

ML.NET

SUCCINCTLY

BY ED FREITAS

ML.NET Succinctly

Ed Freitas

Foreword by Daniel Jebaraj



Copyright © 2024 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 111

Morrisville, NC 27560

USA

All rights reserved.

ISBN: 978-1-64200-235-5

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Tres Watkins, VP of content, Syncfusion, Inc.

Acquisitions Coordinator: Tres Watkins, VP of content, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

Introduction.....	10
This book's approach	11
Chapter 1 Getting Started.....	12
The basics.....	12
Common ML approaches	13
Supervised and unsupervised algorithm types	14
How machine learning works in a nutshell	14
Enter ML.NET	15
Using ML.NET with Visual Studio	16
Using the latest model builder	19
Summary.....	20
Chapter 2 Core ML.NET and Binary Classification	21
Quick intro	21
Types of operations.....	21
Data operations.....	22
Model operations.....	23
Data transformations	23
Trainers.....	23
Adding a Model	23
Scenarios and tasks	27
Training with Model Builder	29
Evaluating with Model Builder.....	35
Consuming the model.....	36
Generated model (TestModel.consumption.cs)	38

Generated model (TestModel.training.cs).....	43
Summary.....	47
Chapter 3 Value Prediction.....	48
Quick intro	48
Dataset explanation.....	48
Dataset columns.....	48
Creating the model	49
ValuePredictionModel.consumption.cs	56
ValuePredictionModel.training.cs	61
ValuePredictionModel.training vs. TestModel.training.....	64
Summary.....	66
Chapter 4 Image Classification	67
Quick intro	67
Imaging dataset.....	67
ImgClass project.....	68
ImageClassification.cs.....	69
ImageClassification.cs steps	70
Add Model classes	71
Model path and prediction engine.....	72
Create prediction engine	74
Predict.....	75
Build the pipeline	77
Retrain pipeline	79
Generating the model	81
Evaluating the model.....	83
Consuming the model and app execution.....	86

Installing LG.Microsoft.ML.Vision	88
Fixing the “missing” ImgClass.zip	90
Fixing the missing TensorFlow DLL.....	91
Rerunning the app.....	92
Summary and final thoughts	94

The *Succinctly* Series of Books

Daniel Jebaraj
CEO of Syncfusion, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 1.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas is a consultant on business process automation and a software developer focused on customer success. He likes technology and enjoys learning, playing soccer, running, traveling, and being around his family.

Ed is available at <https://edfreitas.me>.

Acknowledgments

A huge thank you to the fantastic [Syncfusion](#) team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, Graham High, Daniel Jebaraj, and the manuscript managers who thoroughly reviewed the book's organization.

I also want to thank the technical editor, [Dr. James McCaffrey](#) from [Microsoft Research](#). He worked on a predecessor to ML.NET and reviewed the book for quality and technical accuracy. Thank you all.

I dedicate this book to my beloved Lala, Chelin, and Puntico. May your journeys be blessed.

Introduction

Creating applications that stand out has become increasingly complex over the past few years, primarily since apps have become more intelligent. From applications that can make recommendations to apps that can make predictions or recognize specific intents, machine learning is now baked into (almost) every significant app out there.

If you are a .NET developer and have wondered how to get into machine learning (ML) and create models without knowing how machine learning works behind the scenes, you've come to the right place. [ML.NET](#) is a cross-platform library specifically designed for .NET developers with little to no experience with the theory behind machine learning.

ML.NET began as an internal-only Microsoft library named TMSN just a few months after C# and .NET version 1.0 were released in 2002. The TMSN library proved popular and evolved into an internal TLC system ("The Learning Code"). The TLC library was then used as the basis for a publicly available command line tool named MAML (Microsoft Azure Machine Learning). The MAML library was then used as the basis for the first version of the publicly available ML.NET library.

ML.NET allows .NET developers to reuse existing skills and integrate ML capabilities into .NET applications. These include performing sentiment analysis, creating product recommendations, creating price predictions, performing object detection, detecting fraudulent transactions, segmenting customers into groups, forecasting sales, performing image classification, and running many other forecasting and machine-learning activities.

ML.NET can also be extended to work with popular ML libraries, such as [TensorFlow](#), [ONNX](#), and [Infer.NET](#), delivering high accuracy and [outstanding performance](#).

Beyond that, ML.NET powers various well-known [Microsoft](#) products, such as Microsoft Defender, Outlook, [Bing](#), [Power Apps](#), and [Power BI](#), among others.

So, whether you're off to build the next-generation dating app, create a financial market analysis and prediction application, or anything in between, and want to infuse the power of machine learning into your .NET app, using ML.NET is a great way to go.

This book aims to get you started with ML.NET and show you how to integrate this framework into a .NET application. Working with ML.NET does not require any machine learning theoretical know-how or experience; however, it is essential to have some intermediate C# coding knowledge, such as a solid understanding of classes, interfaces, objects, and method-chaining.

Overall, ML.NET empowers .NET developers who do not have previous machine learning experience to add the power of machine learning algorithms to .NET apps with little effort. Given that machine learning is a complex, broad, and ever-expanding subject, I'll present the topics throughout this book using an easy-to-understand language and describing the fundamental concepts behind this technology with an approachable and friendly vocabulary for .NET developers with no prior ML experience.

The book's overall goal is to make an incredibly complex topic easy for anyone in the .NET community to understand, digest, and quickly get up to speed with. Therefore, this book is not a deep and thorough dive into the complex world of machine learning and the theory behind it. It's more like a cheat sheet for .NET developers to get a basic-enough understanding of what machine learning is and how to use it within a .NET application.

So, if this sounds like music to your ears, join me on this journey, and let's explore ML.NET together.

This book's approach

This book will look at ML.NET from a different angle (compared to what you'll find written anywhere else about ML.NET). I will use some of the great features within Visual Studio, specifically Model Builder, to generate the code we'll explore throughout this book. Model Builder is a Visual Studio extension that is an easy-to-use wrapper over the AutoML set of APIs that automatically generate ML.NET code.

Instead of breaking our heads trying to learn the classes and methods required to build different types of machine-learning models, the idea is to let the automated features of the framework do the heavy lifting, so we can learn from the generated code.

Overall, we'll use Model Builder to generate various models, inspect the code generated for each, and compare how they differ. This approach will give us insights into how the code varies (from model to model) depending on the machine-learning algorithms and scenarios employed.



Note: *All the code snippets explained throughout this book will exclude unnecessary using statements and first-line comments that are auto-generated. This is to show the code as concisely as possible.*



Note: *The book's [GitHub repository](#) includes all the Visual Studio projects, code, and datasets used throughout the book. Note that these projects were created using Visual Studio 2022 Enterprise. You can use Visual Studio 2022 Community Edition (free), too.*

Chapter 1 Getting Started

The basics

Before exploring ML.NET, let's understand what machine learning (ML) is and how much value it brings.

In simple terms, machine learning uses historical data to make predictions. It looks at patterns within that data to create a mathematical model of the data, and then it processes similar data to make predictions. The following diagram illustrates this sequence.

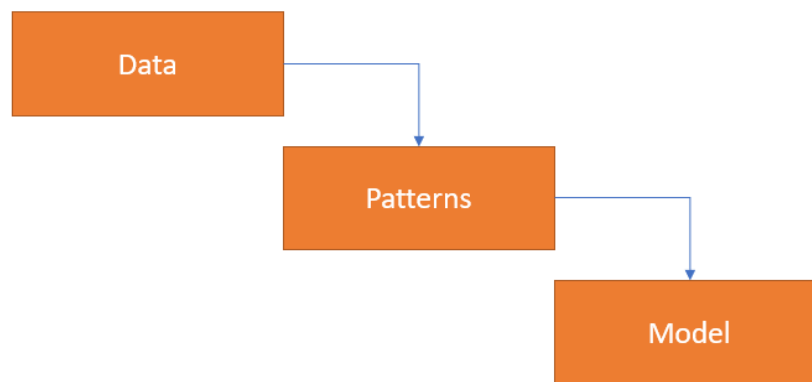


Figure 1-a: Machine Learning—A Conceptual Representation

This process works because the new data is not memorized. Instead, based on the initial dataset supplied for creating the model, the machine learning process can find the same or similar patterns in the new data, and as such, make predictions.

Three common categories of prediction are image classification (is the image a dog or a cat?), tabular data (what is the creditworthiness of the loan applicant?), and natural language processing (is this restaurant review positive or negative?).

It is essential to understand that the initial dataset supplied must be good—in other words, the historical data needs to be of good quality—curated data. Using quality historical data will lead to a model producing good predictions, whereas using low-quality data will lead to a model that will likely make poor or inaccurate predictions, leading to erroneous results.

The better your historical data, the better your model will be. Machine learning uses specific algorithms to find patterns and retrieve insights from data.

Common ML approaches

There are two mainstream approaches to machine learning. One is called [supervised](#) and the other [unsupervised](#). Note that these are not the only ways to do machine learning but are the ones you'll most likely encounter at the beginning of your machine-learning journey.



Figure 1-b: The Most Common Machine Learning Approaches

In simple terms, with supervised machine learning, your dataset will contain a column (label) whose values you'll want to predict.

Let's say, for example, you work at Interpol as an analyst and have a list of fugitives and a column that indicates the probability of finding the offender. You would use this column to produce a prediction—this would be an example of supervised learning.

On the contrary, using the unsupervised learning approach, the dataset would not have the column or label containing the probability of finding the fugitive for the algorithm to predict. In this case, the algorithm is left on its own to find structure within the input provided—thus, it is called unsupervised.

In a nutshell, with supervised learning, example inputs and desired outputs (resultant columns or labels) are presented, allowing the computer to find patterns and rules that map inputs to the desired outcomes.

On the other hand, no result columns or labels are given to the algorithm with unsupervised learning, so it must find patterns and structure within the data provided. The following diagram illustrates these concepts.

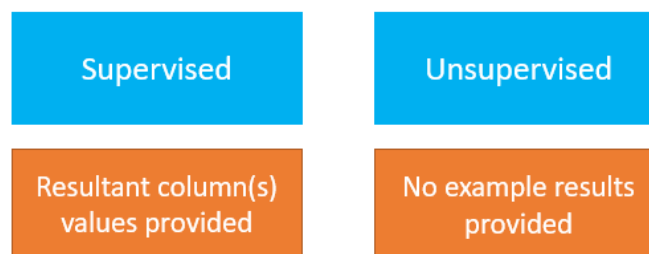


Figure 1-c: The Supervised and Unsupervised Approaches

Beyond supervised and unsupervised learning, there are other types of machine learning approaches, such as (but not limited to) [semi-supervised](#), [reinforcement](#), [meta-learning](#), [topic modeling](#), and [deep learning](#) based on the extensive use of [artificial neural networks](#).

Supervised and unsupervised algorithm types

Whether we use supervised or unsupervised learning, different algorithm types exist. Here are some of the most common algorithm types used by supervised learning:

- **Regression:** Used to predict values, such as an employee's salary. An increase is often used for time-series problems.
- **Classification:** Used for predicting classes, such as classifying pictures into different class types: houses, cars, airplanes, toys, etc. Classification is usually divided into binary classification (prediction with precisely two possible values) or multiclass classification (three or more possible values).

As for unsupervised learning, some of the most common algorithm types employed are:

- **Clustering:** Used for predicting groups with similar patterns, such as grouping online banking users based on their app usage habits.
- **Anomaly detection:** Used to predict elements that don't align with the pattern found in the rest of the data, such as financial fraud.

