Received October 14, 2021, accepted November 24, 2021, date of publication January 11, 2022, date of current version February 2, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3142240

Comparative Analysis of Machine Learning Models for Performance Prediction of the SPEC Benchmarks

ASHKAN TOUSI AND MIKEL LUJÁN
Department of Computer Science, The University of Manchester, Manchester M13 9PL, U.K.

Corresponding author: Ashkan Tousi (ashkan.tousi@gmail.com)

This work is supported by EPSRC grants EP/T026995/1 EnnCore and EP/N035127/1 Lambda. Mikel holds an Arm/RAEng Research Chair award and a Royal Society Wolfson Fellowship.

ABSTRACT Simulation-based performance prediction is cumbersome and time-consuming. An alternative approach is to consider supervised learning as a means of predicting the performance scores of Standard Performance Evaluation Corporation (SPEC) benchmarks. SPEC CPU2017 contains a public dataset of results obtained by executing 43 standardised performance benchmarks organised into 4 suites on various system configurations. This paper analyses the dataset and aims to answer the following questions: I) can we accurately predict the SPEC results based on the configurations provided in the dataset, without having to actually run the benchmarks? II) what are the most important hardware and software features? III) what are the best predictive models and hyperparameters, in terms of prediction error and time? and IV) can we predict the performance of future systems using the past data? We present how to prepare data, select features, tune hyperparameters and evaluate regression models based on Multi-Task Elastic-Net, Decision Tree, Random Forest, and Multi-Layer Perceptron neural networks estimators. Feature selection is performed in three steps: removing zero variance features, removing highly correlated features, and Recursive Feature Elimination based on different feature importance metrics: elastic-net coefficients, tree-based importance measures and Permutation Importance. We select the best models using grid search on the hyperparameter space, and finally, compare and evaluate the performance of the models. We show that tree-based models with the original 29 features provide accurate predictions with an average error of less than 4%. The average error of faster Decision Tree and Random Forest models with 10 features is still below 6% and 5% respectively.

INDEX TERMS Machine learning, performance analysis, predictive models, SPEC CPU2017, supervised learning.

I. INTRODUCTION

Using Machine Learning (ML) to improve system design and predict the performance of computer systems is an active research area [1]. System designers and engineers use prediction results to investigate the impact of configuration changes on system performance and make better design decisions. Vendors seek the best way to position their systems in the market before they are built. Thus, performance prediction of upcoming system configurations is a demanding task [2]. Moreover, consumers try to improve the cost-performance ratio by searching for the best configurations to optimise performance of their systems, or make rational purchasing decisions. Having access to performance results of various workloads, even on large collections of computer systems is not enough, as consumers may require the performance data for new systems or unseen configurations [3].

These challenges motivate the study of performance prediction and evaluation. Nonetheless, accurate performance prediction of unseen configurations in terms of execution time, or throughput (when running concurrent jobs), is challenging. It may involve precise analytical modelling of future systems, which has become increasingly complicated with the advances in computer architecture. Also, modelling techniques relying on exhaustive and time-consuming simulations might not even result in the most accurate predictions [2], [4]. Thus, rather than fine-grained system modelling [5], we rely on regression models for

The associate editor coordinating the review of this manuscript and approving it for publication was Rongbo Zhu.

performance prediction. Such models learn the relationships between the configurations of the systems and their performance for various workloads.

SPEC CPU2017 contains suites of industry-standard compute-intensive benchmarks, mainly considering the processor characteristics, memory subsystems, and compilers. SPEC provides real-world and portable programs which solve problems of various sizes [6] and are divided into four benchmark suites: 1) Floating Point rate: *FP_rate*, 2) Floating Point speed: *FP_speed*, 3) Integer rate: *Int_rate*, and 4) Integer speed: *Int_speed*. This paper uses the published performance results of the SPEC CPU2017 public dataset, and develops supervised learning models based on hardware and software features extracted from the computer systems benchmarked in that dataset. Our models are based on Multitask Elastic-Net (MT_EN), Decision Tree (DT), Random Forest (RF), and Multi-layer Perceptron (MLP) estimators.

Previous studies have focused on the accuracy of prediction of the SPEC benchmarks, mostly using neural networks. On the other hand, the importance of the features in terms of their contribution to the prediction models has been overlooked or neglected. The aim of our study is to build an ML pipeline in order to develop fast and accurate regression models for performance prediction of SPEC CPU2017, and provide a streamlined procedure for comprehensive evaluation. One of the main contributions of this work is to identify the importance of several hardware and software features, and compare the prediction error and latency of various models on the full and selected feature sets. This study also tries to uncover whether the data from existing systems (past data) can be used to predict the performance of future systems. The source code is available on GitHub at https://github.com/ashtou/spec17-ml to encourage use and further development of our work. The open source code is written in Python and relies on the scikit-learn [7] ML library. The ML pipeline and analysis provided have some practical implications too, e.g. they could help engineers to reduce the design space and consumers to consider more important factors when making purchasing decisions.

We consider one predictive model per benchmark suite, but since there are multiple benchmarks in each suite, and therefore multiple target variables, we investigate multi-target (multi-output) regression. The aim of multi-target regression is to predict multiple real-valued target variables simultaneously. Multi-target regression is generally classified into problem-transformation and algorithm-adaptation methods [8]. Problem-transformation methods (local methods) consider the problem as independent single-target problems and solve each problem using a single-output regression algorithm. On the other hand, algorithm-adaptation (known as global or big-bang) methods, adapt the algorithm of specific single-target methods (e.g. decision trees) to directly handle multi-output datasets and model the inter-correlations of the target variables. In addition to better computational efficiency, such methods provide better representation and interpretability of complex real-world problems [9], [10]. We use four of such multi-target regression models: I) Multitask elastic-nets, which estimate sparse coefficients for multiple regression problems jointly, II) Decision trees, which predict the value of target variables by learning decision rules inferred from the labeled data, III) Random forests, which are ensembles models and operate by constructing a number of decision trees, and IV) Multi-Layer Perceptrons, which are a class of neural networks with at least three layers. To train these models, we obtain the performance *Ratio* scores from the published results of the SPEC CPU2017 dataset. We hold out 20% of the dataset as a test set for evaluation of the final models and use 80% of the data for model development and optimisation. During the model development process, k-fold Cross-Validation (CV) is used while we investigate the best features and also tune the hyperparameters of the ML models, i.e. the parameters that are not directly learnt or estimated from the data [11].

All preprocessing and feature selection steps during the model development are performed inside the CV loop, as cross-validation should always be the outermost loop to avoid leakage. To achieve this, we chain the data processing steps into a single estimator using the scikit-learn *Pipeline* and pass it to a CV function. We use 5-fold cross-validation, which splits the training set into 5 subsets of samples, where each subset is given the opportunity of being used as a test subset once. To find the best-performing models, we search the hyperparameter space for the best CV scores using an exhaustive *Grid Search*, which basically tries all combinations of manually specified hyperparameter values and returns the combinations with the highest scores.

The rest of the document is organised as follows. Section 2 presents the data preparation step, which includes obtaining data and defining features, cleaning the data, visualisation, and data transformation. Section 3 introduces the four regression models. Section 4 reduces the number of features in three feature selection steps and identify the most important features. Section 5 selects the best models via hyperparameter grid search. Section 6 presents the results and evaluates the models; it also illustrates the learning curves for performance prediction of the future systems using the past data. Section 7 reviews the related work, and finally Section 8 presents our conclusions. Fig. 1 illustrates the main steps of data preparation, feature selection, model selection, and evaluation of the results.

II. DATA PREPARATION

A. OBTAINING DATA

The SPEC CPU2017 dataset [6] collects two main metrics: *SPECspeed* and *SPECrate*. *SPECspeed* is a time-based metric and captures the time to complete a workload, while *SPECrate* is a throughput metric, concerned with the work completed per unit of time, hence it runs multiple concurrent copies of each benchmark. The dataset also contains Integer (Int) and Floating-Point (FP) benchmarks,

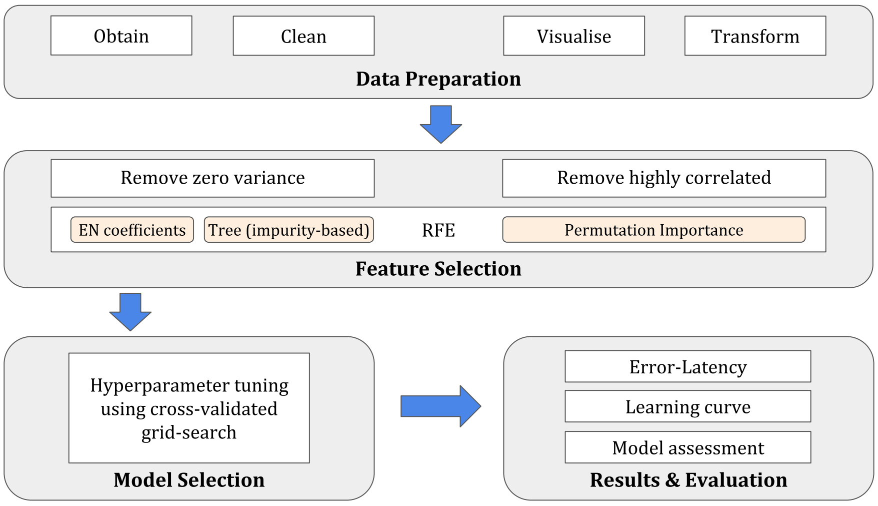


FIGURE 1. ML pipeline: steps taken from obtaining the data to the final evaluation of results.

therefore, the benchmarks are organised into four suites: *FP_rate*, *FP_speed*, *Int_rate*, and *Int_speed*, consisting of 13, 10, 10, and 10 benchmarks respectively. The benchmarks are derived from a wide variety of application domains and are written in C, C++, and Fortran. For each metric, there are *base* and optional *peak* options (with more flexibility for optimisation with different compilers). In our analysis, we use the results of the *base* metric, where all modules have been compiled using the same flags, in the same order.

Our target (output) variables in the regression models are the performance *Ratio*s of the benchmarks. The performance *Ratio* scores for the speed and rate benchmarks, $speed_{Ratio}$ and $rate_{Ratio}$ are calculated in Eq. 1:

$$\begin{aligned} speed_{Ratio} &= t_{Ref}/t_{SUT} \\ rate_{Ratio} &= n \times (t_{Ref}/t_{SUT}) \end{aligned} \tag{1}$$

where t_{Ref} is the time on a reference machine with a single thread (in the case of *speed* benchmarks) or a single copy of the benchmark (in the case of *rate* benchmarks), t_{SUT} is the time on the system under test with multiple threads or copies, and n is the number of copies of the benchmark.

We obtain data for each of the four benchmark suites from the SPEC CPU2017 published results at [12]. In each raw data record (observation), we store 21 fields as the input set as well as the *base Ratio* scores of the benchmarks running on the System Under Test (SUT) as the output variables. There are other inputs that may help with building a model, such as the *vendor*, *model_name*, *test_sponsor*, etc., but we ignore them, as our main objective is to be able to predict the results using only hardware and software configurations.

