12/6/2025

CPU Performance Prediction

A Linear Regression Model

Webb, Will
UNIVERSITY OF KANSAS

Model Briefing

Predictive Modeling in machine learning allows for the complete leverage over large datasets with the objective of producing the most reliable results from an algorithm. At the beginning of any algorithm stages, after the data has been identified, it's worth considering what problem the machine learning algorithm will be designed to solve. The concept of a model being good is difficult to generalize, however, in respect to the objective, a model should be capable of performing accurate results on unseen data. In these next few sections, we'll cover the fundamental procedure behind modeling linear regressions, followed by its application towards the CPU prediction model created.

A guiding formula can often be used when selecting various models for an algorithm. This report will only cover supervised learning algorithm methodologies to algin with the project's construction. A supervised learning algorithm is essentially a dataset written as a set of example-label pairs shown by *equation 1*

$$\{(x_{1,}y_{1,}),\ldots,(x_{n,}y_{n,})\}$$

Equation 1: Supervised learning vector representation

From *equation 1*, y_n is the single predicted function, label, that is dependent on the example x_n, the data.

After any data is in the appropriate vector representation, deciding a predictor, constructing a predictive function that solves for a single output y_n, is often suggested next in the procedure. This suction will introduce selecting a predictor as a function that produces a single, scalar output. Given the set of training inputs x_n this function for a linear regression corresponding noisy observation

$$y_n = f(x_n) + \varepsilon$$

Equation 2: Generalized Linear regression predictor

Where ε is a random variable that describes the noise measurement and any potential unmodeled processes; Moreover, for some input value x_n and *Equation 2* observing noisy functions, the objective becomes finding the function of f and generalize well with new input locations.

The development of y_n in *equation 2* will be covered in the following section.

Modeling a Linear Regression

The basis for finding a regression function requires a formulated outline. This outline will be focused on procedural design of Ordinary Least Squares regression to align with the framework for the project. Initially, given a dataset, a function class that best fits the data needs to be chosen in the form of *equation 2*. After choosing a model, defining a good parameter requires finding a loss function that optimizes to the minimum loss across the predictor. Next, recognizing when the model overfits is import in a regression model because the training data won't generalize to unseen data.

Ordinary Least Squares (OLS)

When selecting the function class, because of the presence of noise function, ε in *equation 2* a probabilistic approach needs to be taken using a likelihood function.

$$p(y \mid x) = N(y \mid f(x), \sigma^2)$$

Equation 3: Regression with likelihood function.

Where $x \in \mathbb{R}^D$ are inputs and $y \in \mathbb{R}$ are noisy function targets. ε is independent Gaussian measurement with mean 0 and variance σ^2.

$$y = x^T\theta + \varepsilon, \varepsilon \sim \mathrm{N}(0,\sigma^2)$$

Equation 4: Linear Regression predictor

To evaluate a good parameter, θ, that is like the unknown function that generated that data, A parametrized function is then chosen. In ordinary least squares, a linear regression is parametrized linearly by the model in *equation 4*.

Parameter Formatting for OLS

Given a training set, *D,* that's represented similarly to *equation 1* and consists of *N* inputs $x \in \mathbb{R}^D$ and its corresponding *N* labels $y \in \mathbb{R}$, *Equation 4* can evaluate to find a good parameter set, θ. A general idea when looking forward is for a label y_n to be conditionally independent given an example x_n.

These conditions would allow for the likelihood to be factorized as

$$p(\mathrm{y} \mid \mathrm{X}, \theta) = p(y_1, \ldots, y_n \mid x_1, \ldots, x_n, \theta)$$

Equation 5: Factorized likelihood

Where *equation 5* yields the set of training inputs and targets; additionally, the factor of *equation 5* is gaussian due to the noise distribution. With this set, optimal parameters $\theta^* \in \mathbb{R}^D$ will need to be found for the regression model. Once this set of parameters are found, function values can be predicted by utilizing *Equation 4*. This, in turn, creates an optimized distribution of

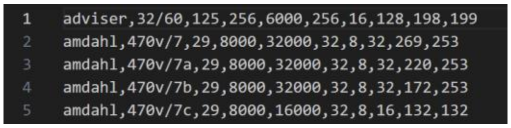
$$p(y_* \mid x_*, \theta^*) = N(y_* \mid x_*^T \theta^*, \sigma^2)$$

Equation 6: optimizes parameter estimation

Equation 6 is the extent to what this briefing will cover, the project is largely technical and not very theoretical so a basic understanding of regression theory will contribute towards the understand of the model – which we'll building constructing in the following section.