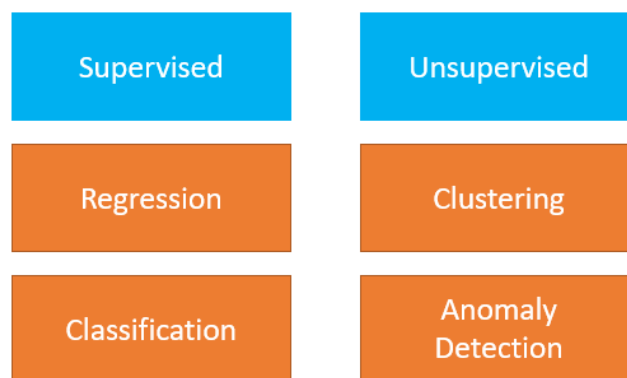


Figure 1-d: Common Machine Learning Algorithm Types (Supervised and Unsupervised)

How machine learning works in a nutshell

Having seen different types and algorithms, let's now look closely at how the machine-learning process works. As you already know, the process starts with data, and it's essential to have good data for the model to perform well and give good results. Insufficient data leads to bad outcomes; as the old saying goes, garbage in, garbage out.

Once we have the historical data we will use, we need to go through data preparation or data cleaning—this includes performing activities such as adding, updating, or removing missing values.

Other data preparation activities include converting text values into numerical values, given that most algorithms work best with numerical values.

After the data preparation step, we must create the model, including deciding which algorithm type to use. For each of the previously discussed machine-learning types, there are many different algorithms to choose from, each with its advantages and disadvantages. Which works best depends on your data.

Once the model has been created, the next step is to evaluate it to check if it executes well and gives good results for new data it hasn't processed.

The reasoning behind incorporating the evaluation step is to avoid providing the same data used to create the model (because that data is well-known). Instead, new data is used to check whether the model can generalize well. One key aspect is that the evaluation step is an iterative process. The model may not perform well when you evaluate it with the new data. In that case, you might have to try with a different algorithm, recreate the model, then try the new model with the new data as often as required to achieve good results.

Once you have a good-enough model, the next step is to deploy it in production where it will be used.

It is possible once the model is in production that the data will change over time, invalidating the previously created and deployed model and degrading the system's performance and results. In that case, you'll have to go over the initial data-gathering step and repeat the process.

The following diagram illustrates the complete process just described.

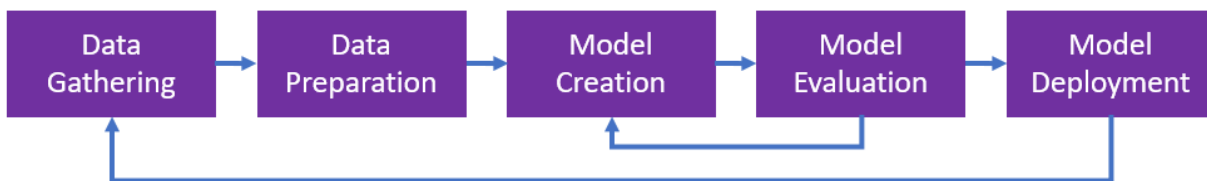


Figure 1-e: Machine Learning Process

As you have seen, the machine learning process itself is not that complicated. Instead, it's a thorough, well thought-out, and iterative process of refining the data and algorithm chosen to create the best possible model.

Enter ML.NET

ML.NET is a machine learning library designed for .NET developers (fully compliant with .NET Standard), mostly C# developers and other developers who use different languages that work with .NET, such as F# or Visual Basic.

Today's most prominent machine-learning frameworks are typically used in combination with Python, and using these frameworks requires machine learning knowledge to some degree. ML.NET has been designed from the ground up in C#, and it specifically caters to .NET developers with no prior experience with machine learning.

In essence, ML.NET is nothing more than a NuGet package and a set of Visual Studio tools with machine learning capabilities baked in and available out of the box, a package you can install and use within Visual Studio with your projects.

[Model Builder](#), which uses AutoML (automated machine learning), provides an approachable and easy user interface to create, train, and deploy machine learning models. With Model Builder, it's possible to employ different algorithms, metrics, and configuration options to create the best possible model for your data.

Another great feature of Model Builder is that it offers several built-in scenarios with different machine-learning use cases. Beyond producing a trained model, Model Builder can work with CSV files and SQL databases and generate the source code required to load your model, make predictions, and connect to cloud resources.

Overall, ML.NET is an excellent choice for any .NET developer coming into this fascinating world, as it includes various high-level features that make it easy to implement machine learning in a .NET application.

Using ML.NET with Visual Studio

ML.NET works best when used with Visual Studio. In my case, I'll be using Visual Studio 2022 Enterprise Edition. However, you may also use [Visual Studio 2022 Community Edition](#). If you don't have Visual Studio installed, ensure it is installed before following along.

With Visual Studio open, click the **File** menu, then click **New**, and then **Project**—this will display the following screen.

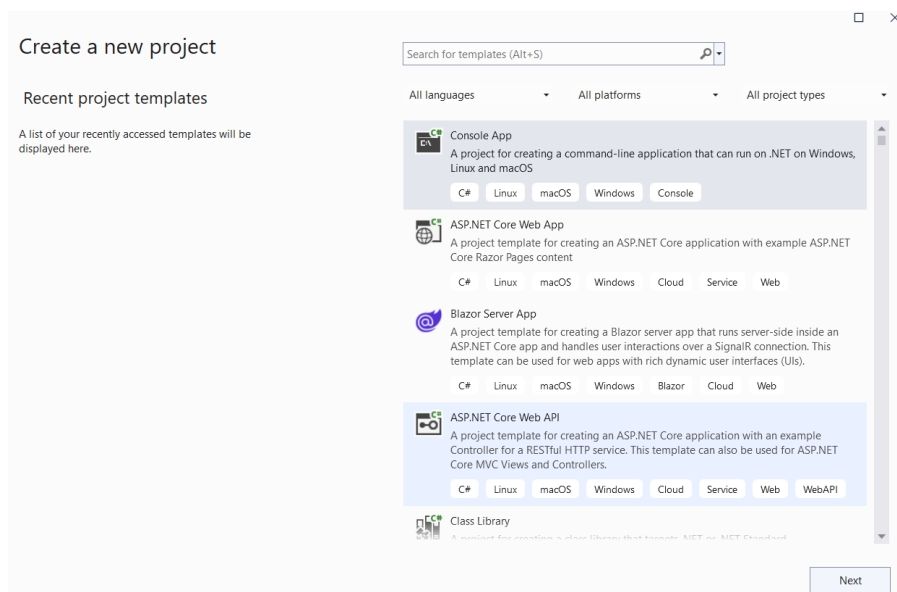


Figure 1-f: The “Create a new project” Screen (Visual Studio)

Then, select the **Console App** project option and click **Next**. You'll be shown the following screen, where you can specify the project's name. I've renamed the project's default name to **TestML** and have chosen the project's **Location**.

You are free to name the project something completely different. However, keeping the same name I use might help you follow along easier. Once done, click **Next**.

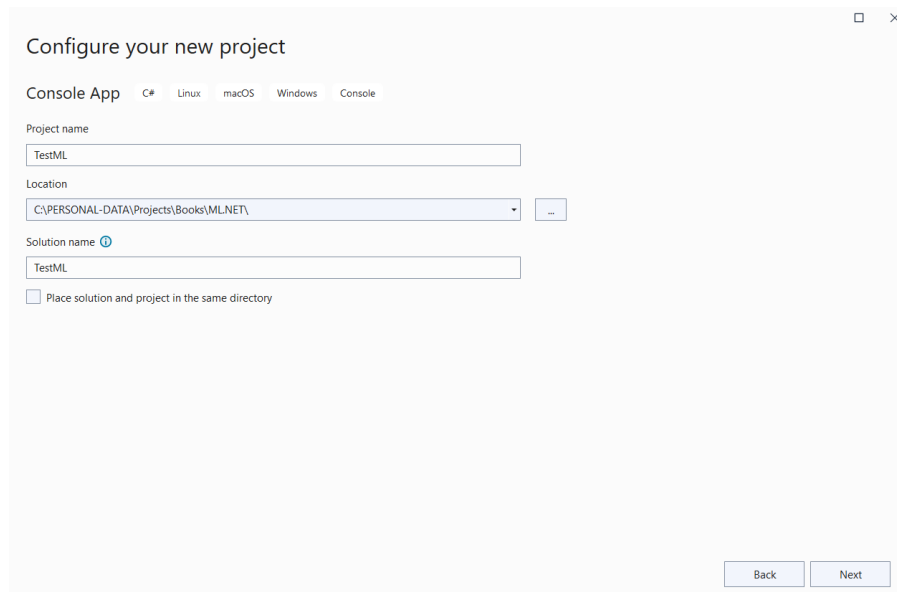
The screenshot shows a dialog box titled "Configure your new project". At the top, there are tabs for "Console App", "C#", "Linux", "macOS", "Windows", and "Console". The "Console App" tab is selected. Below the tabs, there are three input fields: "Project name" with the text "TestML", "Location" with a dropdown menu showing "C:\PERSONAL-DATA\Projects\Books\ML.NET" and a browse button "...", and "Solution name" with the text "TestML". There is a checkbox labeled "Place solution and project in the same directory" which is currently unchecked. At the bottom right, there are "Back" and "Next" buttons.

Figure 1-g: The "Configure your new project" Screen (Visual Studio)

After that, you will be given the option to choose a .NET version. I'm selecting **.NET 6.0 (Long Term Support)**. However, it will also work with **.NET 7.0 (Standard Term Support)**.

Then, click **Create**.

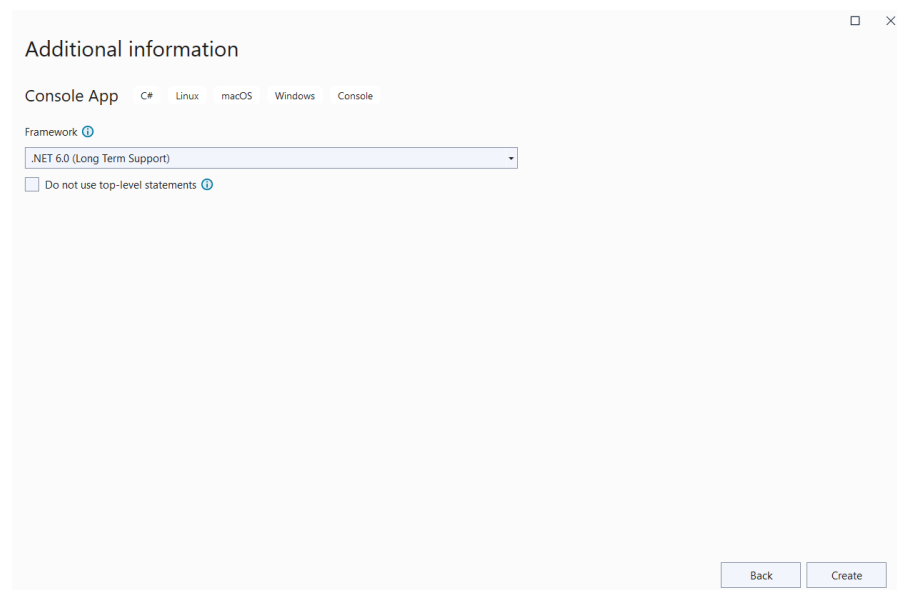
The screenshot shows a dialog box titled "Additional information". At the top, there are tabs for "Console App", "C#", "Linux", "macOS", "Windows", and "Console". The "Console App" tab is selected. Below the tabs, there is a "Framework" dropdown menu showing ".NET 6.0 (Long Term Support)". There is a checkbox labeled "Do not use top-level statements" which is currently unchecked. At the bottom right, there are "Back" and "Create" buttons.

Figure 1-h: The "Additional information" Screen (Visual Studio)

Once the console application has been created, you should see a screen similar to the following one, showing the project in Visual Studio.

