

Contents

ML.NET

Overview

Model Builder & CLI

The ML.NET Model Builder tool

The ML.NET Command-Line interface

API

The ML.NET API

Tutorials

Model Builder & CLI

Predict prices (Model Builder & regression)

Analyze sentiment using the ML.NET CLI

Analyze sentiment in Razor Pages (Model Builder & binary classification)

Categorize health violations (Model Builder & multiclass classification)

Categorize land use from satellite images (Model Builder & image classification)

Detect traffic signs (Model Builder & object detection)

Train a recommendation model using Model Builder

API

Overview

Analyze sentiment (binary classification)

Categorize support issues (multiclass classification)

Predict prices (regression)

Categorize iris flowers (k-means clustering)

Recommend movies (matrix factorization)

Image Classification (transfer learning)

Classify images (model composition)

Forecast bike rental demand (time series)

Call-volume spikes (anomaly detection)

Product Sales Analysis (anomaly detection)

Detect objects in images (object detection)

[Classify sentiment using TensorFlow \(text classification\)](#)

[Infer.NET](#)

[Probabilistic programming with Infer.NET](#)

[Concepts](#)

[ML.NET tasks](#)

[Algorithms](#)

[Data transforms](#)

[Model evaluation metrics](#)

[Improve model accuracy](#)

[How-to guides](#)

[Model Builder & CLI](#)

[Load data into Model Builder](#)

[Label images for object detection](#)

[Install Model Builder](#)

[Install GPU in Model Builder](#)

[Install the CLI](#)

[Use the automated ML API](#)

[API](#)

[Install extra dependencies](#)

[Load data](#)

[Prepare data](#)

[Train, evaluate, and explain the model](#)

[Train and evaluate a model](#)

[Train a model using cross-validation](#)

[Inspect intermediate pipeline data values](#)

[Determine model feature importance with PFI](#)

[Use the trained model](#)

[Save and load a model](#)

[Use a model to make predictions](#)

[Re-train a model](#)

[Deploy a model to Azure Functions](#)

[Deploy a model to a web API](#)

[Consume Azure Machine Learning ONNX model](#)

[Make batch predictions using .NET for Apache Spark](#)

Reference

[ML.NET API Reference](#)

[ML.NET API Preview API Reference](#)

[CLI reference](#)

Resources

[Overview](#)

[Glossary](#)

[Azure training resources](#)

[CLI telemetry](#)

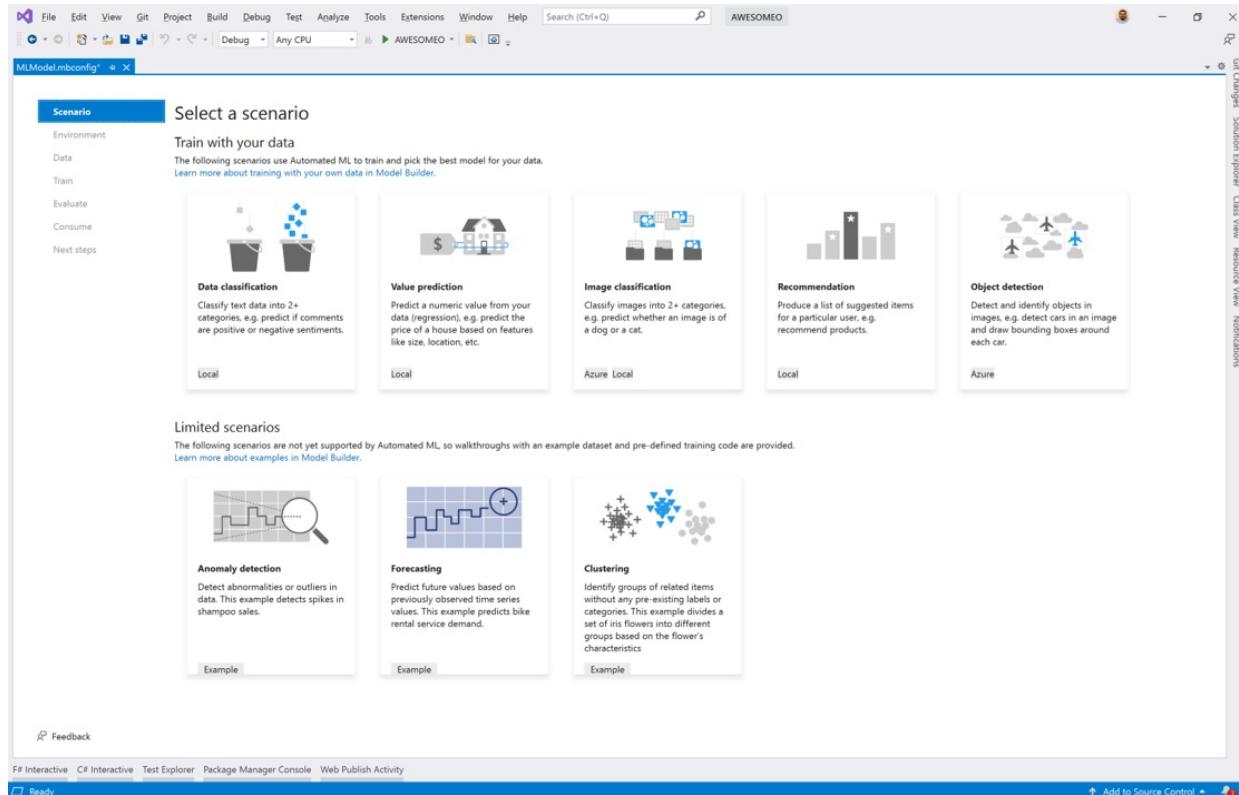
What is Model Builder and how does it work?

10/14/2022 • 9 minutes to read • [Edit Online](#)

ML.NET Model Builder is an intuitive graphical Visual Studio extension to build, train, and deploy custom machine learning models.

Model Builder uses automated machine learning (AutoML) to explore different machine learning algorithms and settings to help you find the one that best suits your scenario.

You don't need machine learning expertise to use Model Builder. All you need is some data, and a problem to solve. Model Builder generates the code to add the model to your .NET application.



NOTE

Model Builder is currently in Preview.

Creating a Model Builder Project

When you first start up Model Builder it will ask for you to name the project. This will create an `.mbconfig` configuration file inside of the project.

The `.mbconfig` file keeps track of everything you do in Model Builder to allow you to reopen the session.

After training, three files are generated under the `*.mbconfig` file:

- **Model.consumption.cs:** This file contains the `ModelInput` and `ModelOutput` schemas as well as the `Predict` function generated for consuming the model.
- **Model.training.cs:** This file contains the training pipeline (data transforms, algorithm, algorithm hyperparameters) chosen by Model Builder to train the model. You can use this pipeline for re-training your

model.

- **Model.zip:** This is a serialized zip file which represents your trained ML.NET model.

When you create your `mbconfig` file, you're prompted for a name. This name is applied to the consumption, training, and model files. In this case, the name used is *Model*.

Scenario

You can bring many different scenarios to Model Builder, to generate a machine learning model for your application.

A scenario is a description of the type of prediction you want to make using your data. For example:

- predict future product sales volume based on historical sales data
- classify sentiments as positive or negative based on customer reviews
- detect whether a banking transaction is fraudulent
- route customer feedback issues to the correct team in your company

Each scenario maps to a different Machine Learning Task which include:

- Binary classification
- Multiclass classification
- Regression
- Clustering
- Anomaly detection
- Ranking
- Recommendation
- Forecasting

For example, the scenario of classifying sentiments as positive or negative would fall under the binary classification task.

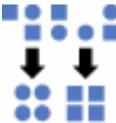
For more information about the different ML Tasks supported by ML.NET see [Machine learning tasks in ML.NET](#).

Which machine learning scenario is right for me?

In Model Builder, you need to select a scenario. The type of scenario depends on what type of prediction you are trying to make.

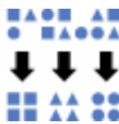
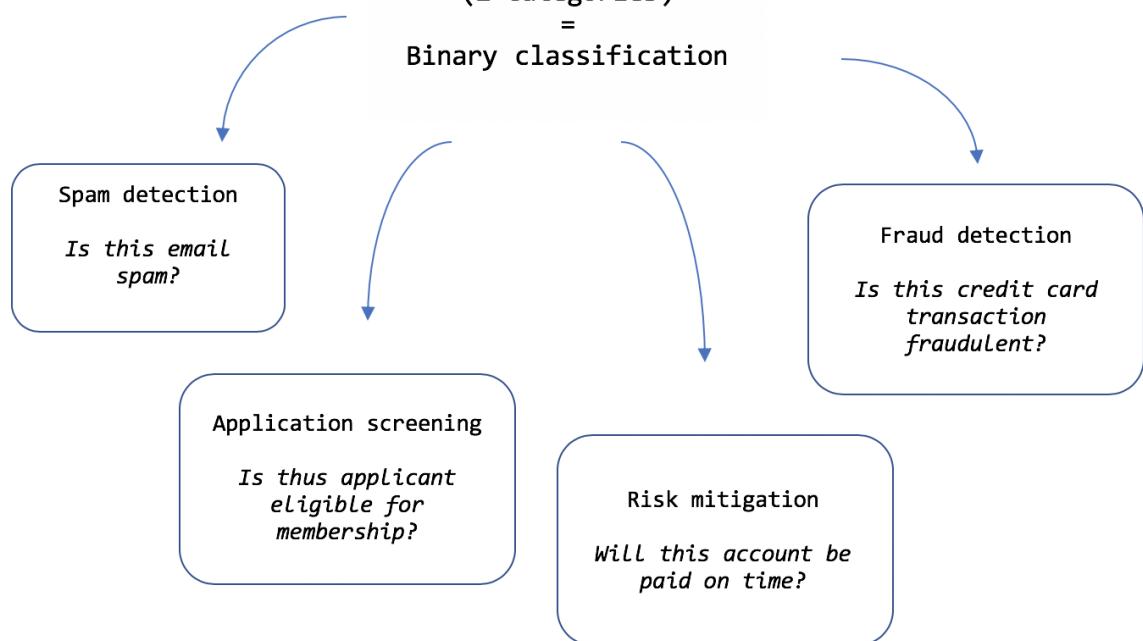
Data classification

Classification is used to categorize data into categories.



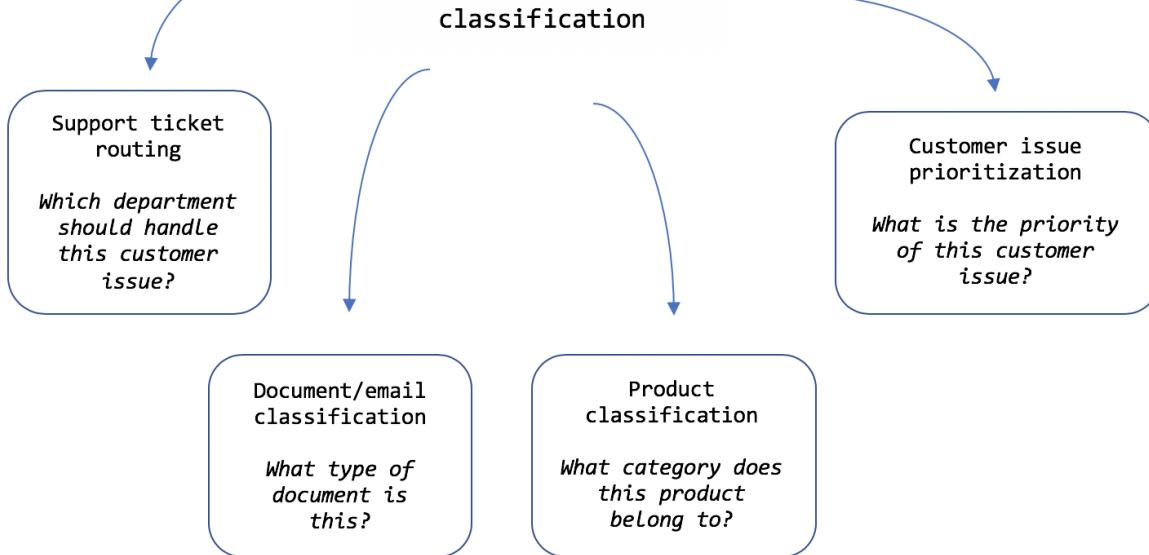
Predict a category
(2 categories)

=
Binary classification



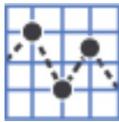
Predict a category
(3+ categories)

=
Multiclass classification



Value prediction

Value prediction, which falls under the regression task, is used to predict numbers.



Predict a number
=
Regression

Price prediction

How much will this house sell for?

Sales forecasting

What will the annual sales for this region be?

Predictive maintenance

After how much time will this part need maintenance?

Machine monitoring

What is the moisture content of this dryer?

Image classification

Image classification is used to identify images of different categories. For example, different types of terrain or animals or manufacturing defects.

You can use the image classification scenario if you have a set of images, and you want to classify the images into different categories.

Object detection

Object detection is used to locate and categorize entities within images. For example, locating and identifying cars and people in an image.

You can use object detection when images contain multiple objects of different types.

Recommendation

The recommendation scenario predicts a list of suggested items for a particular user, based on how similar their likes and dislikes are to other users'.

You can use the recommendation scenario when you have a set of users and a set of "products", such as items to purchase, movies, books, or TV shows, along with a set of users' "ratings" of those products.

Environment

You can train your machine learning model locally on your machine or in the cloud on Azure, depending on the scenario.

When you train locally, you work within the constraints of your computer resources (CPU, memory, and disk).

When you train in the cloud, you can scale up your resources to meet the demands of your scenario, especially for large datasets.

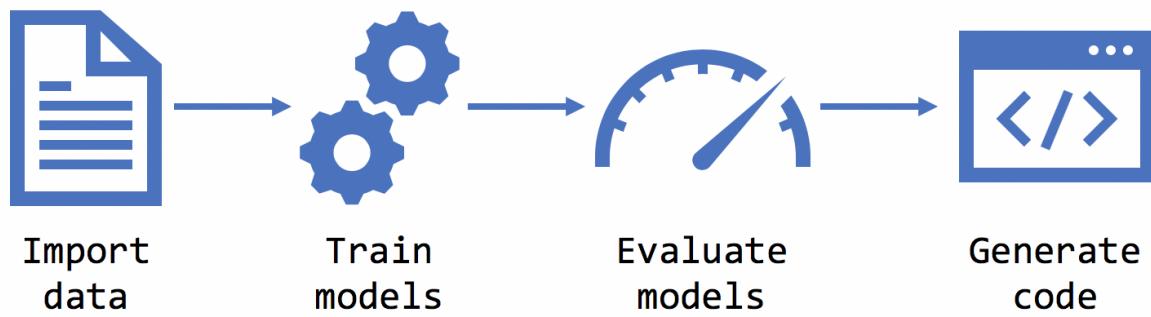
Local CPU training is supported for all scenarios except Object Detection.

Local GPU training is supported for Image Classification.

Azure training is supported for Image Classification and Object Detection.

Data

Once you have chosen your scenario, Model Builder asks you to provide a dataset. The data is used to train, evaluate, and choose the best model for your scenario.



Model Builder supports datasets in .tsv, .csv, .txt formats, as well as SQL database format. If you have a .txt file, columns should be separated with `,`, `,` or `\t`.

If the dataset is made up of images, the supported file types are `.jpg` and `.png`.

For more information, see [Load training data into Model Builder](#).

Choose the output to predict (label)

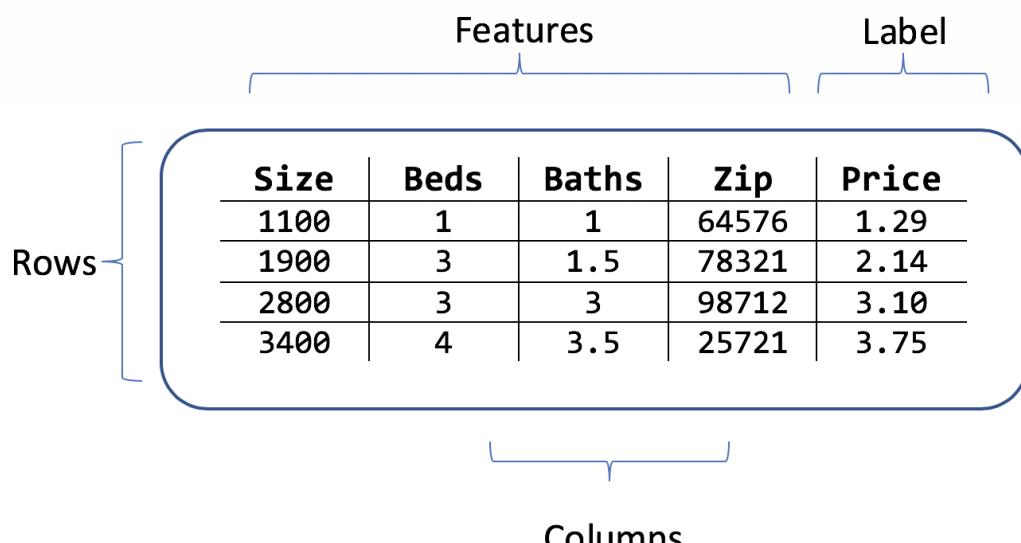
A dataset is a table of rows of training examples, and columns of attributes. Each row has:

- a **label** (the attribute that you want to predict)
- **features** (attributes that are used as inputs to predict the label).

For the house-price prediction scenario, the features could be:

- the square footage of the house
- the number of bedrooms and bathrooms
- the zip code

The label is the historical house price for that row of square footage, bedroom, and bathroom values and zip code.



Example datasets

If you don't have your own data yet, try out one of these datasets:

SCENARIO	EXAMPLE	DATA	LABEL	FEATURES
Classification	Predict sales anomalies	product sales data	Product Sales	Month
	Predict sentiment of website comments	website comment data	Label (0 when negative sentiment, 1 when positive)	Comment, Year
	Predict fraudulent credit card transactions	credit card data	Class (1 when fraudulent, 0 otherwise)	Amount, V1-V28 (anonymized features)
	Predict the type of issue in a GitHub repository	GitHub issue data	Area	Title, Description
Value prediction	Predict taxi fare price	taxi fare data	Fare	Trip time, distance
Image classification	Predict the category of a flower	flower images	The type of flower: daisy, dandelion, roses, sunflowers, tulips	The image data itself
Recommendation	Predict movies that someone will like	movie ratings	Users, Movies	Ratings

Train

Once you select your scenario, environment, data, and label, Model Builder trains the model.

What is training?

Training is an automatic process by which Model Builder teaches your model how to answer questions for your scenario. Once trained, your model can make predictions with input data that it has not seen before. For example, if you are predicting house prices and a new house comes on the market, you can predict its sale price.

Because Model Builder uses automated machine learning (AutoML), it does not require any input or tuning from you during training.

How long should I train for?

Model Builder uses AutoML to explore multiple models to find you the best performing model.

Longer training periods allow AutoML to explore more models with a wider range of settings.

The table below summarizes the average time taken to get good performance for a suite of example datasets, on a local machine.

DATASET SIZE	AVERAGE TIME TO TRAIN
0 - 10 MB	10 sec
10 - 100 MB	10 min

DATASET SIZE	AVERAGE TIME TO TRAIN
100 - 500 MB	30 min
500 - 1 GB	60 min
1 GB+	3+ hours

These numbers are a guide only. The exact length of training is dependent on:

- the number of features (columns) being used as input to the model
- the type of columns
- the ML task
- the CPU, disk, and memory performance of the machine used for training

It's generally advised that you use more than 100 rows as datasets with less than that may not produce any results.

Evaluate

Evaluation is the process of measuring how good your model is. Model Builder uses the trained model to make predictions with new test data, and then measures how good the predictions are.

Model Builder splits the training data into a training set and a test set. The training data (80%) is used to train your model and the test data (20%) is held back to evaluate your model.

How do I understand my model performance?

A scenario maps to a machine learning task. Each ML task has its own set of evaluation metrics.

Value prediction

The default metric for value prediction problems is RSquared, the value of RSquared ranges between 0 and 1. 1 is the best possible value or in other words the closer the value of RSquared to 1 the better your model is performing.

Other metrics reported such as absolute-loss, squared-loss, and RMS loss are additional metrics, which can be used to understand how your model is performing and comparing it against other value prediction models.

Classification (2 categories)

The default metric for classification problems is accuracy. Accuracy defines the proportion of correct predictions your model is making over the test dataset. The closer to 100% or 1.0 the better it is.

Other metrics reported such as AUC (Area under the curve), which measures the true positive rate vs. the false positive rate should be greater than 0.50 for models to be acceptable.

Additional metrics like F1 score can be used to control the balance between Precision and Recall.

Classification (3+ categories)

The default metric for Multi-class classification is Micro Accuracy. The closer the Micro Accuracy to 100% or 1.0 the better it is.

Another important metric for Multi-class classification is Macro-accuracy, similar to Micro-accuracy the closer to 1.0 the better it is. A good way to think about these two types of accuracy is:

- Micro-accuracy: How often does an incoming ticket get classified to the right team?
- Macro-accuracy: For an average team, how often is an incoming ticket correct for their team?

More information on evaluation metrics

For more information, see [model evaluation metrics](#).

Improve

If your model performance score is not as good as you want it to be, you can:

- Train for a longer period of time. With more time, the automated machine learning engine experiments with more algorithms and settings.
- Add more data. Sometimes the amount of data is not sufficient to train a high-quality machine learning model. This is especially true with datasets that have a small number of examples.
- Balance your data. For classification tasks, make sure that the training set is balanced across the categories. For example, if you have four classes for 100 training examples, and the two first classes (tag1 and tag2) are used for 90 records, but the other two (tag3 and tag4) are only used on the remaining 10 records, the lack of balanced data may cause your model to struggle to correctly predict tag3 or tag4.

Consume

After the evaluation phase, Model Builder outputs a model file, and code that you can use to add the model to your application. ML.NET models are saved as a zip file. The code to load and use your model is added as a new project in your solution. Model Builder also adds a sample console app that you can run to see your model in action.

In addition, Model Builder gives you the option to create projects that consume your model. Currently, Model Builder will create the following projects:

- **Console app:** Creates a .NET Core console applications to make predictions from your model.
- **Web API:** Creates an ASP.NET Core Web API that lets you consume your model over the internet.

What's next?

[Install](#) the Model Builder Visual Studio extension

Try [price prediction or any regression scenario](#)

Automate model training with the ML.NET CLI

10/14/2022 • 3 minutes to read • [Edit Online](#)

The ML.NET CLI automates model generation for .NET developers.

To use the ML.NET API by itself, (without the ML.NET AutoML CLI) you need to choose a trainer (implementation of a machine learning algorithm for a particular task), and the set of data transformations (feature engineering) to apply to your data. The optimal pipeline will vary for each dataset and selecting the optimal algorithm from all the choices adds to the complexity. Even further, each algorithm has a set of hyperparameters to be tuned. Hence, you can spend weeks and sometimes months on machine learning model optimization trying to find the best combinations of feature engineering, learning algorithms, and hyperparameters.

The ML.NET CLI simplifies this process using automated machine learning (AutoML).

NOTE

This topic refers to ML.NET CLI and ML.NET AutoML, which are currently in Preview, and material may be subject to change.

What is the ML.NET command-line interface (CLI)?

The ML.NET CLI is a [.NET Core tool](#). Once installed, you give it a machine learning task and a training dataset, and it generates an ML.NET model, as well as the C# code to run to use the model in your application.

As shown in the following figure, it is simple to generate a high quality ML.NET model (serialized model .zip file) plus the sample C# code to run/score that model. In addition, the C# code to create/train that model is also generated, so that you can research and iterate on the algorithm and settings used for that generated "best model".



You can generate those assets from your own datasets without coding by yourself, so it also improves your productivity even if you already know ML.NET.

Currently, the ML Tasks supported by the ML.NET CLI are:

- classification

- regression
- recommendation
- image classification

Example of usage (classification scenario):

```
mlnet classification --dataset "yelp_labelled.txt" --label-col 1 --has-header false --train-time 10
```

A screenshot of a Windows PowerShell window titled 'Windows PowerShell'. The command 'mlnet classification --dataset "yelp_labelled.txt" --label-col 1 --has-header false --train-time 10' is typed into the command line. The window has standard operating system window controls (minimize, maximize, close) at the top right.

You can run it the same way on *Windows PowerShell*, *macOS/Linux bash*, or *Windows CMD*. However, tabular auto-completion (parameter suggestions) won't work on *Windows CMD*.

Output assets generated

The ML task commands in the CLI generate the following assets in the output folder:

- A serialized model .zip ("best model") ready to use for running predictions.
- C# solution with:
 - C# code to run/score that generated model (to make predictions in your end-user apps with that model).
 - C# code with the training code used to generate that model (for learning purposes or model retraining).
- Log file with information of all iterations/sweeps across the multiple algorithms evaluated, including their detailed configuration/pipeline.

The first two assets can directly be used in your end-user apps (ASP.NET Core web app, services, desktop app, etc.) to make predictions with that generated ML model.

The third asset, the training code, shows you what ML.NET API code was used by the CLI to train the generated model, so you can retrain your model and investigate and iterate on which specific trainer/algorithm and hyperparameters were selected by the CLI and AutoML under the covers.

Understanding the quality of the model

When you generate a 'best model' with the CLI tool, you see quality metrics (such as accuracy and R-Squared) as appropriate for the ML task you're targeting.

Here those metrics are summarized grouped by ML task so you can understand the quality of your auto-generated 'best model'.

Metrics for Classification models

The following image displays the classification metrics list for the top five models found by the CLI:

	Trainer	MicroAccuracy	MacroAccuracy	Duration	#Iteration
1	LightGbmMulti	0.7888	0.7249	31.6	3
2	SdcaMaximumEntropyMulti	0.7702	0.6999	32.9	2
3	AveragedPerceptronOva	0.7640	0.5862	5.4	1
4	FastTreeOva	0.7640	0.6686	482.6	5
5	SymbolicSgdLogisticRegressionOva	0.6398	0.5265	2.8	4

Accuracy is a popular metric for classification problems, however accuracy isn't always the best metric to select the best model from as explained in the following references. There are cases where you need to evaluate the quality of your model with additional metrics.

To explore and understand the metrics that are output by the CLI, see [Evaluation metrics for classification](#).

Metrics for Regression and Recommendation models

A regression model fits the data well if the differences between the observed values and the model's predicted values are small and unbiased. Regression can be evaluated with certain metrics.

You'll see a similar list of metrics for the top five quality models found by the CLI, except in this case, the top five are related to a regression ML task:

	Trainer	RSquared	Absolute-loss	Squared-loss	RMS-loss	Duration	#Iteration
1	FastTreeRegression	0.9644	0.39	3.38	1.84	7.2	139
2	FastTreeRegression	0.9638	0.41	3.43	1.85	4.9	109
3	FastTreeRegression	0.9631	0.41	3.50	1.87	5.5	112
4	LightGbmRegression	0.9631	0.39	3.51	1.87	1.6	129
5	FastTreeRegression	0.9623	0.39	3.58	1.89	6.8	136

To explore and understand the metrics that are output by the CLI, see [Evaluation metrics for regression](#).

See also

- [How to install the ML.NET CLI tool](#)
- [Tutorial: Analyze sentiment using the ML.NET CLI](#)
- [ML.NET CLI command reference](#)
- [Telemetry in ML.NET CLI](#)

What is ML.NET and how does it work?

10/14/2022 • 9 minutes to read • [Edit Online](#)

ML.NET gives you the ability to add machine learning to .NET applications, in either online or offline scenarios. With this capability, you can make automatic predictions using the data available to your application. Machine learning applications make use of patterns in the data to make predictions rather than needing to be explicitly programmed.

Central to ML.NET is a machine learning **model**. The model specifies the steps needed to transform your input data into a prediction. With ML.NET, you can train a custom model by specifying an algorithm, or you can import pre-trained TensorFlow and ONNX models.

Once you have a model, you can add it to your application to make the predictions.

ML.NET runs on Windows, Linux, and macOS using .NET Core, or Windows using .NET Framework. 64 bit is supported on all platforms. 32 bit is supported on Windows, except for TensorFlow, LightGBM, and ONNX-related functionality.

Examples of the type of predictions that you can make with ML.NET:

Classification/Categorization

Automatically divide customer feedback into positive and negative categories

Regression/Predict continuous values

Predict the price of houses based on size and location

Anomaly Detection

Detect fraudulent banking transactions

Recommendations

Suggest products that online shoppers may want to buy, based on their previous purchases

Time series/sequential data

Forecast the weather/product sales

Image classification

Categorize pathologies in medical images

Hello ML.NET World

The code in the following snippet demonstrates the simplest ML.NET application. This example constructs a linear regression model to predict house prices using house size and price data.

```

using System;
using Microsoft.ML;
using Microsoft.ML.Data;

class Program
{
    public class HouseData
    {
        public float Size { get; set; }
        public float Price { get; set; }
    }

    public class Prediction
    {
        [ColumnName("Score")]
        public float Price { get; set; }
    }

    static void Main(string[] args)
    {
        MLContext mlContext = new MLContext();

        // 1. Import or create training data
        HouseData[] houseData = {
            new HouseData() { Size = 1.1F, Price = 1.2F },
            new HouseData() { Size = 1.9F, Price = 2.3F },
            new HouseData() { Size = 2.8F, Price = 3.0F },
            new HouseData() { Size = 3.4F, Price = 3.7F } };
        IDataView trainingData = mlContext.Data.LoadFromEnumerable(houseData);

        // 2. Specify data preparation and model training pipeline
        var pipeline = mlContext.Transforms.Concatenate("Features", new[] { "Size" })
            .Append(mlContext.Regression.Trainers.Sdca(labelColumnName: "Price",
maximumNumberOfIterations: 100));

        // 3. Train model
        var model = pipeline.Fit(trainingData);

        // 4. Make a prediction
        var size = new HouseData() { Size = 2.5F };
        var price = mlContext.Model.CreatePredictionEngine<HouseData, Prediction>(model).Predict(size);

        Console.WriteLine($"Predicted price for size: {size.Size*1000} sq ft= {price.Price*100:C}k");

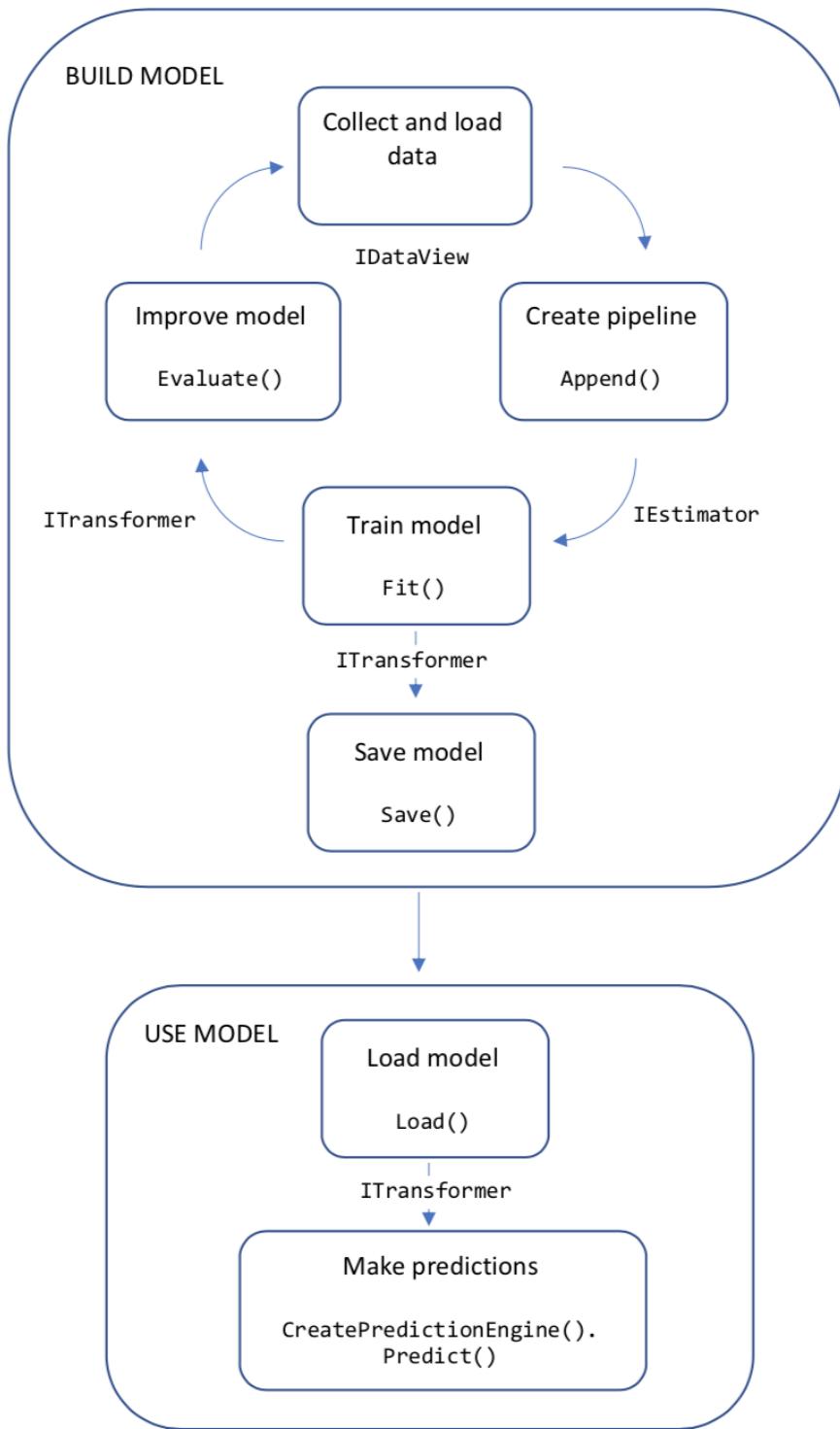
        // Predicted price for size: 2500 sq ft= $261.98k
    }
}

```

Code workflow

The following diagram represents the application code structure, as well as the iterative process of model development:

- Collect and load training data into an **IDataView** object
- Specify a pipeline of operations to extract features and apply a machine learning algorithm
- Train a model by calling **Fit()** on the pipeline
- Evaluate the model and iterate to improve
- Save the model into binary format, for use in an application
- Load the model back into an **ITransformer** object
- Make predictions by calling **CreatePredictionEngine.Predict()**



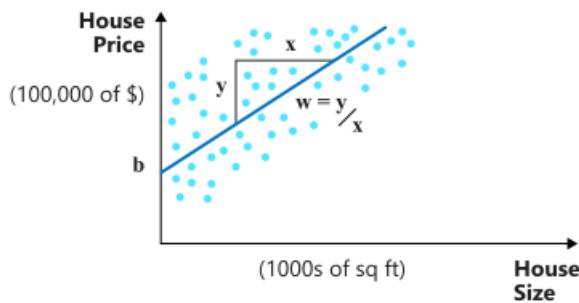
Let's dig a little deeper into those concepts.

Machine learning model

An ML.NET model is an object that contains transformations to perform on your input data to arrive at the predicted output.

Basic

The most basic model is two-dimensional linear regression, where one continuous quantity is proportional to another, as in the house price example above.

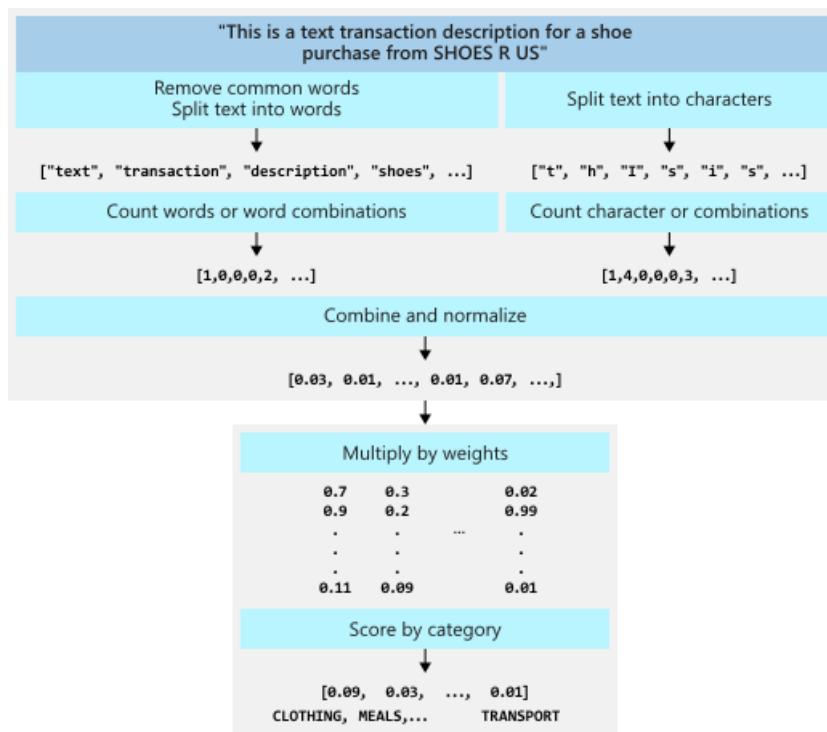


The model is simply: $\text{Price} = b + \text{Size} * w$. The parameters b and w are estimated by fitting a line on a set of (size, price) pairs. The data used to find the parameters of the model is called **training data**. The inputs of a machine learning model are called **features**. In this example, Size is the only feature. The ground-truth values used to train a machine learning model are called **labels**. Here, the Price values in the training data set are the labels.

More complex

A more complex model classifies financial transactions into categories using the transaction text description.

Each transaction description is broken down into a set of features by removing redundant words and characters, and counting word and character combinations. The feature set is used to train a linear model based on the set of categories in the training data. The more similar a new description is to the ones in the training set, the more likely it will be assigned to the same category.



Both the house price model and the text classification model are **linear** models. Depending on the nature of your data and the problem you are solving, you can also use **decision tree** models, **generalized additive** models, and others. You can find out more about the models in [Tasks](#).

Data preparation

In most cases, the data that you have available isn't suitable to be used directly to train a machine learning model. The raw data needs to be prepared, or pre-processed, before it can be used to find the parameters of your model. Your data may need to be converted from string values to a numerical representation. You might have redundant information in your input data. You may need to reduce or expand the dimensions of your input data. Your data might need to be normalized or scaled.

The [ML.NET tutorials](#) teach you about different data processing pipelines for text, image, numerical, and time-series data used for specific machine learning tasks.

[How to prepare your data](#) shows you how to apply data preparation more generally.

You can find an appendix of all of the [available transformations](#) in the resources section.

Model evaluation

Once you have trained your model, how do you know how well it will make future predictions? With ML.NET, you can evaluate your model against some new test data.

Each type of machine learning task has metrics used to evaluate the accuracy and precision of the model against the test data set.

For our house price example, we used the **Regression** task. To evaluate the model, add the following code to the original sample.

```
HouseData[] testHouseData =
{
    new HouseData() { Size = 1.1F, Price = 0.98F },
    new HouseData() { Size = 1.9F, Price = 2.1F },
    new HouseData() { Size = 2.8F, Price = 2.9F },
    new HouseData() { Size = 3.4F, Price = 3.6F }
};

var testHouseDataView = mlContext.Data.LoadFromEnumerable(testHouseData);
var testPriceDataView = model.Transform(testHouseDataView);

var metrics = mlContext.Regression.Evaluate(testPriceDataView, labelColumnName: "Price");

Console.WriteLine($"R^2: {metrics.RSquared:0.##}");
Console.WriteLine($"RMS error: {metrics.RootMeanSquaredError:0.##}");

// R^2: 0.96
// RMS error: 0.19
```

The evaluation metrics tell you that the error is low-ish, and that correlation between the predicted output and the test output is high. That was easy! In real examples, it takes more tuning to achieve good model metrics.

ML.NET architecture

In this section, we go through the architectural patterns of ML.NET. If you are an experienced .NET developer, some of these patterns will be familiar to you, and some will be less familiar. Hold tight, while we dive in!

An ML.NET application starts with an [MLContext](#) object. This singleton object contains **catalogs**. A catalog is a factory for data loading and saving, transforms, trainers, and model operation components. Each catalog object has methods to create the different types of components:

Data loading and saving

[DataOperationsCatalog](#)

Data preparation

[TransformsCatalog](#)

Training algorithms

Binary classification

[BinaryClassificationCatalog](#)

Multiclass classification

[MulticlassClassificationCatalog](#)

Anomaly detection

[AnomalyDetectionCatalog](#)

Clustering

[ClusteringCatalog](#)

Forecasting

[ForecastingCatalog](#)

Ranking

[RankingCatalog](#)

Regression

[RegressionCatalog](#)

Recommendation

[RecommendationCatalog](#)

Add the `Microsoft.ML.Recommender` NuGet package

TimeSeries

[TimeSeriesCatalog](#)

Add the `Microsoft.ML.TimeSeries` NuGet package

Model usage

[ModelOperationsCatalog](#)

You can navigate to the creation methods in each of the above categories. Using Visual Studio, the catalogs show up via IntelliSense.

```
var pipeline = mlContext.Transforms.Concatenate("Features", new[] { "Size" })
    .Append(mlContext.Regression.Trainers._
        M Equals
        M GetHashCode
        M GetType
        X LbfgsPoissonRegression
        X OnlineGradientDescent
        X Sdca
        M ToString
```

Build the pipeline

Inside each catalog is a set of extension methods. Let's look at how extension methods are used to create a training pipeline.

```
var pipeline = mlContext.Transforms.Concatenate("Features", new[] { "Size" })
    .Append(mlContext.Regression.Trainers.Sdca(labelColumnName: "Price", maximumNumberOfIterations:
100));
```

In the snippet, `Concatenate` and `Sdca` are both methods in the catalog. They each create an `IEstimator` object that is appended to the pipeline.

At this point, the objects are created only. No execution has happened.

Train the model

Once the objects in the pipeline have been created, data can be used to train the model.

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

Calling `Fit()` uses the input training data to estimate the parameters of the model. This is known as training the model. Remember, the linear regression model above had two model parameters: `bias` and `weight`. After the `Fit()` call, the values of the parameters are known. Most models will have many more parameters than this.

You can learn more about model training in [How to train your model](#).

The resulting model object implements the `ITransformer` interface. That is, the model transforms input data into predictions.

```
IDataView predictions = model.Transform(inputData);
```

Use the model

You can transform input data into predictions in bulk, or one input at a time. In the house price example, we did both: in bulk for the purpose of evaluating the model, and one at a time to make a new prediction. Let's look at making single predictions.

```
var size = new HouseData() { Size = 2.5F };
var predEngine = mlContext.CreatePredictionEngine<HouseData, Prediction>(model);
var price = predEngine.Predict(size);
```

The `CreatePredictionEngine()` method takes an input class and an output class. The field names and/or code attributes determine the names of the data columns used during model training and prediction. For more information, see [Make predictions with a trained model](#).

Data models and schema

At the core of an ML.NET machine learning pipeline are `DataView` objects.

Each transformation in the pipeline has an input schema (data names, types, and sizes that the transform expects to see on its input); and an output schema (data names, types, and sizes that the transform produces after the transformation).

If the output schema from one transform in the pipeline doesn't match the input schema of the next transform, ML.NET will throw an exception.

A data view object has columns and rows. Each column has a name and a type and a length. For example, the input columns in the house price example are `Size` and `Price`. They are both type and they are scalar quantities rather than vector ones.

Size	Price	Features	Score
Single	Single	Single	Single
Scalar	Scalar	Vector(1)	Scalar
1.1	1.2	[1.1]	1.29
1.9	2.3	[1.9]	2.14
2.8	3.0	[2.8]	3.10
3.4	3.7	[3.4]	3.75

All ML.NET algorithms look for an input column that is a vector. By default this vector column is called **Features**. This is why we concatenated the **Size** column into a new column called **Features** in our house price example.

```
var pipeline = mlContext.Transforms.Concatenate("Features", new[] { "Size" })
```

All algorithms also create new columns after they have performed a prediction. The fixed names of these new columns depend on the type of machine learning algorithm. For the regression task, one of the new columns is called **Score**. This is why we attributed our price data with this name.

```
public class Prediction
{
    [ColumnName("Score")]
    public float Price { get; set; }
}
```

You can find out more about output columns of different machine learning tasks in the [Machine Learning Tasks](#) guide.

An important property of DataView objects is that they are evaluated **lazily**. Data views are only loaded and operated on during model training and evaluation, and data prediction. While you are writing and testing your ML.NET application, you can use the Visual Studio debugger to take a peek at any data view object by calling the [Preview](#) method.

```
var debug = testPriceDataView.Preview();
```

You can watch the `debug` variable in the debugger and examine its contents. Do not use the Preview method in production code, as it significantly degrades performance.

Model Deployment

In real-life applications, your model training and evaluation code will be separate from your prediction. In fact, these two activities are often performed by separate teams. Your model development team can save the model for use in the prediction application.

```
mlContext.Model.Save(model, trainingData.Schema, "model.zip");
```

Next steps

- Learn how to build applications using different machine learning tasks with more realistic data sets in the [tutorials](#).
- Learn about specific topics in more depth in the [How To Guides](#).
- If you're super keen, you can dive straight into the [API Reference documentation](#).

Tutorial: Predict prices using regression with Model Builder

10/14/2022 • 7 minutes to read • [Edit Online](#)

Learn how to use ML.NET Model Builder to build a regression model to predict prices. The .NET console app that you develop in this tutorial predicts taxi fares based on historical New York taxi fare data.

The Model Builder price prediction template can be used for any scenario requiring a numerical prediction value. Example scenarios include: house price prediction, demand prediction, and sales forecasting.

In this tutorial, you learn how to:

- Prepare and understand the data
- Create a Model Builder Config file
- Choose a scenario
- Load the data
- Train the model
- Evaluate the model
- Use the model for predictions

NOTE

Model Builder is currently in Preview.

Pre-requisites

For a list of pre-requisites and installation instructions, visit the [Model Builder installation guide](#).

Create a console application

1. Create a **C# .NET Core Console Application** called "TaxiFarePrediction". Make sure **Place solution and project in the same directory** is **unchecked** (VS 2019).

Prepare and understand the data

1. Create a directory named *Data* in your project to store the data set files.
2. The data set used to train and evaluate the machine learning model is originally from the NYC TLC Taxi Trip data set.
 - a. To download the data set, navigate to the [taxi-fare-train.csv download link](#).
 - b. When the page loads, right-click anywhere on the page and select **Save as**.
 - c. Use the **Save As Dialog** to save the file in the *Data* folder you created at the previous step.
3. In **Solution Explorer**, right-click the *taxi-fare-train.csv* file and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

Each row in the `taxi-fare-train.csv` data set contains details of trips made by a taxi.

1. Open the taxi-fare-train.csv data set

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.

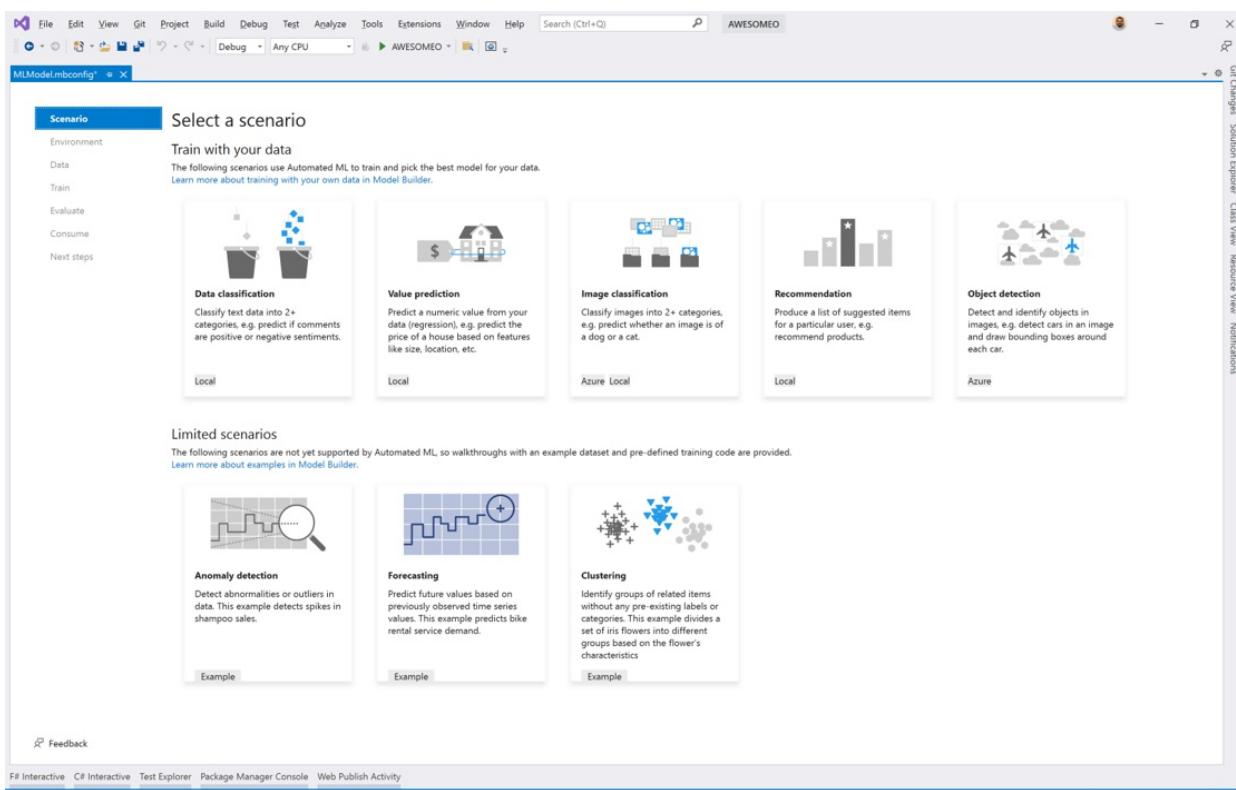
The **label** is the column you want to predict. When performing a regression task, the goal is to predict a numerical value. In this price prediction scenario, the cost of a taxi ride is being predicted. Therefore, the **fare_amount** is the label. The identified **features** are the inputs you give the model to predict the **label**. In this case, the rest of the columns with the exception of **trip_time_in_secs** are used as features or inputs to predict the fare amount.

Create Model Builder Config File

When first adding Model Builder to the solution it will prompt you to create an **mbconfig** file. The **mbconfig** file keeps track of everything you do in Model Builder to allow you to reopen the session.

1. In Solution Explorer, right-click the *TaxiFarePrediction* project, and select Add > Machine Learning Model....
2. Name the **mbconfig** project **TaxiFarePrediction**, and click the Add button.

Choose a scenario



To train your model, you need to select from the list of available machine learning scenarios provided by Model

Builder. In this case, the scenario is `Value prediction`.

1. In the scenario step of the Model Builder tool, select *Value prediction* scenario.

Select the environment

Model Builder can run the training on different environments depending on the scenario that was selected.

1. Confirm the `Local (CPU)` item is selected, and click the **Next step** button.

Load the data

Model Builder accepts data from two sources, a SQL Server database or a local file in csv or tsv format.

1. In the data step of the Model Builder tool, select *File* from the data source type selection.
2. Select the **Browse** button next to the text box and use File Explorer to browse and select the `taxis-fare-test.csv` in the *Data* directory
3. Choose `fare_amount` in the **Column to predict (Label)** dropdown.
4. Click the **Advanced data options** link.
5. In the **Column settings** tab, select the **Purpose** dropdown for the `trip_time_in_secs` column, and select **Ignore** to exclude it as a feature during training. Click the **Save** button to close the dialog.
6. Click the **Next step** button.

Train the model

The machine learning task used to train the price prediction model in this tutorial is regression. During the model training process, Model Builder trains separate models using different regression algorithms and settings to find the best performing model for your dataset.

The time required for the model to train is proportionate to the amount of data. Model Builder automatically selects a default value for **Time to train (seconds)** based on the size of your data source.

1. Leave the default value as is for *Time to train (seconds)* unless you prefer to train for a longer time.
2. Select *Start Training*.

Throughout the training process, progress data is displayed in the `Training results` section of the train step.

- Status displays the completion status of the training process.
- Best accuracy displays the accuracy of the best performing model found by Model Builder so far. Higher accuracy means the model predicted more correctly on test data.
- Best algorithm displays the name of the best performing algorithm performed found by Model Builder so far.
- Last algorithm displays the name of the algorithm most recently used by Model Builder to train the model.

Once training is complete the `mbconfig` file will have the generated model called `TaxiFarePrediction.zip` after training and two C# files with it:

- `TaxiFare.consumption.cs`: This file has a public method that will load the model and create a prediction engine with it and return the prediction.
- `TaxiFare.training.cs`: This file consists of the training pipeline that Model Builder came up with to build the best model including any hyperparameters that it used.

Click the **Next step** button to navigate to the evaluate step.

Evaluate the model

The result of the training step will be one model which had the best performance. In the evaluate step of the

Model Builder tool, in the **Best model** section, will contain the algorithm used by the best performing model in the *Model* entry along with metrics for that model in *RSquared*.

Additionally, in the **Output** window of Visual Studio, there will be a summary table containing top models and their metrics.

This section will also allow you to test your model by performing a single prediction. It will offer text boxes to fill in values and you can click the **Predict** button to get a prediction from the best model. By default this will be filled in by a random row in your dataset.

If you're not satisfied with your accuracy metrics, some easy ways to try and improve model accuracy are to increase the amount of time to train the model or use more data. Otherwise, click **Next step** to navigate to the consume step.

(Optional) Consume the model

This step will have project templates that you can use to consume the model. This step is optional and you can choose the method that best suits your needs on how to serve the model.

- Console App
- Web API

Console App

When adding a console app to your solution, you will be prompted to name the project.

1. Name the console project **TaxiFare_Console**.
2. Click **Add to solution** to add the project to your current solution.
3. Run the application.

The output generated by the program should look similar to the snippet below:

```
Predicted Fare: 15.020833
```

Web API

When adding a web API to your solution, you will be prompted to name the project.

1. Name the Web API project **TaxiFare_API**.
2. Click **Add to solution*** to add the project to your current solution.
3. Run the application.
4. Open PowerShell and enter the following code where PORT is the port your application is listening on.

```
$body = @{
    Vendor_id="CMT"
    Rate_code=1.0
    Passenger_count=1.0
    Trip_distance=3.8
    Payment_type="CRD"
}

Invoke-RestMethod "https://localhost:<PORT>/predict" -Method Post -Body ($body | ConvertTo-Json) -
ContentType "application/json"
```

5. If successful, the output should look similar to the text below:

```
score
-----
15.020833
```

Next Steps

In this tutorial, you learned how to:

- Prepare and understand the data
- Choose a scenario
- Load the data
- Train the model
- Evaluate the model
- Use the model for predictions

Additional Resources

To learn more about topics mentioned in this tutorial, visit the following resources:

- [Model Builder Scenarios](#)
- [Regression](#)
- [Regression Model Metrics](#)
- [NYC TLC Taxi Trip data set](#)

Analyze sentiment using the ML.NET CLI

10/14/2022 • 11 minutes to read • [Edit Online](#)

Learn how to use ML.NET CLI to automatically generate an ML.NET model and underlying C# code. You provide your dataset and the machine learning task you want to implement, and the CLI uses the AutoML engine to create model generation and deployment source code, as well as the classification model.

In this tutorial, you will do the following steps:

- Prepare your data for the selected machine learning task
- Run the 'mlnet classification' command from the CLI
- Review the quality metric results
- Understand the generated C# code to use the model in your application
- Explore the generated C# code that was used to train the model

NOTE

This topic refers to the ML.NET CLI tool, which is currently in Preview, and material may be subject to change. For more information, visit the [ML.NET](#) page.

The ML.NET CLI is part of ML.NET and its main goal is to "democratize" ML.NET for .NET developers when learning ML.NET so you don't need to code from scratch to get started.

You can run the ML.NET CLI on any command-prompt (Windows, Mac, or Linux) to generate good quality ML.NET models and source code based on training datasets you provide.

Pre-requisites

- [.NET Core 6 SDK](#) or later
- (Optional) [Visual Studio](#)
- [ML.NET CLI](#)

You can either run the generated C# code projects from Visual Studio or with `dotnet run` (.NET CLI).

Prepare your data

We are going to use an existing dataset used for a 'Sentiment Analysis' scenario, which is a binary classification machine learning task. You can use your own dataset in a similar way, and the model and code will be generated for you.

1. Download [The UCI Sentiment Labeled Sentences dataset zip file](#) (see citations in the following note), and unzip it on any folder you choose.

NOTE

The datasets this tutorial uses a dataset from the 'From Group to Individual Labels using Deep Features', Kotzias et al., KDD 2015, and hosted at the UCI Machine Learning Repository - Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

2. Copy the `yelp_labelled.txt` file into any folder you previously created (such as `/cli-test`).
3. Open your preferred command prompt and move to the folder where you copied the dataset file. For example:

```
cd /cli-test
```

Using any text editor such as Visual Studio Code, you can open, and explore the `yelp_labelled.txt` dataset file. You can see that the structure is:

- The file has no header. You will use the column's index.
- There are just two columns:

TEXT (COLUMN INDEX 0)	LABEL (COLUMN INDEX 1)
Wow... Loved this place.	1
Crust is not good.	0
Not tasty and the texture was just nasty.	0
...MANY MORE TEXT ROWS...	...(1 or 0)...

Make sure you close the dataset file from the editor.

Now, you are ready to start using the CLI for this 'Sentiment Analysis' scenario.

NOTE

After finishing this tutorial you can also try with your own datasets as long as they are ready to be used for any of the ML tasks currently supported by the ML.NET CLI Preview which are '*Binary Classification*', '*Classification*', '*Regression*', and '*Recommendation*'.

Run the 'mlnet classification' command

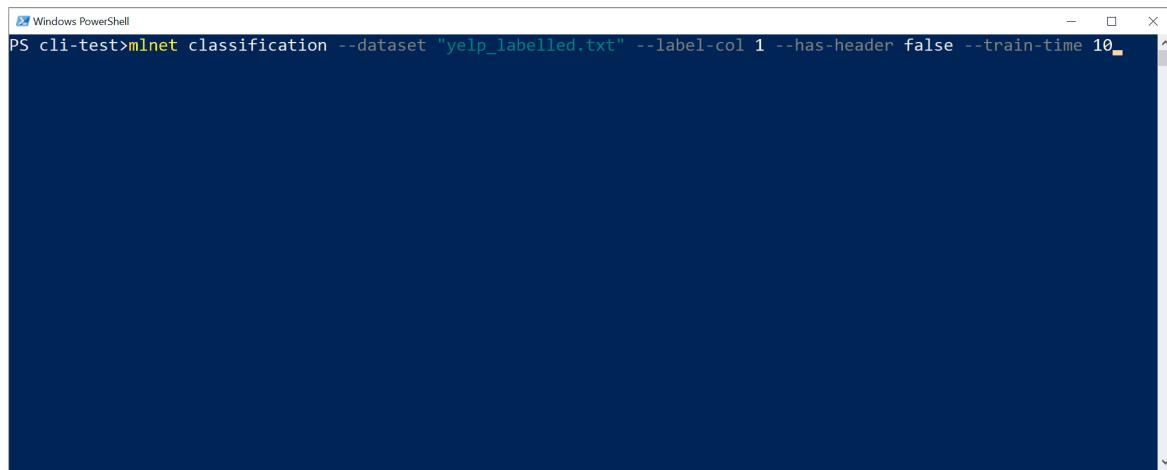
1. Run the following ML.NET CLI command:

```
mlnet classification --dataset "yelp_labelled.txt" --label-col 1 --has-header false --train-time 10
```

This command runs the `mlnet classification` command:

- for the **ML task of *classification***
- uses the **dataset file** `yelp_labelled.txt` as training and testing dataset (internally the CLI will either use cross-validation or split it in two datasets, one for training and one for testing)
- where the **objective/target column** you want to predict (commonly called '**label**') is the **column with index 1** (that is the second column, since the index is zero-based)
- does **not use a file header** with column names since this particular dataset file doesn't have a header
- the **targeted exploration/train time** for the experiment is **10 seconds**

You will see output from the CLI, similar to:



```
Windows PowerShell
PS cli-test>mlnet classification --dataset "yelp_labelled.txt" --label-col 1 --has-header false --train-time 10
```

In this particular case, in only 10 seconds and with the small dataset provided, the CLI tool was able to run quite a few iterations, meaning training multiple times based on different combinations of algorithms/configuration with different internal data transformations and algorithm's hyper-parameters.

Finally, the "best quality" model found in 10 seconds is a model using a particular trainer/algorithm with any specific configuration. Depending on the exploration time, the command can produce a different result. The selection is based on the multiple metrics shown, such as `Accuracy`.

Understanding the model's quality metrics

The first and easiest metric to evaluate a binary-classification model is the accuracy, which is simple to understand. "Accuracy is the proportion of correct predictions with a test data set.". The closer to 100% (1.00), the better.

However, there are cases where just measuring with the Accuracy metric is not enough, especially when the label (0 and 1 in this case) is unbalanced in the test dataset.

For additional metrics and more [detailed information about the metrics](#) such as Accuracy, AUC, AUCPR, and F1-score used to evaluate the different models, see [Understanding ML.NET metrics](#).

NOTE

You can try this very same dataset and specify a few minutes for `--max-exploration-time` (for instance three minutes so you specify 180 seconds) which will find a better "best model" for you with a different training pipeline configuration for this dataset (which is pretty small, 1000 rows).

In order to find a "best/good quality" model that is a "production-ready model" targeting larger datasets, you should make experiments with the CLI usually specifying much more exploration time depending on the size of the dataset. In fact, in many cases you might require multiple hours of exploration time especially if the dataset is large on rows and columns.

2. The previous command execution has generated the following assets:

- A serialized model .zip ("best model") ready to use.
- C# code to run/score that generated model (To make predictions in your end-user apps with that model).
- C# training code used to generate that model (Learning purposes).
- A log file with all the iterations explored having specific detailed information about each algorithm tried with its combination of hyper-parameters and data transformations.

The first two assets (.ZIP file model and C# code to run that model) can directly be used in your end-user apps (ASP.NET Core web app, services, desktop app, etc.) to make predictions with that generated ML model.

The third asset, the training code, shows you what ML.NET API code was used by the CLI to train the generated model, so you can investigate what specific trainer/algorithm and hyper-parameters were selected by the CLI.

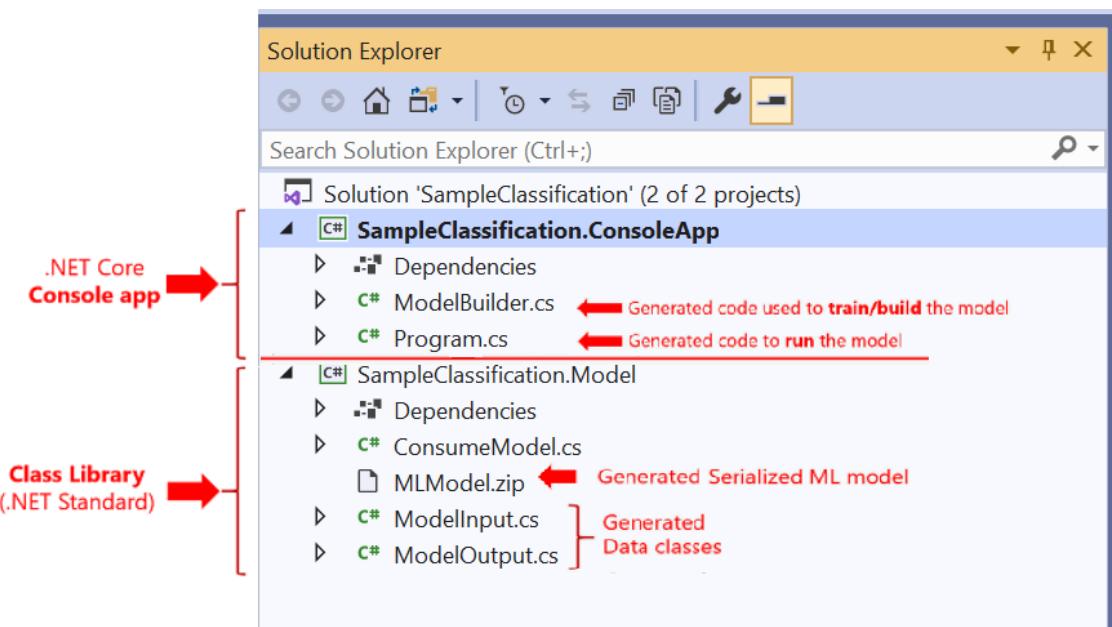
Those enumerated assets are explained in the following steps of the tutorial.

Explore the generated C# code to use for running the model to make predictions

1. In Visual Studio open the solution generated in the folder named `SampleClassification` within your original destination folder (in the tutorial was named `/cli-test`). You should see a solution similar to:

NOTE

In the tutorial we suggest to use Visual Studio, but you can also explore the generated C# code (two projects) with any text editor and run the generated console app with the `dotnet CLI` on macOS, Linux or Windows machine.



- The generated **class library** containing the serialized ML model (.zip file) and the data classes (data models) is something you can directly use in your end-user application, even by directly referencing that class library (or moving the code, as you prefer).
 - The generated **console app** contains execution code that you must review and then you usually reuse the 'scoring code' (code that runs the ML model to make predictions) by moving that simple code (just a few lines) to your end-user application where you want to make the predictions.
2. Open the `ModelInput.cs` and `ModelOutput.cs` class files within the class library project. You will see that these classes are 'data classes' or POCO classes used to hold data. It is 'boilerplate code' but useful to have it generated if your dataset has tens or even hundreds of columns.
 - The `ModelInput` class is used when reading data from the dataset.
 - The `ModelOutput` class is used to get the prediction result (prediction data).
 3. Open the `Program.cs` file and explore the code. In just a few lines, you are able to run the model and make a sample prediction.

```

static void Main(string[] args)
{
    // Create single instance of sample data from first line of dataset for model input
    ModelInput sampleData = new ModelInput()
    {
        Col0 = @"Wow... Loved this place.",
    };

    // Make a single prediction on the sample data and print results
    var predictionResult = ConsumeModel.Predict(sampleData);

    Console.WriteLine("Using model to make single prediction -- Comparing actual Col1 with predicted
Col1 from sample data...\n\n");
    Console.WriteLine($"Col0: {sampleData.Col0}");
    Console.WriteLine($"\\n\\nPredicted Col1 value {predictionResult.PredictedLabel} \\nPredicted Col1
scores: [{String.Join(", ", predictionResult.Score)}]\\n\\n");
    Console.WriteLine("===== End of process, hit any key to finish =====");
    Console.ReadKey();
}

```

- The first lines of code create a *single sample data*, in this case based on the first row of your dataset to be used for the prediction. You can also create your own 'hard-coded' data by updating the code:

```

ModelInput sampleData = new ModelInput()
{
    Col0 = "The ML.NET CLI is great for getting started. Very cool!"
};

```

- The next line of code uses the `ConsumeModel.Predict()` method on the specified input data to make a prediction and return the results (based on the `ModelOutput.cs` schema).
- The last lines of code print out the properties of the sample data (in this case the `Comment`) as well as the Sentiment prediction and corresponding Scores for positive sentiment (1) and negative sentiment (2).

- Run the project, either using the original sample data loaded from the first row of the dataset or by providing your own custom hard-coded sample data. You should get a prediction comparable to:

```

C:\Users\Documents\cli-test\SampleClassification\SampleClassification.ConsoleApp\bin\Debug\netcor...
Using model to make single prediction -- Comparing actual Col1 with predicted Col1 from sample data...

Col0: Wow... Loved this place.

Predicted Col1 value 1
Predicted Col1 scores: [0.9961542, 0.0038458048]

===== End of process, hit any key to finish =====

```

- Try changing the hard-coded sample data to other sentences with different sentiment and see how the model predicts positive or negative sentiment.

Infuse your end-user applications with ML model predictions

You can use similar 'ML model scoring code' to run the model in your end-user application and make predictions.

For instance, you could directly move that code to any Windows desktop application such as **WPF** and **WinForms** and run the model in the same way than it was done in the console app.

However, the way you implement those lines of code to run an ML model should be optimized (that is, cache the model .zip file and load it once) and have singleton objects instead of creating them on every request, especially if your application needs to be scalable such as a web application or distributed service, as explained in the following section.

Running ML.NET models in scalable ASP.NET Core web apps and services (multi-threaded apps)

The creation of the model object (`ITransformer` loaded from a model's .zip file) and the `PredictionEngine` object should be optimized especially when running on scalable web apps and distributed services. For the first case, the model object (`ITransformer`) the optimization is straightforward. Since the `ITransformer` object is thread-safe, you can cache the object as a singleton or static object so you load the model once.