The 21 raw features are listed in Table 1. The architecture, processor(s) details, and the memory subsystem are hardware details. The OS, compiler, file system, parallel flag, and number of threads or copies are software details. The OS, compiler and file system variables are categorical, which need to be encoded into numeric.

There are input variables whose effect on the output changes depending on the values of other input variables. This is known as interaction effect. For example, $cpus = sockets \times cores\_per\_sockets \times threads\_per\_core$ represents a three-way interaction.

It is worth noting that the CPU architecture information such as the number of CPUs, sockets, the Non-Uniform Memory Access (NUMA) nodes, etc. are gathered from the information provided by the test sponsors via the *lscpu* command. Also, to obtain memory details, we check to see if the output of the */proc/meminfo* command is provided, as the information submitted manually by the test sponsors is not always accurate.

SPEC CPU2017 benchmarks are CPU-intensive benchmarks, but as can be seen from Table 1, there are 10 different features that characterise the processing unit; so it would be worthwhile to know about the importance of such features for performance prediction. Moreover, we will see that memory, OS, and compiler features can still have substantial impact on the accuracy of predictions.

At the time of writing, there were 5382 submitted cases for *FP_rate*, 4328 for *FP_speed*, 5607 for *Int_rate* and 4333 for *Int_speed* in all SPEC CPU2017 results published by SPEC (last update: 2020-06-30). For the purposes of this paper, we have used the scikit-learn [7] machine learning library version 0.23.2 on a 12-core (6 physical) Intel Core i9-8950HK machine at 2.90GHz with 32GB memory and an Ubuntu 20.04 LTS OS.

B. DATA CLEANING

After collecting data, we filter out observations (cases) which do not have standard formats as well as those that do not contain the results from the *lscpu* command. The dataset does

TABLE 1. Raw feature set obtained from the SPEC CPU2017 public dataset.

Raw Feature	Description
arch	Architecture from $lscpu$
nominal_mhz	Nominal clock frequency of the CPU
max_mhz	Maximum clock frequency of the CPU
cpus	Number of CPU(s) from $lscpu$
threads_per_core	Thread(s) per core from $lscpu$
cores_per_socket	Core(s) per socket from $lscpu$
sockets	Scoket(s) from $lscpu$
numas	NUMA node(s) from $lscpu$
l1d_cache_kb	L1 data cache in KB
l1i_cache_kb	L1 instruction cache in KB
l2_cache_kb	L2 cache in KB
l3_cache_kb	L3 cache in KB
mem_kb	Main mem in KB (get it from the $Memory$ field, if there is no $/proc/meminfo$)
mem_channels	Number of memory channels
channel_kb	Memory channel's capacity
mem_data_rate	Memory transfer rate in MT/s
os	Operating System
compiler	Compiler
parallel	Compiler parallelism: yes/no
file_system	File system type
threads_or_copies	Number of OpenMP threads in *SPECspeed* or benchmark copies in *SPECrate*

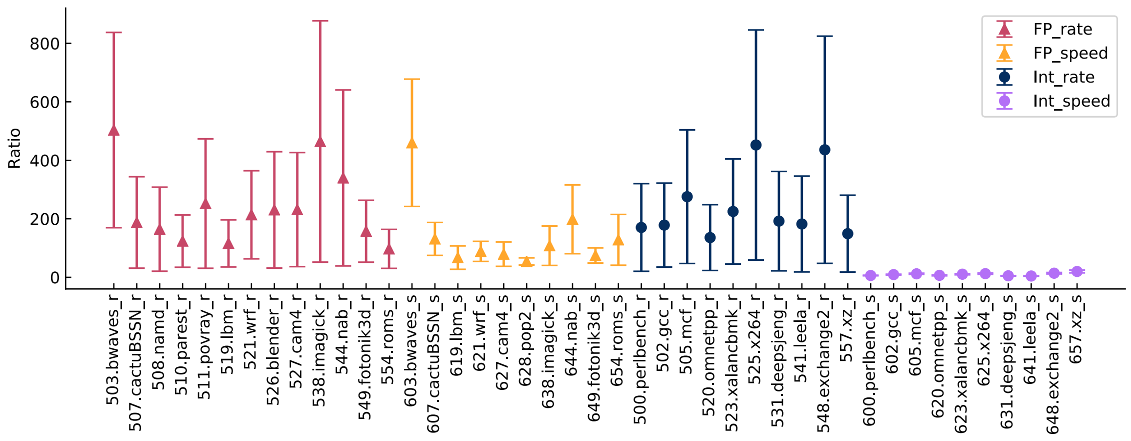


FIGURE 2. Means and standard deviations of the SPEC CPU2017 benchmarks.

not contain duplicates, but there are a few cases with missing values. The total percentage of non-standard cases and cases with missing values is less than 1% of the data in each suite, so, we simply remove such cases from the dataset. Therefore, our cleansed data contains 5331 of 5382 cases for *FP_rate*, 4306 of 4328 cases for *FP_speed*, 5553 of 5607 cases for *Int_rate*, and 4313 of 4333 cases for *Int_speed*.

C. VISUALISATION

Fig. 2 presents the measured performance of all the benchmarks, in terms of the mean and standard deviation of their *Ratio* scores. Note that the systems/configurations used for the evaluation of the four suites are not exactly the same, so we cannot use Fig. 2 for accurate comparison of the benchmarks, nor do we intend to focus on workload characterisation. However, the chart shows that in general, the *rate* benchmarks (names ending with the suffix "_r") have more diverse means and standard deviations compared with the *speed* benchmarks. The large variations of the scores within and across the four suites indicate that the performance prediction is nontrivial. To further demonstrate that, we use a simple baseline model for each suite, which predicts the median of the train set. The average Mean Absolute Percentage Error (MAPE) values for such a model are as follows: *FP_rate*: 94%, *FP_speed*: 73%, *Int_rate*: 107%, and *Int_speed*: 16%.

D. DATA TRANSFORMATION

1) ENCODING CATEGORICAL VARIABLES

Using high-level domain knowledge of the data, we convert categorical variables into numeric as to make them suitable for linear regression. The first step is to create dummy variables from the categorical predictors such as *os* and *compiler*. By converting categorical data to dummy variables we can end up in a situation where perfect multicollinearity occurs. In other words, one value can be predicted from the other values [13]. This is known as the *dummy variable trap*. We drop the first dummy variable to avoid the dummy variable trap. Such a conversion removes two variables with single unique values: *arch* and *parallel*, but also generates extra dummy variables from the OS and compiler data. We have identified four main categories of Operating Systems, namely SUSE Linux Enterprise Server (SLES), Red Hat Enterprise Linux (RHEL), Ubuntu and CentOS, and two unique categories of compilers: Intel and AMD Optimizing C/C++ (AOCC). The version details describe some characteristics of the *explanatory* OS or compiler categorical variables, so we consider them as *nested* variables [14]. Therefore, the regression model looks like Eq. 2:

$$y \sim 1 + explanatory + explanatory : nested + \ldots \quad (2)$$

There is no main effect term for the *nested* variable itself in the above formula (only its interaction term), because it is a conditional variable that only makes sense if the *explanatory* variable equals 1. We omit the reference category of CentOS from the OSes and the reference category of AOCC from the compilers to avoid the dummy variable trap, but we need to keep the interaction terms to show interactions between the main categories and their version numbers, so we end up with 3 main effects and 4 interaction terms for OSes and 1 main effect and 2 interaction terms for compilers. Table 2 shows the data before and after such encoding, and conversion of the OS and Compiler columns into the wide format.

In addition, the file system categorical variable gets encoded into three dummy variables: ext4, tmpfs, and xfs, leaving btrfs as the reference category. After all these transformations, we end up with 29 input variables.

2) LOGARITHMIC TRANSFORMATION

We also transform the output variables and some of the input variables to make the relationships between them more linear and make it fair to compare linear models with non-linear ones. There are more substantial reasons to consider data transformation though. First of all, (linear) regression models are not suitable for strictly positive continuous target variables, as they will predict negative values too; one way to avoid this is to transform the target variable using logarithmic transformation. The *log* transformation can also help make a highly skewed distribution less skewed and patterns in the data more visible. Here, log-transformation of *Ratio* is helpful because the majority of the benchmarks have right-skewed distributions. Fig. 3 shows all four possible combinations of log-transforming input and output variables in a simple linear regression. The charts show the *Ratio* value of the 503.*bwaves_r* benchmark from the *FP_rate* suite against a potentially important feature: *cpus*. It is shown that the best linear relationship can be achieved by log-transforming the predictor *log*(*cpus*) and the output (also known as the log-log model), highlighted in green.

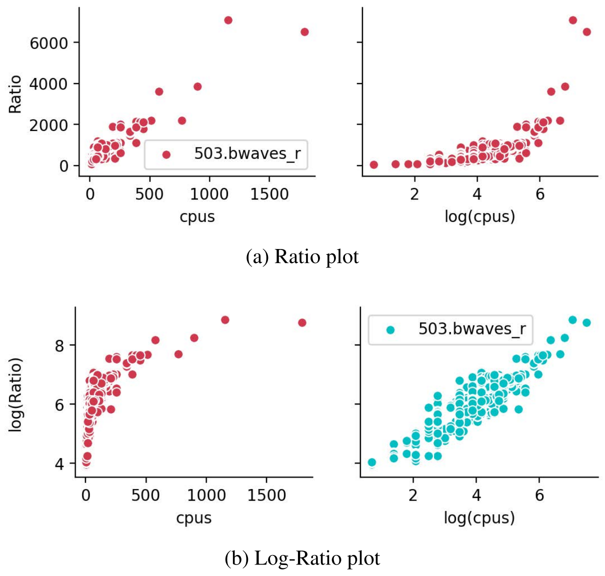


FIGURE 3. The effect of transforming the *cpus* input (the number of logical cores) and the *Ratio* output of 503.*bwaves_r* on increasing linearity between the two.

A log-log regression such as $log(Ratio) = \beta_1 log(cpus)$ can be interpreted as follows: the percentage change in *Ratio* is β_1 times the percentage change in *cpus*. Consider a *speed* benchmark, where *Ratio* represents speedup. In such a case, we can say that if the number of *cpus* is increased by 1%, the speedup is expected to increase by β_1%. Apart from *cpus*, we also log-transform *mem_kb* and *threads_or_copies* features. *threads_or_copies* is highly correlated with *cpus*.

Note that when making predictions, an inverse transform function (*exp*()) back transforms the target variable to its original scale. So, all the reported results are based on the prediction of the actual *Ratio* values. It is also worth stating that the last preprocessing step in our pipeline is to standardise features to have a mean of 0 and variance of 1, using *StandardScaler*. We scale our data, as some models, including neural networks expect all features to vary in a similar way.

