REGRESSION

Asmaliza Ahzan
IVERSON ASSOCIATES SDN BHD

Table of Contents

Predict prices using regression with ML.NET

https://docs.microsoft.com/en-us/dotnet/machine-learning/tutorials/predict-prices

This exercise illustrates how to build a regression model using ML.NET to predict prices, specifically, New York City taxi fares.

In this exercis, you learn how to:

- Prepare and understand the data
- Load and transform the data
- Choose a learning algorithm
- Train the model
- Evaluate the model
- Use the model for predictions

Prerequisites

- Visual Studio 2019 or later or Visual Studio 2017 version 15.6 or later with the ".NET Core cross-platform development" workload installed.

Create a console application

1. Create a .NET Core Console Application called "TaxiFarePrediction".
2. Create a directory named Data in your project to store the data set and model files.
3. Install the Microsoft.ML and Microsoft.ML.FastTree NuGet Package:
    a. In Solution Explorer, right-click the project and select Manage NuGet Packages. Choose "nuget.org" as the Package source, select the Browse tab, search for Microsoft.ML, select the package in the list, and select the Install button. Select the OK button on the Preview Changes dialog and then select the I Accept button on the License Acceptance dialog if you agree with the license terms for the packages listed. Do the same for the Microsoft.ML.FastTree NuGet package.

Prepare and understand the data

1. Download the taxi-fare-train.csv and the taxi-fare-test.csv data sets and save them to the Data folder you've created at the previous step. We use these data sets to train the machine learning model and then evaluate how accurate the model is. These data sets are originally from the NYC TLC Taxi Trip data set.
2. In Solution Explorer, right-click each of the *.csv files and select Properties. Change the value of Copy to Output Directory to Copy if newer.
3. Open the taxi-fare-train.csv data set and look at column headers in the first row. Take a look at each of the columns. Understand the data and decide which columns are features and which one is the label.

The label is the column you want to predict. The identified Featuresare the inputs you give the model to predict the Label.

The provided data set contains the following columns:

- vendor_id: The ID of the taxi vendor is a feature.
- rate_code: The rate type of the taxi trip is a feature.
- passenger_count: The number of passengers on the trip is a feature.
- trip_time_in_secs: The amount of time the trip took. You want to predict the fare of the trip before the trip is completed. At that moment, you don't know how long the trip would take. Thus, the trip time is not a feature and you'll exclude this column from the model.
- trip_distance: The distance of the trip is a feature.
- payment_type: The payment method (cash or credit card) is a feature.
- fare_amount: The total taxi fare paid is the label.

Create data classes

Create classes for the input data and the predictions:

1. In Solution Explorer, right-click the project, and then select Add > New Item.
2. In the Add New Item dialog box, select Class and change the Name field to TaxiTrip.cs. Then, select the Add button.
3. Add the following using directives to the new file:

```
using Microsoft.ML.Data;
```

Remove the existing class definition and add the following code, which has two classes TaxiTrip and TaxiTripFarePrediction, to the TaxiTrip.cs file:

```
public class TaxiTrip
 {
     [LoadColumn(0)]
     public string VendorId;

     [LoadColumn(1)]
     public string RateCode;

     [LoadColumn(2)]
     public float PassengerCount;

     [LoadColumn(3)]
     public float TripTime;

     [LoadColumn(4)]
     public float TripDistance;

     [LoadColumn(5)]
     public string PaymentType;

     [LoadColumn(6)]
     public float FareAmount;
 }

 public class TaxiTripFarePrediction
 {
     [ColumnName("Score")]
     public float FareAmount;
    }
```

TaxiTrip is the input data class and has definitions for each of the data set columns. Use the LoadColumnAttribute attribute to specify the indices of the source columns in the data set.

The TaxiTripFarePrediction class represents predicted results. It has a single float field, FareAmount, with a Score ColumnNameAttribute attribute applied. In case of the regression task, the Score column contains predicted label values.

Define data and model paths

Add the following additional using statements to the top of the Program.cs file:

```
using System;
using System.IO;
using Microsoft.ML;
```

You need to create three fields to hold the paths to the files with data sets and the file to save the model:

- _trainDataPath contains the path to the file with the data set used to train the model.
- _testDataPath contains the path to the file with the data set used to evaluate the model.
- _modelPath contains the path to the file where the trained model is stored.