For the second object, the `PredictionEngine` object, it is not so easy because the `PredictionEngine` object is not thread-safe, therefore you cannot instantiate this object as singleton or static object in an ASP.NET Core app. This thread-safe and scalability problem is deeply discussed in this [Blog Post](#).

However, things got a lot easier for you than what's explained in that blog post. We worked on a simpler approach for you and have created a nice '**.NET Core Integration Package**' that you can easily use in your ASP.NET Core apps and services by registering it in the application DI services (Dependency Injection services) and then directly use it from your code. Check the following tutorial and example for doing that:

- [Tutorial: Running ML.NET models on scalable ASP.NET Core web apps and WebAPIs](#)
- [Sample: Scalable ML.NET model on ASP.NET Core WebAPI](#)

Explore the generated C# code that was used to train the "best quality" model

For more advanced learning purposes, you can also explore the generated C# code that was used by the CLI tool to train the generated model.

That 'training model code' is currently generated in the custom class generated named `ModelBuilder` so you can investigate that training code.

More importantly, for this particular scenario (Sentiment Analysis model) you can also compare that generated training code with the code explained in the following tutorial:

- Compare: [Tutorial: Use ML.NET in a sentiment analysis binary classification scenario](#).

It is interesting to compare the chosen algorithm and pipeline configuration in the tutorial with the code generated by the CLI tool. Depending on how much time you spend iterating and searching for better models, the chosen algorithm might be different along with its particular hyper-parameters and pipeline configuration.

In this tutorial, you learned how to:

- Prepare your data for the selected ML task (problem to solve)
- Run the 'mlnet classification' command in the CLI tool
- Review the quality metric results
- Understand the generated C# code to run the model (Code to use in your end-user app)
- Explore the generated C# code that was used to train the "best quality" model (earning purposes)

See also

- [Automate model training with the ML.NET CLI](#)
- [Tutorial: Running ML.NET models on scalable ASP.NET Core web apps and WebAPIs](#)

- [Sample: Scalable ML.NET model on ASP.NET Core WebAPI](#)
- [ML.NET CLI auto-train command reference guide](#)
- [Telemetry in ML.NET CLI](#)

Tutorial: Analyze sentiment of website comments in a web application using ML.NET Model Builder

10/14/2022 • 9 minutes to read • [Edit Online](#)

Learn how to analyze sentiment from comments in real time inside a web application.

This tutorial shows you how to create an ASP.NET Core Razor Pages application that classifies sentiment from website comments in real time.

In this tutorial, you learn how to:

- Create an ASP.NET Core Razor Pages application
- Prepare and understand the data
- Choose a scenario
- Load the data
- Train the model
- Evaluate the model
- Use the model for predictions

NOTE

Model Builder is currently in Preview.

You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) repository.

Pre-requisites

For a list of pre-requisites and installation instructions, visit the [Model Builder installation guide](#).

Create a Razor Pages application

1. Create an **ASP.NET Core Razor Pages Application**.
 - a. Open Visual Studio and select **File > New > Project** from the menu bar.
 - b. In the **New Project** dialog, select the **Visual C#** node followed by the **Web** node.
 - c. Then select the **ASP.NET Core Web Application** project template.
 - d. In the **Name** text box, type "SentimentRazor".
 - e. Make sure **Place solution and project in the same directory** is **unchecked** (VS 2019), or **Create directory for solution** is **checked** (VS 2017).
 - f. Select the **OK** button.
 - g. Choose **Web Application** in the window that displays the different types of ASP.NET Core Projects, and then select the **OK** button.

Prepare and understand the data

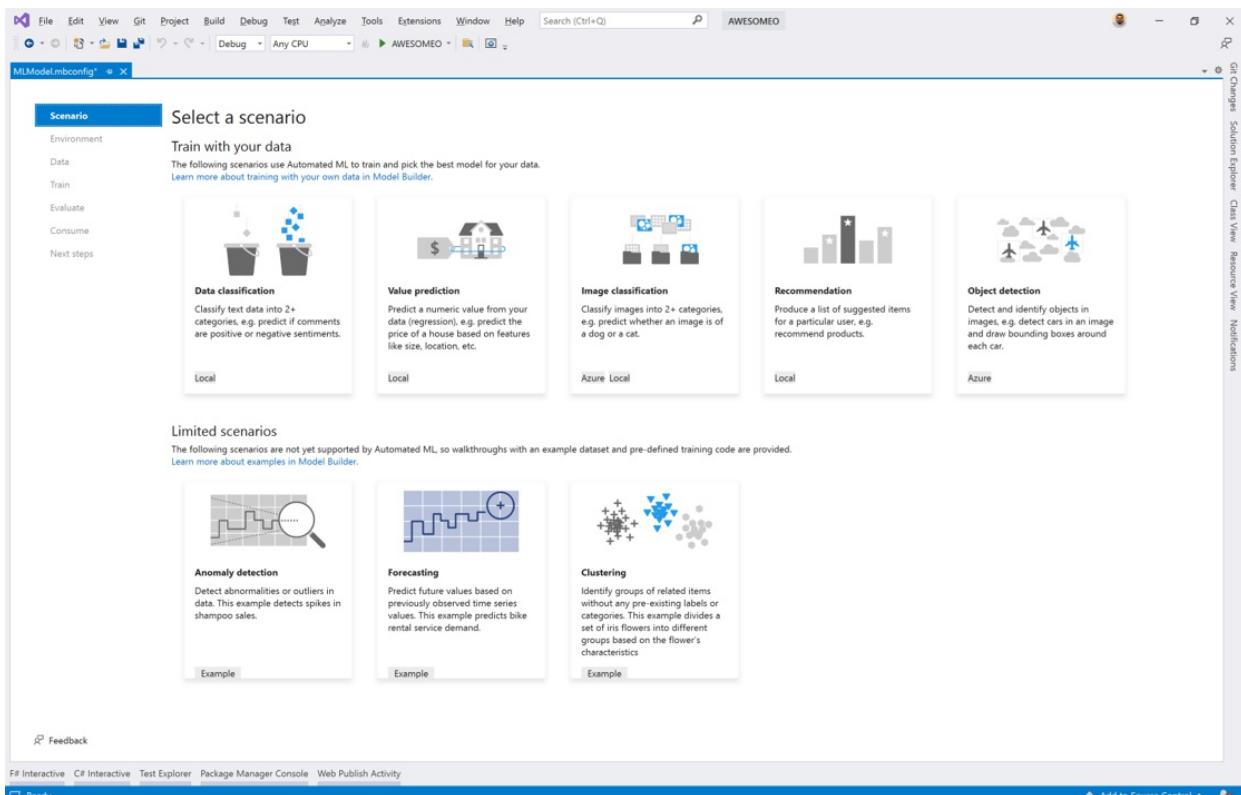
Download [Wikipedia detox dataset](#). When the webpage opens, right-click on the page, select **Save As** and save the file anywhere on your computer.

Each row in the *wikipedia-detox-250-line-data.tsv* dataset represents a different review left by a user on

Wikipedia. The first column represents the sentiment of the text (0 is non-toxic, 1 is toxic), and the second column represents the comment left by the user. The columns are separated by tabs. The data looks like the following:

SENTIMENT	SENTIMENTTEXT
1	==RUDE== Dude, you are rude upload that carl picture back, or else.
1	== OK! == IM GOING TO VANDALIZE WILD ONES WIKI THEN!!!
0	I hope this helps.

Choose a scenario



To train your model, you need to select from the list of available machine learning scenarios provided by Model Builder.

1. In **Solution Explorer**, right-click the *SentimentRazor* project, and select **Add > Machine Learning Model....**
2. For this sample, the scenario is sentiment analysis. In the *scenario* step of the Model Builder tool, select the **Sentiment Analysis** scenario.

Load the data

Model Builder accepts data from two sources, a SQL Server database or a local file in **.csv** or **.tsv** format.

1. In the data step of the Model Builder tool, select **File** from the data source dropdown.
2. Select the button next to the **Select a file** text box and use File Explorer to browse and select the *wikipedia-detox-250-line-data.tsv* file.
3. Choose **Sentiment** in the **Column to predict (Label)** dropdown.

4. Leave the default values for the **Input Columns (Features)** dropdown.
5. Select the **Train** link to move to the next step in the Model Builder tool.

Train the model

The machine learning task used to train the sentiment analysis model in this tutorial is binary classification. During the model training process, Model Builder trains separate models using different binary classification algorithms and settings to find the best performing model for your dataset.

The time required for the model to train is proportionate to the amount of data. Model Builder automatically selects a default value for **Time to train (seconds)** based on the size of your data source.

1. Although Model Builder sets the value of **Time to train (seconds)** to 10 seconds, increase it to 30 seconds. Training for a longer period of time allows Model Builder to explore a larger number of algorithms and combination of parameters in search of the best model.
2. Select **Start Training**.

Throughout the training process, progress data is displayed in the **Progress** section of the train step.

- **Status** displays the completion status of the training process.
- **Best accuracy** displays the accuracy of the best performing model found by Model Builder so far. Higher accuracy means the model predicted more correctly on test data.
- **Best algorithm** displays the name of the best performing algorithm performed found by Model Builder so far.
- **Last algorithm** displays the name of the algorithm most recently used by Model Builder to train the model.

3. Once training is complete, select the **evaluate** link to move to the next step.

Evaluate the model

The result of the training step will be one model that has the best performance. In the evaluate step of the Model Builder tool, the output section will contain the algorithm used by the best-performing model in the **Best Model** entry along with metrics in **Best Model Accuracy**. Additionally, a summary table containing the top five models and their metrics is shown.

If you're not satisfied with your accuracy metrics, some easy ways to try to improve model accuracy are to increase the amount of time to train the model or use more data. Otherwise, select the **code** link to move to the final step in the Model Builder tool.

Add the code to make predictions

Two projects will be created as a result of the training process.

Reference the trained model

1. In the **code** step of the Model Builder tool, select **Add Projects** to add the autogenerated projects to the solution.

The following projects should appear in the **Solution Explorer**:

- *SentimentRazorML.ConsoleApp*: A .NET Core Console application that contains the model training and prediction code.
- *SentimentRazorML.Model*: A .NET Standard class library containing the data models that define the schema of input and output model data as well as the saved version of the best performing model during training.

For this tutorial, only the *SentimentRazorML.Model* project is used because predictions will be made in the *SentimentRazor* web application rather than in the console. Although the *SentimentRazorML.ConsoleApp* won't be used for scoring, it can be used to retrain the model using new data at a later time. Retraining is outside the scope of this tutorial though.

Configure the PredictionEngine pool

To make a single prediction, you have to create a `PredictionEngine<TSrc,TDst>`. `PredictionEngine<TSrc,TDst>` is not thread-safe. Additionally, you have to create an instance of it everywhere it's needed within your application. As your application grows, this process can become unmanageable. For improved performance and thread safety, use a combination of dependency injection and the `PredictionEnginePool` service, which creates an `ObjectPool<T>` of `PredictionEngine<TSrc,TDst>` objects for use throughout your application.

1. Install the *Microsoft.Extensions.ML* NuGet package:
 - a. In **Solution Explorer**, right-click the project and select **Manage NuGet Packages**.
 - b. Choose "nuget.org" as the Package source.
 - c. Select the **Browse** tab and search for **Microsoft.Extensions.ML**.
 - d. Select the package in the list, and select the **Install** button.
 - e. Select the **OK** button on the **Preview Changes** dialog
 - f. Select the **I Accept** button on the **License Acceptance** dialog if you agree with the license terms for the packages listed.
2. Open the *Startup.cs* file in the *SentimentRazor* project.
3. Add the following using statements to reference the *Microsoft.Extensions.ML* NuGet package and *SentimentRazorML.Model* project:

```
using System.IO;
using Microsoft.Extensions.ML;
using SentimentRazorML.Model;
```

4. Create a global variable to store the location of the trained model file.

```
private readonly string _modelPath;
```

5. The model file is stored in the build directory alongside the assembly files of your application. To make it easier to access, create a helper method called `GetAbsolutePath` after the `Configure` method

```
public static string GetAbsolutePath(string relativePath)
{
    FileInfo _dataRoot = new FileInfo(typeof(Program).Assembly.Location);
    string assemblyFolderPath = _dataRoot.Directory.FullName;

    string fullPath = Path.Combine(assemblyFolderPath, relativePath);
    return fullPath;
}
```

6. Use the `GetAbsolutePath` method in the `Startup` class constructor to set the `_modelPath`.

```
_modelPath = GetAbsolutePath("MLModel.zip");
```

7. Configure the `PredictionEnginePool` for your application in the `ConfigureServices` method:

```
services.AddPredictionEnginePool<ModelInput, ModelOutput>()
    .FromFile(_modelPath);
```

Create sentiment analysis handler

Predictions will be made inside the main page of the application. Therefore, a method that takes the user input and uses the `PredictionEnginePool` to return a prediction needs to be added.

1. Open the `Index.cshtml.cs` file located in the `Pages` directory and add the following using statements:

```
using Microsoft.Extensions.ML;
using SentimentRazorML.Model;
```

In order to use the `PredictionEnginePool` configured in the `Startup` class, you have to inject it into the constructor of the model where you want to use it.

2. Add a variable to reference the `PredictionEnginePool` inside the `IndexModel` class.

```
private readonly PredictionEnginePool<ModelInput, ModelOutput> _predictionEnginePool;
```

3. Create a constructor in the `IndexModel` class and inject the `PredictionEnginePool` service into it.

```
public IndexModel(PredictionEnginePool<ModelInput, ModelOutput> predictionEnginePool)
{
    _predictionEnginePool = predictionEnginePool;
}
```

4. Create a method handler that uses the `PredictionEnginePool` to make predictions from user input received from the web page.

- a. Below the `OnGet` method, create a new method called `OnGetAnalyzeSentiment`

```
public IActionResult OnGetAnalyzeSentiment([FromQuery] string text)
{
}
```

- b. Inside the `OnGetAnalyzeSentiment` method, return *Neutral*/sentiment if the input from the user is blank or null.

```
if (String.IsNullOrEmpty(text)) return Content("Neutral");
```

- c. Given a valid input, create a new instance of `ModelInput`.

```
var input = new ModelInput { SentimentText = text };
```

- d. Use the `PredictionEnginePool` to predict sentiment.

```
var prediction = _predictionEnginePool.Predict(input);
```

- e. Convert the predicted `bool` value into toxic or not toxic with the following code.

```
var sentiment = Convert.ToBoolean(prediction.Prediction) ? "Toxic" : "Not Toxic";
```

- f. Finally, return the sentiment back to the web page.

```
return Content(sentiment);
```

Configure the web page

The results returned by the `OnGetAnalyzeSentiment` will be dynamically displayed on the `Index` web page.

1. Open the `Index.cshtml` file in the `Pages` directory and replace its contents with the following code:

```
@page
@model IndexModel
 @{
    ViewData["Title"] = "Home page";
}

<div class="text-center">
    <h2>Live Sentiment</h2>

    <p><textarea id="Message" cols="45" placeholder="Type any text like a short review"></textarea>
    </p>

    <div class="sentiment">
        <h4>Your sentiment is...</h4>
        <p>utral</p>

        <div class="marker">
            <div id="markerPosition" style="left: 45%; ">
                <div>▲</div>
                <label id="markerValue">Neutral</label>
            </div>
        </div>
    </div>
</div>
```

2. Next, add css styling code to the end of the `site.css` page in the `wwwroot\css` directory:

```
/* Style for sentiment display */
```

```
.sentiment {  
    background-color: #eee;  
    position: relative;  
    display: inline-block;  
    padding: 1rem;  
    padding-bottom: 0;  
    border-radius: 1rem;  
}  
  
.sentiment h4 {  
    font-size: 16px;  
    text-align: center;  
    margin: 0;  
    padding: 0;  
}  
  
.sentiment p {  
    font-size: 50px;  
}  
  
.sentiment .marker {  
    position: relative;  
    left: 22px;  
    width: calc(100% - 68px);  
}  
  
.sentiment .marker > div {  
    transition: 0.3s ease-in-out;  
    position: absolute;  
    margin-left: -30px;  
    text-align: center;  
}  
  
.sentiment .marker > div > div {  
    font-size: 50px;  
    line-height: 20px;  
    color: green;  
}  
  
.sentiment .marker > div label {  
    font-size: 30px;  
    color: gray;  
}
```

3. After that, add code to send inputs from the web page to the `OnGetAnalyzeSentiment` handler.

- In the `site.js` file located in the `wwwroot/js` directory, create a function called `getSentiment` to make a GET HTTP request with the user input to the `OnGetAnalyzeSentiment` handler.

```
function getSentiment(userInput) {  
    return fetch(`Index?handler=AnalyzeSentiment&text=${userInput}`)  
        .then((response) => {  
            return response.text();  
        })  
}
```

- Below that, add another function called `updateMarker` to dynamically update the position of the marker on the web page as sentiment is predicted.

```

function updateMarker(markerPosition, sentiment) {
    $("#markerPosition").attr("style", `left:${markerPosition}%`);
    $("#markerValue").text(sentiment);
}

```

- c. Create an event handler function called `updateSentiment` to get the input from the user, send it to the `OnGetAnalyzeSentiment` function using the `getSentiment` function and update the marker with the `updateMarker` function.

```

function updateSentiment() {

    var userInput = $("#Message").val();

    getSentiment(userInput)
        .then((sentiment) => {
            switch (sentiment) {
                case "Not Toxic":
                    updateMarker(100.0,sentiment);
                    break;
                case "Toxic":
                    updateMarker(0.0,sentiment);
                    break;
                default:
                    updateMarker(45.0, "Neutral");
            }
        });
}

```

- d. Finally, register the event handler and bind it to the `textarea` element with the `id=Message` attribute.

```

$("#Message").on('change input paste', updateSentiment)

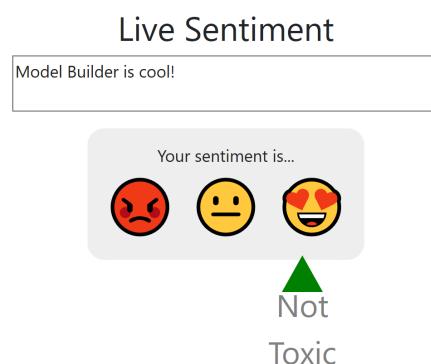
```

Run the application

Now that your application is set up, run the application, which should launch in your browser.

When the application launches, enter *Model Builder is cool!* into the text area. The predicted sentiment displayed should be *Not Toxic*.

SentimentRazor Home Privacy



If you need to reference the Model Builder generated projects at a later time inside of another solution, you can find them inside the `C:\Users\%USERNAME%\AppData\Local\Temp\MLVSTools` directory.

Next steps

In this tutorial, you learned how to:

- Create an ASP.NET Core Razor Pages application
- Prepare and understand the data
- Choose a scenario
- Load the data
- Train the model
- Evaluate the model
- Use the model for predictions

Additional Resources

To learn more about topics mentioned in this tutorial, visit the following resources:

- [Model Builder Scenarios](#)
- [Binary Classification](#)
- [Binary Classification Model Metrics](#)

Tutorial: Classify the severity of restaurant health violations with Model Builder

10/14/2022 • 6 minutes to read • [Edit Online](#)

Learn how to build a multiclass classification model using Model Builder to categorize the risk level of restaurant violations found during health inspections.

In this tutorial, you learn how to:

- Prepare and understand the data
- Create a Model Builder config file
- Choose a scenario
- Load data from a database
- Train the model
- Evaluate the model
- Use the model for predictions

NOTE

Model Builder is currently in Preview.

Prerequisites

For a list of prerequisites and installation instructions, visit the [Model Builder installation guide](#).

Model Builder multiclass classification overview

This sample creates a C# .NET Core console application that categorizes the risk of health violations using a machine learning model built with Model Builder. You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) GitHub repository.

Create a console application

1. Create a C# .NET Core console application called "RestaurantViolations".

Prepare and understand the data

The data set used to train and evaluate the machine learning model is originally from the [San Francisco Department of Public Health Restaurant Safety Scores](#). For convenience, the dataset has been condensed to only include the columns relevant to train the model and make predictions. Visit the following website to learn more about the [dataset](#).

[Download the Restaurant Safety Scores dataset](#) and unzip it.

Each row in the dataset contains information regarding violations observed during an inspection from the Health Department and a risk assessment of the threat those violations present to public health and safety.

INSPECTIONTYPE	VIOLATIONDESCRIPTION	RISKCATEGORY
Routine - Unscheduled	Inadequately cleaned or sanitized food contact surfaces	Moderate Risk
New Ownership	High risk vermin infestation	High Risk
Routine - Unscheduled	Wiping cloths not clean or properly stored or inadequate sanitizer	Low Risk

- **InspectionType:** the type of inspection. This can either be a first-time inspection for a new establishment, a routine inspection, a complaint inspection, and many other types.
- **ViolationDescription:** a description of the violation found during inspection.
- **RiskCategory:** the risk severity a violation poses to public health and safety.

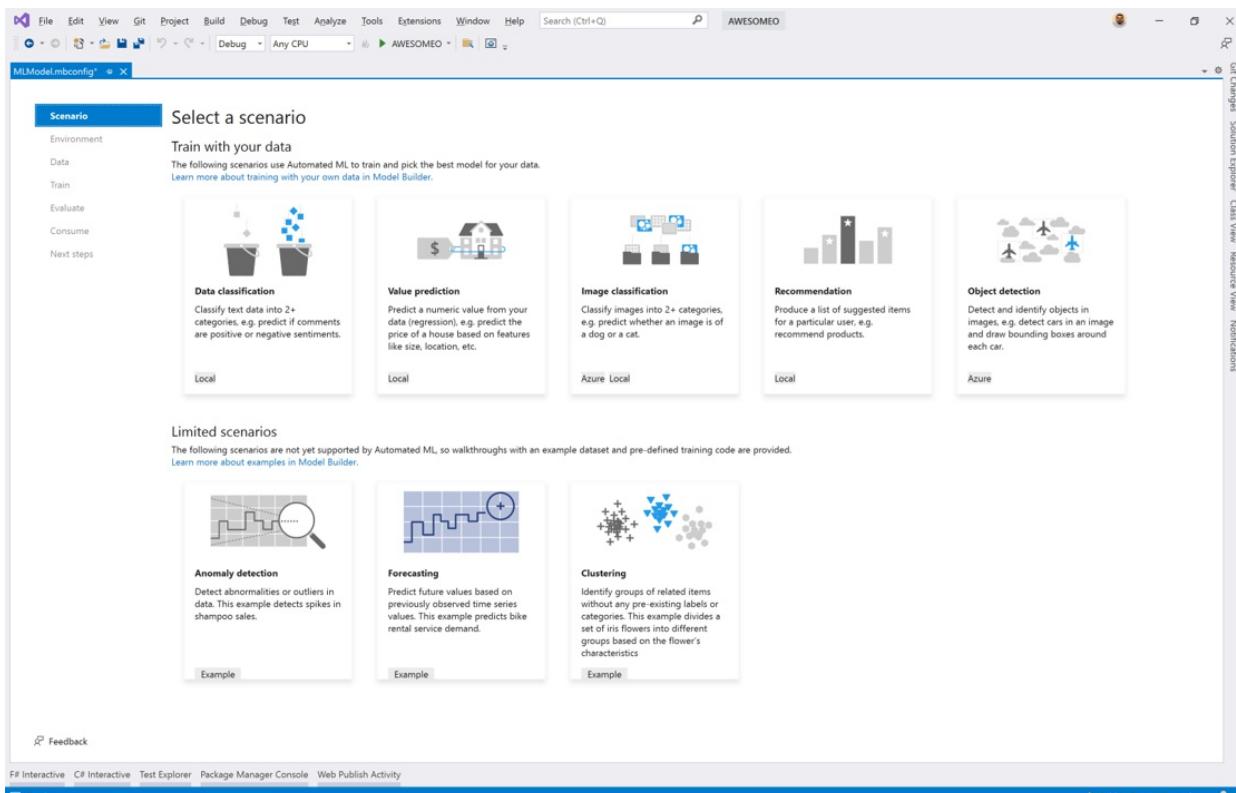
The `label` is the column you want to predict. When performing a classification task, the goal is to assign a category (text or numerical). In this classification scenario, the severity of the violation is assigned the value of low, moderate, or high risk. Therefore, the **RiskCategory** is the label. The `features` are the inputs you give the model to predict the `label`. In this case, the **InspectionType** and **ViolationDescription** are used as features or inputs to predict the **RiskCategory**.

Create Model Builder Config File

When first adding Model Builder to the solution it will prompt you to create an `mbconfig` file. The `mbconfig` file keeps track of everything you do in Model Builder to allow you to reopen the session.

1. In **Solution Explorer**, right-click the *RestaurantViolations* project, and select **Add > Machine Learning Model....**
2. Name the `mbconfig` project **RestaurantViolationsPrediction**, and click the **Add** button.

Choose a scenario



To train your model, select from the list of available machine learning scenarios provided by Model Builder. In this case, the scenario is *Data classification*.

1. For this sample, the task is multiclass classification. In the *Scenario* step of Model Builder, select the **Data classification** scenario.

Load the data

Model Builder accepts data from a SQL Server database or a local file in `csv`, `tsv`, or `txt` format.

1. In the data step of the Model Builder tool, select **SQL Server** from the data source type selection.
2. Select the **Choose data source** button.
 - a. In the **Choose Data Source** dialog, select **Microsoft SQL Server Database File**.
 - b. Uncheck the **Always use this selection** checkbox and click **Continue**.
 - c. In the **Connection Properties** dialog, select **Browse** and select the downloaded `RestaurantScores.mdf` file.
 - d. Select **OK**.
3. Choose **Violations** from the **Table** dropdown.
4. Choose **RiskCategory** in the **Column to predict (Label)** dropdown.
5. Leave the default selections in **Advanced data options**.
6. Click the **Next step** button to move to the train step in Model Builder.

Train the model

The machine learning task used to train the issue classification model in this tutorial is multiclass classification. During the model training process, Model Builder trains separate models using different multiclass classification algorithms and settings to find the best performing model for your dataset.

The time required for the model to train is proportional to the amount of data. Model Builder automatically selects a default value for **Time to train (seconds)** based on the size of your data source.

1. Model Builder sets the value of **Time to train (seconds)** to 60 seconds. Training for a longer period of time allows Model Builder to explore a larger number of algorithms and combination of parameters in search of the best model.
2. Click **Start Training**.

Throughout the training process, progress data is displayed in the `Training results` section of the train step.

- Status displays the completion status of the training process.
- Best accuracy displays the accuracy of the best performing model found by Model Builder so far. Higher accuracy means the model predicted more correctly on test data.
- Best algorithm displays the name of the best performing algorithm performed found by Model Builder so far.
- Last algorithm displays the name of the algorithm most recently used by Model Builder to train the model.

Once training is complete the `mbconfig` file will have the generated model called `RestaurantViolationsPrediction.zip` after training and two C# files with it:

- `RestaurantViolationsPrediction.consumption.cs`: This file has a public method that will load the model and create a prediction engine with it and return the prediction.
- `RestaurantViolationsPrediction.training.cs`: This file consists of the training pipeline that Model Builder came up with to build the best model including any hyperparameters that it used.

Click the **Next step** button to navigate to the evaluate step.

Evaluate the model

The result of the training step will be one model which had the best performance. In the evaluate step of the Model Builder tool, in the **Best model** section, will contain the algorithm used by the best performing model in the *Model* entry along with metrics for that model in *Accuracy*.

Additionally, in the **Output** window of Visual Studio, there will be a summary table containing top models and their metrics.

This section will also allow you to test your model by performing a single prediction. It will offer text boxes to fill in values and you can click the **Predict** button to get a prediction from the best model. By default this will be filled in by a random row in your dataset.

(Optional) Consume the model

This step will have project templates that you can use to consume the model. This step is optional and you can choose the method that best suits your needs on how to serve the model.

- Console App
- Web API

Console App

When adding a console app to your solution, you will be prompted to name the project.

1. Name the console project **RestaurantViolationsPrediction_Console**.
2. Click **Add to solution** to add the project to your current solution.
3. Run the application.

The output generated by the program should look similar to the snippet below:

```
InspectionType: Routine - Unscheduled  
ViolationDescription: Moderate risk food holding temperature  
  
Predicted RiskCategory: Moderate Risk
```

Web API

When adding a web API to your solution, you will be prompted to name the project.

1. Name the Web API project **RestaurantViolationsPrediction_API**.
2. Click **Add to solution*** to add the project to your current solution.
3. Run the application.
4. Open PowerShell and enter the following code where PORT is the port your application is listening on.

```
$body = @  
    InspectionType="Reinspection/Followup"  
    ViolationDescription="Inadequately cleaned or sanitized food contact surfaces"  
}  
  
Invoke-RestMethod "https://localhost:<PORT>/predict" -Method Post -Body ($body | ConvertTo-Json) -  
ContentType "application/json"
```

5. If successful, the output should look similar to the text below. The output has the predicted **RiskCategory** as *Moderate Risk* and it has the scores of each of the input labels - *Low Risk, High Risk*,

and *Moderate Risk*.

```
prediction      score
-----      -----
Moderate Risk {0.055566575, 0.058012854, 0.88642055}
```

Congratulations! You've successfully built a machine learning model to categorize the risk of health violations using Model Builder. You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) GitHub repository.

Additional resources

To learn more about topics mentioned in this tutorial, visit the following resources:

- [Model Builder Scenarios](#)
- [Multiclass Classification](#)
- [Multiclass Classification Model Metrics](#)

Train an image classification model in Azure using Model Builder

10/14/2022 • 8 minutes to read • [Edit Online](#)

Learn how to train an image classification model in Azure using Model Builder to categorize land use from satellite images.

This tutorial shows you how to create a C# class library to categorize land use based on satellite images with Model Builder.

In this tutorial, you:

- Prepare and understand the data
- Create a Model Builder config file
- Choose a scenario
- Load the data
- Create an experiment in Azure
- Train the model
- Evaluate the model
- Consume the model

NOTE

Model Builder is currently in Preview.

Prerequisites

- For a list of pre-requisites and installation instructions, visit the [Model Builder installation guide](#).
- Azure account. If you don't have one, [create a free Azure account](#).
- ASP.NET and web development workload.

Model Builder image classification overview

This sample creates C# class library that categorizes land use from map satellite imagery using a deep learning model trained on Azure with Model Builder. You can find the source code for this tutorial in the [dotnet/machinelearning-samples](#) GitHub repository

Create a C# Class Library

Create a C# Class Library called "LandUse".

Prepare and understand the data

NOTE

The data for this tutorial is from:

- Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification. Patrick Helber, Benjamin Bischke, Andreas Dengel, Damian Borth. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2019.
- Introducing EuroSAT: A Novel Dataset and Deep Learning Benchmark for Land Use and Land Cover Classification. Patrick Helber, Benjamin Bischke, Andreas Dengel. 2018 IEEE International Geoscience and Remote Sensing Symposium, 2018.

The EuroSAT dataset contains a collection of satellite images divided into ten categories (rural, industrial, river, etc.). The original dataset contains 27,000 images. For convenience, this tutorial only uses 2,000 of those images.



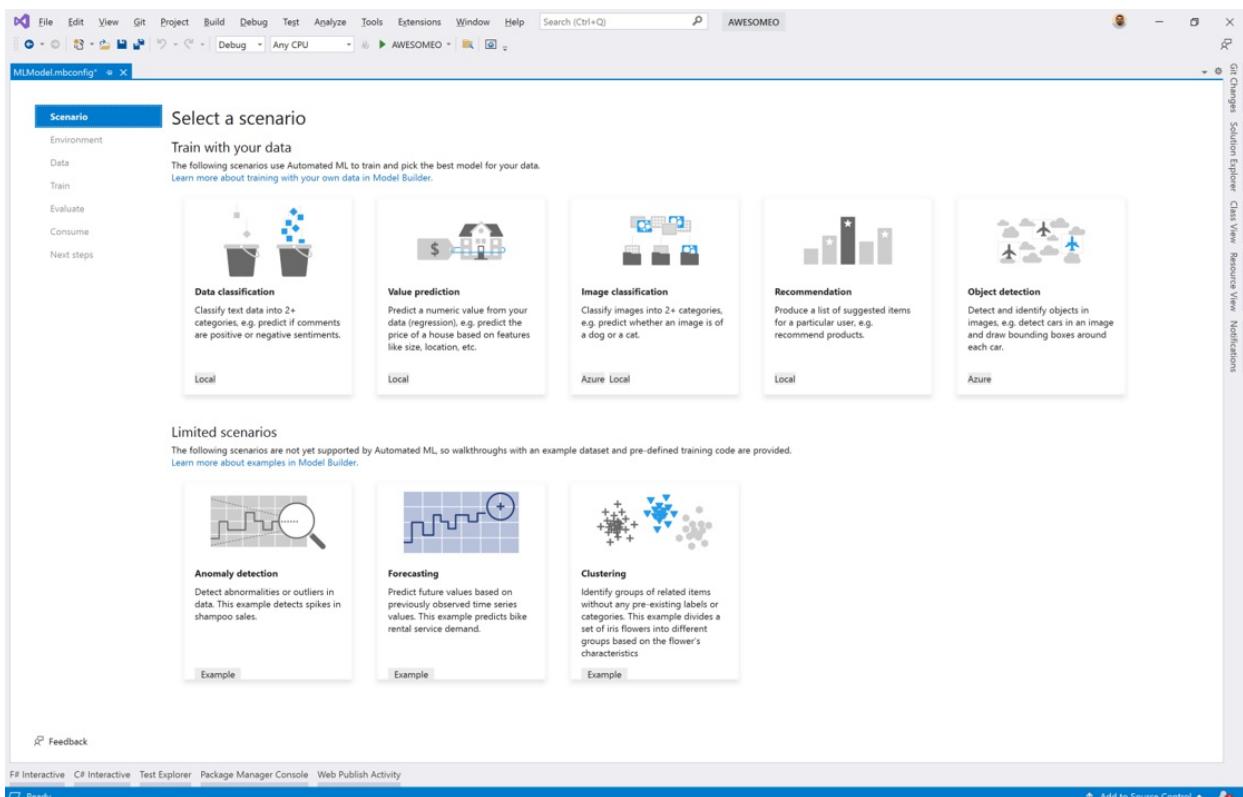
1. Download the subset of the [EuroSAT dataset](#) and save it anywhere on your computer.
2. Unzip it.

Create a Model Builder config file

When first adding Model Builder to the solution it will prompt you to create an `mbconfig` file. The `mbconfig` file keeps track of everything you do in Model Builder to allow you to reopen the session.

1. In Solution Explorer, right-click the **LandUse** project, and select **Add > Machine Learning Model....**
2. In the dialog, name the Model Builder project **LandUse**, and click **Add**.

Choose a scenario



To train your model, you need to select from the list of available machine learning scenarios provided by Model Builder.

For this sample, the task is image classification. In the scenario step of the Model Builder tool, select the **Image Classification** scenario.

Select an environment

Model Builder can run the training on different environments depending on the scenario that was selected.

Select Azure as your environment and click the **Set up workspace** button.

Create experiment in Azure

An Azure Machine Learning experiment is a resource that needs to be created before running Model Builder training on Azure.

The experiment encapsulates the configuration and results for one or more machine learning training runs. Experiments belong to a specific workspace. The first time an experiment is created, its name is registered in the workspace. Any subsequent runs - if the same experiment name is used - are logged as part of the same experiment. Otherwise, a new experiment is created.

In the Create New Experiment dialog, choose your subscription from the **Subscription** dropdown.

Create workspace

A workspace is an Azure Machine Learning resource that provides a central place for all Azure Machine Learning resources and artifacts created as part of a training run.

1. In the Create New Experiment dialog, select the **New** link next to the **Machine Learning Workspace name** dropdown.
2. In the Create A New Workspace dialog, type "landuse-wkspc" in the **Machine Learning Workspace name** text box.
3. Choose **East US** from the **Regions** dropdown. A region is the geographic location of the data center where your workspace and resources are deployed to. It is recommended that you choose a location

close to where you or your customers are.

4. Select the **New** link next to the **Resource Groups** dropdown.
 - a. In the Create New Resource Group dialog, type "landuse-rg" in the **Resource Group name** text box.
 - b. Select **OK**.
5. Choose your newly created resource group from the **Resource Groups** dropdown.
6. Select **Create**.

The provisioning process takes a few minutes. A request is made to Azure to provision the following cloud resources:

- Azure Machine Learning workspace
- Azure Storage Account
- Azure Application Insights
- Azure Container Registry
- Azure Key Vault

7. Once the provisioning process is complete, choose your newly created workspace from the **Machine Learning Workspace name** dropdown in the Create New Experiment dialog.

Create compute

An Azure Machine Learning compute is a cloud-based Linux VM used for training.

1. In the Create New Experiment dialog, select the **New** link next to the **Compute name** dropdown.
2. In the Create New Compute dialog, type "landuse-cpt" in the **Compute name** text box.
3. Choose **Standard_NC24** from the **Compute size** dropdown. Model Builder uses GPU-optimized compute types. Visit the [NC-series Linux VM documentation](#) for more details on GPU optimized compute types.
4. Select **Create**. The compute resources may take a few minutes to provision.
5. Once the provisioning process is complete, choose your newly created workspace from the **Compute name** dropdown in the Create New Experiment dialog.
6. Select the **Next step** button to load in the data.

Load the data

1. In the data step of the Model Builder tool, select the button next to the **Select a folder** text box.
2. Use File Explorer to browse and select the unzipped directory containing the images.
3. Select the **Next step** button to move to the next step in the Model Builder tool.

Train the model

Training on Azure is only available for the Model Builder image classification scenario. The algorithm used to train these models is a Deep Neural Network based on the ResNet50 architecture. During the model training process, Model Builder trains separate models using ResNet50 algorithm and settings to find the best performing model for your dataset.

Start training

Once you've configured your workspace and compute type, it's time to finish creating the experiment and start training.

1. Click the **Start Training** button.

The training process takes some time and the amount of time may vary depending on the size of compute selected as well as amount of data. The first time a model is trained, you can expect a slightly longer training time because resources have to be provisioned. You can track the progress of your runs

by selecting the **Monitor current run in Azure portal** link in Visual Studio.

Throughout the training process, progress data is displayed in the Progress section of the train step.

- Status displays the completion status of the training process.
- Best accuracy displays the accuracy of the best performing model found by Model Builder so far. Higher accuracy means the model predicted more correctly on test data.
- Algorithm displays the name of the best performing algorithm performed found by Model Builder so far.

2. Once training is complete, select the **Next step** button to move to evaluate the model.

Evaluate the model

The result of the training step is one model that had the best performance. In the evaluate step of the Model Builder tool, the **Details** tab in the output section, will contain the algorithm used by the best performing model in the **Algorithm** entry along with metrics in the **Accuracy** entry in the **Best model** details.

If you're not satisfied with your accuracy metrics, some easy ways to try and improve model accuracy are to use more data or augment the existing data. Otherwise, select the **Next step** button to move to the final step in the Model Builder tool.

(Optional) Consume the model

This step will have project templates that you can use to consume the model. This step is optional and you can choose the method that best suits your needs on how to serve the model.

- Console App
- Web API

Console App

When adding a console app to your solution, you will be prompted to name the project.

1. Name the console project **LandUse_Console**.
2. Click **Add to solution** to add the project to your current solution.
3. Run the application.

The output generated by the program should look similar to the snippet below:

```
Predicted Label value: AnnualCrop

Predicted Label scores: [0.9941197, 3.3146807E-06, 4.4344174E-06, 0.000101028825, 7.763133E-06, 0.0015898133, 0.0040994748, 1.6308518E-06, 6.265567E-05, 1.0236401E-05]
```

Web API

When adding a web API to your solution, you will be prompted to name the project.

1. Name the Web API project **LandUse_API**.
2. Click **Add to solution** to add the project to your current solution.
3. Run the application.
4. Open PowerShell and enter the following code where PORT is the port your application is listening on.

```
$body = @{
    ImageSource = <Image location on your local machine>
}

Invoke-RestMethod "https://localhost:<PORT>/predict" -Method Post -Body ($body | ConvertTo-Json) -
ContentType "application/json"
```

5. If successful, the output should look similar to the text below.

```
output1                                prediction score
-----                                -----
{9.508701, -3.1025503, -2.8115153, 0.31449434...} AnnualCrop {0.9941197, 3.3146807E-06, 4.4344174E-06, 0.00010102882...}
```

Clean up resources

If you no longer plan to use the Azure resources you created, delete them. This prevents you from being charged for unutilized resources that are still running.

1. Navigate to the [Azure portal](#) and select **Resource groups** in the portal menu.
2. From the list of resource groups, select the resource group you created. In this case, it's "landuse-rg".
3. Select **Delete resource group**.
4. Type the resource group name, "landuse-rg", into the text box and then select Enter.

Next steps

In this tutorial you learned how to:

- Prepare and understand the data
- Create a Model Builder config file
- Choose a scenario
- Load the data
- Create an experiment in Azure
- Train the model
- Evaluate the model
- Consume the model

Try one of the other Model Builder scenarios:

- [Predict NYC taxi fares](#)
- [Analyze sentiment of website comments in a Razor Pages application](#)
- [Categorize the severity of restaurant health violations](#)

Tutorial: Detect stop signs in images with Model Builder

10/14/2022 • 10 minutes to read • [Edit Online](#)

Learn how to build an object detection model using ML.NET Model Builder and Azure Machine Learning to detect and locate stop signs in images.

In this tutorial, you learn how to:

- Prepare and understand the data
- Create a Model Builder config file
- Choose the scenario
- Choose the training environment
- Load the data
- Train the model
- Evaluate the model
- Use the model for predictions

NOTE

Model Builder is currently in Preview.

Prerequisites

For a list of prerequisites and installation instructions, visit the [Model Builder installation guide](#).

Model Builder object detection overview

Object detection is a computer vision problem. While closely related to image classification, object detection performs image classification at a more granular scale. Object detection both locates *and* categorizes entities within images. Object detection models are commonly trained using deep learning and neural networks. See [Deep learning vs machine learning](#) for more information.

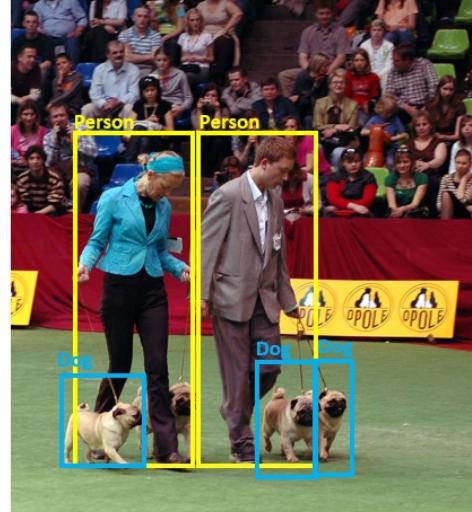
Use object detection when images contain multiple objects of different types.

Image Classification



{Dog}

Object Detection



{Dog, Dog, Dog, Person, Person}

Some use cases for object detection include:

- Self-Driving Cars
- Robotics
- Face Detection
- Workplace Safety
- Object Counting
- Activity Recognition

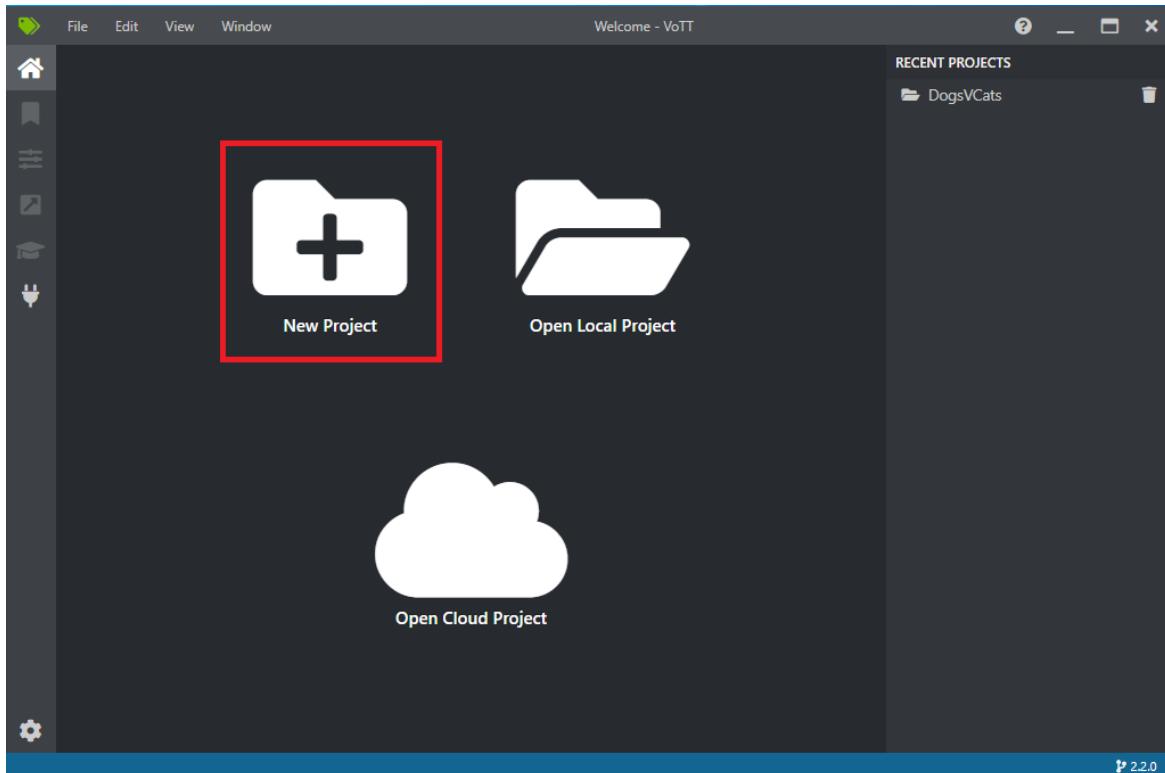
This sample creates a C# .NET Core console application that detects stop signs in images using a machine learning model built with Model Builder. You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) GitHub repository.

Prepare and understand the data

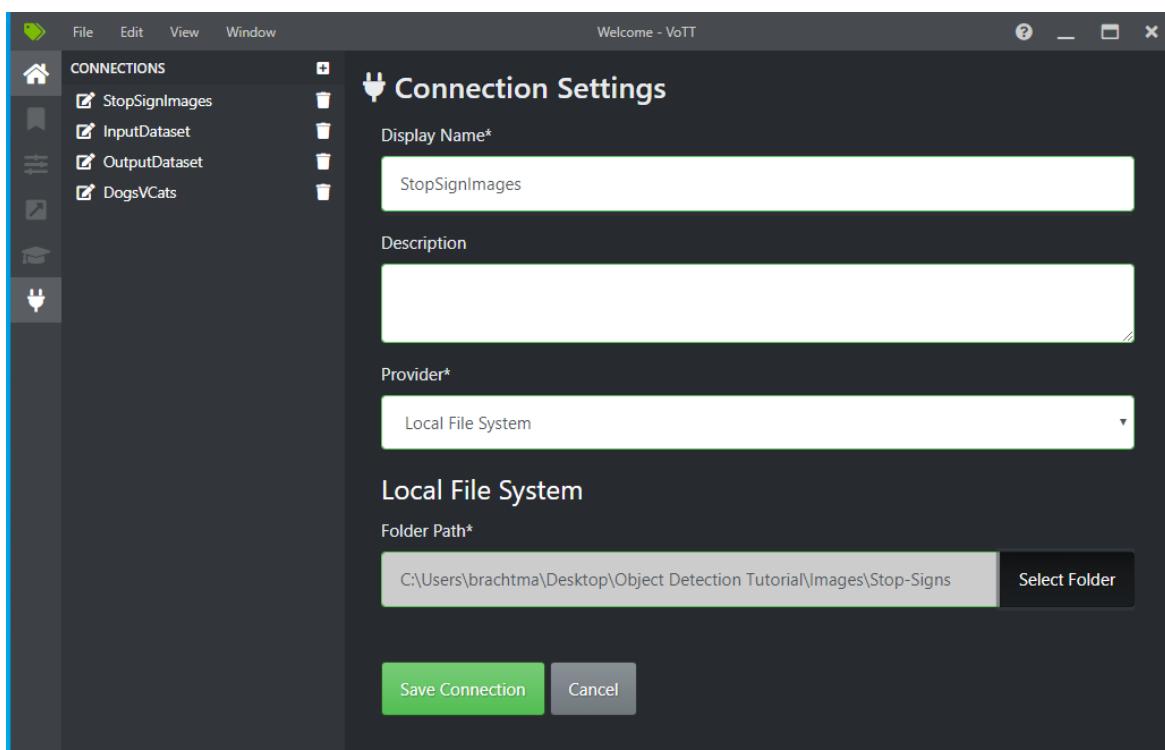
The Stop Sign dataset consists of 50 images downloaded from [Unsplash](#), each of which contain at least one stop sign.

Create a new VoTT project

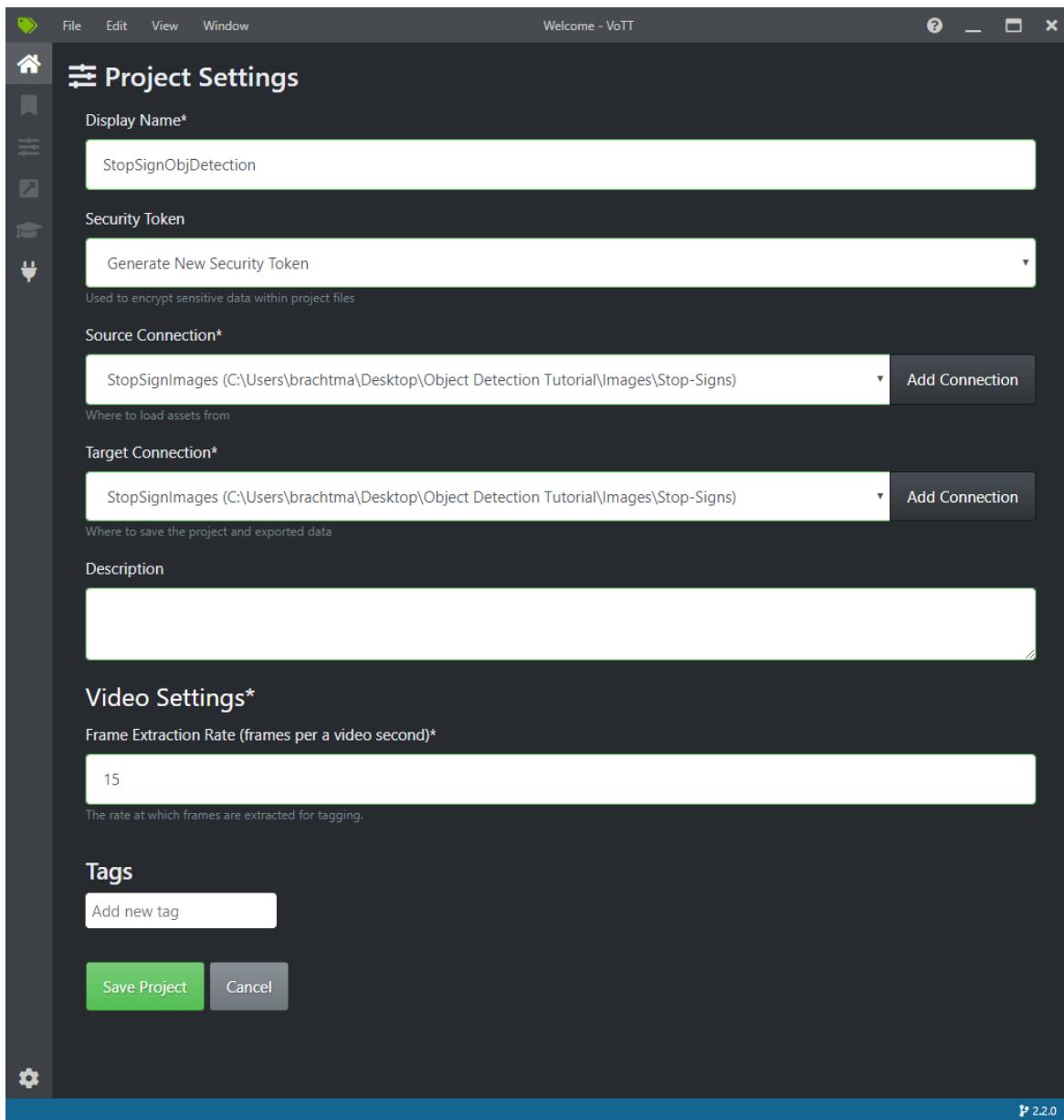
1. [Download the dataset](#) of 50 stop sign images and unzip.
2. [Download VoTT](#) (Visual Object Tagging Tool).
3. Open VoTT and select New Project.



4. In **Project Settings**, change the **Display Name** to "StopSignObjDetection".
5. Change the **Security Token** to *Generate New Security Token*.
6. Next to **Source Connection**, select **Add Connection**.
7. In **Connection Settings**, change the **Display Name** for the source connection to "StopSignImages", and select *Local File System* as the **Provider**. For the **Folder Path**, select the *Stop-Signs* folder which contains the 50 training images, and then select **Save Connection**.



8. In **Project Settings**, change the **Source Connection** to *StopSignImages* (the connection you just created).
9. Change the **Target Connection** to *StopSignImages* as well. Your **Project Settings** should now look similar to this screenshot:

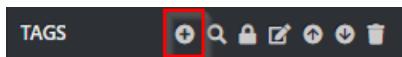


10. Select Save Project.

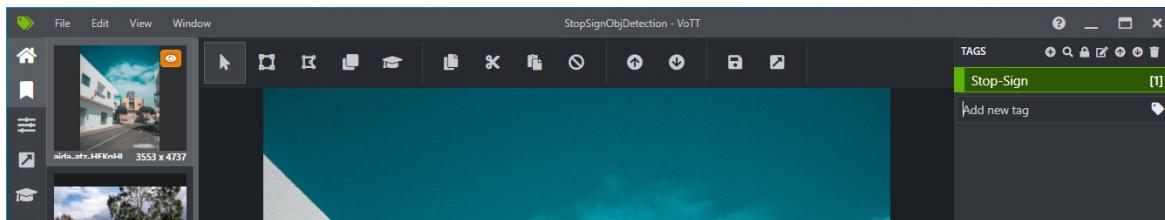
Add tag and label images

You should now see a window with preview images of all the training images on the left, a preview of the selected image in the middle, and a **Tags** column on the right. This screen is the **Tags editor**.

1. Select the first (plus-shaped) icon in the **Tags** toolbar to add a new tag.

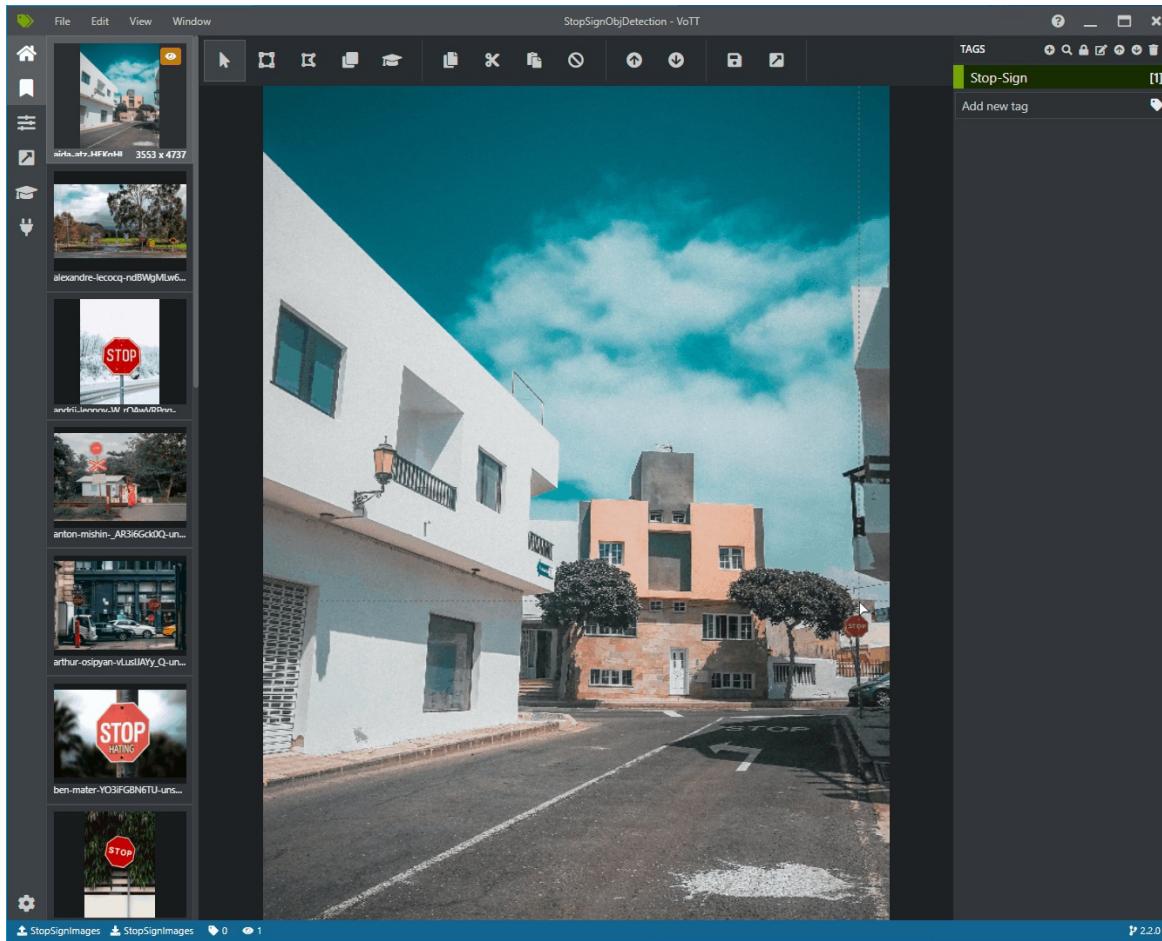


2. Name the tag "Stop-Sign" and hit Enter on your keyboard.



3. Click and drag to draw a rectangle around each stop sign in the image. If the cursor does not let you draw a rectangle, try selecting the **Draw Rectangle** tool from the toolbar on the top, or use the keyboard shortcut R.

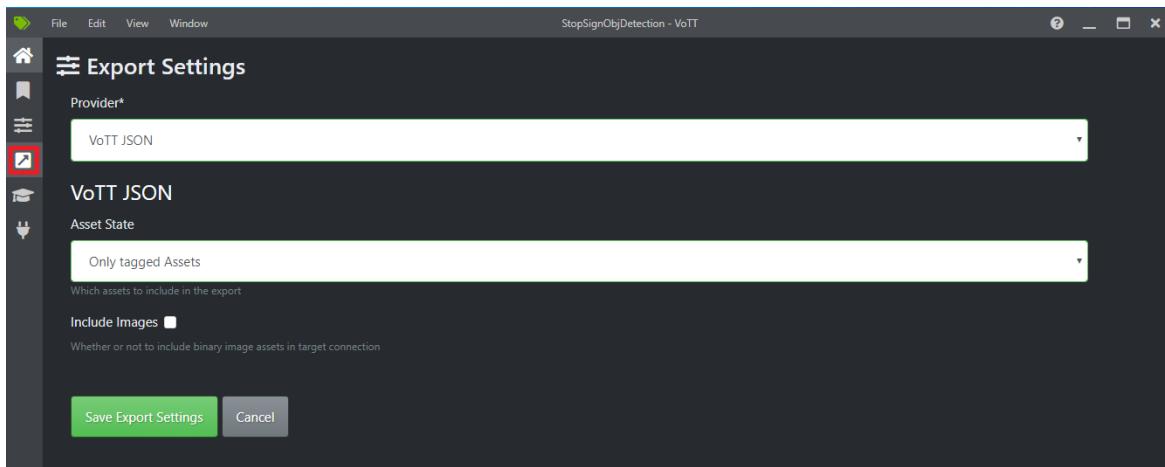
4. After drawing your rectangle, select the **Stop-Sign** tag that you created in the previous steps to add the tag to the bounding box.
5. Click on the preview image for the next image in the dataset and repeat this process.
6. Continue steps 3-4 for every stop sign in every image.



Export your VoTT JSON

Once you have labeled all of your training images, you can export the file that will be used by Model Builder for training.

1. Select the fourth icon in the left toolbar (the one with the diagonal arrow in a box) to go to the **Export Settings**.
2. Leave the **Provider** as *VoTT JSON*.
3. Change the **Asset State** to *Only tagged Assets*.
4. Uncheck **Include Images**. If you include the images, then the training images will be copied to the export folder that is generated, which is not necessary.
5. Select **Save Export Settings**.



6. Go back to the **Tags editor** (the second icon in the left toolbar shaped like a ribbon). In the top toolbar, select the **Export Project** icon (the last icon shaped like an arrow in a box), or use the keyboard shortcut **Ctrl+E**.



This export will create a new folder called *vott-json-export* in your *Stop-Sign-Images* folder and will generate a JSON file named *StopSignObjDetection-export* in that new folder. You will use this JSON file in the next steps for training an object detection model in Model Builder.

Create a console application

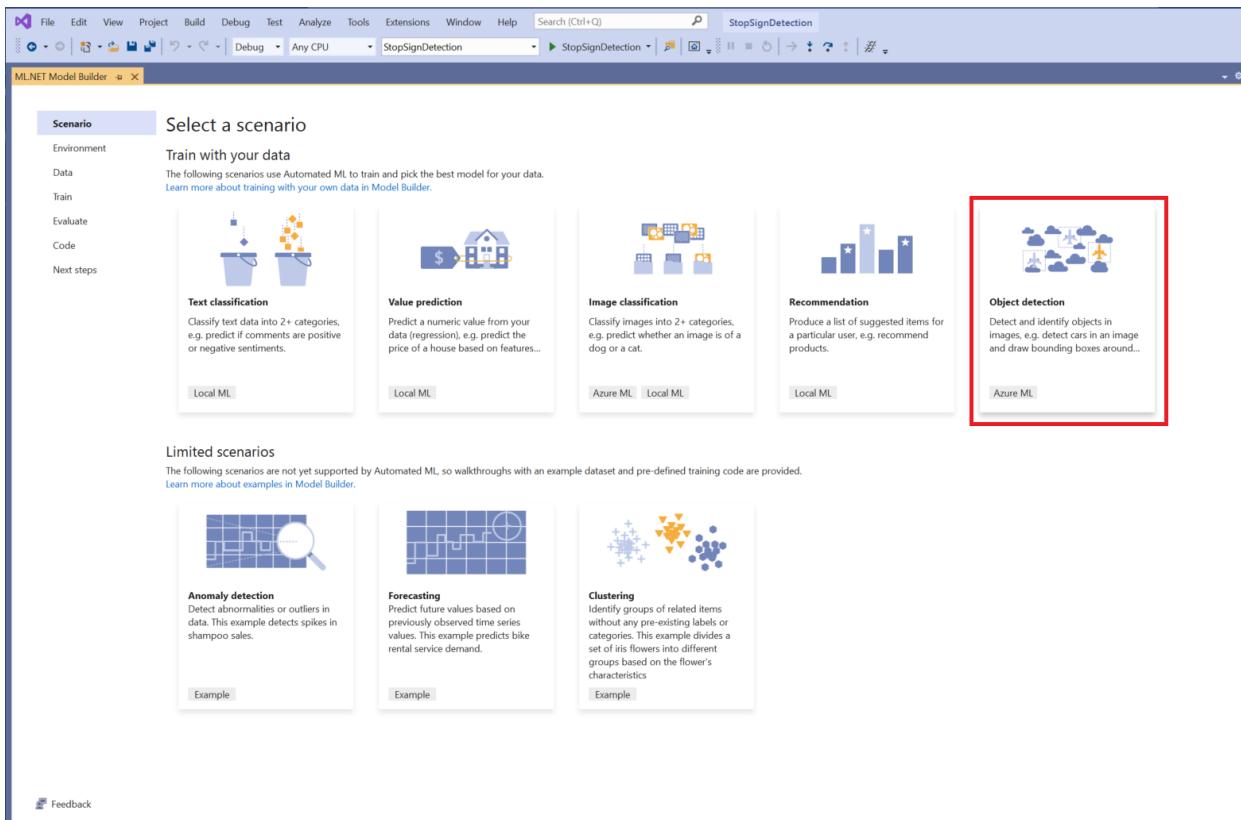
In Visual Studio, create a C# .NET Core **console application** called *StopSignDetection*.

Create an `mbconfig` file

1. In **Solution Explorer**, right-click the *StopSignDetection* project, and select **Add > Machine Learning Model...** to open the Model Builder UI.
2. In the dialog, name the Model Builder project **StopSignDetection**, and click **Add**.

Choose a scenario

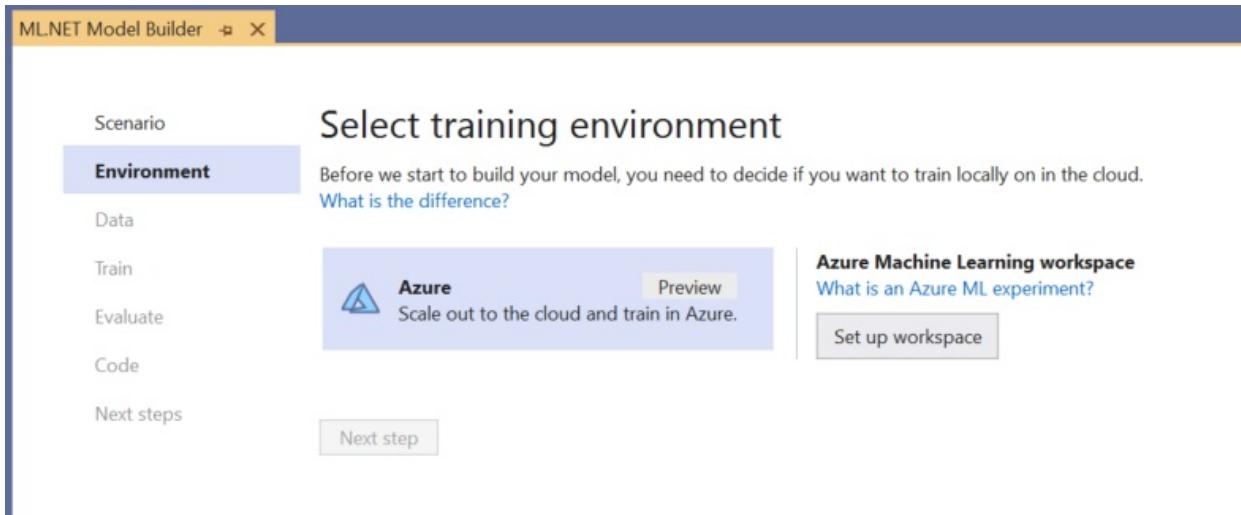
For this sample, the scenario is object detection. In the **Scenario** step of Model Builder, select the **Object Detection** scenario.



If you don't see *Object Detection* in the list of scenarios, you may need to [update your version of Model Builder](#).

Choose the training environment

Currently, Model Builder supports training object detection models with Azure Machine Learning only, so the Azure training environment is selected by default.



To train a model using Azure ML, you must create an Azure ML experiment from Model Builder.

An **Azure ML experiment** is a resource that encapsulates the configuration and results for one or more machine learning training runs.

To create an Azure ML experiment, you first need to configure your environment in Azure. An experiment needs the following to run:

- An Azure subscription
- A **workspace**: an Azure ML resource that provides a central place for all Azure ML resources and artifacts

created as part of a training run.

- A **compute**: an Azure Machine Learning compute is a cloud-based Linux VM used for training. Learn more about [compute types supported by Model Builder](#).