After the preprocessing steps and as an initial analysis, we compare two multiple linear regression models using all 29 available numerical features in predicting the *Ratio* outcome of benchmarks: one model without target transformation and one with log-transformation of the target. In Fig. 4, we illustrate the results of out-of-sample prediction (test set prediction) using MLR (Multiple Linear Regression) models for the benchmark 505.*mcf_r* from the *Int_rate* suite. The charts show that with log-transformation of the target, not only is the prediction of negative values avoided, but also the R2 score (goodness-of-fit or Coefficient of determination), the Mean Absolute Error (MAE), and the Mean Absolute Percentage Error (MAPE) of the predictions are improved.

III. MULTI-TARGET REGRESSION

As each SPEC suite contains multiple benchmarks, and hence multiple target variables, we need multi-target regression

TABLE 2. Encoding of categorical OS and compiler variables. IA means an interaction term.

Ind	OS	Compiler
0	Red Hat Enterprise Linux Server release 7.3	C/C++: Version 18.0.0.128 of Intel C/C++
1	SUSE Linux Enterprise Server 12 SP3 (x86_64)	C/C++: Version 1.0.0 of AOCC

Transformed

Ind	OS_ SLES	OS_ RHEL	OS_ Ubuntu	IA_ SLES	IA_ RHEL	IA_ Ubuntu	IA_ CentOS	CMP_ Intel	IA_ Intel	IA_ AOCC
0	0	1	0	0.0	7.3	0.0	0.0	1	1800	0
1	1	0	0	12.3	0.0	0.0	0.0	0	0	100

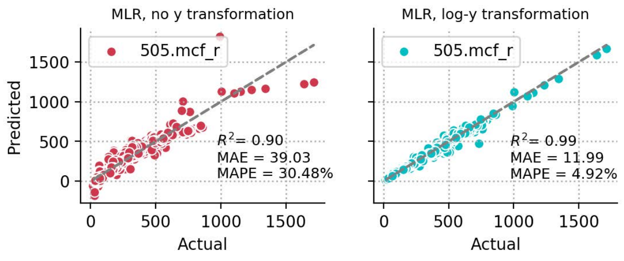


FIGURE 4. The effect of log-y transformation on Multiple Linear Regression (MLR) R2 and errors of predicting the *Ratio* output of 505.*mcf_r* using all 29 features (3 of which are log-transformed).

models to predict the performance of the benchmarks in each suite. We have chosen models which natively support multi-target regression, namely: Multitask Elastic-Net, Decision Tree, Random Forest, and Multi-Layer Perceptron (Neural Network). For instance, in tree-based models, the splitting criteria computes the average reduction across all targets.

A. MULTITASK ELASTIC-NET (MT_EN)

Elastic-Nets are regularised linear regression models that linearly combine the L1 and L2 penalties of the Lasso and Ridge regressions. Ridge regression decreases the complexity of the model and reduce variance by shrinking the effect of input variables (coefficients) and not reducing the number of them, while Lasso eliminates some features entirely.

The combination of L1 and L2 norm regularisation allows for learning a sparse model where coefficients are shrunk (as in Ridge), while some of them are set to zero (as in Lasso). Here, we control the convex combination of L1 and L2 with the *l1_ratio* parameter. Elastic-net is useful when multicollinearity exists. In such cases, Lasso is likely to pick one of the correlated features randomly, while elastic-net groups and shrinks the parameters associated with correlated features and thus can pick them all or remove them at once.

Multitask Elastic-Nets (MT_EN) estimate the sparse coefficients for multiple linear regression problems with n outputs jointly, supposing that the features for all the individual regression problems (tasks) are the same. MT_ENs use coordinate descent to fit the coefficients [7].

B. DECISION TREE (DT)

Decision tree learning uses a Decision Tree (DT) to predict target values at the leaves using observations in the branches [15]. By default, the quality of a split at each node of DTs (impurity) is measured by how much it can reduce the *Mean Squared Error* (MSE). Later in section V, we will consider other criteria for model selection, e.g. MAE, which minimises the L1 loss using the median of each terminal node.

Decision trees in the scikit-learn library can readily provide multi-target joint feature selection and prediction by considering inter-correlations between the target variables. They are yet easy to understand and highly interpretable too [15]. Moreover, decision trees do not require scaling of data and can capture non-linear relationships.

DTs tend to overfit though. Overfitting happens when the model models the training data too exactly, to the extent that negatively impact its ability to generalise, so it may fail to predict future observations reliably. For the initial models, we set the *max_depth* of the trees to 15 to avoid overfitting.

C. RANDOM FOREST (RF)

A Random Forest (RF) is an averaging ensemble method that injects randomness into the tree building process to create a number of different decision tress. It then fits such trees on the dataset and averages their prediction results to improve accuracy and reduce overfitting. Each tree is likely to overfit on part of the data, but the overall variance and amount of overfitting is reduced by averaging the results of diverse trees [16].

Random Forests do not require scaling of data. For hyperparameter tuning in Section V, we will not consider pre-pruning options such as *max_depth*, since fully grown trees work do not cost much in random forests [15]. Also, a larger *n_estimators* is always useful, because averaging more trees yields a more robust ensemble model by reducing overfitting; on the other hand, more estimators (trees) require more memory and increase the training time [16]. Therefore, for the initial models, we set *n_estimators* to 30, and investigate the effect of this hyperparameter in Section V. Another hyperparameter is *max_features*, which determines the number of features to consider at each split, and is the source of randomness in the tree creation process. Smaller *max_features* reduces overfitting, and larger values mean that the trees in the random forest will be quite similar. We set it to 0.5 here, which means that half of the features are considered at each split.

D. MULTI-LAYER PERCEPTRON (MLP)

Multi-Layer Perceptrons (MLP) are known as feed-forward neural networks. An MLP is composed of an *input layer*, one or more *hidden layer*s, and an *output layer* of *neuron*s. MLPs can be considered as generalisations of linear models with multiple stages of learning *weight*s (coefficients). Neurons of the hidden layer transform the values from the previous layer using weighted linear summation followed by a nonlinear activation function, such as the *rectified linear unit (relu)* or the *tangens hyperbolicus (tanh)*. The result is then used in the weighted summation that computes the output values. The solvers we use for weight optimisation are *adam* and *lbfgs*. *adam* is a stochastic gradient-based optimiser, while *lbfgs* is an optimiser in the family of quasi-Newton methods.

MLPs are trained using the backpropagation algorithm, which calculates the gradient of a loss function with respect to the weights in the network. The gradient is then used by the optimisation algorithm to update the weights. MLPs can learn nonlinear models, but are sensitive to feature scaling. They are powerful models, if provided with enough data and careful hyperparameter tuning, but can take longer time to train, compared to the other models [16]. For the initial models, we consider one hidden layer of size 20.

IV. FEATURE SELECTION

Reducing the dimensionality of the feature space provides several benefits. It results in a low dimensional model, can provide better interpretability, requires less memory space, reduces the time complexity (training and prediction time), and also decreases the risk of overfitting. Generally, dimensionality reduction can be done using Feature Selection or Feature Transformation approaches.

Feature selection is basically reducing the number of features by eliminating some of them; a process which maintains interpretability of the variables, but also eliminates any benefits/information the dropped variables would bring. Feature transformation, on the other hand, is about creating new predictor variables by combining all of the old predictor variables, ordering the new ones by how well they predict the response variable and finally dropping the least important ones. Principal Component Analysis (PCA) is a technique for feature transformation. We use feature selection as we would like to keep our predictor variables fully interpretable. The goal is to identify the most important hardware and software predictors out of the initial feature set.

A. FEATURE IMPORTANCE ESTIMATORS

As stated, having a large number of features makes the models more complex and increases the chance of overfitting [16]. An estimator that assigns weights to features can be used to determine the importance of the features. Linear models provide coefficients and tree-based models provide impurity-based feature importance. The third approach that can be used with any of the models is called *Permutation Importance* (PI), a technique that randomly permutes the values of a feature and computes the decrease in a model score. A feature is considered ‘‘important’’ if shuffling its values decreases the model score (or increases the error).

Therefore, we use three different feature importance methods: L1/EN-based, Tree-based, and Permutation Importance. In Table 3, the feature importance methods as well as hyperparameters used for feature selection for each model are specified.

1) ELASTIC-NET IMPORTANCE

Linear regression models with L1 regularisation shrink some of the coefficients to zero, and hence can be used as feature selection methods. Lasso regression is a least squares method that imposes an L1 penalty on the regression coefficients. Although Lasso performs continuous shrinkage and automatic feature selection at the same time, it has some limitations when there are correlations between features [17]:

- When the number of observations is larger than the number of features, if there are high correlations between the features, Ridge regression (with L2 regularisation) can achieve better prediction performance than Lasso
- When there are highly correlated features, Lasso fails at group selection and selects only one of the correlated features at random. Ridge regression on the other hand, tends to shrink coefficients for the correlated features together.

By combining Lasso and Ridge regressions, Elastic-Net (EN) regressor groups and shrinks the coefficients associated with the correlated features, and therefore EN-based feature selection tends to select all such features or remove them all at once.

2) TREE-BASED IMPORTANCE

Tree-based estimators such as Decision Tree and Random Forest regressors can be used to compute impurity-based feature importance values, which in turn can be used to select important features. The importance of a feature is computed as the (normalised) total reduction of the criterion (e.g. MSE) by that feature.

Generally, impurity-based feature importance methods can be misleading for two reasons: firstly, they are measured on the training data and do not necessarily show which features are more important for predicting the held-out test data. Secondly, they favour high cardinality features (features with many unique values) [7]. That is why we use permutation importance as an alternative feature selection method for our tree-based models.

3) PERMUTATION IMPORTANCE (PI)

Permeation Importance (PI) is an effective technique for the inspection of any models on tabular data, and is defined at the decrease in the model score when a single feature value is randomly permuted [18]. So, we do not need to retrain the inspected model at each modification. More importantly, PI can be computed on the validation set (or using cross-validation) to tell us which features contribute the most to the generalisation power of the model. We use *neg_mean_absolute_error* as its scoring parameter, which

TABLE 3. Models' hyperparameters for feature selection.

Model	Feat. Importance	Hyperparameters
MT_EN	EN-based	`alpha=1e-3, l1_ratio=0.5, max_iter=5000, tol=1e-4`
DT	Tree-based & Permutation	`max_depth=15`
RF	Tree-based & Permutation	`n_estimators=30, max_features=0.5`
MLP adam	Permutation	`hidden_layer_sizes=(20,), solver="adam", activation="tanh", max_iter=10000, tol=1e-3, early_stopping=True`
MLP lbfgs	Permutation	`hidden_layer_sizes=(20,), solver="lbfgs", activation="tanh", max_iter=10000, tol=1e-3`

means that PI will measure the decrease in the negated value of MAE, since MAE is a lower-better metric.

PI can also be misleading when there are correlated features, because when one of such features is permuted, PI can still have access to the same information that the feature provides through the correlated ones. As a result, it provides lower importance values for the correlated features, since their true importance cannot be detected.