Add the following code right above the Main method to specify those paths and for the _textLoader variable:

```
static readonly string _trainDataPath = Path.Combine(Environment.CurrentDirectory,
"Data", "taxi-fare-train.csv");
static readonly string _testDataPath = Path.Combine(Environment.CurrentDirectory,
"Data", "taxi-fare-test.csv");
static readonly string _modelPath = Path.Combine(Environment.CurrentDirectory, "Data",
"Model.zip");
```

All ML.NET operations start in the MLContext class. Initializing mlContext creates a new ML.NET environment that can be shared across the model creation workflow objects. It's similar, conceptually, to DBContext in Entity Framework.

Initialize variables in Main

Replace the Console.WriteLine("Hello World!") line in the Main method with the following code to declare and initialize the mlContext variable:

```
MLContext mlContext = new MLContext(seed: 0);
```

Add the following as the next line of code in the Main method to call the Train method:

```
var model = Train(mlContext, _trainDataPath);
```

The Train() method executes the following tasks:

- Loads the data.
- Extracts and transforms the data.
- Trains the model.
- Returns the model.

The Train method trains the model. Create that method just below Main, using the following code:

```
public static ITransformer Train(MLContext mlContext, string dataPath)
{

}
```

Load and transform data

ML.NET uses the IDataView interface as a flexible, efficient way of describing numeric or text tabular data. IDataView can load either text files or in real time (for example, SQL database or log files). Add the following code as the first line of the Train() method:

```
IDataView dataView = mlContext.Data.LoadFromTextFile<TaxiTrip>(dataPath, hasHeader:
true, separatorChar: ',');
```

As you want to predict the taxi trip fare, the FareAmount column is the Label that you will predict (the output of the model). Use the CopyColumnsEstimator transformation class to copy FareAmount, and add the following code:

```
var pipeline = mlContext.Transforms.CopyColumns(outputColumnName: "Label",
inputColumnName: "FareAmount")
```

The algorithm that trains the model requires numeric features, so you have to transform the categorical data (VendorId, RateCode, and PaymentType) values into numbers (VendorIdEncoded, RateCodeEncoded, and PaymentTypeEncoded).

To do that, use the OneHotEncodingTransformer transformation class, which assigns different numeric key values to the different values in each of the columns, and add the following code:

```
.Append(mlContext.Transforms.Categorical.OneHotEncoding(outputColumnName:
"VendorIdEncoded", inputColumnName: "VendorId"))

.Append(mlContext.Transforms.Categorical.OneHotEncoding(outputColumnName:
"RateCodeEncoded", inputColumnName: "RateCode"))

.Append(mlContext.Transforms.Categorical.OneHotEncoding(outputColumnName:
"PaymentTypeEncoded", inputColumnName: "PaymentType"))
```

The last step in data preparation combines all of the feature columns into the Features column using the mlContext.Transforms.Concatenate transformation class. By default, a learning algorithm processes only features from the Features column. Add the following code:

```
.Append(mlContext.Transforms.Concatenate("Features", "VendorIdEncoded",
"RateCodeEncoded", "PassengerCount", "TripDistance", "PaymentTypeEncoded"))
```

Choose a learning algorithm

This problem is about predicting a taxi trip fare in New York City. At first glance, it may seem to depend simply on the distance traveled. However, taxi vendors in New York charge varying amounts for other factors such as additional passengers or paying with a credit card instead of cash. You want to predict the price value, which is a real value, based on the other factors in the dataset. To do that, you choose a regression machine learning task.

Append the FastTreeRegressionTrainer machine learning task to the data transformation definitions by adding the following as the next line of code in Train():

```
.Append(mlContext.Regression.Trainers.FastTree());
```

Train the model

Fit the model to the training dataview and return the trained model by adding the following line of code in the Train() method:

```
var model = pipeline.Fit(dataView);
```

The Fit() method trains your model by transforming the dataset and applying the training.

Return the trained model with the following line of code in the Train() method:

```
return model;
```

Evaluate the model

Next, evaluate your model performance with your test data for quality assurance and validation. Create the Evaluate() method, just after Train(), with the following code:

```
private static void Evaluate(MLContext mlContext, ITransformer model)
{

}
```

The Evaluate method executes the following tasks:

- Loads the test dataset.
- Creates the regression evaluator.
- Evaluates the model and creates metrics.
- Displays the metrics.

