Chapter 4 Evaluation Metrics

After you make predictions, you need to know if they are any good. There are standard measures that we can use to summarize how good a set of predictions actually is. Knowing how good a set of predictions is allows you to make estimates about the skill of a given machine learning model of your problem. In this tutorial, you will discover how to implement four standard prediction evaluation metrics from scratch in Perl. After reading this tutorial, you will know:

- How to implement classification accuracy.
- How to implement and interpret a confusion matrix.
- How to implement mean absolute error for regression.
- How to implement root mean squared error for regression.

Let's get started.

4.1 Description

You must estimate the quality of a set of predictions when training a machine learning model. Performance metrics like classification accuracy and root mean squared error can give you a clear objective idea of how good a set of predictions is, and in turn how good the model is that generated them. This is important as it allows you to tell the difference and select among:

- Different transforms of the data used to train the same machine learning model.
- Different machine learning models trained on the same data.
- Different configurations for a machine learning model trained on the same data.

As such, performance metrics are a required building block in implementing machine learning algorithms from scratch.

4.2. Tutorial

This tutorial is divided into 4 parts:

1. Classification Accuracy.
2. Confusion Matrix.
3. Mean Absolute Error.
4. Root Mean Squared Error.

These steps will provide the foundations you need to handle evaluating predictions made by machine learning algorithms.

4.2.1 Classification Accuracy

A quick way to evaluate a set of predictions on a classification problem is by using accuracy. Classification accuracy is a ratio of the number of correct predictions out of all predictions that were made. It is often presented as a percentage between 0% for the worst possible accuracy and 100% for the best possible accuracy.

accuracy = correct predictions x 100 / total predictions

(4.1)

We can implement this in a function that takes the expected outcomes and the predictions as arguments. Below is this function named accuracy_metric() that returns classification accuracy as a percentage. Notice that we use == to compare the equality actual to predicted values. This allows us to compare integers or strings, two main data types that we may choose to use when loading classification data.

In [1]:
```
use strict;
use warnings;
use Data::Dump qw(dump);
use List::Util qw(zip min max sum uniq);
use sml; # Statistical Machine Learning Library
IPerl->load_plugin('Chart::Plotly');
```

In [9]:
```
# Defined in Section 4.2.1 Classification Accuracy
# Function To Calculate Classification Accuracy.
# Calculate accuracy percentage between two lists
my $accuracy_metric = sub{
  my ($self, $actual, $predicted) = @_;
  my $correct = 0;
  for my $pair (zip $actual, $predicted){
    $correct++ if ($pair->[0] == $pair->[1]);
  }
  return sprintf '%0.1f', $correct / @$actual * 100.0;
};

sml->add_to_class('sml', 'accuracy_metric', $accuracy_metric);
```

Out[9]:
```
*sml::accuracy_metric
```

```
Subroutine sml::accuracy_metric redefined at /usr/local/lib/perl5/site_perl/5.
32.1/x86_64-linux/sml.pm line 22.
```

We can contrive a small dataset to test this function. Below are a set of 10 actual and predicted integer values. There are two mistakes in the set of predictions.

Running this example produces the expected accuracy of 80% or 8/10.

In [10]:
```
# Example of calculating classification accuracy

# Test accuracy
my $actual    = [0,0,0,0,0,1,1,1,1,1];
my $predicted = [0,1,0,0,0,1,0,1,1,1];
print "actual\tpredicted\n";
print $_->[0], "\t", $_->[1], "\n" for zip ($actual, $predicted);
my $accuracy  = sml->accuracy_metric($actual, $predicted);
print $accuracy;
```

```
# Example Output From Calculating Classification Accuracy.
# actual        predicted
# 0     0
# 0     1
# 0     0
# 0     0
# 0     0
# 1     1
# 1     0
# 1     1
# 1     1
# 1     1
# 80.0
```

```
actual  predicted
0       0
0       1
0       0
0       0
0       0
1       1
1       0
1       1
1       1
1       1
80.0
```

Out[10]:

```
1
```

Accuracy is a good metric to use when you have a small number of class values, such as 2, also called a binary classification problem. Accuracy starts to lose it's meaning when you have more class values and you may need to review a different perspective on the results, such as a confusion matrix.