As discussed, highly correlated features make the feature selection process difficult. We therefore use two techniques to alleviate the problem: I) removing highly correlated features using Spearman rank-order correlation and II) using Recursive Feature Elimination (RFE). We cover these two techniques in the following subsection.

B. FEATURE SELECTION STEPS

Referring back to Fig. 1, three feature selection steps are implemented in the ML pipeline, which are described below in the same order as they are applied to the dataset.

1) REMOVE ZERO VARIANCE

As the initial step, we remove single unique variables. Features with one unique value have zero variance. The categorical variables *arch* and *parallel* are already removed when encoding the data at the preprocessing step. At this stage, *l1d_cache_kb* feature is also removed.

2) REMOVE HIGHLY CORRELATED

Highly correlated features bring almost the same effect on the outputs. Also, as discussed earlier, they can make it difficult to accurately measure the importance of features. Thus, after removing the zero-variance features, we cluster correlated features and only keep a single feature from each cluster. Spearman rank-order correlation is used as a nonparametric measure of the association between the features, known as the monotonic relationship, where variables tend to move in the same relative direction, but not necessarily at a constant rate. We perform hierarchical clustering on the absolute values of the features' Spearman correlations, use the cluster distance threshold of 0.35, and keep only one of the highly correlated features. We have manually picked this threshold by visual inspection of the dendrograms representing the hierarchical relationships between the features. After this step, the number of features in each suite is 24.

3) RECURSIVE FEATURE ELIMINATION (RFE)

Feature selection methods are generally classified into three types. The Filter-based methods, which are independent of the learning algorithm and are based on statistical measures. The Wrapper methods, which measure the usefulness of features by actually training models. And the Embedded methods, which use ensemble learning and hybrid learning for feature selection, but are specific to certain learning models. Venkatesh *et al.* [19] have reviewed these feature selection methods. Based on their analysis, although wrapper methods such as Recursive Feature Elimination (RFE) are computationally expensive, they consider correlations between features and are more accurate than filter-based methods such as variance threshold, Chi-square or Mutual Information (MI). Wrapper methods are the preferred feature selection approach in several contemporary real-life case studies of machine learning [20], [21].

RFE is an iterative backward selection procedure, which is based on eliminating less important variables and starts by building a model with all features. It then ranks the features, removes the least important one(s) and builds a new model with the remaining features at each step of its recursive elimination process. Our RFE method uses the feature importance estimators introduced in the previous subsection (based on the corresponding predictive model) to obtain feature importance values. The estimator is initially trained on the full feature set. The process of removing the least important feature is then recursively repeated until the desired number of features is reached.

As noted before, RFE [22] can be useful in the presence of correlated features, because informative features are ranked highly in the last steps of the elimination process, even if that is not the case in the early steps [23]. Let us consider the aforementioned problem with the PI method to clarify this further. Suppose that there are two correlated important features. As discussed previously, their permutation importance values will be lower than the other features (when one is permuted, its original information will be still accessible through the other feature), so one of them will be removed by RFE in the earlier steps. Once one of correlated features is removed, the permutation importance for the other one in the next RFE steps will be high, and therefore RFE can successfully keep and select this feature in line with its true importance.

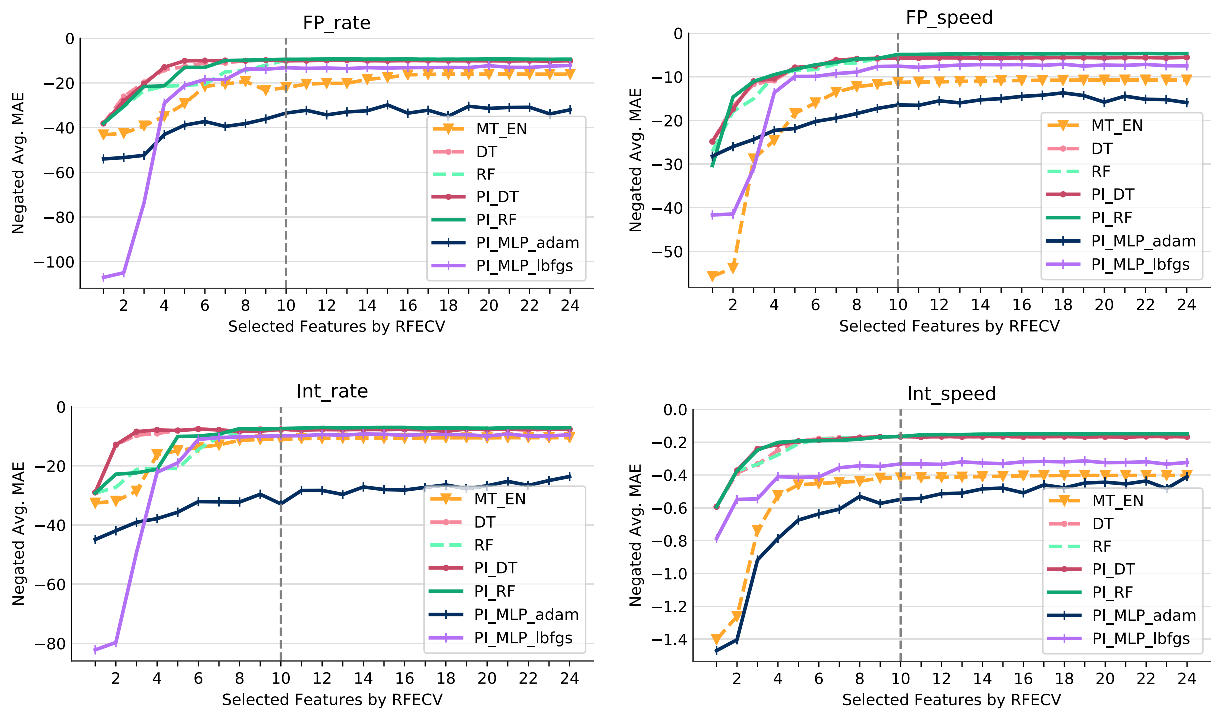


FIGURE 5. **Average cross-validation scores for varying number of features selected by RFECV.**

In order to find a proper number of features to select, we use RFE in a 5-fold cross-validation loop, known as RFECV. RFECV can provide the optimal number of features based on the CV score, however we would like to select the same number of features for all estimators, so we use RFECV to calculate the CV scores for all different numbers of selected features from 24 to 1. By looking at the resulting charts in Fig. 5, we decided to select 10 features, which for most of the cases provide a near-minimum MAE. Note that the negated MAE is used as the RFECV scoring parameter, and the y-axis in the charts presents the average of its cross-validated value across all benchmarks of a suite.

Based the results generated by RFECV, we therefore set the number of features to select by RFE to 10. Since RFE is computationally expensive, we use it once to select the top-10 features for each estimator and then use the masks of the selected features in the ML pipeline. This is done by using a *mask transformer*, which simply applies the feature masks to the data. It is especially useful when performing time-consuming operations such as hyperparameter tuning.

After the transformations at the preprocessing step, the MT_EN linear model can be fairly compared with the non-linear models. In fact, Fig. 5 shows its performance is between that of the MLP_lbfgs and MLP_adam models. It is also evident that tree-based models provide the lowest MAEs. They also reach their optimal levels with a fewer number of features, compared with the other models. Tree-based models using PI perform slightly better with fewer number of features, but at 10 features and above, their performance is almost identical to the models with impurity-based importance.

Fig. 6 presents the feature importance ranking by each estimator. Note that the rankings are obtained using the permutation importance (PI) method during feature selection, except for the MT_EN, which is based on the elastic net coefficients. For the PI-based methods, we have set the number of times to permute a feature to 5, hence we use box plots to show the mean and standard deviation of the results. It can be seen that in the tree-based models which have the lowest MAEs, only a few features are of high importance; the top 2-3 features for the *rate* benchmarks and the top 3-4 features for the *speed* benchmarks. MT_EN, DT, and RF have ranked the number of CPUs (or *log_cpus* as we have transformed this feature) as the most important feature in all the suites, except for the *Int_speed*, where they all rank *max_mhz* as the most important feature. MLP_adam follows a similar pattern, except for the *FP_rate*, where *sockets* comes first and *log_cpus* comes close second. MLP_lbfgs ranks the features quite differently, but we will see in the next section that it provides a significantly better MAE compared to MLP_adam in all four suites.

Some of other observations regarding the tree-based models include: they both rank all the processor-related features except *threads_per_core* (*threads/core*) among the top-10 in all suites but the *Int_speed*. *max_mhz*, *log_cpus*, and *cores_per_socket* are always amongst the top-10. From the memory-related features, *l3_cache* is always in the top-5.

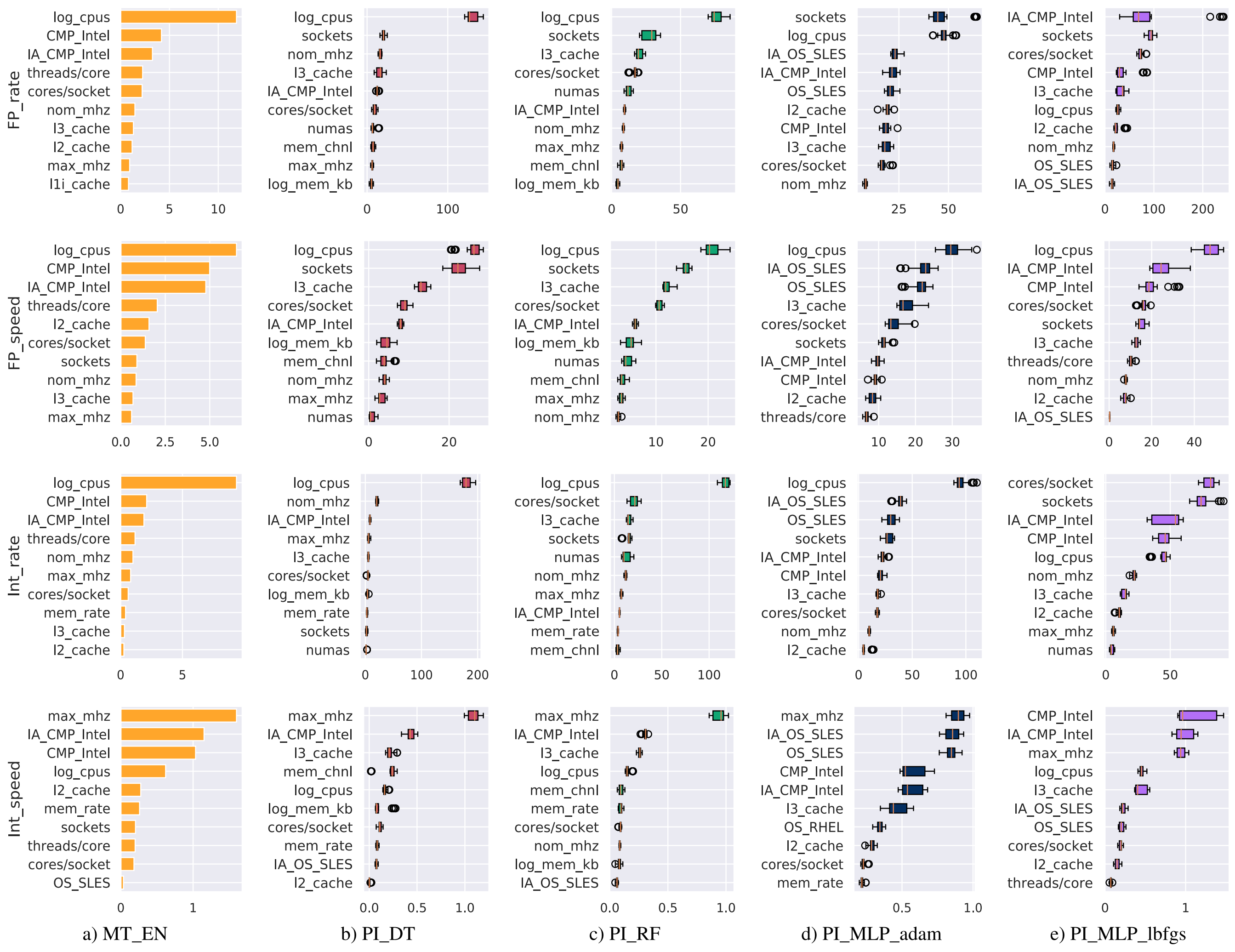


FIGURE 6. Top-10 features using RFE. Feature importance values are calculated using: coefficients for MT_EN, and permutation importance for DT, RF, MLP_adam, and MLP_lbfgs.