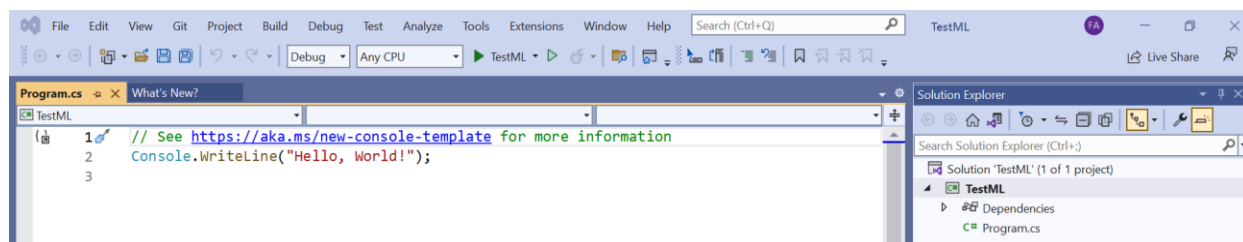


Figure 1-i: The Project Created (Visual Studio) Using [Top-Level Statements](#)

From .NET 6, C# console project [templates](#) use top-level statements. These allow you, as a developer, to avoid placing your program's entry point in a static method within a class, thus enabling you to write less code.

I think this is a great feature. However, throughout this book, I'll be using the `Program.Main` style, instead. It is the traditional (old) program style that has been used since the early days of .NET for C# applications. You can easily switch from top-level statements to the `Program.Main` style by clicking on the paintbrush or light bulb icon within the Visual Studio editor.



Note: From now on, all the code snippets explained throughout this book will be using the `Program.Main` style and not top-level statements.

Then, within the **Solution Explorer**, click on **Dependencies**, right-click, and after the menu opens, click **Manage NuGet Package**.

From the **Browse** tab, search for **Microsoft.ML**, click on the **Microsoft.ML** package to select it, and then click **Install**.

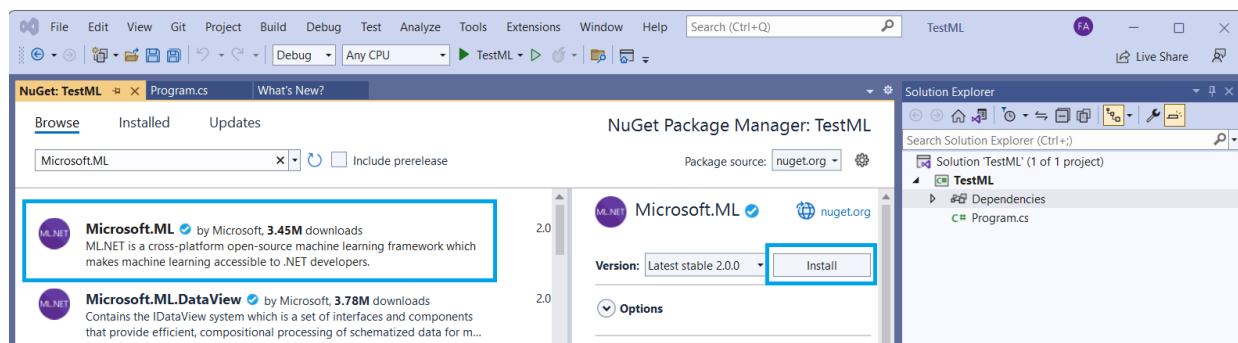


Figure 1-j: The NuGet Package Manager (Visual Studio)

When prompted, click **OK** on the **Preview Changes** dialog, as seen in the following image, to continue with the package installation. Review any license terms shortly after when prompted.

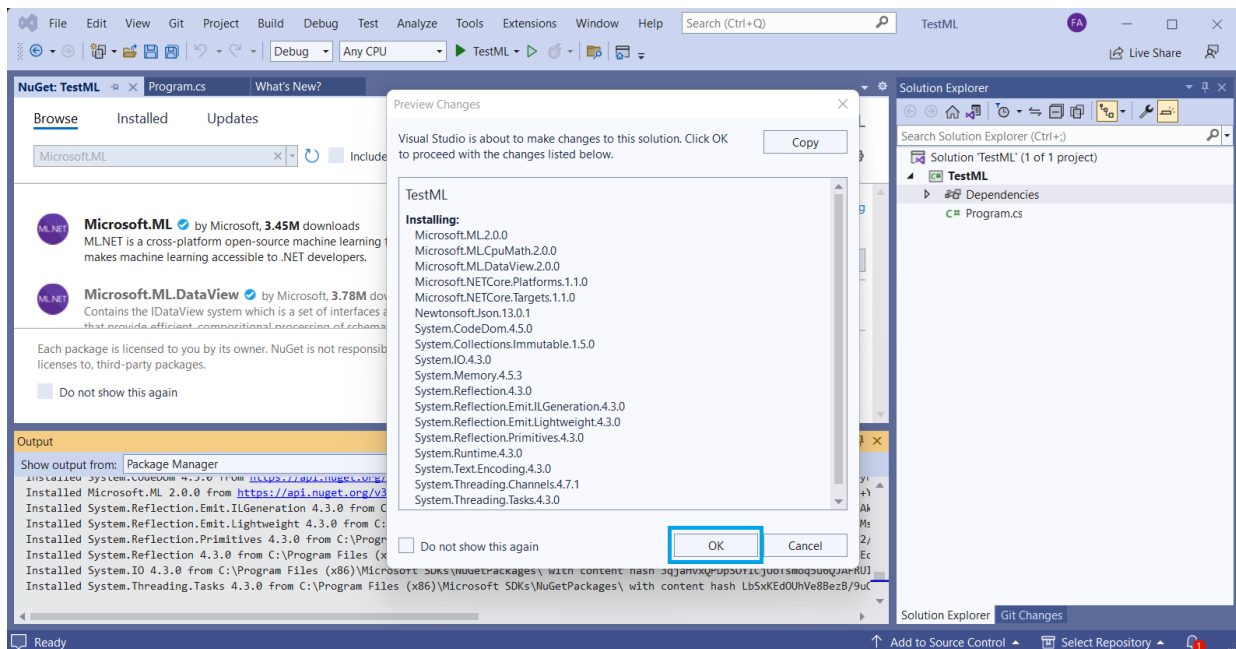


Figure 1-k: Installing the Microsoft.ML NuGet Package (Preview Changes–Visual Studio)

Once the package installation process has been finalized, you'll be able to see the **Microsoft.ML** dependency within **Solution Explorer**.

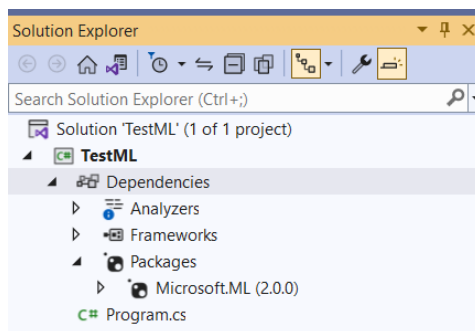


Figure 1-l: Project Dependencies within the Solution Explorer (Visual Studio)

Using the latest model builder

Having an environment with the latest version of Model Builder is critical. Typically, this is the case if Visual Studio was installed recently. If your Visual Studio installation is not recent, I suggest you download and install the latest version of Model Builder from this [URL](#).

Summary

Well done! We had a quick overview of machine learning, learned the basics, installed ML.NET, and are ready to use it. The following chapters will dig deeper into ML.NET and explore the ML context object and Model Builder.

Chapter 2 Core ML.NET and Binary Classification

Quick intro

Let's dive a bit deeper into ML.NET. For that, we need to look at ML context, which is the starting point for all the operations executed with ML.NET, such as loading data, creating and evaluating models, and getting detailed information about what is happening with the pipeline, such as errors or other events.

ML context can also seed the environment for splitting data and helping others run your code to achieve similar results. In the broader context of things, you can think of ML context as the starting point of ML.NET.

So, within our project's **Program.cs** file, let's implement ML context as follows—highlighted in bold.

Code Listing 2-a: Program.cs - TestML

```
using Microsoft.ML;

internal class Program
{
    private static void Main(string[] args)
    {
        var context = new MLContext();

        Console.WriteLine("Hello, World!");
    }
}
```

By invoking **new MLContext()**, we have implemented ML context within our application and have a starting point to work with ML.NET.

ML context provides a set of properties and methods, commonly referred to as the ML.NET API, that allow us to execute various (types of) operations.

Types of operations

ML context provides various operations that the ML.NET API can perform, which can be divided into four categories. They are data operations, model operations, data transformations, and algorithms that can be used for training (trainers), as illustrated in the following figure.

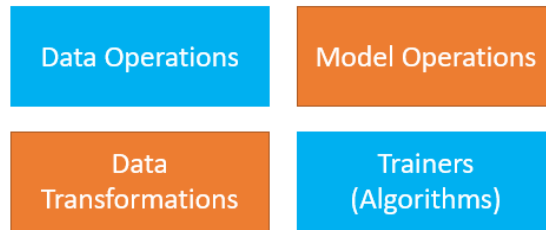


Figure 2-a: Types of ML.NET Operations

The **data operations** category includes methods that allow ML.NET to load data from different sources.

The **model operations** category includes methods that can be executed on the model itself, either on an existing model or a completely new one.

The **data transformations** category includes methods used to process the data to get it into specific formats that machine learning algorithms might require.

The **trainers** category is a set of algorithms built into ML.NET that can be used for different machine-learning scenarios.

So, let's continue and explore the methods available to load and use data from different sources.

Data operations

The ML.NET API is part of ML context, and we can use various methods to load data into our ML.NET-powered app—these methods are known as data operations.

The data loading methods are available as part of the **Data** property of ML context (**context**)—these are:

- **LoadFromBinary**: Loads data from a binary file.
- **LoadFromTextFile**: Loads data from text files, including CSVs.
- **LoadFromEnumerable**: Loads data like arrays or lists.
- **CreateDatabaseLoader**: Connects to a SQL Server instance to retrieve data.
- **Filter**: As its name suggests, it is used for filtering data.
- **TestTrainSplit**: As its name suggests, it divides source data into a set for training a model and a set for testing and evaluating the trained model.
- **Shuffle**: As its name suggests, it is used to randomize the order in which data is processed during training, which is necessary to prevent training from stalling.

Model operations

Next, we have operations we can execute for the model itself to start making predictions, and we can save a model once we are happy with it. Some of the methods to do this are available as part of the **Model** property of ML context (**context**):

- **Load**: As its name suggests, it is used for loading an existing model.
- **Save**: As its name suggests, it is used for saving a model.
- **CreatePredictionEngine**: This method will let you make a prediction given a specific model input.

Data transformations

Some changes may have to be done to your data during processing—that's where data transformations come into play. Some of the methods and properties to do this are available as part of the **Transforms** property of ML context (**context**):

- **Categorical**: Applicable when working with data that can be categorized.
- **Conversion**: Applicable when converting from one data type to another.
- **Text**: When working with text columns, it is possible to transform those string values into their equivalent numerical values.
- **ReplaceMissingValues**: Replaces missing values within other data.
- **DropColumns**: Removes specific columns that might not be required.
- **Concatenate**: Concatenates multiple columns into one column.

Trainers

ML.NET comes with a set of built-in algorithms that are used for training your model based on input data for different scenarios—these algorithms are often referred to as trainers. ML.NET provides algorithms for scenarios such as clustering, regression, anomaly detection, ranking, multiclass or binary classification, etc.

Adding a Model

Having gone over the different types of operations that ML.NET can perform, and having created the console app, let's add a machine learning model to our application. To do that, within **Solution Explorer**, click on **TestML** to select the app, right-click, choose **Add**, and click on the **Machine Learning Model** option.