Set up an Azure ML workspace

To configure your environment:

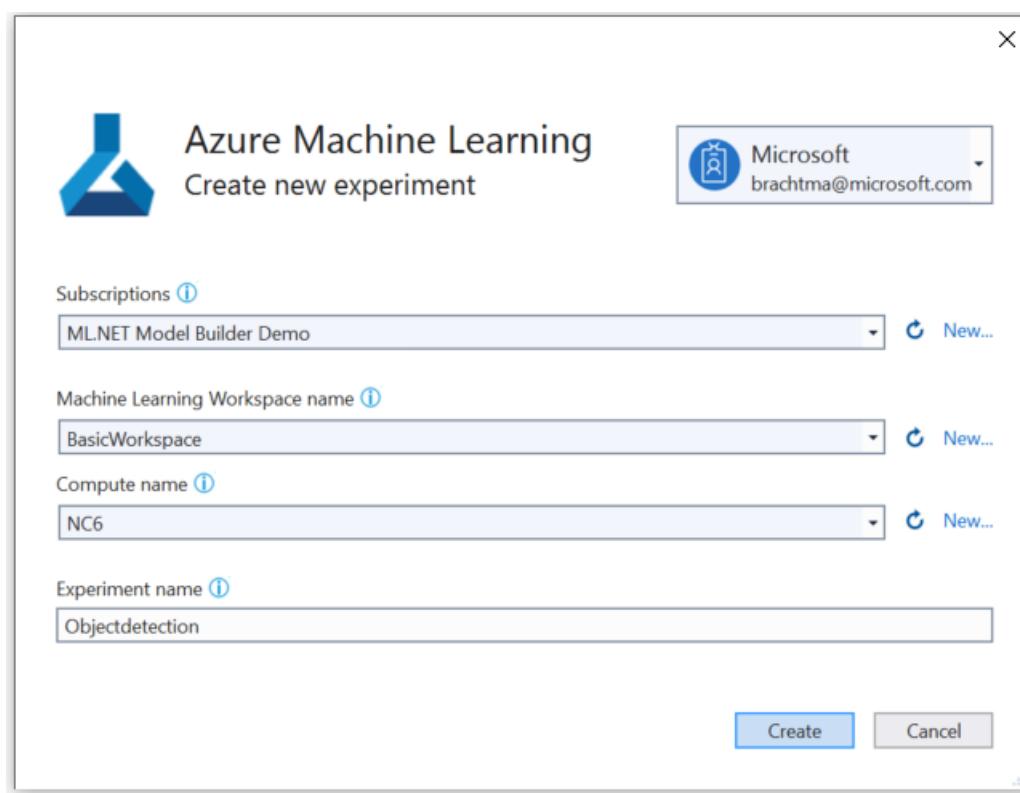
1. Select the **Set up workspace** button.
2. In the **Create new experiment** dialog, select your Azure subscription.
3. Select an existing workspace or create a new Azure ML workspace.

When you create a new workspace, the following resources are provisioned:

- Azure Machine Learning workspace
- Azure Storage
- Azure Application Insights
- Azure Container Registry
- Azure Key Vault

As a result, this process may take a few minutes.

4. Select an existing compute or create a new Azure ML compute. This process may take a few minutes.
5. Leave the default experiment name and select **Create**.



The first experiment is created, and the experiment name is registered in the workspace. Any subsequent runs (if the same experiment name is used) are logged as part of the same experiment. Otherwise, a new experiment is created.

If you're satisfied with your configuration, select the **Next step** button in Model Builder to move to the **Data** step.

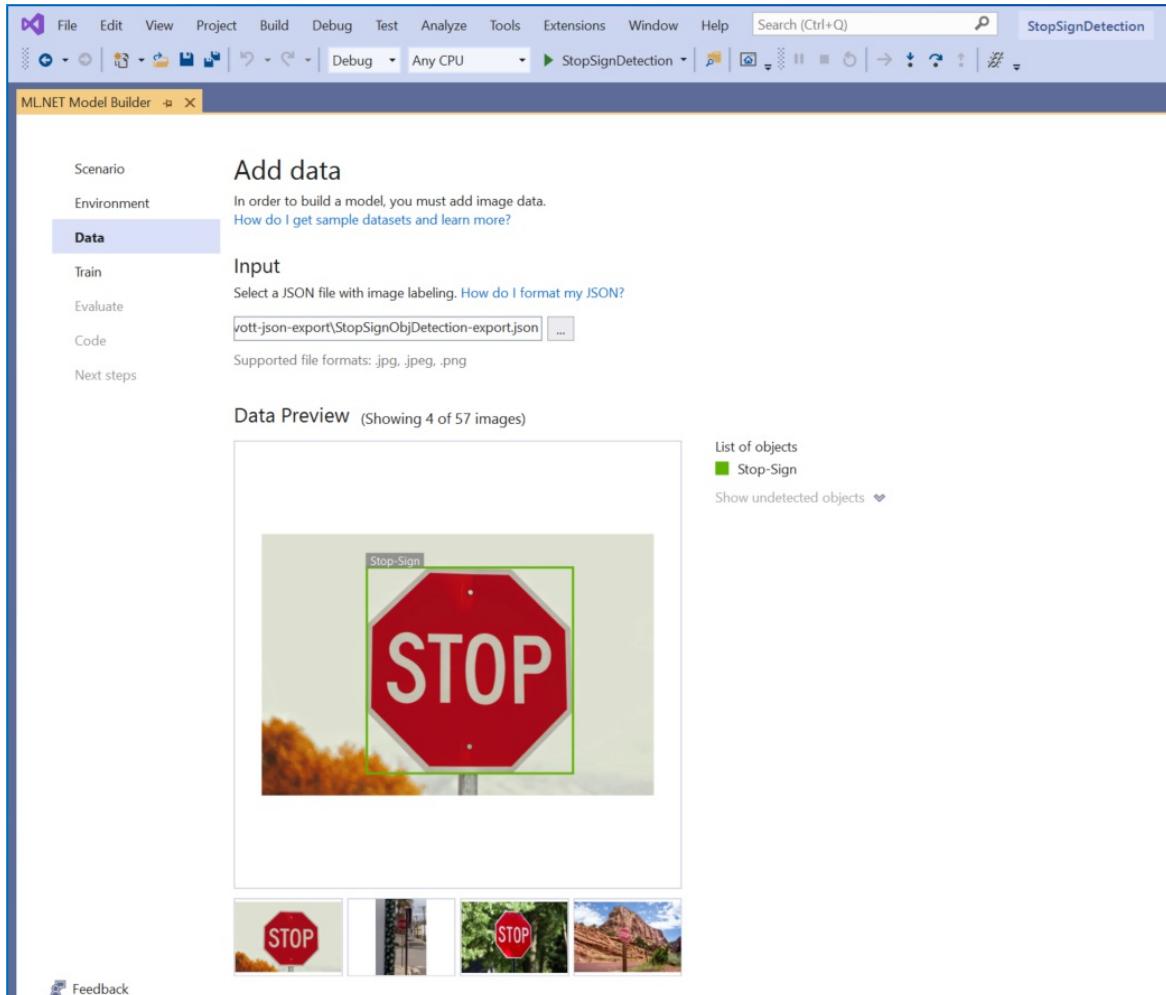
Load the data

In the **Data** step of Model Builder, you will select your training dataset.

IMPORTANT

Model Builder currently only accepts the format of JSON [generated by VoTT](#).

1. Select the button inside **Input** section and use the File Explorer to find the `StopSignObjDetection-export.json` which should be located in the *Stop-Signs/vott-json-export* directory.



2. If your data looks correct in the **Data Preview**, select **Next step** to move on to the **Train** step.

Train the model

The next step is to train your model.

In the Model Builder **Train** screen, select the **Start training** button.

At this point, your data is uploaded to Azure Storage and the training process begins in Azure ML.

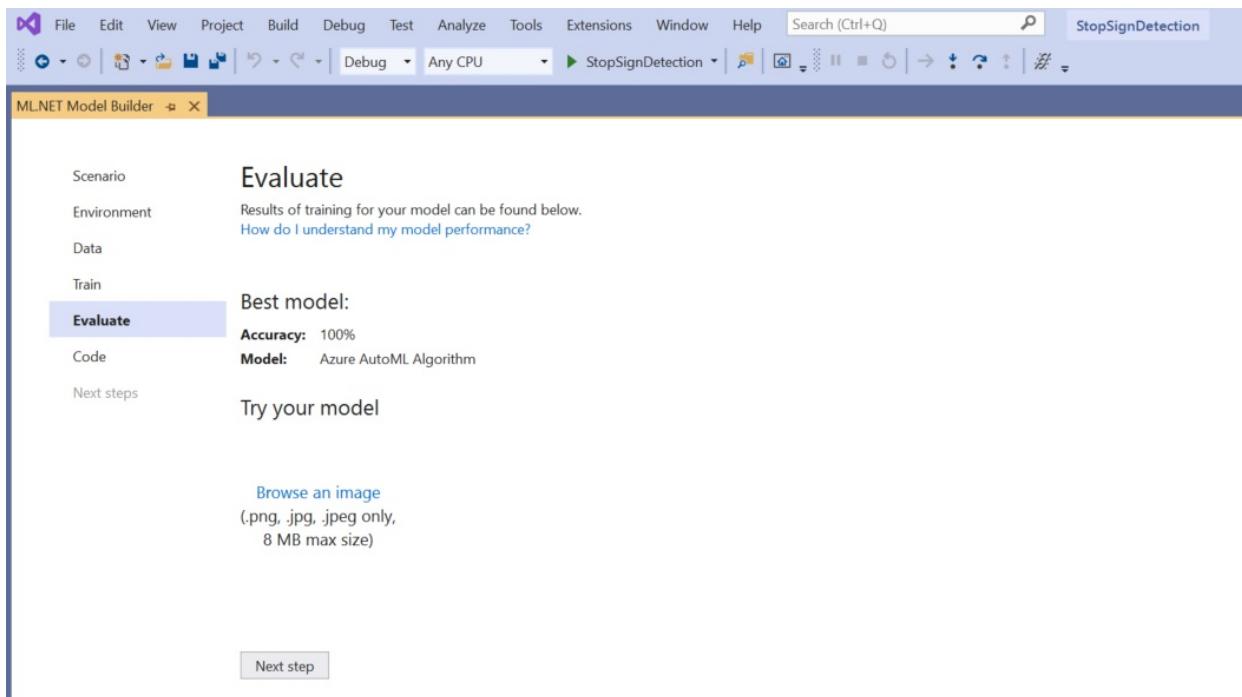
The training process takes some time, and the amount of time may vary depending on the size of compute selected as well as the amount of data. The first time a model is trained in Azure, you can expect a slightly longer training time because resources have to be provisioned. For this sample of 50 images, training took about 16 minutes.

You can track the progress of your runs in the Azure Machine Learning portal by selecting the **Monitor current run in Azure portal** link in Visual Studio.

Once training is complete, select the **Next step** button to move on to the **Evaluate** step.

Evaluate the model

In the Evaluate screen, you get an overview of the results from the training process, including the model accuracy.



In this case, the accuracy says 100%, which means that the model is more than likely *overfit* due to too few images in the dataset.

You can use the **Try your model** experience to quickly check whether your model is performing as expected.

Select **Browse an image** and provide a test image, preferably one that the model did not use as part of training.

ML.NET Model Builder

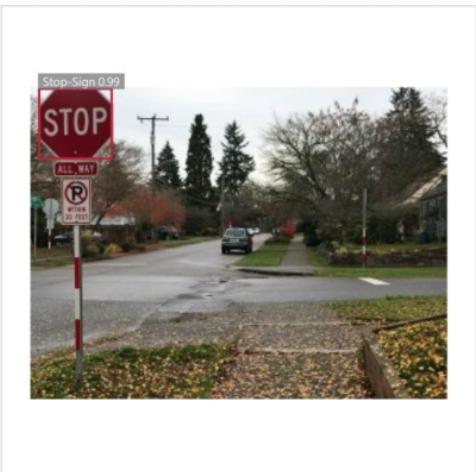
Scenario
Environment
Data
Train
Evaluate
Code
Next steps

Evaluate

Results of training for your model can be found below.
[How do I understand my model performance?](#)

Best model:
Accuracy: 100%
Model: Azure AutoML Algorithm

Try your model



Score threshold ⓘ

0.8

List of objects

Stop-Sign

Show undetected objects ▾

Browse a different image (.png, .jpg, .jpeg)

Next step

The score shown on each detected bounding box indicates the **confidence** of the detected object. For instance, in the screenshot above, the score on the bounding box around the stop sign indicates that the model is 99% sure that the detected object is a stop sign.

The **Score threshold**, which can be increased or decreased with the threshold slider, will add and remove detected objects based on their scores. For instance, if the threshold is .51, then the model will only show objects that have a confidence score of .51 or above. As you increase the threshold, you will see less detected objects, and as you decrease the threshold, you will see more detected objects.

If you're not satisfied with your accuracy metrics, one easy way to try to improve model accuracy is to use more data. Otherwise, select the **Next step** link to move on to the **Consume** step in Model Builder.

(Optional) Consume the model

This step will have project templates that you can use to consume the model. This step is optional and you can choose the method that best suits your needs on how to serve the model.

- Console App
- Web API

Console App

When adding a console app to your solution, you will be prompted to name the project.

1. Name the console project **StopSignDetection_Console**.
2. Click **Add to solution** to add the project to your current solution.
3. Run the application.

The output generated by the program should look similar to the snippet below:

```
Predicted Boxes:
```

```
Top: 73.225296, Left: 256.89764, Right: 533.8884, Bottom: 484.24243, Label: stop-sign, Score: 0.9970765
```

Web API

When adding a web API to your solution, you will be prompted to name the project.

1. Name the Web API project **StopSignDetection_API**.
2. Click **Add to solution** to add the project to your current solution.
3. Run the application.
4. Open PowerShell and enter the following code where PORT is the port your application is listening on.

```
$body = @{
    ImageSource = <Image location on your local machine>
}

Invoke-RestMethod "https://localhost:<PORT>/predict" -Method Post -Body ($body | ConvertTo-Json) -ContentType "application/json"
```

5. If successful, the output should look similar to the text below.

boxes	labels	scores	boundingBoxes
-----	-----	-----	-----
{339.97797, 154.43184, 472.6338, 245.0796} {1}	{0.99273646} {}		

- The `boxes` column gives the bounding box coordinates of the object that was detected. The values here belong to the left, top, right, and bottom coordinates respectively.
- The `labels` are the index of the predicted labels. In this case, the value 1 is a stop sign.
- The `scores` defines how confident the model is that the bounding box belongs to that label.

NOTE

(Optional) The bounding box coordinates are normalized for a width of 800 pixels and a height of 600 pixels. To scale the bounding box coordinates for your image in further post-processing, you need to:

1. Multiply the top and bottom coordinates by the original image height, and multiply the left and right coordinates by the original image width.
2. Divide the top and bottom coordinates by 600, and divide the left and right coordinates by 800.

For example, given the original image dimensions, `actualImageHeight` and `actualImageWidth`, and a `ModelOutput` called `prediction`, the following code snippet shows how to scale the `BoundingBox` coordinates:

```
var top = originalImageHeight * prediction.Top / 600;
var bottom = originalImageHeight * prediction.Bottom / 600;
var left = originalImageWidth * prediction.Left / 800;
var right = originalImageWidth * prediction.Right / 800;
```

An image may have more than one bounding box, so the same process needs to be applied to each of the bounding boxes in the image.

Congratulations! You've successfully built a machine learning model to detect stop signs in images using Model Builder. You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) GitHub repository.

Additional resources

To learn more about topics mentioned in this tutorial, visit the following resources:

- [Model Builder Scenarios](#)
- [Object Detection using ONNX in ML.NET](#)

Train a recommendation model using Model Builder

10/14/2022 • 5 minutes to read • [Edit Online](#)

Learn how to train a recommendation model using Model Builder to recommend movies.

In this tutorial, you:

- Prepare and understand the data
- Create a Model Builder config file
- Choose a scenario
- Load the data
- Train the model
- Evaluate the model
- Consume the model

NOTE

Model Builder is currently in Preview.

Prerequisites

For a list of pre-requisites and installation instructions, visit the [Model Builder installation guide](#).

Create a C# Class Library

Create a C# Class Library called "MovieRecommender".

Prepare and understand the data

There are several ways to approach recommendation problems, such as recommending a list of movies or recommending a list of related products, but in this case you will predict what rating (1-5) a user will give to a particular movie and recommend that movie if it's higher than a defined threshold (the higher the rating, the higher the likelihood of a user liking a particular movie).

Right click on [recommendation-ratings-train.csv](#) and select "Save Link (or Target) As..."

Each row in the dataset contains information regarding a movie rating.

USERID	MOVIEID	RATING	TIMESTAMP
1	1	4	964982703

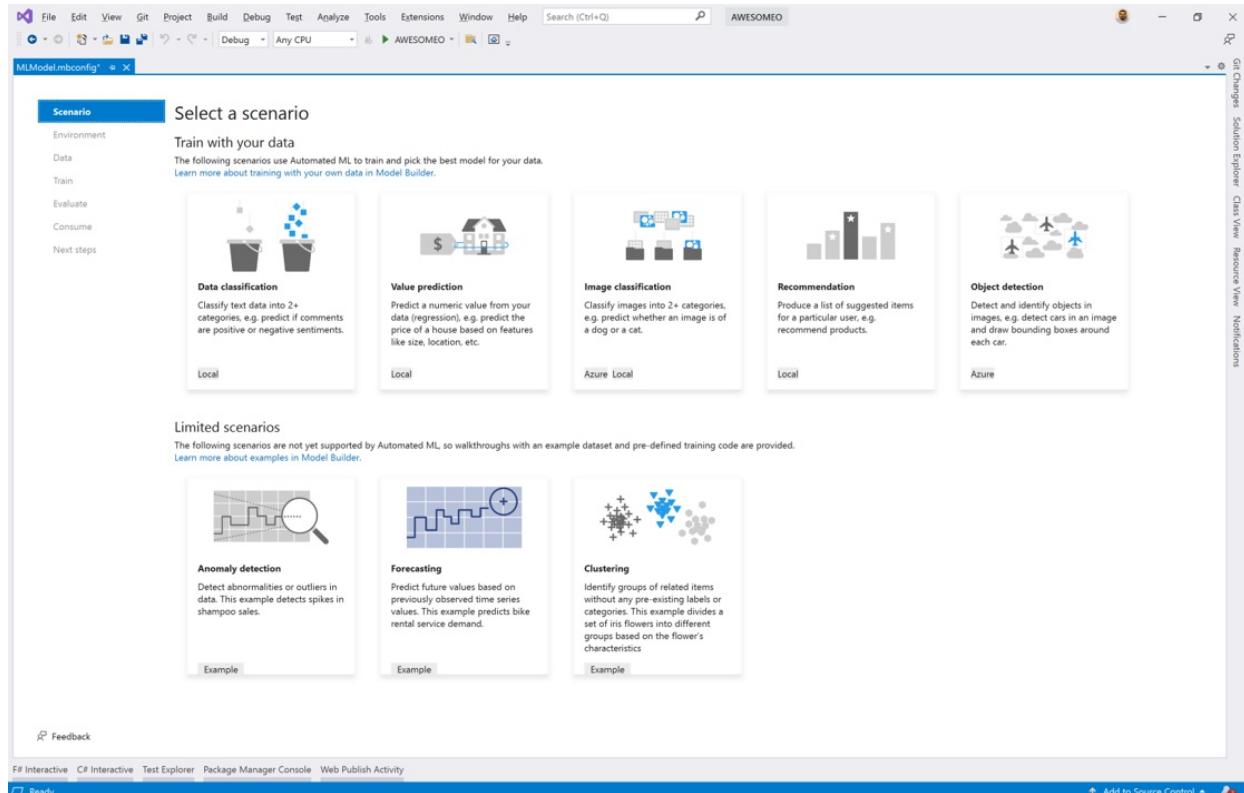
- **userId**: The ID of the user
- **movieId**: The ID of the movie
- **rating**: The rating the user made to the movie
- **timestamp**: The timestamp the review was made

Create a Model Builder config file

When first adding Model Builder to the solution it will prompt you to create an `mbconfig` file. The `mbconfig` file keeps track of everything you do in Model Builder to allow you to reopen the session.

1. In Solution Explorer, right-click the **MovieRecommender** project, and select **Add > Machine Learning Model....**
2. In the dialog, name the Model Builder project **MovieRecommender**, and click **Add**.

Choose a scenario



To train your model, you need to select from the list of available machine learning scenarios provided by Model Builder.

For this sample, the task is image classification. In the scenario step of the Model Builder tool, select the **Recommendation** scenario.

Select an environment

Model Builder can run the training on different environments depending on the scenario that was selected.

Select **Local** as your environment and click the **Next step** button.

Load the data

1. In the data step of the Model Builder tool, select the button next to the **Select a folder** text box.
2. Use File Explorer to browse and select the downloaded file - **recommendation-ratings-train.csv**.
3. Select the **Next step** button to move to the next step in the Model Builder tool.
4. Once the data is loaded, in the **Column to predict** dropdown select **Rating**.
5. For the **User column** dropdown select **userId**.
6. For the **Item column** dropdown select **movieId**.

Train the model

The machine learning algorithm used to train the recommendation model is Matrix Factorization. During the

model training process, Model Builder uses different settings for the algorithm to find the best performing model for your dataset.

The time required for the model to train is proportional to the amount of data. Model Builder automatically selects a default value for **Time to train (seconds)** based on the size of your data source.

1. Model Builder sets the value of **Time to train (seconds)** to 60 seconds. Training for a longer period of time allows Model Builder to explore a larger number of algorithms and combination of parameters in search of the best model.
2. Click **Start Training**.

Throughout the training process, progress data is displayed in the **Training results** section of the train step.

- Status displays the completion status of the training process.
- Best quality the R Squared of the best performing model found by Model Builder so far. The lower R Squared means the model predicted more correctly on test data.
- Best algorithm displays the name of the best performing algorithm performed found by Model Builder so far.
- Last algorithm displays the name of the algorithm most recently used by Model Builder to train the model.

Once training is complete the `mbconfig` file will have the generated model called `MovieRecommender.zip` after training and two C# files with it:

- **MovieRecommender.consumption.cs**: This file has a public method `Predict` that loads the model and creates a `PredictionEngine` to make predictions. `PredictionEngine` is a convenience API for making predictions on a single data instance.
- **MovieRecommender.training.cs**: This file consists of the training pipeline that Model Builder came up with to build the best model including any hyperparameters that it used.

Click the **Next step** button to navigate to the evaluate step.

Evaluate the model

The result of the training step will be one model which had the best performance. In the evaluate step of the Model Builder tool, in the **Best model** section, will contain the algorithm used by the best performing model in the *Model* entry along with metrics for that model in *RSquared*.

Additionally, in the **Output** window of Visual Studio, there will be a summary table containing top models and their metrics.

In this section you can also test your model by performing a single prediction. It provides you with text boxes to input values for each of your feature columns and you can select the **Predict** button to get a prediction using the best model. By default this will be filled in by the first row in your dataset.

(Optional) Consume the model

This step will have project templates that you can use to consume the model. This step is optional and you can choose the method that best suits your needs on how to serve the model.

- Console App
- Web API

Console App

When adding a console app to your solution, you will be prompted to name the project.

1. Select **Add to solution** for the console template.
2. Name the console project `MovieRecommender_Console`.

3. Click **Add to solution** to add the project to your current solution.

4. Run the application.

The output generated by the program should look similar to the snippet below:

```
UserId: 1
MovieId: 1
Rating: 4

Predicted Rating: 4.577113
```

Web API

When adding a web API to your solution, you will be prompted to name the project.

1. Select **Add to solution** for the Web API template.

2. Name the Web API project **MovieRecommender_WebApi**.

3. Click *Add to solution** to add the project to your current solution.

4. Run the application.

5. Open PowerShell and enter the following code where PORT is the port your application is listening on.

```
$body = @{
    UserId=1.0
    MovieId=1.0
}

Invoke-RestMethod "https://localhost:<PORT>/predict" -Method Post -Body ($body | ConvertTo-Json) -
ContentType "application/json"
```

6. If successful, the output should look similar to the text below. The **score** output will be the predicted rating for the requested user ID and movie ID.

```
score
-----
4.577113
```

Congratulations! You've successfully built a machine learning model to categorize the risk of health violations using Model Builder. You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) GitHub repository.

Additional resources

To learn more about topics mentioned in this tutorial, visit the following resources:

- [Model Builder Scenarios](#)
- [Movie Recommendation in the ML.NET API](#)

ML.NET tutorials

10/14/2022 • 2 minutes to read • [Edit Online](#)

The following tutorials enable you to understand how to use [ML.NET](#) to build custom machine learning solutions and integrate them into your .NET applications:

- [Sentiment analysis](#): demonstrates how to apply a **binary classification** task using ML.NET.
- [GitHub issue classification](#): demonstrates how to apply a **multiclass classification** task using ML.NET.
- [Price predictor](#): demonstrates how to apply a **regression** task using ML.NET.
- [Iris clustering](#): demonstrates how to apply a **clustering** task using ML.NET.
- [Recommendation](#): generate movie **recommendations** based on previous user ratings
- [Image classification](#): demonstrates how to retrain an existing TensorFlow model to create a custom image classifier using ML.NET.
- [Anomaly detection](#): demonstrates how to build an anomaly detection application for product sales data analysis.
- [Detect objects in images](#): demonstrates how to detect objects in images using a pre-trained ONNX model.
- [Classify sentiment of movie reviews](#): learn to load a pre-trained TensorFlow model to classify the sentiment of movie reviews.

Next Steps

For more examples that use ML.NET, check out the [dotnet/machinelearning-samples](#) GitHub repository.

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

10/14/2022 • 12 minutes to read • [Edit Online](#)

This tutorial 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 2022.

In this tutorial, 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

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Prerequisites

- [Visual Studio 2022](#).
- [UCI Sentiment Labeled Sentences dataset](#) (ZIP file)

Create a console application

1. Create a **C# Console Application** called "SentimentAnalysis". Click the **Next** button.
2. Choose **.NET 6** as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to save your data set files.
4. Install the **Microsoft.ML NuGet Package**:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

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

NOTE

The datasets for this tutorial are from the 'From Group to Individual Labels using Deep Features', Kotzias et. al., KDD 2015, and hosted at the UCI Machine Learning Repository - Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: University of California, School of Information and Computer Science.

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_labelled.txt` file and select **Properties**. Under **Advanced**, 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 Microsoft.ML;
using Microsoft.ML.Data;
using SentimentAnalysis;
using static Microsoft.ML.DataOperationsCatalog;
```

2. Add the following code to the line right below the `using` statements, to create a field to hold the recently downloaded dataset file path:

```
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:

- In **Solution Explorer**, right-click the project, and then select **Add > New Item**.
- 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 tutorial, 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 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:

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

3. Create a `LoadData()` method at the bottom of the `Program.cs` file using the following code:

```
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 `testFraction` parameter. 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 `BuildAndTrainModel` method below the call to the `LoadData` 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, below the `LoadData()` method, using the following code:

```
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:

```
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 below 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:

```
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 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](#).

NOTE

`PredictionEnginePool` service extension is currently in preview.

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:

```
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 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 tutorial 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 tutorial at the [dotnet/samples](#) repository.

Next steps

In this tutorial, you learned 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

Advance to the next tutorial to learn more

[Issue Classification](#)

Tutorial: Categorize support issues using multiclass classification with ML.NET

10/14/2022 • 12 minutes to read • [Edit Online](#)

This sample tutorial illustrates using ML.NET to create a GitHub issue classifier to train a model that classifies and predicts the Area label for a GitHub issue via a .NET Core console application using C# in Visual Studio.

In this tutorial, you learn how to:

- Prepare your data
- Transform the data
- Train the model
- Evaluate the model
- Predict with the trained model
- Deploy and Predict with a loaded model

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Prerequisites

- [Visual Studio 2022](#) with the ".NET Desktop Development" workload installed.
- The [GitHub issues tab separated file \(issues_train.tsv\)](#).
- The [GitHub issues test tab separated file \(issues_test.tsv\)](#).

Create a console application

Create a project

1. Create a C# Console Application called "GitHubIssueClassification". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to save your data set files:

In **Solution Explorer**, right-click on your project and select **Add > New Folder**. Type "Data" and hit Enter.

4. Create a directory named *Models* in your project to save your model:
In **Solution Explorer**, right-click on your project and select **Add > New Folder**. Type "Models" and hit Enter.
5. Install the **Microsoft.ML** NuGet Package:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

In **Solution Explorer**, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.ML** 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.

Prepare your data

1. Download the [issues_train.tsv](#) and the [issues_test.tsv](#) data sets and save them to the *Data* folder previously created. The first dataset trains the machine learning model and the second can be used to evaluate how accurate your model is.
2. In Solution Explorer, right-click each of the *.tsv files and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

Create classes and define paths

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

```
using Microsoft.ML;
using GitHubIssueClassification;
```

Create three global fields to hold the paths to the recently downloaded files, and global variables for the `MLContext`, `DataView`, and `PredictionEngine`:

- `_trainDataPath` has the path to the dataset used to train the model.
- `_testDataPath` has the path to the dataset used to evaluate the model.
- `_modelPath` has the path where the trained model is saved.
- `_mlContext` is the `MLContext` that provides processing context.
- `_trainingDataView` is the `IDataView` used to process the training dataset.
- `_predEngine` is the `PredictionEngine<TSrc,TDst>` used for single predictions.

Add the following code to the line directly below the using statements to specify those paths and the other variables:

```
string _appPath = Path.GetDirectoryName(Environment.GetCommandLineArgs()[0]);
string _trainDataPath = Path.Combine(_appPath, "..", "..", "Data", "issues_train.tsv");
string _testDataPath = Path.Combine(_appPath, "..", "..", "..", "Data", "issues_test.tsv");
string _modelPath = Path.Combine(_appPath, "..", "..", "..", "Models", "model.zip");

MLContext _mlContext;
PredictionEngine<GitHubIssue, IssuePrediction> _predEngine;
ITransformer _trainedModel;
IDataView _trainingDataView;
```

Create some classes for your input data and predictions. Add a new class to your project:

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 *GitHubIssueData.cs*. Then, select the **Add** button.

The *GitHubIssueData.cs* file opens in the code editor. Add the following `using` statement to the top of *GitHubIssueData.cs*:

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

Remove the existing class definition and add the following code, which has two classes `GitHubIssue` and `IssuePrediction`, to the *GitHubIssueData.cs* file:

```

public class GitHubIssue
{
    [LoadColumn(0)]
    public string ID { get; set; }
    [LoadColumn(1)]
    public string Area { get; set; }
    [LoadColumn(2)]
    public string Title { get; set; }
    [LoadColumn(3)]
    public string Description { get; set; }
}

public class IssuePrediction
{
    [ColumnName("PredictedLabel")]
    public string Area;
}

```

The `label` is the column you want to predict. The identified `Features` are the inputs you give the model to predict the Label.

Use the [LoadColumnAttribute](#) to specify the indices of the source columns in the data set.

`GitHubIssue` is the input dataset class and has the following [String](#) fields:

- the first column `ID` (GitHub Issue ID)
- the second column `Area` (the prediction for training)
- the third column `Title` (GitHub issue title) is the first `feature` used for predicting the `Area`
- the fourth column `Description` is the second `feature` used for predicting the `Area`

`IssuePrediction` is the class used for prediction after the model has been trained. It has a single `string` (`Area`) and a `PredictedLabel` `ColumnName` attribute. The `PredictedLabel` is used during prediction and evaluation. For evaluation, an input with training data, the predicted values, and the model are used.

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

Initialize the `_mlContext` global variable with a new instance of `MLContext` with a random seed (`seed: 0`) for repeatable/deterministic results across multiple trainings. Replace the `Console.WriteLine("Hello World!")` line with the following code:

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

Load the 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).

To initialize and load the `_trainingDataView` global variable in order to use it for the pipeline, add the following code after the `mlContext` initialization:

```
_trainingDataView = _mlContext.Data.LoadFromTextFile<GitHubIssue>(_trainDataPath, hasHeader: true);
```

The `LoadFromTextFile()` defines the data schema and reads in the file. It takes in the data path variables and

returns an `IDataView`.

Add the following after calling the `LoadFromTextFile()` method:

```
var pipeline = ProcessData();
```

The `ProcessData` method executes the following tasks:

- Extracts and transforms the data.
- Returns the processing pipeline.

Create the `ProcessData` method at the bottom of the `Program.cs` file using the following code:

```
IEstimator<ITransformer> ProcessData()
{
}
```

Extract Features and transform the data

As you want to predict the Area GitHub label for a `GitHubIssue`, use the `MapValueToKey()` method to transform the `Area` column into a numeric key type `Label` column (a format accepted by classification algorithms) and add it as a new dataset column:

```
var pipeline = _mlContext.Transforms.Conversion.MapValueToKey(inputColumnName: "Area", outputColumnName: "Label")
```

Next, call `_mlContext.Transforms.Text.FeaturizeText`, which transforms the text (`Title` and `Description`) columns into a numeric vector for each called `TitleFeaturized` and `DescriptionFeaturized`. Append the featurization for both columns to the pipeline with the following code:

```
.Append(_mlContext.Transforms.Text.FeaturizeText(inputColumnName: "Title", outputColumnName: "TitleFeaturized"))
.Append(_mlContext.Transforms.Text.FeaturizeText(inputColumnName: "Description", outputColumnName: "DescriptionFeaturized"))
```

The last step in data preparation combines all of the feature columns into the `Features` column using the `Concatenate()` method. By default, a learning algorithm processes only features from the `Features` column. Append this transformation to the pipeline with the following code:

```
.Append(_mlContext.Transforms.Concatenate("Features", "TitleFeaturized", "DescriptionFeaturized"))
```

Next, append a `AppendCacheCheckpoint` to cache the `DataView` so when you iterate over the data multiple times using the cache might get better performance, as with the following code:

```
.AppendCacheCheckpoint(_mlContext);
```

WARNING

Use `AppendCacheCheckpoint` for small/medium datasets to lower training time. Do NOT use it (remove `.AppendCacheCheckpoint()`) when handling very large datasets.

Return the pipeline at the end of the `ProcessData` method.

```
return pipeline;
```

This step handles preprocessing/featurization. Using additional components available in ML.NET can enable better results with your model.

Build and train the model

Add the following call to the `BuildAndTrainModel` method as the next line after the call to the `ProcessData()` method:

```
var trainingPipeline = BuildAndTrainModel(_trainingDataView, pipeline);
```

The `BuildAndTrainModel` method executes the following tasks:

- Creates the training algorithm class.
- Trains the model.
- Predicts area based on training data.
- Returns the model.

Create the `BuildAndTrainModel` method, just after the declaration of the `ProcessData()` method, using the following code:

```
IEstimator<ITransformer> BuildAndTrainModel(IDataView trainingDataView, IEstimator<ITransformer> pipeline)
{
}
```

About the classification task

Classification is a machine learning task that uses data to **determine** the category, type, or class of an item or row of data and is frequently one of the following types:

- Binary: either A or B.
- Multiclass: multiple categories that can be predicted by using a single model.

For this type of problem, use a Multiclass classification learning algorithm, since your issue category prediction can be one of multiple categories (multiclass) rather than just two (binary).

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

```
var trainingPipeline =
    pipeline.Append(_mlContext.MulticlassClassification.Trainers.SdcaMaximumEntropy("Label", "Features"))
        .Append(_mlContext.Transforms.Conversion.MapKeyToValue("PredictedLabel"));
```

The `SdcaMaximumEntropy` is your multiclass classification training algorithm. This is appended to the `pipeline` and accepts the featurized `Title` and `Description` (`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:

```
_trainedModel = trainingPipeline.Fit(trainingDataView);
```

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

The [PredictionEngine](#) is a convenience API, which allows you to pass in and then perform a prediction on a single instance of data. Add this as the next line in the `BuildAndTrainModel()` method:

```
_predEngine = _mlContext.Model.CreatePredictionEngine<GitHubIssue, IssuePrediction>(_trainedModel);
```

Predict with the trained model

Add a GitHub issue to test the trained model's prediction in the `Predict` method by creating an instance of `GitHubIssue`:

```
GitHubIssue issue = new GitHubIssue() {
    Title = "WebSockets communication is slow in my machine",
    Description = "The WebSockets communication used under the covers by SignalR looks like is going slow in
my development machine.."
};
```

Use the `Predict()` function makes a prediction on a single row of data:

```
var prediction = _predEngine.Predict(issue);
```

Using the model: prediction results

Display `GitHubIssue` and corresponding `Area` label prediction in order to share the results and act on them accordingly. Create a display for the results using the following [Console.WriteLine\(\)](#) code:

```
Console.WriteLine($"===== Single Prediction just-trained-model - Result: {prediction.Area}
=====");
```

Return the model trained to use for evaluation

Return the model at the end of the `BuildAndTrainModel` method.

```
return trainingPipeline;
```

Evaluate the model

Now that you've created and trained the model, you need to evaluate it with a different dataset for quality assurance and validation. In the `Evaluate` method, the model created in `BuildAndTrainModel` is passed in to be evaluated. Create the `Evaluate` method, just after `BuildAndTrainModel`, as in the following code:

```
void Evaluate(DataViewSchema trainingDataViewSchema)
{
}
```

The `Evaluate` method executes the following tasks:

- Loads the test dataset.
- Creates the multiclass evaluator.

- Evaluates the model and create metrics.
- Displays the metrics.

Add a call to the new method, right under the `BuildAndTrainModel` method call, using the following code:

```
Evaluate(_trainingDataView.Schema);
```

As you did previously with the training dataset, load the test dataset by adding the following code to the `Evaluate` method:

```
var testDataView = _mlContext.Data.LoadFromTextFile<GitHubIssue>(_testDataPath, hasHeader: true);
```

The `Evaluate()` method computes the quality metrics for the model using the specified dataset. It returns a `MulticlassClassificationMetrics` object that contains the overall metrics computed by multiclass classification evaluators. To display the metrics to determine the quality of the model, you need to get them first. Notice the use of the `Transform()` method of the machine learning `_trainedModel` global variable (an `ITransformer`) to input the features and return predictions. Add the following code to the `Evaluate` method as the next line:

```
var testMetrics = _mlContext.MulticlassClassification.Evaluate(_trainedModel.Transform(testDataView));
```

The following metrics are evaluated for multiclass classification:

- Micro Accuracy - Every sample-class pair contributes equally to the accuracy metric. You want Micro Accuracy to be as close to one as possible.
- Macro Accuracy - Every class contributes equally to the accuracy metric. Minority classes are given equal weight as the larger classes. You want Macro Accuracy to be as close to one as possible.
- Log-loss - see [Log Loss](#). You want Log-loss to be as close to zero as possible.
- Log-loss reduction - Ranges from [-inf, 1.00], where 1.00 is perfect predictions and 0 indicates mean predictions. You want Log-loss reduction to be as close to one as possible.

Displaying the metrics for model validation

Use the following code to display the metrics, share the results, and then act on them:

```
Console.WriteLine($"*****");
Console.WriteLine($"*      Metrics for Multi-class Classification model - Test Data      ");
Console.WriteLine($"*-----");
Console.WriteLine($"*      MicroAccuracy: {testMetrics.MicroAccuracy:0.###}");
Console.WriteLine($"*      MacroAccuracy: {testMetrics.MacroAccuracy:0.###}");
Console.WriteLine($"*      LogLoss:       {testMetrics.LogLoss:#.###}");
Console.WriteLine($"*      LogLossReduction: {testMetrics.LogLossReduction:#.###}");
Console.WriteLine($"*****");
```

Save the model to a file

Once satisfied with your model, save it to a file to make predictions at a later time or in another application. Add the following code to the `Evaluate` method.

```
SaveModelAsFile(_mlContext, trainingDataViewSchema, _trainedModel);
```

Create the `SaveModelAsFile` method below your `Evaluate` method.

```
void SaveModelAsFile(MLContext mlContext, DataViewSchema trainingDataViewSchema, ITransformer model)
{
}
```

Add the following code to your `SaveModelAsFile` method. This code uses the `Save` method to serialize and store the trained model as a zip file.

```
mlContext.Model.Save(model, trainingDataViewSchema, _modelPath);
```

Deploy and Predict with a model

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

```
PredictIssue();
```

Create the `PredictIssue` method, just after the `Evaluate` method (and just before the `SaveModelAsFile` method), using the following code:

```
void PredictIssue()
{
}
```

The `PredictIssue` method executes the following tasks:

- Loads the saved model
- Creates a single issue of test data.
- Predicts Area based on test data.
- Combines test data and predictions for reporting.
- Displays the predicted results.

Load the saved model into your application by adding the following code to the `PredictIssue` method:

```
ITransformer loadedModel = _mlContext.Model.Load(_modelPath, out var modelInputSchema);
```

Add a GitHub issue to test the trained model's prediction in the `Predict` method by creating an instance of `GitHubIssue`:

```
GitHubIssue singleIssue = new GitHubIssue() { Title = "Entity Framework crashes", Description = "When connecting to the database, EF is crashing" };
```

As you did previously, create a `PredictionEngine` instance with the following code:

```
_predEngine = _mlContext.Model.CreatePredictionEngine<GitHubIssue, IssuePrediction>(loadedModel);
```

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.

NOTE

`PredictionEnginePool` service extension is currently in preview.

Use the `PredictionEngine` to predict the Area GitHub label by adding the following code to the `PredictIssue` method for the prediction:

```
var prediction = _predEngine.Predict(singleIssue);
```

Using the loaded model for prediction

Display `Area` in order to categorize the issue and act on it accordingly. Create a display for the results using the following `Console.WriteLine()` code:

```
Console.WriteLine($"===== Single Prediction - Result: {prediction.Area} =====");
```

Results

Your results should be similar to the following. As the pipeline processes, it displays messages. You may see warnings, or processing messages. These messages have been removed from the following results for clarity.

```
===== Single Prediction just-trained-model - Result: area-System.Net =====
*****
* Metrics for Multi-class Classification model - Test Data
*-----
-
* MicroAccuracy: 0.738
* MacroAccuracy: 0.668
* LogLoss: .919
* LogLossReduction: .643
*****
===== Single Prediction - Result: area-System.Data =====
```

Congratulations! You've now successfully built a machine learning model for classifying and predicting an Area label for a GitHub issue. You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Next steps

In this tutorial, you learned how to:

- Prepare your data
- Transform the data
- Train the model
- Evaluate the model
- Predict with the trained model
- Deploy and Predict with a loaded model

Advance to the next tutorial to learn more

Taxi Fare Predictor

Tutorial: Predict prices using regression with ML.NET

10/14/2022 • 10 minutes to read • [Edit Online](#)

This tutorial illustrates how to build a [regression model](#) using ML.NET to predict prices, specifically, New York City taxi fares.

In this tutorial, 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 2022](#) with the ".NET Desktop Development" workload installed.

Create a console application

1. Create a **C# Console Application** called "TaxiFarePrediction".
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to store the data set and model files.
4. Install the **Microsoft.ML** and **Microsoft.ML.FastTree** NuGet Package:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

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**. Under **Advanced**, 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 **Features** are 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.

NOTE

Use the `float` type to represent floating-point values in the input and prediction data classes.

Define data and model paths

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

```
using Microsoft.ML;
using TaxiFarePrediction;
```

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 below the usings section to specify those paths and for the `_textLoader` variable:

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

Replace the `Console.WriteLine("Hello World!")` line 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 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 using the following code:

```
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:

```
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 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:

```
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 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.

NOTE

`PredictionEnginePool` service extension is currently in preview.

This tutorial 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 tutorial at the [dotnet/samples](#) GitHub repository.

Next steps

In this tutorial, you learned how to:

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

Advance to the next tutorial to learn more.

Iris clustering



Tutorial: Categorize iris flowers using k-means clustering with ML.NET

10/14/2022 • 7 minutes to read • [Edit Online](#)

This tutorial illustrates how to use ML.NET to build a [clustering model](#) for the [iris flower data set](#).

In this tutorial, you learn how to:

- Understand the problem
- Select the appropriate machine learning task
- Prepare the data
- Load and transform the data
- Choose a learning algorithm
- Train the model
- Use the model for predictions

Prerequisites

- [Visual Studio 2022](#).

Understand the problem

This problem is about dividing the set of iris flowers in different groups based on the flower features. Those features are the length and width of a sepal and the length and width of a petal. For this tutorial, assume that the type of each flower is unknown. You want to learn the structure of a data set from the features and predict how a data instance fits this structure.

Select the appropriate machine learning task

As you don't know to which group each flower belongs to, you choose the [unsupervised machine learning](#) task. To divide a data set in groups in such a way that elements in the same group are more similar to each other than to those in other groups, use a [clustering](#) machine learning task.

Create a console application

1. Create a [C# Console Application](#) called "IrisFlowerClustering". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to store the data set and model files:

In **Solution Explorer**, right-click the project and select **Add > New Folder**. Type "Data" and hit Enter.

4. Install the [Microsoft.ML](#) NuGet package:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

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** 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.

Prepare the data

1. Download the [iris.data](#) data set and save it to the *Data* folder you've created at the previous step. For more information about the iris data set, see the [Iris flower data set](#) Wikipedia page and the [Iris Data Set](#) page, which is the source of the data set.
2. In **Solution Explorer**, right-click the *iris.data* file and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

The *iris.data* file contains five columns that represent:

- sepal length in centimeters
- sepal width in centimeters
- petal length in centimeters
- petal width in centimeters
- type of iris flower

For the sake of the clustering example, this tutorial ignores the last column.

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 *IrisData.cs*. Then, select the **Add** button.
3. Add the following `using` directive to the new file:

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

Remove the existing class definition and add the following code, which defines the classes `IrisData` and `ClusterPrediction`, to the *IrisData.cs* file:

```

public class IrisData
{
    [LoadColumn(0)]
    public float SepalLength;

    [LoadColumn(1)]
    public float SepalWidth;

    [LoadColumn(2)]
    public float PetalLength;

    [LoadColumn(3)]
    public float PetalWidth;
}

public class ClusterPrediction
{
    [ColumnName("PredictedLabel")]
    public uint PredictedClusterId;

    [ColumnName("Score")]
    public float[] Distances;
}

```

`IrisData` is the input data class and has definitions for each feature from the data set. Use the `LoadColumn` attribute to specify the indices of the source columns in the data set file.

The `clusterPrediction` class represents the output of the clustering model applied to an `IrisData` instance. Use the `ColumnName` attribute to bind the `PredictedClusterId` and `Distances` fields to the `PredictedLabel` and `Score` columns respectively. In case of the clustering task those columns have the following meaning:

- `PredictedLabel` column contains the ID of the predicted cluster.
- `Score` column contains an array with squared Euclidean distances to the cluster centroids. The array length is equal to the number of clusters.

NOTE

Use the `float` type to represent floating-point values in the input and prediction data classes.

Define data and model paths

Go back to the `Program.cs` file and add two fields to hold the paths to the data set file and to the file to save the model:

- `_dataPath` contains the path to the file with the data set used to train the model.
- `_modelPath` contains the path to the file where the trained model is stored.

Add the following code under the using statements to specify those paths:

```

string _dataPath = Path.Combine(Environment.CurrentDirectory, "Data", "iris.data");
string _modelPath = Path.Combine(Environment.CurrentDirectory, "Data", "IrisClusteringModel.zip");

```

Create ML context

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

```
using Microsoft.ML;
using IrisFlowerClustering;
```

Replace the `Console.WriteLine("Hello World!");` line with the following code:

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

The [Microsoft.ML.MLContext](#) class represents the machine learning environment and provides mechanisms for logging and entry points for data loading, model training, prediction, and other tasks. This is comparable conceptually to using `DbContext` in Entity Framework.

Set up data loading

Add the following code below the `MLContext` to set up the way to load data:

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

The generic [MLContext.Data.LoadFromTextFile](#) extension method infers the data set schema from the provided `IrisData` type and returns `IDataView` which can be used as input for transformers.

Create a learning pipeline

For this tutorial, the learning pipeline of the clustering task comprises two following steps:

- concatenate loaded columns into one **Features** column, which is used by a clustering trainer;
- use a [KMeansTrainer](#) trainer to train the model using the k-means++ clustering algorithm.

Add the following after loading the data:

```
string featuresColumnName = "Features";
var pipeline = mlContext.Transforms
    .Concatenate(featuresColumnName, "SepalLength", "SepalWidth", "PetalLength", "PetalWidth")
    .Append(mlContext.Clustering.Trainers.KMeans(featuresColumnName, numberOfClusters: 3));
```

The code specifies that the data set should be split in three clusters.

Train the model

The steps added in the preceding sections prepared the pipeline for training, however, none have been executed.

Add the following line at the bottom of the file to perform data loading and model training:

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

Save the model

At this point, you have a model that can be integrated into any of your existing or new .NET applications. To save your model to a .zip file, add the following code below calling the `Fit` method:

```
using (var fileStream = new FileStream(_modelPath, FileMode.Create, FileAccess.Write, FileShare.Write))
{
    mlContext.Model.Save(model, dataView.Schema, fileStream);
}
```

Use the model for predictions

To make predictions, use the `PredictionEngine<TSrc,TDst>` class that takes instances of the input type through the transformer pipeline and produces instances of the output type. Add the following line to create an instance of that class:

```
var predictor = mlContext.Model.CreatePredictionEngine<irisData, ClusterPrediction>(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](#).

NOTE

`PredictionEnginePool` service extension is currently in preview.

Create the `TestIrisData` class to house test data instances:

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 `TestIrisData.cs`. Then, select the **Add** button.
3. Modify the class to be static like in the following example:

```
static class TestIrisData
```

This tutorial introduces one iris data instance within this class. You can add other scenarios to experiment with the model. Add the following code into the `TestIrisData` class:

```
internal static readonly IrisData Setosa = new IrisData
{
    SepalLength = 5.1f,
    SepalWidth = 3.5f,
    PetalLength = 1.4f,
    PetalWidth = 0.2f
};
```

To find out the cluster to which the specified item belongs to, go back to the `Program.cs` file and add the following code at the bottom of the file:

```
var prediction = predictor.Predict(TestIrisData.Setosa);
Console.WriteLine($"Cluster: {prediction.PredictedClusterId}");
Console.WriteLine($"Distances: {string.Join(" ", prediction.Distances)}");
```

Run the program to see which cluster contains the specified data instance and squared distances from that

instance to the cluster centroids. Your results should be similar to the following:

```
Cluster: 2
Distances: 11.69127 0.02159119 25.59896
```

Congratulations! You've now successfully built a machine learning model for iris clustering and used it to make predictions. You can find the source code for this tutorial at the [dotnet/samples](#) GitHub repository.

Next steps

In this tutorial, you learned how to:

- Understand the problem
- Select the appropriate machine learning task
- Prepare the data
- Load and transform the data
- Choose a learning algorithm
- Train the model
- Use the model for predictions

Check out our GitHub repository to continue learning and find more samples.

[dotnet/machinelearning GitHub repository](#)

Tutorial: Build a movie recommender using matrix factorization with ML.NET

10/14/2022 • 16 minutes to read • [Edit Online](#)

This tutorial shows you how to build a movie recommender with ML.NET in a .NET Core console application. The steps use C# and Visual Studio 2019.

In this tutorial, you learn how to:

- Select a machine learning algorithm
- Prepare and load your data
- Build and train a model
- Evaluate a model
- Deploy and consume a model

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Machine learning workflow

You will use the following steps to accomplish your task, as well as any other ML.NET task:

1. [Load your data](#)
2. [Build and train your model](#)
3. [Evaluate your model](#)
4. [Use your model](#)

Prerequisites

- [Visual Studio 2022](#).

Select the appropriate machine learning task

There are several ways to approach recommendation problems, such as recommending a list of movies or recommending a list of related products, but in this case you will predict what rating (1-5) a user will give to a particular movie and recommend that movie if it's higher than a defined threshold (the higher the rating, the higher the likelihood of a user liking a particular movie).

Create a console application

Create a project

1. Create a C# Console Application called "MovieRecommender". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to store the data set:

In **Solution Explorer**, right-click the project and select **Add > New Folder**. Type "Data" and hit Enter.

4. Install the **Microsoft.ML** and **Microsoft.ML.Recommender** NuGet Packages:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

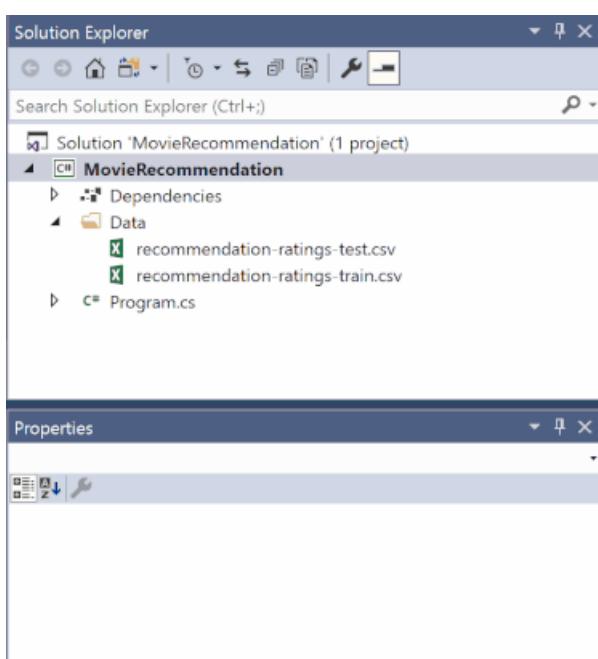
In **Solution Explorer**, right-click the project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the **Browse** tab, search for **Microsoft.ML**, select the package in the list, and select the **Install** button. Select the **OK** button on the **Preview Changes** dialog and then select the **I Accept** button on the **License Acceptance** dialog if you agree with the license terms for the packages listed. Repeat these steps for **Microsoft.ML.Recommender**.

5. Add the following `using` statements at the top of your *Program.cs* file:

```
using Microsoft.ML;
using Microsoft.ML.Trainers;
using MovieRecommendation;
```

Download your data

1. Download the two datasets and save them to the *Data* folder you previously created:
 - Right click on *recommendation-ratings-train.csv* and select "Save Link (or Target) As..."
 - Right click on *recommendation-ratings-test.csv* and select "Save Link (or Target) As..."Make sure you either save the *.csv files to the *Data* folder, or after you save it elsewhere, move the *.csv files to the *Data* folder.
2. In Solution Explorer, right-click each of the *.csv files and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.



Load your data

The first step in the ML.NET process is to prepare and load your model training and testing data.

The recommendation ratings data is split into `Train` and `Test` datasets. The `Train` data is used to fit your model. The `Test` data is used to make predictions with your trained model and evaluate model performance. It's common to have an 80/20 split with `Train` and `Test` data.

Below is a preview of the data from your *.csv files:

userId	movieId	rating	timestamp
1	1	4	964982703
1	3	4	964981247
1	6	4	964982224
1	47	5	964983815
1	50	5	964982931
1	70	3	964982400
1	101	5	964980868
1	110	4	964982176
1	151	5	964984041

In the *.csv files, there are four columns:

- `userId`
- `movieId`
- `rating`
- `timestamp`

In machine learning, the columns that are used to make a prediction are called **Features**, and the column with the returned prediction is called the **Label**.

You want to predict movie ratings, so the rating column is the `Label`. The other three columns, `userId`, `movieId`, and `timestamp` are all `Features` used to predict the `Label`.

FEATURES	LABEL
<code>userId</code>	<code>rating</code>
<code>movieId</code>	
<code>timestamp</code>	

It's up to you to decide which `Features` are used to predict the `Label`. You can also use methods like [permutation feature importance](#) to help with selecting the best `Features`.

In this case, you should eliminate the `timestamp` column as a `Feature` because the timestamp does not really affect how a user rates a given movie and thus would not contribute to making a more accurate prediction:

FEATURES	LABEL
<code>userId</code>	<code>rating</code>
<code>movieId</code>	

Next you must define your data structure for the input class.

Add a new class to your project:

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 *MovieRatingData.cs*. Then, select the **Add** button.

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

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

Create a class called `MovieRating` by removing the existing class definition and adding the following code in `MovieRatingData.cs`:

```
public class MovieRating
{
    [LoadColumn(0)]
    public float userId;
    [LoadColumn(1)]
    public float movieId;
    [LoadColumn(2)]
    public float Label;
}
```

`MovieRating` specifies an input data class. The `LoadColumn` attribute specifies which columns (by column index) in the dataset should be loaded. The `userId` and `movieId` columns are your `Features` (the inputs you will give the model to predict the `Label`), and the rating column is the `Label` that you will predict (the output of the model).

Create another class, `MovieRatingPrediction`, to represent predicted results by adding the following code after the `MovieRating` class in `MovieRatingData.cs`:

```
public class MovieRatingPrediction
{
    public float Label;
    public float Score;
}
```

In `Program.cs`, replace the `Console.WriteLine("Hello World!")` with the following code:

```
MLContext mlContext = new MLContext();
```

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

At the bottom of the file, create a method called `LoadData()`:

```
(IDataView training, IDataView test) LoadData(MLContext mlContext)
{
}
```

NOTE

This method will give you an error until you add a return statement in the following steps.

Initialize your data path variables, load the data from the *.csv files, and return the `Train` and `Test` data as `IDataView` objects by adding the following as the next line of code in `LoadData()`:

```

var trainingDataPath = Path.Combine(Environment.CurrentDirectory, "Data", "recommendation-ratings-
train.csv");
var testDataPath = Path.Combine(Environment.CurrentDirectory, "Data", "recommendation-ratings-test.csv");

IDataView trainingDataView = mlContext.Data.LoadFromTextFile<MovieRating>(trainingDataPath, hasHeader: true,
separatorChar: ',');
IDataView testDataView = mlContext.Data.LoadFromTextFile<MovieRating>(testDataPath, hasHeader: true,
separatorChar: ',');

return (trainingDataView, testDataView);

```

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 [LoadFromTextFile\(\)](#) defines the data schema and reads in the file. It takes in the data path variables and returns an `IDataView`. In this case, you provide the path for your `Test` and `Train` files and indicate both the text file header (so it can use the column names properly) and the comma character data separator (the default separator is a tab).

Add the following code to call your `LoadData()` method and return the `Train` and `Test` data:

```
(IDataView trainingDataView, IDataView testDataView) = LoadData(mlContext);
```

Build and train your model

Create the `BuildAndTrainModel()` method, just after the `LoadData()` method, using the following code:

```

ITransformer BuildAndTrainModel(MLContext mlContext, IDataView trainingDataView)
{
}
```

NOTE

This method will give you an error until you add a return statement in the following steps.

Define the data transformations by adding the following code to `BuildAndTrainModel()`:

```

IEstimator<ITransformer> estimator = mlContext.Transforms.Conversion.MapValueToKey(outputColumnName:
"userEncoded", inputColumnName: "userId")
    .Append(mlContext.Transforms.Conversion.MapValueToKey(outputColumnName: "movieIdEncoded",
inputColumnName: "movieId"));
```

Since `userId` and `movieId` represent users and movie titles, not real values, you use the [MapValueToKey\(\)](#) method to transform each `userId` and each `movieId` into a numeric key type `Feature` column (a format accepted by recommendation algorithms) and add them as new dataset columns:

USERID	MOVIEID	LABEL	USERIDENCODED	MOVIEIDENCODED
1	1	4	userKey1	movieKey1
1	3	4	userKey1	movieKey2

USERID	MOVIEID	LABEL	USERIDENCODED	MOVIEIDENCODED
1	6	4	userKey1	movieKey3

Choose the machine learning algorithm and append it to the data transformation definitions by adding the following as the next line of code in `BuildAndTrainModel()`:

```
var options = new MatrixFactorizationTrainer.Options
{
    MatrixColumnIndexColumnName = "userIdEncoded",
    MatrixRowIndexColumnName = "movieIdEncoded",
    LabelColumnName = "Label",
    NumberOfIterations = 20,
    ApproximationRank = 100
};

var trainerEstimator = estimator.Append(mlContext.Recommendation().Trainers.MatrixFactorization(options));
```

The [MatrixFactorizationTrainer](#) is your recommendation training algorithm. [Matrix Factorization](#) is a common approach to recommendation when you have data on how users have rated products in the past, which is the case for the datasets in this tutorial. There are other recommendation algorithms for when you have different data available (see the [Other recommendation algorithms](#) section below to learn more).

In this case, the [Matrix Factorization](#) algorithm uses a method called "collaborative filtering", which assumes that if User 1 has the same opinion as User 2 on a certain issue, then User 1 is more likely to feel the same way as User 2 about a different issue.

For instance, if User 1 and User 2 rate movies similarly, then User 2 is more likely to enjoy a movie that User 1 has watched and rated highly:

	INCREDIBLES 2 (2018)	THE AVENGERS (2012)	GUARDIANS OF THE GALAXY (2014)
User 1	Watched and liked movie	Watched and liked movie	Watched and liked movie
User 2	Watched and liked movie	Watched and liked movie	Has not watched -- RECOMMEND movie

The [Matrix Factorization](#) trainer has several [Options](#), which you can read more about in the [Algorithm hyperparameters](#) section below.

Fit the model to the [Train](#) data and return the trained model by adding the following as the next line of code in the `BuildAndTrainModel()` method:

```
Console.WriteLine("===== Training the model =====");
ITransformer model = trainerEstimator.Fit(training DataView);

return model;
```

The [Fit\(\)](#) method trains your model with the provided training dataset. Technically, it executes the [Estimator](#) definitions by transforming the data and applying the training, and it returns back the trained model, which is a [Transformer](#).

For more information on the model training workflow in ML.NET, see [What is ML.NET and how does it work?](#).

Add the following as the next line of code below the call to the `LoadData()` method to call your `BuildAndTrainModel()` method and return the trained model:

```
ITransformer model = BuildAndTrainModel(mlContext, trainingDataView);
```

Evaluate your model

Once you have trained your model, use your test data to evaluate how your model is performing.

Create the `EvaluateModel()` method, just after the `BuildAndTrainModel()` method, using the following code:

```
void EvaluateModel(MLContext mlContext, IDataView testDataView, ITransformer model)
{
}
```

Transform the `Test` data by adding the following code to `EvaluateModel()`:

```
Console.WriteLine("===== Evaluating the model =====");
var prediction = model.Transform(testDataView);
```

The `Transform()` method makes predictions for multiple provided input rows of a test dataset.

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

```
var metrics = mlContext.Regression.Evaluate(prediction, labelColumnName: "Label", scoreColumnName: "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.

Print your evaluation metrics to the console by adding the following as the next line of code in the `EvaluateModel()` method:

```
Console.WriteLine("Root Mean Squared Error : " + metrics.RootMeanSquaredError.ToString());
Console.WriteLine("RSquared: " + metrics.RSquared.ToString());
```

Add the following as the next line of code below the call to the `BuildAndTrainModel()` method to call your `EvaluateModel()` method:

```
EvaluateModel(mlContext, testDataView, model);
```

The output so far should look similar to the following text:

```

===== Training the model =====
iter      tr_rmse      obj
 0        1.5403    3.1262e+05
 1        0.9221    1.6030e+05
 2        0.8687    1.5046e+05
 3        0.8416    1.4584e+05
 4        0.8142    1.4209e+05
 5        0.7849    1.3907e+05
 6        0.7544    1.3594e+05
 7        0.7266    1.3361e+05
 8        0.6987    1.3110e+05
 9        0.6751    1.2948e+05
10       0.6530    1.2766e+05
11       0.6350    1.2644e+05
12       0.6197    1.2541e+05
13       0.6067    1.2470e+05
14       0.5953    1.2382e+05
15       0.5871    1.2342e+05
16       0.5781    1.2279e+05
17       0.5713    1.2240e+05
18       0.5660    1.2230e+05
19       0.5592    1.2179e+05
===== Evaluating the model =====
Rms: 0.994051469730769
RSquared: 0.412556298844873

```

In this output, there are 20 iterations. In each iteration, the measure of error decreases and converges closer and closer to 0.

The `root of mean squared error` (RMS or RMSE) is used to measure the differences between the model predicted values and the test dataset observed values. Technically it's the square root of the average of the squares of the errors. The lower it is, the better the model is.

`R Squared` indicates how well data fits a model. Ranges from 0 to 1. A value of 0 means that the data is random or otherwise can't be fit to the model. A value of 1 means that the model exactly matches the data. You want your `R Squared` score to be as close to 1 as possible.

Building successful models is an iterative process. This model has initial lower quality as the tutorial 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. For more information, check out the [Improve your model](#) section below.

Use your model

Now you can use your trained model to make predictions on new data.

Create the `UseModelForSinglePrediction()` method, just after the `EvaluateModel()` method, using the following code:

```

void UseModelForSinglePrediction(MLContext mlContext, ITransformer model)
{
}

```

Use the `PredictionEngine` to predict the rating by adding the following code to `UseModelForSinglePrediction()`:

```

Console.WriteLine("===== Making a prediction =====");
var predictionEngine = mlContext.Model.CreatePredictionEngine<MovieRating, MovieRatingPrediction>(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.

NOTE

[PredictionEnginePool](#) service extension is currently in preview.

Create an instance of [MovieRating](#) called [testInput](#) and pass it to the Prediction Engine by adding the following as the next lines of code in the [UseModelForSinglePrediction\(\)](#) method:

```
var testInput = new MovieRating { userId = 6, movieId = 10 };

var movieRatingPrediction = predictionEngine.Predict(testInput);
```

The [Predict\(\)](#) function makes a prediction on a single column of data.

You can then use the [Score](#), or the predicted rating, to determine whether you want to recommend the movie with movielid 10 to user 6. The higher the [Score](#), the higher the likelihood of a user liking a particular movie. In this case, let's say that you recommend movies with a predicted rating of > 3.5.

To print the results, add the following as the next lines of code in the [UseModelForSinglePrediction\(\)](#) method:

```
if (Math.Round(movieRatingPrediction.Score, 1) > 3.5)
{
    Console.WriteLine("Movie " + testInput.movieId + " is recommended for user " + testInput.userId);
}
else
{
    Console.WriteLine("Movie " + testInput.movieId + " is not recommended for user " + testInput.userId);
}
```

Add the following as the next line of code after the call to the [EvaluateModel\(\)](#) method to call your [UseModelForSinglePrediction\(\)](#) method:

```
UseModelForSinglePrediction(mlContext, model);
```

The output of this method should look similar to the following text:

```
===== Making a prediction =====
Movie 10 is recommended for user 6
```

Save your model

To use your model to make predictions in end-user applications, you must first save the model.

Create the [SaveModel\(\)](#) method, just after the [UseModelForSinglePrediction\(\)](#) method, using the following code:

```
void SaveModel(MLContext mlContext, DataViewSchema trainingDataViewSchema, ITransformer model)
{ }

}
```

Save your trained model by adding the following code in the `SaveModel()` method:

```
var modelPath = Path.Combine(Environment.CurrentDirectory, "Data", "MovieRecommenderModel.zip");

Console.WriteLine("===== Saving the model to a file =====");
mlContext.Model.Save(model, trainingDataViewSchema, modelPath);
```

This method saves your trained model to a .zip file (in the "Data" folder), which can then be used in other .NET applications to make predictions.

Add the following as the next line of code after the call to the `UseModelForSinglePrediction()` method to call your `SaveModel()` method:

```
SaveModel(mlContext, trainingDataView.Schema, model);
```

Use your saved model

Once you have saved your trained model, you can consume the model in different environments. See [Save and load trained models](#) to learn how to operationalize a trained machine learning model in apps.

Results

After following the steps above, run your console app (Ctrl + F5). Your results from the single prediction above should be similar to the following. You may see warnings or processing messages, but these messages have been removed from the following results for clarity.