Out of the software-based features, Intel compiler's version *IA_CMP_Intel* is always ranked in the top-10.

V. MODEL SELECTION

Making decisions about model selection based on the test set score would *leak* information from the test set to the model, because one can tweak the model's settings until it performs optimally on the test set. Resampling methods can avoid this problem during the model development process. They are common ways to economically make use of the available data by splitting the train set into sub train-test sets multiple times and estimating the mean performance on out-of-sample (unseen) data. The most common resampling model selection method is cross-validation (CV). As before, we use a 5-fold cross-validation.

In this section, we focus on the process of selecting the final model(s) from among the models based on MT_EN, DT, RF, and MLP estimators. This process also includes selecting the optimal hyperparameter settings for each estimator. We start with hyperparameter tuning to identify the top-10 hyperparameter sets for each of the above estimators and then compare all the candidate models in terms of prediction latency and error to select the best-performing model(s).

A. HYPERPARAMETER TUNING

The aim of hyperparameter tuning is to improve the model's generalisation performance by finding the hyperparameter values that provide the best out-of-sample performance. One of the most widely used hyperparameter tuning method is to perform an exhaustive search over a grid (all possible combinations) of specified hyperparameter values, known as *grid search*. To reliably estimate the generalisation performance, we analyse the performance of each hyperparameter combination using cross-validation, also known as cross-validated grid search (*GridSearchCV*) [16]. Finally, to evaluate how well the best found combination generalises, we measure its score on the held-out test set. As before, negated MAE is used as the scoring parameter. The hyperparameter values used for grid search can be found in Table 4.

TABLE 4. Hyperparameter grid used in cross-validated grid search.

Model	Hyperparameters
MT_EN	"alpha": `[1e-4, 1e-3, 1e-2, 1e-1]`, "l1_ratio": `[0.25, 0.5, 0.75, 1]`
DT	"criterion": `["mse", "friedman_mse", "mae"]`, "max_depth": `[5, 10, 15, 20, 25, None]`
RF	"n_estimators": `[10, 30, 50, 100, 200]`, "max_features": `[0.5, 0.7, 0.9, None]`, "max_samples": `[0.8, 0.9, None]`
MLP	"hidden_layer_sizes": `[(10,), (20,), (30,), (50,)]`, "alpha": `[1e-5, 1e-4, 1e-3, 1e-2]`, "tol": `[5e-4, 1e-3]`, "activation": `["tanh", "relu"]`

VI. RESULTS AND EVALUATION

In this section, we first present the results of model selection to identify the best-performing models. We then focus on the learning curves for the data sorted by date, and finally measure the generalisation error of the selected models.

A. PREDICTION ERROR AND LATENCY

For each estimator, we select its top-10 models with the highest predictive power, i.e. with the best average *GridSearchCV* scores, and then compare them all together in terms of the prediction error and latency. This time, for the tree-based models, we have included the results of both feature importance estimators, the impurity-based (tree-based) feature importance method and the PI method. We can see in Fig. 7 that the difference between the two methods is negligible in all cases, but we generally recommend using PI due to the two drawbacks of the impurity-based method stated earlier: I) its ranking is based on the training set statistics II) it favours high cardinality features.

It is also apparent from Fig. 7 that the best-performing models in terms of both the prediction error and the prediction time (latency) are the tree-based models. Among RF-based models, we observe that some models have the same accuracy levels as the others, but higher prediction latency. These are models with larger numbers of estimators. We can thus conclude that for this dataset, more complex RF models ($n\_estimators > 50$), which also require more training time, do not reduce the prediction error and only increase the prediction time (latency).

The neural network MLP_lbfgs models provide good results too, however both their prediction time and error are higher than those of the top-performing tree-based models. Furthermore, we have observed that the training time of the MLP_lbfgs models can get an order of magnitude larger than that of the tree-based approaches, especially for the floating-point (FP) benchmarks.

1) DT UNDERFITTING AND OVERFITTING

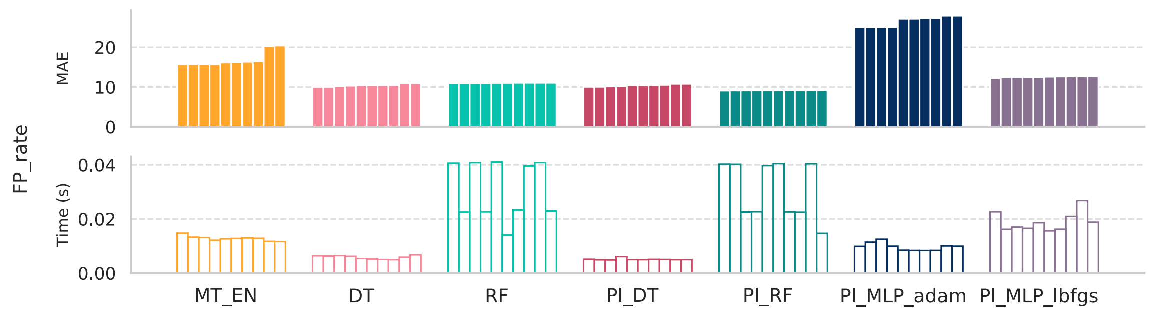
We have observed that the tree-based models provide the best performance. However, since decision trees tend to overfit, we need to investigate this before going forward. Overfitting occurs when the learned function fits the training data too exactly and thus becomes too sensitive to noise, it cannot generalise or perform well on unseen data. So, overfitting results in high generalisation error. In contrast, underfitting, which is when the model cannot adequately fit the training data, also results in poor generalisation.

Generally speaking, the error can be decomposed into the bias, variance and irreducible error. The irreducible error is a property of the data and cannot be reduced regardless of what model is used. The bias error is the model's average error for different train sets and is a result of the assumptions made by the model to make the target function easier to learn. And finally, the variance error indicates how sensitive the model is to varying train sets, i.e. how much its estimate will change. Overfitting occurs when the model has low bias and high variance, and underfitting happens when the model shows high bias and low variance.

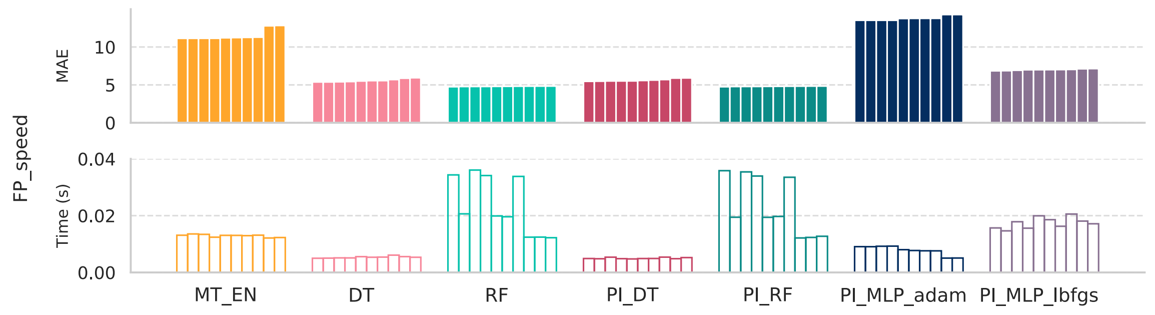
Not limiting the depth of a decision tree can lead to overfitting. One way to check the influence of the *max_depth* hyperparameter is to compare the training and validation score by performing cross-validation using different *max_depth* values. We have observed that a *max_depth* value between 15 and 20 provides a good fit across all suites, i.e. it does not underfit (low training and validation scores) or overfit (high training score and low validation score). We therefore choose trees with $max\_depth = 20$ in our final models.

2) MLP DISCUSSIONS AND FINAL MODELS

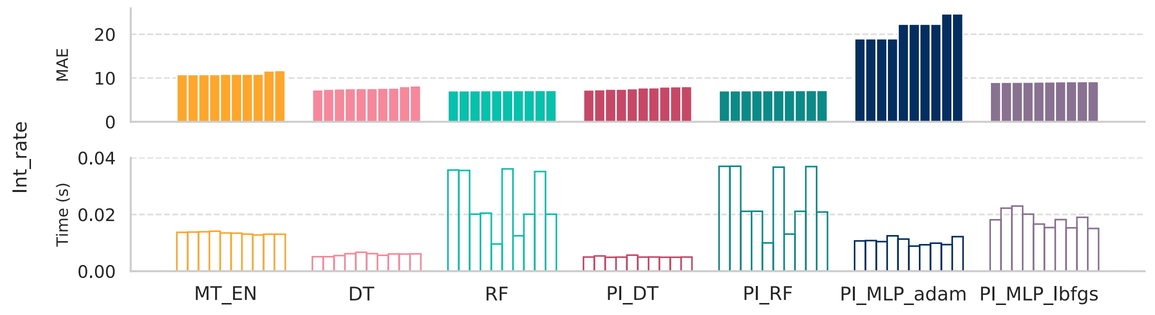
Before we continue further and present the final models, let us make a decision about the solver (optimiser) of our final MLP model by considering neural networks used in related work. Based on the models developed in previous studies, we explored *adam* models with three hidden layers of (12, 12, 12), (16, 16, 16) [2], (50, 100, 50) [24], and (100, 100, 100) [3] with the maximum 10000 iterations (which determines the number of epochs), and early stopping enabled. The SPEC datasets, the development frameworks, and the exact features used in these studies are not the same as ours, but we have attempted to replicate their models as closely as possible. We use the full feature set for this experiment (no feature selection). It is observed that the hidden layer size of (100, 100, 100) provide the best average CV MAE across all four suites. Compared with a single hidden layer of size (50), which is our top-performing *adam* model, the (100, 100, 100) model improves the average MAE by approximately 20%, 16%, 14%, and 15% for the *FP_rate*, *FP_speed*, *Int_rate*, and *Int_speed* suites respectively, while the average fit time is increased in the range of 40% to 170%, and the average score



