BINARY CLASSIFICATION

Asmaliza Ahzan
IVERSON ASSOCIATES SDN BHD

Table of Contents

Analyze sentiment of website comments with binary classification in ML.NET

https://docs.microsoft.com/en-us/dotnet/machine-learning/tutorials/sentiment-analysis

This exercise shows you how to create a .NET Core console application that classifies sentiment from website comments and takes the appropriate action. The binary sentiment classifier uses C# in Visual Studio 2019.

In this exercise, you learn how to:

- Create a console application
- Prepare data
- Load the data
- Build and train the model
- Evaluate the model
- Use the model to make a prediction
- See the results

Prerequisites

- Visual Studio 2019 or later with the ".NET Core cross-platform development" workload installed
- UCI Sentiment Labeled Sentences dataset (ZIP file)

Create a console application

1. Create a .NET Core Console Application called "SentimentAnalysis".
2. Create a directory named Data in your project to save your data set files.
3. Install the Microsoft.ML NuGet Package.
    a. In Solution Explorer, right-click on your project and select Manage NuGet Packages. Choose "nuget.org" as the package source, and then select the Browse tab. Search for Microsoft.ML, select the package you want, and then select the Install button. Proceed with the installation by agreeing to the license terms for the package you choose.

Prepare your data

1. Download UCI Sentiment Labeled Sentences dataset ZIP file, and unzip.
2. Copy the **yelp_labelled.txt** file into the Data directory you created.
3. In Solution Explorer, right-click the **yelp_labeled.txt** file and select Properties. Change the value of Copy to Output Directory to Copy if newer.

Create classes and define paths

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

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Microsoft.ML;
using Microsoft.ML.Data;
using static Microsoft.ML.DataOperationsCatalog;
using Microsoft.ML.Trainers;
using Microsoft.ML.Transforms.Text;
```

2. Add the following code to the line right above the Main method, to create a field to hold the recently downloaded dataset file path:

```
static readonly string _dataPath = Path.Combine(Environment.CurrentDirectory, "Data",
"yelp_labelled.txt");
```

3. Next, create classes for your input data and predictions. Add a new class to your project:
    a. In Solution Explorer, right-click the project, and then select Add > New Item.
    b. In the Add New Item dialog box, select Class and change the Name field to SentimentData.cs. Then, select the Add button.

4. The SentimentData.cs file opens in the code editor. Add the following using statement to the top of SentimentData.cs:

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

5. Remove the existing class definition and add the following code, which has two classes SentimentData and SentimentPrediction, to the SentimentData.cs file:

```
public class SentimentData
{
    [LoadColumn(0)]
    public string SentimentText;

    [LoadColumn(1), ColumnName("Label")]
    public bool Sentiment;
}

public class SentimentPrediction : SentimentData
{

    [ColumnName("PredictedLabel")]
```

```
    public bool Prediction { get; set; }

    public float Probability { get; set; }

    public float Score { get; set; }
}
```

How the data was prepared

- The input dataset class, SentimentData, has a string for user comments (SentimentText) and a bool (Sentiment) value of either 1 (positive) or 0 (negative) for sentiment.
- Both fields have LoadColumn attributes attached to them, which describes the data file order of each field.
- In addition, the Sentiment property has a ColumnName attribute to designate it as the Label field.
- The following example file doesn't have a header row, and looks like this:

SentimentText	Sentiment (Label)
Waitress was a little slow in service.	0
Crust is not good.	0
Wow... Loved this place.	1
Service was very prompt.	1

- SentimentPrediction is the prediction class used after model training. It inherits from SentimentData so that the input SentimentText can be displayed along with the output prediction. The Prediction boolean is the value that the model predicts when supplied with new input SentimentText.
- The output class SentimentPrediction contains two other properties calculated by the model: Score - the raw score calculated by the model, and Probability - the score calibrated to the likelihood of the text having positive sentiment.
- For this exercise, the most important property is Prediction.

Load the data

Data in ML.NET is represented as an IDataView interface. IDataView is a flexible, efficient way of describing tabular data (numeric and text). Data can be loaded from a text file or in real time (for example, SQL database or log files) to an IDataView object.

The MLContext class is a starting point for all ML.NET operations. 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.

You prepare the app, and then load data:

1. 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();
```

2. Add the following as the next line of code in the Main() method:

```
TrainTestData splitDataView = LoadData(mlContext);
```

3. Create the LoadData() method, just after the Main() method, using the following code:

```
public static TrainTestData LoadData(MLContext mlContext)
{

}
```

The LoadData() method executes the following tasks:

- Loads the data.
- Splits the loaded dataset into train and test datasets.
- Returns the split train and test datasets.

4. Add the following code as the first line of the LoadData() method:

```
IDataView dataView = mlContext.Data.LoadFromTextFile<SentimentData>(_dataPath,
hasHeader: false);
```

The LoadFromTextFile() method defines the data schema and reads in the file. It takes in the data path variables and returns an IDataView.