Selecting a Dataset

The data selected was the open source, Computer Hardware dataset from UCI Machine learning repository. The file was in a .txt format with 209 instances with an integer feature type. Upon first glances, there isn't much data to work with in this file; this makes the regression model susceptible to overfitting if overly optimized.



```
adviser,32/60,125,256,6000,256,16,128,198,199
amdahl,470v/7,29,8000,32000,32,8,32,269,253
amdahl,470v/7a,29,8000,32000,32,8,32,220,253
amdahl,470v/7b,29,8000,32000,32,8,32,172,253
amdahl,470v/7c,29,8000,16000,32,8,16,132,132
```

Picture 1: First 5 instances of the machine.data.txt file

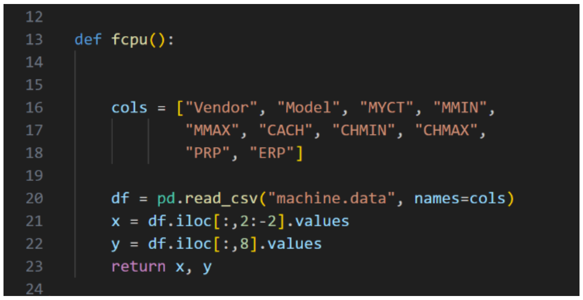
Picture 1 shows the head data set that will be used. The following columns are:

1. Vendor name
2. Model Name
3. Machine cycle time in nanosecond
4. Minimum main memory in kilobytes
5. Maximum main memory in kilobytes
6. CACH memory in kilobytes
7. Minimum channels in units
8. Maximum channels in units
9. Published relative performance
10. Estimated relative performance from the origin

Upon having surveyed the data set, next objective is to clean the data set before implementing a linear regression to act preemptively towards the model producing unreliable data

Preprocessing Dataset

In *picture 1*, and the rest of the data set, every iteration of column 9, Published relative performance, will be the label, y_n, that will be the chosen predictor to generalize any unseen data; therefore, we can review the dataset to identify any redundant columns that are independent of our predictor, y_n. Omitting any features that aren't inherently integers is a good place to start, this includes: vendor name and Model name. Following that, the estimated relative performance from the origin, column 10, is just another predictor that could have been chosen; likewise, it also won't be trained on.



```python

def fcpu():


    cols = ["Vendor", "Model", "MYCT", "MMIN",
            "MMAX", "CACH", "CHMIN", "CHMAX",
            "PRP", "ERP"]

    df = pd.read_csv("machine.data", names=cols)
    x = df.iloc[:,2:-2].values
    y = df.iloc[:,8].values
    return x, y

```

Picture 2: Preprocessed data in python using Pandas

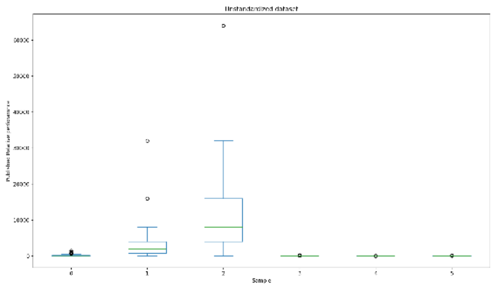
Implied by *picture 2*, all programming will be done in Python for its accessibility to implement ML strategies. The libraries used in this environment include pandas, matplotlib, and scikit-learn, and the IDE editor is in VScode.

All data cleaning was initially written using the panda's library for the function fcpu (). Each column is indexed to its corresponding name for testing purposes when analyzing data in the terminal. Iloc is also an attribute used from the panda's library that allows for indexing data frames. Shown in *picture 2* the examples, x_n, are defined under the conditions stated previously: Examples can only include columns that the label is dependent on. Iloc assigns the label, y_n, as every row of the PRP column – also stated previously.

This dataset was especially useful for a supervised learning environment because of its natural cleanliness: No missing values needed to be handled so I could move on to standardizing without having to implement an uncertainty model.

Visualizing Raw Performance

Even though the dataset is generally clean, there are still subtle irregularities in the numerical values, for example-labeled pairs. These irregularities can show as a result large standard deviation, resulting large anomalies that need to be addressed. A good way of visualizing raw data is using box charts

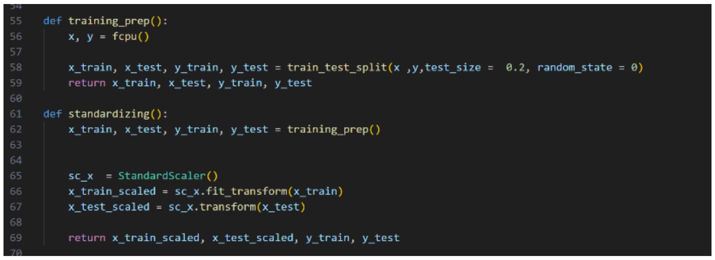


Picture 3: Box chart of the filtered machine.data.txt file

In *Picture 3*, I coded a testing function that visualizes the selected example-labeled pairs. *Picture* 3 suggests outliers in the form of dots as well as a skewed distribution that needs to be symmetric; therefore, the data still needs to be standardized before moving onto the regression model.

