

5.2: Training and Testing

With the data set partitioned into two randomly selected portions, we can train the model on the first portion, and test it on the second portion. Figure 5.1 shows the overall flow of this training and testing process. We next explain the details of this process to train and test the model we previously developed for the Int2000 benchmark results.

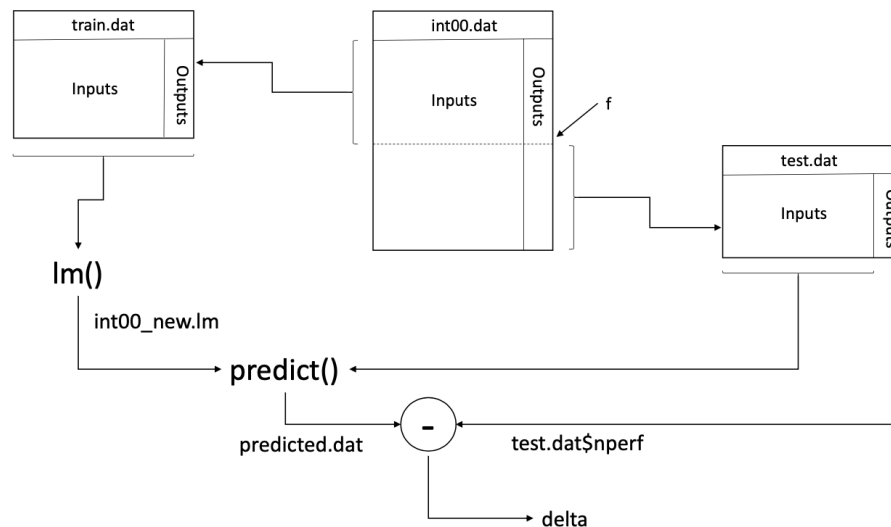


Figure 5.1: The training and testing process for evaluating the predictions produced by a regression model.

The following statement calls the `lm()` function to generate a regression model using the predictors we identified in Chapter 4 and the `train.dat` data frame we extracted in the previous section. It then assigns this model to the variable `int00_new.lm`. We refer to this process of computing the model's coefficients as *training* the regression model.

```
int00_new.lm <- lm(nperf ~ clock + cores + voltage + channel + L1icache +
  sqrt(L1icache) + L1dcache + sqrt(L1dcache) + L2cache + sqrt(L2cache), data = train.dat)
```

The `predict()` function takes this new model as one of its arguments. It uses this model to compute the predicted outputs when we use the `test.dat` data frame as the input, as follows:

```
predicted.dat <- predict(int00_new.lm, newdata=test.dat)
```

We define the difference between the predicted and measured performance for each processor i to be $\Delta_i = \text{Predicted}_i - \text{Measured}_i$, where Predicted_i is the value predicted by the model, which is stored in the data frame `predicted.dat`, and Measured_i is the actual measured performance response, which we previously assigned to the `test.dat` data frame. The following statement computes the entire vector of these Δ_i values and assigns the vector to the variable `delta`.

```
delta <- predicted.dat - test.dat$nperf
```

Note that we use the `$` notation to select the column with the output value, `nperf`, from the `test.dat` data frame.

The mean of these Δ differences for n different processors is:

$$\bar{\Delta} = \frac{1}{n} \sum_{i=1}^n \Delta_i$$

A confidence interval computed for this mean will give us some indication of how well a model trained on the `train.dat` data set predicted the performance of the processors in the `test.dat` data set. The `t.test()` function computes a confidence interval for the desired confidence level of these Δ_i values as follows:

```
> t.test(delta, conf.level = 0.95)
One Sample t-test

data: delta
t = -0.65496, df = 41, p-value = 0.5161
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval: -2.232621 1.139121
sample estimates: mean of x -0.5467502
```

If the prediction were perfect, then $\Delta_i = 0$. If $\Delta_i > 0$, then the model predicted that the performance would be greater than it actually was. A $\Delta_i < 0$, on the other hand, means that the model predicted that the performance was lower than it actually was. Consequently, if the predictions were reasonably good, we would expect to see a tight confidence interval around zero. In this case, we obtain a 95 percent confidence interval of $[-2.23, 1.14]$. Given that *nperf* is scaled to between 0 and 100, this is a reasonably tight confidence interval that includes zero. Thus, we conclude that the model is reasonably good at predicting values in the `test.dat` data set when trained on the `train.dat` data set.

Another way to get a sense of the predictions' quality is to generate a scatter plot of the Δ_i values using the `plot()` function:

```
plot(delta)
```

This function call produces the plot shown in Figure 5.2. Good predictions would produce a tight band of values uniformly scattered around zero. In this figure, we do see such a distribution, although there are a few outliers that are more than ten points above or below zero.

It is important to realize that the `sample()` function will return a different random permutation each time we execute it. These differing permutations will partition different processors (i.e., rows in the data frame) into the train and test sets. Thus, if we run this experiment again with exactly the same inputs, we will likely get a different confidence interval and Δ_i scatter plot. For example, when we repeat the same test five times with identical inputs, we obtain the following confidence intervals: $[-1.94, 1.46]$, $[-1.95, 2.68]$, $[-2.66, 3.81]$, $[-6.13, 0.75]$, $[-4.21, 5.29]$. Similarly, varying the fraction of the data we assign to the train and test sets by changing `f = 0.5` also changes the results.

It is good practice to run this type of experiment several times and observe how the results change. If you see the results vary wildly when you re-run these tests, you have good reason for concern. On the other hand, a series of similar results does not necessarily mean your results are good, only that they are consistently reproducible. It is often easier to spot a bad model than to determine that a model is good.

Based on the repeated confidence interval results and the corresponding scatter plot, similar to Figure 5.2, we conclude that this model is reasonably good at predicting the performance of a set of processors when the model is trained on a different set of processors executing the same benchmark program. It is not perfect, but it is also not too bad. Whether the differences are large enough to warrant concern is up to you.

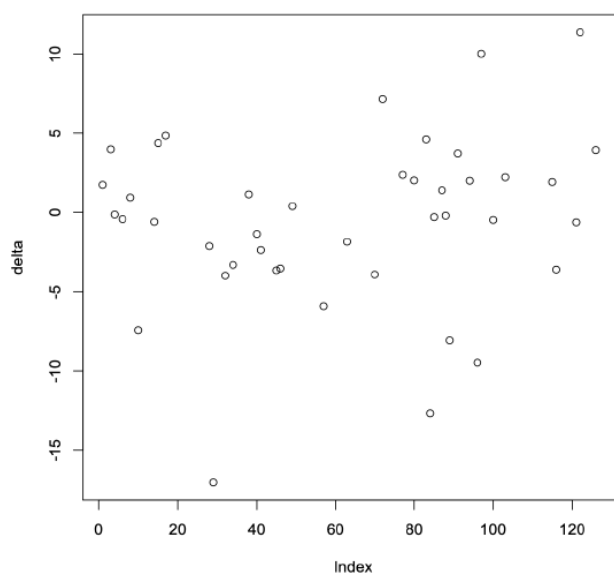


Figure 5.2: An example scatter plot of the differences between the predicted and actual performance results for the Int2000 benchmark when using the data-splitting technique to train and test the model.

This page titled [5.2: Training and Testing](#) is shared under a [CC BY-NC 4.0](#) license and was authored, remixed, and/or curated by [David Lilja](#) ([University of Minnesota Libraries Publishing](#)) via [source content](#) that was edited to the style and standards of the LibreTexts platform.