4.2.2 Confusion Matrix

A confusion matrix provides a summary of all of the predictions made compared to the expected actual values. The results are presented in a matrix with counts in each cell. The counts of predicted class values are summarized horizontally (rows), whereas the counts of actual values for each class values are presented vertically (columns). A perfect set of predictions is shown as a diagonal line from the top left to the bottom right of the matrix. The value of a confusion matrix for classification problems is that you can clearly see which predictions were wrong and the type of mistake that was made. Let's create a function to calculate a confusion matrix. We can start off by defining the function to calculate the confusion matrix given a list of actual class values and a list of predictions. The function is listed below and is named confusion matrix(). It first makes a list of all of the unique class values and assigns each class value a unique integer or index into the confusion matrix. The confusion matrix is always square, with the number of class values indicating the number of rows and columns required. Here, the first index into the matrix is the row for actual values and the second is the column for predicted values. After the square confusion matrix is created and initialized to zero counts in each cell, it is a matter of looping through all predictions and incrementing the count in each cell. The function returns two objects. The first is the set of unique class values, so that they can be displayed when the confusion matrix is drawn. The second is the confusion matrix itself with the counts in each cell.

In [11]:
```
# Example of Calculating and Displaying a Pretty Confusion Matrix

# Function To Calculate a Confusion Matrix.
# calculate a confusion matrix
my $confusion_matrix = sub{
  my ($self, $actual, $predicted) = @_;

  my @unique = uniq @$actual;
  my $matrix = [map {[]} 0 .. $#unique];
  for my $i (0 .. $#unique){
    $matrix->[$i] = [0, map {$_} 0 .. $#unique -1];
  }
  my (%lookup, $x, $y);
  while (my ($i, $value) = each @unique) {
    $lookup{$value} = $i;
  }
  for my $i (0 .. $#$actual){
    $x = $lookup{$actual->[$i]};
    $y = $lookup{$predicted->[$i]};
    $matrix->[$x][$y] += 1;
  }

  return \@unique, $matrix;
};

sml->add_to_class('sml', 'confusion_matrix', $confusion_matrix);
```

Out[11]:
```
*sml::confusion_matrix
```

```
Subroutine sml::confusion_matrix redefined at /usr/local/lib/perl5/site_perl/
5.32.1/x86_64-linux/sml.pm line 22.
```

Let's make this concrete with an example. Below is another contrived dataset, this time with 3 mistakes.

In [12]:
```
# Example of a Set of Contrived Predictions and Expected Values.
my $actual    = [0,0,0,0,0,1,1,1,1,1];
my $predicted = [0,1,1,0,0,1,0,1,1,1];
print "actual\tpredicted\n";
print $_->[0], "\t", $_->[1], "\n" for zip ($actual, $predicted);
```

```
actual  predicted
0       0
0       1
0       1
0       0
0       0
1       1
1       0
1       1
1       1
1       1
```

We can calculate and print the confusion matrix for this dataset as follows:

In [72]:
```
# Test confusion matrix with integers
my ($unique, $matrix) = sml->confusion_matrix($actual, $predicted);
print dump $unique;
print "\n", dump $matrix;
# Example Output From Calculating a Confusion Matrix.
# [0, 1]
# [[3, 2], [1, 4]]
```

```
[0, 1]
[[3, 2], [1, 4]]
```
Out[72]:
```
1
```

It’s hard to interpret the results this way. It would help if we could display the matrix as intended with rows and columns. Below is a function to correctly display the matrix. The function is named print confusion matrix(). It names the columns as Z for Actual and the rows as P for Predicted. Each column and row are named for the class value to which it corresponds. The matrix is laid out with the expectation that each class label is a single character or single digit integer and that the counts are also single digit integers. You could extend it to handle large class labels or prediction counts as an exercise.