Add a call to the new method from the Main method, right under the Train method call, using the following code:

```
Evaluate(mlContext, model);
```

Load the test dataset using the LoadFromTextFile() method. Evaluate the model using this dataset as a quality check by adding the following code in the Evaluate method:

```
IDataView dataView = mlContext.Data.LoadFromTextFile<TaxiTrip>(_testDataPath,
hasHeader: true, separatorChar: ',');
```

Next, transform the Test data by adding the following code to Evaluate():

```
var predictions = model.Transform(dataView);
```

The Transform() method makes predictions for the test dataset input rows.

The RegressionContext.Evaluate method computes the quality metrics for the PredictionModel using the specified dataset. It returns a RegressionMetrics object that contains the overall metrics computed by regression evaluators.

To display these to determine the quality of the model, you need to get the metrics first. Add the following code as the next line in the Evaluate method:

```
var metrics = mlContext.Regression.Evaluate(predictions, "Label", "Score");
```

Once you have the prediction set, the Evaluate() method assesses the model, which compares the predicted values with the actual Labels in the test dataset and returns metrics on how the model is performing.

Add the following code to evaluate the model and produce the evaluation metrics:

```
Console.WriteLine();
Console.WriteLine($"*************************************************");
Console.WriteLine($"*       Model quality metrics evaluation         ");
Console.WriteLine($"*------------------------------------------------");
```

RSquared is another evaluation metric of the regression models. RSquared takes values between 0 and 1. The closer its value is to 1, the better the model is. Add the following code into the Evaluate method to display the RSquared value:

```
Console.WriteLine($"*       RSquared Score:      {metrics.RSquared:0.##}");
```

RMS is one of the evaluation metrics of the regression model. The lower it is, the better the model is. Add the following code into the Evaluate method to display the RMS value:

```
Console.WriteLine($"*       Root Mean Squared Error:
{metrics.RootMeanSquaredError:#.##}");
```

Use the model for predictions

Create the TestSinglePrediction method, just after the Evaluate method, using the following code:

```
private static void TestSinglePrediction(MLContext mlContext, ITransformer model)
{

}
```

The TestSinglePrediction method executes the following tasks:

- Creates a single comment of test data.
- Predicts fare amount based on test data.
- Combines test data and predictions for reporting.
- Displays the predicted results.

Add a call to the new method from the Main method, right under the Evaluate method call, using the following code:

```
TestSinglePrediction(mlContext, model);
```

Use the PredictionEngine to predict the fare by adding the following code to TestSinglePrediction():

```
var predictionFunction = mlContext.Model.CreatePredictionEngine<TaxiTrip,
TaxiTripFarePrediction>(model);
```

The PredictionEngine is a convenience API, which allows you to perform a prediction on a single instance of data. PredictionEngine is not thread-safe. It's acceptable to use in single-threaded or prototype environments. For improved performance and thread safety in production environments, use the PredictionEnginePool service, which creates an ObjectPool of PredictionEngine objects for use throughout your application. See this guide on how to use PredictionEnginePool in an ASP.NET Core Web API.

This exercise uses one test trip within this class. Later you can add other scenarios to experiment with the model. Add a trip to test the trained model's prediction of cost in the TestSinglePrediction() method by creating an instance of TaxiTrip:

```
var taxiTripSample = new TaxiTrip()
            {
                VendorId = "VTS",
                RateCode = "1",
                PassengerCount = 1,
```

```
            TripTime = 1140,
            TripDistance = 3.75f,
            PaymentType = "CRD",
            FareAmount = 0 // To predict. Actual/Observed = 15.5
        };
```

Next, predict the fare based on a single instance of the taxi trip data and pass it to the PredictionEngine by adding the following as the next lines of code in the TestSinglePrediction() method:

```
var prediction = predictionFunction.Predict(taxiTripSample);
```

The Predict() function makes a prediction on a single instance of data.

To display the predicted fare of the specified trip, add the following code into the TestSinglePrediction method:

```
Console.WriteLine($"*****************************************************************
****");
Console.WriteLine($"Predicted fare: {prediction.FareAmount:0.####}, actual fare:
15.5");
Console.WriteLine($"*****************************************************************
****");
```

Run the program to see the predicted taxi fare for your test case.

Congratulations! You've now successfully built a machine learning model for predicting taxi trip fares, evaluated its accuracy, and used it to make predictions. You can find the source code for this exercise at the dotnet/samples GitHub repository.