Standardizing

The procedure for scaling the data included splitting arrays into training and test subsets, then applying a Z-score standardizing method that transforms a single example, x_n, by removing its mean value and scaling down to a unit variance among all data between 0-1. This will create a more robust dataset that we'll visualize using box charts once implemented



```
def training_prep():
    x, y = fcpu()

    x_train, x_test, y_train, y_test = train_test_split(x ,y,test_size =  0.2, random_state = 0)
    return x_train, x_test, y_train, y_test

def standardizing():
    x_train, x_test, y_train, y_test = training_prep()


    sc_x  = StandardScaler()
    x_train_scaled = sc_x.fit_transform(x_train)
    x_test_scaled = sc_x.transform(x_test)

    return x_train_scaled, x_test_scaled, y_train, y_test
```

Picture 4: Standardizing dataset

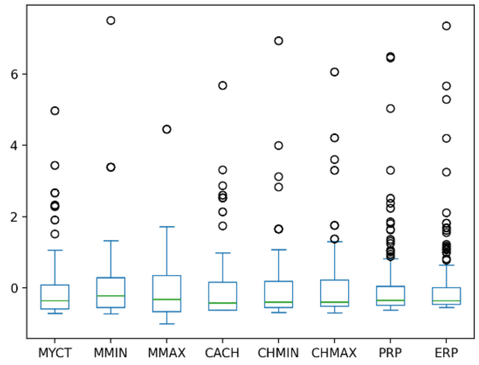
The Scikit Learn library was imported into python to fulfill these requirements. From the library, the following classes were used:

- From sklearn.model_selection import train_test_split
- From sklearn.preprocessing import StandardScaler

From these imports, *picture 4* was then constructed. The function training_prep () consists of training and testing variables assigned to a random subset of the raw data produced by *picture 2*. Knowing the dataset is already small, the training size has been set to 0.2 to generalize well. The standardizing function then assigns the testing and training examples, x_train and x_test, to scale to the Z score method previously stated.

Testing Standardized Performance

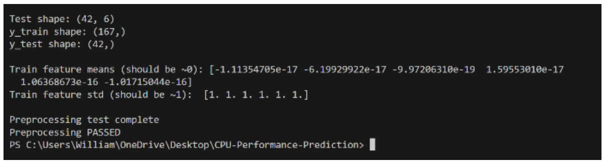
Now that the function has been standardized, we can visualize it using the box method.



Picture 5: Box chart of standardized machine.data.txt file

Compared to *picture 3*, the dataset is now looking a lot more symmetrical. Even though there appears to be more outliers, the range of variants is a lot smaller and shouldn't define the results of the model too much. The model is almost ready to implement a linear regression; however, there's one more test that'll verify all the preprocessing that's been completed.

There are criteria that the model needs to have now that it's been standardized, which includes: the examples trained and tested on need to have the same dimension, The data can't have any NaN values, the mean of the examples trained on need to be ~0, and the standard deviation need to be ~1. For this testing function I used the NumPy library to approximate the mean and standard deviation with the np.allclose() method.



```
Test shape: (42, 6)
y_train shape: (167,)
y_test shape: (42,)

Train feature means (should be ~0): [-1.11354705e-17 -6.19929922e-17 -9.97206310e-19  1.59553010e-17
  1.06368673e-16 -1.01715044e-16]
Train feature std (should be ~1):  [1. 1. 1. 1. 1. 1.]

Preprocessing test complete
Preprocessing PASSED
PS C:\Users\William\OneDrive\Desktop\CPU-Performance-Prediction> 
```

Picture 6: Preprocessing test

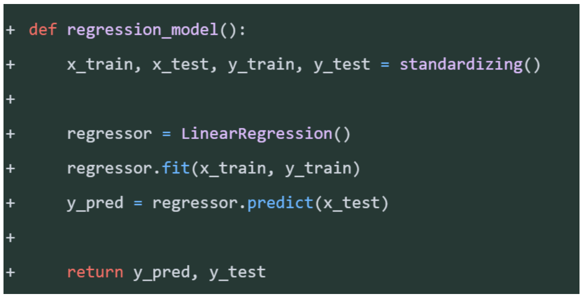
Picture 6 verifies that the preprocessing implementation has passed. Moving forward, the linear regression will now be implemented.