```
===== Training the model =====
iter      tr_rmse      obj
 0        1.5382    3.1213e+05
 1        0.9223    1.6051e+05
 2        0.8691    1.5050e+05
 3        0.8413    1.4576e+05
 4        0.8145    1.4208e+05
 5        0.7848    1.3895e+05
 6        0.7552    1.3613e+05
 7        0.7259    1.3357e+05
 8        0.6987    1.3121e+05
 9        0.6747    1.2949e+05
10       0.6533    1.2766e+05
11       0.6353    1.2636e+05
12       0.6209    1.2561e+05
13       0.6072    1.2462e+05
14       0.5965    1.2394e+05
15       0.5868    1.2352e+05
16       0.5782    1.2279e+05
17       0.5713    1.2227e+05
18       0.5637    1.2190e+05
19       0.5604    1.2178e+05
===== Evaluating the model =====
Rms: 0.977175077487166
RSquared: 0.43233349213192
===== Making a prediction =====
Movie 10 is recommended for user 6
===== Saving the model to a file =====
```

Congratulations! You've now successfully built a machine learning model for recommending movies. You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Improve your model

There are several ways that you can improve the performance of your model so that you can get more accurate predictions.

Data

Adding more training data that has enough samples for each user and movie id can help improve the quality of the recommendation model.

[Cross validation](#) is a technique for evaluating models that randomly splits up data into subsets (instead of extracting out test data from the dataset like you did in this tutorial) and takes some of the groups as train data and some of the groups as test data. This method outperforms making a train-test split in terms of model quality.

Features

In this tutorial, you only use the three `Features` (`user id`, `movie id`, and `rating`) that are provided by the dataset.

While this is a good start, in reality you might want to add other attributes or `Features` (for example, age, gender, geo-location, etc.) if they are included in the dataset. Adding more relevant `Features` can help improve the performance of your recommendation model.

If you are unsure about which `Features` might be the most relevant for your machine learning task, you can also make use of Feature Contribution Calculation (FCC) and [permutation feature importance](#), which ML.NET provides to discover the most influential `Features`.

Algorithm hyperparameters

While ML.NET provides good default training algorithms, you can further fine-tune performance by changing the algorithm's [hyperparameters](#).

For `Matrix Factorization`, you can experiment with hyperparameters such as [NumberOfIterations](#) and [ApproximationRank](#) to see if that gives you better results.

For instance, in this tutorial the algorithm options are:

```
var options = new MatrixFactorizationTrainer.Options
{
    MatrixColumnIndexColumnName = "userIdEncoded",
    MatrixRowIndexColumnName = "movieIdEncoded",
    LabelColumnName = "Label",
    NumberOfIterations = 20,
    ApproximationRank = 100
};
```

Other Recommendation Algorithms

The matrix factorization algorithm with collaborative filtering is only one approach for performing movie recommendations. In many cases, you may not have the ratings data available and only have movie history available from users. In other cases, you may have more than just the user's rating data.

ALGORITHM	SCENARIO	SAMPLE
One Class Matrix Factorization	Use this when you only have userId and movieId. This style of recommendation is based upon the co-purchase scenario, or products frequently bought together, which means it will recommend to customers a set of products based upon their own purchase order history.	> Try it out

ALGORITHM	SCENARIO	SAMPLE
Field Aware Factorization Machines	Use this to make recommendations when you have more Features beyond userId, productId, and rating (such as product description or product price). This method also uses a collaborative filtering approach.	> Try it out

New user scenario

One common problem in collaborative filtering is the cold start problem, which is when you have a new user with no previous data to draw inferences from. This problem is often solved by asking new users to create a profile and, for instance, rate movies they have seen in the past. While this method puts some burden on the user, it provides some starting data for new users with no rating history.

Resources

The data used in this tutorial is derived from [MovieLens Dataset](#).

Next steps

In this tutorial, you learned how to:

- Select a machine learning algorithm
- Prepare and load your data
- Build and train a model
- Evaluate a model
- Deploy and consume a model

Advance to the next tutorial to learn more

[Sentiment Analysis](#)

Tutorial: Automated visual inspection using transfer learning with the ML.NET Image Classification API

10/14/2022 • 16 minutes to read • [Edit Online](#)

Learn how to train a custom deep learning model using transfer learning, a pretrained TensorFlow model and the ML.NET Image Classification API to classify images of concrete surfaces as cracked or uncracked.

In this tutorial, you learn how to:

- Understand the problem
- Learn about ML.NET Image Classification API
- Understand the pretrained model
- Use transfer learning to train a custom TensorFlow image classification model
- Classify images with the custom model

Prerequisites

- [Visual Studio 2022](#).

Image classification transfer learning sample overview

This sample is a C# .NET Core console application that classifies images using a pretrained deep learning TensorFlow model. The code for this sample can be found on the [samples browser](#).

Understand the problem

Image classification is a computer vision problem. Image classification takes an image as input and categorizes it into a prescribed class. Image classification models are commonly trained using deep learning and neural networks. See [Deep learning vs. machine learning](#) for more information.

Some scenarios where image classification is useful include:

- Facial recognition
- Emotion detection
- Medical diagnosis
- Landmark detection

This tutorial trains a custom image classification model to perform automated visual inspection of bridge decks to identify structures that are damaged by cracks.

ML.NET Image Classification API

ML.NET provides various ways of performing image classification. This tutorial applies transfer learning using the Image Classification API. The Image Classification API makes use of [TensorFlow.NET](#), a low-level library that provides C# bindings for the TensorFlow C++ API.

What is transfer learning?

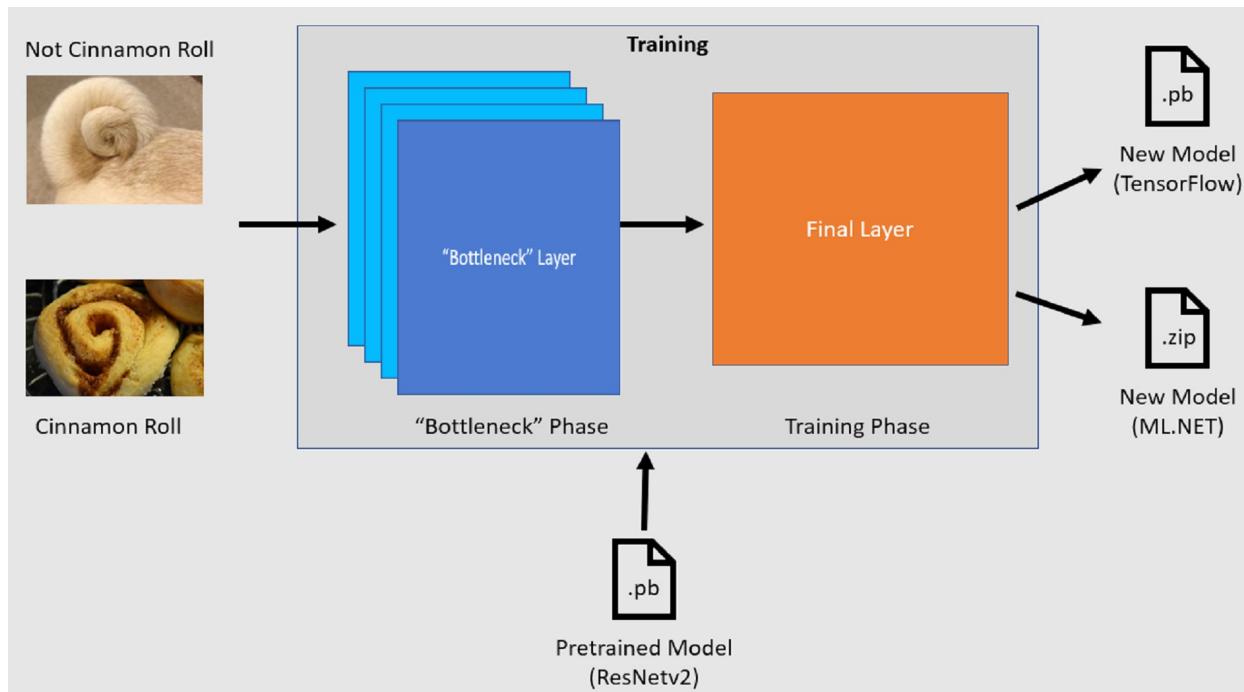
Transfer learning applies knowledge gained from solving one problem to another related problem.

Training a deep learning model from scratch requires setting several parameters, a large amount of labeled training data, and a vast amount of compute resources (hundreds of GPU hours). Using a pretrained model along with transfer learning allows you to shortcut the training process.

Training process

The Image Classification API starts the training process by loading a pretrained TensorFlow model. The training process consists of two steps:

1. Bottleneck phase
2. Training phase



Bottleneck phase

During the bottleneck phase, the set of training images is loaded and the pixel values are used as input, or features, for the frozen layers of the pretrained model. The frozen layers include all of the layers in the neural network up to the penultimate layer, informally known as the bottleneck layer. These layers are referred to as frozen because no training will occur on these layers and operations are pass-through. It's at these frozen layers where the lower-level patterns that help a model differentiate between the different classes are computed. The larger the number of layers, the more computationally intensive this step is. Fortunately, since this is a one-time calculation, the results can be cached and used in later runs when experimenting with different parameters.

Training phase

Once the output values from the bottleneck phase are computed, they are used as input to retrain the final layer of the model. This process is iterative and runs for the number of times specified by model parameters. During each run, the loss and accuracy are evaluated. Then, the appropriate adjustments are made to improve the model with the goal of minimizing the loss and maximizing the accuracy. Once training is finished, two model formats are output. One of them is the .pb version of the model and the other is the .zip ML.NET serialized version of the model. When working in environments supported by ML.NET, it is recommended to use the .zip version of the model. However, in environments where ML.NET is not supported, you have the option of using the .pb version.

Understand the pretrained model

The pretrained model used in this tutorial is the 101-layer variant of the Residual Network (ResNet) v2 model. The original model is trained to classify images into a thousand categories. The model takes as input an image

of size 224 x 224 and outputs the class probabilities for each of the classes it's trained on. Part of this model is used to train a new model using custom images to make predictions between two classes.

Create console application

Now that you have a general understanding of transfer learning and the Image Classification API, it's time to build the application.

1. Create a C# Console Application called "DeepLearning_ImageClassification_Binary". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Install the **Microsoft.ML** NuGet Package:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

- a. In Solution Explorer, right-click on your project and select **Manage NuGet Packages**.
- b. Choose "nuget.org" as the Package source.
- c. Select the **Browse** tab.
- d. Check the **Include prerelease** checkbox.
- e. Search for **Microsoft.ML**.
- f. Select the **Install** button.
- g. 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.
- h. Repeat these steps for the **Microsoft.ML.Vision**, **SciSharp.TensorFlow.Redist** version 2.3.1, and **Microsoft.ML.ImageAnalytics** NuGet packages.

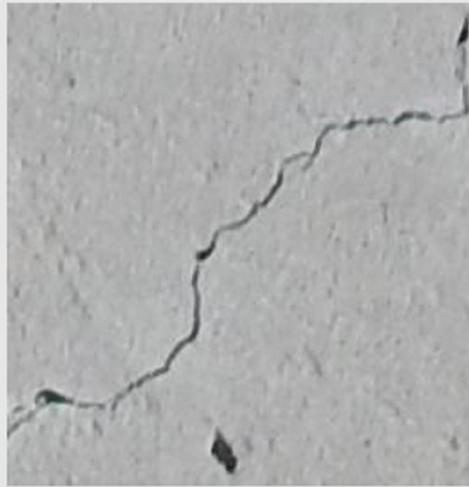
Prepare and understand the data

NOTE

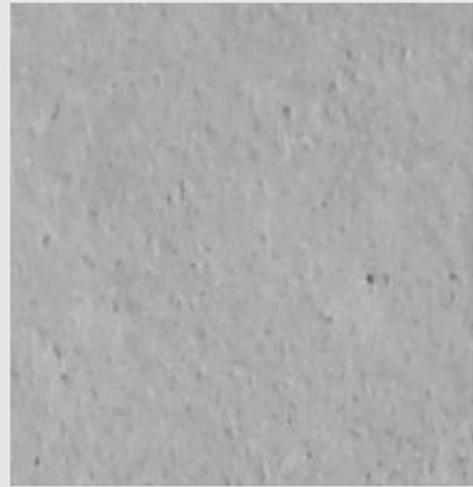
The datasets for this tutorial are from Maguire, Marc; Dorafshan, Sattar; and Thomas, Robert J., "SDNET2018: A concrete crack image dataset for machine learning applications" (2018). Browse all Datasets. Paper 48.

https://digitalcommons.usu.edu/all_datasets/48

SDNET2018 is an image dataset that contains annotations for cracked and non-cracked concrete structures (bridge decks, walls, and pavement).



Cracked Deck



Uncracked Deck

The data is organized in three subdirectories:

- D contains bridge deck images
- P contains pavement images
- W contains wall images

Each of these subdirectories contains two additional prefixed subdirectories:

- C is the prefix used for cracked surfaces
- U is the prefix used for uncracked surfaces

In this tutorial, only bridge deck images are used.

1. Download the [dataset](#) and unzip.
2. Create a directory named "assets" in your project to save your dataset files.
3. Copy the *CD* and *UD* subdirectories from the recently unzipped directory to the *assets* directory.

Create input and output classes

1. Open the *Program.cs* file and replace the existing `using` statements at the top of the file with the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;
using Microsoft.ML;
using static Microsoft.ML.DataOperationsCatalog;
using Microsoft.ML.Vision;
```

2. Below the `Program` class in *Program.cs*, create a class called `ImageData`. This class is used to represent the initially loaded data.

```
class ImageData
{
    public string ImagePath { get; set; }

    public string Label { get; set; }
}
```

`ImageData` contains the following properties:

- `ImagePath` is the fully qualified path where the image is stored.
- `Label` is the category the image belongs to. This is the value to predict.

3. Create classes for your input and output data

- a. Below the `ImageData` class, define the schema of your input data in a new class called `ModelInput`.

```
class ModelInput
{
    public byte[] Image { get; set; }

    public UInt32 LabelAsKey { get; set; }

    public string ImagePath { get; set; }

    public string Label { get; set; }
}
```

`ModelInput` contains the following properties:

- `Image` is the `byte[]` representation of the image. The model expects image data to be of this type for training.
- `LabelAsKey` is the numerical representation of the `Label`.
- `ImagePath` is the fully qualified path where the image is stored.
- `Label` is the category the image belongs to. This is the value to predict.

Only `Image` and `LabelAsKey` are used to train the model and make predictions. The `ImagePath` and `Label` properties are kept for convenience to access the original image file name and category.

- b. Then, below the `ModelInput` class, define the schema of your output data in a new class called `ModelOutput`.

```
class ModelOutput
{
    public string ImagePath { get; set; }

    public string Label { get; set; }

    public string PredictedLabel { get; set; }
}
```

`ModelOutput` contains the following properties:

- `ImagePath` is the fully qualified path where the image is stored.
- `Label` is the original category the image belongs to. This is the value to predict.
- `PredictedLabel` is the value predicted by the model.

Similar to `ModelInput`, only the `PredictedLabel` is required to make predictions since it contains the prediction made by the model. The `ImagePath` and `Label` properties are retained for convenience to access the original image file name and category.

Create workspace directory

When training and validation data do not change often, it is good practice to cache the computed bottleneck values for further runs.

1. In your project, create a new directory called `workspace` to store the computed bottleneck values and `.pb`

version of the model.

Define paths and initialize variables

- Under the using statements, define the location of your assets, computed bottleneck values and .pb version of the model.

```
var projectDirectory = Path.GetFullPath(Path.Combine(AppContext.BaseDirectory, "../../../../../"));
var workspaceRelativePath = Path.Combine(projectDirectory, "workspace");
var assetsRelativePath = Path.Combine(projectDirectory, "assets");
```

- Initialize the `m1Context` variable with a new instance of [MLContext](#).

```
MLContext m1Context = new MLContext();
```

The [MLContext](#) class is a starting point for all ML.NET operations, and initializing `m1Context` 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.

Load the data

Create data loading utility method

The images are stored in two subdirectories. Before loading the data, it needs to be formatted into a list of `ImageData` objects. To do so, create the `LoadImagesFromDirectory` method.

```
IEnumerable<ImageData> LoadImagesFromDirectory(string folder, bool useFolderNameAsLabel = true)
{
}
```

- Inside the `LoadImagesFromDirectory`, add the following code to get all of the file paths from the subdirectories:

```
var files = Directory.GetFiles(folder, "*",
    searchOption: SearchOption.AllDirectories);
```

- Then, iterate through each of the files using a `foreach` statement.

```
foreach (var file in files)
{
}
```

- Inside the `foreach` statement, check that the file extensions are supported. The Image Classification API supports JPEG and PNG formats.

```
if ((Path.GetExtension(file) != ".jpg") && (Path.GetExtension(file) != ".png"))
    continue;
```

- Then, get the label for the file. If the `useFolderNameAsLabel` parameter is set to `true`, then the parent directory where the file is saved is used as the label. Otherwise, it expects the label to be a prefix of the file name or the file name itself.

```

var label = Path.GetFileName(file);

if (useFolderNameAsLabel)
    label = Directory.GetParent(file).Name;
else
{
    for (int index = 0; index < label.Length; index++)
    {
        if (!char.IsLetter(label[index]))
        {
            label = label.Substring(0, index);
            break;
        }
    }
}

```

- Finally, create a new instance of `ModelInput`.

```

yield return new ImageData()
{
    ImagePath = file,
    Label = label
};

```

Prepare the data

- Call the `LoadImagesFromDirectory` utility method to get the list of images used for training after initializing the `mlContext` variable.

```

IEnumerable<ImageData> images = LoadImagesFromDirectory(folder: assetsRelativePath,
useFolderNameAsLabel: true);

```

- Then, load the images into an `IDataView` using the `LoadFromEnumerable` method.

```

IDataView imageData = mlContext.Data.LoadFromEnumerable(images);

```

- The data is loaded in the order it was read from the directories. To balance the data, shuffle it using the `ShuffleRows` method.

```

IDataView shuffledData = mlContext.Data.ShuffleRows(imageData);

```

- Machine learning models expect input to be in numerical format. Therefore, some preprocessing needs to be done on the data prior to training. Create an `EstimatorChain` made up of the `MapValueToKey` and `LoadRawImageBytes` transforms. The `MapValueToKey` transform takes the categorical value in the `Label` column, converts it to a numerical `KeyType` value and stores it in a new column called `LabelAsKey`. The `LoadImages` takes the values from the `ImagePath` column along with the `imageFolder` parameter to load images for training.

```

var preprocessingPipeline = mlContext.Transforms.Conversion.MapValueToKey(
    inputColumnName: "Label",
    outputColumnName: "LabelAsKey")
.Append(mlContext.Transforms.LoadRawImageBytes(
    outputColumnName: "Image",
    imageFolder: assetsRelativePath,
    inputColumnName: "ImagePath"));

```

5. Use the `Fit` method to apply the data to the `preprocessingPipeline` `EstimatorChain` followed by the `Transform` method, which returns an `IDataView` containing the pre-processed data.

```
IDataView preProcessedData = preprocessingPipeline
    .Fit(shuffledData)
    .Transform(shuffledData);
```

6. To train a model, it's important to have a training dataset as well as a validation dataset. The model is trained on the training set. How well it makes predictions on unseen data is measured by the performance against the validation set. Based on the results of that performance, the model makes adjustments to what it has learned in an effort to improve. The validation set can come from either splitting your original dataset or from another source that has already been set aside for this purpose. In this case, the pre-processed dataset is split into training, validation and test sets.

```
TrainTestData trainSplit = mlContext.Data.TrainTestSplit(data: preProcessedData, testFraction: 0.3);
TrainTestData validationTestSplit = mlContext.Data.TrainTestSplit(trainSplit.TestSet);
```

The code sample above performs two splits. First, the pre-processed data is split and 70% is used for training while the remaining 30% is used for validation. Then, the 30% validation set is further split into validation and test sets where 90% is used for validation and 10% is used for testing.

A way to think about the purpose of these data partitions is taking an exam. When studying for an exam, you review your notes, books, or other resources to get a grasp on the concepts that are on the exam. This is what the train set is for. Then, you might take a mock exam to validate your knowledge. This is where the validation set comes in handy. You want to check whether you have a good grasp of the concepts before taking the actual exam. Based on those results, you take note of what you got wrong or didn't understand well and incorporate your changes as you review for the real exam. Finally, you take the exam. This is what the test set is used for. You've never seen the questions that are on the exam and now use what you learned from training and validation to apply your knowledge to the task at hand.

7. Assign the partitions their respective values for the train, validation and test data.

```
IDataView trainSet = trainSplit.TrainSet;
IDataView validationSet = validationTestSplit.TrainSet;
IDataView testSet = validationTestSplit.TestSet;
```

Define the training pipeline

Model training consists of a couple of steps. First, Image Classification API is used to train the model. Then, the encoded labels in the `PredictedLabel` column are converted back to their original categorical value using the `MapKeyToValue` transform.

1. Create a new variable to store a set of required and optional parameters for an

```
ImageClassificationTrainer .
```

```

var classifierOptions = new ImageClassificationTrainer.Options()
{
    FeatureColumnName = "Image",
    LabelColumnName = "LabelAsKey",
    ValidationSet = validationSet,
    Arch = ImageClassificationTrainer.Architecture.ResnetV2101,
    MetricsCallback = (metrics) => Console.WriteLine(metrics),
    TestOnTrainSet = false,
    ReuseTrainSetBottleneckCachedValues = true,
    ReuseValidationSetBottleneckCachedValues = true
};

```

An `ImageClassificationTrainer` takes several optional parameters:

- `FeatureColumnName` is the column that is used as input for the model.
- `LabelColumnName` is the column for the value to predict.
- `ValidationSet` is the `IDataView` containing the validation data.
- `Arch` defines which of the pretrained model architectures to use. This tutorial uses the 101-layer variant of the ResNetv2 model.
- `MetricsCallback` binds a function to track the progress during training.
- `TestOnTrainSet` tells the model to measure performance against the training set when no validation set is present.
- `ReuseTrainSetBottleneckCachedValues` tells the model whether to use the cached values from the bottleneck phase in subsequent runs. The bottleneck phase is a one-time pass-through computation that is computationally intensive the first time it is performed. If the training data does not change and you want to experiment using a different number of epochs or batch size, using the cached values significantly reduces the amount of time required to train a model.
- `ReuseValidationSetBottleneckCachedValues` is similar to `ReuseTrainSetBottleneckCachedValues` only that in this case it's for the validation dataset.
- `WorkspacePath` defines the directory where to store the computed bottleneck values and `.pb` version of the model.

2. Define the `EstimatorChain` training pipeline that consists of both the `mapLabelEstimator` and the `ImageClassificationTrainer`.

```

var trainingPipeline =
    mlContext.MulticlassClassification.Trainers.ImageClassification(classifierOptions)
        .Append(mlContext.Transforms.Conversion.MapKeyToValue("PredictedLabel"));

```

3. Use the `Fit` method to train your model.

```
ITransformer trainedModel = trainingPipeline.Fit(trainSet);
```

Use the model

Now that you have trained your model, it's time to use it to classify images.

Create a new utility method called `outputPrediction` to display prediction information in the console.

```

private static void OutputPrediction(ModelOutput prediction)
{
    string imageName = Path.GetFileName(prediction.ImagePath);
    Console.WriteLine($"Image: {imageName} | Actual Value: {prediction.Label} | Predicted Value: {prediction.PredictedLabel}");
}

```

Classify a single image

1. Create a new method called `ClassifySingleImage` to make and output a single image prediction.

```

void ClassifySingleImage(MLContext mlContext, IDataView data, ITransformer trainedModel)
{
}

```

2. Create a `PredictionEngine` inside the `ClassifySingleImage` method. The `PredictionEngine` is a convenience API, which allows you to pass in and then perform a prediction on a single instance of data.

```

PredictionEngine<ModelInput, ModelOutput> predictionEngine =
    mlContext.Model.CreatePredictionEngine<ModelInput, ModelOutput>(trainedModel);

```

3. To access a single `ModelInput` instance, convert the `data` `IDataView` into an `IEnumerable` using the `CreateEnumerable` method and then get the first observation.

```

ModelInput image = mlContext.Data.CreateEnumerable<ModelInput>(data, reuseRowObject:true).First();

```

4. Use the `Predict` method to classify the image.

```

ModelOutput prediction = predictionEngine.Predict(image);

```

5. Output the prediction to the console with the `OutputPrediction` method.

```

Console.WriteLine("Classifying single image");
OutputPrediction(prediction);

```

6. Call `ClassifySingleImage` below calling the `Fit` method using the test set of images.

```

ClassifySingleImage(mlContext, testSet, trainedModel);

```

Classify multiple images

1. Add a new method called `ClassifyImages` below the `ClassifySingleImage` method to make and output multiple image predictions.

```

void ClassifyImages(MLContext mlContext, IDataView data, ITransformer trainedModel)
{
}

```

2. Create an `IDataView` containing the predictions by using the `Transform` method. Add the following code inside the `ClassifyImages` method.

```
IDataView predictionData = trainedModel.Transform(data);
```

3. In order to iterate over the predictions, convert the `predictionData` `IDataView` into an `IEnumerable` using the `CreateEnumerable` method and then get the first 10 observations.

```
IEnumerable<ModelOutput> predictions = mlContext.Data.CreateEnumerable<ModelOutput>(predictionData,  
reuseRowObject: true).Take(10);
```

4. Iterate and output the original and predicted labels for the predictions.

```
Console.WriteLine("Classifying multiple images");  
foreach (var prediction in predictions)  
{  
    OutputPrediction(prediction);  
}
```

5. Finally, call `classifyImages` below the `ClassifySingleImage()` method using the test set of images.

```
ClassifyImages(mlContext, testSet, trainedModel);
```

Run the application

Run your console app. The output should be similar to that below. You may see warnings or processing messages, but these messages have been removed from the following results for clarity. For brevity, the output has been condensed.

Bottleneck phase

No value is printed for the image name because the images are loaded as a `byte[]` therefore there is no image name to display.

```
Phase: Bottleneck Computation, Dataset used: Train, Image Index: 279  
Phase: Bottleneck Computation, Dataset used: Train, Image Index: 280  
Phase: Bottleneck Computation, Dataset used: Validation, Image Index: 1  
Phase: Bottleneck Computation, Dataset used: Validation, Image Index: 2
```

Training phase

```
Phase: Training, Dataset used: Validation, Batch Processed Count: 6, Epoch: 21, Accuracy: 0.6797619  
Phase: Training, Dataset used: Validation, Batch Processed Count: 6, Epoch: 22, Accuracy: 0.7642857  
Phase: Training, Dataset used: Validation, Batch Processed Count: 6, Epoch: 23, Accuracy: 0.7916667
```

Classify images output

```
Classifying single image  
Image: 7001-220.jpg | Actual Value: UD | Predicted Value: UD  
  
Classifying multiple images  
Image: 7001-220.jpg | Actual Value: UD | Predicted Value: UD  
Image: 7001-163.jpg | Actual Value: UD | Predicted Value: UD  
Image: 7001-210.jpg | Actual Value: UD | Predicted Value: UD
```

Upon inspection of the `7001-220.jpg` image, you can see that it in fact is not cracked.



Congratulations! You've now successfully built a deep learning model for classifying images.

Improve the model

If you're not satisfied with the results of your model, you can try to improve its performance by trying some of the following approaches:

- **More Data:** The more examples a model learns from, the better it performs. Download the full [SDNET2018 dataset](#) and use it to train.
- **Augment the data:** A common technique to add variety to the data is to augment the data by taking an image and applying different transforms (rotate, flip, shift, crop). This adds more varied examples for the model to learn from.
- **Train for a longer time:** The longer you train, the more tuned the model will be. Increasing the number of epochs may improve the performance of your model.
- **Experiment with the hyper-parameters:** In addition to the parameters used in this tutorial, other parameters can be tuned to potentially improve performance. Changing the learning rate, which determines the magnitude of updates made to the model after each epoch may improve performance.
- **Use a different model architecture:** Depending on what your data looks like, the model that can best learn its features may differ. If you're not satisfied with the performance of your model, try changing the architecture.

Next steps

In this tutorial, you learned how to build a custom deep learning model using transfer learning, a pretrained image classification TensorFlow model and the ML.NET Image Classification API to classify images of concrete surfaces as cracked or uncracked.

Advance to the next tutorial to learn more.

[Object Detection](#)

Tutorial: Train an ML.NET classification model to categorize images

10/14/2022 • 12 minutes to read • [Edit Online](#)

Learn how to train a classification model to categorize images using a pre-trained TensorFlow model for image processing.

The TensorFlow model was trained to classify images into a thousand categories. Because the TensorFlow model knows how to recognize patterns in images, the ML.NET model can make use of part of it in its pipeline to convert raw images into features or inputs to train a classification model.

In this tutorial, you learn how to:

- Understand the problem
- Incorporate the pre-trained TensorFlow model into the ML.NET pipeline
- Train and evaluate the ML.NET model
- Classify a test image

You can find the source code for this tutorial at the [dotnet/samples](#) repository. By default, the .NET project configuration for this tutorial targets .NET core 2.2.

Prerequisites

- [Visual Studio 2022](#).
- [The tutorial assets directory ZIP file](#)
- [The InceptionV1 machine learning model](#)

Select the right machine learning task

Deep learning

[Deep learning](#) is a subset of Machine Learning, which is revolutionizing areas like computer vision and speech recognition.

Deep learning models are trained by using large sets of [labeled data](#) and [neural networks](#) that contain multiple learning layers. Deep learning:

- Performs better on some tasks like computer vision.
- Requires huge amounts of training data.

Image classification is a specific classification task that allows us to automatically classify images into categories such as:

- Detecting a human face in an image or not.
- Detecting cats vs. dogs.

Or as in the following images, determining if an image is a food, toy, or appliance:



NOTE

The preceding images belong to Wikimedia Commons and are attributed as follows:

- "220px-Pepperoni_pizza.jpg" Public Domain, <https://commons.wikimedia.org/w/index.php?curid=79505>,
- "119px-Nalle_-_a_small_brown_teddy_bear.jpg" By [Jonik](#) - Self-photographed, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=48166>.
- "193px-Broodrooster.jpg" By [M.Minderhoud](#) - Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=27403>

Training an [image classification](#) model from scratch requires setting millions of parameters, a ton of labeled training data and a vast amount of compute resources (hundreds of GPU hours). While not as effective as training a custom model from scratch, using a pre-trained model allows you to shortcut this process by working with thousands of images vs. millions of labeled images and build a customized model fairly quickly (within an hour on a machine without a GPU). This tutorial scales that process down even further, using only a dozen training images.

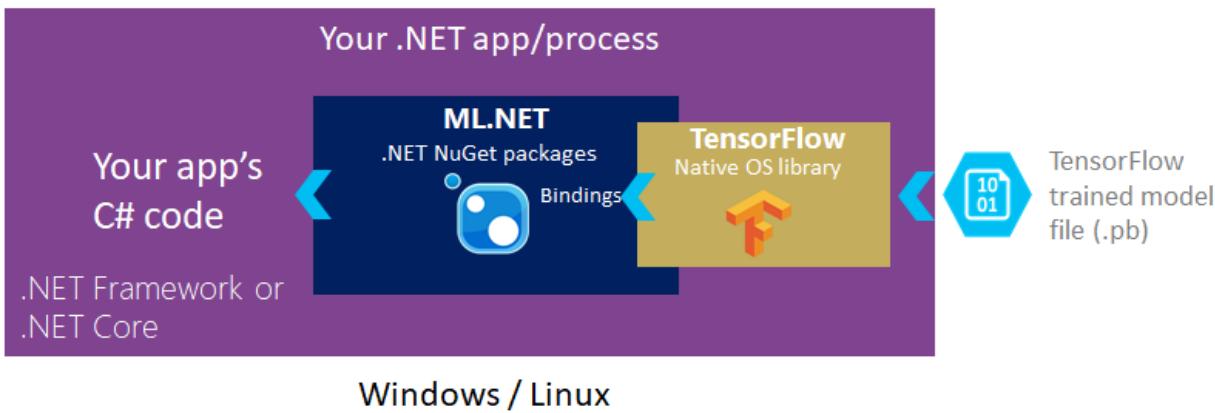
The [Inception model](#) is trained to classify images into a thousand categories, but for this tutorial, you need to classify images in a smaller category set, and only those categories. You can use the [Inception model](#)'s ability to recognize and classify images to the new limited categories of your custom image classifier.

- Food
- Toy
- Appliance

This tutorial uses the TensorFlow [Inception](#) deep learning model, a popular image recognition model trained on the [ImageNet](#) dataset. The TensorFlow model classifies entire images into a thousand classes, such as "Umbrella", "Jersey", and "Dishwasher".

Because the [Inception model](#) has already been pre-trained on thousands of different images, internally it contains the [image features](#) needed for image identification. We can make use of these internal image features in the model to train a new model with far fewer classes.

As shown in the following diagram, you add a reference to the ML.NET NuGet packages in your .NET Core or .NET Framework applications. Under the covers, ML.NET includes and references the native [TensorFlow](#) library that allows you to write code that loads an existing trained [TensorFlow](#) model file.



Multiclass classification

After using the TensorFlow inception model to extract features suitable as input for a classical machine learning algorithm, we add an ML.NET [multi-class classifier](#).

The specific trainer used in this case is the [multinomial logistic regression algorithm](#).

The algorithm implemented by this trainer performs well on problems with a large number of features, which is the case for a deep learning model operating on image data.

See [Deep learning vs. machine learning](#) for more information.

Data

There are two data sources: the `.tsv` file, and the image files. The `tags.tsv` file contains two columns: the first one is defined as `ImagePath` and the second one is the `Label` corresponding to the image. The following example file doesn't have a header row, and looks like this:

```
broccoli.jpg food
pizza.jpg food
pizza2.jpg food
teddy2.jpg toy
teddy3.jpg toy
teddy4.jpg toy
toaster.jpg appliance
toaster2.png appliance
```

The training and testing images are located in the assets folders that you'll download in a zip file. These images belong to Wikimedia Commons.

Wikimedia Commons, the free media repository. Retrieved 10:48, October 17, 2018 from:
<https://commons.wikimedia.org/wiki/Pizza> <https://commons.wikimedia.org/wiki/Toaster>
https://commons.wikimedia.org/wiki/Teddy_bear

Setup

Create a project

1. Create a C# Console Application called "TransferLearningTF". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Install the **Microsoft.ML** NuGet Package:

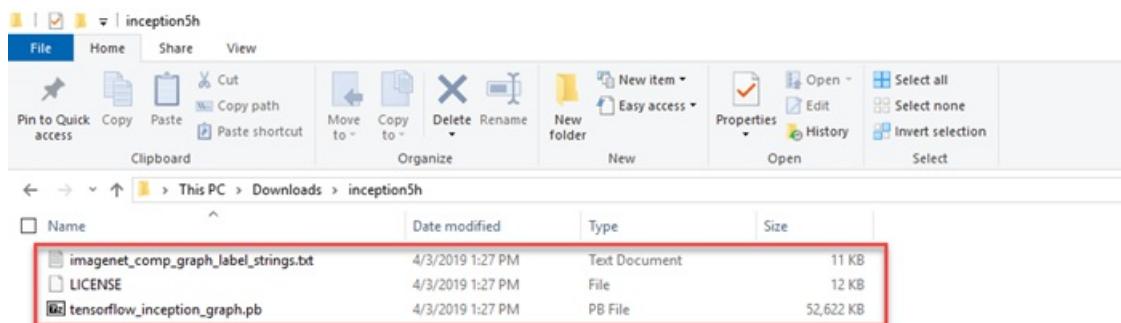
NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

- In Solution Explorer, right-click on your project and select **Manage NuGet Packages**.
- Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.ML**.
- Select the **Install** button.
- Select the **OK** button on the **Preview Changes** dialog.
- Select the **I Accept** button on the **License Acceptance** dialog if you agree with the license terms for the packages listed.
- Repeat these steps for **Microsoft.ML.ImageAnalytics**, **SciSharp.TensorFlow.Redist**, and **Microsoft.ML.TensorFlow**.

Download assets

1. Download [The project assets directory zip file](#), and unzip.
2. Copy the `assets` directory into your *TransferLearningTF* project directory. This directory and its subdirectories contain the data and support files (except for the Inception model, which you'll download and add in the next step) needed for this tutorial.
3. Download the [Inception model](#), and unzip.
4. Copy the contents of the `inception5h` directory just unzipped into your *TransferLearningTF* project `assets/inception` directory. This directory contains the model and additional support files needed for this tutorial, as shown in the following image:



5. In Solution Explorer, right-click each of the files in the asset directory and subdirectories and select **Properties**. Under **Advanced**, 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 Microsoft.ML;
using Microsoft.ML.Data;
```

2. Add the following code to the line right below the `using` statements to specify the asset paths:

```

string _assetsPath = Path.Combine(Environment.CurrentDirectory, "assets");
string _imagesFolder = Path.Combine(_assetsPath, "images");
string _trainTagsTsv = Path.Combine(_imagesFolder, "tags.tsv");
string _testTagsTsv = Path.Combine(_imagesFolder, "test-tags.tsv");
string _predictSingleImage = Path.Combine(_imagesFolder, "toaster3.jpg");
string _inceptionTensorFlowModel = Path.Combine(_assetsPath, "inception",
"tensorflow_inception_graph.pb");

```

3. Create classes for your input data, and predictions.

```

public class ImageData
{
    [LoadColumn(0)]
    public string ImagePath;

    [LoadColumn(1)]
    public string Label;
}

```

`ImageData` is the input image data class and has the following `String` fields:

- `ImagePath` contains the image file name.
- `Label` contains a value for the image label.

4. Add a new class to your project for `ImagePrediction`:

```

public class ImagePrediction : ImageData
{
    public float[] Score;

    public string PredictedLabelValue;
}

```

`ImagePrediction` is the image prediction class and has the following fields:

- `Score` contains the confidence percentage for a given image classification.
- `PredictedLabelValue` contains a value for the predicted image classification label.

`ImagePrediction` is the class used for prediction after the model has been trained. It has a `string (ImagePath)` for the image path. The `Label` is used to reuse and train the model. The `PredictedLabelValue` is used during prediction and evaluation. For evaluation, an input with training data, the predicted values, and the model are used.

Initialize variables

1. Initialize the `mlContext` variable with a new instance of `MLContext`. Replace the `Console.WriteLine("Hello World!")` line with the following code:

```
MLContext mlContext = new MLContext();
```

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

Create a struct for Inception model parameters

1. The Inception model has several parameters you need to pass in. Create a struct to map the parameter values to friendly names with the following code, just after initializing the `mlContext` variable:

```

struct InceptionSettings
{
    public const int ImageHeight = 224;
    public const int ImageWidth = 224;
    public const float Mean = 117;
    public const float Scale = 1;
    public const bool ChannelsLast = true;
}

```

Create a display utility method

Since you'll display the image data and the related predictions more than once, create a display utility method to handle displaying the image and prediction results.

1. Create the `DisplayResults()` method, just after the `InceptionSettings` struct, using the following code:

```

void DisplayResults(IEnumerable<ImagePrediction> imagePredictionData)
{
}

```

2. Fill in the body of the `DisplayResults` method:

```

foreach (ImagePrediction prediction in imagePredictionData)
{
    Console.WriteLine($"Image: {Path.GetFileName(prediction.ImagePath)} predicted as:
{prediction.PredictedLabelValue} with score: {prediction.Score.Max()} ");
}

```

Create a method to make a prediction

1. Create the `ClassifySingleImage()` method, just before the `DisplayResults()` method, using the following code:

```

void ClassifySingleImage(MLContext mlContext, ITransformer model)
{
}

```

2. Create an `ImageData` object that contains the fully qualified path and image file name for the single `ImagePath`. Add the following code as the next lines in the `ClassifySingleImage()` method:

```

var imageData = new ImageData()
{
    ImagePath = _predictSingleImage
};

```

3. Make a single prediction, by adding the following code as the next line in the `ClassifySingleImage` method:

```

// Make prediction function (input = ImageData, output = ImagePrediction)
var predictor = mlContext.Model.CreatePredictionEngine<ImageData, ImagePrediction>(model);
var prediction = predictor.Predict(imageData);

```

To get the prediction, use the `Predict()` method. 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](#).

NOTE

`PredictionEnginePool` service extension is currently in preview.

4. Display the prediction result as the next line of code in the `ClassifySingleImage()` method:

```
Console.WriteLine($"Image: {Path.GetFileName(imageData.ImagePath)} predicted as:  
{prediction.PredictedLabelValue} with score: {prediction.Score.Max()}");
```

Construct the ML.NET model pipeline

An ML.NET model pipeline is a chain of estimators. No execution happens during pipeline construction. The estimator objects are created but not executed.

1. Add a method to generate the model

This method is the heart of the tutorial. It creates a pipeline for the model, and trains the pipeline to produce the ML.NET model. It also evaluates the model against some previously unseen test data.

Create the `GenerateModel()` method, just after the `InceptionSettings` struct and just before the `DisplayResults()` method, using the following code:

```
ITransformer GenerateModel(MLContext mlContext)  
{  
}
```

2. Add the estimators to load, resize, and extract the pixels from the image data:

```
IEstimator<ITransformer> pipeline = mlContext.Transforms.LoadImages(outputColumnName: "input",  
imageFolder: _imagesFolder, inputColumnName: nameof(ImageData.ImagePath))  
    // The image transforms transform the images into the model's expected format.  
    .Append(mlContext.Transforms.ResizeImages(outputColumnName: "input", imageWidth:  
InceptionSettings.ImageWidth, imageHeight: InceptionSettings.ImageHeight, inputColumnName: "input"))  
    .Append(mlContext.Transforms.ExtractPixels(outputColumnName: "input",  
interleavePixelColors: InceptionSettings.ChannelsLast, offsetImage: InceptionSettings.Mean))
```

The image data needs to be processed into the format that the TensorFlow model expects. In this case, the images are loaded into memory, resized to a consistent size, and the pixels are extracted into a numeric vector.

3. Add the estimator to load the TensorFlow model, and score it:

```
.Append(mlContext.Model.LoadTensorFlowModel(_inceptionTensorFlowModel)).  
    ScoreTensorFlowModel(outputColumnNames: new[] { "softmax2_pre_activation" }, inputColumnNames:  
new[] { "input" }, addBatchDimensionInput: true))
```

This stage in the pipeline loads the TensorFlow model into memory, then processes the vector of pixel values through the TensorFlow model network. Applying inputs to a deep learning model, and generating

an output using the model, is referred to as **Scoring**. When using the model in its entirety, scoring makes an inference, or prediction.

In this case, you use all of the TensorFlow model except the last layer, which is the layer that makes the inference. The output of the penultimate layer is labeled `softmax_2_preactivation`. The output of this layer is effectively a vector of features that characterize the original input images.

This feature vector generated by the TensorFlow model will be used as input to an ML.NET training algorithm.

4. Add the estimator to map the string labels in the training data to integer key values:

```
.Append(mlContext.Transforms.Conversion.MapValueToKey(outputColumnName: "LabelKey", inputColumnName: "Label"))
```

The ML.NET trainer that is appended next requires its labels to be in `key` format rather than arbitrary strings. A key is a number that has a one to one mapping to a string value.

5. Add the ML.NET training algorithm:

```
.Append(mlContext.MulticlassClassification.Trainers.LbfgsMaximumEntropy(labelColumnName: "LabelKey", featureColumnName: "softmax2_pre_activation"))
```

6. Add the estimator to map the predicted key value back into a string:

```
.Append(mlContext.Transforms.Conversion.MapKeyToValue("PredictedLabelValue", "PredictedLabel"))
.AppendCacheCheckpoint(mlContext);
```

Train the model

1. Load the training data using the `LoadFromTextFile` wrapper. Add the following code as the next line in the `GenerateModel()` method:

```
IDataView trainingData = mlContext.Data.LoadFromTextFile<ImageData>(path: _trainTagsTsv, hasHeader: false);
```

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.

2. Train the model with the data loaded above:

```
ITransformer model = pipeline.Fit(trainingData);
```

The `Fit()` method trains your model by applying the training dataset to the pipeline.

Evaluate the accuracy of the model

1. Load and transform the test data, by adding the following code to the next line of the `GenerateModel` method:

```

IDataView testData = mlContext.Data.LoadFromTextFile<ImageData>(path: _testTagsTsv, hasHeader:
false);
IDataView predictions = model.Transform(testData);

// Create an IEnumerable for the predictions for displaying results
IEnumerable<ImagePrediction> imagePredictionData = mlContext.Data.CreateEnumerable<ImagePrediction>
(predictions, true);
DisplayResults(imagePredictionData);

```

There are a few sample images that you can use to evaluate the model. Like the training data, these need to be loaded into an `IDataView`, so that they can be transformed by the model.

2. Add the following code to the `GenerateModel()` method to evaluate the model:

```

MulticlassClassificationMetrics metrics =
    mlContext.MulticlassClassification.Evaluate(predictions,
        labelColumnName: "LabelKey",
        predictedLabelColumnName: "PredictedLabel");

```

Once you have the prediction set, the `Evaluate()` method:

- Assesses the model (compares the predicted values with the test dataset `labels`).
- Returns the model performance metrics.

3. Display the model accuracy metrics

Use the following code to display the metrics, share the results, and then act on them:

```

Console.WriteLine($"LogLoss is: {metrics.LogLoss}");
Console.WriteLine($"PerClassLogLoss is: {String.Join(" , ", metrics.PerClassLogLoss.Select(c =>
c.ToString()))}");

```

The following metrics are evaluated for image classification:

- `Log-loss` - see [Log Loss](#). You want Log-loss to be as close to zero as possible.
- `Per class Log-loss` . You want per class Log-loss to be as close to zero as possible.

4. Add the following code to return the trained model as the next line:

```

return model;

```

Run the application!

1. Add the call to `GenerateModel` after the creation of the `MLContext` class:

```

ITransformer model = GenerateModel(mlContext);

```

2. Add the call to the `ClassifySingleImage()` method after the call to the `GenerateModel()` method:

```

ClassifySingleImage(mlContext, model);

```

3. Run your console app (Ctrl + F5). Your results should be similar to the following output. You may see warnings or processing messages, but these messages have been removed from the following results for clarity.

```
===== Training classification model =====
Image: broccoli2.jpg predicted as: food with score: 0.8955513
Image: pizza3.jpg predicted as: food with score: 0.9667718
Image: teddy6.jpg predicted as: toy with score: 0.9797683
===== Classification metrics =====
LogLoss is: 0.0653774699265059
PerClassLogLoss is: 0.110315812569315 , 0.0204391272836966 , 0
===== Making single image classification =====
Image: toaster3.jpg predicted as: appliance with score: 0.9646884
```

Congratulations! You've now successfully built a classification model in ML.NET to categorize images by using a pre-trained TensorFlow for image processing.

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

In this tutorial, you learned how to:

- Understand the problem
- Incorporate the pre-trained TensorFlow model into the ML.NET pipeline
- Train and evaluate the ML.NET model
- Classify a test image

Check out the Machine Learning samples GitHub repository to explore an expanded image classification sample.

[dotnet/machinelearning-samples](#) GitHub repository

Tutorial: Forecast bike rental service demand with time series analysis and ML.NET

10/14/2022 • 9 minutes to read • [Edit Online](#)

Learn how to forecast demand for a bike rental service using univariate time series analysis on data stored in a SQL Server database with ML.NET.

In this tutorial, you learn how to:

- Understand the problem
- Load data from a database
- Create a forecasting model
- Evaluate forecasting model
- Save a forecasting model
- Use a forecasting model

Prerequisites

- [Visual Studio 2022](#) with the ".NET Desktop Development" workload installed.

Time series forecasting sample overview

This sample is a **C# .NET Core console application** that forecasts demand for bike rentals using a univariate time series analysis algorithm known as Singular Spectrum Analysis. The code for this sample can be found on the [dotnet/machinelearning-samples](#) repository on GitHub.

Understand the problem

In order to run an efficient operation, inventory management plays a key role. Having too much of a product in stock means unsold products sitting on the shelves not generating any revenue. Having too little product leads to lost sales and customers purchasing from competitors. Therefore, the constant question is, what is the optimal amount of inventory to keep on hand? Time-series analysis helps provide an answer to these questions by looking at historical data, identifying patterns, and using this information to forecast values some time in the future.

The technique for analyzing data used in this tutorial is univariate time-series analysis. Univariate time-series analysis takes a look at a single numerical observation over a period of time at specific intervals such as monthly sales.

The algorithm used in this tutorial is [Singular Spectrum Analysis\(SSA\)](#). SSA works by decomposing a time-series into a set of principal components. These components can be interpreted as the parts of a signal that correspond to trends, noise, seasonality, and many other factors. Then, these components are reconstructed and used to forecast values some time in the future.

Create console application

1. Create a **C# Console Application** called "BikeDemandForecasting". Click the **Next** button.
2. Choose **.NET 6** as the framework to use. Click the **Create** button.

3. Install Microsoft.ML version NuGet package

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

- a. In Solution Explorer, right-click on your project and select **Manage NuGet Packages**.
- b. Choose "nuget.org" as the Package source, select the **Browse** tab, search for **Microsoft.ML**.
- c. Check the **Include prerelease** checkbox.
- d. Select the **Install** button.
- e. 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.
- f. Repeat these steps for **System.Data.SqlClient** and **Microsoft.ML.TimeSeries**.

Prepare and understand the data

1. Create a directory called *Data*.
2. Download the [DailyDemand.mdf database file](#) and save it to the *Data* directory.

NOTE

The data used in this tutorial comes from the [UCI Bike Sharing Dataset](#). Fanaee-T, Hadi, and Gama, Joao, 'Event labeling combining ensemble detectors and background knowledge', Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg, [Web Link](#).

The original dataset contains several columns corresponding to seasonality and weather. For brevity and because the algorithm used in this tutorial only requires the values from a single numerical column, the original dataset has been condensed to include only the following columns:

- **dteday**: The date of the observation.
- **year**: The encoded year of the observation (0=2011, 1=2012).
- **cnt**: The total number of bike rentals for that day.

The original dataset is mapped to a database table with the following schema in a SQL Server database.

```
CREATE TABLE [Rentals] (
    [RentalDate] DATE NOT NULL,
    [Year] INT NOT NULL,
    [TotalRentals] INT NOT NULL
);
```

The following is a sample of the data:

RENTALDATE	YEAR	TOTALRENTALS
1/1/2011	0	985
1/2/2011	0	801
1/3/2011	0	1349

Create input and output classes

1. Open *Program.cs* file and replace the existing `using` statements with the following:

```
using Microsoft.ML;
using Microsoft.ML.Data;
using Microsoft.ML.Transforms.TimeSeries;
using System.Data.SqlClient;
```

2. Create `ModelInput` class. Below the `Program` class, add the following code.

```
public class ModelInput
{
    public DateTime RentalDate { get; set; }

    public float Year { get; set; }

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

The `ModelInput` class contains the following columns:

- **RentalDate**: The date of the observation.
- **Year**: The encoded year of the observation (0=2011, 1=2012).
- **TotalRentals**: The total number of bike rentals for that day.

3. Create `ModelOutput` class below the newly created `ModelInput` class.

```
public class ModelOutput
{
    public float[] ForecastedRentals { get; set; }

    public float[] LowerBoundRentals { get; set; }

    public float[] UpperBoundRentals { get; set; }
}
```

The `ModelOutput` class contains the following columns:

- **ForecastedRentals**: The predicted values for the forecasted period.
- **LowerBoundRentals**: The predicted minimum values for the forecasted period.
- **UpperBoundRentals**: The predicted maximum values for the forecasted period.

Define paths and initialize variables

1. Below the using statements define variables to store the location of your data, connection string, and where to save the trained model.

```
string rootDir = Path.GetFullPath(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "../../../../../"));
string dbFilePath = Path.Combine(rootDir, "Data", "DailyDemand.mdf");
string modelPath = Path.Combine(rootDir, "MLModel.zip");
var connectionString = $"Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename={dbFilePath};Integrated Security=True;Connect Timeout=30;"
```

2. Initialize the `mlContext` variable with a new instance of `MLContext` by adding the following line after defining the paths.

```
MLContext mlContext = new MLContext();
```

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

Load the data

1. Create `DatabaseLoader` that loads records of type `ModelInput`.

```
DatabaseLoader loader = mlContext.Data.CreateDatabaseLoader<ModelInput>();
```

2. Define the query to load the data from the database.

```
string query = "SELECT RentalDate, CAST(Year as REAL) as Year, CAST(TotalRentals as REAL) as TotalRentals FROM Rentals";
```

ML.NET algorithms expect data to be of type `Single`. Therefore, numerical values coming from the database that are not of type `Real`, a single-precision floating-point value, have to be converted to `Real`.

The `Year` and `TotalRental` columns are both integer types in the database. Using the `CAST` built-in function, they are both cast to `Real`.

3. Create a `DataSource` to connect to the database and execute the query.

```
DataSource dbSource = new DataSource(SqlClientFactory.Instance,
                                      connectionString,
                                      query);
```

4. Load the data into an `IDataView`.

```
IDataView dataView = loader.Load(dbSource);
```

5. The dataset contains two years worth of data. Only data from the first year is used for training, the second year is held out to compare the actual values against the forecast produced by the model. Filter the data using the `FilterRowsByColumn` transform.

```
IDataView firstYearData = mlContext.Data.FilterRowsByColumn(dataView, "Year", upperBound: 1);
IDataView secondYearData = mlContext.Data.FilterRowsByColumn(dataView, "Year", lowerBound: 1);
```

For the first year, only the values in the `Year` column less than 1 are selected by setting the `upperBound` parameter to 1. Conversely, for the second year, values greater than or equal to 1 are selected by setting the `lowerBound` parameter to 1.

Define time series analysis pipeline

1. Define a pipeline that uses the `SsaForecastingEstimator` to forecast values in a time-series dataset.

```
var forecastingPipeline = mlContext.Forecasting.ForecastBySsa(
    outputColumnName: "ForecastedRentals",
    inputColumnName: "TotalRentals",
    windowSize: 7,
    seriesLength: 30,
    trainSize: 365,
    horizon: 7,
    confidenceLevel: 0.95f,
    confidenceLowerBoundColumn: "LowerBoundRentals",
    confidenceUpperBoundColumn: "UpperBoundRentals");
```

The `forecastingPipeline` takes 365 data points for the first year and samples or splits the time-series dataset into 30-day (monthly) intervals as specified by the `seriesLength` parameter. Each of these samples is analyzed through weekly or a 7-day window. When determining what the forecasted value for the next period(s) is, the values from previous seven days are used to make a prediction. The model is set to forecast seven periods into the future as defined by the `horizon` parameter. Because a forecast is an informed guess, it's not always 100% accurate. Therefore, it's good to know the range of values in the best and worst-case scenarios as defined by the upper and lower bounds. In this case, the level of confidence for the lower and upper bounds is set to 95%. The confidence level can be increased or decreased accordingly. The higher the value, the wider the range is between the upper and lower bounds to achieve the desired level of confidence.

2. Use the `Fit` method to train the model and fit the data to the previously defined `forecastingPipeline`.

```
SsaForecastingTransformer forecaster = forecastingPipeline.Fit(firstYearData);
```

Evaluate the model

Evaluate how well the model performs by forecasting next year's data and comparing it against the actual values.

1. Create a new utility method called `Evaluate` at the bottom of the `Program.cs` file.

```
Evaluate(IDataView testData, ITransformer model, MLContext mlContext)
{
}
```

2. Inside the `Evaluate` method, forecast the second year's data by using the `Transform` method with the trained model.

```
IDataView predictions = model.Transform(testData);
```

3. Get the actual values from the data by using the `CreateEnumerable` method.

```
IEnumerable<float> actual =
    mlContext.Data.CreateEnumerable<ModelInput>(testData, true)
        .Select(observed => observed.TotalRentals);
```

4. Get the forecast values by using the `CreateEnumerable` method.

```
IEnumerable<float> forecast =
    mlContext.Data.CreateEnumerable<ModelOutput>(predictions, true)
        .Select(prediction => prediction.ForecastedRentals[0]);
```

5. Calculate the difference between the actual and forecast values, commonly referred to as the error.

```
var metrics = actual.Zip(forecast, (actualValue, forecastValue) => actualValue - forecastValue);
```

6. Measure performance by computing the Mean Absolute Error and Root Mean Squared Error values.

```
var MAE = metrics.Average(error => Math.Abs(error)); // Mean Absolute Error
var RMSE = Math.Sqrt(metrics.Average(error => Math.Pow(error, 2))); // Root Mean Squared Error
```

To evaluate performance, the following metrics are used:

- **Mean Absolute Error:** Measures how close predictions are to the actual value. This value ranges between 0 and infinity. The closer to 0, the better the quality of the model.
- **Root Mean Squared Error:** Summarizes the error in the model. This value ranges between 0 and infinity. The closer to 0, the better the quality of the model.

7. Output the metrics to the console.

```
Console.WriteLine("Evaluation Metrics");
Console.WriteLine("-----");
Console.WriteLine($"Mean Absolute Error: {MAE:F3}");
Console.WriteLine($"Root Mean Squared Error: {RMSE:F3}\n");
```

8. Call the `Evaluate` method below calling the `Fit()` method.

```
Evaluate(secondYearData, forecaster, mlContext);
```

Save the model

If you're satisfied with your model, save it for later use in other applications.

1. Below the `Evaluate()` method create a `TimeSeriesPredictionEngine`. `TimeSeriesPredictionEngine` is a convenience method to make single predictions.

```
var forecastEngine = forecaster.CreateTimeSeriesEngine<ModelInput, ModelOutput>(mlContext);
```

2. Save the model to a file called `MLModel.zip` as specified by the previously defined `modelPath` variable. Use the `CheckPoint` method to save the model.

```
forecastEngine.CheckPoint(mlContext, modelPath);
```

Use the model to forecast demand

1. Below the `Evaluate` method, create a new utility method called `Forecast`.

```
void Forecast(IDataView testData, int horizon, TimeSeriesPredictionEngine<ModelInput, ModelOutput>
forecaster, MLContext mlContext)
{
}
```

2. Inside the `Forecast` method, use the `Predict` method to forecast rentals for the next seven days.

```
    ModelOutput forecast = forecaster.Predict();
```

3. Align the actual and forecast values for seven periods.

```
IEnumerable<string> forecastOutput =
    mlContext.Data.CreateEnumerable<ModelInput>(testData, reuseRowObject: false)
        .Take(horizon)
        .Select((ModelInput rental, int index) =>
    {
        string rentalDate = rental.RentalDate.ToShortDateString();
        float actualRentals = rental.TotalRentals;
        float lowerEstimate = Math.Max(0, forecast.LowerBoundRentals[index]);
        float estimate = forecast.ForecastedRentals[index];
        float upperEstimate = forecast.UpperBoundRentals[index];
        return $"Date: {rentalDate}\n" +
            $"Actual Rentals: {actualRentals}\n" +
            $"Lower Estimate: {lowerEstimate}\n" +
            $"Forecast: {estimate}\n" +
            $"Upper Estimate: {upperEstimate}\n";
    });
});
```

4. Iterate through the forecast output and display it on the console.

```
Console.WriteLine("Rental Forecast");
Console.WriteLine("-----");
foreach (var prediction in forecastOutput)
{
    Console.WriteLine(prediction);
}
```

Run the application

1. Below calling the `Checkpoint()` method call the `Forecast` method.

```
Forecast(secondYearData, 7, forecastEngine, mlContext);
```

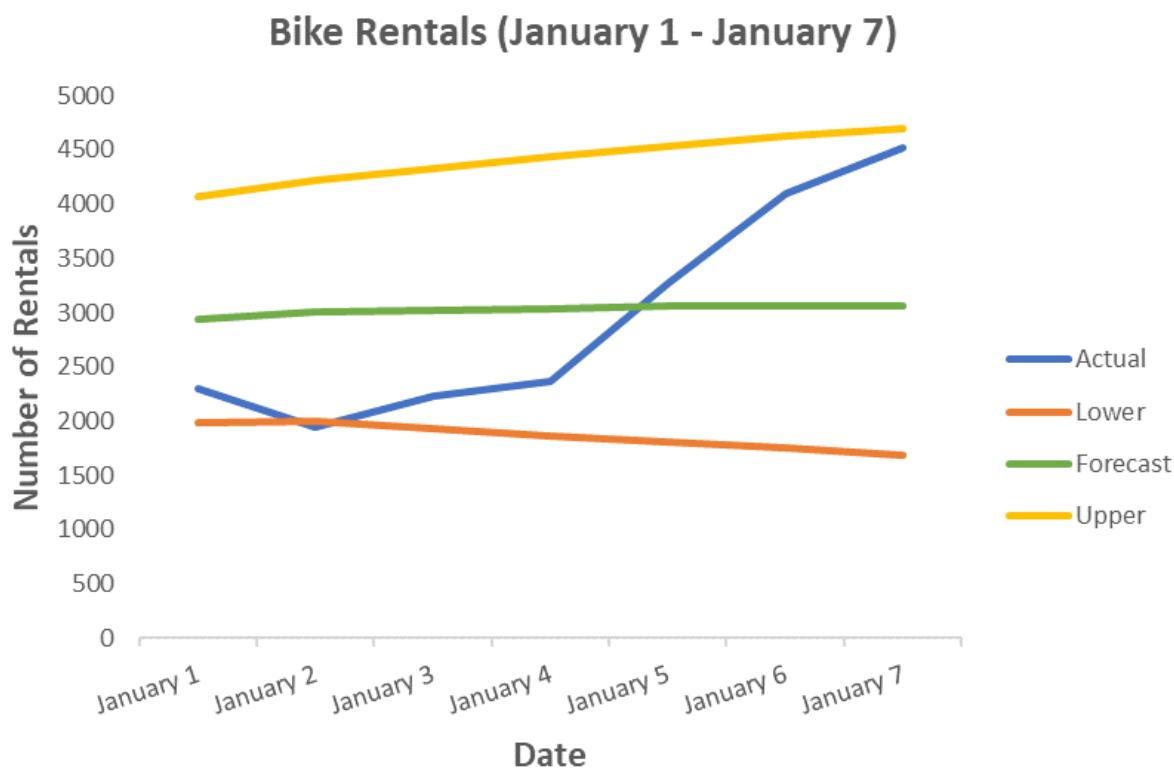
2. Run the application. Output similar to that below should appear on the console. For brevity, the output has been condensed.

```
Evaluation Metrics
-----
Mean Absolute Error: 726.416
Root Mean Squared Error: 987.658
```

```
Rental Forecast
-----
Date: 1/1/2012
Actual Rentals: 2294
Lower Estimate: 1197.842
Forecast: 2334.443
Upper Estimate: 3471.044

Date: 1/2/2012
Actual Rentals: 1951
Lower Estimate: 1148.412
Forecast: 2360.861
Upper Estimate: 3573.309
```

Inspection of the actual and forecasted values shows the following relationships:



While the forecasted values are not predicting the exact number of rentals, they provide a more narrow range of values that allows an operation to optimize their use of resources.

Congratulations! You've now successfully built a time series machine learning model to forecast bike rental demand.

You can find the source code for this tutorial at the [dotnet/machinelearning-samples](#) repository.

Next steps

- [Machine learning tasks in ML.NET](#)
- [Improve model accuracy](#)

Tutorial: Detect anomalies in time series with ML.NET

10/14/2022 • 7 minutes to read • [Edit Online](#)

Learn how to build an anomaly detection application for time series data. This tutorial creates a .NET Core console application using C# in Visual Studio 2019.

In this tutorial, you learn how to:

- Load the data
- Detect period for a time series
- Detect anomaly for a periodical time series

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Prerequisites

- [Visual Studio 2022](#) with the ".NET Desktop Development" workload installed.
- [The phone-calls.csv dataset](#).

Create a console application

1. Create a C# Console Application called "PhoneCallsAnomalyDetection". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to save your data set files.
4. Install the **Microsoft.ML NuGet Package** version 1.5.2:
 - a. In Solution Explorer, right-click on your project and select **Manage NuGet Packages**.
 - b. Choose "nuget.org" as the Package source.
 - c. Select the **Browse** tab.
 - d. Search for **Microsoft.ML**.
 - e. Select **Microsoft.ML** from the list of packages and choose version 1.5.2 from the **Version** dropdown.
 - f. Select the **Install** button.
 - g. 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.

Repeat these steps for **Microsoft.ML.TimeSeries** version 1.5.2.

5. Add the following `using` statements at the top of your *Program.cs* file:

```
using Microsoft.ML;
using Microsoft.ML.TimeSeries;
using PhoneCallsAnomalyDetection;
```

Download your data

1. Download the dataset and save it to the *Data* folder you previously created:

Right click on [phone-calls.csv](#) and select "Save Link (or Target) As..."

Make sure you either save the *.csv file to the *Data* folder, or after you save it elsewhere, move the *.csv file to the *Data* folder.

2. In Solution Explorer, right-click the *.csv file and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

The following table is a data preview from your *.csv file:

TIMESTAMP	VALUE
2018/9/3	36.69670857
2018/9/4	35.74160571
.....
2018/10/3	34.49893429
...	...

This file represents a time-series. Each row in the file is a data point. Each data point has two attributes, namely, `timestamp` and `value`, to represent the number of phone calls at each day. The number of phone calls is transformed to de-sensitivity.

Create classes and define paths

Next, define your input and prediction class data structures.

Add a new class to your project:

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 *PhoneCallsData.cs*. Then, select the **Add** button.

The *PhoneCallsData.cs* file opens in the code editor.

3. Add the following `using` statement to the top of *PhoneCallsData.cs*:

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

4. Remove the existing class definition and add the following code, which has two classes `PhoneCallsData` and `PhoneCallsPrediction`, to the *PhoneCallsData.cs* file:

```

public class PhoneCallsData
{
    [LoadColumn(0)]
    public string timestamp;

    [LoadColumn(1)]
    public double value;
}

public class PhoneCallsPrediction
{
    //vector to hold anomaly detection results. Including isAnomaly, anomalyScore, magnitude,
    //expectedValue, boundaryUnits, upperBoundary and lowerBoundary.
    [VectorType(7)]
    public double[] Prediction { get; set; }
}

```

`PhoneCallsData` specifies an input data class. The `LoadColumn` attribute specifies which columns (by column index) in the dataset should be loaded. It has two attributes `timestamp` and `value` that correspond to the same attributes in the data file.

`PhoneCallsPrediction` specifies the prediction data class. For SR-CNN detector, the prediction depends on the `detect mode` specified. In this sample, we select the `AnomalyAndMargin` mode. The output contains seven columns. In most cases, `IsAnomaly`, `ExpectedValue`, `UpperBoundary`, and `LowerBoundary` are informative enough. They tell you if a point is an anomaly, the expected value of the point and the lower / upper boundary region of the point.

- Add the following code to the line right below the using statements to specify the path to your data file:

```
string _dataPath = Path.Combine(Environment.CurrentDirectory, "Data", "phone-calls.csv");
```

Initialize variables

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

```
MLContext mlContext = new MLContext();
```

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

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 from other sources (for example, SQL database or log files) to an `IDataView` object.

- Add the following code below the creation of the `mlContext` variable:

```
IDataView dataView = mlContext.Data.LoadFromTextFile<PhoneCallsData>(path: _dataPath, hasHeader: true, separatorChar: ',');
```

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

Time series anomaly detection

Time series anomaly detection is the process of detecting time-series data outliers; points on a given input time-series where the behavior isn't what was expected, or "weird". These anomalies are typically indicative of some events of interest in the problem domain: a cyber-attack on user accounts, power outage, bursting RPS on a server, memory leak, etc.

To find anomaly on time series, you should first determine the period of the series. Then, the time series can be decomposed into several components as $Y = T + S + R$, where Y is the original series, T is the trend component, S is the seasonal component, and R is the residual component of the series. This step is called **decomposition**. Finally, detection is performed on the residual component to find the anomalies. In ML.NET, The SR-CNN algorithm is an advanced and novel algorithm that is based on Spectral Residual (SR) and Convolutional Neural Network(CNN) to detect anomaly on time-series. For more information on this algorithm, see [Time-Series Anomaly Detection Service at Microsoft](#).

In this tutorial, you will see that these procedures can be completed using two functions.

Detect Period

In the first step, we invoke the `DetectSeasonality` function to determine the period of the series.

Create the DetectPeriod method

1. Create the `DetectPeriod` method at the bottom of the `Program.cs` file using the following code:

```
int DetectPeriod(MLContext mlContext, IDataView phoneCalls)
{
}
```

2. Use the `DetectSeasonality` function to detect period. Add it to the `DetectPeriod` method with the following code:

```
int period = mlContext.AnomalyDetection.DetectSeasonality(phoneCalls, nameof(PhoneCallsData.value));
```

3. Display the period value by adding the following as the next line of code in the `DetectPeriod` method:

```
Console.WriteLine("Period of the series is: {0}.", period);
```

4. Return the period value.

```
// <SnippetSetupSrCnnParameters>
```

5. Add the following call to the `DetectPeriod` method below the call to the `LoadFromFile()` method:

```
int period = DetectPeriod(mlContext, dataView);
```

Period detection results

Run the application. Your results should be similar to the following.

```
Period of the series is: 7.
```

Detect Anomaly

In this step, you use the [DetectEntireAnomalyBySrCnn](#) method to find anomalies.

Create the DetectAnomaly method

1. Create the `DetectAnomaly` method, just below the `DetectPeriod` method, using the following code:

```
void DetectAnomaly(MLContext mlContext, IDataView phoneCalls, int period)
{
}
```

2. Set up `SrCnnEntireAnomalyDetectorOptions` in the `DetectAnomaly` method with the following code:

```
var options = new SrCnnEntireAnomalyDetectorOptions()
{
    Threshold = 0.3,
    Sensitivity = 64.0,
    DetectMode = SrCnnDetectMode.AnomalyAndMargin,
    Period = period,
};
```

3. Detect anomaly by SR-CNN algorithm by adding the following line of code in the `DetectAnomaly` method:

```
var output DataView = mlContext.AnomalyDetection.DetectEntireAnomalyBySrCnn(phoneCalls,
    nameof(PhoneCallsPrediction.Prediction), nameof(PhoneCallsData.value), options);
```

4. Convert the output data view into a strongly typed `IEnumerable` for easier display using the `CreateEnumerable` method with the following code:

```
var predictions = mlContext.Data.CreateEnumerable<PhoneCallsPrediction>(
    output DataView, reuseRowObject: false);
```

5. Create a display header with the following code as the next line in the `DetectAnomaly` method:

```
Console.WriteLine("Index,Data,Anomaly,AnomalyScore,Mag,ExpectedValue,BoundaryUnit,UpperBoundary,Lower
Boundary");
```

You'll display the following information in your change point detection results:

- `Index` is the index of each point.
- `Anomaly` is the indicator of whether each point is detected as anomaly.
- `ExpectedValue` is the estimated value of each point.
- `LowerBoundary` is the lowest value each point can be to be not anomaly.
- `UpperBoundary` is the highest value each point can be to be not anomaly.