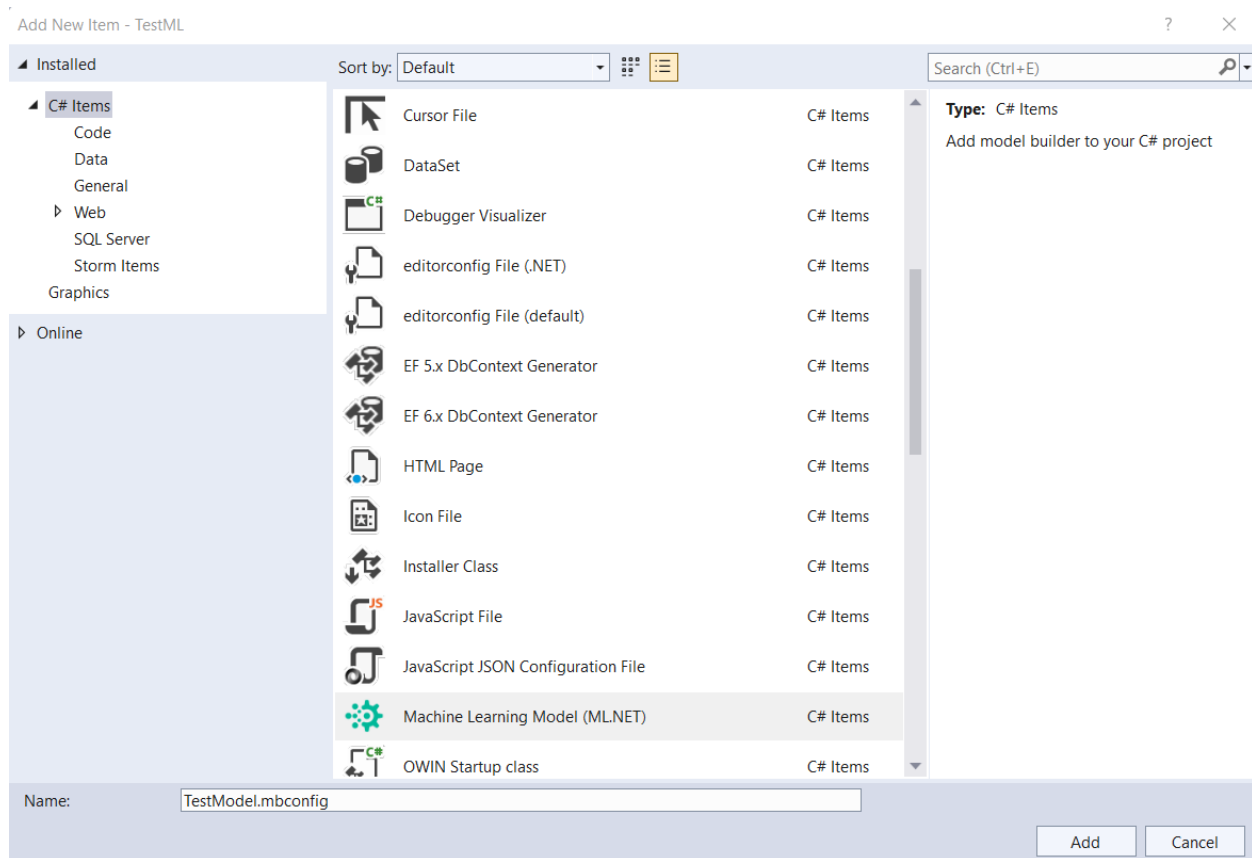


Figure 2-b: Adding a Machine Learning Model (ML.NET)

I'll give it the name, **TestModel**, and then click **Add**. Once that has been done, you'll notice that the TestModel.mbconfig file is added to the project, and a new UI is displayed—this UI is the Model Builder.

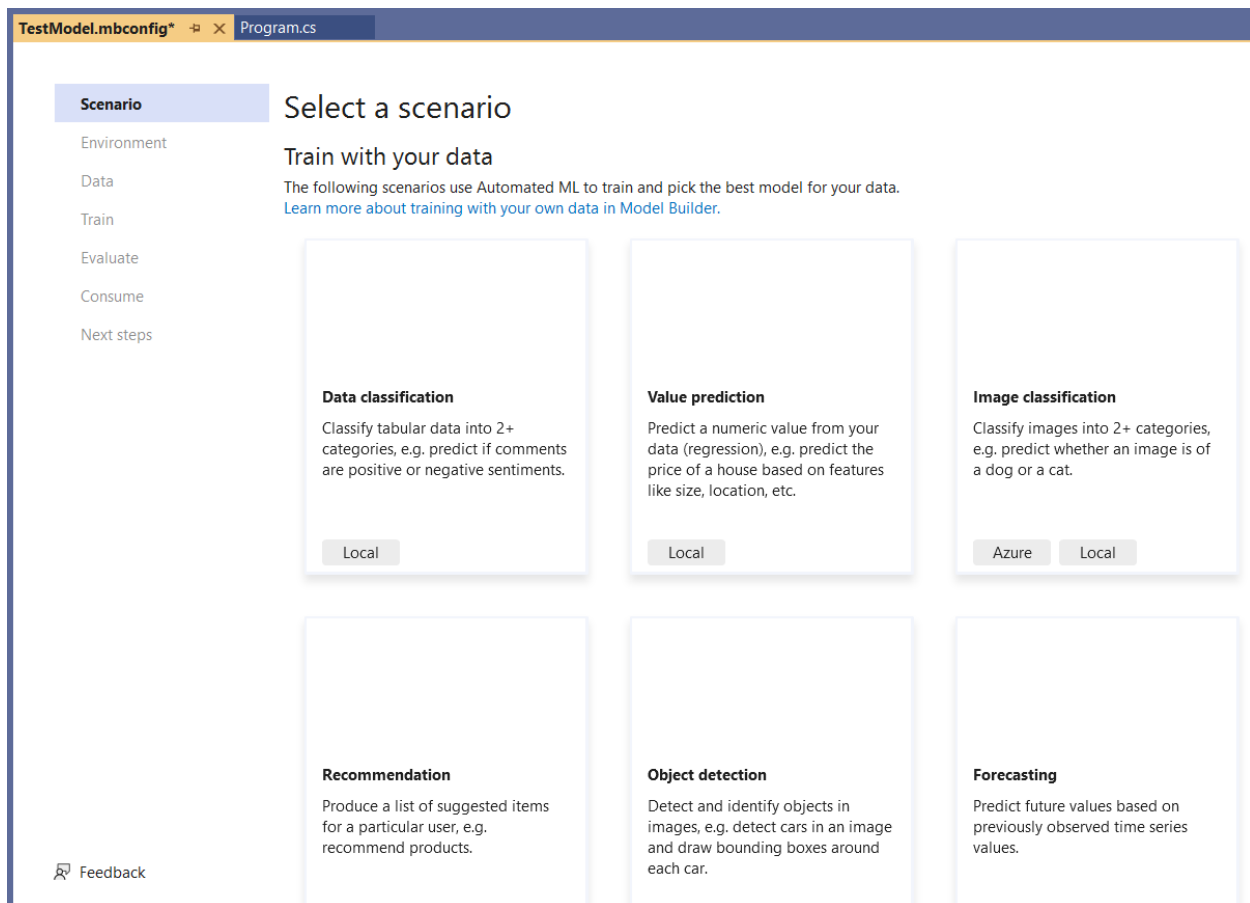




Figure 2-c: The Model Builder UI (Not Up to Date)

Notice that the Model Builder UI (previous screen) seems to be missing some details (those white areas above each scenario's title and description should not be empty).

 **Note:** If the previous issue doesn't happen to you, that's great! It means your Model Builder is up to date. In that case, feel free to skip the rest of this section and head over to the Scenarios and tasks section.

The preceding situation indicates that the Model Builder installed is not up to date (it's not the latest version). In that case, please download the most up-to-date version of the Model Builder from this [URL](#).

 **Note:** If you run into this situation, please make sure you download and install the latest version of Model Builder. Ensure you don't have Visual Studio running when installing the latest version of Model Builder.

Once you have downloaded the most recent version of Model Builder, execute the downloaded installer, and you should see the following screen.

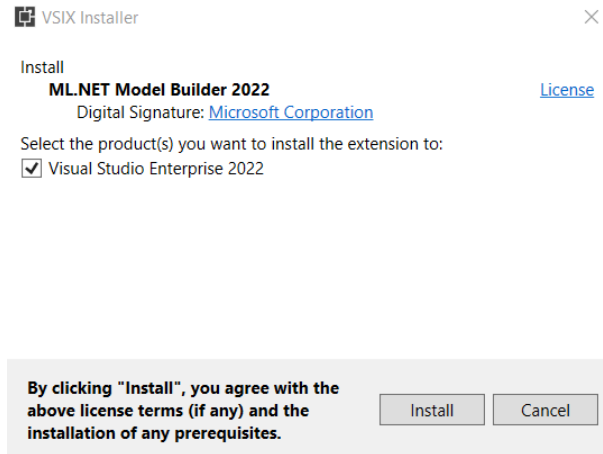


Figure 2-d: Model Builder Installer (Initial Screen)

To proceed with the installation, click **Install**. Once the installation has been finalized, you'll see the following screen.

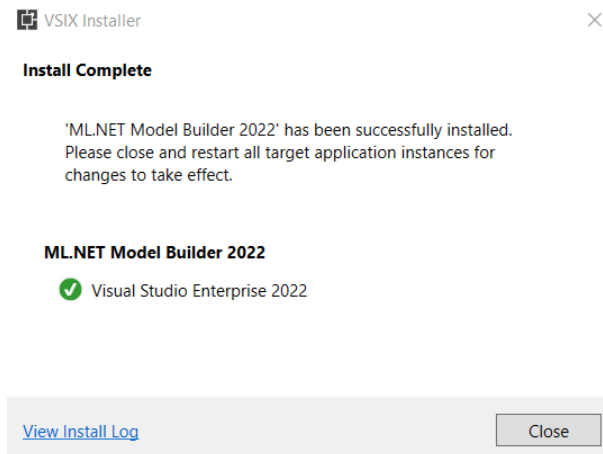


Figure 2-e: Model Builder Installer (Final Screen)

To quit the installer, click **Close**. With that done, rerun Visual Studio and open your project. Go to **Solution Explorer** and click on the **TestModel.mbconfig** file to open it.



Tip: Within Visual Studio, you can also update Model Builder by going to the **Extensions** menu, then clicking **Manage Extensions**.

Once done, the Model Builder UI should look as follows.