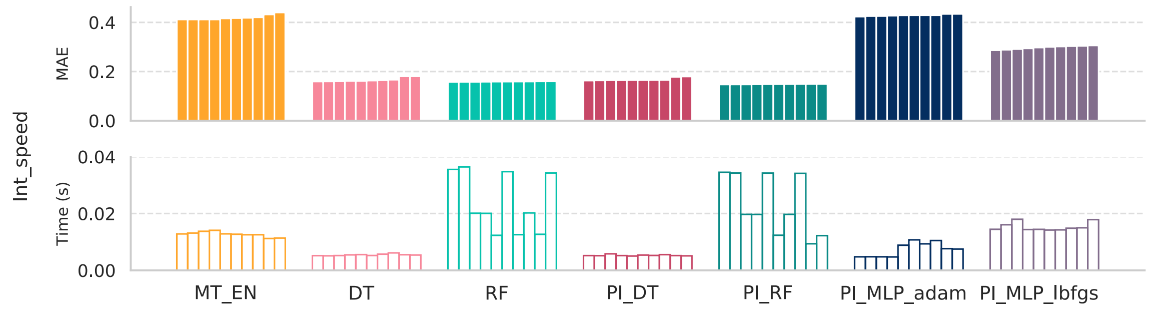
(a) FP_rate: Prediction Error and Time



(b) FP_speed: Prediction Error and Time



(c) Int_rate: Prediction Error and Time



(d) Int_speed: Prediction Error and Time

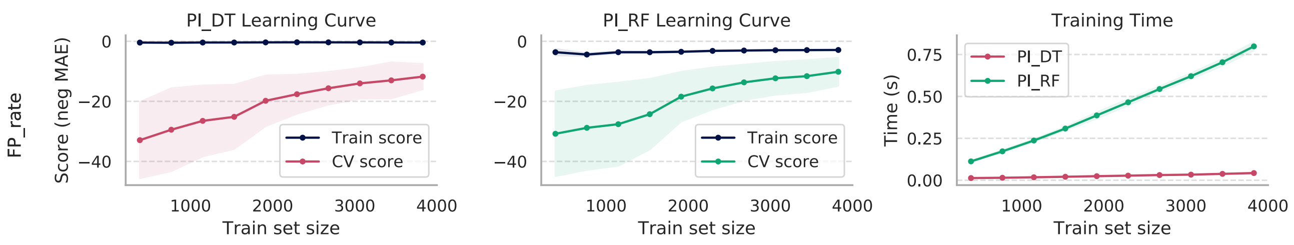
FIGURE 7. Model selection: average cross-validation prediction error (MAE) and prediction time (s). Lower is better.

time is also increased in the range of 90% to 130% across different suites.

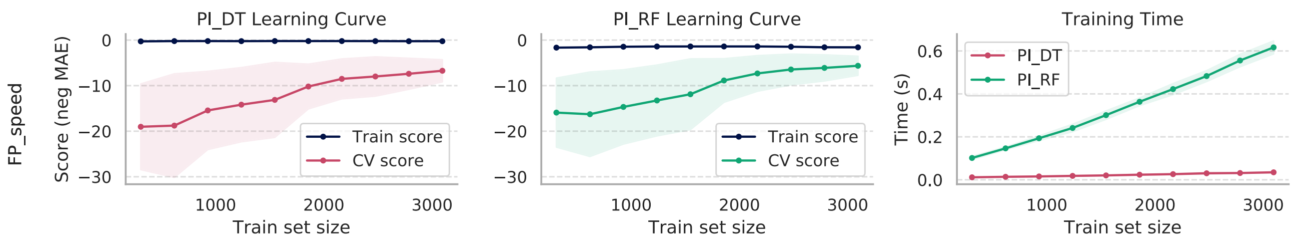
At the cost of a reasonably more complex neural network with three hidden layers and thus increased fit (training) and score (prediction) time, *adam* can improve the prediction accuracy across the four suites by 14-20%, but cannot still reach the accuracy of our single hidden layer *lbfgs* models. On the other hand, the training time of the *lbfgs* models could get an order of magnitude larger than that of the *adam* or other competing models, so adding more complexity in the form of

TABLE 5. Hyperparameters of the final models.

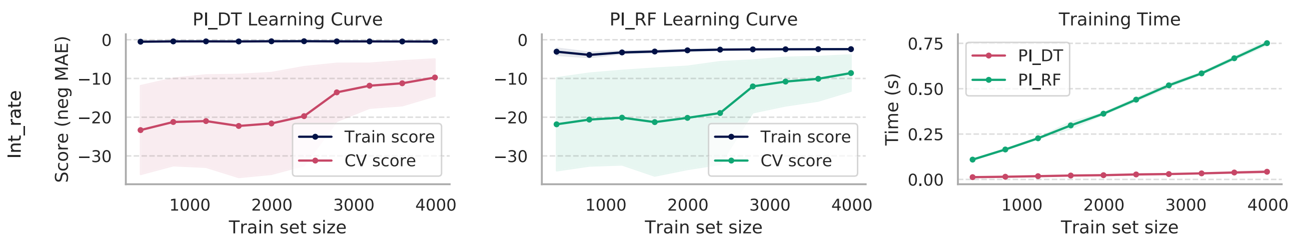
Model	Hyperparameters
MT_EN	`alpha=1e-4, l1_ratio=0.75, max_iter=5000, tol=1e-4`
DT	`max_depth=20`
RF	`n_estimators=50, max_features=0.7`
MLP	`hidden_layer_sizes=(50,), solver="lbfgs", activation="tanh", max_iter=10000, tol=5e-4`



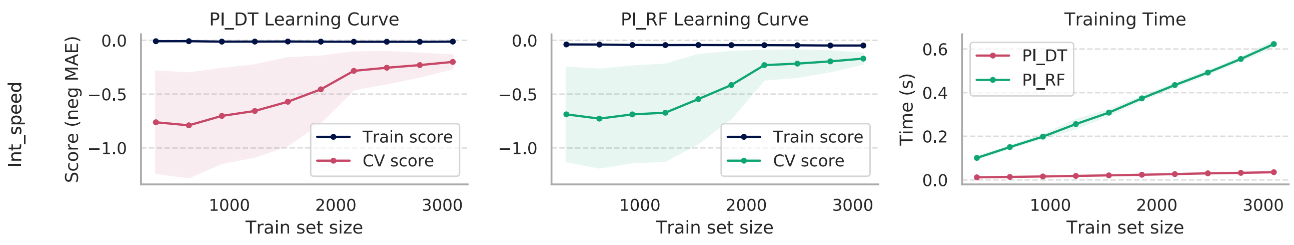
(a) FP_rate: Learning Curves and Scalability



(b) FP_speed: Learning Curves and Scalability



(c) Int_rate: Learning Curves and Scalability



(d) Int_speed: Learning Curves and Scalability

FIGURE 8. Model selection: average 10-fold cross-validation prediction error (MAE) vs. prediction time (s).

extra hidden layers to the *lbfgs* models could not be justified. These are the reasons why we have decided to choose *lbfgs* over *adam* in our final MLP model, and also why we have only considered MLP models with one hidden layer when grid searching.

The slow convergence and large training time of the MLPs are mainly due to the backpropagation method we mentioned before. As we observed, these problems get worse as the number of hidden layers increases. To alleviate such problems, non-iterative methods have been introduced. We refer readers to articles on non-iterative approaches in training feed-forward neural networks [25], such as Neural Networks with Random Weights (NNRW) [26], in which the weights between the hidden layer and input layer are randomly selected from a given range and the weights between the hidden layer and the output layer are obtained analytically.

Table 5 lists the hyperparameters of the final models. We manually picked one of the best-performing models identified by grid search for each of the four estimators, considering training time, prediction time, and goodness-of-fit (R2) metrics, in addition to the MAE (prediction error).

B. LEARNING CURVES

In this section, we analyse the models' performance as a function of the dataset size, i.e. the learning curves. The aim of this analysis is to see if we can predict the performance of future systems given the past data. For this purpose, we sort the data by date, train the model on the configuration data of the older systems and predict the performance of the newer ones. We use tree-based models as our best-preforming final models to run this experiment.

To generate the learning curves, we change the input size from 10% to 100% of the original train set size, which is itself 80% of the dataset size (as before, all the experiments are conducted on the train set split, which is 80% of the data). If we use feature selection, with smaller datasets (e.g. input sizes of 10-30%), there will be a few more features with zero variance, which will in turn result in a slightly different set of selected features in such cases. Therefore, in this analysis, we consider the complete set of features to be consistent across different dataset sizes.

Learning curves help to find out how much we can benefit from adding more training data. Fig. 8 shows that the gain using more than 70% of the available data for training is not significant, and the standard deviation (the shaded area) also gets smaller after the 70% threshold. Also, if we look back at Fig. 2 and consider the range of the outputs, even MAEs obtained with 10% (the smallest) train set can still be acceptable in certain circumstances. So, we can argue that tree-based models can be trained with a relatively small dataset of the older systems and still be useful in predicting performance of the future systems. If the training time is an issue, as shown in the last column of Fig. 8, DT models can be significantly faster than RF ones while providing almost the same levels of accuracy.

C. MODEL ASSESSMENT

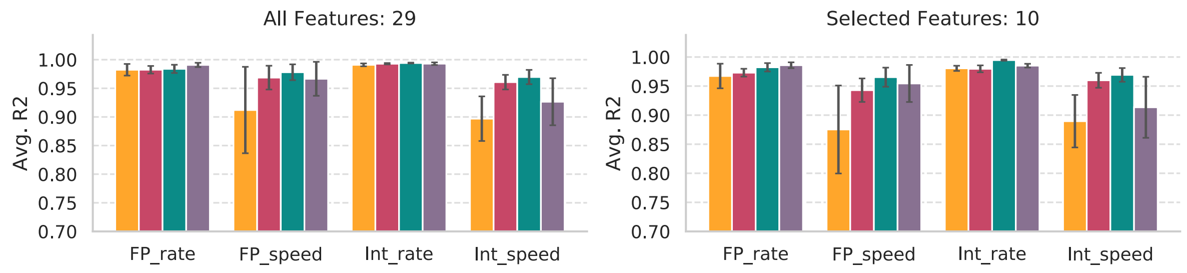
The last step is to estimate the generalisation error of the final models on new data, i.e. the test set, known as model assessment [15]. For this analysis, we use the final models with hyperparameters specified in Table 5. We train the models on the initial train set (80% of the shuffled data) and then predict the performance scores of the test set (20% of the shuffled data). Fig. 9 presents the average R2 (goodness-of-fit) and MAPE of the models across the benchmarks of each suite as well as their mean prediction delay of 10 repetitions. The error bars display the standard deviation of the measurements. Using MAPE is useful to show how the percentage error varies across the four suites.