6. Iterate through the `predictions` `IEnumerable` and display the results with the following code:

```
var index = 0;

foreach (var p in predictions)
{
    if (p.Prediction[0] == 1)
    {
        Console.WriteLine("{0},{1},{2},{3},{4},  -- alert is on! detected anomaly", index,
                          p.Prediction[0], p.Prediction[3], p.Prediction[5], p.Prediction[6]);
    }
    else
    {
        Console.WriteLine("{0},{1},{2},{3},{4}", index,
                          p.Prediction[0], p.Prediction[3], p.Prediction[5], p.Prediction[6]);
    }
    ++index;
}

Console.WriteLine("");
```

7. Add the following call to the `DetectAnomaly` method below the `DetectPeriod()` method call:

```
DetectAnomaly(mlContext, dataView, period);
```

Anomaly detection results

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

```

Detect period of the series
Period of the series is: 7.
Detect anomaly points in the series
Index Data Anomaly AnomalyScore Mag ExpectedValue BoundaryUnit UpperBoundary
LowerBoundary
0,0,36.841787256739266,41.14206982401966,32.541504689458876
1,0,35.67303618137362,39.97331874865401,31.372753614093227
2,0,34.710132999891826,39.029491313022824,30.390774686760828
3,0,33.44765248883495,37.786086547816545,29.10921842985335
4,0,28.937110922276364,33.25646923540736,24.61775260914537
5,0,5.143895892785781,9.444178460066171,0.843613325505391
6,0,5.163325228419392,9.463607795699783,0.8630426611390014
7,0,36.76414836240396,41.06443092968435,32.46386579512357
8,0,35.77908590657007,40.07936847385046,31.478803339289676
9,0,34.547259536635245,38.847542103915636,30.246976969354854
10,0,33.55193524820608,37.871293561337076,29.23257693507508
11,0,29.091800129624648,33.392082696905035,24.79151756234426
12,0,5.154836630338823,9.455119197619213,0.8545540630584334
13,0,5.234332502492464,9.534615069772855,0.934049935212073
14,0,36.54992549471526,40.85020806199565,32.24964292743487
15,0,35.79526470980883,40.095547277089224,31.494982142528443
16,0,34.34099013096804,38.64127269824843,30.040707563687647
17,0,33.61201516582131,37.9122977331017,29.31173259854092
18,0,29.223563320561812,33.5238458878422,24.923280753281425
19,0,5.170512168851533,9.470794736131923,0.8702296015711433
20,0,5.2614938889462834,9.561776456226674,0.9612113216658926
21,0,36.37103858487317,40.67132115215356,32.07075601759278
22,0,35.813544599026855,40.113827166307246,31.513262031746464
23,0,34.05600492733225,38.356287494612644,29.755722360051863
24,0,33.65828319077884,37.95856575805923,29.358000623498448
25,0,29.381125690882463,33.681408258162854,25.080843123602072
26,0,5.261543539820418,9.561826107100808,0.9612609725400283
27,0,5.4873712582971805,9.787653825577571,1.1870886910167897
28,1,36.504694001629254,40.804976568909645,32.20441143434886 <-- alert is on, detected anomaly
...

```

Congratulations! You've now successfully built machine learning models for detecting period and anomaly on a periodical series.

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

In this tutorial, you learned how to:

- Load the data
- Detect period on the time series data
- Detect anomaly on the time series data

Next steps

Check out the Machine Learning samples GitHub repository to explore a Power Consumption Anomaly Detection sample.

[dotnet/machinelearning-samples](#) GitHub repository

Tutorial: Detect anomalies in product sales with ML.NET

10/14/2022 • 12 minutes to read • [Edit Online](#)

Learn how to build an anomaly detection application for product sales data. This tutorial creates a .NET Core console application using C# in Visual Studio.

In this tutorial, you learn how to:

- Load the data
- Create a transform for spike anomaly detection
- Detect spike anomalies with the transform
- Create a transform for change point anomaly detection
- Detect change point anomalies with the transform

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Prerequisites

- [Visual Studio 2022](#) with the ".NET Desktop Development" workload installed.
- [The product-sales.csv dataset](#)

NOTE

The data format in `product-sales.csv` is based on the dataset "Shampoo Sales Over a Three Year Period" originally sourced from DataMarket and provided by Time Series Data Library (TSDL), created by Rob Hyndman. "Shampoo Sales Over a Three Year Period" Dataset Licensed Under the DataMarket Default Open License.

Create a console application

1. Create a **C# Console Application** called "ProductSalesAnomalyDetection". Click the **Next** button.
2. Choose **.NET 6** as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to save your data set files.
4. Install the **Microsoft.ML NuGet Package**:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

In Solution Explorer, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.ML** 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. Repeat these steps for **Microsoft.ML.TimeSeries**.

5. Add the following `using` statements at the top of your *Program.cs* file:

```
using Microsoft.ML;
using ProductSalesAnomalyDetection;
```

Download your data

1. Download the dataset and save it to the *Data* folder you previously created:

- Right click on *product-sales.csv* and select "Save Link (or Target) As..."

Make sure you either save the *.csv file to the *Data* folder, or after you save it elsewhere, move the *.csv file to the *Data* folder.

2. In Solution Explorer, right-click the *.csv file and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

The following table is a data preview from your *.csv file:

MONTH	PRODUCTSALES
1-Jan	271
2-Jan	150.9
....
1-Feb	199.3
....

Create classes and define paths

Next, define your input and prediction class data structures.

Add a new class to your project:

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 *ProductSalesData.cs*. Then, select the **Add** button.

The *ProductSalesData.cs* file opens in the code editor.

3. Add the following `using` statement to the top of *ProductSalesData.cs*:

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

4. Remove the existing class definition and add the following code, which has two classes `ProductSalesData` and `ProductSalesPrediction`, to the *ProductSalesData.cs* file:

```

public class ProductSalesData
{
    [LoadColumn(0)]
    public string Month;

    [LoadColumn(1)]
    public float numSales;
}

public class ProductSalesPrediction
{
    //vector to hold alert,score,p-value values
    [VectorType(3)]
    public double[] Prediction { get; set; }
}

```

`ProductSalesData` specifies an input data class. The `LoadColumn` attribute specifies which columns (by column index) in the dataset should be loaded.

`ProductSalesPrediction` specifies the prediction data class. For anomaly detection, the prediction consists of an alert to indicate whether there is an anomaly, a raw score, and p-value. The closer the p-value is to 0, the more likely an anomaly has occurred.

5. Create two global fields to hold the recently downloaded dataset file path and the saved model file path:

- `_dataPath` has the path to the dataset used to train the model.
- `_docszie` has the number of records in dataset file. You'll use `_docSize` to calculate `pvalueHistoryLength`.

6. Add the following code to the line right below the using statements to specify those paths:

```

string _dataPath = Path.Combine(Environment.CurrentDirectory, "Data", "product-sales.csv");
//assign the Number of records in dataset file to constant variable
const int _docszie = 36;

```

Initialize variables

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

```
MLContext mlContext = new MLContext();
```

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

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 from other sources (for example, SQL database or log files) to an `IDataView` object.

1. Add the following code after creating the `mlContext` variable:

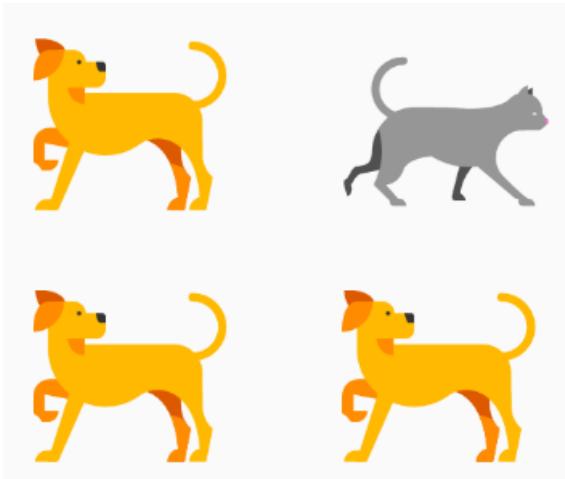
```
IDataView dataView = mlContext.Data.LoadFromTextFile<ProductSalesData>(path: _dataPath, hasHeader: true, separatorChar: ',');
```

The `LoadFromTextFile()` defines the data schema and reads in the file. It takes in the data path variables

and returns an `IDataView`.

Time series anomaly detection

Anomaly detection flags unexpected or unusual events or behaviors. It gives clues where to look for problems and helps you answer the question "Is this weird?".



Anomaly detection is the process of detecting time-series data outliers; points on a given input time-series where the behavior isn't what was expected, or "weird".

Anomaly detection can be useful in lots of ways. For instance:

If you have a car, you might want to know: Is this oil gauge reading normal, or do I have a leak? If you're monitoring power consumption, you'd want to know: Is there an outage?

There are two types of time series anomalies that can be detected:

- **Spikes** indicate temporary bursts of anomalous behavior in the system.
- **Change points** indicate the beginning of persistent changes over time in the system.

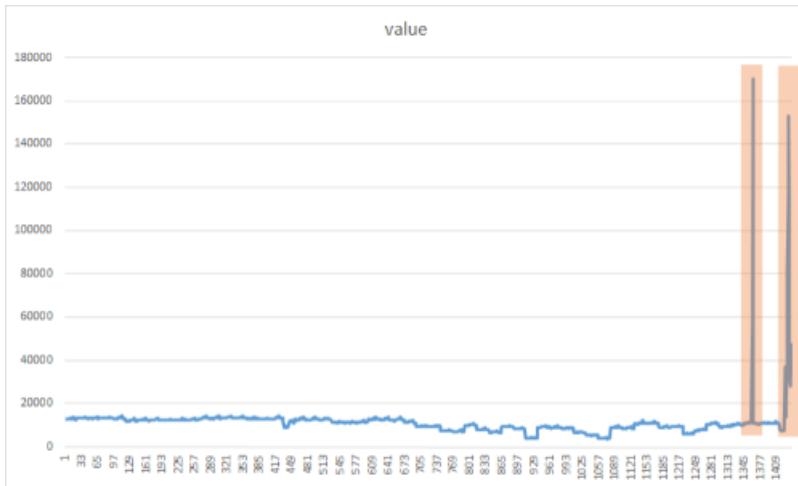
In ML.NET, The IID Spike Detection or IID Change point Detection algorithms are suited for [independent and identically distributed datasets](#). They assume that your input data is a sequence of data points that are independently sampled from [one stationary distribution](#).

Unlike the models in the other tutorials, the time series anomaly detector transforms operate directly on input data. The `IEstimator.Fit()` method does not need training data to produce the transform. It does need the data schema though, which is provided by a data view generated from an empty list of `ProductSalesData`.

You'll analyze the same product sales data to detect spikes and change points. The building and training model process is the same for spike detection and change point detection; the main difference is the specific detection algorithm used.

Spike detection

The goal of spike detection is to identify sudden yet temporary bursts that significantly differ from the majority of the time series data values. It's important to detect these suspicious rare items, events, or observations in a timely manner to be minimized. The following approach can be used to detect a variety of anomalies such as: outages, cyber-attacks, or viral web content. The following image is an example of spikes in a time series dataset:



Add the CreateEmptyDataView() method

Add the following method to `Program.cs`:

```
I DataView CreateEmptyDataView(MLContext mlContext) {
    // Create empty DataView. We just need the schema to call Fit() for the time series transforms
    I Enumerable<ProductSalesData> enumerableData = new List<ProductSalesData>();
    return mlContext.Data.LoadFromEnumerable(enumerableData);
}
```

The `CreateEmptyDataView()` produces an empty data view object with the correct schema to be used as input to the `I Estimator.Fit()` method.

Create the DetectSpike() method

The `DetectSpike()` method:

- Creates the transform from the estimator.
- Detects spikes based on historical sales data.
- Displays the results.

1. Create the `DetectSpike()` method at the bottom of the `Program.cs` file using the following code:

```
DetectSpike(MLContext mlContext, int docSize, IDataView productSales)
{
}
```

2. Use the `IidSpikeEstimator` to train the model for spike detection. Add it to the `DetectSpike()` method with the following code:

```
var iidSpikeEstimator = mlContext.Transforms.DetectIidSpike(outputColumnName:
    nameof(ProductSalesPrediction.Prediction), inputColumnName: nameof(ProductSalesData.numSales),
    confidence: 95, pvalueHistoryLength: docSize / 4);
```

3. Create the spike detection transform by adding the following as the next line of code in the `DetectSpike()` method:

TIP

The `confidence` and `pvalueHistoryLength` parameters impact how spikes are detected. `confidence` determines how sensitive your model is to spikes. The lower the confidence, the more likely the algorithm is to detect "smaller" spikes. The `pvalueHistoryLength` parameter defines the number of data points in a sliding window. The value of this parameter is usually a percentage of the entire dataset. The lower the `pvalueHistoryLength`, the faster the model forgets previous large spikes.

```
ITransformer iidSpikeTransform = iidSpikeEstimator.Fit(CreateEmptyDataView(mlContext));
```

4. Add the following line of code to transform the `productSales` data as the next line in the `DetectSpike()` method:

```
IDataView transformedData = iidSpikeTransform.Transform(productSales);
```

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

5. Convert your `transformedData` into a strongly typed `IEnumerable` for easier display using the [CreateEnumerable\(\)](#) method with the following code:

```
var predictions = mlContext.Data.CreateEnumerable<ProductSalesPrediction>(transformedData,  
reuseRowObject: false);
```

6. Create a display header line using the following [Console.WriteLine\(\)](#) code:

```
Console.WriteLine("Alert\tScore\tP-Value");
```

You'll display the following information in your spike detection results:

- `Alert` indicates a spike alert for a given data point.
- `Score` is the `ProductSales` value for a given data point in the dataset.
- `P-Value` The "P" stands for probability. The closer the p-value is to 0, the more likely the data point is an anomaly.

7. Use the following code to iterate through the `predictions` `IEnumerable` and display the results:

```
foreach (var p in predictions)  
{  
    var results = $"{p.Prediction[0]}\t{p.Prediction[1]:f2}\t{p.Prediction[2]:F2}";  
  
    if (p.Prediction[0] == 1)  
    {  
        results += " --- Spike detected";  
    }  
  
    Console.WriteLine(results);  
}  
Console.WriteLine("");
```

8. Add the call to the `DetectSpike()` method below the call to the `LoadFromFile()` method:

```
DetectSpike(mlContext, _docszie, dataView);
```

Spike detection results

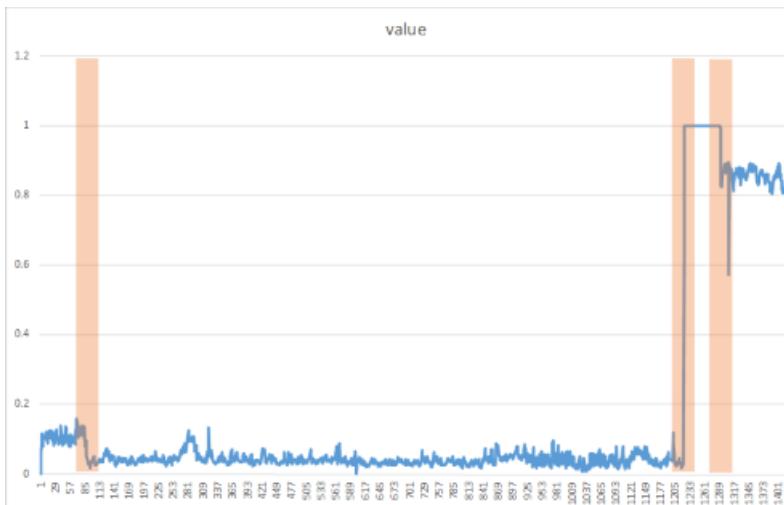
Your results should be similar to the following. During processing, messages are displayed. You may see warnings, or processing messages. Some of the messages have been removed from the following results for clarity.

```
Detect temporary changes in pattern
===== Training the model =====
===== End of training process =====
Alert Score P-Value
0 271.00 0.50
0 150.90 0.00
0 188.10 0.41
0 124.30 0.13
0 185.30 0.47
0 173.50 0.47
0 236.80 0.19
0 229.50 0.27
0 197.80 0.48
0 127.90 0.13
1 341.50 0.00 <-- Spike detected
0 190.90 0.48
0 199.30 0.48
0 154.50 0.24
0 215.10 0.42
0 278.30 0.19
0 196.40 0.43
0 292.00 0.17
0 231.00 0.45
0 308.60 0.18
0 294.90 0.19
1 426.60 0.00 <-- Spike detected
0 269.50 0.47
0 347.30 0.21
0 344.70 0.27
0 445.40 0.06
0 320.90 0.49
0 444.30 0.12
0 406.30 0.29
0 442.40 0.21
1 580.50 0.00 <-- Spike detected
0 412.60 0.45
1 687.00 0.01 <-- Spike detected
0 480.30 0.40
0 586.30 0.20
0 651.90 0.14
```

Change point detection

Change points are persistent changes in a time series event stream distribution of values, like level changes and trends. These persistent changes last much longer than **spikes** and could indicate catastrophic event(s).

Change points are not usually visible to the naked eye, but can be detected in your data using approaches such as in the following method. The following image is an example of a change point detection:



Create the DetectChangepoint() method

The `DetectChangepoint()` method executes the following tasks:

- Creates the transform from the estimator.
- Detects change points based on historical sales data.
- Displays the results.

- Create the `DetectChangepoint()` method, just after the `DetectSpike()` method declaration, using the following code:

```
void DetectChangepoint(MLContext mlContext, int docSize, IDataView productSales)
{
}
```

- Create the `iidChangePointEstimator` in the `DetectChangepoint()` method with the following code:

```
var iidChangePointEstimator = mlContext.Transforms.DetectIidChangePoint(outputColumnName:
    nameof(ProductSalesPrediction.Prediction), inputColumnName: nameof(ProductSalesData.numSales),
    confidence: 95, changeHistoryLength: docSize / 4);
```

- As you did previously, create the transform from the estimator by adding the following line of code in the `DetectChangepoint()` method:

TIP

The detection of change points happens with a slight delay as the model needs to make sure the current deviation is a persistent change and not just some random spikes before creating an alert. The amount of this delay is equal to the `changeHistoryLength` parameter. By increasing the value of this parameter, change detection alerts on more persistent changes, but the trade-off would be a longer delay.

```
var iidChangePointTransform = iidChangePointEstimator.Fit(CreateEmpty DataView(mlContext));
```

- Use the `Transform()` method to transform the data by adding the following code to `DetectChangepoint()`:

:

```
IDataView transformedData = iidChangePointTransform.Transform(productSales);
```

5. As you did previously, convert your `transformedData` into a strongly typed `IEnumerable` for easier display using the `CreateEnumerable()` method with the following code:

```
var predictions = mlContext.Data.CreateEnumerable<ProductSalesPrediction>(transformedData,  
reuseRowObject: false);
```

6. Create a display header with the following code as the next line in the `DetectChangePoint()` method:

```
Console.WriteLine("Alert\tScore\tP-Value\tMartingale value");
```

You'll display the following information in your change point detection results:

- `Alert` indicates a change point alert for a given data point.
- `Score` is the `ProductSales` value for a given data point in the dataset.
- `P-Value` The "P" stands for probability. The closer the P-value is to 0, the more likely the data point is an anomaly.
- `Martingale value` is used to identify how "weird" a data point is, based on the sequence of P-values.

7. Iterate through the `predictions` `IEnumerable` and display the results with the following code:

```
foreach (var p in predictions)  
{  
    var results = $"  
{p.Prediction[0]}\t{p.Prediction[1]:f2}\t{p.Prediction[2]:f2}\t{p.Prediction[3]:f2}";  
  
    if (p.Prediction[0] == 1)  
    {  
        results += " --- alert is on, predicted changepoint";  
    }  
    Console.WriteLine(results);  
}  
Console.WriteLine("");
```

8. Add the following call to the `DetectChangepoint()` method after the call to the `DetectSpike()` method:

```
DetectChangepoint(mlContext, _docszie, dataView);
```

Change point detection results

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

```

Detect Persistent changes in pattern
===== Training the model Using Change Point Detection Algorithm=====
===== End of training process =====
Alert Score P-Value Martingale value
0    271.00  0.50   0.00
0    150.90  0.00   2.33
0    188.10  0.41   2.80
0    124.30  0.13   9.16
0    185.30  0.47   9.77
0    173.50  0.47   10.41
0    236.80  0.19   24.46
0    229.50  0.27   42.38
1    197.80  0.48   44.23 <-- alert is on, predicted changepoint
0    127.90  0.13   145.25
0    341.50  0.00   0.01
0    190.90  0.48   0.01
0    199.30  0.48   0.00
0    154.50  0.24   0.00
0    215.10  0.42   0.00
0    278.30  0.19   0.00
0    196.40  0.43   0.00
0    292.00  0.17   0.01
0    231.00  0.45   0.00
0    308.60  0.18   0.00
0    294.90  0.19   0.00
0    426.60  0.00   0.00
0    269.50  0.47   0.00
0    347.30  0.21   0.00
0    344.70  0.27   0.00
0    445.40  0.06   0.02
0    320.90  0.49   0.01
0    444.30  0.12   0.02
0    406.30  0.29   0.01
0    442.40  0.21   0.01
0    580.50  0.00   0.01
0    412.60  0.45   0.01
0    687.00  0.01   0.12
0    480.30  0.40   0.08
0    586.30  0.20   0.03
0    651.90  0.14   0.09

```

Congratulations! You've now successfully built machine learning models for detecting spikes and change point anomalies in sales data.

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

In this tutorial, you learned how to:

- Load the data
- Train the model for spike anomaly detection
- Detect spike anomalies with the trained model
- Train the model for change point anomaly detection
- Detect change point anomalies with the trained mode

Next steps

Check out the Machine Learning samples GitHub repository to explore a seasonality data anomaly detection sample.

[dotnet/machinelearning-samples](#) GitHub repository

Tutorial: Detect objects using ONNX in ML.NET

10/14/2022 • 30 minutes to read • [Edit Online](#)

Learn how to use a pre-trained ONNX model in ML.NET to detect objects in images.

Training an object detection model from scratch requires setting millions of parameters, a large amount of labeled training data and a vast amount of compute resources (hundreds of GPU hours). Using a pre-trained model allows you to shortcut the training process.

In this tutorial, you learn how to:

- Understand the problem
- Learn what ONNX is and how it works with ML.NET
- Understand the model
- Reuse the pre-trained model
- Detect objects with a loaded model

Pre-requisites

- [Visual Studio 2022](#).
- [Microsoft.ML NuGet Package](#)
- [Microsoft.ML.ImageAnalytics NuGet Package](#)
- [Microsoft.ML.OnnxTransformer NuGet Package](#)
- [Tiny YOLOv2 pre-trained model](#)
- [Netron](#) (optional)

ONNX object detection sample overview

This sample creates a .NET core console application that detects objects within an image using a pre-trained deep learning ONNX model. The code for this sample can be found on the [dotnet/machinelearning-samples repository](#) on GitHub.

What is object detection?

Object detection is a computer vision problem. While closely related to image classification, object detection performs image classification at a more granular scale. Object detection both locates *and* categorizes entities within images. Object detection models are commonly trained using deep learning and neural networks. See [Deep learning vs machine learning](#) for more information.

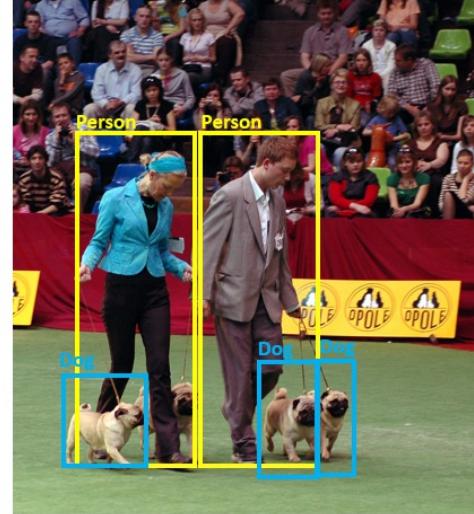
Use object detection when images contain multiple objects of different types.

Image Classification



{Dog}

Object Detection



{Dog, Dog, Dog, Person, Person}

Some use cases for object detection include:

- Self-Driving Cars
- Robotics
- Face Detection
- Workplace Safety
- Object Counting
- Activity Recognition

Select a deep learning model

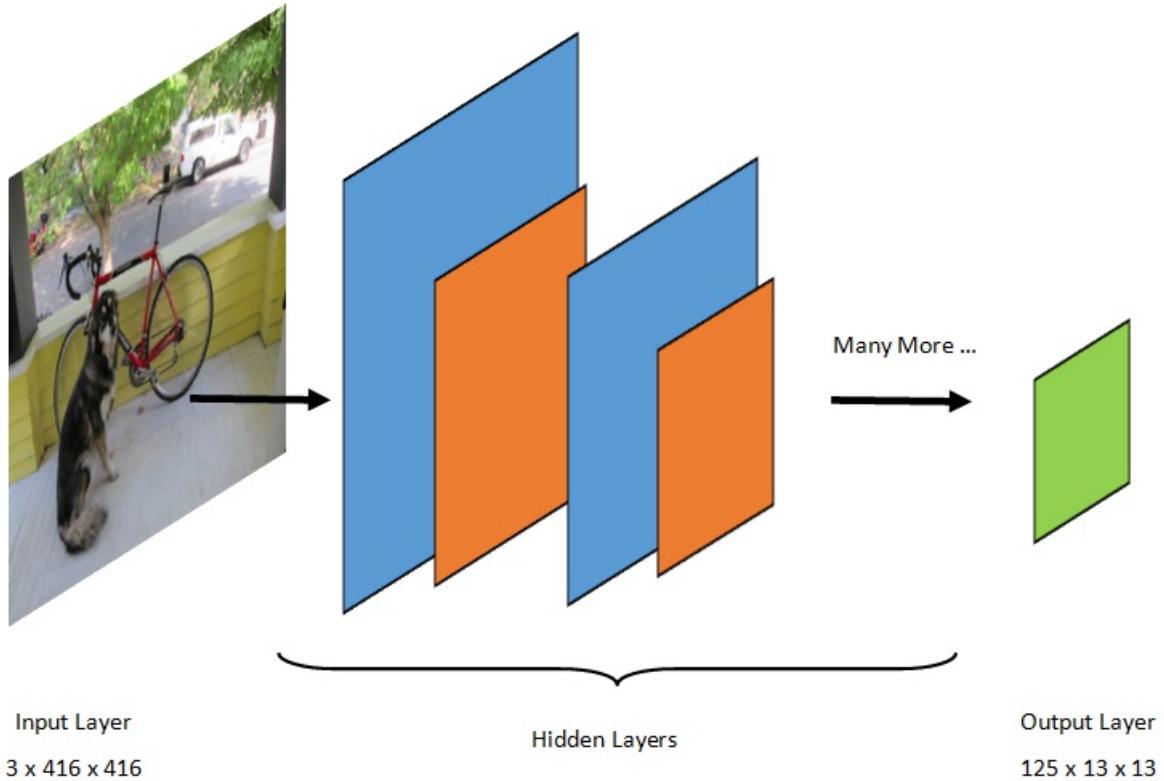
Deep learning is a subset of machine learning. To train deep learning models, large quantities of data are required. Patterns in the data are represented by a series of layers. The relationships in the data are encoded as connections between the layers containing weights. The higher the weight, the stronger the relationship. Collectively, this series of layers and connections are known as artificial neural networks. The more layers in a network, the "deeper" it is, making it a deep neural network.

There are different types of neural networks, the most common being Multi-Layered Perceptron (MLP), Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN). The most basic is the MLP, which maps a set of inputs to a set of outputs. This neural network is good when the data does not have a spatial or time component. The CNN makes use of convolutional layers to process spatial information contained in the data. A good use case for CNNs is image processing to detect the presence of a feature in a region of an image (for example, is there a nose in the center of an image?). Finally, RNNs allow for the persistence of state or memory to be used as input. RNNs are used for time-series analysis, where the sequential ordering and context of events is important.

Understand the model

Object detection is an image-processing task. Therefore, most deep learning models trained to solve this problem are CNNs. The model used in this tutorial is the Tiny YOLOv2 model, a more compact version of the YOLOv2 model described in the paper: "[YOLO9000: Better, Faster, Stronger](#)" by Redmon and Farhad. Tiny YOLOv2 is trained on the Pascal VOC dataset and is made up of 15 layers that can predict 20 different classes of objects. Because Tiny YOLOv2 is a condensed version of the original YOLOv2 model, a tradeoff is made between speed and accuracy. The different layers that make up the model can be visualized using tools like Netron. Inspecting the model would yield a mapping of the connections between all the layers that make up the neural network, where each layer would contain the name of the layer along with the dimensions of the respective

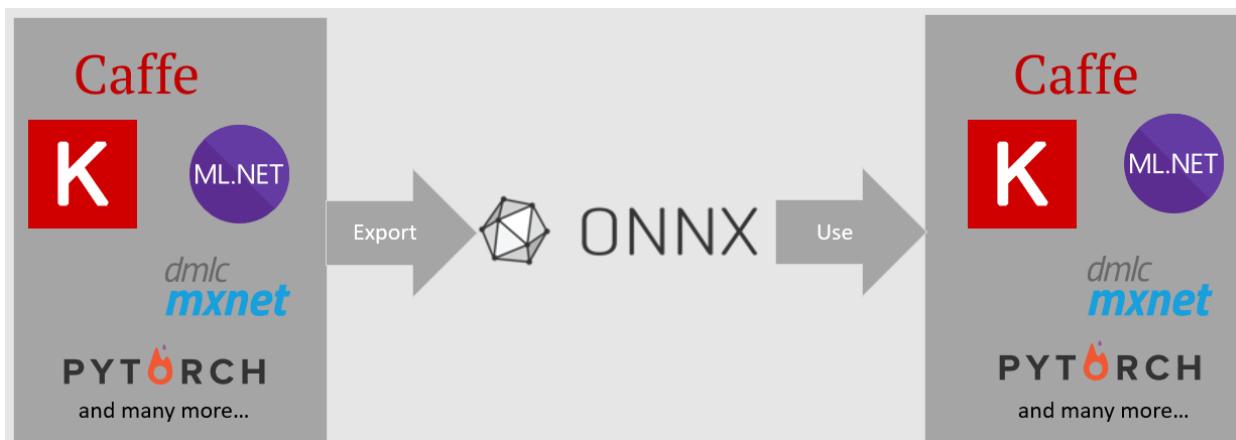
input / output. The data structures used to describe the inputs and outputs of the model are known as tensors. Tensors can be thought of as containers that store data in N-dimensions. In the case of Tiny YOLOv2, the name of the input layer is `image` and it expects a tensor of dimensions $3 \times 416 \times 416$. The name of the output layer is `grid` and generates an output tensor of dimensions $125 \times 13 \times 13$.



The YOLO model takes an image $3(\text{RGB}) \times 416\text{px} \times 416\text{px}$. The model takes this input and passes it through the different layers to produce an output. The output divides the input image into a 13×13 grid, with each cell in the grid consisting of 125 values.

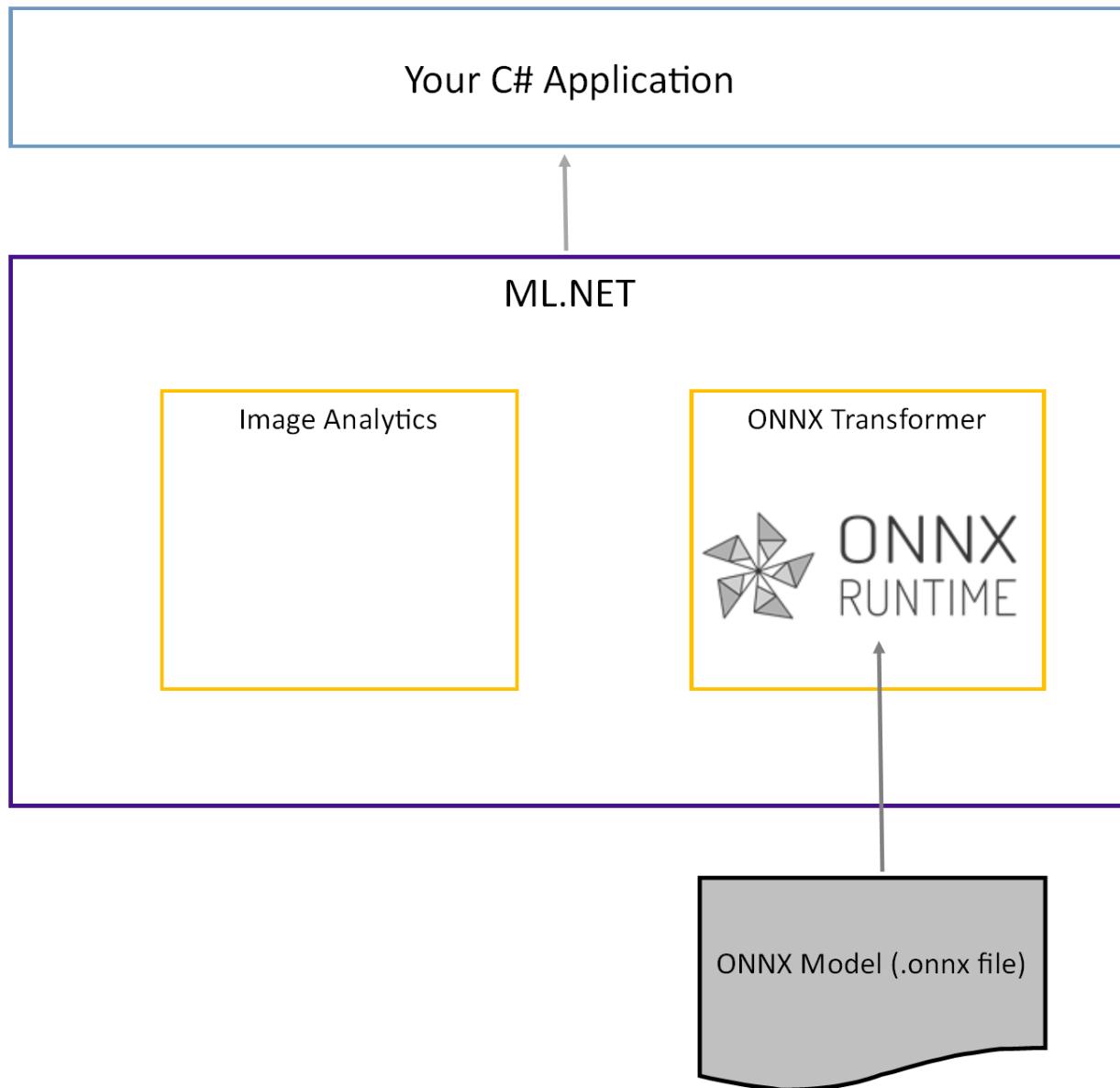
What is an ONNX model?

The Open Neural Network Exchange (ONNX) is an open source format for AI models. ONNX supports interoperability between frameworks. This means you can train a model in one of the many popular machine learning frameworks like PyTorch, convert it into ONNX format and consume the ONNX model in a different framework like ML.NET. To learn more, visit the [ONNX website](#).



The pre-trained Tiny YOLOv2 model is stored in ONNX format, a serialized representation of the layers and learned patterns of those layers. In ML.NET, interoperability with ONNX is achieved with the `ImageAnalytics` and `OnnxTransformer` NuGet packages. The `ImageAnalytics` package contains a series of transforms that take an image and encode it into numerical values that can be used as input into a prediction or training pipeline. The `OnnxTransformer` package leverages the ONNX Runtime to load an ONNX model and use it to make predictions.

based on input provided.



Set up the .NET Console project

Now that you have a general understanding of what ONNX is and how Tiny YOLOv2 works, it's time to build the application.

Create a console application

1. Create a **C# Console Application** called "ObjectDetection". Click the **Next** button.
2. Choose **.NET 6** as the framework to use. Click the **Create** button.
3. Install the **Microsoft.ML NuGet Package**:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

- In Solution Explorer, right-click on your project and select **Manage NuGet Packages**.
- Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.ML**.
- 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.

- Repeat these steps for **Microsoft.ML.ImageAnalytics**, **Microsoft.ML.OnnxTransformer** and **Microsoft.ML.OnnxRuntime**.

Prepare your data and pre-trained model

1. Download [The project assets directory zip file](#) and unzip.
2. Copy the `assets` directory into your *ObjectDetection* project directory. This directory and its subdirectories contain the image files (except for the Tiny YOLOv2 model, which you'll download and add in the next step) needed for this tutorial.
3. Download the Tiny YOLOv2 model from the [ONNX Model Zoo](#).
4. Copy the `model.onnx` file into your *ObjectDetection* project `assets\Model` directory and rename it to `TinyYolo2_model.onnx`. This directory contains the model needed for this tutorial.
5. In Solution Explorer, right-click each of the files in the asset directory and subdirectories and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

Create classes and define paths

Open the `Program.cs` file and add the following additional `using` statements to the top of the file:

```
using System.Drawing;
using System.Drawing.Drawing2D;
using ObjectDetection.YoloParser;
using ObjectDetection.DataStructures;
using ObjectDetection;
using Microsoft.ML;
```

Next, define the paths of the various assets.

1. First, create the `GetAbsolutePath` method at the bottom of the `Program.cs` file.

```
string GetAbsolutePath(string relativePath)
{
    FileInfo _dataRoot = new FileInfo(typeof(Program).Assembly.Location);
    string assemblyFolderPath = _dataRoot.Directory.FullName;

    string fullPath = Path.Combine(assemblyFolderPath, relativePath);

    return fullPath;
}
```

2. Then, below the using statements, create fields to store the location of your assets.

```
var assetsRelativePath = @"../../../../../assets";
string assetsPath = GetAbsolutePath(assetsRelativePath);
var modelFilePath = Path.Combine(assetsPath, "Model", "TinyYolo2_model.onnx");
var imagesFolder = Path.Combine(assetsPath, "images");
var outputFolder = Path.Combine(assetsPath, "images", "output");
```

Add a new directory to your project to store your input data and prediction classes.

In **Solution Explorer**, right-click the project, and then select **Add > New Folder**. When the new folder appears in the Solution Explorer, name it "DataStructures".

Create your input data class in the newly created *DataStructures* directory.

1. In **Solution Explorer**, right-click the *DataStructures* directory, and then select **Add > New Item**.
2. In the **Add New Item** dialog box, select **Class** and change the **Name** field to *ImageNetData.cs*. Then, select the **Add** button.

The *ImageNetData.cs* file opens in the code editor. Add the following `using` statement to the top of *ImageNetData.cs*:

```
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Microsoft.ML.Data;
```

Remove the existing class definition and add the following code for the `ImageNetData` class to the *ImageNetData.cs* file:

```
public class ImageNetData
{
    [LoadColumn(0)]
    public string ImagePath;

    [LoadColumn(1)]
    public string Label;

    public static IEnumerable<ImageNetData> ReadFromFile(string imageFolder)
    {
        return Directory
            .GetFiles(imageFolder)
            .Where(filePath => Path.GetExtension(filePath) != ".md")
            .Select(filePath => new ImageNetData { ImagePath = filePath, Label =
        Path.GetFileName(filePath) });
    }
}
```

`ImageNetData` is the input image data class and has the following `String` fields:

- `ImagePath` contains the path where the image is stored.
- `Label` contains the name of the file.

Additionally, `ImageNetData` contains a method `ReadFromFile` that loads multiple image files stored in the `imageFolder` path specified and returns them as a collection of `ImageNetData` objects.

Create your prediction class in the *DataStructures* directory.

1. In **Solution Explorer**, right-click the *DataStructures* directory, and then select **Add > New Item**.
2. In the **Add New Item** dialog box, select **Class** and change the **Name** field to *ImageNetPrediction.cs*. Then, select the **Add** button.

The *ImageNetPrediction.cs* file opens in the code editor. Add the following `using` statement to the top of *ImageNetPrediction.cs*:

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

Remove the existing class definition and add the following code for the `ImageNetPrediction` class to the *ImageNetPrediction.cs* file:

```

public class ImageNetPrediction
{
    [ColumnName("grid")]
    public float[] PredictedLabels;
}

```

`ImageNetPrediction` is the prediction data class and has the following `float[]` field:

- `PredictedLabel` contains the dimensions, objectness score, and class probabilities for each of the bounding boxes detected in an image.

Initialize variables

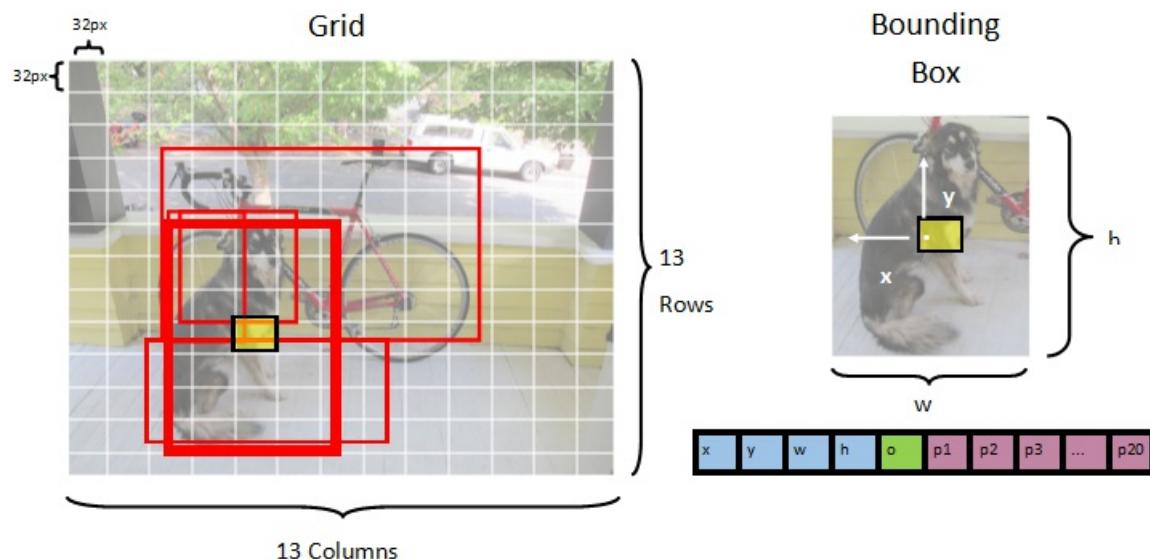
The `MLContext` class is a starting point for all ML.NET operations, and initializing `mlContext` creates a new ML.NET environment that can be shared across the model creation workflow objects. It's similar, conceptually, to `DbContext` in Entity Framework.

Initialize the `mlContext` variable with a new instance of `MLContext` by adding the following line below the `outputFolder` field.

```
MLContext mlContext = new MLContext();
```

Create a parser to post-process model outputs

The model segments an image into a 13×13 grid, where each grid cell is $32px \times 32px$. Each grid cell contains 5 potential object bounding boxes. A bounding box has 25 elements:



- `x` the x position of the bounding box center relative to the grid cell it's associated with.
- `y` the y position of the bounding box center relative to the grid cell it's associated with.
- `w` the width of the bounding box.
- `h` the height of the bounding box.
- `o` the confidence value that an object exists within the bounding box, also known as objectness score.
- `p1-p20` class probabilities for each of the 20 classes predicted by the model.

In total, the 25 elements describing each of the 5 bounding boxes make up the 125 elements contained in each grid cell.

The output generated by the pre-trained ONNX model is a float array of length `21125`, representing the

elements of a tensor with dimensions `125 x 13 x 13`. In order to transform the predictions generated by the model into a tensor, some post-processing work is required. To do so, create a set of classes to help parse the output.

Add a new directory to your project to organize the set of parser classes.

1. In **Solution Explorer**, right-click the project, and then select **Add > New Folder**. When the new folder appears in the Solution Explorer, name it "YoloParser".

Create bounding boxes and dimensions

The data output by the model contains coordinates and dimensions of the bounding boxes of objects within the image. Create a base class for dimensions.

1. In **Solution Explorer**, right-click the `YoloParser` directory, and then select **Add > New Item**.
2. In the **Add New Item** dialog box, select **Class** and change the **Name** field to `DimensionsBase.cs`. Then, select the **Add** button.

The `DimensionsBase.cs` file opens in the code editor. Remove all `using` statements and existing class definition.

Add the following code for the `DimensionsBase` class to the `DimensionsBase.cs` file:

```
public class DimensionsBase
{
    public float X { get; set; }
    public float Y { get; set; }
    public float Height { get; set; }
    public float Width { get; set; }
}
```

`DimensionsBase` has the following `float` properties:

- `X` contains the position of the object along the x-axis.
- `Y` contains the position of the object along the y-axis.
- `Height` contains the height of the object.
- `Width` contains the width of the object.

Next, create a class for your bounding boxes.

1. In **Solution Explorer**, right-click the `YoloParser` directory, and then select **Add > New Item**.
2. In the **Add New Item** dialog box, select **Class** and change the **Name** field to `YoloBoundingBox.cs`. Then, select the **Add** button.

The `YoloBoundingBox.cs` file opens in the code editor. Add the following `using` statement to the top of `YoloBoundingBox.cs`.

```
using System.Drawing;
```

Just above the existing class definition, add a new class definition called `BoundingBoxDimensions` that inherits from the `DimensionsBase` class to contain the dimensions of the respective bounding box.

```
public class BoundingBoxDimensions : DimensionsBase { }
```

Remove the existing `YoloBoundingBox` class definition and add the following code for the

`YoloBoundingBox` class to the `YoloBoundingBox.cs` file:

```
public class YoloBoundingBox
{
    public BoundingBoxDimensions Dimensions { get; set; }

    public string Label { get; set; }

    public float Confidence { get; set; }

    public RectangleF Rect
    {
        get { return new RectangleF(Dimensions.X, Dimensions.Y, Dimensions.Width, Dimensions.Height); }
    }

    public Color BoxColor { get; set; }
}
```

`YoloBoundingBox` has the following properties:

- `Dimensions` contains dimensions of the bounding box.
- `Label` contains the class of object detected within the bounding box.
- `Confidence` contains the confidence of the class.
- `Rect` contains the rectangle representation of the bounding box's dimensions.
- `BoxColor` contains the color associated with the respective class used to draw on the image.

Create the parser

Now that the classes for dimensions and bounding boxes are created, it's time to create the parser.

1. In **Solution Explorer**, right-click the `YoloParser` directory, and then select **Add > New Item**.
2. In the **Add New Item** dialog box, select **Class** and change the **Name** field to `YoloOutputParser.cs`. Then, select the **Add** button.

The `YoloOutputParser.cs` file opens in the code editor. Add the following `using` statements to the top of `YoloOutputParser.cs`:

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
```

Inside the existing `YoloOutputParser` class definition, add a nested class that contains the dimensions of each of the cells in the image. Add the following code for the `CellDimensions` class that inherits from the `DimensionsBase` class at the top of the `YoloOutputParser` class definition.

```
class CellDimensions : DimensionsBase { }
```

3. Inside the `YoloOutputParser` class definition, add the following constants and field.

```

public const int ROW_COUNT = 13;
public const int COL_COUNT = 13;
public const int CHANNEL_COUNT = 125;
public const int BOXES_PER_CELL = 5;
public const int BOX_INFO_FEATURE_COUNT = 5;
public const int CLASS_COUNT = 20;
public const float CELL_WIDTH = 32;
public const float CELL_HEIGHT = 32;

private int channelStride = ROW_COUNT * COL_COUNT;

```

- `ROW_COUNT` is the number of rows in the grid the image is divided into.
- `COL_COUNT` is the number of columns in the grid the image is divided into.
- `CHANNEL_COUNT` is the total number of values contained in one cell of the grid.
- `BOXES_PER_CELL` is the number of bounding boxes in a cell,
- `BOX_INFO_FEATURE_COUNT` is the number of features contained within a box (x,y,height,width,confidence).
- `CLASS_COUNT` is the number of class predictions contained in each bounding box.
- `CELL_WIDTH` is the width of one cell in the image grid.
- `CELL_HEIGHT` is the height of one cell in the image grid.
- `channelStride` is the starting position of the current cell in the grid.

When the model makes a prediction, also known as scoring, it divides the `416px x 416px` input image into a grid of cells the size of `13 x 13`. Each cell contains `32px x 32px`. Within each cell, there are 5 bounding boxes each containing 5 features (x, y, width, height, confidence). In addition, each bounding box contains the probability of each of the classes, which in this case is 20. Therefore, each cell contains 125 pieces of information (5 features + 20 class probabilities).

Create a list of anchors below `channelStride` for all 5 bounding boxes:

```

private float[] anchors = new float[]
{
    1.08F, 1.19F, 3.42F, 4.41F, 6.63F, 11.38F, 9.42F, 5.11F, 16.62F, 10.52F
};

```

Anchors are pre-defined height and width ratios of bounding boxes. Most object or classes detected by a model have similar ratios. This is valuable when it comes to creating bounding boxes. Instead of predicting the bounding boxes, the offset from the pre-defined dimensions is calculated therefore reducing the computation required to predict the bounding box. Typically these anchor ratios are calculated based on the dataset used. In this case, because the dataset is known and the values have been pre-computed, the anchors can be hard-coded.

Next, define the labels or classes that the model will predict. This model predicts 20 classes, which is a subset of the total number of classes predicted by the original YOLOv2 model.

Add your list of labels below the `anchors`.

```

private string[] labels = new string[]
{
    "aeroplane", "bicycle", "bird", "boat", "bottle",
    "bus", "car", "cat", "chair", "cow",
    "diningtable", "dog", "horse", "motorbike", "person",
    "pottedplant", "sheep", "sofa", "train", "tvmonitor"
};

```

There are colors associated with each of the classes. Assign your class colors below your `labels`:

```

private static Color[] classColors = new Color[]
{
    Color.Khaki,
    Color.Fuchsia,
    Color.Silver,
    Color.RoyalBlue,
    Color.Green,
    Color.DarkOrange,
    Color.Purple,
    Color.Gold,
    Color.Red,
    Color.Aquamarine,
    Color.Lime,
    Color.AliceBlue,
    Color.Sienna,
    Color.Orchid,
    Color.Tan,
    Color.LightPink,
    Color.Yellow,
    Color.HotPink,
    Color.OliveDrab,
    Color.SandyBrown,
    Color.DarkTurquoise
};

```

Create helper functions

There are a series of steps involved in the post-processing phase. To help with that, several helper methods can be employed.

The helper methods used in by the parser are:

- `Sigmoid` applies the sigmoid function that outputs a number between 0 and 1.
- `Softmax` normalizes an input vector into a probability distribution.
- `GetOffset` maps elements in the one-dimensional model output to the corresponding position in a `125 x 13 x 13` tensor.
- `ExtractBoundingBoxes` extracts the bounding box dimensions using the `GetOffset` method from the model output.
- `GetConfidence` extracts the confidence value that states how sure the model is that it has detected an object and uses the `Sigmoid` function to turn it into a percentage.
- `MapBoundingBoxToCell` uses the bounding box dimensions and maps them onto its respective cell within the image.
- `ExtractClasses` extracts the class predictions for the bounding box from the model output using the `GetOffset` method and turns them into a probability distribution using the `Softmax` method.
- `GetTopResult` selects the class from the list of predicted classes with the highest probability.
- `IntersectionOverUnion` filters overlapping bounding boxes with lower probabilities.

Add the code for all the helper methods below your list of `classColors`.

```

private float Sigmoid(float value)
{
    var k = (float)Math.Exp(value);
    return k / (1.0f + k);
}

private float[] Softmax(float[] values)
{
    var maxVal = values.Max();
    var exp = values.Select(v => Math.Exp(v - maxVal));
    var sumExp = exp.Sum();
}

```

```

        return exp.Select(v => (float)(v / sumExp)).ToArray();
    }

    private int GetOffset(int x, int y, int channel)
    {
        // YOLO outputs a tensor that has a shape of 125x13x13, which
        // WinML flattens into a 1D array. To access a specific channel
        // for a given (x,y) cell position, we need to calculate an offset
        // into the array
        return (channel * this.channelStride) + (y * COL_COUNT) + x;
    }

    private BoundingBoxDimensions ExtractBoundingBoxDimensions(float[] modelOutput, int x, int y, int channel)
    {
        return new BoundingBoxDimensions
        {
            X = modelOutput[GetOffset(x, y, channel)],
            Y = modelOutput[GetOffset(x, y, channel + 1)],
            Width = modelOutput[GetOffset(x, y, channel + 2)],
            Height = modelOutput[GetOffset(x, y, channel + 3)]
        };
    }

    private float GetConfidence(float[] modelOutput, int x, int y, int channel)
    {
        return Sigmoid(modelOutput[GetOffset(x, y, channel + 4)]);
    }

    private CellDimensions MapBoundingBoxToCell(int x, int y, int box, BoundingBoxDimensions boxDimensions)
    {
        return new CellDimensions
        {
            X = ((float)x + Sigmoid(boxDimensions.X)) * CELL_WIDTH,
            Y = ((float)y + Sigmoid(boxDimensions.Y)) * CELL_HEIGHT,
            Width = (float)Math.Exp(boxDimensions.Width) * CELL_WIDTH * anchors[box * 2],
            Height = (float)Math.Exp(boxDimensions.Height) * CELL_HEIGHT * anchors[box * 2 + 1],
        };
    }

    public float[] ExtractClasses(float[] modelOutput, int x, int y, int channel)
    {
        float[] predictedClasses = new float[CLASS_COUNT];
        int predictedClassOffset = channel + BOX_INFO_FEATURE_COUNT;
        for (int predictedClass = 0; predictedClass < CLASS_COUNT; predictedClass++)
        {
            predictedClasses[predictedClass] = modelOutput[GetOffset(x, y, predictedClass +
predictedClassOffset)];
        }
        return Softmax(predictedClasses);
    }

    private ValueTuple<int, float> GetTopResult(float[] predictedClasses)
    {
        return predictedClasses
            .Select((predictedClass, index) => (Index: index, Value: predictedClass))
            .OrderByDescending(result => result.Value)
            .First();
    }

    private float IntersectionOverUnion(RectangleF boundingBoxA, RectangleF boundingBoxB)
    {
        var areaA = boundingBoxA.Width * boundingBoxA.Height;

        if (areaA <= 0)
            return 0;

        var areaB = boundingBoxB.Width * boundingBoxB.Height;
    }
}

```

```

    if (areaB <= 0)
        return 0;

    var minX = Math.Max(boundingBoxA.Left, boundingBoxB.Left);
    var minY = Math.Max(boundingBoxA.Top, boundingBoxB.Top);
    var maxX = Math.Min(boundingBoxA.Right, boundingBoxB.Right);
    var maxY = Math.Min(boundingBoxA.Bottom, boundingBoxB.Bottom);

    var intersectionArea = Math.Max(maxY - minY, 0) * Math.Max(maxX - minX, 0);

    return intersectionArea / (areaA + areaB - intersectionArea);
}

```

Once you have defined all of the helper methods, it's time to use them to process the model output.

Below the `IntersectionOverUnion` method, create the `ParseOutputs` method to process the output generated by the model.

```

public IList<YoloBoundingBox> ParseOutputs(float[] yoloModelOutputs, float threshold = .3F)
{
}

```

Create a list to store your bounding boxes and define variables inside the `ParseOutputs` method.

```
var boxes = new List<YoloBoundingBox>();
```

Each image is divided into a grid of `13 x 13` cells. Each cell contains five bounding boxes. Below the `boxes` variable, add code to process all of the boxes in each of the cells.

```

for (int row = 0; row < ROW_COUNT; row++)
{
    for (int column = 0; column < COL_COUNT; column++)
    {
        for (int box = 0; box < BOXES_PER_CELL; box++)
        {
        }
    }
}

```

Inside the inner-most loop, calculate the starting position of the current box within the one-dimensional model output.

```
var channel = (box * (CLASS_COUNT + BOX_INFO_FEATURE_COUNT));
```

Directly below that, use the `ExtractBoundingBoxDimensions` method to get the dimensions of the current bounding box.

```
BoundingBoxDimensions boundingBoxDimensions = ExtractBoundingBoxDimensions(yoloModelOutputs, row, column, channel);
```

Then, use the `GetConfidence` method to get the confidence for the current bounding box.

```
float confidence = GetConfidence(yoloModelOutputs, row, column, channel);
```

After that, use the `MapBoundingBoxToCell` method to map the current bounding box to the current cell being processed.

```
CellDimensions mappedBoundingBox = MapBoundingBoxToCell(row, column, box, boundingBoxDimensions);
```

Before doing any further processing, check whether your confidence value is greater than the threshold provided. If not, process the next bounding box.

```
if (confidence < threshold)
    continue;
```

Otherwise, continue processing the output. The next step is to get the probability distribution of the predicted classes for the current bounding box using the `ExtractClasses` method.

```
float[] predictedClasses = ExtractClasses(yoloModelOutputs, row, column, channel);
```

Then, use the `GetTopResult` method to get the value and index of the class with the highest probability for the current box and compute its score.

```
var (topResultIndex, topResultScore) = GetTopResult(predictedClasses);
var topScore = topResultScore * confidence;
```

Use the `topScore` to once again keep only those bounding boxes that are above the specified threshold.

```
if (topScore < threshold)
    continue;
```

Finally, if the current bounding box exceeds the threshold, create a new `BoundingBox` object and add it to the `boxes` list.

```
boxes.Add(new YoloBoundingBox()
{
    Dimensions = new BoundingBoxDimensions
    {
        X = (mappedBoundingBox.X - mappedBoundingBox.Width / 2),
        Y = (mappedBoundingBox.Y - mappedBoundingBox.Height / 2),
        Width = mappedBoundingBox.Width,
        Height = mappedBoundingBox.Height,
    },
    Confidence = topScore,
    Label = labels[topResultIndex],
    BoxColor = classColors[topResultIndex]
});
```

Once all cells in the image have been processed, return the `boxes` list. Add the following return statement below the outer-most for-loop in the `ParseOutputs` method.

```
return boxes;
```

Filter overlapping boxes

Now that all of the highly confident bounding boxes have been extracted from the model output, additional filtering needs to be done to remove overlapping images. Add a method called `FilterBoundingBoxes` below the

`ParseOutputs` method:

```
public IList<YoloBoundingBox> FilterBoundingBoxes(IList<YoloBoundingBox> boxes, int limit, float threshold)
{
}
```

Inside the `FilterBoundingBoxes` method, start off by creating an array equal to the size of detected boxes and marking all slots as active or ready for processing.

```
var activeCount = boxes.Count;
var isActiveBoxes = new bool[boxes.Count];

for (int i = 0; i < isActiveBoxes.Length; i++)
    isActiveBoxes[i] = true;
```

Then, sort the list containing your bounding boxes in descending order based on confidence.

```
var sortedBoxes = boxes.Select((b, i) => new { Box = b, Index = i })
    .OrderByDescending(b => b.Box.Confidence)
    .ToList();
```

After that, create a list to hold the filtered results.

```
var results = new List<YoloBoundingBox>();
```

Begin processing each bounding box by iterating over each of the bounding boxes.

```
for (int i = 0; i < boxes.Count; i++)
{
}
```

Inside of this for-loop, check whether the current bounding box can be processed.

```
if (isActiveBoxes[i])
{
}
```

If so, add the bounding box to the list of results. If the results exceed the specified limit of boxes to be extracted, break out of the loop. Add the following code inside the if-statement.

```
var boxA = sortedBoxes[i].Box;
results.Add(boxA);

if (results.Count >= limit)
    break;
```

Otherwise, look at the adjacent bounding boxes. Add the following code below the box limit check.

```
for (var j = i + 1; j < boxes.Count; j++)  
{  
}
```

Like the first box, if the adjacent box is active or ready to be processed, use the `IntersectionOverUnion` method to check whether the first box and the second box exceed the specified threshold. Add the following code to your innermost for-loop.

```
if (isActiveBoxes[j])  
{  
    var boxB = sortedBoxes[j].Box;  
  
    if (IntersectionOverUnion(boxA.Rect, boxB.Rect) > threshold)  
    {  
        isActiveBoxes[j] = false;  
        activeCount--;  
  
        if (activeCount <= 0)  
            break;  
    }  
}
```

Outside of the inner-most for-loop that checks adjacent bounding boxes, see whether there are any remaining bounding boxes to be processed. If not, break out of the outer for-loop.

```
if (activeCount <= 0)  
    break;
```

Finally, outside of the initial for-loop of the `FilterBoundingBoxes` method, return the results:

```
return results;
```

Great! Now it's time to use this code along with the model for scoring.

Use the model for scoring

Just like with post-processing, there are a few steps in the scoring steps. To help with this, add a class that will contain the scoring logic to your project.

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 `OnnxModelScorer.cs`. Then, select the **Add** button.

The `OnnxModelScorer.cs` file opens in the code editor. Add the following `using` statements to the top of `OnnxModelScorer.cs`:

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using Microsoft.ML;  
using Microsoft.ML.Data;  
using ObjectDetection.DataStructures;  
using ObjectDetection.YoloParser;
```

Inside the `OnnxModelScorer` class definition, add the following variables.

```
private readonly string imagesFolder;
private readonly string modelLocation;
private readonly MLContext mlContext;

private IList<YoloBoundingBox> _boundingBoxes = new List<YoloBoundingBox>();
```

Directly below that, create a constructor for the `OnnxModelScorer` class that will initialize the previously defined variables.

```
public OnnxModelScorer(string imagesFolder, string modelLocation, MLContext mlContext)
{
    this.imagesFolder = imagesFolder;
    this.modelLocation = modelLocation;
    this.mlContext = mlContext;
}
```

Once you have created the constructor, define a couple of structs that contain variables related to the image and model settings. Create a struct called `ImageNetSettings` to contain the height and width expected as input for the model.

```
public struct ImageNetSettings
{
    public const int imageHeight = 416;
    public const int imageWidth = 416;
}
```

After that, create another struct called `TinyYoloModelSettings` that contains the names of the input and output layers of the model. To visualize the name of the input and output layers of the model, you can use a tool like [Netron](#).

```
public struct TinyYoloModelSettings
{
    // for checking Tiny yolo2 Model input and output parameter names,
    //you can use tools like Netron,
    // which is installed by Visual Studio AI Tools

    // input tensor name
    public const string ModelInput = "image";

    // output tensor name
    public const string ModelOutput = "grid";
}
```

Next, create the first set of methods use for scoring. Create the `LoadModel` method inside of your `OnnxModelScorer` class.

```
private ITransformer LoadModel(string modelLocation)
{
```

Inside the `LoadModel` method, add the following code for logging.

```
Console.WriteLine("Read model");
Console.WriteLine($"Model location: {modelLocation}");
Console.WriteLine($"Default parameters: image size={({ImageNetSettings.imageWidth},
{ImageNetSettings.imageHeight})}");
```

ML.NET pipelines need to know the data schema to operate on when the `Fit` method is called. In this case, a process similar to training will be used. However, because no actual training is happening, it is acceptable to use an empty `IDataView`. Create a new `IDataView` for the pipeline from an empty list.

```
var data = mlContext.Data.LoadFromEnumerable(new List<ImageNetData>());
```

Below that, define the pipeline. The pipeline will consist of four transforms.

- `LoadImages` loads the image as a Bitmap.
- `ResizeImages` rescales the image to the size specified (in this case, `416 x 416`).
- `ExtractPixels` changes the pixel representation of the image from a Bitmap to a numerical vector.
- `ApplyOnnxModel` loads the ONNX model and uses it to score on the data provided.

Define your pipeline in the `LoadModel` method below the `data` variable.

```
var pipeline = mlContext.Transforms.LoadImages(outputColumnName: "image", imageFolder: "",
inputColumnName: nameof(ImageNetData.ImagePath))
    .Append(mlContext.Transforms.ResizeImages(outputColumnName: "image", imageWidth:
ImageNetSettings.imageWidth, imageHeight: ImageNetSettings.imageHeight, inputColumnName: "image"))
    .Append(mlContext.Transforms.ExtractPixels(outputColumnName: "image"))
    .Append(mlContext.Transforms.ApplyOnnxModel(modelFile: modelLocation,
outputColumnNames: new[] { TinyYoloModelSettings.ModelOutput }, inputColumnNames: new[] {
TinyYoloModelSettings.ModelInput }));
```

Now it's time to instantiate the model for scoring. Call the `Fit` method on the pipeline and return it for further processing.

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

return model;
```

Once the model is loaded, it can then be used to make predictions. To facilitate that process, create a method called `PredictDataUsingModel` below the `LoadModel` method.

```
private IEnumerable<float[]> PredictDataUsingModel(IDataView testData, ITransformer model)
{
}
```

Inside the `PredictDataUsingModel`, add the following code for logging.

```
Console.WriteLine($"Images location: {imagesFolder}");
Console.WriteLine("");
Console.WriteLine("=====Identify the objects in the images=====");
Console.WriteLine("");
```

Then, use the `Transform` method to score the data.

```
IDataView scoredData = model.Transform(testData);
```

Extract the predicted probabilities and return them for additional processing.

```
IEnumerable<float[]> probabilities = scoredData.GetColumn<float[]>(TinyYoloModelSettings.ModelOutput);

return probabilities;
```

Now that both steps are set up, combine them into a single method. Below the `PredictDataUsingModel` method, add a new method called `Score`.

```
public IEnumerable<float[]> Score(IDataView data)
{
    var model = LoadModel(modelLocation);

    return PredictDataUsingModel(data, model);
}
```

Almost there! Now it's time to put it all to use.

Detect objects

Now that all of the setup is complete, it's time to detect some objects.

Score and parse model outputs

Below the creation of the `mlContext` variable, add a try-catch statement.

```
try
{
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```

Inside of the `try` block, start implementing the object detection logic. First, load the data into an `IDataView`.

```
IEnumerable<ImageNetData> images = ImageNetData.ReadFromFile(imagesFolder);
IDataView imageDataView = mlContext.Data.LoadFromEnumerable(images);
```

Then, create an instance of `OnnxModelScorer` and use it to score the loaded data.

```
// Create instance of model scorer
var modelScorer = new OnnxModelScorer(imagesFolder, modelFilePath, mlContext);

// Use model to score data
IEnumerable<float[]> probabilities = modelScorer.Score(imageDataView);
```

Now it's time for the post-processing step. Create an instance of `YoloOutputParser` and use it to process the model output.

```

YoloOutputParser parser = new YoloOutputParser();

var boundingBoxes =
    probabilities
        .Select(probability => parser.ParseOutputs(probability))
        .Select(boxes => parser.FilterBoundingBoxes(boxes, 5, .5F));

```

Once the model output has been processed, it's time to draw the bounding boxes on the images.