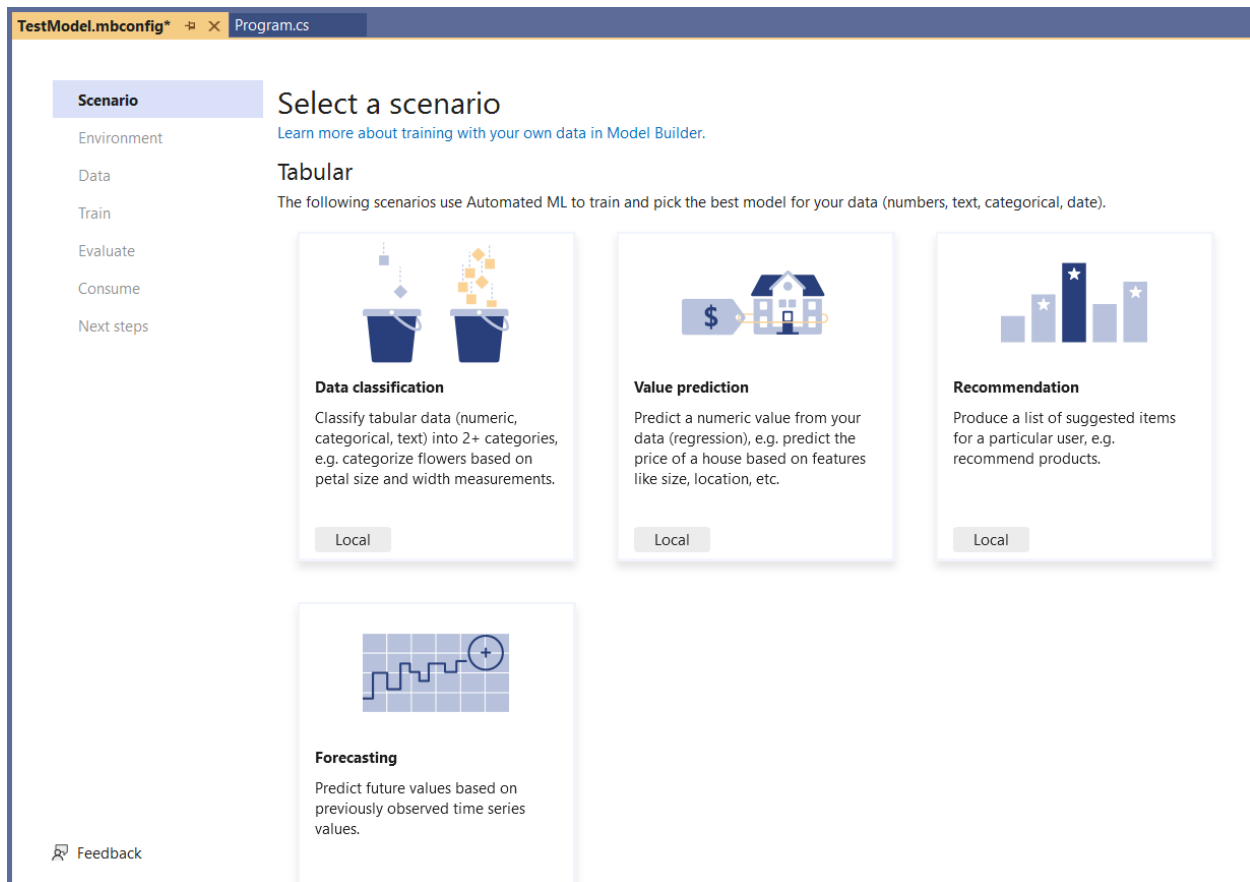


Figure 2-f: The Model Builder UI (Up to Date)

Great! As you can see, the Model Builder UI is now up to date, and the icons are displayed for each scenario.

Scenarios and tasks

Before we build the model and select the scenario we want to create, we need to understand how these different scenarios listed within the Model Builder UI correspond to machine learning tasks.

A scenario is how Model Builder describes the type of prediction that you want to make with your data, which correlates to a machine learning task. The task is the type of prediction based on the question being asked.

Think of the scenario as a wrapper around a task; the task specifies the prediction type and uses various trainers (algorithms) to train the model. The following diagram illustrates this.

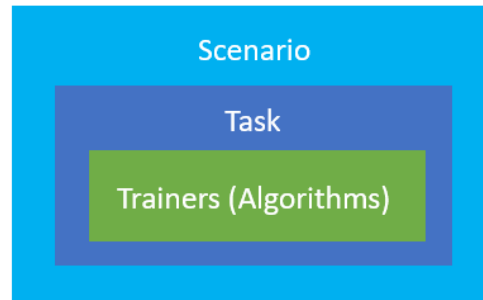


Figure 2-g: The Scenario-Task-Trainer(s) Hierarchy in ML.NET

For the model we will build shortly, we'll use this [dataset](#) to predict whether a text is a spam message (or not) using [binary classification](#). The idea is to build the model using Model Builder and then expand in greater detail on what happens behind the scenes.

Given that binary classification doesn't appear as an option within the Model Builder UI, we need to understand how the scenarios listed in the Model Builder UI relate to different machine learning tasks, such as binary classification.

To do that, let's have a look at the following diagram.

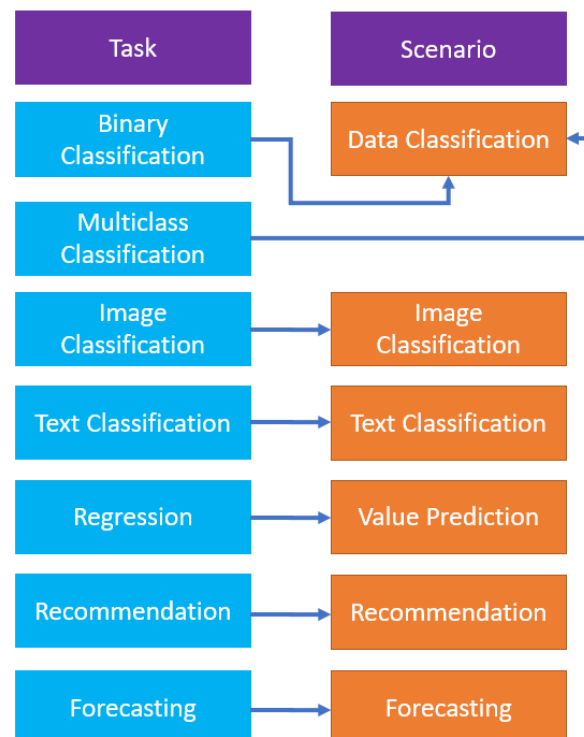


Figure 2-h: The Relationships between Machine Learning Tasks and Model Builder Scenarios

So, from the preceding figure, we can see that the binary classification task corresponds to the data classification scenario (as seen in the Model Builder UI).

Binary classification is used to understand if something is positive or negative, if an email is a spam message or not, or in general, whether a particular item has a specific trait or property (or not).

Training with Model Builder

Model Builder abstracts almost all the technical complexity of creating a model by hand (writing the code manually) and works similarly to a UI wizard.

To begin, let's choose the scenario to use. Because we will do binary classification to detect spam messages, we must select **Data classification**.

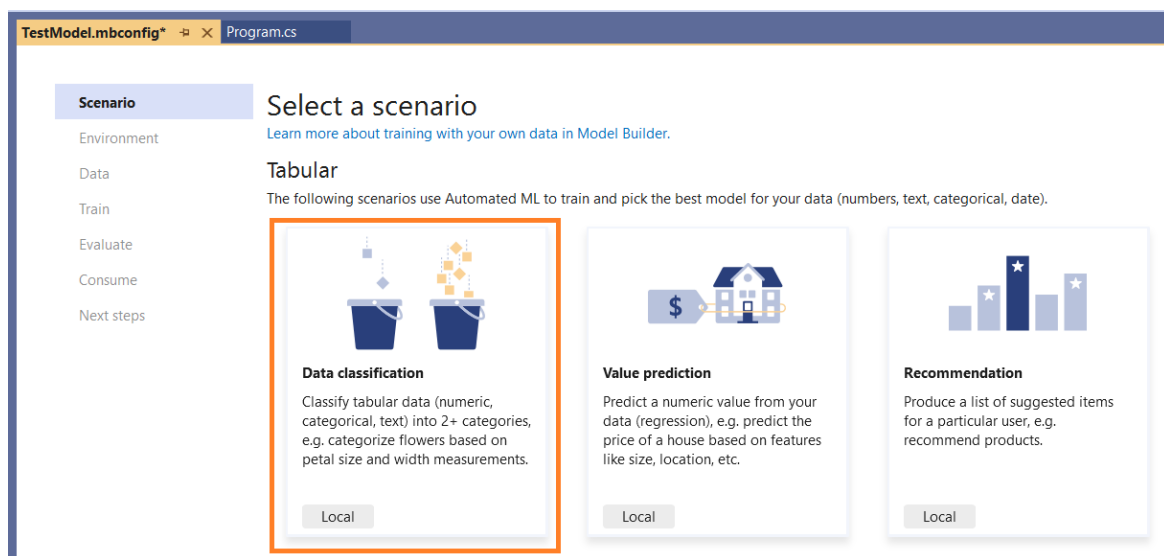


Figure 2-i: Model Builder—Select a Scenario (Data Classification)

To select the data classification scenario, click the **Local** button under the scenario's description. Once done, you'll be presented with the Select training environment screen, as follows.

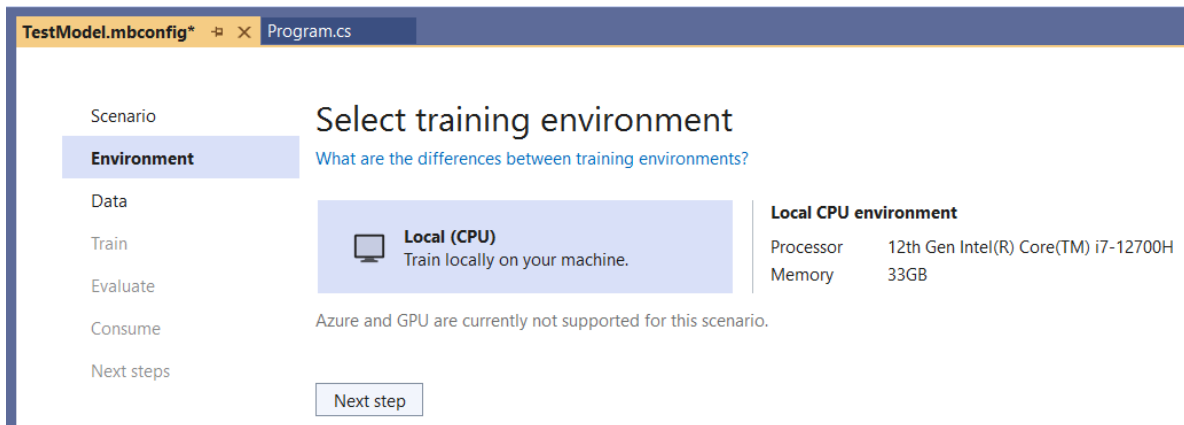


Figure 2-j: Model Builder—Select Training Environment

By default, Model Builder shows the details of the environment available where the data training and model creation will be done. Given that we previously clicked on **Local**, the local environment details (in this case, from my computer) are displayed. In your case, you'll see the details of your machine.

To continue, click on the **Next step** button, which will present you with the option to **Add data** to the model we are creating.

Before we continue, let's pause for a moment so that you can download the training data. The CSV file being used, [spam_assassin_tiny.csv](#), is a subset (the first 100 rows) of this [dataset](#).

Once you have downloaded it, you can import it by clicking **Browse**, as seen in the following screenshot.

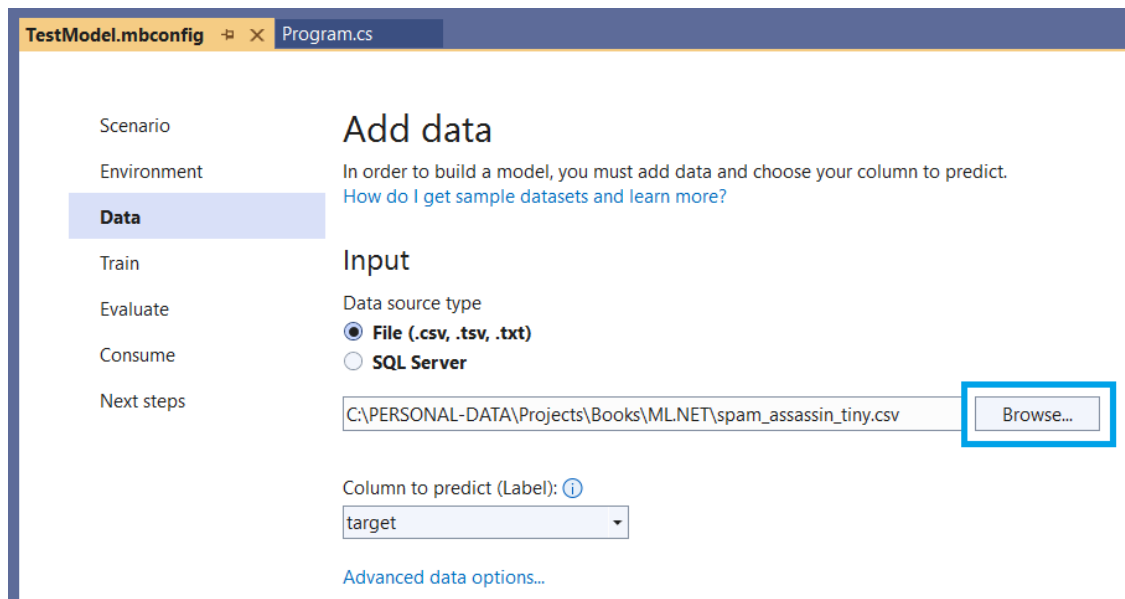


Figure 2-k: Model Builder—Add Data (Importing the Data)

With that done, the next step is to ensure that the **Column to predict (Label)** is set to **target** (which is the column that indicates whether the text is spam or not).