In [21]:
```
# Function To Pretty Print a Confusion Matrix.
# pretty print a confusion matrix
my $print_confusion_matrix = sub{
  my ($self, $unique, $matrix) = @_;
  print 'A/P ' . join(' ', map {$_} @$unique), "\n";
  while (my ($i, $x) = each @$unique){
    print sprintf " %s| %s\n", $x, join(' ', map {$_} @{$matrix->[$i]});
  }
};

sml->add_to_class('sml', 'print_confusion_matrix', $print_confusion_matrix);
```

Out[21]:
```
*sml::print_confusion_matrix
```

```
Subroutine sml::print_confusion_matrix redefined at /usr/local/lib/perl5/site_
perl/5.32.1/x86_64-linux/sml.pm line 22.
```

Running the example produces the output below. We can see the class labels of 0 and 1 across the top and bottom. Looking down the diagonal of the matrix from the top left to bottom right, we can see that 3 predictions of 0 were correct and 4 predictions of 1 were correct. Looking in the other cells, we can see 2 + 1 or 3 prediction errors. We can see that 2 predictions were made as a 1 that were in fact actually a 0 class value. And we can see 1 prediction that was a 0 that was in fact actually a 1.

In [22]:
```
# Example Output From Printing a Pretty Confusion Matrix.
# Test confusion matrix with integers
my $actual    = [0,0,0,0,0,1,1,1,1,1];
my $predicted = [0,1,1,0,0,1,0,1,1,1];
my ($unique, $matrix) = sml->confusion_matrix($actual, $predicted);
sml->print_confusion_matrix($unique, $matrix);
# (A)0 1 # Example Output From Printing a Pretty Confusion Matrix.
# 0| 3 2
# 1| 1 4
```

```
A/P 0 1
 0| 3 2
 1| 1 4
```

Out[22]:
```
0
```

A confusion matrix is always a good idea to use in addition to classification accuracy to help interpret the predictions.

4.2.3 Mean Absolute Error

Regression problems are those where a real value is predicted. An easy metric to consider is the error in the predicted values as compared to the expected values. The Mean Absolute Error or MAE for short is a good first error metric to use. It is calculated as the average of the absolute error values, where absolute means made positive so that they can be added together.

$$MAE = \sum_{i=1}^{n} abs(predicted_i - actual_i) \; / \; totalpredictions$$

(4.2)

Below is a function named mae metric() that implements this metric. As above, it expects a list of actual outcome values and a list of predictions. We use the built-in abs() Python function to calculate the absolute error values that are summed together.

In [23]:
```
# Function To Calculate Mean Absolute Error.
# Calculate mean absolute error
my $mae_metric = sub{
  my ($self, $actual, $predicted) = @_;
  my $sum_error = 0.0;
  for my $pair (zip $actual, $predicted){
    $sum_error += abs($pair->[0] - $pair->[1]);
  }
  return sprintf '%0.3f', $sum_error / @$actual;
};

sml->add_to_class('sml', 'mae_metric', $mae_metric);
```

Out[23]:
```
*sml::mae_metric
```

```
Subroutine sml::mae_metric redefined at /usr/local/lib/perl5/site_perl/5.32.1/
x86_64-linux/sml.pm line 22.
```

We can contrive a small regression dataset to test this function.

In [24]:
```
# Small Set of Contrived Regression Predictions and Actual Values.
$actual    = [0.1, 0.2, 0.3, 0.4, 0.5];
$predicted = [0.11, 0.19, 0.29, 0.41, 0.5];
print "actual\tpredicted\n";
print $_->[0], "\t", $_->[1], "\n" for zip ($actual, $predicted);
```

```
actual  predicted
0.1     0.11
0.2     0.19
0.3     0.29
0.4     0.41
0.5     0.5
```

Only one prediction (0.5) is correct, whereas all other predictions are wrong by 0.01. Therefore, we would expect the mean absolute error (or the average positive error) for these predictions to be a little less than 0.01. Below is an example that tests the mae metric() function with the contrived dataset.

Running this example prints the output below. We can see that as expected, the MAE was 0.008, a small value slightly lower than 0.01.