Fig. 9 compares the final models developed using the original 29 features with each other and against the models developed with their corresponding set of 10 features. It can be observed that a very high R2 value can be achieved for the *rate* benchmarks. With 29 features, tree-based models provide an average R2 of over 0.95 as well as an average MAPE of less than of 4% in all suites (recall that the MAPE values for the simple baseline model range from 16% to 107%). It is notable that after removing almost two thirds of the features (19 out of 29), we can still accurately predict the performance scores at the cost of only 1-2% higher percentage error. Tree-based models can still provide high R2 values and MAPEs below 6% with DT and below 5% with RF. Reducing the number of features can significantly improve the training and prediction time too. As shown in Fig. 9(c), DT and RF models with 10 features can decrease the prediction time by approximately 65-80%

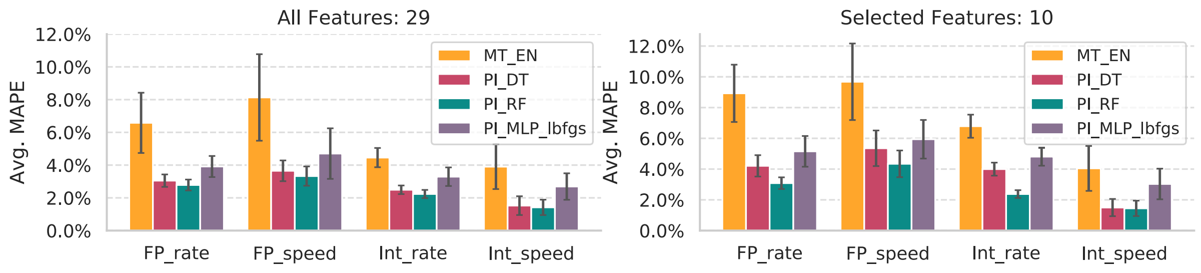
The standard deviations of the MAPE values in the tree-based models are smaller compared with the other approaches, which means that the percentage error values across the benchmarks of a suite are less dispersed. This is also evident in the radar charts in Fig. 10. Another observation is that the MLP_lbfgs models also perform well, keeping the average MAPE below 5% with 29 features and below 6% with 10 features. Unlike linear models, neural networks are able to capture complex data patterns. One of the reasons why they did not come first could be the size of the data. It would be interesting to find out how they would perform once more data (more than a few thousand samples) become available. Moreover, neural networks work best with homogeneous data, where features are similar. For data that have very different kinds of features, tree-based models can often perform better [16], as observed in our experiments. Furthermore, we need to consider that MLP_lbfgs models take a long time to train, e.g. ∼10-50s to train the models with 29 features on our system, compared with ∼0.1s for DT and ∼2-2.5s for RF.

Although here, the DT models show a slightly lower performance than the RF ones, they can generally provide a compact and interpretable representation of the decision-making process. On the other hand, while building random forests may be more time consuming, the process can be easily parallelised if need be. Moreover, with fewer number of features, the time to train the RF models will be reduced too. Therefore, if interpretability is more important, the decision trees (DT models) may be used, and if prediction accuracy is the only concern, the random forests (RF models) can provide the best performance.

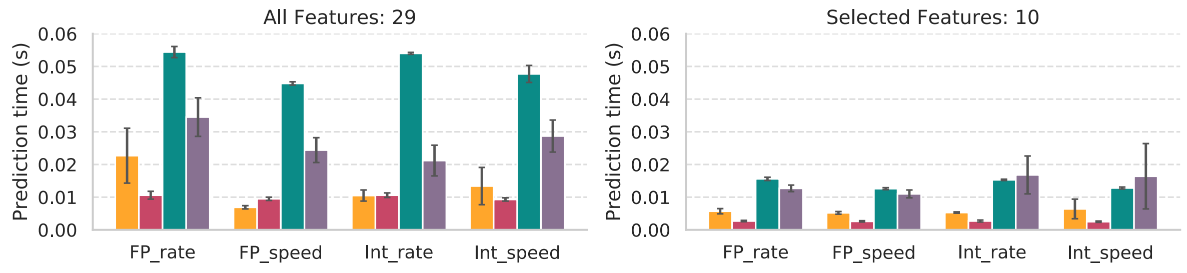
Korneva *et al.* [27] highlighted the challenges of evaluating multi-target regression models. They have argued that instead of running statistical tests on aggregated results, ranking techniques such as Pareto ranks can be used for better comparison of models. A model is Pareto-optimal on a dataset if for each target, it yields better prediction accuracy than the other models. So for each case, the Pareto-optimal models get rank 1, then the models that are optimal without considering



(a) Average R2 of the selected models across benchmarks



(b) Average MAPE of the selected models across benchmarks; lower is better



(c) Average prediction time (10 iterations) of the selected models; lower is better

FIGURE 9. Final models with the full set of 29 and selected set of 10 features: comparison of the models in terms of R2, MAPE, and prediction time.

the rank 1 models, get rank 2, and so on. They have also demonstrated that radar charts can be used to visualise such differences between the models, which is what we have also used in Fig. 10. It is evident that tree-based RF models yield the best prediction accuracy for every benchmark. After the tree-based models, the MLP_lbfgs neural networks models provide a better performance for all the individual benchmarks compared with the MT_EN linear models.

VII. RELATED WORK

SPEC benchmarks have been analysed and evaluated in several studies. For example, performance, energy, and event characterisation of SPEC CPU2017 on modern Intel processors have been discussed in [28]. They showed that *Int_speed* benchmarks provide minimal performance improvements as the thread count increases (because the majority of the integer benchmarks are not parallelisable). This is in line with our finding that the only suite where *log_cpus* is not one of the most important features is *Int_speed*. They have also pointed out that although SPEC CPU is perceived as a CPU-intensive benchmark package, it emphasises on memory and compiler efficiency too, which is again consistent with our results.

Authors in [29] have used the SimPoint statistical methodology to identify long, repetitive, and large-grain phases in programs in order to predict the performance of the SPEC benchmarks. They have compared the accuracy of the SimPoint methodology for the SPEC CPU2006 versus CPU2000. Hoste *et al.* [30] have also looked at the prediction of the performance scores of SPEC CPU2000 benchmarks on different platforms, but from a slightly different angle. Their performance prediction is based on the similarity of microarchitecture-independent characteristics of the benchmarks in the benchmark suite.

Hebbar *et al.* [31] have explored the scalability of the SPEC CPU2017 benchmarks and classified the benchmarks into groups that scale "well," "moderately," and "poorly." They have observed that *speed* benchmarks reach saturation at different levels, and only certain compute-intensive ones can benefit from the increased numbers of threads. Also, the memory-intensive *rate* benchmarks do not scale well,

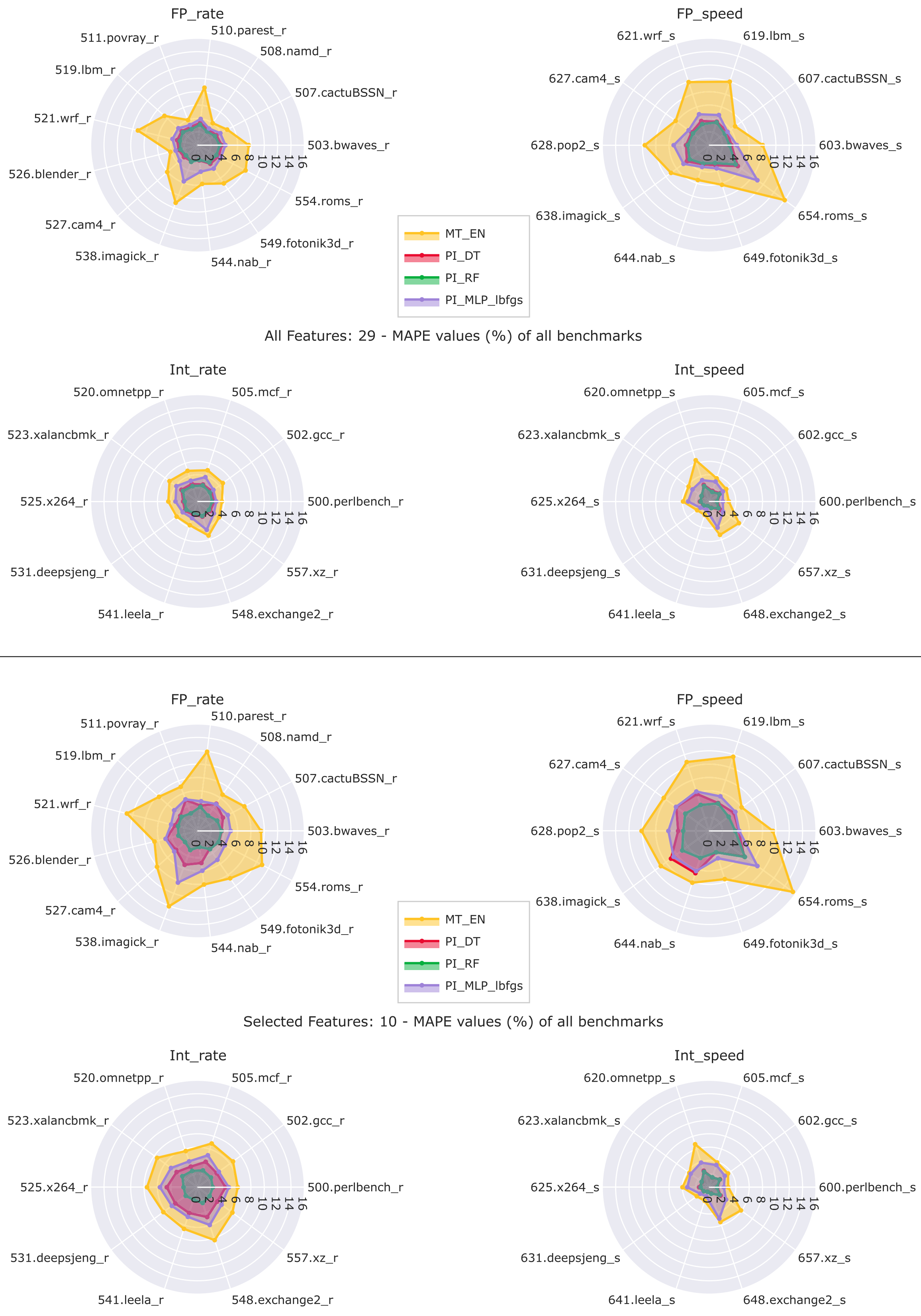


FIGURE 10. **MAPE values of final models with the full set of 29 and selected set of 10 features for all benchmarks.**

as even the highest-spec machines cannot meet their memory bandwidth requirements.

