Microsoft

Introducing Machine Learning

Professional

Dino Esposito
Francesco Esposito

Microsoft

Introducing Machine Learning

Dino Esposito
Francesco Esposito

Hosting Scenarios

At the end of the training phase, you have a model that contains instructions on which algorithm to run in production and using which configuration. The model file is a zipped file in some serialization format.

In production, the client application will be making calls to a façade API offered by the ML.NET framework, as follows:

```
var mlContext = new MLContext();
var model = mlContext.Model.Load("model.zip", out var schema);
var predictionEngine = mlContext.Model.CreatePredictionEngine<ModelInput, ModelOutput>(model);
var result = predictionEngine.Predict(input);
```

As you can guess, in production the model is only a black box that accepts a *ModelInput* class and returns a *ModelOutput* class. Needless to say, those classes are defined during the training phase and depend on the dataset and the problem.

Note Each machine learning library has its own serialization format and uses its own schema to save information to be used in production. A universal, interoperable format also exists: the ONNX format. ML.NET supports it. It should be noted, though, that ONNX is a sort of common denominator across machine learning frameworks and may not support certain features of certain frameworks.

Summary

This chapter offered a condensed, but hopefully juicy, summary of the pillars of the ML.NET library slated to become the reference platform for machine learning in the .NET space. Version 1.0 was released in the spring of 2019, and the library is growing at a rapid pace. For example, it already supports deep learning training via TensorFlow models for image classification.

Although Python is popular among data scientists, there's no strict reason why machine learning models can't be developed and tested in .NET or other languages, including Java and Go. It's all about the ecosystem and ease of use. ML.NET relies on the .NET Core infrastructure and Visual Studio 2019. The main points leading to ML.NET are as follows:

- Many enterprise .NET and .NET Core applications want to directly deploy and consume machine learning models natively instead of having to install some additional Python environment in production. In addition, in many enterprise production environments, Python might be hard to get approved and, even if it is, it would require adding additional HTTP services. ML.NET allows precisely that: training natively in .NET and deploying natively into the .NET application.
- Many .NET developers don't want to have to learn Python to be able to create custom machine learning models and infuse them into their .NET applications.

Finally, there is NimbusML, namely ML.NET bindings for Python. NimbusML is a Python module—fully interoperable with scikit-learn, NumPy, and Pandas—that allows data scientists or developers familiar with Python to code and then have the model saved as a .zip file fully compatible with the ML.NET library and then natively hostable in .NET applications.

Let's now go with a simple but not-so-trivial and complete example: taxi fare prediction. In the next chapter, we'll see a bit of feature engineering, feature selection, and more importantly, an ASP.NET client application.

CHAPTER 7

Implementing the ML.NET Pipeline

You are not thinking; you are just being logical.

—Niels Bohr, Nobel Prize in Physics in 1922 and father of quantum mechanics

Since the early days of software, humans have dreamed of it as a tool capable of looking into the future and reporting bits and pieces of what is to come. Price prediction is a canonical example, whether it is in regard to the stock exchange, real estate, the supply chain, energy, or individual services such as taxi rides.

We looked deep into statistics to try to make sense of the future fluctuations of prices but found out that statistics is great only at post-mortem analysis of gathered data. Statistics excels at dissecting data to extract models; it is great to understand with 100 percent accuracy what led to those numbers and why. However, there's not much that statistics can tell about the future with sufficient reliability and credibility.

Looking into the future is the job of machine learning. Machine learning builds on statistics but elaborates on a model that starts from gathered data to try to guess what will happen. The elaborated model is never 100 percent accurate with regard to gathered data but is expected to provide some higher reliability (ideally over 90 percent) with any data in line with those it was trained on.

Let's look at a full example of price prediction in ML.NET.

The Data to Start From

The example we consider here is based on a sample dataset available from the Github website of the ML.NET library. It refers to over one million taxi rides logged in New York City. The ultimate purpose of the example is predicting the price of a taxi ride in New York.

Note that it is reasonable to expect that the prediction made by a model trained on that dataset would work also for any city whose dynamics of taxi rides and prices is comparable to the one that emerges from the file, regardless of geography.

Exploring the Dataset

The sample file is a CSV file made of seven columns: the ID of the taxi company, the code of the rate, the number of passengers, the time it took to complete the ride, the distance, the type of payment (cash or card), and fare paid. Before embarking on a deeper analysis of the actual information contained by the CSV file, let's figure out how to use it.