The column **target** of the [spam_assassin_tiny.csv](#) file has a value of 0 if the text is not spam and a value of 1 if the text is spam—which you’ll notice if you open the file with any text editor.

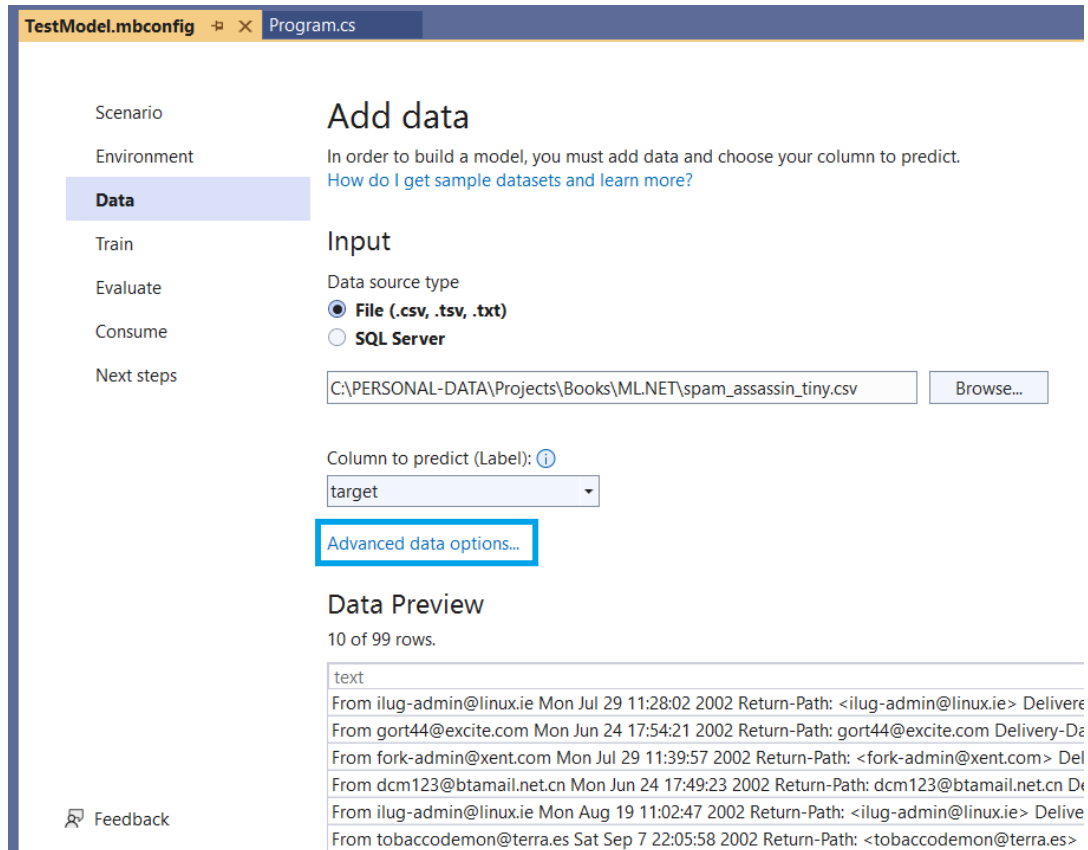


Figure 2-1: Model Builder—Add Data (Data Imported)

By setting the **target** as the **Column to predict (Label)**, we are saying that ML.NET will use this column to predict the value based on what is read from the **text** column of the dataset (as seen in the previous figure’s Data Preview section).

To fully understand this, let’s look at the advanced options by clicking **Advanced data options**.

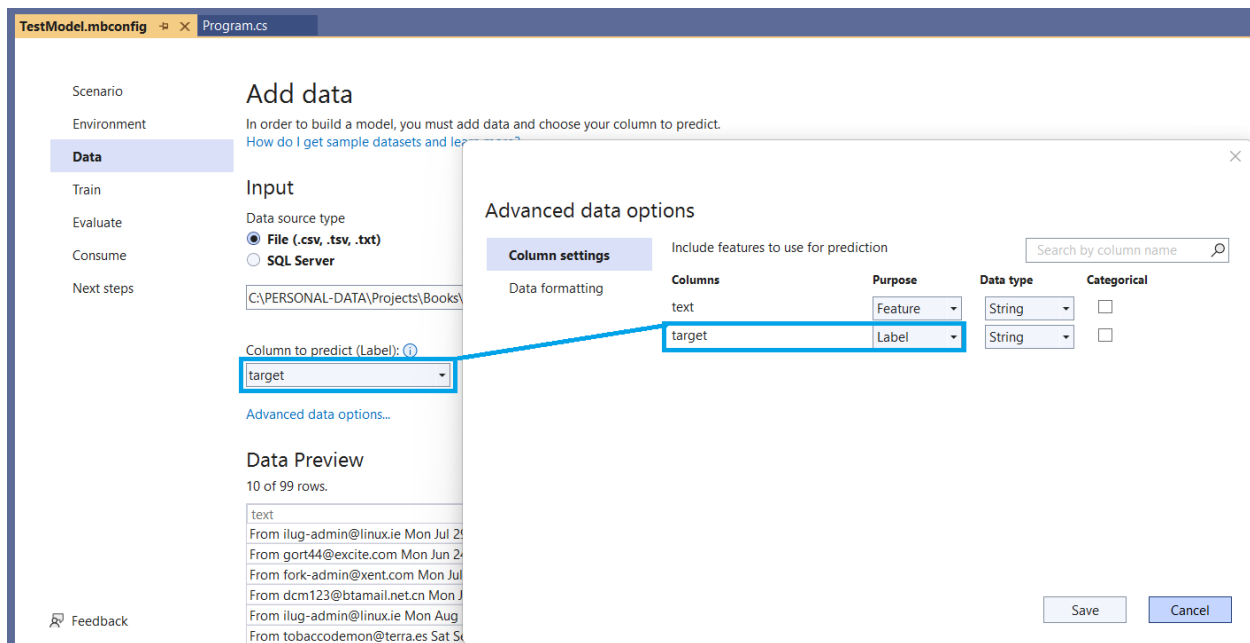


Figure 2-m: Model Builder—Advanced Data Options

Model Builder has identified that the dataset contains two columns, **text** and **target**, as seen within **Advanced data options, Column settings**. The **text** column trains the model—it contains the actual email messages. In contrast, the **target** column contains the value (1 or 0) that is the value to predict.

We can click **Cancel** to continue, as no changes have been made within the **Advanced data options** window.

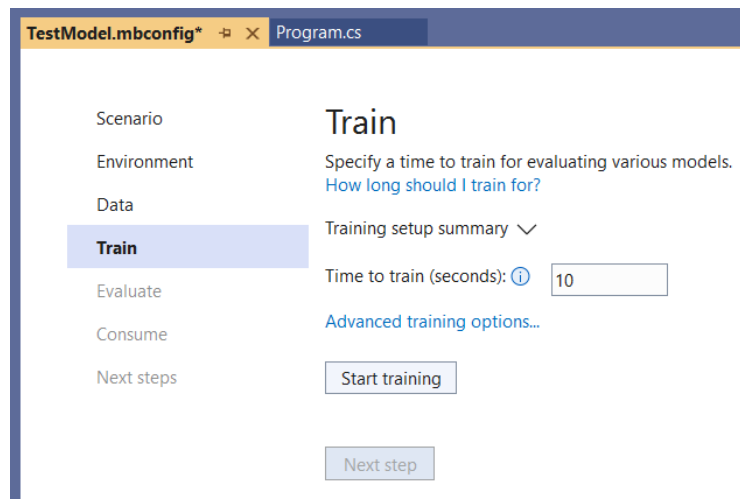


Figure 2-n: Model Builder—Train (Training Not Started)

Next, we can train the model with the dataset provided, and to do that, we need to click on **Start training**.

The time required to train the model is, in most cases, directly proportional to the size of the dataset. So, the larger the dataset, the more computing resources and time are required. Typically, time is available; however, computing resources are mostly limited to the specs of the environment used.

Instead of using the complete [dataset](#), which includes 5,329 rows, I created a tiny [subset](#) with only the first 100 rows (99, given that the first row is a header).

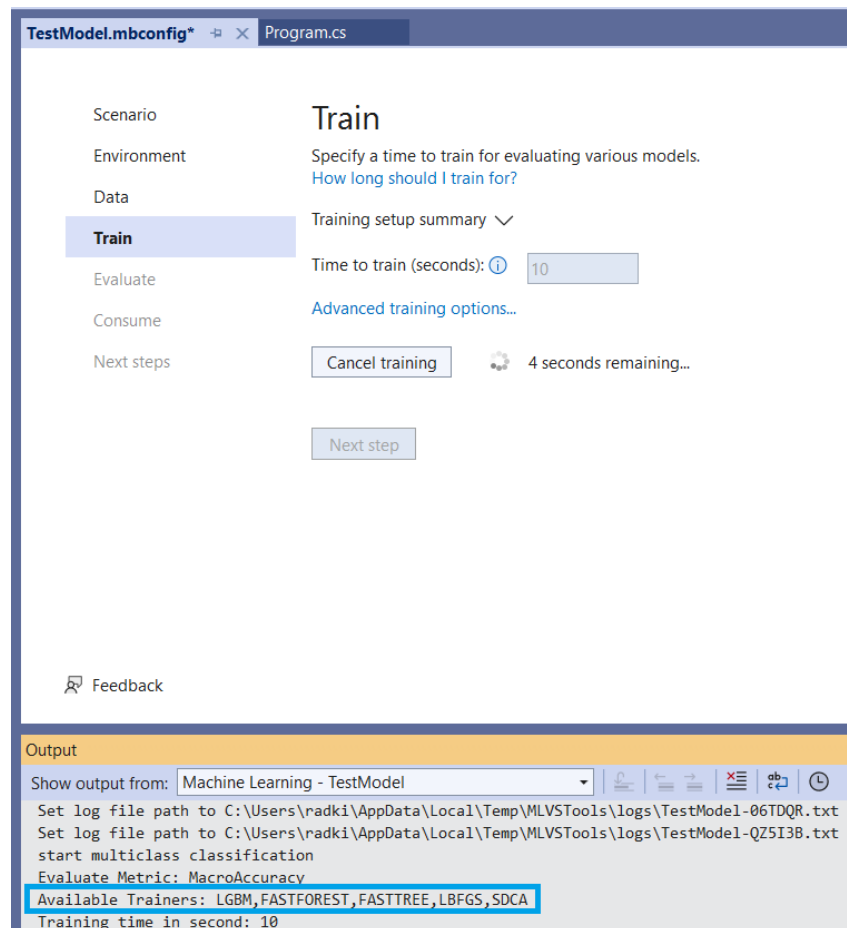


Figure 2-o: Model Builder—Train (Training Started)

While the training takes place, unless otherwise specified, the different trainers (algorithms) available for the machine learning task will be used—highlighted in the previous screen.