In [77]:
```
# Test MAE
my $mae = sml->mae_metric($actual, $predicted);
print $mae;
# Example Output From Calculating the Mean Absolute Error.
# 0.008
```

```
0.008
1
```
Out[77]:

4.2.4 Root Mean Squared Error

Another popular way to calculate the error in a set of regression predictions is to use the Root Mean Squared Error. Shortened as RMSE, the metric is sometimes called Mean Squared Error or MSE, dropping the Root part from the calculation and the name. RMSE is calculated as the square root of the mean of the squared differences between actual outcomes and predictions. Squaring each error forces the values to be positive, and the square root of the mean squared error returns the error metric back to the original units for comparison.

$$RMSE = \sqrt(\sum_{i=1}^{n}(predicted_i - actual_i)^2 \; / \; totalpredictions)$$

(4.3)

Below is an implementation of this in a function named rmse metric(). It uses the sqrt() function from the math module and uses the ** operator to raise the error to the 2nd power.

In [25]:
```
# Defined in Section 4.2.4 Root Mean Squared Error
# Function To Calculate Root Mean Squared Error.
# Calculate root mean squared error
my $rmse_metric = sub{
  my ($self, $actual, $predicted) = @_;
  my $sum_error = 0.0;
  for my $pair (zip $actual, $predicted){
    $sum_error += (($pair->[0] - $pair->[1]) ** 2);
  }
  my $mean_error = $sum_error / @$actual;
  return sprintf '%0.4f', sqrt($mean_error);
};

sml->add_to_class('sml', 'rmse_metric', $rmse_metric);
```

Out[25]:
```
*sml::rmse_metric
```

```
Subroutine sml::rmse_metric redefined at /usr/local/lib/perl5/site_perl/5.32.
1/x86_64-linux/sml.pm line 22.
```

We can test this metric on the same dataset used to test the calculation of Mean Absolute Error above. Below is a complete example. Again, we would expect an error value to be generally close to 0.01.

In [26]:
```
# Test RMSE
$actual    = [0.1, 0.2, 0.3, 0.4, 0.5];
$predicted = [0.11, 0.19, 0.29, 0.41, 0.5];
my $rmse = sml->rmse_metric($actual, $predicted);
print $rmse;
# Example Output From Calculating the Root Mean Squared Error.
# 0.0089
```

```
0.0089
```

Out[26]:
```
1
```

4.2.5 ROC curves

Computing AUC ROC from scratch in Perl without using any libraries.

In [27]:
```
# Function to calculate the ROC metrics
my $perf_metrics = sub{
  my ($self, $actual, $y_hat, $threshold) = @_;

  my ($tp, $fp, $tn, $fn, $tpr, $fpr) = (0, 0, 0, 0);
  for my $i (0 .. $#$y_hat) {
    if ($y_hat->[$i] >= $threshold){
      if ($actual->[$i] == 1) {
        $tp++;
      }else {
        $fp++;
      }
    }else{
      if ($actual->[$i] == 0){
        $tn++;
      }else{
        $fn++;
      }
    }
  }

  $tpr = $tp / ($tp + $fn); # True Positive Rate
  $fpr = $fp / ($fp + $tn); # False Positive Rate

  return $fpr, $tpr;
};

sml->add_to_class('sml', 'perf_metrics', $perf_metrics);
```

Out[27]:
```
*sml::perf_metrics
```

In [28]:
```
# Function to calculate the integral using the trapezoid rule
my $trapz = sub {
  my ($self, $x, $y) = @_;
  my $sum = 0;
  for my $i (0 .. @$x - 2){
    $sum += ($x->[$i + 1] - $x->[$i]) * ($y->[$i] + $y->[$i + 1]) / 2;
  }
  return $sum;
};

sml->add_to_class('sml', 'trapz', $trapz);
```

Out[28]:
```
*sml::trapz
```

In [29]:
```
my ($dataset, $header) = sml->load_csv('../data/model.csv');

my ($class, $predicted_prob) = (zip map {[$_->[1], $_->[2]]} @$dataset);

printf "class: %s\n", dump @$class[0 .. 4];
printf "predicted_prob: %s\n", dump @$predicted_prob[0 .. 4];
```

```
class: (0, 1, 0, 1, 0)
predicted_prob: (0.592837, 0.624829, 0.073848, 0.544891, 0.015118)
```