The rows of the dataset should be turned into a C# class to make it easier to build and consume the final model. The following class represents one row of the raw 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;
}
```

The *LoadColumn* attribute establishes a static binding between the specific property and corresponding column—indicated by name or position—in the dataset. This class needs to be placed in a separate assembly because it must be also referenced by any .NET client applications entitled to use the model.

Applying Common Data Transformations

Any machine learning algorithm requires numbers to work nicely. In the sample dataset, instead, a few columns are made of text—the vendor ID, rate code, and payment type. The values in those columns must be turned into numbers in some way that doesn't alter the distribution and relevance of individual values:

```
var vendor = mlContext.Transforms.Categorical.OneHotEncoding(V_Id, "VendorId");
var rate = mlContext.Transforms.Categorical.OneHotEncoding(Rate_Code, "RateCode");
var payment = mlContext.Transforms.Categorical.OneHotEncoding(Payment_Type, "PaymentType");
mlContext.Append(vendor)
         .Append(rate)
         .Append(payment);
```

The *OneHotEncoding* object applies a common data transformation algorithm to categorical values. The algorithm consists of adding one binary (0/1) column for each distinct categorical value found in the specified column. The first parameter of the method is the prefix to name new columns.

Another transformation that might make sense to apply is the normalization of mean variance on numeric columns:

```
mlContext.Append(mlContext.Transforms.NormalizeMeanVariance("PassengerCount"));
```

In addition, you might want to remove outliers, namely values that are too far away from the mean. This step may not be necessary all the time, but if you have reason to believe that outliers affect results, by all means you should do so. You remove outliers by simply filtering the loaded dataset. For the example, remove from the dataset all rows that have a value of the *FareAmount* column lower than 1 and higher than 150:

```
mlContext.Data.FilterRowsByColumn(rawData, "FareAmount", 1, 150);
```

Finally, a couple of further transformations are required because of the internal mechanics of the ML.NET library. You need to have a column named *Label* that represents the target of the prediction and a column named *Features* that contains all values of the row serialized in an array:

```
mlContext.Transforms.CopyColumns("Label", "FareAmount");
mlContext.Transforms.Concatenate("Features", ...);
```

In this way, you tell the training algorithm to target the values of the original FareAmount column (now duplicated in the Label column) and to process the input values in the Features column made by the concatenation of all other values in the row.

Considerations on the Dataset

Any machine learning model is essentially a transformer that works on whatever you pass to produce whatever it can figure out. If the input data is inadequate or insufficient or in some way unbalanced, you will get inadequate or insufficient or unbalanced answers.

It is therefore particularly important that the training dataset contains information about all possible factors that could influence the prediction. Sometimes, if two or more individual values assume a particular relevance when combined, you might want to add an ad hoc column. This is ultimately the realm of feature engineering.

Is there anything about the sample dataset that you can argue is wrong?

For one thing, it completely misses the traffic factor. The traffic factor can be expressed as a normalized value in the 0–1 range to indicate the level or even a categorical value. It depends on the desired accuracy and also on the data feed available. If no feed is available, you can even think of adding some traffic context information to calculate the categorical value by looking at the time of the day the ride took place. And here's another one! The time of the day is not in the sample dataset.

Even in such a simple scenario, we have found a couple of arguable points to dissect and to test on. Just imagine how many and how deep they could be in a much more complex and sophisticated prediction scenario!

The Training Step

When it comes to predicting a numeric value like the price of a service, the class of algorithms that works most of the time is regression. (We'll tackle the internals of the most common classes of machine learning algorithms in Part III, "Fundamentals of Shallow Learning.") A number of different specific algorithms fall under the umbrella of regression, and choosing the one to try first is a matter of experience, knowledge of the domain, and sometimes even a gut feeling.

Whatever algorithm you choose for the first run of training needs to survive the metrics of the post-training test. If numbers don't support the choice, you might consider trying a different algorithm or shape the training set differently. Machine learning is almost always a matter of trial and error.

Picking an Algorithm

Conceptually, price prediction is a (relatively) easy regression problem to solve. If you have good and detailed data, prediction should be essentially a matter of choosing the fastest algorithm. In ML.NET, the trainers available for the regression task are grouped under the Regression property of the context. Here's how to add a regression trainer to the pipeline:

```
// Identify the training algorithm
var trainer = mlContext
       .Regression
       .Trainers
       .OnlineGradientDescent("Label", "Features", new SquaredLoss());

// Add it to the current ML pipeline
mlContext.Append(trainer);

// Start training of the model
var trainedModel = mlContext.Fit(dataView);
```

The selected algorithm—the online gradient descent algorithm—is generally a good choice, but faster and more precise algorithms exist, such as the *LightGbmRegression* algorithm. You can use any of those more sophisticated algorithms by referencing additional Nuget packages. With the default configuration of ML.NET, the online gradient descent is commonly a good option.

The algorithm takes two string parameters to denote the names of the input and output columns (or features) in the dataset. The output column is the column to predict. The third parameter indicates the error function that will be used during the testing phase to measure the distance between the predicted value and the expected value. The *SquaredLoss* object refers to the R-squared metric—a fairly