In this case, we are running a binary classification task, which employs the available trainers highlighted in the previous figure by default. If we would like to tweak the training properties and trainers to use, we can either wait for the training to finalize or cancel the training in execution.

Let's wait for the training to finish to look at the **Advanced training options** available.

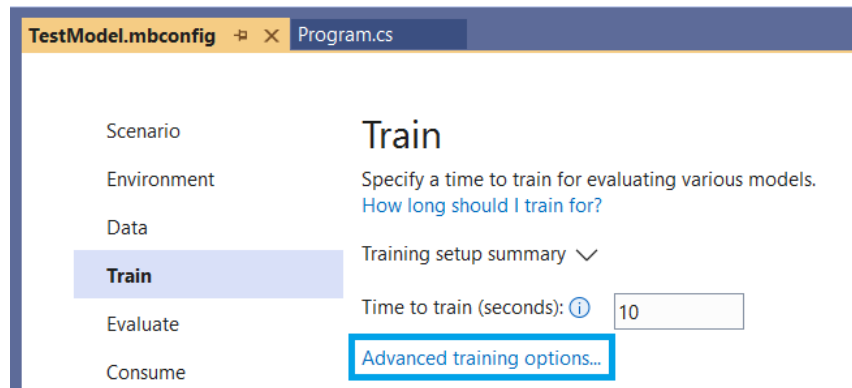


Figure 2-p: Model Builder—Train

After clicking **Advanced training options** and **Trainers**, you'll see the following screen.

Advanced training options

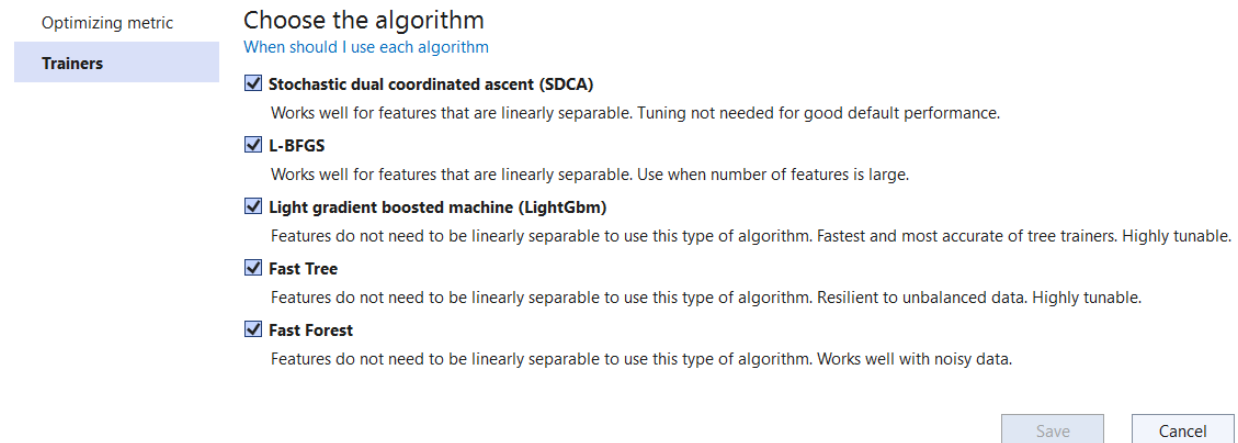


Figure 2-q: Model Builder—Advanced Training Options (Trainers)

Here we can see a list of the trainers that are available and used. By default, all the trainers are selected. It is also possible to use fewer trainers, which can be done by unchecking one or more.

ML.NET has good documentation that dives deeper into what these algorithms do and how to choose one by clicking on the **When should I use each algorithm** option.

We'll use all the trainers for our use case, so let's leave this as is and close the window by clicking **Cancel**.

At this stage, I wanted to show you that it is possible to change (enable or disable) some of the predefined algorithms for training a model in case the evaluation results are not optimal.

Evaluating with Model Builder

Now that we have trained the model, we can evaluate it. To do that, let's first click on the **Evaluate** option.

From the [spam assassin tiny.csv](#) file, copy one of the rows (without the target column value) and paste it into the text field above the Predict button. In my case, I copied row number six.

After clicking **Predict**, you'll see how the model predicts based on the text input. The prediction is correct given that in this case the text input is not spam, but a legitimate message. Thus, the result is **0** (not spam) with a value of 67% certainty. The percentage is not a true mathematical probability, sometimes called a pseudo-probability.

TestModel.mbconfig Program.cs

Scenario
Environment
Data
Train
Evaluate
Consume
Next steps

Evaluate

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

Best model:
MacroAccuracy: 1.0000
Model: FastForestOva

Try your model

Sample data
The following fields are pre-filled by a row of your data.

text
Linux systems by default, so from the tone of Padraigs mail that's no

Predict

Next step

Feedback

Results	
0	67%
1	33%

Figure 2-r: Model Builder—Evaluate

Feel free to try with other **text** input from the complete [dataset](#). Remember that the evaluation process is an opportunity to tweak and improve the model if the results are not as expected.

Because we have trained this model with a [tiny dataset](#), not using the entire [dataset](#), the percentages (confidence) of the results will not be as high (when correct) or low (when incorrect) as they would be if the model had been trained using the complete dataset. So, evaluate as many times as needed and feel free to retrain the model with a slightly larger dataset to improve the accuracy of the results.

The main reason I chose to explain all these Model Builder steps with a small dataset, not the large one, was to save you time while following along, as larger datasets require significantly more training time and resources from the system.

In your spare time, try training the model with the large dataset. Just make sure you have enough coffee available.

Consuming the model

With the model evaluation complete, the next step is to consume (use) the model created by Model Builder within our application.

As seen in the previous screenshot, Model Builder makes it very easy. There are two options: 1) either copy the **Code snippet** highlighted, or 2) use one of the available project templates, which can be added to your Visual Studio solution by clicking **Add to solution**.

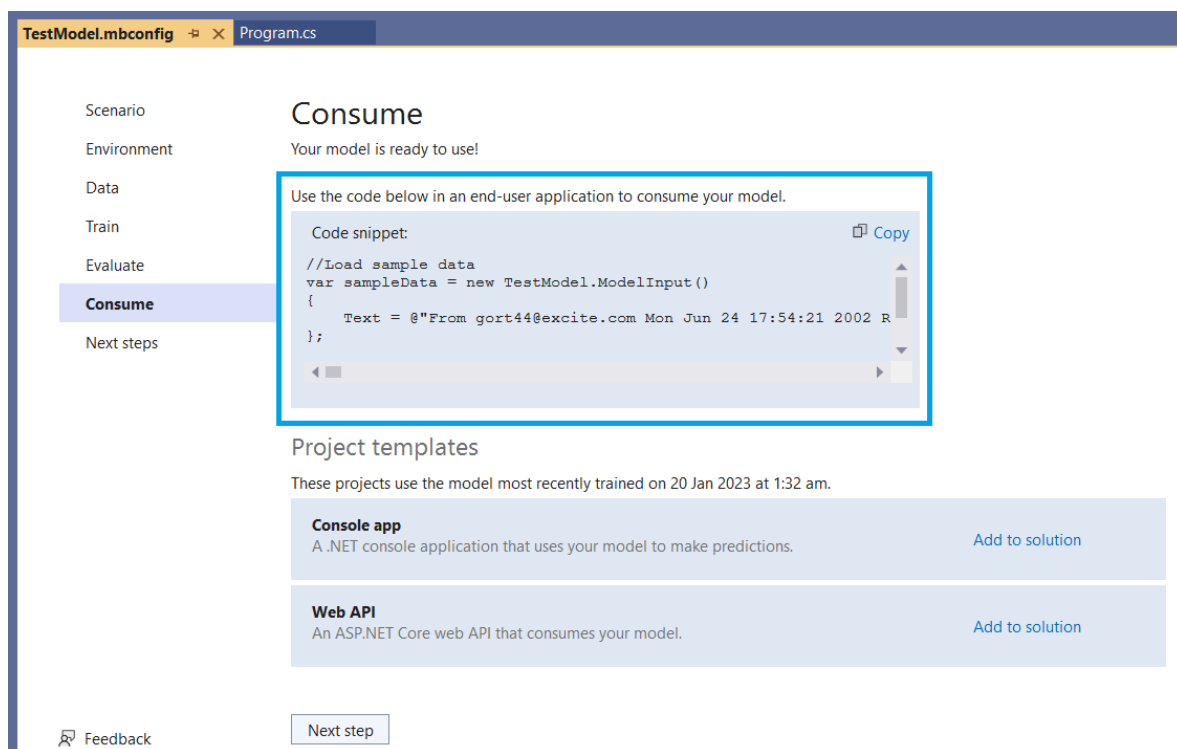



Figure 2-s: Model Builder—Consume

In my case, I'll go with option 1, which is to copy the code snippet and paste it into the **Main** method of **Program.cs**, as follows (code highlighted in **bold**).

 **Note:** The line highlighted with a **blue** background is a new line I added (not part of the code snippet) to see the prediction result when running the program.

 **Note:** The ellipsis markers (...) indicate extra text that I have explicitly removed to make the code more readable. The code snippet you'll copy from Model Builder will have the full text. Please refer to the [GitHub repo](#) to get the complete code of [Program.cs](#).



Note: The “`using TestML;`” statement is not part of the code snippet copied. It’s usually added (automatically) by Model Builder after the model is created, given that some additional C# files and a zip file are added to the project when the training finishes (we’ll explore these files later). If the “`using TestML;`” statement has not been added, please add it manually.

Code Listing 2-b: Program.cs–TestML (After copying the code snippet from Model Builder)

```
using Microsoft.ML;
using TestML;

internal class Program
{
    private static void Main(string[] args)
    {
        var context = new MLContext();

        Console.WriteLine("Hello, World!");

        //Load sample data.
        var sampleData = new TestModel.ModelInput()
        {
            Text = @"From gort44@excite.com...Mortgage Lenders & Brokers
Are Ready to compete for your business. Whether a new home loan is what
you seek or to refinance your current home loan at a lower interest rate,
we can help!...This service is fast and free. Free information request
form: PLEASE VISIT http://builtit4unow.com/pos...***",
        };

        //Load model and predict output.
        var result = TestModel.Predict(sampleData);

        Console.WriteLine($"Predicted: {result.PredictedLabel}");
    }
}
```



Note: It is plausible that the snippet you copied from Model Builder might have a different `Text` value than mine, which means your prediction result might differ.

From the preceding code, some things stand out if you look closely at it. The first is that the content of the `Text` property of the `sampleData` object is spam to the naked eye. The second is that the `sampleData` object is passed as a parameter to the `Predict` method of the `TestModel` class.

As its name implies, the `Predict` method executes a prediction for the input data provided (`sampleData`) for the model created.

Finally, the result of the prediction is displayed by invoking `result.PredictedLabel`. To see this in action, let's run the application. To do that, click on the run button in Visual Studio.

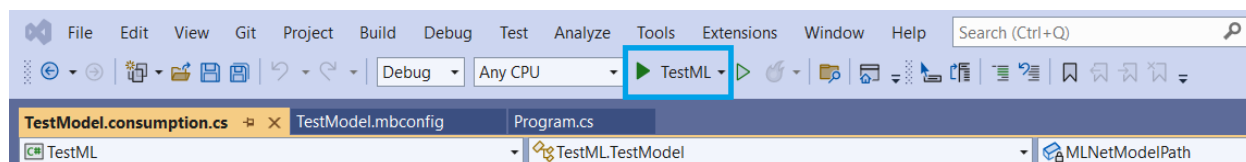


Figure 2-t: The Run Button Highlighted—Visual Studio

Once the application executes, you should see the following screen with the correct prediction value of 1 (indicating that the message is indeed spam).