Out[29]:
```
1
```

In [30]:
```
# Calculate TPR and FPR for a specific threshold
my ($fpr, $tpr) = sml->perf_metrics($class, $predicted_prob, 0.5);

# Print sensitivity and specificity
printf "Sensibilidad: %.2f, Especificidad: %.2f\n", $tpr, 1 - $fpr;
```

```
Sensibilidad: 1.00, Especificidad: 0.99
```

Out[30]:
```
1
```

In [31]:
```
# Calculate TPR and FPR for various decision thresholds
my @thresholds = map {$_ / 20 } (0 .. 20);
```

Out[31]:
```
00.050.10.150.20.250.30.350.40.450.50.550.60.650.70.750.80.850.90.951
```

In [32]:
```
my ($fprs, $tprs) = (zip map {[sml->perf_metrics($class, $predicted_prob, $_)]

print dump $tprs, $fprs;
```

```
(
  [
    1,
    1,
    1,
    1,
    1,
    1,
    1,
    1,
    1,
    1,
    1,
    0.891891891891892,
    0.77027027027027,
    0.608108108108108,
    0.378378378378378,
    0.121621621621622,
    0.0135135135135135,
    0,
    0,
    0,
    0,
  ],
  [
    1,
    0.855263157894737,
    0.842105263157895,
    0.657894736842105,
    0.657894736842105,
    0.25,
    0.25,
    0.0131578947368421,
    0.0131578947368421,
    0.0131578947368421,
    0.0131578947368421,
    0.0131578947368421,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
  ],
)
```

Out[32]:
```
1
```

In [33]:
```
# Calculate the area under the ROC curve (AUC)
# First, sort the points by ascending FPR
my @sorted_indices = sort { $fprs->[$a] <=> $fprs->[$b] } 0 .. $#$fprs;
my @sorted_fprs = @$fprs[@sorted_indices];
my @sorted_tprs = @$tprs[@sorted_indices];
```

Out[33]:
```
0.770270270270270.6081081081081080.3783783783783780.1216216216216220.013513513
5135135000011110.89189189189189211111111
```

In [34]:
```
# Then, calculate the AUC using the trapezoid rule
my $auc = sml->trapz(\@sorted_fprs, \@sorted_tprs);
printf "Area under the ROC curve (AUC): %0.3f\n", $auc;
```

```
Area under the ROC curve (AUC): 0.981
```
Out[34]:
```
1
```

In [38]:
```
# Plot the ROC curve using Chart::Plotly
my $trace1 = new Chart::Plotly::Trace::Scatter(
    x => $fprs,
    y => $tprs,
    mode => 'lines',
    name => 'ROC Curve'
);

my $trace2 = new Chart::Plotly::Trace::Scatter(
    x => [0, 1],
    y => [0, 1],
    mode => 'lines',
    name => 'ROC Curve'
);

my $chart = new Chart::Plotly::Plot(
    traces => [$trace1, $trace2],
    layout => {
        title => 'ROC curve',
        xaxis => { title => 'False Positive Rate (FPR)' },
        yaxis => { title => 'True Positive Rate (TPR)' }
    }
);

# Show the graph directly in IPerl
IPerl->display($chart);
```

4.3 Extensions

You have only seen a small sample of the most widely used performance metrics. There are many other performance metrics that you may require. Below is a list of 5 additional performance metrics that you may wish to implement to extend this tutorial

- Precision for classification.
- Recall for classification.
- F1 for classification.
- Area Under ROC Curve or AUC for classification.
- Goodness of Fit or R 2 (R squared) for regression.

4.4 Review

In this tutorial, you discovered how to implement algorithm prediction performance metrics from scratch in Perl. Specifically, you learned:

- How to implement and interpret classification accuracy.
- How to implement and interpret the confusion matrix for classification problems.
- How to implement and interpret mean absolute error for regression.
- How to implement and interpret root mean squared error for regression.