Linear Regression Iteration No.1

This first iteration of a linear regression model was implemented directly after the preprocessing procedure was completed and verified in *picture 6*. from scikit learn, the following class was imported into the model:

- from sklearn.linear_model import LinearRegression

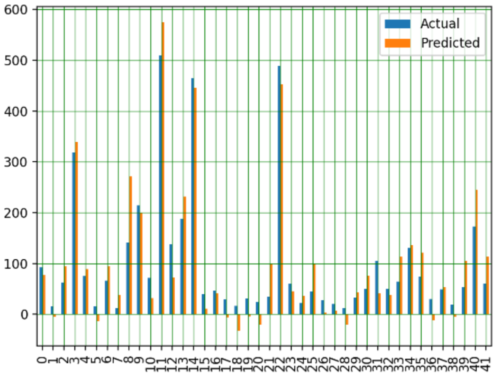
This linear regression class provided by scikit learn follows an ordinary least squares regression model that aims to "minimize the residual sum of the squared between the observed targets in the dataset, and the targets predicted by the linear approximation" source. A concept where we saw in the first two sections of the report when a predictor minimized with a loss function.



```
+ def regression_model():
+     x_train, x_test, y_train, y_test = standardizing()
+
+     regressor = LinearRegression()
+     regressor.fit(x_train, y_train)
+     y_pred = regressor.predict(x_test)
+
+     return y_pred, y_test
```

Picture 7: GitHub documentation of first linear regression iteration

The preprocessed data was directly fit into a linear regression model shown in *picture 7*. Here, the model fit according to the example-labeled training data and then the predictor was modeled for unseen data with the testing examples as its parameter.



Picture 8: Supervised Regression Model trained on CPU performance data.

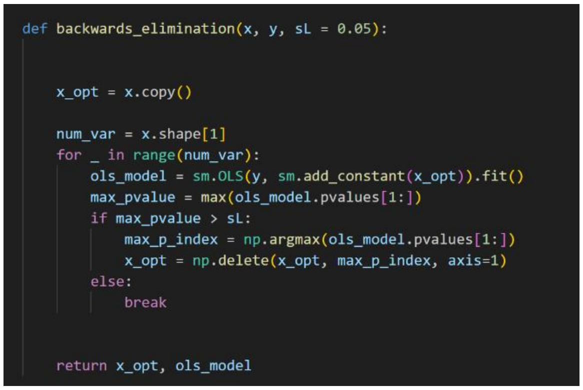
Matplotlib was then used to plot a bar graph comparison between all 209 instances of the actual data and the predicted data trained on only 20% of this data. *Picture* 8 has the predictor, published relative performance, is on the y axis and the exampled data is on the x axis.

The results show generalizing decently to unseen data. Some data overfits but nothing catastrophic. There are a few reasons this comparison isn't close though: there is not enough data and various examples to train on. Originally, when cleaning the data, a function was designed to omit any independent columns that the predictor obviously wouldn't depend on. The design process stopped there, but a statistical technique could be used to model the relationship between the predictor, PRP, and its multiple independent variables. This model would then iterate through each example vector and select only the most significant predictors. This technique is referred to as Backwards Elimination for multiple linear regression and is designed in the second iteration for improving mode efficiency and interpretability

Backwards Elimination

Linear Regression Iteration NO.2

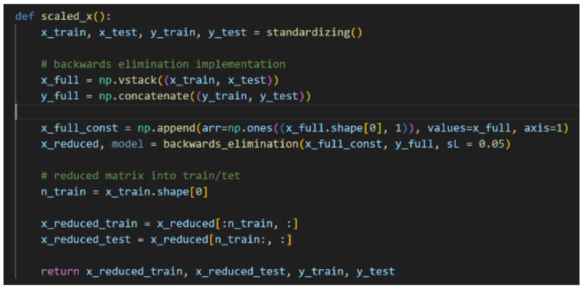
For this final iteration, the Statsmodel API was downloaded for assistance in implementing that backwards elimination technique



```python
def backwards_elimination(x, y, sL = 0.05):


    x_opt = x.copy()

    num_var = x.shape[1]
    for _ in range(num_var):
        ols_model = sm.OLS(y, sm.add_constant(x_opt)).fit()
        max_pvalue = max(ols_model.pvalues[1:])
        if max_pvalue > sL:
            max_p_index = np.argmax(ols_model.pvalues[1:])
            x_opt = np.delete(x_opt, max_p_index, axis=1)
        else:
            break


    return x_opt, ols_model
```

