applications we choose for this analysis are VGG16, ResNet50, and Inception V3 from the Family of DNNs. We run the following modules to analyze the transferability of a model within the family of applications:

|  |  |
| --- | --- |
| Model | Training Data Source |
| 25%FS | Same as experiment 2, we use scaled-down and a few samples of full-scaled data generated for the target application to train the model. This is the baseline system for these experiments. |
| OneVsOne | We train the model on scaled-down and full-scaled samples of *one*  application and predict for the other applications in the family |
| One+VsOne | We train the model on scaled-down and full-scaled samples of *one* application and 25% of full-scale samples from the other (target) application to predict for the target application. |
| OneVsOthers | We train the model on scaled-down and/or full-scaled samples of *all* applications except the target and predict for the target application |
| One+VsOthers | We train the model on scaled-down and full-scaled samples of *all* applications except the target with 25% and 50% of full-scale samples from the target to predict for the target application |

UPAM Scores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Type of Model (Training data) | Target Application | | | | | |
| VGG16 | | ResNet50 | | InceptionV3 | |
| 1LNN | DTR | 1LNN | DTR | 1LNN | DTR |
| VGG16 | 40 | 30 | 0 | 330 | 0 | 110 |
| VGG16+25%FS | N/A | | 0 | 70 | 0 | 100 |
| ResNet50 | 110 | 140 | 10 | 260 | 280 | 360 |
| ResNet50+25%FS | 90 | 60 | N/A | | 190 | 140 |
| InceptionV3 | 500 | 270 | 10 | 60 | 10 | 140 |
| InceptionV3+25%FS | 140 | 40 | 10 | 70 | N/A | |
| OthersNoFS | *40* | 140 | *70* | 930 | 80 | 50 |
| OthersAll | **100** | 100 | 20 | 70 | 280 | 50 |
| Others+25%FS | 50 | 80 | **40** | 80 | **100** | 60 |
| Others+50%FS | 40 | 80 | 60 | 80 | 90 | 10 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Type of Model (Training data) | Target Application | | | | | | | | | | | |
| VGG16 | | | | ResNet50 | | | | InceptionV3 | | | |
| 1LNN | | DTR | | 1LNN | | DTR | | 1LNN | | DTR | |
| MAE | UP% | MAE | UP% | MAE | UP% | MAE | UP% | MAE | UP% | MAE | UP% |
| VGG16 | **1043** | **10.5** | **1272** | **26.31** | *1141* | *0.00* | 597 | 21.05 | 3283 | 0.00 | 305 | 52.63 |
| ResNet50 | 3224 | 5.26 | 253 | 21.05 | **1097** | **0.00** | **1546** | **10.52** | 1295 | 15.78 | 1197 | 5.26 |
| InceptionV3 | 51229 | 0.00 | 426 | 36.84 | 3075 | 21.05 | 245 | 36.84 | **1041** | **0.00** | **1058** | **0.00** |
| OthersNoFS | *700* | *5.26* | 1117 | 5.26 | *373* | *5.26* | 6130 | 0.00 | 812 | 26.32 | *917* | *0.00* |
| OthersAll | **951** | **0.00** | 242 | 31.57 | 256 | 31.57 | 747 | 26.31 | *1988* | *0.00* | *752* | *0.00* |
| Others+25%FS | 1399 | 10.52 | 1733 | 10.52 | **439** | **0.00** | 769.17 | 10.52 | **524** | **0.00** | 1246 | 5.2 |
| Others+50%FS | 371 | 15.78 | 727 | 15.78 | 506 | 0.0 | 606 | 10.52 | 222 | 31.57 | 485 | 10.52 |

In above table, the model vs model (VGG16 vs VGG16; RN50 vs RN50; IncV3 vs InsV3) **scores** indicate the baselines (BaelineFS) for the target application. We use the model selection criteira while choosing **between** the 1LHNN and DTR within a target application (scores) per source of training data. For example, for InsV3 target application, when we train on OthersAll, the model selection scritiea picks DTR over 1LHNN.

When we have some existing training data and models in a family of applications, we can achieve a resonable execution time prediction for a new application in family without including new training runs from target application. OtheraAll for all target applications perform better than the baseline with at least one regression model. For example, 1LHNN gets better predictiosns with OthersAll for VGG16 than it’s own baseline. Similarly 1LHNN and DTR work better for RN50 and InvV3 respectively. We can further improve these scores by adding newer training samples from the target application. Based on the the target application the number of additional training samples reuiqred to improve the score varies. The **UPAM** values help in deciding “how many additional target application training samples are required to improve the prediction accuracies?”. As we include more target application specific training samples into our model, if the UPAM value keeps decreasing, then we can continue adding more of these training samples. But if notice an increase in this value, we should stop including more of these samplse and exclude the newly added samples from training data (i.e. reverting to the previous model). For example, for VGG16 and InsV3, 1LHNN predictions improve if we keep adding more training samples from target application (going from Others25\%FS to Others\%50FS) beacuse the UPAM value keeps decresing. But for RN50, if we keep adding more target application training samples, the 1LHNN models accuracy drops (incasre in MSE) because of an increase in the UPAM value.

5. Conclusion and Future work

Through this work, we have explored the feasibility of building a framework that can generate training data for a given application. By utilizing an existing experimental harness called Cheetah, we were able to generate training data automatically and capture both application and system-specific features with minimal human intervention.

The proposed models were able to scale the predictions if either of the execution environment or application-specific configurations change. These models were able to transfer within a family. Along with these, when we try to apply a model generated on the family of experiments for a new application in the family, we need to include the adjustment percentage that can trip a model into predicting very high over-estimations.

We propose future work in three areas, as follows:

*Model selection*: From the results (Experiments 1 & 2), it is evident that choosing a regression model specific to the application doesn't depend on one metric (like MSE or UP%). We plan to evaluate other heuristics for model selection.

*Dealing with missing value*s: The current models predict execution time for the test data and not inference data. Since the inference data do not have runtime features like I/O bandwidth, we need to fill in the missing values for these features before predicting execution time; (b)

*Cost models*: To choose a regression model, we need to compute a cost metric that measures the trade-off between successful execution of an application due to over-estimation (expecting higher wait times) and failure to execute an application due to under-prediction.