Visualize predictions

After the model has scored the images and the outputs have been processed, the bounding boxes have to be drawn on the image. To do so, add a method called `DrawBoundingBox` below the `GetAbsolutePath` method inside of `Program.cs`.

```

void DrawBoundingBox(string inputImageLocation, string outputImageLocation, string imageName,
IList<YoloBoundingBox> filteredBoundingBoxes)
{
}

```

First, load the image and get the height and width dimensions in the `DrawBoundingBox` method.

```

Image image = Image.FromFile(Path.Combine(inputImageLocation, imageName));

var originalImageHeight = image.Height;
var originalImageWidth = image.Width;

```

Then, create a for-each loop to iterate over each of the bounding boxes detected by the model.

```

foreach (var box in filteredBoundingBoxes)
{
}

```

Inside of the for-each loop, get the dimensions of the bounding box.

```

var x = (uint)Math.Max(box.Dimensions.X, 0);
var y = (uint)Math.Max(box.Dimensions.Y, 0);
var width = (uint)Math.Min(originalImageWidth - x, box.Dimensions.Width);
var height = (uint)Math.Min(originalImageHeight - y, box.Dimensions.Height);

```

Because the dimensions of the bounding box correspond to the model input of `416 x 416`, scale the bounding box dimensions to match the actual size of the image.

```

x = (uint)originalImageWidth * x / OnnxModelScorer.ImageNetSettings.imageWidth;
y = (uint)originalImageHeight * y / OnnxModelScorer.ImageNetSettings.imageHeight;
width = (uint)originalImageWidth * width / OnnxModelScorer.ImageNetSettings.imageWidth;
height = (uint)originalImageHeight * height / OnnxModelScorer.ImageNetSettings.imageHeight;

```

Then, define a template for text that will appear above each bounding box. The text will contain the class of the object inside of the respective bounding box as well as the confidence.

```

string text = $"{box.Label} ({(box.Confidence * 100).ToString("0")})";

```

In order to draw on the image, convert it to a `Graphics` object.

```
using (Graphics thumbnailGraphic = Graphics.FromImage(image))
{
}
```

Inside the `using` code block, tune the graphic's `Graphics` object settings.

```
thumbnailGraphic.CompositingQuality = CompositingQuality.HighQuality;
thumbnailGraphic.SmoothingMode = SmoothingMode.HighQuality;
thumbnailGraphic.InterpolationMode = InterpolationMode.HighQualityBicubic;
```

Below that, set the font and color options for the text and bounding box.

```
// Define Text Options
Font drawFont = new Font("Arial", 12, FontStyle.Bold);
SizeF size = thumbnailGraphic.MeasureString(text, drawFont);
SolidBrush fontBrush = new SolidBrush(Color.Black);
Point atPoint = new Point((int)x, (int)y - (int)size.Height - 1);

// Define BoundingBox options
Pen pen = new Pen(box.BoxColor, 3.2f);
SolidBrush colorBrush = new SolidBrush(box.BoxColor);
```

Create and fill a rectangle above the bounding box to contain the text using the `FillRectangle` method. This will help contrast the text and improve readability.

```
thumbnailGraphic.FillRectangle(colorBrush, (int)x, (int)(y - size.Height - 1), (int)size.Width,
(int)size.Height);
```

Then, Draw the text and bounding box on the image using the `DrawString` and `DrawRectangle` methods.

```
thumbnailGraphic.DrawString(text, drawFont, fontBrush, atPoint);

// Draw bounding box on image
thumbnailGraphic.DrawRectangle(pen, x, y, width, height);
```

Outside of the for-each loop, add code to save the images in the `outputFolder`.

```
if (!Directory.Exists(outputImageLocation))
{
    Directory.CreateDirectory(outputImageLocation);
}

image.Save(Path.Combine(outputImageLocation, imageName));
```

For additional feedback that the application is making predictions as expected at run time, add a method called `LogDetectedObjects` below the `DrawBoundingBox` method in the *Program.cs* file to output the detected objects to the console.

```
void LogDetectedObjects(string imageName, IList<YoloBoundingBox> boundingBoxes)
{
    Console.WriteLine($".....The objects in the image {imageName} are detected as below....");

    foreach (var box in boundingBoxes)
    {
        Console.WriteLine($"{box.Label} and its Confidence score: {box.Confidence}");
    }

    Console.WriteLine("");
}
```

Now that you have helper methods to create visual feedback from the predictions, add a for-loop to iterate over each of the scored images.

```
for (var i = 0; i < images.Count(); i++)
{
```

Inside of the for-loop, get the name of the image file and the bounding boxes associated with it.

```
string imageFileName = images.ElementAt(i).Label;
IList<YoloBoundingBox> detectedObjects = boundingBoxes.ElementAt(i);
```

Below that, use the `DrawBoundingBox` method to draw the bounding boxes on the image.

```
DrawBoundingBox(imagesFolder, outputFolder, imageFileName, detectedObjects);
```

Lastly, use the `LogDetectedObjects` method to output predictions to the console.

```
LogDetectedObjects(imageFileName, detectedObjects);
```

After the try-catch statement, add additional logic to indicate the process is done running.

```
Console.WriteLine("===== End of Process..Hit any Key =====");
```

That's it!

Results

After following the previous steps, run your console app (Ctrl + F5). Your results should be similar to the following output. You may see warnings or processing messages, but these messages have been removed from the following results for clarity.

=====Identify the objects in the images=====

.....The objects in the image image1.jpg are detected as below....
car and its Confidence score: 0.9697262
car and its Confidence score: 0.6674225
person and its Confidence score: 0.5226039
car and its Confidence score: 0.5224892
car and its Confidence score: 0.4675332

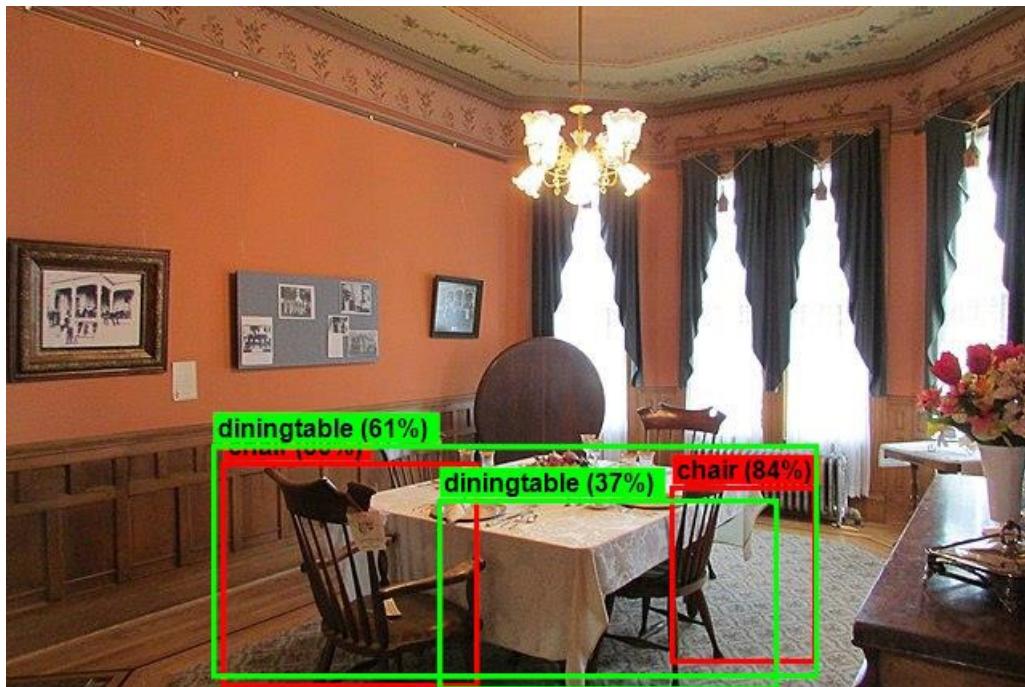
.....The objects in the image image2.jpg are detected as below....
cat and its Confidence score: 0.6461141
cat and its Confidence score: 0.6400049

.....The objects in the image image3.jpg are detected as below....
chair and its Confidence score: 0.840578
chair and its Confidence score: 0.796363
diningtable and its Confidence score: 0.6056048
diningtable and its Confidence score: 0.3737402

.....The objects in the image image4.jpg are detected as below....
dog and its Confidence score: 0.7608147
person and its Confidence score: 0.6321323
dog and its Confidence score: 0.5967442
person and its Confidence score: 0.5730394
person and its Confidence score: 0.5551759

===== End of Process..Hit any Key ======

To see the images with bounding boxes, navigate to the `assets/images/output/` directory. Below is a sample from one of the processed images.



Congratulations! You've now successfully built a machine learning model for object detection by reusing a pre-trained `ONNX` model in ML.NET.

You can find the source code for this tutorial at the [dotnet/machinelearning-samples](https://github.com/dotnet/machinelearning-samples) repository.

In this tutorial, you learned how to:

- Understand the problem
- Learn what ONNX is and how it works with ML.NET
- Understand the model

- Reuse the pre-trained model
- Detect objects with a loaded model

Check out the Machine Learning samples GitHub repository to explore an expanded object detection sample.

[dotnet/machinelearning-samples GitHub repository](#)

Tutorial: Analyze sentiment of movie reviews using a pre-trained TensorFlow model in ML.NET

10/14/2022 • 10 minutes to read • [Edit Online](#)

This tutorial shows you how to use a pre-trained TensorFlow model to classify sentiment in website comments. The binary sentiment classifier is a C# console application developed using Visual Studio.

The TensorFlow model used in this tutorial was trained using movie reviews from the IMDB database. Once you have finished developing the application, you will be able to supply movie review text and the application will tell you whether the review has positive or negative sentiment.

In this tutorial, you learn how to:

- Load a pre-trained TensorFlow model
- Transform website comment text into features suitable for the model
- Use the model to make a prediction

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

Prerequisites

- [Visual Studio 2022](#) with the ".NET Desktop Development" workload installed.

Setup

Create the application

1. Create a C# Console Application called "TextClassificationTF". Click the **Next** button.
2. Choose .NET 6 as the framework to use. Click the **Create** button.
3. Create a directory named *Data* in your project to save your data set files.
4. Install the **Microsoft.ML** NuGet Package:

NOTE

This sample uses the latest stable version of the NuGet packages mentioned unless otherwise stated.

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. Repeat these steps for **Microsoft.ML.TensorFlow**, **Microsoft.ML.SampleUtils** and **SciSharp.TensorFlow.Redist**.

Add the TensorFlow model to the project

NOTE

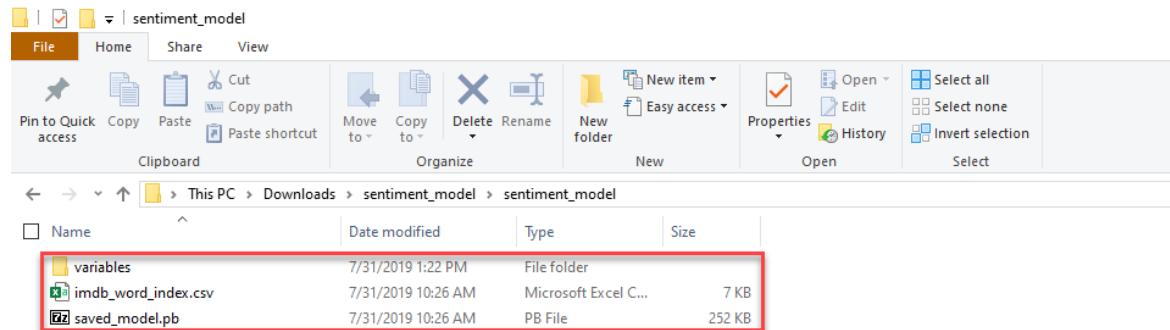
The model for this tutorial is from the [dotnet/machinelearning-testdata](#) GitHub repo. The model is in TensorFlow SavedModel format.

1. Download the [sentiment_model zip file](#), and unzip.

The zip file contains:

- `saved_model.pb`: the TensorFlow model itself. The model takes a fixed length (size 600) integer array of features representing the text in an IMDB review string, and outputs two probabilities which sum to 1: the probability that the input review has positive sentiment, and the probability that the input review has negative sentiment.
- `imdb_word_index.csv`: a mapping from individual words to an integer value. The mapping is used to generate the input features for the TensorFlow model.

2. Copy the contents of the innermost `sentiment_model` directory into your *TextClassificationTF* project `sentiment_model` directory. This directory contains the model and additional support files needed for this tutorial, as shown in the following image:



3. In Solution Explorer, right-click each of the files in the `sentiment_model` directory and subdirectory and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

Add using statements and global variables

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

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

2. Create a global variable right after the using statements to hold the saved model file path.

```
string _modelPath = Path.Combine(Environment.CurrentDirectory, "sentiment_model");
```

- `_modelPath` is the file path of the trained model.

Model the data

Movie reviews are free form text. Your application converts the text into the input format expected by the model in a number of discrete stages.

The first is to split the text into separate words and use the provided mapping file to map each word onto an integer encoding. The result of this transformation is a variable length integer array with a length corresponding to the number of words in the sentence.

PROPERTY	VALUE	TYPE
ReviewText	this film is really good	string
VariableLengthFeatures	14,22,9,66,78,...	int[]

The variable length feature array is then resized to a fixed length of 600. This is the length that the TensorFlow model expects.

PROPERTY	VALUE	TYPE
ReviewText	this film is really good	string
VariableLengthFeatures	14,22,9,66,78,...	int[]
Features	14,22,9,66,78,...	int[600]

1. Create a class for your input data at the bottom of the `Program.cs` file:

```
/// <summary>
/// Class to hold original sentiment data.
/// </summary>
public class MovieReview
{
    public string ReviewText { get; set; }
}
```

The input data class, `MovieReview`, has a `string` for user comments (`ReviewText`).

2. Create a class for the variable length features after the `MovieReview` class:

```
/// <summary>
/// Class to hold the variable length feature vector. Used to define the
/// column names used as input to the custom mapping action.
/// </summary>
public class VariableLength
{
    /// <summary>
    /// This is a variable length vector designated by VectorType attribute.
    /// Variable length vectors are produced by applying operations such as 'TokenizeWords' on
    strings
    /// resulting in vectors of tokens of variable lengths.
    /// </summary>
    [VectorType]
    public int[] VariableLengthFeatures { get; set; }
}
```

The `VariableLengthFeatures` property has a `VectorType` attribute to designate it as a vector. All of the vector elements must be the same type. In data sets with a large number of columns, loading multiple columns as a single vector reduces the number of data passes when you apply data transformations.

This class is used in the `ResizeFeatures` action. The names of its properties (in this case only one) are used to indicate which columns in the `DataView` can be used as the `input` to the custom mapping action.

3. Create a class for the fixed length features, after the `VariableLength` class:

```

/// <summary>
/// Class to hold the fixed length feature vector. Used to define the
/// column names used as output from the custom mapping action,
/// </summary>
public class FixedLength
{
    /// <summary>
    /// This is a fixed length vector designated by VectorType attribute.
    /// </summary>
    [VectorType(Config.FeatureLength)]
    public int[] Features { get; set; }
}

```

This class is used in the `ResizeFeatures` action. The names of its properties (in this case only one) are used to indicate which columns in the `DataView` can be used as the *output* of the custom mapping action.

Note that the name of the property `Features` is determined by the TensorFlow model. You cannot change this property name.

4. Create a class for the prediction after the `FixedLength` class:

```

/// <summary>
/// Class to contain the output values from the transformation.
/// </summary>
public class MovieReviewSentimentPrediction
{
    [VectorType(2)]
    public float[] Prediction { get; set; }
}

```

`MovieReviewSentimentPrediction` is the prediction class used after the model training.

`MovieReviewSentimentPrediction` has a single `float` array (`Prediction`) and a `VectorType` attribute.

5. Create another class to hold configuration values, such as the feature vector length:

```

static class Config
{
    public const int FeatureLength = 600;
}

```

Create the `MLContext`, lookup dictionary, and action to resize features

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.

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

```
MLContext mlContext = new MLContext();
```

2. Create a dictionary to encode words as integers by using the `LoadFromTextFile` method to load mapping data from a file, as seen in the following table:

WORD	INDEX
kids	362

WORD	INDEX
want	181
wrong	355
effects	302
feeling	547

Add the code below to create the lookup map:

```
var lookupMap = mlContext.Data.LoadFromTextFile(Path.Combine(_modelPath, "imdb_word_index.csv"),
    columns: new[]
    {
        new TextLoader.Column("Words", DataKind.String, 0),
        new TextLoader.Column("Ids", DataKind.Int32, 1),
    },
    separatorChar: ',',
);
```

3. Add an `Action` to resize the variable length word integer array to an integer array of fixed size, with the next lines of code:

```
Action<VariableLength, FixedLength> ResizeFeaturesAction = (s, f) =>
{
    var features = s.VariableLengthFeatures;
    Array.Resize(ref features, Config.FeatureLength);
    f.Features = features;
};
```

Load the pre-trained TensorFlow model

1. Add code to load the TensorFlow model:

```
TensorFlowModel tensorflowModel = mlContext.Model.LoadTensorFlowModel(_modelPath);
```

Once the model is loaded, you can extract its input and output schema. The schemas are displayed for interest and learning only. You do not need this code for the final application to function:

```
DataViewSchema schema = tensorflowModel.GetModelSchema();
Console.WriteLine(" ===== TensorFlow Model Schema ===== ");
var featuresType = (VectorDataViewType)schema["Features"].Type;
Console.WriteLine($"Name: Features, Type: {featuresType.ItemType.RawType}, Size: {featuresType.Dimensions[0]}");
var predictionType = (VectorDataViewType)schema["Prediction/Softmax"].Type;
Console.WriteLine($"Name: Prediction/Softmax, Type: {predictionType.ItemType.RawType}, Size: {predictionType.Dimensions[0]}");
```

The input schema is the fixed-length array of integer encoded words. The output schema is a float array of probabilities indicating whether a review's sentiment is negative, or positive . These values sum to 1, as the probability of being positive is the complement of the probability of the sentiment being negative.

Create the ML.NET pipeline

1. Create the pipeline and split the input text into words using `TokenizeIntoWords` transform to break the text into words as the next line of code:

```
IEstimator<ITransformer> pipeline =
    // Split the text into individual words
    mlContext.Transforms.Text.TokenizeIntoWords("TokenizedWords", "ReviewText")
```

The `TokenizeIntoWords` transform uses spaces to parse the text/string into words. It creates a new column and splits each input string to a vector of substrings based on the user-defined separator.

2. Map the words onto their integer encoding using the lookup table that you declared above:

```
// Map each word to an integer value. The array of integer makes up the input features.
.Append(mlContext.Transforms.Conversion.MapValue("VariableLengthFeatures", lookupMap,
    lookupMap.Schema["Words"], lookupMap.Schema["Ids"], "TokenizedWords"))
```

3. Resize the variable length integer encodings to the fixed-length one required by the model:

```
// Resize variable length vector to fixed length vector.
.Append(mlContext.Transforms.CustomMapping(ResizeFeaturesAction, "Resize"))
```

4. Classify the input with the loaded TensorFlow model:

```
// Passes the data to TensorFlow for scoring
.Append(tensorFlowModel.ScoreTensorFlowModel("Prediction/Softmax", "Features"))
```

The TensorFlow model output is called `Prediction/Softmax`. Note that the name `Prediction/Softmax` is determined by the TensorFlow model. You cannot change this name.

5. Create a new column for the output prediction:

```
// Retrieves the 'Prediction' from TensorFlow and copies to a column
.Append(mlContext.Transforms.CopyColumns("Prediction", "Prediction/Softmax"));
```

You need to copy the `Prediction/Softmax` column into one with a name that can be used as a property in a C# class: `Prediction`. The `/` character is not allowed in a C# property name.

Create the ML.NET model from the pipeline

1. Add the code to create the model from the pipeline:

```
// Create an executable model from the estimator pipeline
IDataView dataView = mlContext.Data.LoadFromEnumerable(new List<MovieReview>());
ITransformer model = pipeline.Fit(dataView);
```

An ML.NET model is created from the chain of estimators in the pipeline by calling the `Fit` method. In this case, we are not fitting any data to create the model, as the TensorFlow model has already been previously trained. We supply an empty data view object to satisfy the requirements of the `Fit` method.

Use the model to make a prediction

1. Add the `PredictSentiment` method above the `MovieReview` class:

```
void PredictSentiment(MLContext mlContext, ITransformer model)
{
}
```

2. Add the following code to create the `PredictionEngine` as the first line in the `PredictSentiment()` method:

```
var engine = mlContext.Model.CreatePredictionEngine<MovieReview, MovieReviewSentimentPrediction>(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](#).

NOTE

`PredictionEnginePool` service extension is currently in preview.

3. Add a comment to test the trained model's prediction in the `Predict()` method by creating an instance of `MovieReview`:

```
var review = new MovieReview()
{
    ReviewText = "this film is really good"
};
```

4. Pass the test comment data to the `Prediction Engine` by adding the next lines of code in the `PredictSentiment()` method:

```
var sentimentPrediction = engine.Predict(review);
```

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

PROPERTY	VALUE	TYPE
Prediction	[0.5459937, 0.454006255]	float[]

6. Display sentiment prediction using the following code:

```
Console.WriteLine("Number of classes: {0}", sentimentPrediction.Prediction.Length);
Console.WriteLine("Is sentiment/review positive? {0}", sentimentPrediction.Prediction[1] > 0.5 ?
    "Yes." : "No.");
```

7. Add a call to `PredictSentiment` after calling the `Fit()` method:

```
PredictSentiment(mlContext, model);
```

Results

Build and run your application.

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

```
Number of classes: 2
Is sentiment/review positive ? Yes
```

Congratulations! You've now successfully built a machine learning model for classifying and predicting messages sentiment by reusing a pre-trained `TensorFlow` model in ML.NET.

You can find the source code for this tutorial at the [dotnet/samples](#) repository.

In this tutorial, you learned how to:

- Load a pre-trained TensorFlow model
- Transform website comment text into features suitable for the model
- Use the model to make a prediction

Create a game match up list app with Infer.NET and probabilistic programming

10/14/2022 • 4 minutes to read • [Edit Online](#)

This how-to guide teaches you about probabilistic programming using Infer.NET. Probabilistic programming is a machine learning approach where custom models are expressed as computer programs. It allows for incorporating domain knowledge in the models and makes the machine learning system more interpretable. It also supports online inference – the process of learning as new data arrives. Infer.NET is used in various products at Microsoft in Azure, Xbox, and Bing.

What is probabilistic programming?

Probabilistic programming allows you to create statistical models of real-world processes.

Prerequisites

- Local development environment setup

This how-to guide expects you to have a machine you can use for development. The .NET tutorial [Hello World in 10 minutes](#) has instructions for setting up your local development environment on macOS, Windows, or Linux.

Create your app

1. Open a new command prompt and run the following commands:

```
dotnet new console -o myApp  
cd myApp
```

The `dotnet` command creates a `new` application of type `console`. The `-o` parameter creates a directory named `myApp` where your app is stored and populates it with the required files. The `cd myApp` command puts you into the newly created app directory.

Install Infer.NET package

To use Infer.NET, you need to install the `Microsoft.ML.Probabilistic.Compiler` package. In your command prompt, run the following command:

```
dotnet add package Microsoft.ML.Probabilistic.Compiler
```

Design your model

The example sample uses table tennis or foosball matches played in the office. You have the participants and outcome of each match. You want to infer the player's skills from this data. Assume that each player has a normally distributed latent skill and their given match performance is a noisy version of this skill. The data constrains the winner's performance to be greater than the loser's performance. This is a simplified version of the popular [TrueSkill](#) model, which also supports teams, draws, and other extensions. An [advanced version](#) of this model is used for matchmaking in the best-selling game titles Halo and Gears of War.

You need to list the inferred player skills, alongside with their variance – the measure of uncertainty around the skills.

Game result sample data

GAME	WINNER	LOSER
1	Player 0	Player 1
2	Player 0	Player 3
3	Player 0	Player 4
4	Player 1	Player 2
5	Player 3	Player 1
6	Player 4	Player 2

With a closer look at the sample data, you'll notice that players 3 and 4 both have one win and one loss. Let's see what the rankings look like using probabilistic programming. Notice also there is a player zero because even office match up lists are zero based to us developers.

Write some code

Having designed the model, it's time to express it as a probabilistic program using the Infer.NET modeling API.

Open `Program.cs` in your favorite text editor and replace all of its contents with the following code:

```

namespace myApp

{
    using System;
    using System.Linq;
    using Microsoft.ML.Probabilistic;
    using Microsoft.ML.Probabilistic.Distributions;
    using Microsoft.ML.Probabilistic.Models;

    class Program
    {

        static void Main(string[] args)
        {
            // The winner and loser in each of 6 samples games
            var winnerData = new[] { 0, 0, 0, 1, 3, 4 };
            var loserData = new[] { 1, 3, 4, 2, 1, 2 };

            // Define the statistical model as a probabilistic program
            var game = new Range(winnerData.Length);
            var player = new Range(winnerData.Concat(loserData).Max() + 1);
            var playerSkills = Variable.Array<double>(player);
            playerSkills[player] = Variable.GaussianFromMeanAndVariance(6, 9).ForEach(player);

            var winners = Variable.Array<int>(game);
            var losers = Variable.Array<int>(game);

            using (Variable.ForEach(game))
            {
                // The player performance is a noisy version of their skill
                var winnerPerformance = Variable.GaussianFromMeanAndVariance(playerSkills[winners[game]], 1.0);
                var loserPerformance = Variable.GaussianFromMeanAndVariance(playerSkills[losers[game]], 1.0);

                // The winner performed better in this game
                Variable.ConstrainTrue(winnerPerformance > loserPerformance);
            }

            // Attach the data to the model
            winners.ObservedValue = winnerData;
            losers.ObservedValue = loserData;

            // Run inference
            var inferenceEngine = new InferenceEngine();
            var inferredSkills = inferenceEngine.Infer<Gaussian[]>(playerSkills);

            // The inferred skills are uncertain, which is captured in their variance
            var orderedPlayerSkills = inferredSkills
                .Select((s, i) => new { Player = i, Skill = s })
                .OrderByDescending(ps => ps.Skill.GetMean());

            foreach (var playerSkill in orderedPlayerSkills)
            {
                Console.WriteLine($"Player {playerSkill.Player} skill: {playerSkill.Skill}");
            }
        }
    }
}

```

Run your app

In your command prompt, run the following command:

```
dotnet run
```

Results

Your results should be similar to the following:

```
Compiling model...done.  
Iterating:  
.....|.....|.....|.....|.....| 50  
Player 0 skill: Gaussian(9.517, 3.926)  
Player 3 skill: Gaussian(6.834, 3.892)  
Player 4 skill: Gaussian(6.054, 4.731)  
Player 1 skill: Gaussian(4.955, 3.503)  
Player 2 skill: Gaussian(2.639, 4.288)
```

In the results, notice that player 3 ranks slightly higher than player 4 according to our model. That's because the victory of player 3 over player 1 is more significant than the victory of player 4 over player 2 – note that player 1 beats player 2. Player 0 is the overall champ!

Keep learning

Designing statistical models is a skill on its own. The Microsoft Research Cambridge team has written a [free online book](#), which gives a gentle introduction to the article. Chapter 3 of this book covers the TrueSkill model in more detail. Once you have a model in mind, you can transform it into code using the [extensive documentation](#) on the Infer.NET website.

Next steps

Check out the Infer.NET GitHub repository to continue learning and find more samples.

[dotnet/infer GitHub repository](#)

Machine learning tasks in ML.NET

10/14/2022 • 9 minutes to read • [Edit Online](#)

A machine learning task is the type of prediction or inference being made, based on the problem or question that is being asked, and the available data. For example, the classification task assigns data to categories, and the clustering task groups data according to similarity.

Machine learning tasks rely on patterns in the data rather than being explicitly programmed.

This article describes the different machine learning tasks that you can choose from in ML.NET and some common use cases.

Once you have decided which task works for your scenario, then you need to choose the best algorithm to train your model. The available algorithms are listed in the section for each task.

Binary classification

A [supervised machine learning](#) task that is used to predict which of two classes (categories) an instance of data belongs to. The input of a classification algorithm is a set of labeled examples, where each label is an integer of either 0 or 1. The output of a binary classification algorithm is a classifier, which you can use to predict the class of new unlabeled instances. Examples of binary classification scenarios include:

- [Understanding sentiment of Twitter comments](#) as either "positive" or "negative".
- Diagnosing whether a patient has a certain disease or not.
- Making a decision to mark an email as "spam" or not.
- Determining if a photo contains a particular item or not, such as a dog or fruit.

For more information, see the [Binary classification](#) article on Wikipedia.

Binary classification trainers

You can train a binary classification model using the following algorithms:

- [AveragedPerceptronTrainer](#)
- [SdcaLogisticRegressionBinaryTrainer](#)
- [SdcaNonCalibratedBinaryTrainer](#)
- [SymbolicSgdLogisticRegressionBinaryTrainer](#)
- [LbfsgsLogisticRegressionBinaryTrainer](#)
- [LightGbmBinaryTrainer](#)
- [FastTreeBinaryTrainer](#)
- [FastForestBinaryTrainer](#)
- [GamBinaryTrainer](#)
- [FieldAwareFactorizationMachineTrainer](#)
- [PriorTrainer](#)
- [LinearSvmTrainer](#)

Binary classification inputs and outputs

For best results with binary classification, the training data should be balanced (that is, equal numbers of positive and negative training data). Missing values should be handled before training.

The input label column data must be [Boolean](#). The input features column data must be a fixed-size vector of

Single.

These trainers output the following columns:

OUTPUT COLUMN NAME	COLUMN TYPE	DESCRIPTION
Score	Single	The raw score that was calculated by the model
PredictedLabel	Boolean	The predicted label, based on the sign of the score. A negative score maps to false and a positive score maps to true .

Multiclass classification

A [supervised machine learning](#) task that is used to predict the class (category) of an instance of data. The input of a classification algorithm is a set of labeled examples. Each label normally starts as text. It is then run through the TermTransform, which converts it to the Key (numeric) type. The output of a classification algorithm is a classifier, which you can use to predict the class of new unlabeled instances. Examples of multi-class classification scenarios include:

- Categorizing flights as "early", "on time", or "late".
- Understanding movie reviews as "positive", "neutral", or "negative".
- Categorizing hotel reviews as "location", "price", "cleanliness", etc.

For more information, see the [Multiclass classification](#) article on Wikipedia.

NOTE

One vs all upgrades any [binary classification learner](#) to act on multiclass datasets. More information on [Wikipedia](#).

Multiclass classification trainers

You can train a multiclass classification model using the following training algorithms:

- [LightGbmMulticlassTrainer](#)
- [SdcaMaximumEntropyMulticlassTrainer](#)
- [SdcaNonCalibratedMulticlassTrainer](#)
- [LbfgsMaximumEntropyMulticlassTrainer](#)
- [NaiveBayesMulticlassTrainer](#)
- [OneVersusAllTrainer](#)
- [PairwiseCouplingTrainer](#)

Multiclass classification inputs and outputs

The input label column data must be [key](#) type. The feature column must be a fixed size vector of [Single](#).

This trainer outputs the following:

OUTPUT NAME	TYPE	DESCRIPTION

OUTPUT NAME	TYPE	DESCRIPTION
Score	Vector of Single	The scores of all classes. Higher value means higher probability to fall into the associated class. If the i-th element has the largest value, the predicted label index would be i. Note that i is zero-based index.
PredictedLabel	key type	The predicted label's index. If its value is i, the actual label would be the i-th category in the key-valued input label type.

Regression

A [supervised machine learning](#) task that is used to predict the value of the label from a set of related features. The label can be of any real value and is not from a finite set of values as in classification tasks. Regression algorithms model the dependency of the label on its related features to determine how the label will change as the values of the features are varied. The input of a regression algorithm is a set of examples with labels of known values. The output of a regression algorithm is a function, which you can use to predict the label value for any new set of input features. Examples of regression scenarios include:

- Predicting house prices based on house attributes such as number of bedrooms, location, or size.
- Predicting future stock prices based on historical data and current market trends.
- Predicting sales of a product based on advertising budgets.

Regression trainers

You can train a regression model using the following algorithms:

- [LbfgsPoissonRegressionTrainer](#)
- [LightGbmRegressionTrainer](#)
- [SdcaRegressionTrainer](#)
- [OlsTrainer](#)
- [OnlineGradientDescentTrainer](#)
- [FastTreeRegressionTrainer](#)
- [FastTreeTweedieTrainer](#)
- [FastForestRegressionTrainer](#)
- [GamRegressionTrainer](#)

Regression inputs and outputs

The input label column data must be [Single](#).

The trainers for this task output the following:

OUTPUT NAME	TYPE	DESCRIPTION
Score	Single	The raw score that was predicted by the model

Clustering

An [unsupervised machine learning](#) task that is used to group instances of data into clusters that contain similar characteristics. Clustering can also be used to identify relationships in a dataset that you might not logically

derive by browsing or simple observation. The inputs and outputs of a clustering algorithm depends on the methodology chosen. You can take a distribution, centroid, connectivity, or density-based approach. ML.NET currently supports a centroid-based approach using K-Means clustering. Examples of clustering scenarios include:

- Understanding segments of hotel guests based on habits and characteristics of hotel choices.
- Identifying customer segments and demographics to help build targeted advertising campaigns.
- Categorizing inventory based on manufacturing metrics.

Clustering trainer

You can train a clustering model using the following algorithm:

- [KMeansTrainer](#)

Clustering inputs and outputs

The input features data must be [Single](#). No labels are needed.

This trainer outputs the following:

OUTPUT NAME	TYPE	DESCRIPTION
<code>Score</code>	vector of Single	The distances of the given data point to all clusters' centroids
<code>PredictedLabel</code>	key type	The closest cluster's index predicted by the model.

Anomaly detection

This task creates an anomaly detection model by using Principal Component Analysis (PCA). PCA-Based Anomaly Detection helps you build a model in scenarios where it is easy to obtain training data from one class, such as valid transactions, but difficult to obtain sufficient samples of the targeted anomalies.

An established technique in machine learning, PCA is frequently used in exploratory data analysis because it reveals the inner structure of the data and explains the variance in the data. PCA works by analyzing data that contains multiple variables. It looks for correlations among the variables and determines the combination of values that best captures differences in outcomes. These combined feature values are used to create a more compact feature space called the principal components.

Anomaly detection encompasses many important tasks in machine learning:

- Identifying transactions that are potentially fraudulent.
- Learning patterns that indicate that a network intrusion has occurred.
- Finding abnormal clusters of patients.
- Checking values entered into a system.

Because anomalies are rare events by definition, it can be difficult to collect a representative sample of data to use for modeling. The algorithms included in this category have been especially designed to address the core challenges of building and training models by using imbalanced data sets.

Anomaly detection trainer

You can train an anomaly detection model using the following algorithm:

- [RandomizedPcaTrainer](#)

Anomaly detection inputs and outputs

The input features must be a fixed-sized vector of [Single](#).

This trainer outputs the following:

OUTPUT NAME	TYPE	DESCRIPTION
Score	Single	The non-negative, unbounded score that was calculated by the anomaly detection model
PredictedLabel	Boolean	A true/false value representing whether the input is an anomaly (PredictedLabel=true) or not (PredictedLabel=false)

Ranking

A ranking task constructs a ranker from a set of labeled examples. This example set consists of instance groups that can be scored with a given criteria. The ranking labels are { 0, 1, 2, 3, 4 } for each instance. The ranker is trained to rank new instance groups with unknown scores for each instance. ML.NET ranking learners are [machine learned ranking](#) based.

Ranking training algorithms

You can train a ranking model with the following algorithms:

- [LightGbmRankingTrainer](#)
- [FastTreeRankingTrainer](#)

Ranking input and outputs

The input label data type must be [key](#) type or [Single](#). The value of the label determines relevance, where higher values indicate higher relevance. If the label is a [key](#) type, then the key index is the relevance value, where the smallest index is the least relevant. If the label is a [Single](#), larger values indicate higher relevance.

The feature data must be a fixed size vector of [Single](#) and input row group column must be [key](#) type.

This trainer outputs the following:

OUTPUT NAME	TYPE	DESCRIPTION
Score	Single	The unbounded score that was calculated by the model to determine the prediction

Recommendation

A recommendation task enables producing a list of recommended products or services. ML.NET uses [Matrix factorization \(MF\)](#), a [collaborative filtering](#) algorithm for recommendations when you have historical product rating data in your catalog. For example, you have historical movie rating data for your users and want to recommend other movies they are likely to watch next.

Recommendation training algorithms

You can train a recommendation model with the following algorithm:

- [MatrixFactorizationTrainer](#)

Forecasting

The forecasting task use past time-series data to make predictions about future behavior. Scenarios applicable to forecasting include weather forecasting, seasonal sales predictions, and predictive maintenance.

Forecasting trainers

You can train a forecasting model with the following algorithm:

[ForecastBySsa](#)

Image Classification

A [supervised machine learning](#) task that is used to predict the class (category) of an image. The input is a set of labeled examples. Each label normally starts as text. It is then run through the TermTransform, which converts it to the Key (numeric) type. The output of the image classification algorithm is a classifier, which you can use to predict the class of new images. The image classification task is a type of multiclass classification. Examples of image classification scenarios include:

- Determining the breed of a dog as a "Siberian Husky", "Golden Retriever", "Poodle", etc.
- Determining if a manufacturing product is defective or not.
- Determining what types of flowers as "Rose", "Sunflower", etc.

Image classification trainers

You can train an image classification model using the following training algorithms:

- [ImageClassificationTrainer](#)

Image classification inputs and outputs

The input label column data must be [key](#) type. The feature column must be a variable-sized vector of [Byte](#).

This trainer outputs the following columns:

OUTPUT NAME	TYPE	DESCRIPTION
Score	Single	The scores of all classes. Higher value means higher probability to fall into the associated class. If the i-th element has the largest value, the predicted label index would be i. Note that i is zero-based index.
PredictedLabel	Key type	The predicted label's index. If its value is i, the actual label would be the i-th category in the key-valued input label type.

Object Detection

A [supervised machine learning](#) task that is used to predict the class (category) of an image but also gives a bounding box to where that category is within the image. Instead of classifying a single object in an image, object detection can detect multiple objects within an image. Examples of object detection include:

- Detecting cars, signs, or people on images of a road.
- Detecting defects on images of products.
- Detecting areas of concern on X-Ray images.

Object detection model training is currently only available in [Model Builder](#) using Azure Machine Learning.

How to choose an ML.NET algorithm

10/14/2022 • 6 minutes to read • [Edit Online](#)

For each [ML.NET task](#), there are multiple training algorithms to choose from. Which one to choose depends on the problem you are trying to solve, the characteristics of your data, and the compute and storage resources you have available. It is important to note that training a machine learning model is an iterative process. You might need to try multiple algorithms to find the one that works best.

Algorithms operate on **features**. Features are numerical values computed from your input data. They are optimal inputs for machine learning algorithms. You transform your raw input data into features using one or more [data transforms](#). For example, text data is transformed into a set of word counts and word combination counts. Once the features have been extracted from a raw data type using data transforms, they are referred to as **featurized**. For example, featurized text, or featurized image data.

Trainer = Algorithm + Task

An algorithm is the math that executes to produce a **model**. Different algorithms produce models with different characteristics.

With ML.NET, the same algorithm can be applied to different tasks. For example, Stochastic Dual Coordinate Ascent can be used for Binary Classification, Multiclass Classification, and Regression. The difference is in how the output of the algorithm is interpreted to match the task.

For each algorithm/task combination, ML.NET provides a component that executes the training algorithm and makes the interpretation. These components are called trainers. For example, the [SdcaRegressionTrainer](#) uses the [StochasticDualCoordinatedAscent](#) algorithm applied to the [Regression](#) task.

Linear algorithms

Linear algorithms produce a model that calculates **scores** from a linear combination of the input data and a set of **weights**. The weights are parameters of the model estimated during training.

Linear algorithms work well for features that are [linearly separable](#).

Before training with a linear algorithm, the features should be normalized. This prevents one feature from having more influence over the result than others.

In general, linear algorithms are scalable, fast, cheap to train, and cheap to predict. They scale by the number of features and approximately by the size of the training data set.

Linear algorithms make multiple passes over the training data. If your dataset fits into memory, then adding a [cache checkpoint](#) to your ML.NET pipeline before appending the trainer will make the training run faster.

Averaged perceptron

Best for text classification.

TRAINER	TASK	ONNX EXPORTABLE
AveragedPerceptronTrainer	Binary classification	Yes

Stochastic dual coordinated ascent

Tuning not needed for good default performance.

TRAINER	TASK	ONNX EXPORTABLE
SdcaLogisticRegressionBinaryTrainer	Binary classification	Yes
SdcaNonCalibratedBinaryTrainer	Binary classification	Yes
SdcaMaximumEntropyMulticlassTrainer	Multiclass classification	Yes
SdcaNonCalibratedMulticlassTrainer	Multiclass classification	Yes
SdcaRegressionTrainer	Regression	Yes

L-BFGS

Use when number of features is large. Produces logistic regression training statistics, but doesn't scale as well as the AveragedPerceptronTrainer.

TRAINER	TASK	ONNX EXPORTABLE
LbfgsLogisticRegressionBinaryTrainer	Binary classification	Yes
LbfgsMaximumEntropyMulticlassTrainer	Multiclass classification	Yes
LbfgsPoissonRegressionTrainer	Regression	Yes

Symbolic stochastic gradient descent

Fastest and most accurate linear binary classification trainer. Scales well with number of processors.

TRAINER	TASK	ONNX EXPORTABLE
SymbolicSgdLogisticRegressionBinaryTrainer	Binary classification	Yes

Online gradient descent

Implements the standard (non-batch) stochastic gradient descent, with a choice of loss functions, and an option to update the weight vector using the average of the vectors seen over time.

TRAINER	TASK	ONNX EXPORTABLE
OnlineGradientDescentTrainer	Regression	Yes

Decision tree algorithms

Decision tree algorithms create a model that contains a series of decisions: effectively a flow chart through the data values.

Features do not need to be linearly separable to use this type of algorithm. And features do not need to be normalized, because the individual values in the feature vector are used independently in the decision process.

Decision tree algorithms are generally very accurate.

Except for Generalized Additive Models (GAMs), tree models can lack explainability when the number of features is large.

Decision tree algorithms take more resources and do not scale as well as linear ones do. They do perform well on datasets that can fit into memory.

Boosted decision trees are an ensemble of small trees where each tree scores the input data and passes the score onto the next tree to produce a better score, and so on, where each tree in the ensemble improves on the previous.

Light gradient boosted machine

Fastest and most accurate of the binary classification tree trainers. Highly tunable.

TRAINER	TASK	ONNX EXPORTABLE
LightGbmBinaryTrainer	Binary classification	Yes
LightGbmMulticlassTrainer	Multiclass classification	Yes
LightGbmRegressionTrainer	Regression	Yes
LightGbmRankingTrainer	Ranking	No

Fast tree

Use for featurized image data. Resilient to unbalanced data. Highly tunable.

TRAINER	TASK	ONNX EXPORTABLE
FastTreeBinaryTrainer	Binary classification	Yes
FastTreeRegressionTrainer	Regression	Yes
FastTreeTweedieTrainer	Regression	Yes
FastTreeRankingTrainer	Ranking	No

Fast forest

Works well with noisy data.

TRAINER	TASK	ONNX EXPORTABLE
FastForestBinaryTrainer	Binary classification	Yes
FastForestRegressionTrainer	Regression	Yes

Generalized additive model (GAM)

Best for problems that perform well with tree algorithms but where explainability is a priority.

TRAINER	TASK	ONNX EXPORTABLE
GamBinaryTrainer	Binary classification	No
GamRegressionTrainer	Regression	No

Matrix factorization

Matrix Factorization

Used for [collaborative filtering](#) in recommendation.

TRAINER	TASK	ONNX EXPORTABLE
MatrixFactorizationTrainer	Recommendation	No

Field Aware Factorization Machine

Best for sparse categorical data, with large datasets.

TRAINER	TASK	ONNX EXPORTABLE
FieldAwareFactorizationMachineTrainer	Binary classification	No

Meta algorithms

These trainers create a multiclass trainer from a binary trainer. Use with [AveragedPerceptronTrainer](#), [LbfgsLogisticRegressionBinaryTrainer](#), [SymbolicSgdLogisticRegressionBinaryTrainer](#), [LightGbmBinaryTrainer](#), [FastTreeBinaryTrainer](#), [FastForestBinaryTrainer](#), [GamBinaryTrainer](#).

One versus all

This multiclass classifier trains one binary classifier for each class, which distinguishes that class from all other classes. Is limited in scale by the number of classes to categorize.

TRAINER	TASK	ONNX EXPORTABLE
OneVersusAllTrainer	Multiclass classification	Yes

Pairwise coupling

This multiclass classifier trains a binary classification algorithm on each pair of classes. Is limited in scale by the number of classes, as each combination of two classes must be trained.

TRAINER	TASK	ONNX EXPORTABLE
PairwiseCouplingTrainer	Multiclass classification	No

K-Means

Used for clustering.

TRAINER	TASK	ONNX EXPORTABLE
KMeansTrainer	Clustering	Yes

Principal component analysis

Used for anomaly detection.

TRAINER	TASK	ONNX EXPORTABLE
RandomizedPcaTrainer	Anomaly detection	No

Naive Bayes

Use this multi-class classification algorithm when the features are independent, and the training dataset is small.

TRAINER	TASK	ONNX EXPORTABLE
NaiveBayesMulticlassTrainer	Multiclass classification	Yes

Prior Trainer

Use this binary classification algorithm to baseline the performance of other trainers. To be effective, the metrics of the other trainers should be better than the prior trainer.

TRAINER	TASK	ONNX EXPORTABLE
PriorTrainer	Binary classification	Yes

Support vector machines

Support vector machines (SVMs) are an extremely popular and well-researched class of supervised learning models, which can be used in linear and non-linear classification tasks.

Recent research has focused on ways to optimize these models to efficiently scale to larger training sets.

Linear SVM

Predicts a target using a linear binary classification model trained over boolean labeled data. Alternates between stochastic gradient descent steps and projection steps.

TRAINER	TASK	ONNX EXPORTABLE
LinearSvmTrainer	Binary classification	Yes

Local Deep SVM

Predicts a target using a non-linear binary classification model. Reduces the prediction time cost; the prediction cost grows logarithmically with the size of the training set, rather than linearly, with a tolerable loss in classification accuracy.

TRAINER	TASK	ONNX EXPORTABLE
LdSvmTrainer	Binary classification	Yes

Ordinary least squares

Ordinary least squares (OLS) is one of the most commonly used techniques in linear regression.

Ordinary least squares refers to the loss function, which computes error as the sum of the square of distance from the actual value to the predicted line, and fits the model by minimizing the squared error. This method assumes a strong linear relationship between the inputs and the dependent variable.

TRAINER	TASK	ONNX EXPORTABLE
OlsTrainer	Regression	Yes

Data transformations

10/14/2022 • 6 minutes to read • [Edit Online](#)

Data transformations are used to:

- prepare data for model training
- apply an imported model in TensorFlow or ONNX format
- post-process data after it has been passed through a model

The transformations in this guide return classes that implement the [IEstimator](#) interface. Data transformations can be chained together. Each transformation both expects and produces data of specific types and formats, which are specified in the linked reference documentation.

Some data transformations require training data to calculate their parameters. For example: the [NormalizeMeanVariance](#) transformer calculates the mean and variance of the training data during the `Fit()` operation, and uses those parameters in the `Transform()` operation.

Other data transformations don't require training data. For example: the [ConvertToGrayscale](#) transformation can perform the `Transform()` operation without having seen any training data during the `Fit()` operation.

Column mapping and grouping

TRANSFORM	DEFINITION	ONNX EXPORTABLE
Concatenate	Concatenate one or more input columns into a new output column	Yes
CopyColumns	Copy and rename one or more input columns	Yes
DropColumns	Drop one or more input columns	Yes
SelectColumns	Select one or more columns to keep from the input data	Yes

Normalization and scaling

TRANSFORM	DEFINITION	ONNX EXPORTABLE
NormalizeMeanVariance	Subtract the mean (of the training data) and divide by the variance (of the training data)	Yes
NormalizeLogMeanVariance	Normalize based on the logarithm of the training data	Yes
NormalizeLpNorm	Scale input vectors by their l_p-norm , where p is 1, 2 or infinity. Defaults to the l2 (Euclidean distance) norm	Yes

TRANSFORM	DEFINITION	ONNX EXPORTABLE
NormalizeGlobalContrast	Scale each value in a row by subtracting the mean of the row data and divide by either the standard deviation or l2-norm (of the row data), and multiply by a configurable scale factor (default 2)	Yes
NormalizeBinning	Assign the input value to a bin index and divide by the number of bins to produce a float value between 0 and 1. The bin boundaries are calculated to evenly distribute the training data across bins	Yes
NormalizeSupervisedBinning	Assign the input value to a bin based on its correlation with label column	Yes
NormalizeMinMax	Scale the input by the difference between the minimum and maximum values in the training data	Yes
NormalizeRobustScaling	Scale each value using statistics that are robust to outliers that will center the data around 0 and scales the data according to the quantile range.	Yes

Conversions between data types

TRANSFORM	DEFINITION	ONNX EXPORTABLE
ConvertType	Convert the type of an input column to a new type	Yes
MapValue	Map values to keys (categories) based on the supplied dictionary of mappings	No
MapValueToKey	Map values to keys (categories) by creating the mapping from the input data	Yes
MapKeyToValue	Convert keys back to their original values	Yes
MapKeyToVector	Convert keys back to vectors of original values	Yes
MapKeyToBinaryVector	Convert keys back to a binary vector of original values	No
Hash	Hash the value in the input column	Yes

Text transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
FeaturizeText	Transform a text column into a float array of normalized ngrams and char-grams counts	No
TokenizeIntoWords	Split one or more text columns into individual words	Yes
TokenizeIntoCharactersAsKeys	Split one or more text columns into individual characters floats over a set of topics	Yes
NormalizeText	Change case, remove diacritical marks, punctuation marks, and numbers	Yes
ProduceNgrams	Transform text column into a bag of counts of ngrams (sequences of consecutive words)	Yes
ProduceWordBags	Transform text column into a bag of counts of ngrams vector	Yes
ProduceHashedNgrams	Transform text column into a vector of hashed ngram counts	No
ProduceHashedWordBags	Transform text column into a bag of hashed ngram counts	Yes
RemoveDefaultStopWords	Remove default stop words for the specified language from input columns	Yes
RemoveStopWords	Removes specified stop words from input columns	Yes
LatentDirichletAllocation	Transform a document (represented as a vector of floats) into a vector of floats over a set of topics	Yes
ApplyWordEmbedding	Convert vectors of text tokens into sentence vectors using a pre-trained model	Yes

Image transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
ConvertToGrayscale	Convert an image to grayscale	No
ConvertToImage	Convert a vector of pixels to ImageDataViewType	No
ExtractPixels	Convert pixels from input image into a vector of numbers	No

TRANSFORM	DEFINITION	ONNX EXPORTABLE
LoadImages	Load images from a folder into memory	No
LoadRawImageBytes	Loads images of raw bytes into a new column.	No
ResizeImages	Resize images	No
DnnFeaturizeImage	Applies a pre-trained deep neural network (DNN) model to transform an input image into a feature vector	No

Categorical data transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
OneHotEncoding	Convert one or more text columns into one-hot encoded vectors	Yes
OneHotHashEncoding	Convert one or more text columns into hash-based one-hot encoded vectors	No

Time series data transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
DetectAnomalyBySrCnn	Detect anomalies in the input time series data using the Spectral Residual (SR) algorithm	No
DetectChangePointBySsa	Detect change points in time series data using singular spectrum analysis (SSA)	No
DetectIdChangePoint	Detect change points in independent and identically distributed (IID) time series data using adaptive kernel density estimations and martingale scores	No
ForecastBySsa	Forecast time series data using singular spectrum analysis (SSA)	No
DetectSpikeBySsa	Detect spikes in time series data using singular spectrum analysis (SSA)	No
DetectIdSpike	Detect spikes in independent and identically distributed (IID) time series data using adaptive kernel density estimations and martingale scores	No

TRANSFORM	DEFINITION	ONNX EXPORTABLE
DetectEntireAnomalyBySrCnn	Detect anomalies for the entire input data using the SRCNN algorithm.	No
DetectSeasonality	Detect seasonality using fourier analysis.	No
LocalizeRootCause	Localizes root cause from time series input using a decision tree algorithm.	No
LocalizeRootCauses	Localizes root causes from tie series input.	No

Missing values

TRANSFORM	DEFINITION	ONNX EXPORTABLE
IndicateMissingValues	Create a new boolean output column, the value of which is true when the value in the input column is missing	Yes
ReplaceMissingValues	Create a new output column, the value of which is set to a default value if the value is missing from the input column, and the input value otherwise	Yes

Feature selection

TRANSFORM	DEFINITION	ONNX EXPORTABLE
SelectFeaturesBasedOnCount	Select features whose non-default values are greater than a threshold	Yes
SelectFeaturesBasedOnMutualInformation	Select the features on which the data in the label column is most dependent	Yes

Feature transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
ApproximatedKernelMap	Map each input vector onto a lower dimensional feature space, where inner products approximate a kernel function, so that the features can be used as inputs to the linear algorithms	No
ProjectToPrincipalComponents	Reduce the dimensions of the input feature vector by applying the Principal Component Analysis algorithm	

Explainability transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
CalculateFeatureContribution	Calculate contribution scores for each element of a feature vector	No

Calibration transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
Platt(String, String, String)	Transforms a binary classifier raw score into a class probability using logistic regression with parameters estimated using the training data	Yes
Platt(Double, Double, String)	Transforms a binary classifier raw score into a class probability using logistic regression with fixed parameters	Yes
Naive	Transforms a binary classifier raw score into a class probability by assigning scores to bins, and calculating the probability based on the distribution among the bins	Yes
Isotonic	Transforms a binary classifier raw score into a class probability by assigning scores to bins, where the position of boundaries and the size of bins are estimated using the training data	No

Deep learning transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
ApplyOnnxModel	Transform the input data with an imported ONNX model	No
LoadTensorFlowModel	Transform the input data with an imported TensorFlow model	No

Custom transformations

TRANSFORM	DEFINITION	ONNX EXPORTABLE
FilterByCustomPredicate	Drops rows where a specified predicate returns true.	No
FilterByStatefulCustomPredicate	Drops rows where a specified predicate returns true, but allows for a specified state.	No
CustomMapping	Transform existing columns onto new ones with a user-defined mapping	No

TRANSFORM	DEFINITION	ONNX EXPORTABLE
Expression	Apply an expression to transform columns into new ones	No

Evaluate your ML.NET model with metrics

10/14/2022 • 9 minutes to read • [Edit Online](#)

Understand the metrics used to evaluate an ML.NET model.

Evaluation metrics are specific to the type of machine learning task that a model performs.

For example, for the classification task, the model is evaluated by measuring how well a predicted category matches the actual category. And for clustering, evaluation is based on how close clustered items are to each other, and how much separation there is between the clusters.

Evaluation metrics for Binary Classification

METRICS	DESCRIPTION	LOOK FOR
Accuracy	Accuracy is the proportion of correct predictions with a test data set. It is the ratio of number of correct predictions to the total number of input samples. It works well if there are similar number of samples belonging to each class.	The closer to 1.00, the better. But exactly 1.00 indicates an issue (commonly: label/target leakage, overfitting, or testing with training data). When the test data is unbalanced (where most of the instances belong to one of the classes), the dataset is small, or scores approach 0.00 or 1.00, then accuracy doesn't really capture the effectiveness of a classifier and you need to check additional metrics.
AUC	aucROC or <i>Area under the curve</i> measures the area under the curve created by sweeping the true positive rate vs. the false positive rate.	The closer to 1.00, the better. It should be greater than 0.50 for a model to be acceptable. A model with AUC of 0.50 or less is worthless.
AUCPR	aucPR or <i>Area under the curve of a Precision-Recall curve</i> . Useful measure of success of prediction when the classes are imbalanced (highly skewed datasets).	The closer to 1.00, the better. High scores close to 1.00 show that the classifier is returning accurate results (high precision), and returning a majority of all positive results (high recall).
F1-score	F1 score also known as <i>balanced F-score or F-measure</i> . It's the harmonic mean of the precision and recall. F1 Score is helpful when you want to seek a balance between Precision and Recall.	The closer to 1.00, the better. An F1 score reaches its best value at 1.00 and worst score at 0.00. It tells you how precise your classifier is.

For further details on binary classification metrics read the following articles:

- [Accuracy, Precision, Recall, or F1?](#)
- [Binary Classification Metrics class](#)
- [The Relationship Between Precision-Recall and ROC Curves](#)

Evaluation metrics for Multi-class Classification

METRICS	DESCRIPTION	LOOK FOR
Micro-Accuracy	Micro-average Accuracy aggregates the contributions of all classes to compute the average metric. It is the fraction of instances predicted correctly. The micro-average does not take class membership into account. Basically, every sample-class pair contributes equally to the accuracy metric.	The closer to 1.00, the better. In a multi-class classification task, micro-accuracy is preferable over macro-accuracy if you suspect there might be class imbalance (i.e you may have many more examples of one class than of other classes).
Macro-Accuracy	Macro-average Accuracy is the average accuracy at the class level. The accuracy for each class is computed and the macro-accuracy is the average of these accuracies. Basically, every class contributes equally to the accuracy metric. Minority classes are given equal weight as the larger classes. The macro-average metric gives the same weight to each class, no matter how many instances from that class the dataset contains.	The closer to 1.00, the better. It computes the metric independently for each class and then takes the average (hence treating all classes equally)
Log-loss	Logarithmic loss measures the performance of a classification model where the prediction input is a probability value between 0.00 and 1.00. Log-loss increases as the predicted probability diverges from the actual label.	The closer to 0.00, the better. A perfect model would have a log-loss of 0.00. The goal of our machine learning models is to minimize this value.
Log-Loss Reduction	Logarithmic loss reduction can be interpreted as the advantage of the classifier over a random prediction.	Ranges from -inf and 1.00, where 1.00 is perfect predictions and 0.00 indicates mean predictions. For example, if the value equals 0.20, it can be interpreted as "the probability of a correct prediction is 20% better than random guessing"

Micro-accuracy is generally better aligned with the business needs of ML predictions. If you want to select a single metric for choosing the quality of a multiclass classification task, it should usually be micro-accuracy.

Example, for a support ticket classification task: (maps incoming tickets to support teams)

- Micro-accuracy—how often does an incoming ticket get classified to the right team?
- Macro-accuracy—for an average team, how often is an incoming ticket correct for their team?

Macro-accuracy overweights small teams in this example; a small team that gets only 10 tickets per year counts as much as a large team with 10k tickets per year. Micro-accuracy in this case correlates better with the business need of, "how much time/money can the company save by automating my ticket routing process".

For further details on multi-class classification metrics read the following articles:

- [Micro- and Macro-average of Precision, Recall, and F-Score](#)
- [Multiclass Classification with Imbalanced Dataset](#)

Evaluation metrics for Regression and Recommendation

Both the regression and recommendation tasks predict a number. In the case of regression, the number can be any output property that is influenced by the input properties. For recommendation, the number is usually a rating value (between 1 and 5 for example), or a yes/no recommendation (represented by 1 and 0 respectively).

METRIC	DESCRIPTION	LOOK FOR
R-Squared	<p>R-squared (R2), or <i>Coefficient of determination</i> represents the predictive power of the model as a value between -inf and 1.00. 1.00 means there is a perfect fit, and the fit can be arbitrarily poor so the scores can be negative. A score of 0.00 means the model is guessing the expected value for the label. A negative R2 value indicates the fit does not follow the trend of the data and the model performs worse than random guessing. This is only possible with non-linear regression models or constrained linear regression. R2 measures how close the actual test data values are to the predicted values.</p>	<p>The closer to 1.00, the better quality. However, sometimes low R-squared values (such as 0.50) can be entirely normal or good enough for your scenario and high R-squared values are not always good and be suspicious.</p>
Absolute-loss	<p>Absolute-loss or <i>Mean absolute error (MAE)</i> measures how close the predictions are to the actual outcomes. It is the average of all the model errors, where model error is the absolute distance between the predicted label value and the correct label value. This prediction error is calculated for each record of the test data set. Finally, the mean value is calculated for all recorded absolute errors.</p>	<p>The closer to 0.00, the better quality. The mean absolute error uses the same scale as the data being measured (is not normalized to specific range). Absolute-loss, Squared-loss, and RMS-loss can only be used to make comparisons between models for the same dataset or dataset with a similar label value distribution.</p>
Squared-loss	<p>Squared-loss or <i>Mean Squared Error (MSE)</i>, also called <i>Mean Squared Deviation (MSD)</i>, tells you how close a regression line is to a set of test data values by taking the distances from the points to the regression line (these distances are the errors E) and squaring them. The squaring gives more weight to larger differences.</p>	<p>It is always non-negative, and values closer to 0.00 are better. Depending on your data, it may be impossible to get a very small value for the mean squared error.</p>

METRIC	DESCRIPTION	LOOK FOR
RMS-loss	RMS-loss or <i>Root Mean Squared Error (RMSE)</i> (also called <i>Root Mean Square Deviation, RMSD</i>), measures the difference between values predicted by a model and the values observed from the environment that is being modeled. RMS-loss is the square root of Squared-loss and has the same units as the label, similar to the absolute-loss though giving more weight to larger differences. Root mean square error is commonly used in climatology, forecasting, and regression analysis to verify experimental results.	It is always non-negative, and values closer to 0.00 are better . RMSD is a measure of accuracy, to compare forecasting errors of different models for a particular dataset and not between datasets, as it is scale-dependent.

For further details on regression metrics, read the following articles:

- [Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit?](#)
- [How To Interpret R-squared in Regression Analysis](#)
- [R-Squared Definition](#)
- [The Coefficient of Determination and the Assumptions of Linear Regression Models](#)
- [Mean Squared Error Definition](#)
- [What are Mean Squared Error and Root Mean Squared Error?](#)

Evaluation metrics for Clustering

METRIC	DESCRIPTION	LOOK FOR
Average Distance	Average of the distance between data points and the center of their assigned cluster. The average distance is a measure of proximity of the data points to cluster centroids. It's a measure of how 'tight' the cluster is.	Values closer to 0 are better. The closer to zero the average distance is, the more clustered the data is. Note though, that this metric will decrease if the number of clusters is increased, and in the extreme case (where each distinct data point is its own cluster) it will be equal to zero.
Davies Bouldin Index	The average ratio of within-cluster distances to between-cluster distances. The tighter the cluster, and the further apart the clusters are, the lower this value is.	Values closer to 0 are better. Clusters that are farther apart and less dispersed will result in a better score.
Normalized Mutual Information	Can be used when the training data used to train the clustering model also comes with ground truth labels (that is, supervised clustering). The Normalized Mutual Information metric measures whether similar data points get assigned to the same cluster and disparate data points get assigned to different clusters. Normalized mutual information is a value between 0 and 1	Values closer to 1 are better

Evaluation metrics for Ranking

METRIC	DESCRIPTION	LOOK FOR
Discounted Cumulative Gains	Discounted cumulative gain (DCG) is a measure of ranking quality. It is derived from two assumptions. One: Highly relevant items are more useful when appearing higher in ranking order. Two: Usefulness tracks relevance that is, the higher the relevance, the more useful an item. Discounted cumulative gain is calculated for a particular position in the ranking order. It sums the relevance grading divided by the logarithm of the ranking index up to the position of interest. It is calculated using $\sum_{i=0}^p \frac{\text{rel}_i}{\log_{e}(i+1)}$. Relevance gradings are provided to a ranking training algorithm as ground truth labels. One DCG value is provided for each position in the ranking table, hence the name Discounted Cumulative Gains.	Higher values are better
Normalized Discounted Cumulative Gains	Normalizing DCG allows the metric to be compared for ranking lists of different lengths	Values closer to 1 are better

Evaluation metrics for Anomaly Detection

METRIC	DESCRIPTION	LOOK FOR
Area Under ROC Curve	Area under the receiver operator curve measures how well the model separates anomalous and usual data points.	Values closer to 1 are better. Only values greater than 0.5 demonstrate effectiveness of the model. Values of 0.5 or below indicate that the model is no better than randomly allocating the inputs to anomalous and usual categories
Detection Rate At False Positive Count	Detection rate at false positive count is the ratio of the number of correctly identified anomalies to the total number of anomalies in a test set, indexed by each false positive. That is, there is a value for detection rate at false positive count for each false positive item.	Values closer to 1 are better. If there are no false positives, then this value is 1

Improve your ML.NET model

10/14/2022 • 2 minutes to read • [Edit Online](#)

Learn how to improve your ML.NET model.

Reframe the problem

Sometimes, improving a model may have nothing to do with the data or techniques used to train the model. Instead, it may just be that the wrong question is being asked. Consider looking at the problem from different angles and leverage the data to extract latent indicators and hidden relationships in order to refine the question.

Provide more data samples

Like humans, the more training algorithms get, the likelihood of better performance increases. One way to improve model performance is to provide more training data samples to the algorithms. The more data it learns from, the more cases it is able to correctly identify.

Add context to the data

The meaning of a single data point can be difficult to interpret. Building context around the data points helps algorithms as well as subject matter experts better make decisions. For example, the fact that a house has three bedrooms does not on its own give a good indication of its price. However, if you add context and now know that it is in a suburban neighborhood outside of a major metropolitan area where average age is 38, average household income is \$80,000 and schools are in the top 20th percentile then the algorithm has more information to base its decisions on. All of this context can be added as input to the machine learning model as features.

Use meaningful data and features

Although more data samples and features can help improve the accuracy of the model, they may also introduce noise since not all data and features are meaningful. Therefore, it is important to understand which features are the ones that most heavily impact decisions made by the algorithm. Using techniques like Permutation Feature Importance (PFI) can help identify those salient features and not only help explain the model but also use the output as a feature selection method to reduce the amount of noisy features going into the training process.

For more information about using PFI, see [Explain model predictions using Permutation Feature Importance](#).

Cross-validation

Cross-validation is a training and model evaluation technique that splits the data into several partitions and trains multiple algorithms on these partitions. This technique improves the robustness of the model by holding out data from the training process. In addition to improving performance on unseen observations, in data-constrained environments it can be an effective tool for training models with a smaller dataset.

Visit the following link to learn [how to use cross validation in ML.NET](#)

Hyperparameter tuning

Training machine learning models is an iterative and exploratory process. For example, what is the optimal number of clusters when training a model using the K-Means algorithm? The answer depends on many factors

such as the structure of the data. Finding that number would require experimenting with different values for k and then evaluating performance to determine which value is best. The practice of tuning the parameters that guide the training process to find an optimal model is known as hyperparameter tuning.

Choose a different algorithm

Machine learning tasks like regression and classification contain various algorithm implementations. It may be the case that the problem you are trying to solve and the way your data is structured does not fit well into the current algorithm. In such case, consider using a different algorithm for your task to see if it learns better from your data.

The following link provides more [guidance on which algorithm to choose](#).

Load training data into Model Builder

10/14/2022 • 5 minutes to read • [Edit Online](#)

Learn how to load your training datasets from a file or a SQL Server database for use in one of the Model Builder scenarios for ML.NET. Model Builder scenarios can use SQL Server databases, image files, and CSV or TSV file formats as training data.

Model Builder only accepts TSV, CSV, and TXT files with comma, tab, and semi-colon delimiters and PNG and JPG images.

Model Builder scenarios

Model Builder helps you create models for the following machine learning scenarios:

- Data classification (binary & multiclass classification): Classify text data into two or more categories.
- Value prediction (regression): Predict a numeric value.
- Image classification (deep learning): Classify images into two or more categories.
- Recommendation (recommendation): Produce a list of suggested items for a particular user.
- Object detection (deep learning): Detect and identify object in images. This can find one or more objects and label them accordingly.

This article covers classification and regression with textual or numerical data, image classification, and object detection scenarios.

Load text or numeric data from a file

You can load text or numeric data from a file into Model Builder. It accepts comma-delimited (CSV) or tab-delimited (TSV) file formats.

1. In the data step of Model Builder, select **File** as the data source type.
2. Select the **Browse** button next to the text box, and use File Explorer to browse and select the data file.
3. Choose a category in the **Column to predict (Label)** dropdown.

NOTE

(Optional) data classification scenarios: If the data type of your label column (the value in the "Column to predict (Label)" dropdown) is set to Boolean (True/False), a binary classification algorithm is used in your model training pipeline. Otherwise, a multiclass classification trainer is used. Use **Advanced data options** to modify the data type for your label column and inform Model Builder which type of trainer it should use for your data.

4. Update the data in the **Advanced data options** link to set column settings or to update the data formatting.

You're done setting up your data source file for Model Builder. Click the **Next step** button to move to the next step in Model Builder.

Load data from a SQL Server database

Model Builder supports loading data from local and remote SQL Server databases.

Local database file

To load data from a SQL Server database file into Model Builder:

1. In the data step of Model Builder, select **SQL Server** as the data source type.
2. Select the **Choose data source** button.
 - a. In the **Choose Data Source** dialog, select **Microsoft SQL Server Database File**.
 - b. Uncheck the **Always use this selection** checkbox and select **Continue**
 - c. In the **Connection Properties** dialog, select **Browse** and select the downloaded .MDF file.
 - d. Select **OK**
3. Choose the dataset name from the **Table Name** dropdown.
4. From the **Column to predict (Label)** dropdown, choose the data category on which you want to make a prediction.

NOTE

(Optional) data classification scenarios: If the data type of your label column (the value in the "Column to predict (Label)" dropdown) is set to Boolean (True/False), a binary classification algorithm is used in your model training pipeline. Otherwise, a multiclass classification trainer is used. Use **Advanced data options** to modify the data type for your label column and inform Model Builder which type of trainer it should use for your data.

5. Update the data in the **Advanced data options** link to set column settings or to update the data formatting.

Remote database

To load data from a SQL Server database connection into Model Builder:

1. In the data step of Model Builder, select **SQL Server** as the data source type.
2. Select the **Choose data source** button.
 - a. In the **Choose Data Source** dialog, select **Microsoft SQL Server**.
3. In the **Connection Properties** dialog, input the properties of your Microsoft SQL database.
 - a. Provide the server name that has the table that you want to connect to.
 - b. Set up the authentication to the server. If **SQL Server Authentication** is selected, input the server's username and password.
 - c. Select what database to connect to in the **Select or enter a database name** dropdown. This should auto-populate if the server name and log in information are correct.
 - d. Select **OK**
4. Choose the dataset name from the **Table Name** dropdown.
5. From the **Column to predict (Label)** dropdown, choose the data category on which you want to make a prediction.

NOTE

(Optional) data classification scenarios: If the data type of your label column (the value in the "Column to predict (Label)" dropdown) is set to Boolean (True/False), a binary classification algorithm is used in your model training pipeline. Otherwise, a multiclass classification trainer is used. Use **Advanced data options** to modify the data type for your label column and inform Model Builder which type of trainer it should use for your data.

6. Update the data in the **Advanced data options** link to set column settings or to update the data formatting.

You're done setting up your data source file for Model Builder. Click the **Next step** button link to move to the next step in Model Builder.

Set up image classification data files

Model Builder expects image classification data to be JPG or PNG files organized in folders that correspond to the categories of the classification.

To load images into Model Builder, provide the path to a single top-level directory:

- This top-level directory contains one subfolder for each of the categories to predict.
- Each subfolder contains the image files belonging to its category.

In the folder structure illustrated below, the top-level directory is *flower_photos*. There are five subdirectories corresponding to the categories you want to predict: daisy, dandelion, roses, sunflowers, and tulips. Each of these subdirectories contains images belonging to its respective category.