Picture 9: optimizing predictors with backwards elimination

The code in *picture 9* starts with the entire model with all predictors using the Statsmodel API as sm which fits for ordinary least squares. Sequentially, for each iteration, conditions are placed to remove the predictor with the highest significant level. NumPy is the primary library used to set these conditions: methods such as np.argmax and np.delete allow for easy manipulation over this numeric data. The process stops when all predictors have values below the significant level. These predictors as well as the fitted OLS model are then returned.



```python
def scaled_x():
    x_train, x_test, y_train, y_test = standardizing()

    # backwards elimination implementation
    x_full = np.vstack((x_train, x_test))
    y_full = np.concatenate((y_train, y_test))

    x_full_const = np.append(arr=np.ones((x_full.shape[0], 1)), values=x_full, axis=1)
    x_reduced, model = backwards_elimination(x_full_const, y_full, sL = 0.05)

    # reduced matrix into train/tet
    n_train = x_train.shape[0]

    x_reduced_train = x_reduced[:n_train, :]
    x_reduced_test = x_reduced[n_train:, :]

    return x_reduced_train, x_reduced_test, y_train, y_test
```

Picture 10: splitting optimized training and testing data.

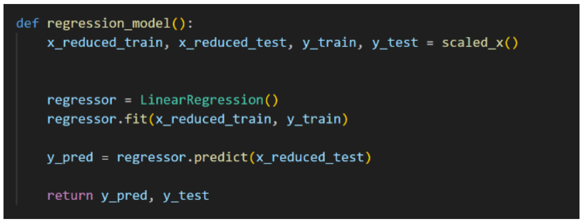
NumPy recombines the predictors into row-wise arrays as well as combine the labels, y_n, of the testing and training set. These recombined arrays of example-label pairs are needed when adding an intercept term in the model.

$$y = b_0 + b_1x_1 + \ldots + b_nx_n + \varepsilon$$

Equation 7: General equation for a multi linear regression

Statsmodel doesn't include an interception b_0 of the multi linear regression in *equation 7*. This inception is a constant term in the linear model that accounts for the value of the target variable when all features are set to zero: this helps determine the baseline relationship between features and the target variables.

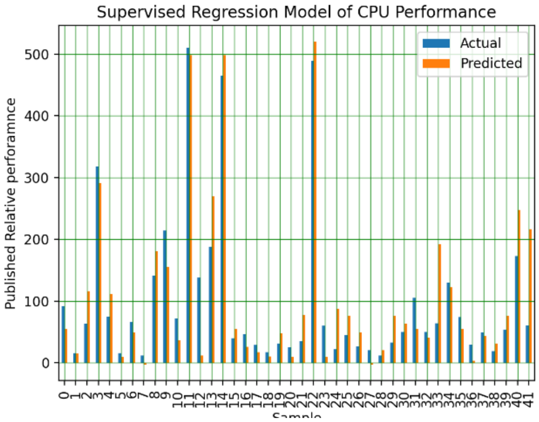
NumPy is used to create this intercept shown in *picture 10* which is then passed as a parameter into the previous backwards elimination as the vector of features, where the labels are the array of training and testing data. The features are then reduced once again into matrices of training and testing sets.



```python
def regression_model():
    x_reduced_train, x_reduced_test, y_train, y_test = scaled_x()


    regressor = LinearRegression()
    regressor.fit(x_reduced_train, y_train)

    y_pred = regressor.predict(x_reduced_test)

    return y_pred, y_test
```

Picture 11: Regression model with backwards elimination iteration

The predictor, y_n, is then predicted again fitted to the backwards elimination algorithm where *picture 12* shows the resulting bar graph



Picture 12: Regression model with backwards elimination implementation

Final Statement

The objective of this project was to predict CPU performance by modeling under a supervised learning linear regression algorithm. The data used was open sourced on UCI Machine Learning Repository with a limited number of instances that could be trained on. The language programmed in was python with the library's pandas, NumPy, Matplotlib, and Sci kit learn. These libraries were useful for cleaning, preprocessing, visualizing, and applying regression models too.

The first iteration fits the preprocessed data directly into a linear regression model and the results are shown in *picture 8*; alternatively, the second iteration applies a backward elimination model before fitting to a linear regression model to account or the multi-independent features that the dataset has. This second iteration is shown in *picture 12.* This iteration showed to generalize better compared to the first iteration on unseen data.