Split the dataset for model training and testing

When preparing a model, you use part of the dataset to train it and part of the dataset to test the model's accuracy.

1. To split the loaded data into the needed datasets, add the following code as the next line in the LoadData() method:

```
TrainTestData splitDataView = mlContext.Data.TrainTestSplit(dataView,
testFraction: 0.2);
```

The previous code uses the TrainTestSplit() method to split the loaded dataset into train and test datasets and return them in the DataOperationsCatalog.TrainTestData class. Specify the test set percentage of data with the testFractionparameter. The default is 10%, in this case you use 20% to evaluate more data.

2. Return the splitDataView at the end of the LoadData() method:

```
return splitDataView;
```

Build and train the model

1. Add the following call to the BuildAndTrainModelmethod as the next line of code in the Main() method:

```
ITransformer model = BuildAndTrainModel(mlContext, splitDataView.TrainSet);
```

The BuildAndTrainModel() method executes the following tasks:

- Extracts and transforms the data.
- Trains the model.
- Predicts sentiment based on test data.
- Returns the model.

2. Create the BuildAndTrainModel() method, just after the Main() method, using the following code:

```
public static ITransformer BuildAndTrainModel(MLContext mlContext, IDataView
splitTrainSet)
{

}
```

Extract and transform the data

1. Call FeaturizeText as the next line of code:

```
var estimator = mlContext.Transforms.Text.FeaturizeText(outputColumnName:
"Features", inputColumnName: nameof(SentimentData.SentimentText));
```

The FeaturizeText() method in the previous code converts the text column (SentimentText) into a numeric key type Features column used by the machine learning algorithm and adds it as a new dataset column:

SentimentText	Sentiment	Features
Waitress was a little slow in service.	0	[0.76, 0.65, 0.44, ...]
Crust is not good.	0	[0.98, 0.43, 0.54, ...]
Wow... Loved this place.	1	[0.35, 0.73, 0.46, ...]
Service was very prompt.	1	[0.39, 0, 0.75, ...]

Add a learning algorithm

This app uses a classification algorithm that categorizes items or rows of data. The app categorizes website comments as either positive or negative, so use the binary classification task.

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

```
.Append(mlContext.BinaryClassification.Trainers.SdcaLogisticRegression(labelColumnName
: "Label", featureColumnName: "Features"))
```

The SdcaLogisticRegressionBinaryTrainer is your classification training algorithm. This is appended to the estimator and accepts the featurized SentimentText (Features) and the Label input parameters to learn from the historic data.

Train the model

Fit the model to the splitTrainSet data and return the trained model by adding the following as the next line of code in the BuildAndTrainModel() method:

```
Console.WriteLine("=============== Create and Train the Model ===============");
var model = estimator.Fit(splitTrainSet);
Console.WriteLine("=============== End of training ===============");
Console.WriteLine();
```

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

Return the model trained to use for evaluation

Return the model at the end of the BuildAndTrainModel() method:

```
return model;
```

Evaluate the model

After your model is trained, use your test data to validate the model's performance.

1. Create the Evaluate() method, just after BuildAndTrainModel(), with the following code:

```
public static void Evaluate(MLContext mlContext, ITransformer model, IDataView
splitTestSet)
{

}
```

The Evaluate() method executes the following tasks:

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

2. Add a call to the new method from the Main() method, right under the BuildAndTrainModel() method call, using the following code:

```
Evaluate(mlContext, model, splitDataView.TestSet);
```

3. Transform the splitTestSet data by adding the following code to Evaluate():

```
Console.WriteLine("=============== Evaluating Model accuracy with Test
data===============");
IDataView predictions = model.Transform(splitTestSet);
```

The previous code uses the Transform() method to make predictions for multiple provided input rows of a test dataset.

4. Evaluate the model by adding the following as the next line of code in the Evaluate() method:

```
CalibratedBinaryClassificationMetrics metrics =
mlContext.BinaryClassification.Evaluate(predictions, "Label");
```

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

Displaying the metrics for model validation

Use the following code to display the metrics:

```
Console.WriteLine();
Console.WriteLine("Model quality metrics evaluation");
Console.WriteLine("--------------------------------");
Console.WriteLine($"Accuracy: {metrics.Accuracy:P2}");
Console.WriteLine($"Auc: {metrics.AreaUnderRocCurve:P2}");
Console.WriteLine($"F1Score: {metrics.F1Score:P2}");
Console.WriteLine("=============== End of model evaluation ===============");
```

- The Accuracy metric gets the accuracy of a model, which is the proportion of correct predictions in the test set.
- The AreaUnderRocCurve metric indicates how confident the model is correctly classifying the positive and negative classes. You want the AreaUnderRocCurve to be as close to one as possible.
- The F1Score metric gets the model's F1 score, which is a measure of balance between precision and recall. You want the F1Score to be as close to one as possible.