```
\---flower_photos
    +---daisy
        |   100080576_f52e8ee070_n.jpg
        |   102841525_bd6628ae3c.jpg
        |   105806915_a9c13e2106_n.jpg
        |
    +---dandelion
        |   10443973_aeb97513fc_m.jpg
        |   10683189_bd6e371b97.jpg
        |   10919961_0af657c4e8.jpg
        |
    +---roses
        |   102501987_3cdb8e5394_n.jpg
        |   110472418_87b6a3aa98_m.jpg
        |   118974357_0faa23cce9_n.jpg
        |
    +---sunflowers
        |   127192624_afaa3d9cb84.jpg
        |   145303599_2627e23815_n.jpg
        |   147804446_ef9244c8ce_m.jpg
        |
    \---tulips
        100930342_92e8746431_n.jpg
        107693873_86021ac4ea_n.jpg
        10791227_7168491604.jpg
```

Set up object detection image data files

Model Builder expects object detection image data to be in JSON format generated from [VoTT](#). The JSON file is located in the **vott-json-export** folder in the **Target Location** that is specified in the project settings.

The JSON file consists of the following information generated from VoTT:

- All tags that were created
- The image file locations
- The image bounding box information
- The tag associated with the image

For more information on preparing data for object detection, see [Generate object detection data from VoTT](#).

Next steps

Follow these tutorials to build machine learning apps with Model Builder:

- [Generate object detection data from VoTT](#)
- [Predict prices using regression](#)
- [Analyze sentiment in a web application using binary classification](#)

If you're training a model using code, [learn how to load data using the ML.NET API](#).

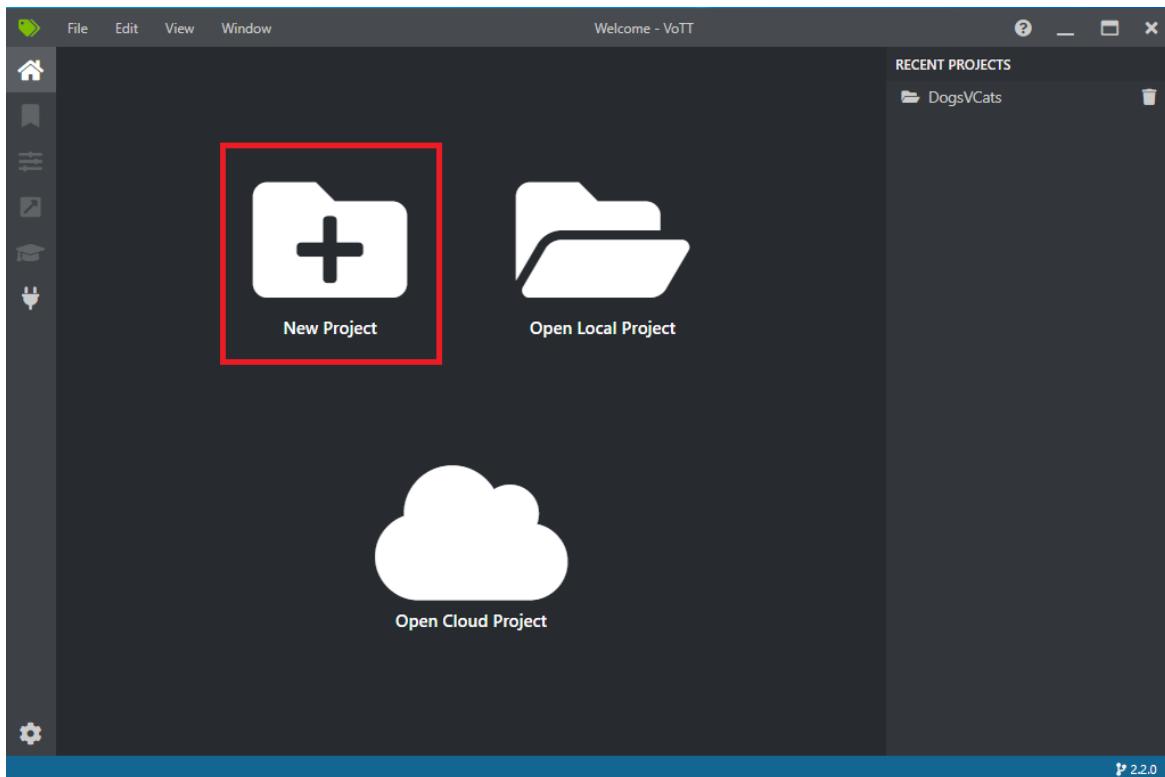
Label Images for Object Detection using VoTT

10/14/2022 • 2 minutes to read • [Edit Online](#)

Learn how to use VoTT (Visual Object Tagging Tool) to label images for object detection to be used within Model Builder.

Create a new VoTT Project

1. [Download VoTT](#) (Visual Object Tagging Tool).
2. Open VoTT and select **New Project**.

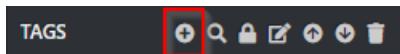


3. In **Project Settings**, change the **Display Name** to the name of your choosing.
4. Change the **Security Token** to *Generate New Security Token*.
5. Next to **Source Connection**, select **Add Connection**.
6. In **Connection Settings**, change the **Display Name** for the source connection to a name of your choosing, and select *Local File System* as the **Provider**. For the **Folder Path**, select the folder that contains the training images, and then select **Save Connection**.
7. In **Project Settings**, change the **Source Connection** to the connection you just created.
8. Change the **Target Connection** to the same connection as well.
9. Select **Save Project**.

Add tag and label images

You should now see a window with preview images of all the training images on the left, a preview of the selected image in the middle, and a **Tags** column on the right. This screen is the **Tags editor**.

1. Select the first (plus-shaped) icon in the **Tags** toolbar to add a new tag.



2. Name the tag and hit **Enter** on your keyboard.
3. Click and drag to draw a rectangle around each item in the image you want to tag. If the cursor does not let you draw a rectangle, try selecting the **Draw Rectangle** tool from the toolbar on the top, or use the keyboard shortcut **R**.
4. After drawing your rectangle, select the appropriate tag that you created in the previous steps to add the tag to the bounding box.
5. Click on the preview image for the next image in the dataset and repeat this process.
6. Continue steps 3-4 for every image.

Export your VoTT JSON

Once you have labeled all of your training images, you can export the file that will be used by Model Builder for training.

1. Select the fourth icon in the left toolbar (the one with the diagonal arrow in a box) to go to the **Export Settings**.
2. Leave the **Provider** as *VoTT JSON*.
3. Change the **Asset State** to *Only tagged Assets*.
4. Uncheck **Include Images**. If you include the images, then the training images will be copied to the export folder that is generated, which is not necessary.
5. Select **Save Export Settings**.



This export will create a new folder called *vott-json-export* in your project folder and will generate a JSON file named *ProjectName-export* in that new folder. You will use this JSON file for training an object detection model in Model Builder.

Next steps

- To use VoTT with an object detection scenario in Model Builder, see [Detect stop signs in images with Model Builder](#).

How to install ML.NET Model Builder

10/14/2022 • 2 minutes to read • [Edit Online](#)

Learn how to install ML.NET Model Builder to add machine learning to your .NET applications.

Prerequisites

- Visual Studio 2022 or Visual Studio 2019.
- .NET Core 3.1 SDK or later.

Limitations

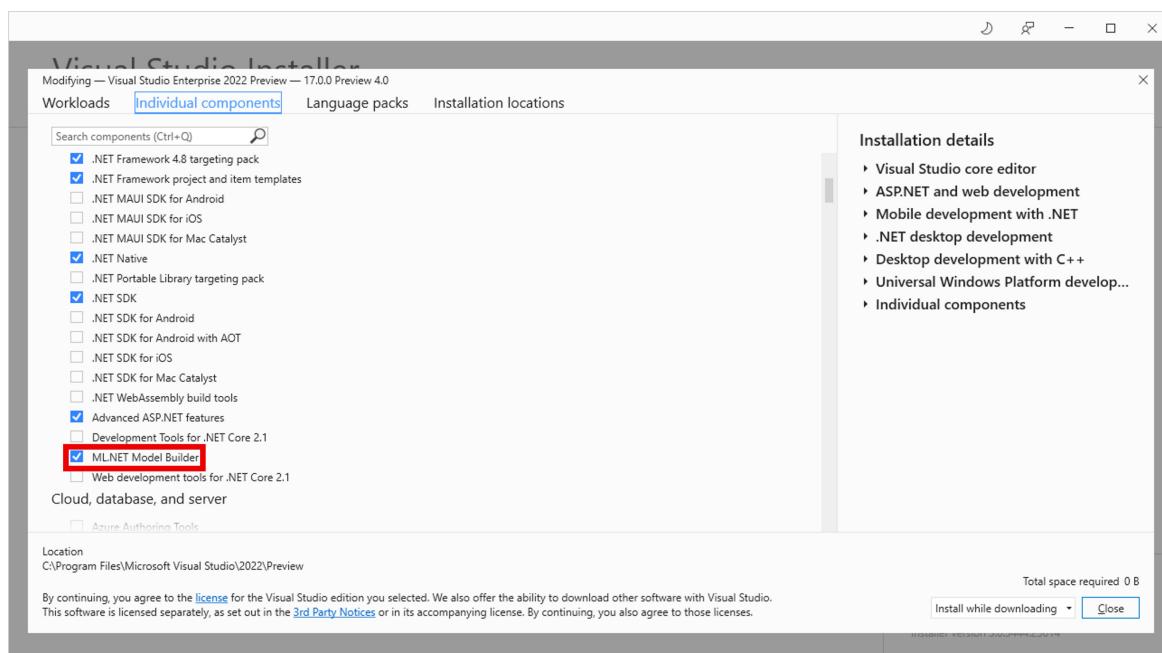
- ML.NET Model Builder Extension currently only works on Visual Studio on Windows.

Install Model Builder

ML.NET Model builder is built into Visual Studio. To enable it:

- [Visual Studio 2022](#)
- [Visual Studio 2019](#)

1. Open the Visual Studio Installer.
2. Select **Modify** to modify your current version of Visual Studio.
3. In the Visual Studio Installer, select the **Individual components** tab.
4. From the list of .NET components, check the **ML.NET Model Builder** checkbox.



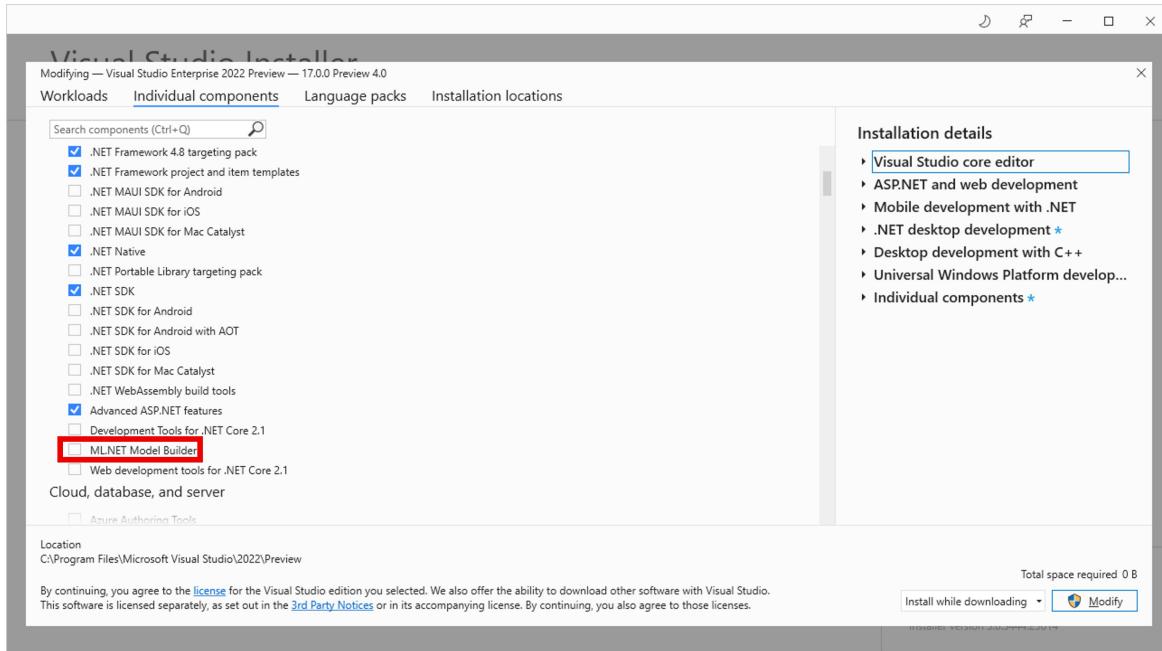
Uninstall Model Builder

ML.NET Model builder is built into Visual Studio. To disable it:

- [Visual Studio 2022](#)

- [Visual Studio 2019](#)

1. Open the Visual Studio Installer.
2. Select **Modify** to modify your current version of Visual Studio.
3. In the Visual Studio Installer, select the **Individual components** tab.
4. From the list of .NET components, **uncheck** the **ML.NET Model Builder** checkbox.



Upgrade Model Builder

Model Builder automatically updates when there's a new version.

However, if you'd prefer to manually install the latest version, either download it from [Visual Studio Marketplace](#) or use the Extensions Manager in Visual Studio. See [how to update a Visual Studio extension](#) for more information.

How to install GPU support in Model Builder

10/14/2022 • 2 minutes to read • [Edit Online](#)

Learn how to install the GPU drivers to use your GPU with Model Builder.

Prerequisites

- Model Builder Visual Studio extension. The extension is built into Visual Studio as of version 16.6.1.
- [Model Builder Visual Studio GPU support extension](#).
- At least one CUDA compatible GPU. For a list of compatible GPUs, see [NVIDIA's guide](#).
- NVIDIA developer account. If you don't have one, [create a free account](#).
- Make sure the appropriate [driver](#) is installed for the GPU.

Install dependencies

1. Install [CUDA v10.1](#). Make sure you install CUDA v10.1, not any other newer version.
2. Install [cuDNN v7.6.4 for CUDA 10.1](#). You cannot have multiple versions of cuDNN installed. After downloading cuDNN v7.6.4 zip file and unpacking it, copy `<CUDDNN_zip_files_path>\cuda\bin\cudnn64_7.dll` to `<YOUR_DRIVE>\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v10.1\bin`.

Troubleshooting

How do I know what GPU I have?

Follow instructions provided:

1. Right-click on desktop
2. If you see "NVIDIA Control Panel" or "NVIDIA Display" in the pop-up window, you have an NVIDIA GPU
3. Click on "NVIDIA Control Panel" or "NVIDIA Display" in the pop-up window
4. Look at "Graphics Card Information"
5. You will see the name of your NVIDIA GPU

I don't see NVIDIA Control Panel (or it fails to open) but I know I have an NVIDIA GPU.

1. Open Device Manager
2. Look at Display adapters
3. Install appropriate [driver](#) for your GPU.

How do I see what version of CUDA I have?

1. Open a PowerShell or command line window
2. Type in `nvcc --version`

How to install the ML.NET Command-Line Interface (CLI) tool

10/14/2022 • 3 minutes to read • [Edit Online](#)

Learn how to install the ML.NET CLI (command-line interface) on Windows, Mac, or Linux.

The ML.NET CLI generates good quality ML.NET models and source code using automated machine learning (AutoML) and a training dataset.

NOTE

This topic refers to ML.NET CLI and ML.NET AutoML, which are currently in Preview, and material may be subject to change.

Pre-requisites

- [.NET 6 SDK](#)
- (Optional) [Visual Studio 2022](#)

You can run the generated C# code projects with Visual Studio by pressing the F5 key or with `dotnet run` (.NET CLI).

Note: If after installing .NET SDK the `dotnet tool` command is not working, sign out from Windows and sign in again.

Install

The ML.NET CLI is installed like any other dotnet Global Tool. You use the `dotnet tool install --global` .NET CLI command.

The ML.NET CLI is available for Windows, macOS, and Linux. Depending on your processor architecture, choose the x64 or ARM version.

NOTE

ARM based versions of the ML.NET CLI don't support image classification scenarios.

- [Windows](#)
- [Mac](#)
- [Linux](#)

x64

```
dotnet tool install --global mlnet-win-x64
```

ARM

```
dotnet tool install --global mlnet-win-arm64
```

If the tool can't be installed (that is, if it is not available at the default NuGet feed), error messages are displayed. Check that the feeds you expected are being checked.

If installation is successful, a message is displayed showing the command used to call the tool and the version installed, similar to the following example:

```
You can invoke the tool using the following command: mlnet
Tool 'mlnet-<OS>-<ARCH>' (version 'X.X.X') was successfully installed.
```

The `os` and `ARCH` in this case should match the values for the operating system and processor architecture you selected when installing the ML.NET CLI.

You can confirm the installation was successful by typing the following command:

```
mlnet
```

You should see the help for available commands for the `mlnet` tool such as the 'classification' command.

IMPORTANT

If you are running Linux or macOS, note that if you're using a console other than Bash (for example, zsh, which is the new default for macOS), then you'll need to give `mlnet` executable permissions and include `mlnet` to the system path. Instructions on how to do this should appear in the terminal when you install `mlnet` (or any global tool).

Alternatively, you can try using the following command to run the `mlnet` tool:

```
~/.dotnet/tools/mlnet
```

Install a specific release version

If you're trying to install a pre-release version or a specific version of the tool, you can specify the OS, processor architecture, and [framework](#) using the following format:

```
dotnet tool install -g mlnet-<OS>-<ARCH> --framework <FRAMEWORK>
```

You can also check if the package is properly installed by typing the following command:

```
dotnet tool list -g
```

Uninstall the CLI package

To uninstall the ML.NET CLI use the package ID you can get from running the `dotnet tool list --global` command. Then, use the `dotnet tool uninstall --global` command.

- [Windows](#)
- [Mac](#)
- [Linux](#)

```
dotnet tool uninstall --global mlnet-win-x64
```

ARM

```
dotnet tool uninstall --global mlnet-win-arm64
```

Update the CLI package

To update the ML.NET CLI use the package ID you can get from running the `dotnet tool list --global` command. Then, use the `dotnet tool update --global` command.

- [Windows](#)
- [Mac](#)
- [Linux](#)

x64

```
dotnet tool update --global mlnet-win-x64
```

ARM

```
dotnet tool update --global mlnet-win-arm64
```

Installation directory

The ML.NET CLI can be installed in the default directory or in a specific location. The default directories are:

OS	PATH
Linux/macOS	<code>\$HOME/.dotnet/tools</code>
Windows	<code>%USERPROFILE%\.dotnet\tools</code>

These locations are added to the user's path when the SDK is first run, so Global Tools installed there can be called directly.

Note: the Global Tools are user-specific, not machine global. Being user-specific means you cannot install a Global Tool that is available to all users of the machine. The tool is only available for each user profile where the tool was installed.

Global Tools can also be installed in a specific directory. When installed in a specific directory, the user must ensure the command is available, by including that directory in the path, by calling the command with the directory specified, or calling the tool from within the specified directory. In this case, the .NET CLI doesn't add this location automatically to the PATH environment variable.

See also

- [ML.NET CLI overview](#)
- [Tutorial: Analyze sentiment with the ML.NET CLI](#)
- [ML.NET CLI auto-train command reference guide](#)
- [Telemetry in ML.NET CLI](#)

How to use the ML.NET automated machine learning API

10/14/2022 • 4 minutes to read • [Edit Online](#)

Automated machine learning (AutoML) automates the process of applying machine learning to data. Given a dataset, you can run an AutoML **experiment** to iterate over different data featurizations, machine learning algorithms, and hyperparameters to select the best model.

NOTE

This topic refers to the automated machine learning API for ML.NET, which is currently in preview. Material may be subject to change.

Load data

Automated machine learning supports loading a dataset into an [IDataView](#). Data can be in the form of tab-separated value (TSV) files and comma separated value (CSV) files.

Example:

```
using Microsoft.ML;
using Microsoft.ML.AutoML;
// ...
MLContext mlContext = new MLContext();
IDataView trainDataView = mlContext.Data.LoadFromTextFile<SentimentIssue>("my-data-file.csv", hasHeader: true);
```

Select the machine learning task type

Before creating an experiment, determine the kind of machine learning problem you want to solve. Automated machine learning supports the following ML tasks:

- Binary Classification
- Multiclass Classification
- Regression
- Recommendation
- Ranking

Create experiment settings

Create experiment settings for the determined ML task type:

- Binary Classification

```
var experimentSettings = new BinaryExperimentSettings();
```

- Multiclass Classification

```
var experimentSettings = new MulticlassExperimentSettings();
```

- Regression

```
var experimentSettings = new RegressionExperimentSettings();
```

- Recommendation

```
var experimentSettings = new RecommendationExperimentSettings();
```

- Ranking

```
var experimentSettings = new RankingExperimentSettings();
```

Configure experiment settings

Experiments are highly configurable. See the [AutoML API docs](#) for a full list of configuration settings.

Some examples include:

1. Specify the maximum time that the experiment is allowed to run.

```
experimentSettings.MaxExperimentTimeInSeconds = 3600;
```

2. Use a cancellation token to cancel the experiment before it is scheduled to finish.

```
experimentSettings.CancellationToken = cts.Token;  
  
// Cancel experiment after the user presses any key  
CancelExperimentAfterAnyKeyPress(cts);
```

3. Specify a different optimizing metric.

```
var experimentSettings = new RegressionExperimentSettings();  
experimentSettings.OptimizingMetric = RegressionMetric.MeanSquaredError;
```

4. The `CacheDirectory` setting is a pointer to a directory where all models trained during the AutoML task will be saved. If `CacheDirectory` is set to null, models will be kept in memory instead of written to disk.

```
experimentSettings.CacheDirectory = null;
```

5. Instruct automated ML not to use certain trainers.

A default list of trainers to optimize are explored per task. This list can be modified for each experiment. For instance, trainers that run slowly on your dataset can be removed from the list. To optimize on one specific trainer call `experimentSettings.Trainers.Clear()`, then add the trainer that you want to use.

```
var experimentSettings = new RegressionExperimentSettings();  
experimentSettings.Trainers.Remove(RegressionTrainer.LbfgsPoissonRegression);  
experimentSettings.Trainers.Remove(RegressionTrainer.OnlineGradientDescent);
```

The list of supported trainers per ML task can be found at the corresponding link below:

- [Supported Binary Classification Algorithms](#)
- [Supported Multiclass Classification Algorithms](#)
- [Supported Regression Algorithms](#)
- [Supported Recommendation Algorithms](#)
- [Supported Ranking Algorithms](#)

Optimizing metric

The optimizing metric, as shown in the example above, determines the metric to be optimized during model training. The optimizing metric you can select is determined by the task type you choose. Below is a list of available metrics.

BINARY CLASSIFICATION	MULTICLASS CLASSIFICATION	REGRESSION & RECOMMENDATION	RANKING
Accuracy	LogLoss	RSquared	Discounted Cumulative Gains
AreaUnderPrecisionRecallCurve	LogLossReduction	MeanAbsoluteError	Normalized Discounted Cumulative Gains
AreaUnderRocCurve	MacroAccuracy	MeanSquaredError	
F1Score	MicroAccuracy	RootMeanSquaredError	
NegativePrecision	TopKAccuracy		
NegativeRecall			
PositivePrecision			
PositiveRecall			

Data pre-processing and featurization

NOTE

The feature column only supports types of [Boolean](#), [Single](#), and [String](#).

Data pre-processing happens by default and the following steps are performed automatically for you:

1. Drop features with no useful information

Drop features with no useful information from training and validation sets. These include features with all values missing, same value across all rows or with extremely high cardinality (e.g., hashes, IDs or GUIDs).

2. Missing value indication and imputation

Fill missing value cells with the default value for the datatype. Append indicator features with the same number of slots as the input column. The value in the appended indicator features is `1` if the value in the input column is missing and `0` otherwise.

3. Generate additional features

For text features: Bag-of-word features using unigrams and tri-character-grams.

For categorical features: One-hot encoding for low cardinality features, and one-hot-hash encoding for high cardinality categorical features.

4. Transformations and encodings

Text features with very few unique values transformed into categorical features. Depending on cardinality of categorical features, perform one-hot encoding or one-hot hash encoding.

Exit criteria

Define the criteria to complete your task:

1. Exit after a length of time - Using `MaxExperimentTimeInSeconds` in your experiment settings you can define how long in seconds that a task should continue to run.
2. Exit on a cancellation token - You can use a cancellation token that lets you cancel the task before it is scheduled to finish.

```
var cts = new CancellationTokenSource();
var experimentSettings = new RegressionExperimentSettings();
experimentSettings.MaxExperimentTimeInSeconds = 3600;
experimentSettings.CancellationToken = cts.Token;
```

Create an experiment

Once you have configured the experiment settings, you are ready to create the experiment.

```
RegressionExperiment experiment = mlContext.Auto().CreateRegressionExperiment(experimentSettings);
```

Run the experiment

Running the experiment triggers data pre-processing, learning algorithm selection, and hyperparameter tuning. AutoML will continue to generate combinations of featurization, learning algorithms, and hyperparameters until the `MaxExperimentTimeInSeconds` is reached or the experiment is terminated.

```
ExperimentResult<RegressionMetrics> experimentResult = experiment
    .Execute(trainingDataView, LabelColumnName, progressHandler: progressHandler);
```

Explore other overloads for `Execute()` if you want to pass in validation data, column information indicating the column purpose, or prefeaturizers.

Training modes

Training dataset

AutoML provides an overloaded experiment execute method which allows you to provide training data. Internally, automated ML divides the data into train-validate splits.

```
experiment.Execute(trainDataView);
```

Custom validation dataset

Use custom validation dataset if random split is not acceptable, as is usually the case with time series data. You can specify your own validation dataset. The model will be evaluated against the validation dataset specified instead of one or more random datasets.

```
experiment.Execute(train DataView, validation DataView);
```

Explore model metrics

After each iteration of an ML experiment, metrics relating to that task are stored.

For example, we can access validation metrics from the best run:

```
RegressionMetrics metrics = experimentResult.BestRun.ValidationMetrics;
Console.WriteLine($"R-Squared: {metrics.RSquared:0.##}");
Console.WriteLine($"Root Mean Squared Error: {metrics.RootMeanSquaredError:0.##}");
```

The following are all the available metrics per ML task:

- [Binary classification metrics](#)
- [Multiclass classification metrics](#)
- [Regression & recommendation metrics](#)
- [Ranking](#)

See also

For full code samples and more visit the [dotnet/machinelearning-samples](#) GitHub repository.

Install extra ML.NET dependencies

10/14/2022 • 2 minutes to read • [Edit Online](#)

In most cases, on all operating systems, installing ML.NET is as simple as referencing the appropriate NuGet package.

```
dotnet add package Microsoft.ML
```

In some cases though, there are additional installation requirements, particularly when native components are required. This document describes the installation requirements for those cases. The sections are broken down by the specific `Microsoft.ML.*` NuGet package that has the additional dependency.

Microsoft.ML.TimeSeries, Microsoft.ML.AutoML

Both of these packages have a dependency on `Microsoft.ML.MKL.Redist`, which has a dependency on `libomp`.

Windows

No extra installation steps required. The library is installed when the NuGet package is added to the project.

Linux

1. Install the GPG key for the repository

```
sudo bash
# <type your user password when prompted. this will put you in a root shell>
# cd to /tmp where this shell has write permission
cd /tmp
# now get the key:
wget https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS-2019.PUB
# now install that key
apt-key add GPG-PUB-KEY-INTEL-SW-PRODUCTS-2019.PUB
# now remove the public key file exit the root shell
rm GPG-PUB-KEY-INTEL-SW-PRODUCTS-2019.PUB
exit
```

2. Add the APT Repository for MKL

```
sudo sh -c 'echo deb https://apt.repos.intel.com/mkl all main > /etc/apt/sources.list.d/intel-mkl.list'
```

3. Update packages

```
sudo apt-get update
```

4. Install MKL

```
sudo apt-get install <COMPONENT>-<VERSION>.<UPDATE>-<BUILD_NUMBER>
```

For example:

```
sudo apt-get install intel-mkl-64bit-2020.0-088
```

Determine the location of `libiomp.so`

```
find /opt -name "libiomp5.so"
```

For example:

```
/opt/intel/compilers_and_libraries_2020.0.166/linux/compiler/lib/intel64_lin/libiomp5.so
```

5. Add this location to the load library path:

```
sudo ldconfig /opt/intel/compilers_and_libraries_2020.0.166/linux/compiler/lib/intel64_lin
```

Mac

1. Install the library with `Homebrew`

```
wget https://raw.githubusercontent.com/Homebrew/homebrew-core/fb8323f2b170bd4ae97e1bac9bf3e2983af3fdb0/Formula/libomp.rb && brew install ./libomp.rb && brew link libomp --force
```

Load data from files and other sources

10/14/2022 • 5 minutes to read • [Edit Online](#)

Learn how to load data for processing and training into ML.NET using the API. The data is originally stored in files or other data sources such as databases, JSON, XML or in-memory collections.

If you're using Model Builder, see [Load training data into Model Builder](#).

Create the data model

ML.NET enables you to define data models via classes. For example, given the following input data:

```
Size (Sq. ft.), HistoricalPrice1 ($), HistoricalPrice2 ($), HistoricalPrice3 ($), Current Price ($)
700, 100000, 300000, 250000, 500000
1000, 600000, 400000, 650000, 700000
```

Create a data model that represents the snippet below:

```
public class HousingData
{
    [LoadColumn(0)]
    public float Size { get; set; }

    [LoadColumn(1, 3)]
    [VectorType(3)]
    public float[] HistoricalPrices { get; set; }

    [LoadColumn(4)]
    [ColumnName("Label")]
    public float CurrentPrice { get; set; }
}
```

Annotating the data model with column attributes

Attributes give ML.NET more information about the data model and the data source.

The `LoadColumn` attribute specifies your properties' column indices.

IMPORTANT

`LoadColumn` is only required when loading data from a file.

Load columns as:

- Individual columns like `Size` and `CurrentPrices` in the `HousingData` class.
- Multiple columns at a time in the form of a vector like `HistoricalPrices` in the `HousingData` class.

If you have a vector property, apply the `VectorType` attribute to the property in your data model. It's important to note that all of the elements in the vector need to be the same type. Keeping the columns separated allows for ease and flexibility of feature engineering, but for a very large number of columns, operating on the individual columns causes an impact on training speed.

ML.NET Operates through column names. If you want to change the name of a column to something other than

the property name, use the `ColumnName` attribute. When creating in-memory objects, you still create objects using the property name. However, for data processing and building machine learning models, ML.NET overrides and references the property with the value provided in the `ColumnName` attribute.

Load data from a single file

To load data from a file use the `LoadFromTextFile` method along with the data model for the data to be loaded. Since `separatorChar` parameter is tab-delimited by default, change it for your data file as needed. If your file has a header, set the `hasHeader` parameter to `true` to ignore the first line in the file and begin to load data from the second line.

```
//Create MLContext
MLContext mlContext = new MLContext();

//Load Data
IDataView data = mlContext.Data.LoadFromTextFile<HousingData>("my-data-file.csv", separatorChar: ',',
hasHeader: true);
```

Load data from multiple files

In the event that your data is stored in multiple files, as long as the data schema is the same, ML.NET allows you to load data from multiple files that are either in the same directory or multiple directories.

Load from files in a single directory

When all of your data files are in the same directory, use wildcards in the `LoadFromTextFile` method.

```
//Create MLContext
MLContext mlContext = new MLContext();

//Load Data File
IDataView data = mlContext.Data.LoadFromTextFile<HousingData>("Data/*", separatorChar: ',', hasHeader:
true);
```

Load from files in multiple directories

To load data from multiple directories, use the `CreateTextLoader` method to create a `TextLoader`. Then, use the `TextLoader.Load` method and specify the individual file paths (wildcards can't be used).

```
//Create MLContext
MLContext mlContext = new MLContext();

// Create TextLoader
TextLoader textLoader = mlContext.Data.CreateTextLoader<HousingData>(separatorChar: ',', hasHeader: true);

// Load Data
IDataView data = textLoader.Load("DataFolder/SubFolder1/1.txt", "DataFolder/SubFolder2/1.txt");
```

Load data from a relational database

ML.NET supports loading data from a variety of relational databases supported by `System.Data` that include SQL Server, Azure SQL Database, Oracle, SQLite, PostgreSQL, Progress, IBM DB2, and many more.

NOTE

To use `DatabaseLoader`, reference the [System.Data.SqlClient](#) NuGet package.

Given a database with a table named `House` and the following schema:

```
CREATE TABLE [House] (
    [HouseId] INT NOT NULL IDENTITY,
    [Size] INT NOT NULL,
    [NumBed] INT NOT NULL,
    [Price] REAL NOT NULL
    CONSTRAINT [PK_House] PRIMARY KEY ([HouseId])
);
```

The data can be modeled by a class like `HouseData`.

```
public class HouseData
{
    public float Size { get; set; }

    public float NumBed { get; set; }

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

Then, inside of your application, create a `DatabaseLoader`.

```
MLContext m1Context = new MLContext();

DatabaseLoader loader = m1Context.Data.CreateDatabaseLoader<HouseData>();
```

Define your connection string as well as the SQL command to be executed on the database and create a `DataSource` instance. This sample uses a LocalDB SQL Server database with a file path. However, `DatabaseLoader` supports any other valid connection string for databases on-premises and in the cloud.

```
string connectionString = @"Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=<YOUR-DB-FILEPATH>;Database=<YOUR-DB-NAME>;Integrated Security=True;Connect Timeout=30";

string sqlCommand = "SELECT CAST(Size as REAL) as Size, CAST(NumBed as REAL) as NumBed, Price FROM House";

DataSource dbSource = new DataSource(SqlClientFactory.Instance, connectionString, sqlCommand);
```

Numerical data that is not of type `Real` has to be converted to `Real`. The `Real` type is represented as a single-precision floating-point value or `Single`, the input type expected by ML.NET algorithms. In this sample, the `Size` and `NumBed` columns are integers in the database. Using the `CAST` built-in function, it's converted to `Real`. Because the `Price` property is already of type `Real` it is loaded as is.

Use the `Load` method to load the data into an `IDataView`.

```
IDataView data = loader.Load(dbSource);
```

Load data from other sources

In addition to loading data stored in files, ML.NET supports loading data from sources that include but are not

limited to:

- In-memory collections
- JSON/XML

Note that when working with streaming sources, ML.NET expects input to be in the form of an in-memory collection. Therefore, when working with sources like JSON/XML, make sure to format the data into an in-memory collection.

Given the following in-memory collection:

```
HousingData[] inMemoryCollection = new HousingData[]
{
    new HousingData
    {
        Size =700f,
        HistoricalPrices = new float[]
        {
            100000f, 300000f, 250000f
        },
        CurrentPrice = 500000f
    },
    new HousingData
    {
        Size =1000f,
        HistoricalPrices = new float[]
        {
            600000f, 400000f, 650000f
        },
        CurrentPrice=700000f
    }
};
```

Load the in-memory collection into an [IDataView](#) with the [LoadFromEnumerable](#) method:

IMPORTANT

[LoadFromEnumerable](#) assumes that the [IEnumerable](#) it loads from is thread-safe.

```
// Create MLContext
MLContext mlContext = new MLContext();

//Load Data
IDataView data = mlContext.Data.LoadFromEnumerable<HousingData>(inMemoryCollection);
```

Next steps

- To clean or otherwise process data, see [Prepare data for building a model](#).
- When you're ready to build a model, see [Train and evaluate a model](#).

Prepare data for building a model

10/14/2022 • 8 minutes to read • [Edit Online](#)

Learn how to use ML.NET to prepare data for additional processing or building a model.

Data is often unclean and sparse. ML.NET machine learning algorithms expect input or features to be in a single numerical vector. Similarly, the value to predict (label), especially when it's categorical data, has to be encoded. Therefore one of the goals of data preparation is to get the data into the format expected by ML.NET algorithms.

Filter data

Sometimes, not all data in a dataset is relevant for analysis. An approach to remove irrelevant data is filtering.

The `DataOperationsCatalog` contains a set of filter operations that take in an `IDataView` containing all of the data and return an `IDataView` containing only the data points of interest. It's important to note that because filter operations are not an `IEstimator` or `ITransformer` like those in the `TransformsCatalog`, they cannot be included as part of an `EstimatorChain` or `TransformerChain` data preparation pipeline.

Take the following input data and load it into an `IDataView` called `data`:

```
HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        NumberOfBedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        NumberOfBedrooms=6f,
        Price=600000f
    }
};
```

To filter data based on the value of a column, use the `FilterRowsByColumn` method.

```
// Apply filter
IDataView filteredData = mlContext.Data.FilterRowsByColumn(data, "Price", lowerBound: 200000, upperBound: 1000000);
```

The sample above takes rows in the dataset with a price between 200000 and 1000000. The result of applying this filter would return only the last two rows in the data and exclude the first row because its price is 100000 and not between the specified range.

Replace missing values

Missing values are a common occurrence in datasets. One approach to dealing with missing values is to replace them with the default value for the given type if any or another meaningful value such as the mean value in the data.

Take the following input data and load it into an `IDataView` called `data`:

```
HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        NumberOfBedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        NumberOfBedrooms=6f,
        Price=float.NaN
    }
};
```

Notice that the last element in our list has a missing value for `Price`. To replace the missing values in the `Price` column, use the `ReplaceMissingValues` method to fill in that missing value.

IMPORTANT

`ReplaceMissingValue` only works with numerical data.

```
// Define replacement estimator
var replacementEstimator = mlContext.Transforms.ReplaceMissingValues("Price", replacementMode:
MissingValueReplacingEstimator.ReplacementMode.Mean);

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimator
ITransformer replacementTransformer = replacementEstimator.Fit(data);

// Transform data
IDataView transformedData = replacementTransformer.Transform(data);
```

ML.NET supports various [replacement modes](#). The sample above uses the `Mean` replacement mode, which fills in the missing value with that column's average value. The replacement's result fills in the `Price` property for the last element in our data with 200,000 since it's the average of 100,000 and 300,000.

Use normalizers

[Normalization](#) is a data pre-processing technique used to scale features to be in the same range, usually between 0 and 1, so that they can be more accurately processed by a machine learning algorithm. For example, the ranges for age and income vary significantly with age generally being in the range of 0-100 and income generally being in the range of zero to thousands. Visit the [transforms page](#) for a more detailed list and description of normalization transforms.

Min-Max normalization

Take the following input data and load it into an `IDataView` called `data`:

```

HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms = 2f,
        Price = 200000f
    },
    new HomeData
    {
        NumberOfBedrooms = 1f,
        Price = 100000f
    }
};

```

Normalization can be applied to columns with single numerical values as well as vectors. Normalize the data in the `Price` column using min-max normalization with the [NormalizeMinMax](#) method.

```

// Define min-max estimator
var minMaxEstimator = mlContext.Transforms.NormalizeMinMax("Price");

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimator
ITransformer minMaxTransformer = minMaxEstimator.Fit(data);

// Transform data
IDataView transformedData = minMaxTransformer.Transform(data);

```

The original price values `[200000,100000]` are converted to `[1, 0.5]` using the [MinMax](#) normalization formula that generates output values in the range of 0-1.

Binning

[Binning](#) converts continuous values into a discrete representation of the input. For example, suppose one of your features is age. Instead of using the actual age value, binning creates ranges for that value. 0-18 could be one bin, another could be 19-35 and so on.

Take the following input data and load it into an [IDataView](#) called `data`:

```

HomeData[] homeDataList = new HomeData[]
{
    new HomeData
    {
        NumberOfBedrooms=1f,
        Price=100000f
    },
    new HomeData
    {
        NumberOfBedrooms=2f,
        Price=300000f
    },
    new HomeData
    {
        NumberOfBedrooms=6f,
        Price=600000f
    }
};

```

Normalize the data into bins using the [NormalizeBinning](#) method. The `maximumBinCount` parameter enables you to specify the number of bins needed to classify your data. In this example, data will be put into two bins.

```

// Define binning estimator
var binningEstimator = mlContext.Transforms.NormalizeBinning("Price", maximumBinCount: 2);

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimator
var binningTransformer = binningEstimator.Fit(data);

// Transform Data
IDataView transformedData = binningTransformer.Transform(data);

```

The result of binning creates bin bounds of `[0,200000,Infinity]`. Therefore the resulting bins are `[0,1,1]` because the first observation is between 0-200000 and the others are greater than 200000 but less than infinity.

Work with categorical data

One of the most common types of data is categorical data. Categorical data has a finite number of categories. For example, the states of the USA, or a list of the types of animals found in a set of pictures. Whether the categorical data are features or labels, they must be mapped onto a numerical value so they can be used to generate a machine learning model. There are a number of ways of working with categorical data in ML.NET, depending on the problem you are solving.

Key value mapping

In ML.NET, a key is an integer value that represents a category. Key value mapping is most often used to map string labels into unique integer values for training, then back to their string values when the model is used to make a prediction.

The transforms used to perform key value mapping are [MapValueToKey](#) and [MapKeyToValue](#).

[MapValueToKey](#) adds a dictionary of mappings in the model, so that [MapKeyToValue](#) can perform the reverse transform when making a prediction.

One hot encoding

One hot encoding takes a finite set of values and maps them onto integers whose binary representation has a single `1` value in unique positions in the string. One hot encoding can be the best choice if there is no implicit ordering of the categorical data. The following table shows an example with zip codes as raw values.

RAW VALUE	ONE HOT ENCODED VALUE
98052	00...01
98100	00...10
...	...
98109	10...00

The transform to convert categorical data to one-hot encoded numbers is [OneHotEncoding](#).

Hashing

Hashing is another way to convert categorical data to numbers. A hash function maps data of an arbitrary size (a string of text for example) onto a number with a fixed range. Hashing can be a fast and space-efficient way of vectorizing features. One notable example of hashing in machine learning is email spam filtering where, instead of maintaining a dictionary of known words, every word in the email is hashed and added to a large feature vector. Using hashing in this way avoids the problem of malicious spam filtering circumvention by the use of words that are not in the dictionary.

ML.NET provides [Hash](#) transform to perform hashing on text, dates, and numerical data. Like value key mapping, the outputs of the hash transform are key types.

Work with text data

Like categorical data, text data needs to be transformed into numerical features before using it to build a machine learning model. Visit the [transforms page](#) for a more detailed list and description of text transforms.

Using data like the data below that has been loaded into an [IDataView](#):

```
ReviewData[] reviews = new ReviewData[]
{
    new ReviewData
    {
        Description="This is a good product",
        Rating=4.7f
    },
    new ReviewData
    {
        Description="This is a bad product",
        Rating=2.3f
    }
};
```

ML.NET provides the [FeaturizeText](#) transform that takes a text's string value and creates a set of features from the text, by applying a series of individual transforms.

```
// Define text transform estimator
var textEstimator = mlContext.Transforms.Text.FeaturizeText("Description");

// Fit data to estimator
// Fitting generates a transformer that applies the operations of defined by estimator
ITransformer textTransformer = textEstimator.Fit(data);

// Transform data
IDataView transformedData = textTransformer.Transform(data);
```

The resulting transform converts the text values in the `Description` column to a numerical vector that looks similar to the output below:

```
[ 0.2041241, 0.2041241, 0.2041241, 0.4082483, 0.2041241, 0.2041241, 0.2041241, 0.2041241,
 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.2041241,
 0.2041241, 0.2041241, 0.2041241, 0.2041241, 0.4472136, 0.4472136, 0.4472136, 0.4472136, 0 ]
```

The transforms that make up [FeaturizeText](#) can also be applied individually for finer grain control over feature generation.

```
// Define text transform estimator
var textEstimator = mlContext.Transforms.Text.NormalizeText("Description")
    .Append(mlContext.Transforms.Text.TokenizeIntoWords("Description"))
    .Append(mlContext.Transforms.Text.RemoveDefaultStopWords("Description"))
    .Append(mlContext.Transforms.Conversion.MapValueToKey("Description"))
    .Append(mlContext.Transforms.Text.ProduceNgrams("Description"))
    .Append(mlContext.Transforms.NormalizeLpNorm("Description"));
```

`textEstimator` contains a subset of operations performed by the [FeaturizeText](#) method. The benefit of a more complex pipeline is control and visibility over the transformations applied to the data.

Using the first entry as an example, the following is a detailed description of the results produced by the transformation steps defined by `textEstimator`:

Original Text: This is a good product

TRANSFORM	DESCRIPTION	RESULT
1. NormalizeText	Converts all letters to lowercase by default	this is a good product
2. TokenizeWords	Splits string into individual words	["this","is","a","good","product"]
3. RemoveDefaultStopWords	Removes stopwords like <i>is</i> and <i>a</i> .	["good","product"]
4. MapValueToKey	Maps the values to keys (categories) based on the input data	[1,2]
5. ProduceNGrams	Transforms text into sequence of consecutive words	[1,1,1,0,0]
6. NormalizeLpNorm	Scale inputs by their lp-norm	[0.577350529, 0.577350529, 0.577350529, 0, 0]

Train and evaluate a model

10/14/2022 • 6 minutes to read • [Edit Online](#)

Learn how to build machine learning models, collect metrics, and measure performance with ML.NET. Although this sample trains a regression model, the concepts are applicable throughout a majority of the other algorithms.

Split data for training and testing

The goal of a machine learning model is to identify patterns within training data. These patterns are used to make predictions using new data.

The data can be modeled by a class like `HousingData`.

```
public class HousingData
{
    [LoadColumn(0)]
    public float Size { get; set; }

    [LoadColumn(1, 3)]
    [VectorType(3)]
    public float[] HistoricalPrices { get; set; }

    [LoadColumn(4)]
    [ColumnName("Label")]
    public float CurrentPrice { get; set; }
}
```

Given the following data which is loaded into an `IDataView`.

```

HousingData[] housingData = new HousingData[]
{
    new HousingData
    {
        Size = 600f,
        HistoricalPrices = new float[] { 100000f, 125000f, 122000f },
        CurrentPrice = 170000f
    },
    new HousingData
    {
        Size = 1000f,
        HistoricalPrices = new float[] { 200000f, 250000f, 230000f },
        CurrentPrice = 225000f
    },
    new HousingData
    {
        Size = 1000f,
        HistoricalPrices = new float[] { 126000f, 130000f, 200000f },
        CurrentPrice = 195000f
    },
    new HousingData
    {
        Size = 850f,
        HistoricalPrices = new float[] { 150000f, 175000f, 210000f },
        CurrentPrice = 205000f
    },
    new HousingData
    {
        Size = 900f,
        HistoricalPrices = new float[] { 155000f, 190000f, 220000f },
        CurrentPrice = 210000f
    },
    new HousingData
    {
        Size = 550f,
        HistoricalPrices = new float[] { 99000f, 98000f, 130000f },
        CurrentPrice = 180000f
    }
};

```

Use the `TrainTestSplit` method to split the data into train and test sets. The result will be a `TrainTestData` object which contains two `IDataView` members, one for the train set and the other for the test set. The data split percentage is determined by the `testFraction` parameter. The snippet below is holding out 20 percent of the original data for the test set.

```

DataOperationsCatalog.TrainTestData dataSplit = mlContext.Data.TrainTestSplit(data, testFraction: 0.2);
IDataView trainData = dataSplit.TrainSet;
IDataView testData = dataSplit.TestSet;

```

Prepare the data

The data needs to be pre-processed before training a machine learning model. More information on data preparation can be found on the [data prep how-to article](#) as well as the [transforms page](#).

ML.NET algorithms have constraints on input column types. Additionally, default values are used for input and output column names when no values are specified.

Working with expected column types

The machine learning algorithms in ML.NET expect a float vector of known size as input. Apply the `VectorType` attribute to your data model when all of the data is already in numerical format and is intended to be processed

together (i.e. image pixels).

If data is not all numerical and you want to apply different data transformations on each of the columns individually, use the `Concatenate` method after all of the columns have been processed to combine all of the individual columns into a single feature vector that is output to a new column.

The following snippet combines the `Size` and `HistoricalPrices` columns into a single feature vector that is output to a new column called `Features`. Because there is a difference in scales, `NormalizeMinMax` is applied to the `Features` column to normalize the data.

```
// Define Data Prep Estimator
// 1. Concatenate Size and Historical into a single feature vector output to a new column called Features
// 2. Normalize Features vector
IEstimator<ITransformer> dataPrepEstimator =
    mlContext.Transforms.Concatenate("Features", "Size", "HistoricalPrices")
    .Append(mlContext.Transforms.NormalizeMinMax("Features"));

// Create data prep transformer
ITransformer dataPrepTransformer = dataPrepEstimator.Fit(trainData);

// Apply transforms to training data
IDataView transformedTrainingData = dataPrepTransformer.Transform(trainData);
```

Working with default column names

ML.NET algorithms use default column names when none are specified. All trainers have a parameter called `featureColumnName` for the inputs of the algorithm and when applicable they also have a parameter for the expected value called `labelColumnName`. By default those values are `Features` and `Label` respectively.

By using the `Concatenate` method during pre-processing to create a new column called `Features`, there is no need to specify the feature column name in the parameters of the algorithm since it already exists in the pre-processed `IDataView`. The label column is `CurrentPrice`, but since the `ColumnName` attribute is used in the data model, ML.NET renames the `CurrentPrice` column to `Label` which removes the need to provide the `labelColumnName` parameter to the machine learning algorithm estimator.

If you don't want to use the default column names, pass in the names of the feature and label columns as parameters when defining the machine learning algorithm estimator as demonstrated by the subsequent snippet:

```
var UserDefinedColumnSdcaEstimator = mlContext.Regression.Trainers.Sdca(labelColumnName:
    "MyLabelColumnName", featureColumnName: "MyFeatureColumnName");
```

Caching data

By default, when data is processed, it is lazily loaded or streamed which means that trainers may load the data from disk and iterate over it multiple times during training. Therefore, caching is recommended for datasets that fit into memory to reduce the number of times data is loaded from disk. Caching is done as part of an `EstimatorChain` by using `AppendCacheCheckpoint`.

It's recommended to use `AppendCacheCheckpoint` before any trainers in the pipeline.

Using the following `EstimatorChain`, adding `AppendCacheCheckpoint` before the `StochasticDualCoordinateAscent` trainer caches the results of the previous estimators for later use by the trainer.

```
// 1. Concatenate Size and Historical into a single feature vector output to a new column called Features  
// 2. Normalize Features vector  
// 3. Cache prepared data  
// 4. Use Sdca trainer to train the model  
IEstimator<ITransformer> dataPrepEstimator =  
    mlContext.Transforms.Concatenate("Features", "Size", "HistoricalPrices")  
        .Append(mlContext.Transforms.NormalizeMinMax("Features"))  
        .AppendCacheCheckpoint(mlContext);  
    .Append(mlContext.Regression.Trainers.Sdca());
```

Train the machine learning model

Once the data is pre-processed, use the `Fit` method to train the machine learning model with the `StochasticDualCoordinateAscent` regression algorithm.

```
// Define StochasticDualCoordinateAscent regression algorithm estimator  
var sdcaEstimator = mlContext.Regression.Trainers.Sdca();  
  
// Build machine learning model  
var trainedModel = sdcaEstimator.Fit(transformedTrainingData);
```

Extract model parameters

After the model has been trained, extract the learned `ModelParameters` for inspection or retraining. The `LinearRegressionModelParameters` provide the bias and learned coefficients or weights of the trained model.

```
var trainedModelParameters = trainedModel.Model as LinearRegressionModelParameters;
```

NOTE

Other models have parameters that are specific to their tasks. For example, the `K-Means algorithm` puts data into cluster based on centroids and the `KMeansModelParameters` contains a property that stores these learned centroids. To learn more, visit the `Microsoft.ML.Trainers` API Documentation and look for classes that contain `ModelParameters` in their name.

Evaluate model quality

To help choose the best performing model, it is essential to evaluate its performance on test data. Use the `Evaluate` method, to measure various metrics for the trained model.

NOTE

The `Evaluate` method produces different metrics depending on which machine learning task was performed. For more details, visit the `Microsoft.ML.Data` API Documentation and look for classes that contain `Metrics` in their name.

```
// Measure trained model performance
// Apply data prep transformer to test data
IDataView transformedTestData = dataPrepTransformer.Transform(testData);

// Use trained model to make inferences on test data
IDataView testDataPredictions = trainedModel.Transform(transformedTestData);

// Extract model metrics and get RSquared
RegressionMetrics trainedModelMetrics = mlContext.Regression.Evaluate(testDataPredictions);
double rSquared = trainedModelMetrics.RSquared;
```

In the previous code sample:

1. Test data set is pre-processed using the data preparation transforms previously defined.
2. The trained machine learning model is used to make predictions on the test data.
3. In the `Evaluate` method, the values in the `CurrentPrice` column of the test data set are compared against the `Score` column of the newly output predictions to calculate the metrics for the regression model, one of which, R-Squared is stored in the `rSquared` variable.

NOTE

In this small example, the R-Squared is a number not in the range of 0-1 because of the limited size of the data. In a real-world scenario, you should expect to see a value between 0 and 1.

Train a machine learning model using cross validation

10/14/2022 • 3 minutes to read • [Edit Online](#)

Learn how to use cross validation to train more robust machine learning models in ML.NET.

Cross-validation is a training and model evaluation technique that splits the data into several partitions and trains multiple algorithms on these partitions. This technique improves the robustness of the model by holding out data from the training process. In addition to improving performance on unseen observations, in data-constrained environments it can be an effective tool for training models with a smaller dataset.

The data and data model

Given data from a file that has the following format:

```
Size (Sq. ft.), HistoricalPrice1 ($), HistoricalPrice2 ($), HistoricalPrice3 ($), Current Price ($)
620.00, 148330.32, 140913.81, 136686.39, 146105.37
550.00, 557033.46, 529181.78, 513306.33, 548677.95
1127.00, 479320.99, 455354.94, 441694.30, 472131.18
1120.00, 47504.98, 45129.73, 43775.84, 46792.41
```

The data can be modeled by a class like `HousingData` and loaded into an `IDataView`.

```
public class HousingData
{
    [LoadColumn(0)]
    public float Size { get; set; }

    [LoadColumn(1, 3)]
    [VectorType(3)]
    public float[] HistoricalPrices { get; set; }

    [LoadColumn(4)]
    [ColumnName("Label")]
    public float CurrentPrice { get; set; }
}
```

Prepare the data

Pre-process the data before using it to build the machine learning model. In this sample, the `Size` and `HistoricalPrices` columns are combined into a single feature vector, which is output to a new column called `Features` using the `Concatenate` method. In addition to getting the data into the format expected by ML.NET algorithms, concatenating columns optimizes subsequent operations in the pipeline by applying the operation once for the concatenated column instead of each of the separate columns.

Once the columns are combined into a single vector, `NormalizeMinMax` is applied to the `Features` column to get `Size` and `HistoricalPrices` in the same range between 0-1.

```

// Define data prep estimator
IEstimator<ITransformer> dataPrepEstimator =
    mlContext.Transforms.Concatenate("Features", new string[] { "Size", "HistoricalPrices" })
        .Append(mlContext.Transforms.NormalizeMinMax("Features"));

// Create data prep transformer
ITransformer dataPrepTransformer = dataPrepEstimator.Fit(data);

// Transform data
IDataView transformedData = dataPrepTransformer.Transform(data);

```

Train model with cross validation

Once the data has been pre-processed, it's time to train the model. First, select the algorithm that most closely aligns with the machine learning task to be performed. Because the predicted value is a numerically continuous value, the task is regression. One of the regression algorithms implemented by ML.NET is the [StochasticDualCoordinateAscentCoordinator](#) algorithm. To train the model with cross-validation use the [CrossValidate](#) method.

NOTE

Although this sample uses a linear regression model, CrossValidate is applicable to all other machine learning tasks in ML.NET except Anomaly Detection.

```

// Define StochasticDualCoordinateAscent algorithm estimator
IEstimator<ITransformer> sdcaEstimator = mlContext.Regression.Trainers.Sdca();

// Apply 5-fold cross validation
var cvResults = mlContext.Regression.CrossValidate(transformedData, sdcaEstimator, numberOffolds: 5);

```

[CrossValidate](#) performs the following operations:

1. Partitions the data into a number of partitions equal to the value specified in the [numberOfFolds](#) parameter. The result of each partition is a [TrainTestData](#) object.
2. A model is trained on each of the partitions using the specified machine learning algorithm estimator on the training data set.
3. Each model's performance is evaluated using the [Evaluate](#) method on the test data set.
4. The model along with its metrics are returned for each of the models.

The result stored in [cvResults](#) is a collection of [CrossValidationResult](#) objects. This object includes the trained model as well as metrics which are both accessible from the [Model](#) and [Metrics](#) properties respectively. In this sample, the [Model](#) property is of type [ITransformer](#) and the [Metrics](#) property is of type [RegressionMetrics](#).

Evaluate the model

Metrics for the different trained models can be accessed through the [Metrics](#) property of the individual [CrossValidationResult](#) object. In this case, the **R-Squared metric** is accessed and stored in the variable [rSquared](#).

```

IEnumerable<double> rSquared =
    cvResults
        .Select(fold => fold.Metrics.RSquared);

```

If you inspect the contents of the `rSquared` variable, the output should be five values ranging from 0-1 where closer to 1 means best. Using metrics like R-Squared, select the models from best to worst performing. Then, select the top model to make predictions or perform additional operations with.

```
// Select all models
ITransformer[] models =
    cvResults
    .OrderByDescending(fold => fold.Metrics.RSquared)
    .Select(fold => fold.Model)
    .ToArray();

// Get Top Model
ITransformer topModel = models[0];
```

Inspect intermediate data during processing

10/14/2022 • 3 minutes to read • [Edit Online](#)

Learn how to inspect intermediate data during loading, processing, and model training steps in ML.NET.

Intermediate data is the output of each stage in the machine learning pipeline.

Intermediate data like the one represented below which is loaded into an `IDataView` can be inspected in various ways in ML.NET.

```
HousingData[] housingData = new HousingData[]
{
    new HousingData
    {
        Size = 600f,
        HistoricalPrices = new float[] { 100000f, 125000f, 122000f },
        CurrentPrice = 170000f
    },
    new HousingData
    {
        Size = 1000f,
        HistoricalPrices = new float[] { 200000f, 250000f, 230000f },
        CurrentPrice = 225000f
    },
    new HousingData
    {
        Size = 1000f,
        HistoricalPrices = new float[] { 126000f, 130000f, 200000f },
        CurrentPrice = 195000f
    },
    new HousingData
    {
        Size = 850f,
        HistoricalPrices = new float[] { 150000f, 175000f, 210000f },
        CurrentPrice = 205000f
    },
    new HousingData
    {
        Size = 900f,
        HistoricalPrices = new float[] { 155000f, 190000f, 220000f },
        CurrentPrice = 210000f
    },
    new HousingData
    {
        Size = 550f,
        HistoricalPrices = new float[] { 99000f, 98000f, 130000f },
        CurrentPrice = 180000f
    }
};
```

Convert `IDataView` to `IEnumerable`

One of the quickest ways to inspect an `IDataView` is to convert it to an `IEnumerable`. To convert an `IDataView` to `IEnumerable` use the `CreateEnumerable` method.

To optimize performance, set `reuseRowObject` to `true`. Doing so will lazily populate the same object with the data of the current row as it's being evaluated as opposed to creating a new object for each row in the dataset.

```

// Create an IEnumerable of HousingData objects from IDataView
IEnumerable<HousingData> housingDataEnumerable =
    mlContext.Data.CreateEnumerable<HousingData>(data, reuseRowObject: true);

// Iterate over each row
foreach (HousingData row in housingDataEnumerable)
{
    // Do something (print out Size property) with current Housing Data object being evaluated
    Console.WriteLine(row.Size);
}

```

Accessing specific indices with IEnumerable

If you only need access to a portion of the data or specific indices, use `CreateEnumerable` and set the `reuseRowObject` parameter value to `false` so a new object is created for each of the requested rows in the dataset. Then, convert the `IEnumerable` to an array or list.

WARNING

Converting the result of `CreateEnumerable` to an array or list will load all the requested `IDataView` rows into memory which may affect performance.

Once the collection has been created, you can perform operations on the data. The code snippet below takes the first three rows in the dataset and calculates the average current price.

```

// Create an Array of HousingData objects from IDataView
HousingData[] housingdataArray =
    mlContext.Data.CreateEnumerable<HousingData>(data, reuseRowObject: false)
        .Take(3)
        .ToArray();

// Calculate Average CurrentPrice of First Three Elements
HousingData firstRow = housingdataArray[0];
HousingData secondRow = housingdataArray[1];
HousingData thirdRow = housingdataArray[2];
float averageCurrentPrice = (firstRow.CurrentPrice + secondRow.CurrentPrice + thirdRow.CurrentPrice) / 3;

```

Inspect values in a single column

At any point in the model building process, values in a single column of an `IDataView` can be accessed using the `GetColumn` method. The `GetColumn` method returns all of the values in a single column as an `IEnumerable`.

```
IEnumerable<float> sizeColumn = data.GetColumn<float>("Size").ToList();
```

Inspect IDataView values one row at a time

`IDataView` is lazily evaluated. To iterate over the rows of an `IDataView` without converting to an `IEnumerable` as demonstrated in previous sections of this document, create a `DataViewRowCursor` by using the `GetRowCursor` method and passing in the `DataViewSchema` of your `IDataView` as a parameter. Then, to iterate over rows, use the `MoveNext` cursor method along with `ValueGetter` delegates to extract the respective values from each of the columns.

IMPORTANT

For performance purposes, vectors in ML.NET use `VBuffer` instead of native collection types (that is, `Vector`, `float[]`).

```
// Get DataViewSchema of IDataView
DataViewSchema columns = data.Schema;

// Create DataViewCursor
using (DataViewRowCursor cursor = data.GetRowCursor(columns))
{
    // Define variables where extracted values will be stored to
    float size = default;
    VBuffer<float> historicalPrices = default;
    float currentPrice = default;

    // Define delegates for extracting values from columns
    ValueGetter<float> sizeDelegate = cursor.GetGetter<float>(columns[0]);
    ValueGetter<VBuffer<float>> historicalPriceDelegate = cursor.GetGetter<VBuffer<float>>(columns[1]);
    ValueGetter<float> currentPriceDelegate = cursor.GetGetter<float>(columns[2]);

    // Iterate over each row
    while (cursor.MoveNext())
    {
        //Get values from respective columns
        sizeDelegate.Invoke(ref size);
        historicalPriceDelegate.Invoke(ref historicalPrices);
        currentPriceDelegate.Invoke(ref currentPrice);
    }
}
```

Preview result of pre-processing or training on a subset of the data

WARNING

Do not use `Preview` in production code because it is intended for debugging and may reduce performance.

The model building process is experimental and iterative. To preview what data would look like after pre-processing or training a machine learning model on a subset of the data, use the `Preview` method which returns a `DataDebuggerPreview`. The result is an object with `ColumnView` and `RowView` properties which are both an `IEnumerable` and contain the values in a particular column or row. Specify the number of rows to apply the transformation to with the `maxRows` parameter.

▲ ⚒ dataPreview	{3 columns, 5 rows}
▶ ⚒ ColumnView	Length = 3
▶ ⚒ RowView	Length = 5
▶ ⚒ Schema	{3 columns}

The result of inspecting an `IDataView` would look similar to the following:

▲ ⚒ RowView	Length = 5
▲ ⚒ [0]	{3 columns}
▶ ⚒ [0]	{[Size, 600]}
▶ ⚒ [1]	{[HistoricalPrices, Dense vector of size 3]}
▶ ⚒ [2]	{[Label, 170000]}

Interpret model predictions using Permutation Feature Importance

10/14/2022 • 4 minutes to read • [Edit Online](#)

Using Permutation Feature Importance (PFI), learn how to interpret ML.NET machine learning model predictions. PFI gives the relative contribution each feature makes to a prediction.

Machine learning models are often thought of as opaque boxes that take inputs and generate an output. The intermediate steps or interactions among the features that influence the output are rarely understood. As machine learning is introduced into more aspects of everyday life such as healthcare, it's of utmost importance to understand why a machine learning model makes the decisions it does. For example, if diagnoses are made by a machine learning model, healthcare professionals need a way to look into the factors that went into making that diagnosis. Providing the right diagnosis could make a great difference on whether a patient has a speedy recovery or not. Therefore the higher the level of explainability in a model, the greater confidence healthcare professionals have to accept or reject the decisions made by the model.

Various techniques are used to explain models, one of which is PFI. PFI is a technique used to explain classification and regression models that is inspired by [Breiman's Random Forests paper](#) (see section 10). At a high level, the way it works is by randomly shuffling data one feature at a time for the entire dataset and calculating how much the performance metric of interest decreases. The larger the change, the more important that feature is.

Additionally, by highlighting the most important features, model builders can focus on using a subset of more meaningful features which can potentially reduce noise and training time.

Load the data

The features in the dataset being used for this sample are in columns 1-12. The goal is to predict `Price`.

COLUMN	FEATURE	DESCRIPTION
1	CrimeRate	Per capita crime rate
2	ResidentialZones	Residential zones in town
3	CommercialZones	Non-residential zones in town
4	NearWater	Proximity to body of water
5	ToxicWasteLevels	Toxicity levels (PPM)
6	AverageRoomNumber	Average number of rooms in house
7	HomeAge	Age of home
8	BusinessCenterDistance	Distance to nearest business district
9	HighwayAccess	Proximity to highways

COLUMN	FEATURE	DESCRIPTION
10	TaxRate	Property tax rate
11	StudentTeacherRatio	Ratio of students to teachers
12	PercentPopulationBelowPoverty	Percent of population living below poverty
13	Price	Price of the home

A sample of the dataset is shown below:

```
1,24,13,1,0.59,3,96,11,23,608,14,13,32
4,80,18,1,0.37,5,14,7,4,346,19,13,41
2,98,16,1,0.25,10,5,1,8,689,13,36,12
```

The data in this sample can be modeled by a class like `HousingPriceData` and loaded into an `IDataView`.

```
class HousingPriceData
{
    [LoadColumn(0)]
    public float CrimeRate { get; set; }

    [LoadColumn(1)]
    public float ResidentialZones { get; set; }

    [LoadColumn(2)]
    public float CommercialZones { get; set; }

    [LoadColumn(3)]
    public float NearWater { get; set; }

    [LoadColumn(4)]
    public float ToxicWasteLevels { get; set; }

    [LoadColumn(5)]
    public float AverageRoomNumber { get; set; }

    [LoadColumn(6)]
    public float HomeAge { get; set; }

    [LoadColumn(7)]
    public float BusinessCenterDistance { get; set; }

    [LoadColumn(8)]
    public float HighwayAccess { get; set; }

    [LoadColumn(9)]
    public float TaxRate { get; set; }

    [LoadColumn(10)]
    public float StudentTeacherRatio { get; set; }

    [LoadColumn(11)]
    public float PercentPopulationBelowPoverty { get; set; }

    [LoadColumn(12)]
    [ColumnName("Label")]
    public float Price { get; set; }
}
```

Train the model

The code sample below illustrates the process of training a linear regression model to predict house prices.

```
// 1. Get the column name of input features.  
string[] featureColumnNames =  
    data.Schema  
    .Select(column => column.Name)  
    .Where(columnName => columnName != "Label").ToArray();  
  
// 2. Define estimator with data pre-processing steps  
IEstimator<ITransformer> dataPrepEstimator =  
    mlContext.Transforms.Concatenate("Features", featureColumnNames)  
    .Append(mlContext.Transforms.NormalizeMinMax("Features"));  
  
// 3. Create transformer using the data pre-processing estimator  
ITransformer dataPrepTransformer = dataPrepEstimator.Fit(data);  
  
// 4. Pre-process the training data  
IDataView preprocessedTrainData = dataPrepTransformer.Transform(data);  
  
// 5. Define Stochastic Dual Coordinate Ascent machine learning estimator  
var sdcaEstimator = mlContext.Regression.Trainers.Sdca();  
  
// 6. Train machine learning model  
var sdcaModel = sdcaEstimator.Fit(preprocessedTrainData);
```

Explain the model with Permutation Feature Importance (PFI)

In ML.NET use the [PermutationFeatureImportance](#) method for your respective task.

```
ImmutableArray<RegressionMetricsStatistics> permutationFeatureImportance =  
    mlContext  
    .Regression  
    .PermutationFeatureImportance(sdcaModel, preprocessedTrainData, permutationCount:3);
```

NOTE

For pipelines that combine the preprocessing transforms and trainer, assuming that the trainer is at the end of the pipeline, you'll need to extract it using the [LastTransformer](#) property.