Lopez *et al.*. [24] used multiple neural networks to predict the performance of a machine, i.e. a hardware configuration. They used PCA analysis as a preprocessing step and evaluated their models on a set of 50k machine configurations using both SPEC CPU2006 and CPU2017 benchmarks. In this work, we compared their neural network model with three hidden layers of (50,100,50) nodes to our models. Moreover, we have shown that the same accuracy level can be achieved with less training data and smaller number of features as well as less sophisticated and more interpretable (e.g. decision trees) models.

Lee *et al.*. [32] have used regression models to predict both performance and power usage of the SPECjbb and SPEC2000 benchmarks. They have considered hardware parameters and used the most significant parameters to evaluate and predict the performance of benchmarks in *billions of instructions per second (bips)*. Compared to this work, they have used fine-tuned spline functions (piecewise polynomials) to transform the data and account for non-linear correlations between predictors and responses. Although their benchmarks and parameters are quite different, it is worthwhile to note that their reported mean performance prediction errors vary between 4.9% to 13% for different variance stabilised models. They have also developed application-specific models to obtain the maximum accuracy in performance prediction.

Limaye and Adegbija [33] used hardware performance counter from a modern Intel Xeon chip to characterise the CPU2017 workloads extensively. They compared the CPU2017 and CPU2006 datasets to explore similarities and differences, and also looked at the similarity and redundancy between the CPU2017 benchmarks. Using PCA and hierarchical clustering, they identified a subset of benchmarks that represents the whole CPU2017 suite.

Wang *et al.*. [3] have focused on performance prediction of SPEC CPU2006 and Geekbench 3 benchmarks on different Intel CPUs using DNN and linear regression models with L1 regularisation. In their study, there are 19 raw features including CPU specifications from the Intel CPU dataset and dynamic workload features from the Geekbench and SPEC dataset, but OS and compiler factors are not considered. They have used the standardised relative runtime (with the mean of 0 and standard deviation of 1) as the measure of performance and reported mean errors of 5.5% (DNN) and 19.9% (LR) for SPEC and 11.2% (DNN) and 20.0%(LR) for Geekbench in a 90:10 train-test split.

SPECnet [2] used TensorFlow to build a deep neural network (DNN) to predict the SPEC CPU2006 scores. In this study, we have analysed a neural network model similar to their model with the *adam* optimiser and (16,16,16) hidden layers. They also applied a train-test split of 80:20 and the features are based on both hardware and software components, except for one non-technical feature, the *Year of Publication*. The SPEC CPU2006 benchmark suites used in their study are at least three times the size of our datasets. They reported a test error of 6-7% using 13 features.

VIII. CONCLUSION

This study considers whether supervised learning can predict the performance scores of the SPEC benchmarks on parallel systems, without having to actually run the benchmarks. The extensive evaluation has shown that it is possible to accurately predict the performance of parallel and concurrent SPEC CPU2017 benchmarks. Using grid search, we have explored the effect of hyperparameters for each of the above four estimators, and selected the top-10 most accurate models. We have then compared these models together in terms of prediction time (latency) and error. The tree-based models have provided the best results. Also in the RFECV feature selection process, they reach their top performance with smaller feature sets.

We have also looked at the learning curves of the top-performing tree-based models to see how accurately we can predict the performance of future systems from the past data. We have shown that for this dataset, even 10% of the past data as the train set can sufficiently predict the future, but 70% or more data can decrease the mean error by a factor of 2 to 3. The current dataset is almost four years old. It would be interesting to see how the figures change as the dataset evolves over time and new generations of systems are added.

In the final step, we have compared the average goodness-of-fit (R2) and MAPE of the final models on the set-aside test set. We have observed that the tree-based models (DT and RF) provide the best R2 and MAPE, both on average and for individual benchmarks. In comparison with the linear models, a probable explanation is that there are still some non-linear relationships that are not captured by the linear models (MT_EN). Compared to the neural networks MLPs, a likely explanation is that tree-based models generally work better when there are different kinds of features involved. Although, neural networks may work better when the number of samples grows beyond a few thousands. So again, it would be interesting to see how these results change as the dataset expands over time. Decision tree and random forest models can keep the average MAPE under 4% with 29 features. Random forests perform better with 10 features though ($1.5\% < \text{MAPE} < 4.5\%$ across the four suites), which make them suitable for building models with smaller numbers of features. However, if interpretability is the main concern, then decision trees will be a better choice.

Compared to the previous studies, we have provided more interpretable regression models that can predict the SPEC CPU benchmarks more accurately and offered additional insight into the importance of the hardware and software features used in such models. Using the RFE method, we have found that only a few numbers of hardware and software features (less than 5) are of key importance in our models, and that with just 10 features, we can make highly accurate predictions for this dataset. Our study provides an efficient pipeline for similar performance prediction and evaluation, or design space exploration problems.

REFERENCES

[1] N. Wu and Y. Xie, "A survey of machine learning for computer architecture and systems," 2021, *arXiv:2102.07952*.

[2] D. Das, P. Raghavendra, and A. Ramachandran, "SPECNet: Predicting SPEC scores using deep learning," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, Apr. 2018, pp. 29–32.

[3] Y. Wang, V. Lee, G.-Y. Wei, and D. Brooks, "Predicting new workload or CPU performance by analyzing public datasets," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 1–21, Jan. 2019.

[4] B. Ozisikyilmaz, G. Memik, and A. Choudhary, "Machine learning models to predict performance of computer system design alternatives," in *Proc. 37th Int. Conf. Parallel Process.*, Sep. 2008, pp. 495–502.

[5] A. Tousi and C. Zhu, "Arm research starter kit: System modeling using gem5," Arm Res., U.K., Jul. 2017.

[6] J. Bucek, K.-D. Lange, and J. V. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 41–42.

[7] F. Pedregosa, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.

[8] H. Borchani, G. Varando, C. Bielza, and P. Larrañaga, "A survey on multi-output regression," *Data Mining Knowl. Discovery*, vol. 5, no. 5, pp. 216–233, 2015.

[9] D. Kocev, S. Džeroski, M. D. White, G. R. Newell, and P. Griffioen, "Using single- and multi-target regression trees and ensembles to model a compound index of vegetation condition," *Ecol. Model.*, vol. 220, no. 8, pp. 1159–1168, Apr. 2009.

[10] D. Tuia, J. Verrelst, L. Alonso, F. Perez-Cruz, and G. Camps-Valls, "Multioutput support vector regression for remote sensing biophysical parameter estimation," *IEEE Geosci. Remote Sens. Lett.*, vol. 8, no. 4, pp. 804–808, Jul. 2011.

[11] M. Kuhn *et al.*, *Applied Predictive Modeling*, vol. 26. Springer, 2013.

[12] *All Published SPEC CPU2017 Results*. Accessed: Jun. 30, 2020. [Online]. Available: https://www.spec.org/cpu2017/results/cpu2017.html

[13] D. B. Suits, "Use of dummy variables in regression equations," *J. Amer. Stat. Assoc.*, vol. 52, no. 280, pp. 548–551, Dec. 1957.

[14] W. N. Venables and B. D. Ripley, *Modern Applied Statistics With S-PLUS*. Springer, 2013.

[15] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.

[16] A. C. Müller, *Introduction to Machine Learning With Python: A Guide for Data Scientists*. Newton, MA, USA: O'Reilly Media, 2016.

[17] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *J. Roy. Statist. Soc., B Stat. Methodol.*, vol. 67, no. 2, pp. 301–320, 2005.

[18] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.

[19] B. Venkatesh and J. Anuradha, "A review of feature selection and its methods," *Cybern. Inf. Technol.*, vol. 19, no. 1, pp. 3–26, Mar. 2019.

[20] A. U. Haq, J. P. Li, J. Khan, M. H. Memon, S. Nazir, S. Ahmad, G. A. Khan, and A. Ali, "Intelligent machine learning approach for effective recognition of diabetes in E-healthcare using clinical data," *Sensors*, vol. 20, no. 9, p. 2649, May 2020.

[21] R. Pullanagari, G. Kereszturi, and I. Yule, "Integrating airborne hyperspectral, topographic, and soil data for estimating pasture quality using recursive feature elimination with random forest regression," *Remote Sens.*, vol. 10, no. 7, p. 1117, Jul. 2018.

[22] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene selection for cancer classification using support vector machines," *Mach. Learn.*, vol. 46, nos. 1–3, pp. 389–422, 2002.

[23] B. Gregorutti, B. Michel, and P. Saint-Pierre, "Correlation and variable importance in random forests," *Statist. Comput.*, vol. 27, no. 3, pp. 659–678, 2017.

[24] L. Lopez, M. Guynn, and M. Lu, "Predicting computer performance based on hardware configuration using multiple neural networks," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 824–827.

[25] X. Wang and W. Cao, "Non-iterative approaches in training feed-forward neural networks and their applications," *Soft Comput.*, vol. 22, no. 11, pp. 3473–3476, 2018.

[26] W. Cao, X. Wang, Z. Ming, and J. Gao, "A review on neural networks with random weights," *Neurocomputing*, vol. 275, pp. 278–287, Oct. 2018.

[27] E. Korneva and H. Blockeel, "Towards better evaluation of multi-target regression models," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discovery Databases*. Springer, 2020, pp. 353–362.

[28] R. H. S. Raviraj, *Spec CPU2017: Performing, Energy Event Characterization Modern Processors*. Huntsville, AL, USA: The University of Alabama in Huntsville, 2018.

[29] A. A. Nair and L. K. John, "Simulation points for SPEC CPU 2006," in *Proc. IEEE Int. Conf. Comput. Design*, Oct. 2008, pp. 397–403.

[30] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proc. 15th Int. Conf. Parallel Archit. Compilation Techn.*, 2006, pp. 114–122.

[31] R. Hebbar and A. Milenković, "A preliminary scalability analysis of spec cpu2017 benchmarks," in *Proc. SoutheastCon*, 2021, pp. 1–8.

[32] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," *ACM SIGOPS Oper. Syst. Rev.*, vol. 40, no. 5, pp. 185–194, 2006.

[33] A. Limaye and T. Adegbija, "A workload characterization of the SPEC CPU2017 benchmark suite," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Apr. 2018, pp. 149–158.

ASHKAN TOUSI received the Ph.D. degree in computing science (parallel computing) from the University of Glasgow, U.K., in 2015.

He worked in the areas of system design and edge computing at Arm Research. Since 2018, he has been working as a Research Fellow at The University of Manchester. He is currently a Principal Engineer at Safeguard Global, where he leads a team of cloud and data engineers. His research interests include distributed and cloud computing as well as the use of machine learning in performance analysis and optimization.

MIKEL LUJÁN received the Ph.D. degree in computer science from The University of Manchester, U.K., in 2002.

He is currently a Professor with the Department of Computer Science, The University of Manchester, where he holds the ARM/Royal Academy of Engineering Research Chair on Computer Systems. His research interests include runtime environments, low-power architectures, and application-specific systems and optimizations.