Predict the test data outcome

1. Create the UseModelWithSingleItem() method, just after the Evaluate() method, using the following code:

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

}
```

The UseModelWithSingleItem() method executes the following tasks:

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

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

```
UseModelWithSingleItem(mlContext, model);
```

3. Add the following code to create as the first line in the UseModelWithSingleItem() Method:

```
PredictionEngine<SentimentData, SentimentPrediction> predictionFunction =
mlContext.Model.CreatePredictionEngine<SentimentData, SentimentPrediction>(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.

4. Add a comment to test the trained model's prediction in the UseModelWithSingleItem() method by creating an instance of SentimentData:

```
SentimentData sampleStatement = new SentimentData
{
    SentimentText = "This was a very bad steak"
};
```

5. Pass the test comment data to the PredictionEngine by adding the following as the next lines of code in the UseModelWithSingleItem() method:

```
var resultPrediction = predictionFunction.Predict(sampleStatement);
```

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

6. Display SentimentText and corresponding sentiment prediction using the following code:

```
Console.WriteLine();
Console.WriteLine("=============== Prediction Test of model with a single sample and
test dataset ===============");

Console.WriteLine();
Console.WriteLine($"Sentiment: {resultPrediction.SentimentText} | Prediction:
{(Convert.ToBoolean(resultPrediction.Prediction) ? "Positive" : "Negative")} |
Probability: {resultPrediction.Probability} ");

Console.WriteLine("=============== End of Predictions ===============");
Console.WriteLine();
```

Use the model for prediction

Deploy and predict batch items

1. Create the UseModelWithBatchItems() method, just after the UseModelWithSingleItem() method, using the following code:

```
public static void UseModelWithBatchItems(MLContext mlContext, ITransformer model)
 {

 }
```

The UseModelWithBatchItems() method executes the following tasks:

- Creates batch test data.
- Predicts sentiment based on test data.
- Combines test data and predictions for reporting.
- Displays the predicted results.

2. Add a call to the new method from the Main method, right under the UseModelWithSingleItem() method call, using the following code:

```
UseModelWithBatchItems(mlContext, model);
```

3. Add some comments to test the trained model's predictions in the UseModelWithBatchItems() method:

```
IEnumerable<SentimentData> sentiments = new[]
 {
             new SentimentData
             {
                 SentimentText = "This was a horrible meal"
             },
             new SentimentData
             {
                 SentimentText = "I love this spaghetti."
             }
 }
```

Predict comment sentiment

Use the model to predict the comment data sentiment using the Transform() method:

```
IDataView batchComments = mlContext.Data.LoadFromEnumerable(sentiments);

IDataView predictions = model.Transform(batchComments);

// Use model to predict whether comment data is Positive (1) or Negative (0).
IEnumerable<SentimentPrediction> predictedResults =
mlContext.Data.CreateEnumerable<SentimentPrediction>(predictions, reuseRowObject:
false);
```

Combine and display the predictions

Create a header for the predictions using the following code:

```
Console.WriteLine();
Console.WriteLine("=============== Prediction Test of loaded model with multiple
samples ===============");
```

Because SentimentPrediction is inherited from SentimentData, the Transform() method populated SentimentText with the predicted fields. As the ML.NET process processes, each component adds columns, and this makes it easy to display the results:

```
foreach (SentimentPrediction prediction in predictedResults)
{
       Console.WriteLine($"Sentiment: {prediction.SentimentText} | Prediction:
{(Convert.ToBoolean(prediction.Prediction) ? "Positive" : "Negative")} | Probability:
{prediction.Probability} ");
}
Console.WriteLine("=============== End of predictions ===============");
```

Results

Your results should be similar to the following. During processing, messages are displayed. You may see warnings, or processing messages. These have been removed from the following results for clarity.

```
Model quality metrics evaluation
--------------------------------
Accuracy: 83.96%
Auc: 90.51%
F1Score: 84.04%

=============== End of model evaluation ===============

=============== Prediction Test of model with a single sample and test
dataset ===============

Sentiment: This was a very bad steak | Prediction: Negative | Probability:
0.1027377
=============== End of Predictions ===============

=============== Prediction Test of loaded model with a multiple samples
===============

Sentiment: This was a horrible meal | Prediction: Negative | Probability:
0.1369192
Sentiment: I love this spaghetti. | Prediction: Positive | Probability:
0.9960636
=============== End of predictions ===============

=============== End of process ===============
Press any key to continue . . .
```

Congratulations! You've now successfully built a machine learning model for classifying and predicting messages sentiment.

Building successful models is an iterative process. This model has initial lower quality as the exercise uses small datasets to provide quick model training. If you aren't satisfied with the model quality, you can try to improve it by providing larger training datasets or by choosing different training algorithms with different hyper-parameters for each algorithm.

You can find the source code for this exercise at the dotnet/samples repository.