The result of using [PermutationFeatureImportance](#) on the training dataset is an [ImmutableArray](#) of [RegressionMetricsStatistics](#) objects. [RegressionMetricsStatistics](#) provides summary statistics like mean and standard deviation for multiple observations of [RegressionMetrics](#) equal to the number of permutations specified by the [permutationCount](#) parameter.

The metric used to measure feature importance depends on the machine learning task used to solve your problem. For example, regression tasks may use a common evaluation metric such as R-squared to measure importance. For more information on model evaluation metrics, see [evaluate your ML.NET model with metrics](#).

The importance, or in this case, the absolute average decrease in R-squared metric calculated by [PermutationFeatureImportance](#) can then be ordered from most important to least important.

```

// Order features by importance
var featureImportanceMetrics =
    permutationFeatureImportance
        .Select((metric, index) => new { index, metric.RSquared })
        .OrderByDescending(myFeatures => Math.Abs(myFeatures.RSquared.Mean));

Console.WriteLine("Feature\tPFI");

foreach (var feature in featureImportanceMetrics)
{
    Console.WriteLine($"{featureColumnNames[feature.index],-20}|\t{feature.RSquared.Mean:F6}");
}

```

Printing the values for each of the features in `featureImportanceMetrics` would generate output similar to that below. Keep in mind that you should expect to see different results because these values vary based on the data that they are given.

FEATURE	CHANGE TO R-SQUARED
HighwayAccess	-0.042731
StudentTeacherRatio	-0.012730
BusinessCenterDistance	-0.010491
TaxRate	-0.008545
AverageRoomNumber	-0.003949
CrimeRate	-0.003665
CommercialZones	0.002749
HomeAge	-0.002426
ResidentialZones	-0.002319
NearWater	0.000203
PercentPopulationLivingBelowPoverty	0.000031
ToxicWasteLevels	-0.000019

Taking a look at the five most important features for this dataset, the price of a house predicted by this model is influenced by its proximity to highways, student teacher ratio of schools in the area, proximity to major employment centers, property tax rate and average number of rooms in the home.

Next steps

- [Make predictions with a trained model](#)
- [Retrain a model](#)
- [Deploy a model in an ASP.NET Core Web API](#)

Save and load trained models

10/14/2022 • 5 minutes to read • [Edit Online](#)

Learn how to save and load trained models in your application.

Throughout the model building process, a model lives in memory and is accessible throughout the application's lifecycle. However, once the application stops running, if the model is not saved somewhere locally or remotely, it's no longer accessible. Typically models are used at some point after training in other applications either for inference or re-training. Therefore, it's important to store the model. Save and load models using the steps described in subsequent sections of this document when using data preparation and model training pipelines like the one detailed below. Although this sample uses a linear regression model, the same process applies to other ML.NET algorithms.

```
HousingData[] housingData = new HousingData[]
{
    new HousingData
    {
        Size = 600f,
        HistoricalPrices = new float[] { 100000f, 125000f, 122000f },
        CurrentPrice = 170000f
    },
    new HousingData
    {
        Size = 1000f,
        HistoricalPrices = new float[] { 200000f, 250000f, 230000f },
        CurrentPrice = 225000f
    },
    new HousingData
    {
        Size = 1000f,
        HistoricalPrices = new float[] { 126000f, 130000f, 200000f },
        CurrentPrice = 195000f
    }
};

// Create MLContext
MLContext mlContext = new MLContext();

// Load Data
IDataView data = mlContext.Data.LoadFromEnumerable<HousingData>(housingData);

// Define data preparation estimator
EstimatorChain<RegressionPredictionTransformer<LinearRegressionModelParameters>> pipelineEstimator =
    mlContext.Transforms.Concatenate("Features", new string[] { "Size", "HistoricalPrices" })
        .Append(mlContext.Transforms.NormalizeMinMax("Features"))
        .Append(mlContext.Regression.Trainers.Sdca());

// Train model
ITransformer trainedModel = pipelineEstimator.Fit(data);

// Save model
mlContext.Model.Save(trainedModel, data.Schema, "model.zip");
```

Because most models and data preparation pipelines inherit from the same set of classes, the save and load method signatures for these components is the same. Depending on your use case, you can either combine the data preparation pipeline and model into a single `EstimatorChain` which would output a single `ITransformer` or separate them thus creating a separate `ITransformer` for each.

Save a model locally

When saving a model you need two things:

1. The `ITransformer` of the model.
2. The `DataViewSchema` of the `ITransformer`'s expected input.

After training the model, use the `Save` method to save the trained model to a file called `model.zip` using the `DataViewSchema` of the input data.

```
// Save Trained Model  
mlContext.Model.Save(trainedModel, data.Schema, "model.zip");
```

Save an ONNX model locally

To save an ONNX version of your model locally you will need the `Microsoft.ML.OnnxConverter` NuGet package installed.

With the `OnnxConverter` package installed, we can use it to save our model into the ONNX format. This requires a `Stream` object which we can provide as a `FileStream` using the `File.Create` method. The `File.Create` method takes in a string as a parameter which will be the path of the ONNX model.

```
using FileStream stream = File.Create("./onnx_model.onnx");
```

With the stream created, we can call the `ConvertToOnnx` method and give it the trained model, the data used to train the model, and the stream. However, not all trainers and transformers are exportable to ONNX. For a complete list, visit the [Transforms](#) and [How to Choose an ML.NET Algorithm](#) guides.

```
mlContext.Model.ConvertToOnnx(trainedModel, data, stream);
```

Load a model stored locally

Models stored locally can be used in other processes or applications like [ASP .NET Core](#) and [Serverless Web Applications](#). See [Use ML.NET in Web API](#) and [Deploy ML.NET Serverless Web App](#) how-to articles to learn more.

In a separate application or process, use the `Load` method along with the file path to get the trained model into your application.

```
//Define DataViewSchema for data preparation pipeline and trained model  
DataViewSchema modelSchema;  
  
// Load trained model  
ITransformer trainedModel = mlContext.Model.Load("model.zip", out modelSchema);
```

Load an ONNX model locally

To load in an ONNX model for predictions, you will need the `Microsoft.ML.OnnxTransformer` NuGet package.

With the `OnnxTransformer` package installed, you can load an existing ONNX model by using the `ApplyOnnxModel` method. The required parameter is a string which is the path of the local ONNX model.

```
OnnxScoringEstimator estimator = mlContext.Transforms.ApplyOnnxModel("./onnx_model.onnx");
```

The `ApplyOnnxModel` method returns an `OnnxScoringEstimator` object. First, we need to load in the new data.

```
HousingData[] newHousingData = new HousingData[]
{
    new()
    {
        Size = 1000f,
        HistoricalPrices = new[] { 300_000f, 350_000f, 450_000f },
        CurrentPrice = 550_000f
    }
};
```

With the new data we can load that into an `IDataView` using the `LoadFromEnumerable` method.

```
IDataView newHousingDataView = mlContext.Data.LoadFromEnumerable(newHousingData);
```

Now, we can use the new `IDataView` to fit on the new data.

```
estimator.Fit(newHousingDataView);
```

After using the `Fit` method on an estimator from `ApplyOnnxModel`, it can then be saved as a new model using the `Save` method mentioned [save a model locally section](#).

Load a model stored remotely

To load data preparation pipelines and models stored in a remote location into your application, use a `Stream` instead of a file path in the `Load` method.

```
// Create MLContext
MLContext mlContext = new MLContext();

// Define DataViewSchema and ITransformers
DataViewSchema modelSchema;
ITransformer trainedModel;

// Load data prep pipeline and trained model
using (HttpClient client = new HttpClient())
{
    Stream modelFile = await client.GetStreamAsync("<YOUR-REMOTE-FILE-LOCATION>");

    trainedModel = mlContext.Model.Load(modelFile, out modelSchema);
}
```

Working with separate data preparation and model pipelines

NOTE

Working with separate data preparation and model training pipelines is optional. Separation of pipelines makes it easier to inspect the learned model parameters. For predictions, it's easier to save and load a single pipeline that includes the data preparation and model training operations.

When working with separate data preparation pipelines and models, the same process as single pipelines

applies; except now both pipelines need to be saved and loaded simultaneously.

Given separate data preparation and model training pipelines:

```
// Define data preparation estimator
IEstimator<ITransformer> dataPrepEstimator =
    mlContext.Transforms.Concatenate("Features", new string[] { "Size", "HistoricalPrices" })
        .Append(mlContext.Transforms.NormalizeMinMax("Features"));

// Create data preparation transformer
ITransformer dataPrepTransformer = dataPrepEstimator.Fit(data);

// Define StochasticDualCoordinateAscent regression algorithm estimator
var sdcaEstimator = mlContext.Regression.Trainers.Sdca();

// Pre-process data using data prep operations
IDataView transformedData = dataPrepTransformer.Transform(data);

// Train regression model
RegressionPredictionTransformer<LinearRegressionModelParameters> trainedModel =
    sdcaEstimator.Fit(transformedData);
```

Save data preparation pipeline and trained model

To save both the data preparation pipeline and trained model, use the following commands:

```
// Save Data Prep transformer
mlContext.Model.Save(dataPrepTransformer, data.Schema, "data_preparation_pipeline.zip");

// Save Trained Model
mlContext.Model.Save(trainedModel, transformedData.Schema, "model.zip");
```

Load data preparation pipeline and trained model

In a separate process or application, load the data preparation pipeline and trained model simultaneously as follows:

```
// Create MLContext
MLContext mlContext = new MLContext();

// Define data preparation and trained model schemas
DataViewSchema dataPrepPipelineSchema, modelSchema;

// Load data preparation pipeline and trained model
ITransformer dataPrepPipeline = mlContext.Model.Load("data_preparation_pipeline.zip", out
dataPrepPipelineSchema);
ITransformer trainedModel = mlContext.Model.Load("model.zip", out modelSchema);
```

Make predictions with a trained model

10/14/2022 • 3 minutes to read • [Edit Online](#)

Learn how to use a trained model to make predictions

Create data models

Input data

```
public class HousingData
{
    [LoadColumn(0)]
    public float Size { get; set; }

    [LoadColumn(1, 3)]
    [VectorType(3)]
    public float[] HistoricalPrices { get; set; }

    [LoadColumn(4)]
    [ColumnName("Label")]
    public float CurrentPrice { get; set; }
}
```

Output data

Like the `Features` and `Label` input column names, ML.NET has default names for the predicted value columns produced by a model. Depending on the task the name may differ.

Because the algorithm used in this sample is a linear regression algorithm, the default name of the output column is `Score` which is defined by the `ColumnName` attribute on the `PredictedPrice` property.

```
class HousingPrediction
{
    [ColumnName("Score")]
    public float PredictedPrice { get; set; }
}
```

Set up a prediction pipeline

Whether making a single or batch prediction, the prediction pipeline needs to be loaded into the application. This pipeline contains both the data pre-processing transformations as well as the trained model. The code snippet below loads the prediction pipeline from a file named `model.zip`.

```
//Create MLContext
MLContext mlContext = new MLContext();

// Load Trained Model
DataViewSchema predictionPipelineSchema;
ITransformer predictionPipeline = mlContext.Model.Load("model.zip", out predictionPipelineSchema);
```

Single prediction

To make a single prediction, create a `PredictionEngine` using the loaded prediction pipeline.

```
// Create PredictionEngines
PredictionEngine<HousingData, HousingPrediction> predictionEngine =
    mlContext.Model.CreatePredictionEngine<HousingData, HousingPrediction>(predictionPipeline);
```

Then, use the `Predict` method and pass in your input data as a parameter. Notice that using the `Predict` method does not require the input to be an `IDataView`. This is because it conveniently internalizes the input data type manipulation so you can pass in an object of the input data type. Additionally, since `CurrentPrice` is the target or label you're trying to predict using new data, it's assumed there is no value for it at the moment.

```
// Input Data
HousingData inputData = new HousingData
{
    Size = 900f,
    HistoricalPrices = new float[] { 155000f, 190000f, 220000f }
};

// Get Prediction
HousingPrediction prediction = predictionEngine.Predict(inputData);
```

If you access the `Score` property of the `prediction` object, you should get a value similar to `150079`.

TIP

`PredictionEngine` is not thread-safe. Additionally, you have to create an instance of it everywhere it is needed within your application. As your application grows, this process can become unmanageable. For improved performance and thread safety, use a combination of dependency injection and the `PredictionEnginePool` service, which creates an `ObjectPool` of `PredictionEngine` objects for use throughout your application.

For examples on how to use the `PredictionEnginePool` service, see [deploy a model to a web API](#) and [deploy a model to Azure Functions](#).

See [dependency injection in ASP.NET Core](#) for more information.

Multiple predictions (`IDataView`)

Given the following data, load it into an `IDataView`. In this case, the name of the `IDataView` is `inputData`. Because `CurrentPrice` is the target or label you're trying to predict using new data, it's assumed there is no value for it at the moment.

```
// Actual data
HousingData[] housingData = new HousingData[]
{
    new HousingData
    {
        Size = 850f,
        HistoricalPrices = new float[] { 150000f, 175000f, 210000f }
    },
    new HousingData
    {
        Size = 900f,
        HistoricalPrices = new float[] { 155000f, 190000f, 220000f }
    },
    new HousingData
    {
        Size = 550f,
        HistoricalPrices = new float[] { 99000f, 98000f, 130000f }
    }
};
```

Then, use the `Transform` method to apply the data transformations and generate predictions.

```
// Predicted Data
IDataView predictions = predictionPipeline.Transform(inputData);
```

Inspect the predicted values by using the `GetColumn` method.

```
// Get Predictions
float[] scoreColumn = predictions.GetColumn<float>("Score").ToArray();
```

The predicted values in the score column should look like the following:

OBSERVATION	PREDICTION
1	144638.2
2	150079.4
3	107789.8

Multiple predictions (PredictionEnginePool)

To make multiple predictions using `PredictionEnginePool`, you can take an `IEnumerable` containing multiple instances of your model input. For example an `IEnumerable<HousingInput>` and apply the `Predict` method to each element using LINQ's `Select` method.

This code sample assumes you have a `PredictionEnginePool` called `predictionEnginePool` and an `IEnumerable<HousingData>` called `housingData`.

```
IEnumerable<HousingPrediction> predictions = housingData.Select(input =>
predictionEnginePool.Predict(input));
```

The result is an `IEnumerable` containing instances of your predictions. In this case, it would be `IEnumerable<HousingPrediction>`.

Re-train a model

10/14/2022 • 2 minutes to read • [Edit Online](#)

Learn how to retrain a machine learning model in ML.NET.

The world and the data around it change at a constant pace. As such, models need to change and update as well. ML.NET provides functionality for re-training models using learned model parameters as a starting point to continually build on previous experience rather than starting from scratch every time.

The following algorithms are re-trainable in ML.NET:

- [AveragedPerceptronTrainer](#)
- [FieldAwareFactorizationMachineTrainer](#)
- [LbfgsLogisticRegressionBinaryTrainer](#)
- [LbfgsMaximumEntropyMulticlassTrainer](#)
- [LbfgsPoissonRegressionTrainer](#)
- [LinearSvmTrainer](#)
- [OnlineGradientDescentTrainer](#)
- [SgdCalibratedTrainer](#)
- [SgdNonCalibratedTrainer](#)
- [SymbolicSgdLogisticRegressionBinaryTrainer](#)

Load pre-trained model

First, load the pre-trained model into your application. To learn more about loading training pipelines and models, see [Save and load a trained model](#).

```
// Create MLContext
MLContext mlContext = new MLContext();

// Define DataViewSchema of data prep pipeline and trained model
DataViewSchema dataPrepPipelineSchema, modelSchema;

// Load data preparation pipeline
ITransformer dataPrepPipeline = mlContext.Model.Load("data_preparation_pipeline.zip", out
dataPrepPipelineSchema);

// Load trained model
ITransformer trainedModel = mlContext.Model.Load("ogd_model.zip", out modelSchema);
```

Extract pre-trained model parameters

Once the model is loaded, extract the learned model parameters by accessing the [Model](#) property of the pre-trained model. The pre-trained model was trained using the linear regression model

[OnlineGradientDescentTrainer](#) which creates a [RegressionPredictionTransformer](#) that outputs [LinearRegressionModelParameters](#). These model parameters contain the learned bias and weights or coefficients of the model. These values will be used as a starting point for the new re-trained model.

```
// Extract trained model parameters
LinearRegressionModelParameters originalModelParameters =
    ((ISingleFeaturePredictionTransformer<object>)trainedModel).Model as LinearRegressionModelParameters;
```

NOTE

The model parameters output depend on the algorithm used. For example [OnlineGradientDescentTrainer](#) uses [LinearRegressionModelParameters](#), while [LbfgsMaximumEntropyMulticlassTrainer](#) outputs [MaximumEntropyModelParameters](#). When extracting model parameters, cast to the appropriate type.

Re-train model

The process for retraining a model is no different than that of training a model. The only difference is, the [Fit](#) method in addition to the data also takes as input the original learned model parameters and uses them as a starting point in the re-training process.

```
// New Data
HousingData[] housingData = new HousingData[]
{
    new HousingData
    {
        Size = 850f,
        HistoricalPrices = new float[] { 150000f, 175000f, 210000f },
        CurrentPrice = 205000f
    },
    new HousingData
    {
        Size = 900f,
        HistoricalPrices = new float[] { 155000f, 190000f, 220000f },
        CurrentPrice = 210000f
    },
    new HousingData
    {
        Size = 550f,
        HistoricalPrices = new float[] { 99000f, 98000f, 130000f },
        CurrentPrice = 180000f
    }
};

//Load New Data
IDataView newData = mlContext.Data.LoadFromEnumerable<HousingData>(housingData);

// Preprocess Data
IDataView transformedNewData = dataPrepPipeline.Transform(newData);

// Retrain model
RegressionPredictionTransformer<LinearRegressionModelParameters> retrainedModel =
    mlContext.Regression.Trainers.OnlineGradientDescent()
        .Fit(transformedNewData, originalModelParameters);
```

At this point, you can save your re-trained model and use it in your application. For more information, see the [save and load a trained model](#) and [make predictions with a trained model](#) guides.

Compare model parameters

How do you know if re-training actually happened? One way would be to compare whether the re-trained model's parameters are different than those of the original model. The code sample below compares the original against the re-trained model weights and outputs them to the console.

```

// Extract Model Parameters of re-trained model
LinearRegressionModelParameters retrainedModelParameters = retrainedModel.Model as
LinearRegressionModelParameters;

// Inspect Change in Weights
var weightDiffs =
    originalModelParameters.Weights.Zip(
        retrainedModelParameters.Weights, (original, retrained) => original - retrained).ToArray();

Console.WriteLine("Original | Retrained | Difference");
for(int i=0;i < weightDiffs.Count();i++)
{
    Console.WriteLine($"{originalModelParameters.Weights[i]} | {retrainedModelParameters.Weights[i]} | 
{weightDiffs[i]}");
}

```

The table below shows what the output might look like.

ORIGINAL	RETRAINED	DIFFERENCE
33039.86	56293.76	-23253.9
29099.14	49586.03	-20486.89
28938.38	48609.23	-19670.85
30484.02	53745.43	-23261.41

Deploy a model to Azure Functions

10/14/2022 • 8 minutes to read • [Edit Online](#)

Learn how to deploy a pre-trained ML.NET machine learning model for predictions over HTTP through an Azure Functions serverless environment.

NOTE

This sample runs a preview version of the `PredictionEnginePool` service.

Prerequisites

- [Visual Studio 2019](#) or later or Visual Studio 2017 version 15.6 or later with the ".NET Core cross-platform development" and "Azure development" workloads installed.
- [Azure Functions Tools](#)
- PowerShell
- Pre-trained model. Use the [ML.NET Sentiment Analysis tutorial](#) to build your own model or download this [pre-trained sentiment analysis machine learning model](#)

Azure Functions sample overview

This sample is a **C# HTTP Trigger Azure Functions application** that uses a pretrained binary classification model to categorize the sentiment of text as positive or negative. Azure Functions provides an easy way to run small pieces of code at scale on a managed serverless environment in the cloud. The code for this sample can be found on the [dotnet/machinelearning-samples](#) repository on GitHub.

Create Azure Functions project

1. Open Visual Studio 2017. Select **File > New > Project** from the menu bar. In the **New Project** dialog, select the **Visual C#** node followed by the **Cloud** node. Then select the **Azure Functions** project template. In the **Name** text box, type "SentimentAnalysisFunctionsApp" and then select the **OK** button.
2. In the **New Project** dialog, open the dropdown above the project options and select **Azure Functions v2 (.NET Core)**. Then, select the **Http trigger** project and then select the **OK** button.
3. Create a directory named *MLModels* in your project to save your model:

In **Solution Explorer**, right-click on your project and select **Add > New Folder**. Type "MLModels" and hit Enter.

4. Install the **Microsoft.ML NuGet Package** version 1.3.1:

In **Solution Explorer**, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the **Browse** tab, search for **Microsoft.ML**, select that 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.

5. Install the **Microsoft.Azure.Functions.Extensions NuGet Package**:

In **Solution Explorer**, right-click on your project and select **Manage NuGet Packages**. Choose

"nuget.org" as the Package source, select the Browse tab, search for **Microsoft.Azure.Functions.Extensions**, select that 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.

6. Install the **Microsoft.Extensions.ML** NuGet Package version 0.15.1:

In Solution Explorer, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.Extensions.ML**, select that 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.

7. Install the **Microsoft.NET.Sdk.Functions** NuGet Package version 1.0.31:

In Solution Explorer, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the Installed tab, search for **Microsoft.NET.Sdk.Functions**, select that package in the list, select 1.0.31 from the Version dropdown, and select the **Update** 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.

Add pre-trained model to project

1. Copy your pre-built model to the *MLModels* folder.
2. In Solution Explorer, right-click your pre-built model file and select **Properties**. Under **Advanced**, change the value of **Copy to Output Directory** to **Copy if newer**.

Create Azure Function to analyze sentiment

Create a class to predict sentiment. Add a new class to your project:

1. In **Solution Explorer**, right-click the project, and then select **Add > New Item**.
2. In the **Add New Item** dialog box, select **Azure Function** and change the **Name** field to *AnalyzeSentiment.cs*. Then, select the **Add** button.
3. In the **New Azure Function** dialog box, select **Http Trigger**. Then, select the **OK** button.

The *AnalyzeSentiment.cs* file opens in the code editor. Add the following `using` statement to the top of *AnalyzeSentiment.cs*:

```
using System;
using System.IO;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.Http;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Microsoft.Extensions.ML;
using SentimentAnalysisFunctionsApp.DataModels;
```

By default, the `AnalyzeSentiment` class is `static`. Make sure to remove the `static` keyword from the class definition.

```
public class AnalyzeSentiment
{
}
```

Create data models

You need to create some classes for your input data and predictions. Add a new class to your project:

1. Create a directory named *DataModels* in your project to save your data models: In Solution Explorer, right-click on your project and select **Add > New Folder**. Type "DataModels" and hit Enter.
2. In Solution Explorer, right-click the *DataModels* directory, and then select **Add > New Item**.
3. In the **Add New Item** dialog box, select **Class** and change the **Name** field to *SentimentData.cs*. Then, select the **Add** button.

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

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

Remove the existing class definition and add the following code to the *SentimentData.cs* file:

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

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

4. In Solution Explorer, right-click the *DataModels* directory, and then select **Add > New Item**.
5. In the **Add New Item** dialog box, select **Class** and change the **Name** field to *SentimentPrediction.cs*. Then, select the **Add** button. The *SentimentPrediction.cs* file opens in the code editor. Add the following using statement to the top of *SentimentPrediction.cs*:

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

Remove the existing class definition and add the following code to the *SentimentPrediction.cs* file:

```
public class SentimentPrediction : SentimentData
{
    [ColumnName("PredictedLabel")]
    public bool Prediction { get; set; }

    public float Probability { get; set; }

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

`SentimentPrediction` inherits from `SentimentData` which provides access to the original data in the

`SentimentText` property as well as the output generated by the model.

Register PredictionEnginePool service

To make a single prediction, you have to create a `PredictionEngine`. `PredictionEngine` is not thread-safe. Additionally, you have to create an instance of it everywhere it is needed within your application. As your application grows, this process can become unmanageable. For improved performance and thread safety, use a combination of dependency injection and the `PredictionEnginePool` service, which creates an `ObjectPool` of `PredictionEngine` objects for use throughout your application.

The following link provides more information if you want to learn more about [dependency injection](#).

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 *Startup.cs*. Then, select the **Add** button.
3. Add the following using statements to the top of *Startup.cs*.

```
using System;
using System.IO;
using Microsoft.Azure.Functions.Extensions.DependencyInjection;
using Microsoft.Extensions.ML;
using SentimentAnalysisFunctionsApp;
using SentimentAnalysisFunctionsApp.DataModels;
```

4. Remove the existing code below the using statements and add the following code:

```
[assembly: FunctionsStartup(typeof(Startup))]
namespace SentimentAnalysisFunctionsApp
{
    public class Startup : FunctionsStartup
    {
        ...
    }
}
```

5. Define variables to store the environment the app is running in and the file path where the model is located inside the `Startup` class

```
private readonly string _environment;
private readonly string _modelPath;
```

6. Below that, create a constructor to set the values of the `_environment` and `_modelPath` variables. When the application is running locally, the default environment is *Development*.

```

public Startup()
{
    _environment = Environment.GetEnvironmentVariable("AZURE_FUNCTIONS_ENVIRONMENT");

    if (_environment == "Development")
    {
        _modelPath = Path.Combine("MLModels", "sentiment_model.zip");
    }
    else
    {
        string deploymentPath = @"D:\home\site\wwwroot\";
        _modelPath = Path.Combine(deploymentPath, "MLModels", "sentiment_model.zip");
    }
}

```

7. Then, add a new method called `Configure` to register the `PredictionEnginePool` service below the constructor.

```

public override void Configure(IFunctionsHostBuilder builder)
{
    builder.Services.AddPredictionEnginePool<SentimentData, SentimentPrediction>()
        .FromFile(modelName: "SentimentAnalysisModel", filePath: _modelPath, watchForChanges: true);
}

```

At a high level, this code initializes the objects and services automatically for later use when requested by the application instead of having to manually do it.

Machine learning models are not static. As new training data becomes available, the model is retrained and redeployed. One way to get the latest version of the model into your application is to restart or redeploy your application. However, this introduces application downtime. The `PredictionEnginePool` service provides a mechanism to reload an updated model without restarting or redeploying your application.

Set the `watchForChanges` parameter to `true`, and the `PredictionEnginePool` starts a `FileSystemWatcher` that listens to the file system change notifications and raises events when there is a change to the file. This prompts the `PredictionEnginePool` to automatically reload the model.

The model is identified by the `modelName` parameter so that more than one model per application can be reloaded upon change.

TIP

Alternatively, you can use the `FromUri` method when working with models stored remotely. Rather than watching for file changed events, `FromUri` polls the remote location for changes. The polling interval defaults to 5 minutes. You can increase or decrease the polling interval based on your application's requirements. In the code sample below, the `PredictionEnginePool` polls the model stored at the specified URI every minute.

```

builder.Services.AddPredictionEnginePool<SentimentData, SentimentPrediction>()
    .FromUri(
        modelName: "SentimentAnalysisModel",
        uri:"https://github.com/dotnet/samples/raw/main/machine-
learning/models/sentimentanalysis/sentiment_model.zip",
        period: TimeSpan.FromMinutes(1));

```

Load the model into the function

Insert the following code inside the `AnalyzeSentiment` class:

```

private readonly PredictionEnginePool<SentimentData, SentimentPrediction> _predictionEnginePool;

// AnalyzeSentiment class constructor
public AnalyzeSentiment(PredictionEnginePool<SentimentData, SentimentPrediction> predictionEnginePool)
{
    _predictionEnginePool = predictionEnginePool;
}

```

This code assigns the `PredictionEnginePool` by passing it to the function's constructor which you get via dependency injection.

Use the model to make predictions

Replace the existing implementation of `Run` method in `AnalyzeSentiment` class with the following code:

```

[FunctionName("AnalyzeSentiment")]
public async Task<IActionResult> Run(
    [HttpTrigger(AuthorizationLevel.Function, "post", Route = null)] HttpRequest req,
    ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    //Parse HTTP Request Body
    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    SentimentData data = JsonConvert.DeserializeObject<SentimentData>(requestBody);

    //Make Prediction
    SentimentPrediction prediction = _predictionEnginePool.Predict(modelName: "SentimentAnalysisModel",
example: data);

    //Convert prediction to string
    string sentiment = Convert.ToBoolean(prediction.Prediction) ? "Positive" : "Negative";

    //Return Prediction
    return (ActionResult)new OkObjectResult(sentiment);
}

```

When the `Run` method executes, the incoming data from the HTTP request is deserialized and used as input for the `PredictionEnginePool`. The `Predict` method is then called to make predictions using the `SentimentAnalysisModel` registered in the `Startup` class and returns the results back to the user if successful.

Test locally

Now that everything is set up, it's time to test the application:

1. Run the application
2. Open PowerShell and enter the code into the prompt where PORT is the port your application is running on. Typically the port is 7071.

```

Invoke-RestMethod "http://localhost:<PORT>/api/AnalyzeSentiment" -Method Post -Body
(@{SentimentText="This is a very bad steak"} | ConvertTo-Json) -ContentType "application/json"

```

If successful, the output should look similar to the text below:

```
Negative
```

Congratulations! You have successfully served your model to make predictions over the internet using an Azure Function.

Next Steps

- [Deploy to Azure](#)

Deploy a model in an ASP.NET Core Web API

10/14/2022 • 6 minutes to read • [Edit Online](#)

Learn how to serve a pre-trained ML.NET machine learning model on the web using an ASP.NET Core Web API. Serving a model over a web API enables predictions via standard HTTP methods.

Prerequisites

- [Visual Studio 2019](#) or later or Visual Studio 2017 version 15.6 or later with the ".NET Core cross-platform development" workload installed.
- PowerShell.
- Pre-trained model. Use the [ML.NET Sentiment Analysis tutorial](#) to build your own model or download this [pre-trained sentiment analysis machine learning model](#)

Create ASP.NET Core Web API project

1. Open Visual Studio 2017. Select **File > New > Project** from the menu bar. In the New Project dialog, select the **Visual C# node** followed by the **Web node**. Then select the **ASP.NET Core Web Application** project template. In the **Name** text box, type "SentimentAnalysisWebAPI" and then select the **OK** button.
2. In the window that displays the different types of ASP.NET Core Projects, select **API** and then select the **OK** button.
3. Create a directory named *MLModels* in your project to save your pre-built machine learning model files:

In Solution Explorer, right-click on your project and select Add > New Folder. Type "MLModels" and hit Enter.

4. Install the **Microsoft.ML NuGet Package**:

In Solution Explorer, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.ML**, select that 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.

5. Install the **Microsoft.Extensions.ML Nuget Package**:

In Solution Explorer, right-click on your project and select **Manage NuGet Packages**. Choose "nuget.org" as the Package source, select the Browse tab, search for **Microsoft.Extensions.ML**, select that 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.

Add model to ASP.NET Core Web API project

1. Copy your pre-built model to the *MLModels* directory
2. In Solution Explorer, right-click the model zip file and select Properties. Under Advanced, change the value of **Copy to Output Directory** to **Copy if newer**.

Create data models

You need to create some classes for your input data and predictions. Add a new class to your project:

1. Create a directory named *DataModels* in your project to save your data models:

In Solution Explorer, right-click on your project and select Add > New Folder. Type "DataModels" and hit Enter.

2. In Solution Explorer, right-click the *DataModels* directory, and then select Add > New Item.

3. In the Add New Item dialog box, select Class and change the Name field to *SentimentData.cs*. Then, select the Add button. The *SentimentData.cs* file opens in the code editor. Add the following using statement to the top of *SentimentData.cs*:

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

Remove the existing class definition and add the following code to the *SentimentData.cs* file:

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

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

4. In Solution Explorer, right-click the *DataModels* directory, and then select Add > New Item.

5. In the Add New Item dialog box, select Class and change the Name field to *SentimentPrediction.cs*.

Then, select the Add button. The *SentimentPrediction.cs* file opens in the code editor. Add the following using statement to the top of *SentimentPrediction.cs*:

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

Remove the existing class definition and add the following code to the *SentimentPrediction.cs* file:

```
public class SentimentPrediction : SentimentData
{
    [ColumnName("PredictedLabel")]
    public bool Prediction { get; set; }

    public float Probability { get; set; }

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

`SentimentPrediction` inherits from `SentimentData`. This makes it easier to see the original data in the `SentimentText` property along with the output generated by the model.

Register `PredictionEnginePool` for use in the application

To make a single prediction, you have to create a `PredictionEngine`. `PredictionEngine` is not thread-safe. Additionally, you have to create an instance of it everywhere it is needed within your application. As your application grows, this process can become unmanageable. For improved performance and thread safety, use a combination of dependency injection and the `PredictionEnginePool` service, which creates an `ObjectPool` of `PredictionEngine` objects for use throughout your application.

The following link provides more information if you want to learn more about [dependency injection in ASP.NET Core](#).

1. Open the *Startup.cs* class and add the following using statement to the top of the file:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.ML;
using SentimentAnalysisWebAPI.DataModels;
```

2. Add the following code to the *ConfigureServices* method:

```
services.AddPredictionEnginePool<SentimentData, SentimentPrediction>()
    .FromFile(modelName: "SentimentAnalysisModel", filePath:"MLModels/sentiment_model.zip",
    watchForChanges: true);
```

At a high level, this code initializes the objects and services automatically for later use when requested by the application instead of having to manually do it.

Machine learning models are not static. As new training data becomes available, the model is retrained and redeployed. One way to get the latest version of the model into your application is to restart or redeploy your application. However, this introduces application downtime. The `PredictionEnginePool` service provides a mechanism to reload an updated model without restarting or redeploying your application.

Set the `watchForChanges` parameter to `true`, and the `PredictionEnginePool` starts a `FileSystemWatcher` that listens to the file system change notifications and raises events when there is a change to the file. This prompts the `PredictionEnginePool` to automatically reload the model.

The model is identified by the `modelName` parameter so that more than one model per application can be reloaded upon change.

TIP

Alternatively, you can use the `FromUri` method when working with models stored remotely. Rather than watching for file changed events, `FromUri` polls the remote location for changes. The polling interval defaults to 5 minutes. You can increase or decrease the polling interval based on your application's requirements. In the code sample below, the `PredictionEnginePool` polls the model stored at the specified URI every minute.

```
services.AddPredictionEnginePool<SentimentData, SentimentPrediction>()
    .FromUri(
        modelName: "SentimentAnalysisModel",
        uri:"https://github.com/dotnet/samples/raw/main/machine-
learning/models/sentimentanalysis/sentiment_model.zip",
        period: TimeSpan.FromMinutes(1));
```

Create Predict controller

To process your incoming HTTP requests, create a controller.

1. In Solution Explorer, right-click the *Controllers* directory, and then select **Add > Controller**.
2. In the **Add New Item** dialog box, select **API Controller Empty** and select **Add**.
3. In the prompt change the **Controller Name** field to *PredictController.cs*. Then, select the **Add** button. The

PredictController.cs file opens in the code editor. Add the following using statement to the top of *PredictController.cs*:

```
using System;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.ML;
using SentimentAnalysisWebAPI.DataModels;
```

Remove the existing class definition and add the following code to the *PredictController.cs* file:

```
public class PredictController : ControllerBase
{
    private readonly PredictionEnginePool<SentimentData, SentimentPrediction> _predictionEnginePool;

    public PredictController(PredictionEnginePool<SentimentData, SentimentPrediction>
predictionEnginePool)
    {
        _predictionEnginePool = predictionEnginePool;
    }

    [HttpPost]
    public ActionResult<string> Post([FromBody] SentimentData input)
    {
        if(!ModelState.IsValid)
        {
            return BadRequest();
        }

        SentimentPrediction prediction = _predictionEnginePool.Predict(modelName:
"SentimentAnalysisModel", example: input);

        string sentiment = Convert.ToBoolean(prediction.Prediction) ? "Positive" : "Negative";

        return Ok(sentiment);
    }
}
```

This code assigns the `PredictionEnginePool` by passing it to the controller's constructor which you get via dependency injection. Then, the `Predict` controller's `Post` method uses the `PredictionEnginePool` to make predictions using the `SentimentAnalysisModel` registered in the `Startup` class and returns the results back to the user if successful.

Test web API locally

Once everything is set up, it's time to test the application.

1. Run the application.
2. Open PowerShell and enter the following code where PORT is the port your application is listening on.

```
Invoke-RestMethod "https://localhost:<PORT>/api/predict" -Method Post -Body (@{SentimentText="This
was a very bad steak"} | ConvertTo-Json) -ContentType "application/json"
```

If successful, the output should look similar to the text below:

```
Negative
```

Congratulations! You have successfully served your model to make predictions over the internet using an

Next Steps

- [Deploy to Azure](#)

Tutorial: Sentiment analysis with .NET for Apache Spark and ML.NET

10/14/2022 • 5 minutes to read • [Edit Online](#)

This tutorial teaches you how to do sentiment analysis of online reviews using ML.NET and .NET for Apache Spark. [ML.NET](#) is a free, cross-platform, open-source machine learning framework. You can use ML.NET with .NET for Apache Spark to scale the training and prediction of machine learning algorithms.

In this tutorial, you learn how to:

- Create a sentiment analysis model using ML.NET Model Builder in Visual Studio.
- Create a .NET for Apache Spark console app.
- Write and implement a user-defined function.
- Run a .NET for Apache Spark console app.

Prerequisites

- If you haven't developed a .NET for Apache Spark application before, start with the [Getting Started tutorial](#) to become familiar with the basics. Complete all of the prerequisites for the Getting Started tutorial before you continue with this tutorial.
- This tutorial uses the ML.NET Model Builder (preview), a visual interface available in Visual Studio. If you don't already have Visual Studio, you can [download the Community version of Visual Studio](#) for free.
- [Download and install](#) ML.NET Model Builder (preview).
- Download the [yelptrain.csv](#) and [yelptrain.csv](#) Yelp review datasets.

Review the data

The Yelp reviews dataset contains online Yelp reviews about various services. Open [yelptrain.csv](#) and notice the structure of the data. The first column contains review text, and the second column contains sentiment scores. If the sentiment score is 1, the review is positive, and if the sentiment score is 0, the review is negative.

The following table contains sample data:

REVIEWTEXT	SENTIMENT
Wow... Loved this place.	1
Crust is not good.	0

Build your machine learning model

1. Open Visual Studio and create a new C# Console App (.NET Core). Name the project *MLSparkModel*.
2. In **Solution Explorer**, right-click the *MLSparkModel* project. Then select **Add > Machine Learning**.
3. From the ML.NET Model Builder, select the **Sentiment Analysis** scenario tile.
4. On the **Add data** page, upload the *yelptrain.csv* data set.

5. Choose *Sentiment* from the **Columns to Predict** dropdown.
6. On the **Train** page, set the time to train to *60 seconds* and select **Start training**. Notice the status of your training under **Progress**.
7. Once Model Builder is finished training, **Evaluate** the training results. You can type phrases into the text box below **Try your model** and select **Predict** to see the output.
8. Select **Code** and then select **Add Projects** to add the ML model to the solution.
9. Notice that two projects are added to your solutions: **MLSparkModelML.ConsoleApp** and **MLSparkModelML.Model**.
10. Double-click on your *MLSpark C#* project and notice that the following project reference has been added.

```
<ItemGroup>
  <ProjectReference Include="..\MLSparkModelML.Model\MLSparkModelML.Model.csproj" />
</ItemGroup>
```

Create a console app

Model Builder creates a console app for you.

1. Right-click on **MLSparkModelML.Console** in Solution Explorer, and select **Manage NuGet Packages**.
2. Search for **Microsoft.Spark** and install the package. **Microsoft.ML** is automatically installed for you by Model Builder.

Create a SparkSession

1. Open the *Program.cs* file for **MLSparkModelML.ConsoleApp**. This file was autogenerated by Model Builder. Delete the `using` statements, the contents of the `Main()` method, and the `createSingleDataSample` region.
2. Add the following additional `using` statements to the top of the *Program.cs*:

```
using System;
using System.Collections.Generic;
using Microsoft.ML;
using Microsoft.ML.Data;
using Microsoft.Spark.Sql;
using MLSparkModelML.Model;
```

3. Change the `DATA_FILEPATH` to the path of your *yelp/test.csv*.
4. Add the following code to your `Main()` method to create a new `SparkSession`. The Spark Session is the entry point to programming Spark with the Dataset and DataFrame API.

```
SparkSession spark = SparkSession
  .Builder()
  .AppName(".NET for Apache Spark Sentiment Analysis")
  .GetOrCreate();
```

Calling the `spark` object created above allows you to access Spark and DataFrame functionality throughout your program.

Create a DataFrame and print to console

Read in the Yelp review data from the *yelp/test.csv* file as a `DataFrame`. Include `header` and `inferSchema` options.

The `header` option reads the first line of `yelptest.csv` as column names instead of data. The `inferSchema` option infers column types based on the data.

```
DataFrame df = spark
    .ReadStream()
    .Option("header", true)
    .Option("inferSchema", true)
    .Csv(DATA_FILEPATH);

df.Show();
```

Register a user-defined function

You can use UDFs, *user-defined functions*, in Spark applications to do calculations and analysis on your data. In this tutorial, you use ML.NET with a UDF to evaluate each Yelp review.

Add the following code to your `Main` method to register a UDF called `MLudf`.

```
spark.Udf()
    .Register<string, bool>("MLudf", predict);
```

This UDF takes a Yelp review string as input, and outputs true or false for positive or negative sentiments, respectively. It uses the `predict()` method that you define in a later step.

Use Spark SQL to call the UDF

Now that you've read in your data and incorporated ML, use Spark SQL to call the UDF that will run sentiment analysis on each row of your DataFrame. Add the following code to your `Main` method:

```
// Use Spark SQL to call ML.NET UDF
// Display results of sentiment analysis on reviews
df.CreateOrReplaceTempView("Reviews");
DataFrame sqlDf = spark.Sql("SELECT ReviewText, MLudf(ReviewText) FROM Reviews");
sqlDf.Show();

// Print out first 20 rows of data
// Prevent data getting cut off by setting truncate = 0
sqlDf.Show(20, 0, false);

spark.Stop();
```

Create predict() method

Add the following code before your `Main()` method. This code is similar to what is produced by Model Builder in `ConsumeModel.cs`. Moving this method to your console loads the model loading each time you run your app.

```

private static readonly PredictionEngine<ModelInput, ModelOutput> _predictionEngine;

static Program()
{
    MLContext mlContext = new MLContext();
    ITransformer model = mlContext.Model.Load("MLModel.zip", out DataViewSchema schema);
    _predictionEngine = mlContext.Model.CreatePredictionEngine<ModelInput, ModelOutput>(model);
}

static bool predict(string text)
{
    ModelInput input = new ModelInput
    {
        ReviewText = text
    };

    return _predictionEngine.Predict(input).Prediction;
}

```

Add the model to your console app

In Solution Explorer, copy the *MLModel.zip* file from the **MLSparkModelML.Model** project and paste it in the **MLSparkModelML.ConsoleApp** project. A reference is automatically added in *MLSparkModelML.ConsoleApp.csproj*.

Run your code

Use `spark-submit` to run your code. Navigate to your console app's root folder using the command prompt and run the following commands.

First, clean and publish your app.

```

dotnet clean
dotnet publish

```

Then navigate to the console app's publish folder and run the following `spark-submit` command. Remember to update the command with the actual path of your Microsoft Spark jar file.

```
%SPARK_HOME%\bin\spark-submit --class org.apache.spark.deploy.dotnet.DotnetRunner --master local microsoft-spark-2-4_2.11-1.0.0.jar dotnet MLSparkModelML.ConsoleApp.dll
```

Get the code

This tutorial is similar to the code from the [Sentiment Analysis with Big Data](#) example.

Next steps

Advance to the next article to learn how to do Structured Streaming with .NET for Apache Spark.

[Tutorial: Structured Streaming with .NET for Apache Spark](#)

The ML.NET CLI command reference

10/14/2022 • 8 minutes to read • [Edit Online](#)

The `classification`, `regression`, and `recommendation` commands are the main commands provided by the ML.NET CLI tool. These commands allow you to generate good quality ML.NET models for classification, regression, and recommendation models using automated machine learning (AutoML) as well as the example C# code to run/score that model. In addition, the C# code to train the model is generated for you to research the algorithm and settings of the model.

NOTE

This topic refers to ML.NET CLI and ML.NET AutoML, which are currently in Preview, and material may be subject to change.

Overview

Example usage:

```
mlnet regression --dataset "cars.csv" --label-col price
```

The `mlnet` ML task commands (`classification`, `regression`, and `recommendation`) generate the following assets:

- A serialized model .zip ("best model") ready to use.
- C# code to run/score that generated model.
- C# code with the training code used to generate that model.

The first two assets can directly be used in your end-user apps (ASP.NET Core web app, services, desktop app and more) to make predictions with the model.

The third asset, the training code, shows you what ML.NET API code was used by the CLI to train the generated model, so you can investigate the specific algorithm and settings of the model.

Examples

The simplest CLI command for a classification problem (AutoML infers most of the configuration from the provided data):

```
mlnet classification --dataset "customer-feedback.tsv" --label-col Sentiment
```

Another simple CLI command for a regression problem:

```
mlnet regression --dataset "cars.csv" --label-col Price
```

Create and train a classification model with a train dataset, a test dataset, and further customization explicit arguments:

```
mlnet classification --dataset "/MyDataSets/Population-Training.csv" --test-dataset "/MyDataSets/Population-Test.csv" --label-col "InsuranceRisk" --cache on --train-time 600
```

Command options

The `mlnet` ML task commands (`classification`, `regression`, and `recommendation`) train multiple models based on the provided dataset and ML.NET CLI options. These commands also select the best model, save the model as a serialized .zip file, and generate related C# code for scoring and training.

Classification options

Running `mlnet classification` will train a classification model. Choose this command if you want an ML Model to categorize data into 2 or more classes (e.g. sentiment analysis).

```
mlnet classification

--dataset <path> (REQUIRED)

--label-col <col> (REQUIRED)

--cache <option>

--has-header (Default: true)

--ignore-cols <cols>

--log-file-path <path>

--name <name>

-o, --output <path>

--test-dataset <path>

--train-time <time> (Default: 30 minutes, in seconds)

--validation-dataset <path>

-v, --verbosity <v>

-?, -h, --help
```

Regression options

Running `mlnet regression` will train a regression model. Choose this command if you want an ML Model to predict a numeric value (e.g. price prediction).

```
mlnet regression

--dataset <path> (REQUIRED)

--label-col <col> (REQUIRED)

--cache <option>

--has-header (Default: true)

--ignore-cols <cols>

--log-file-path <path>

--name <name>

-o, --output <path>

--test-dataset <path>

--train-time <time> (Default: 30 minutes, in seconds)

--validation-dataset <path>

-v, --verbosity <v>

-?, -h, --help
```

Recommendation options

Running `mlnet recommendation` will train a recommendation model. Choose this command if you want an ML Model to recommend items to users based on ratings (e.g. product recommendation).

```
mlnet recommendation

--dataset <path> (REQUIRED)

--item-col <col> (REQUIRED)

--rating-col <col> (REQUIRED)

--user-col <col> (REQUIRED)

--cache <option>

--has-header (Default: true)

--log-file-path <path>

--name <name>

-o, --output <path>

--test-dataset <path>

--train-time <time> (Default: 30 minutes, in seconds)

--validation-dataset <path>

-v, --verbosity <v>

-?, -h, --help
```

Invalid input options cause the CLI tool to emit a list of valid inputs and an error message.

Dataset

```
--dataset | -d (string)
```

This argument provides the filepath to either one of the following options:

- *A: The whole dataset file:* If using this option and the user is not providing `--test-dataset` and `--validation-dataset`, then cross-validation (k-fold, etc.) or automated data split approaches will be used internally for validating the model. In that case, the user will just need to provide the dataset filepath.
- *B: The training dataset file:* If the user is also providing datasets for model validation (using `--test-dataset` and optionally `--validation-dataset`), then the `--dataset` argument means to only have the "training dataset". For example, when using an 80% - 20% approach to validate the quality of the model and to obtain accuracy metrics, the "training dataset" will have 80% of the data and the "test dataset" would have 20% of the data.

Test dataset

```
--test-dataset | -t (string)
```

File path pointing to the test dataset file, for example when using an 80% - 20% approach when making regular validations to obtain accuracy metrics.

If using `--test-dataset`, then `--dataset` is also required.

The `--test-dataset` argument is optional unless the `--validation-dataset` is used. In that case, the user must use the three arguments.

Validation dataset

```
--validation-dataset | -v (string)
```

File path pointing to the validation dataset file. The validation dataset is optional, in any case.

If using a `validation dataset`, the behavior should be:

- The `test-dataset` and `--dataset` arguments are also required.
- The `validation-dataset` dataset is used to estimate prediction error for model selection.
- The `test-dataset` is used for assessment of the generalization error of the final chosen model. Ideally, the test set should be kept in a "vault," and be brought out only at the end of the data analysis.

Basically, when using a `validation dataset` plus the `test dataset`, the validation phase is split into two parts:

1. In the first part, you just look at your models and select the best performing approach using the validation data (=validation)
2. Then you estimate the accuracy of the selected approach (=test).

Hence, the separation of data could be 80/10/10 or 75/15/10. For example:

- `training-dataset` file should have 75% of the data.
- `validation-dataset` file should have 15% of the data.
- `test-dataset` file should have 10% of the data.

In any case, those percentages will be decided by the user using the CLI who will provide the files already split.

Label column

--label-col (int or string)

With this argument, a specific objective/target column (the variable that you want to predict) can be specified by using the column's name set in the dataset's header or the column's numeric index in the dataset's file (the column index values start at 0).

This argument is used for *classification* and *regression* problems.

Item column

--item-col (int or string)

The item column has the list of items that users rate (items are recommended to users). This column can be specified by using the column's name set in the dataset's header or the column's numeric index in the dataset's file (the column index values start at 0).

This argument is used only for the *recommendation* task.

Rating column

--rating-col (int or string)

The rating column has the list of ratings that are given to items by users. This column can be specified by using the column's name set in the dataset's header or the column's numeric index in the dataset's file (the column index values start at 0).

This argument is used only for the *recommendation* task.

User column

--user-col (int or string)

The user column has the list of users that give ratings to items. This column can be specified by using the column's name set in the dataset's header or the column's numeric index in the dataset's file (the column index values start at 0).

This argument is used only for the *recommendation* task.

Ignore columns

--ignore-columns (string)

With this argument, you can ignore existing columns in the dataset file so they are not loaded and used by the training processes.

Specify the columns names that you want to ignore. Use ',' (comma with space) or ' ' (space) to separate multiple column names. You can use quotes for column names containing whitespace (e.g. "logged in").

Example:

--ignore-columns email, address, id, logged_in

Has header

--has-header (bool)

Specify if the dataset file(s) have a header row. Possible values are:

- `true`
- `false`

The ML.NET CLI will try to detect this property if this argument is not specified by the user.

Train time

`--train-time` (string)

By default, the maximum exploration / train time is 30 minutes.

This argument sets the maximum time (in seconds) for the process to explore multiple trainers and configurations. The configured time may be exceeded if the provided time is too short (say 2 seconds) for a single iteration. In this case, the actual time is the required time to produce one model configuration in a single iteration.

The needed time for iterations can vary depending on the size of the dataset.

Cache

`--cache` (string)

If you use caching, the whole training dataset will be loaded in-memory.

For small and medium datasets, using cache can drastically improve the training performance, meaning the training time can be shorter than when you don't use cache.

However, for large datasets, loading all the data in memory can impact negatively since you might get out of memory. When training with large dataset files and not using cache, ML.NET will be streaming chunks of data from the drive when it needs to load more data while training.

You can specify the following values:

`on` : Forces cache to be used when training. `off` : Forces cache not to be used when training. `auto` : Depending on AutoML heuristics, the cache will be used or not. Usually, small/medium datasets will use cache and large datasets won't use cache if you use the `auto` choice.

If you don't specify the `--cache` parameter, then the cache `auto` configuration will be used by default.

Name

`--name` (string)

The name for the created output project or solution. If no name is specified, the name `sample-{mltask}` is used.

The ML.NET model file (.ZIP file) will get the same name, as well.

Output path

`--output | -o` (string)

Root location/folder to place the generated output. The default is the current directory.

Verbosity

`--verbosity | -v` (string)

Sets the verbosity level of the standard output.

Allowed values are:

- `q[uiet]`
- `m[inimal]` (by default)
- `diag[nostic]` (logging information level)

By default, the CLI tool should show some minimum feedback (`minimal`) when working, such as mentioning that it is working and if possible how much time is left or what % of the time is completed.

Help

`-h | --help`

Prints out help for the command with a description for each command's parameter.

See also

- [How to install the ML.NET CLI tool](#)
- [Overview of the ML.NET CLI](#)
- [Tutorial: Analyze sentiment using the ML.NET CLI](#)
- [Telemetry in ML.NET CLI](#)

ML.NET resources

10/14/2022 • 2 minutes to read • [Edit Online](#)

The following resources help you to learn more about ML.NET

- [Glossary](#) of machine learning and ML.NET terminology
- [Model Builder Azure Training Resources](#)
- [ML.NET CLI telemetry](#)

Next Steps

Apply your learning by doing one of the ML.NET [ML.NET tutorials](#).

Machine learning glossary of important terms

10/14/2022 • 7 minutes to read • [Edit Online](#)

The following list is a compilation of important machine learning terms that are useful as you build your custom models in ML.NET.

Accuracy

In [classification](#), accuracy is the number of correctly classified items divided by the total number of items in the test set. Ranges from 0 (least accurate) to 1 (most accurate). Accuracy is one of evaluation metrics of the model performance. Consider it in conjunction with [precision](#), [recall](#), and [F-score](#).

Area under the curve (AUC)

In [binary classification](#), an evaluation metric that is the value of the area under the curve that plots the true positives rate (on the y-axis) against the false positives rate (on the x-axis). Ranges from 0.5 (worst) to 1 (best). Also known as the area under the ROC curve, i.e., receiver operating characteristic curve. For more information, see the [Receiver operating characteristic](#) article on Wikipedia.

Binary classification

A [classification](#) case where the [label](#) is only one out of two classes. For more information, see the [Binary classification](#) section of the [Machine learning tasks](#) topic.

Calibration

Calibration is the process of mapping a raw score onto a class membership, for binary and multiclass classification. Some ML.NET trainers have a `NonCalibrated` suffix. These algorithms produce a raw score that then must be mapped to a class probability.

Catalog

In ML.NET, a catalog is a collection of extension functions, grouped by a common purpose.

For example, each machine learning task (binary classification, regression, ranking etc) has a catalog of available machine learning algorithms (trainers). The catalog for the binary classification trainers is:

[BinaryClassificationCatalog.BinaryClassificationTrainers](#).

Classification

When the data is used to predict a category, [supervised machine learning](#) task is called classification. [Binary classification](#) refers to predicting only two categories (for example, classifying an image as a picture of either a 'cat' or a 'dog'). [Multiclass classification](#) refers to predicting multiple categories (for example, when classifying an image as a picture of a specific breed of dog).

Coefficient of determination

In [regression](#), an evaluation metric that indicates how well data fits a model. Ranges from 0 to 1. A value of 0 means that the data is random or otherwise cannot be fit to the model. A value of 1 means that the model exactly matches the data. This is often referred to as r^2 , R^2 , or r-squared.

Data

Data is central to any machine learning application. In ML.NET data is represented by [IDataView](#) objects. Data view objects:

- are made up of columns and rows
- are lazily evaluated, that is they only load data when an operation calls for it
- contain a schema that defines the type, format and length of each column

Estimator

A class in ML.NET that implements the [IEstimator<TTransformer>](#) interface.

An estimator is a specification of a transformation (both data preparation transformation and machine learning model training transformation). Estimators can be chained together into a pipeline of transformations. The parameters of an estimator or pipeline of estimators are learned when [Fit](#) is called. The result of [Fit](#) is a [Transformer](#).

Extension method

A .NET method that is part of a class but is defined outside of the class. The first parameter of an extension method is a static `this` reference to the class to which the extension method belongs.

Extension methods are used extensively in ML.NET to construct instances of [estimators](#).

Feature

A measurable property of the phenomenon being measured, typically a numeric (double) value. Multiple features are referred to as a **Feature vector** and typically stored as `double[]`. Features define the important characteristics of the phenomenon being measured. For more information, see the [Feature](#) article on Wikipedia.

Feature engineering

Feature engineering is the process that involves defining a set of [features](#) and developing software that produces feature vectors from available phenomenon data, i.e., feature extraction. For more information, see the [Feature engineering](#) article on Wikipedia.

F-score

In [classification](#), an evaluation metric that balances [precision](#) and [recall](#).

Hyperparameter

A parameter of a machine learning algorithm. Examples include the number of trees to learn in a decision forest or the step size in a gradient descent algorithm. Values of *Hyperparameters* are set before training the model and govern the process of finding the parameters of the prediction function, for example, the comparison points in a decision tree or the weights in a linear regression model. For more information, see the [Hyperparameter](#) article on Wikipedia.

Label

The element to be predicted with the machine learning model. For example, the breed of dog or a future stock price.

Log loss

In [classification](#), an evaluation metric that characterizes the accuracy of a classifier. The smaller log loss is, the more accurate a classifier is.

Loss function

A loss function is the difference between the training label values and the prediction made by the model. The parameters of the model are estimated by minimizing the loss function.

Different trainers can be configured with different loss functions.

Mean absolute error (MAE)

In [regression](#), an evaluation metric that is the average of all the model errors, where model error is the distance between the predicted [label](#) value and the correct label value.

Model

Traditionally, the parameters for the prediction function. For example, the weights in a linear regression model or the split points in a decision tree. In ML.NET, a model contains all the information necessary to predict the [label](#) of a domain object (for example, image or text). This means that ML.NET models include the featurization steps necessary as well as the parameters for the prediction function.

Multiclass classification

A [classification](#) case where the [label](#) is one out of three or more classes. For more information, see the [Multiclass classification](#) section of the [Machine learning tasks](#) topic.

N-gram

A feature extraction scheme for text data: any sequence of N words turns into a [feature](#) value.

Normalization

Normalization is the process of scaling floating point data to values between 0 and 1. Many of the training algorithms used in ML.NET require input feature data to be normalized. ML.NET provides a series of [transforms for normalization](#)

Numerical feature vector

A [feature](#) vector consisting only of numerical values. This is similar to `double[]`.

Pipeline

All of the operations needed to fit a model to a data set. A pipeline consists of data import, transformation, featurization, and learning steps. Once a pipeline is trained, it turns into a model.

Precision

In [classification](#), the precision for a class is the number of items correctly predicted as belonging to that class divided by the total number of items predicted as belonging to the class.

Recall

In [classification](#), the recall for a class is the number of items correctly predicted as belonging to that class divided by the total number of items that actually belong to the class.

Regularization

Regularization penalizes a linear model for being too complicated. There are two types of regularization:

- L_1 regularization zeros weights for insignificant features. The size of the saved model may become smaller after this type of regularization.
- L_2 regularization minimizes weight range for insignificant features. This is a more general process and is less sensitive to outliers.

Regression

A [supervised machine learning](#) task where the output is a real value, for example, double. Examples include predicting stock prices. For more information, see the [Regression](#) section of the [Machine learning tasks](#) topic.

Relative absolute error

In [regression](#), an evaluation metric that is the sum of all absolute errors divided by the sum of distances between correct [label](#) values and the average of all correct label values.

Relative squared error

In [regression](#), an evaluation metric that is the sum of all squared absolute errors divided by the sum of squared distances between correct [label](#) values and the average of all correct label values.

Root of mean squared error (RMSE)

In [regression](#), an evaluation metric that is the square root of the average of the squares of the errors.

Scoring

Scoring is the process of applying new data to a trained machine learning model, and generating predictions. Scoring is also known as inferencing. Depending on the type of model, the score may be a raw value, a probability, or a category.

Supervised machine learning

A subclass of machine learning in which a desired model predicts the label for yet-unseen data. Examples include classification, regression, and structured prediction. For more information, see the [Supervised learning](#) article on Wikipedia.

Training

The process of identifying a [model](#) for a given training data set. For a linear model, this means finding the weights. For a tree, it involves identifying the split points.

Transformer

An ML.NET class that implements the [ITransformer](#) interface.

A transformer transforms one [IDataView](#) into another. A transformer is created by training an [estimator](#), or an estimator pipeline.

Unsupervised machine learning

A subclass of machine learning in which a desired model finds hidden (or latent) structure in data. Examples include clustering, topic modeling, and dimensionality reduction. For more information, see the [Unsupervised learning](#) article on Wikipedia.

Model Builder Azure Training Resources

10/14/2022 • 3 minutes to read • [Edit Online](#)

The following is a guide to help you learn more about resources used to train models in Azure with Model Builder.

What is an Azure Machine Learning experiment?

An Azure Machine Learning experiment is a resource that needs to be created before running Model Builder training on Azure.

The experiment encapsulates the configuration and results for one or more machine learning training runs. Experiments belong to a specific workspace. The first time an experiment is created, its name is registered in the workspace. Any subsequent runs - if the same experiment name is used - are logged as part of the same experiment. Otherwise, a new experiment is created.

What is an Azure Machine Learning workspace?

A workspace is an Azure Machine Learning resource that provides a central place for all Azure Machine Learning resources and artifacts created as part of training run.

To create an Azure Machine Learning workspace, the following are required:

- Name: A name for your workspace between 3-33 characters. Names may only contain alphanumeric characters and hyphens.
- Region: The geographic location of the data center where your workspace and resources are deployed to. It is recommended that you choose a location close to where you or your customers are.
- Resource group: A container that contains all related resources for an Azure solution.

What is an Azure Machine Learning compute?

An Azure Machine Learning compute is a cloud-based Linux VM used for training.

To create an Azure Machine Learning compute, the following are required:

- Name: A name for your compute between 2-16 characters. Names may only contain alphanumeric characters and hyphens.
- Compute size

Model Builder can use one of the following GPU-optimized compute types:

SIZE	VCPU	MEMORY: GIB	TEMP STORAGE (SSD) GIB	GPU	GPU MEMORY: GIB	MAX DATA DISKS	MAX NICs
Standard_NC12	12	112	680	2	24	48	2
Standard_NC24	24	224	1440	4	48	64	4

Visit the [NC-series Linux VM documentation](#) for more details on GPU optimized compute types.

- Compute priority
 - Low-priority: Suited for tasks with shorter execution times. May be impacted by interruptions and lack of availability. Usually costs less because it takes advantage of surplus capacity in Azure.
 - Dedicated: Suited for tasks of any duration, but especially long-running jobs. Not impacted by interruptions or lack of availability. Usually costs more because it reserves a dedicated set of compute resources in Azure for your tasks.

Training

Training on Azure is only available for the Model Builder image classification scenario. The algorithm used to train these models is a Deep Neural Network based on the ResNet50 architecture. The training process takes some time and the amount of time may vary depending on the size of compute selected as well as amount of data. You can track the progress of your runs by selecting the "Monitor current run in Azure portal" link in Visual Studio.

Results

Once training is complete, two projects are added to your solution with the following suffixes:

- *ConsoleApp*: A C# .NET Core console application that provides starter code to build the prediction pipeline and make predictions.
- *Model*: A C# .NET Standard application that contains the data models that define the schema of input and output model data as well as the following assets:
 - *bestModel.onnx*: A serialized version of the model in Open Neural Network Exchange (ONNX) format. ONNX is an open source format for AI models that supports interoperability between frameworks like ML.NET, PyTorch and TensorFlow.
 - *bestModelMap.json*: A list of categories used when making predictions to map the model output to a text category.
 - *MLModel.zip*: A serialized version of the ML.NET prediction pipeline that uses the serialized version of the model *bestModel.onnx* to make predictions and maps outputs using the `bestModelMap.json` file.

Use the machine learning model

The `ModelInput` and `ModelOutput` classes in the *Model* project define the schema of the model's expected input and output respectively.

In an image classification scenario, the `ModelInput` contains two columns:

- `ImageSource` : The string path of the image location.
- `Label` : The actual category the image belongs to. `Label` is only used as an input when training and does not need to be provided when making predictions.

The `ModelOutput` contains two columns:

- `Prediction` : The image's predicted category.
- `Score` : The list of probabilities for all categories (the highest belongs to the `Prediction`).

Troubleshooting

Cannot create compute

If an error occurs during Azure Machine Learning compute creation, the compute resource may still exist, in an errored state. If you try to re-create the compute resource with the same name, the operation fails. To fix this

error, either:

- Create the new compute with a different name
- Go to the Azure portal, and remove the original compute resource

Telemetry collection by the ML.NET CLI

10/14/2022 • 2 minutes to read • [Edit Online](#)

The [ML.NET CLI](#) includes a telemetry feature that collects anonymous usage data that is aggregated for use by Microsoft.

How Microsoft uses the data

The product team uses ML.NET CLI telemetry data to help understand how to improve the tools. For example, if customers infrequently use a particular machine learning task, the product team investigates why and uses findings to prioritize feature development. ML.NET CLI telemetry also helps with debugging of issues such as crashes and code anomalies.

While the product team appreciates this insight, we also know that not everyone wants to send this data. [Find out how to disable telemetry](#).

Scope

The `mlnet` command launches the ML.NET CLI, but the command itself doesn't collect telemetry.

Telemetry *isn't enabled* when you run the `mlnet` command with no other command attached. For example:

- `mlnet`
- `mlnet --help`

Telemetry *is enabled* when you run an [ML.NET CLI command](#), such as `mlnet classification`.

Opt out of data collection

The ML.NET CLI telemetry feature is enabled by default.

Opt out of the telemetry feature by setting the `MLDOTNET_CLI_TELEMETRY_OPTOUT` environment variable to `1` or `true`. This environment variable applies globally to the ML.NET CLI tool.

Data points collected

The feature collects the following data:

- What command was invoked, such as `classification`
- Command-line parameter names used (that is, "dataset, label-col, output-path, train-time, verbosity")
- Hashed MAC address: a cryptographically (SHA256) anonymous and unique ID for a machine
- Timestamp of an invocation
- Three octet IP address (not full IP address) used only to determine geographical location
- Name of all arguments/parameters used. Not the customer's values, such as strings
- Hashed dataset filename
- Dataset file-size bucket
- Operating system and version
- Value of ML task commands: Categorical values, such as `regression`, `classification`, and `recommendation`
- ML.NET CLI version (that is, 0.3.27703.4)

The data is sent securely to Microsoft servers using [Azure Application Insights](#) technology, held under restricted access, and used under strict security controls from secure [Azure Storage](#) systems.

Data points not collected

The telemetry feature *doesn't* collect:

- personal data, such as usernames
- dataset filenames
- data from dataset files

If you suspect that the ML.NET CLI telemetry is collecting sensitive data or that the data is being insecurely or inappropriately handled, file an issue in the [ML.NET](#) repository for investigation.

License

The Microsoft distribution of ML.NET CLI is licensed with the [Microsoft Software License Terms: Microsoft .NET Library](#). For details on data collection and processing, see the section entitled "Data."

Disclosure

When you first run a [ML.NET CLI command](#) such as `mlnet classification`, the ML.NET CLI tool displays disclosure text that tells you how to opt out of telemetry. Text may vary slightly depending on the version of the CLI you're running.

See also

- [ML.NET CLI reference](#)
- [Microsoft Software License Terms: Microsoft .NET Library](#)
- [Privacy at Microsoft](#)
- [Microsoft Privacy Statement](#)