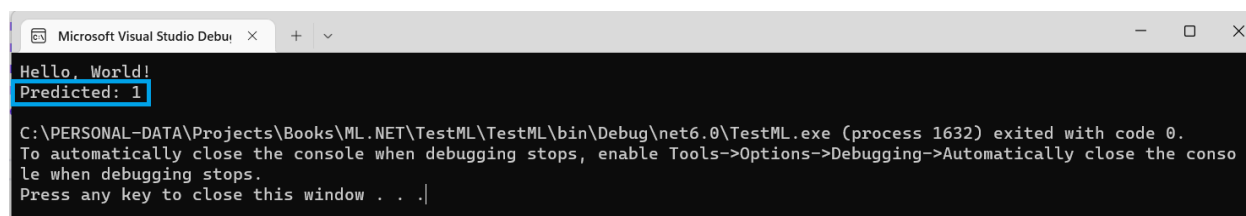



Figure 2-u: The Execution of the Application—Microsoft Visual Studio Debugger

Great! Model Builder has completely abstracted all the complexity of what ML.NET does behind the scenes to give us this prediction, providing us with a quick and easy implementation. The process is impressive, given that we've gone from nothing to having a spam-detection machine-learning model in just a few lines of code. Now, let's have a look behind the scenes.

Generated model (TestModel.consumption.cs)

We've gone through the various steps provided by Model Builder, and behind the scenes a model was created. Let's look at the code that was generated during that process.

 **Tip:** Considering that the generated code we will explore might seem a bit complex, I suggest you review the [high-level definitions](#) of some of the concepts and objects we will encounter to get acquainted with the terminology.

Within **Solution Explorer**, click on the arrow icon next to the **TestModel.mbconfig** file to expand its content.

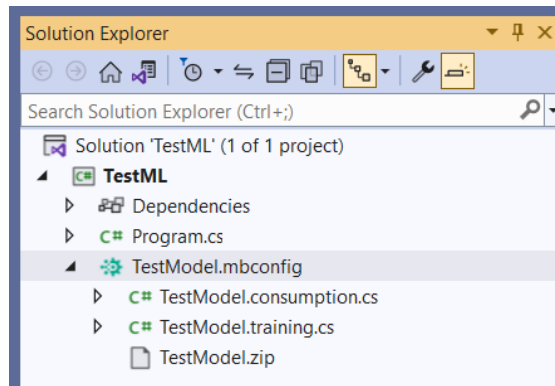



Figure 2-v: TestModel.mbconfig Expanded—Solution Explorer

Notice the three files that Model Builder created. First, let's open **TestModel.consumption.cs** and explore the code behind it.

 **Note:** In most cases, when you open *TestModel.consumption.cs*, you might see one or two lines of comments generated by Model Builder when the file was created. For the listing below, I have explicitly removed those lines.

Code Listing 2-c: TestModel.consumption.cs

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

namespace TestML
{
    public partial class TestModel
    {
        /// <summary>
        /// model input class for TestModel.
        /// </summary>
        #region model input class
        public class ModelInput
        {
            [LoadColumn(0)]
            [ColumnName(@"text")]
            public string Text { get; set; }

            [LoadColumn(1)]
            [ColumnName(@"target")]
            public string Target { get; set; }
        }

        #endregion

        /// <summary>
```

```

/// model output class for TestModel.
/// </summary>
#region model output class
public class ModelOutput
{
    [ColumnName(@"text")]
    public float[] Text { get; set; }

    [ColumnName(@"target")]
    public uint Target { get; set; }

    [ColumnName(@"Features")]
    public float[] Features { get; set; }

    [ColumnName(@"PredictedLabel")]
    public string PredictedLabel { get; set; }

    [ColumnName(@"Score")]
    public float[] Score { get; set; }
}

#endregion

private static string
    MLNetModelPath = Path.GetFullPath("TestModel.zip");

public static readonly
    Lazy<PredictionEngine<ModelInput, ModelOutput>>
    PredictEngine = new Lazy<PredictionEngine<ModelInput,
        ModelOutput>>(() => CreatePredictEngine(), true);

/// <summary>
/// Use this method to predict <see cref="ModelInput"/>.
/// </summary>
/// <param name="input">model input.</param>
/// <returns><seealso cref=" ModelOutput"/></returns>
public static ModelOutput Predict(ModelInput input)
{
    var predEngine = PredictEngine.Value;
    return predEngine.Predict(input);
}

private static PredictionEngine<ModelInput, ModelOutput>
CreatePredictEngine()
{
    var mlContext = new MLContext();
    ITransformer mlModel =
        mlContext.Model.Load(MLNetModelPath, out var _);

```



```

        return mlContext.Model.
            CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
    }
}
}

```

Let's review the code to understand it better. The ML.NET Model Builder automatically created this code.

First, we find the references to the two libraries that the code uses: **Microsoft.ML** and **Microsoft.ML.Data**.

The first (**Microsoft.ML**) includes ML.NET core methods, such as the trainers (algorithms), and the second (**Microsoft.ML.Data**) contains ML.NET methods that interact with the dataset used by the model.

Following that, within the **TestML** namespace, we find the **TestModel** class, declared as a **partial** class because it is partially declared (split) in both the **TestModel.consumption.cs** file and the **TestModel.training.cs** file.

Within the **TestModel** class, we find the **ModelInput** class, which defines as properties the two columns found within the dataset used: **Text** and **Target**.

Notice that both properties have decorators that map them to their respective columns within the dataset (**ColumnName**) and their position within that dataset (**LoadColumn**).

The **ModelInput** class, as its name implies, is used as the model's input. On the other hand, the **ModelOutput** class is used as the model's output.

Contrary to the **ModelInput** class, the properties of the **ModelOutput** class do not include the **LoadColumn** decorator, just **ColumnName**.

As part of the **ModelOutput** class, besides the **Text** and **Target** properties (which are also part of the **ModelInput** class), we find the **Features**, **PredictedLabel**, and **Score** properties.



Note: The **Target** property of the **ModelOutput** class is an unsigned integer (**uint**), contrary to the **Target** property of the **ModelInput** class, which is a **string**—this is because the **Target** property for a binary classification has to be an integer. Because the input dataset is in a CSV file, the value of **Target** as an input (even though it is a number) is a **string**. CSV files only store **string** values (including those that represent numbers).

These properties indicate the **Features**, the predicted value (**PredictedLabel**) column, and the confidence obtained for those results (**Score**).

Next, we find the declaration of the **MLNetModelPath** variable, which points to the model's metadata path (file name)—in our case **TestModel.zip**. This file includes the model's schema, training information, and transformer chain metadata.

Following that, we find the **PredictEngine** variable declared, which is nothing else than the variable that will hold the reference to the prediction engine ([PredictionEngine](#)) to make predictions on the trained model.

The prediction engine (**PredictEngine**) will contain a value of the following type:

Lazy<PredictionEngine<ModelInput, ModelOutput>>

The creation of the prediction engine (**PredictionEngine**) is deferred until it is first used, accomplished by using [Lazy](#) initialization.

Notice that the **ModelInput** and **ModelOutput** classes are the type parameters of the prediction engine (**PredictionEngine**).

The prediction engine (**PredictionEngine**) is instantiated by creating a new object of type **Lazy<PredictionEngine<ModelInput, ModelOutput>>** as in:

```
public static readonly Lazy<PredictionEngine<ModelInput, ModelOutput>>
    PredictEngine = new Lazy<PredictionEngine<ModelInput,
        ModelOutput>>(() => CreatePredictEngine(), true);
```

The constructor receives as a first parameter a lambda function that creates the engine, **() => CreatePredictEngine()**, and the second parameter (**true**) indicates whether the instance can be used by multiple threads (thread-safe).

Next, we have the **Predict** method, which is used for making predictions based on the model, as follows.

```
public static ModelOutput Predict(ModelInput input)
{
    var predEngine = PredictEngine.Value;
    return predEngine.Predict(input);
}
```

The **Predict** method receives as a parameter a **ModelInput** instance (**input**), which, as its name implies, is the **input** data for the model.

The predicted value (**PredictEngine.Value**) is assigned to the **predEngine** variable. The result of invoking the **predEngine.Predict** method (which receives as a parameter the **input** data) is returned by the **Predict** method.

Then, we find the **CreatePredictEngine** method, which, as its name implies, creates the prediction engine instance.

```
private static PredictionEngine<ModelInput, ModelOutput>
CreatePredictEngine()
{
    var mlContext = new MLContext();
    ITransformer mlModel = mlContext.Model.Load(MLNetModelPath, out var _);
    return mlContext.Model.
```

```

        CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
    }

```

The first thing the method does is create an instance of `MLContext`, which is assigned to the `mlContext` variable.

Then, the model is loaded—this is done by invoking the `Load` method from `mlContext.Model`, to which the model's metadata path is passed as a parameter (`MLNetModelPath`). The `out var _` parameter represents the `modelInputSchema`. The `Load` method returns the model loaded as an object (`mlModel`).

The prediction engine (`PredictEngine`) is finally created when the `CreatePredictionEngine` method from `mlContext.Model` is called, to which the object model (`mlModel`) is passed as a parameter.

So, overall, `TestModel.consumption.cs` contains the generated code responsible for creating the prediction engine and invoking it—thus, allowing the *consumption* of the model.

Generated model (TestModel.training.cs)

Next, let's go back to **Solution Explorer**, open `TestModel.training.cs`, and explore the code behind it, which was also automatically generated by Model Builder.



Tip: Gentle reminder. Given that the generated code we will explore could seem a bit complex, I suggest you review the [high-level definition](#) of some of the concepts and objects we will encounter to get acquainted with the terminology.



Note: In most cases, when you open `TestModel.training.cs`, you might see one or two lines of comments generated by Model Builder when the file was created. For the listing below, I have explicitly removed those lines.

Code Listing 2-d: `TestModel.training.cs`

```

using Microsoft.ML.Trainers.FastTree;
using Microsoft.ML;

namespace TestML
{
    public partial class TestModel
    {
        /// <summary>
        /// Retrains model using the pipeline generated as part of the
        /// training process. For more information on how to load data,
        /// see aka.ms/loaddata.
        /// </summary>
        /// <param name="mlContext"></param>

```

```

/// <param name="trainData"></param>
/// <returns></returns>
public static ITransformer RetrainPipeline(MLContext mlContext,
    IDataView trainData)
{
    var pipeline = BuildPipeline(mlContext);
    var model = pipeline.Fit(trainData);

    return model;
}

/// <summary>
/// build the pipeline that is used from model builder. Use this
/// function to retrain model.
/// </summary>
/// <param name="mlContext"></param>
/// <returns></returns>
public static IEstimator<ITransformer> BuildPipeline(
    MLContext mlContext)
{
    // Data process configuration with pipeline data
    // transformations.
    var pipeline = mlContext.Transforms.Text.FeaturizeText
        (inputColumnName:@"text",outputColumnName:@"text")
        .Append(mlContext.Transforms.
            Concatenate(@"Features", new []{@"text"}))
        )
        .Append(mlContext.Transforms.
            Conversion.MapValueToKey(
                outputColumnName:@"target",inputColumnName:@"target")
            )
        .Append(mlContext.MulticlassClassification.
            Trainers.OneVersusAll(binaryEstimator:mlContext.
                BinaryClassification.Trainers.FastTree(new
                    FastTreeBinaryTrainer.Options()
                    {
                        NumberOfLeaves=4,
                        MinimumExampleCountPerLeaf=20,
                        NumberOfTrees=4,
                        MaximumBinCountPerFeature=254,
                        FeatureFraction=1,
                        LearningRate=0.1,
                        LabelColumnName:@"target",
                        FeatureColumnName:@"Features"
                    }
                ),
                labelColumnName: @"target"
            )
        )
}

```

```

        .Append(mlContext.Transforms.Conversion.
            MapKeyToValue(
                outputColumnName:@"PredictedLabel",
                inputColumnName:@"PredictedLabel"
            )
        );

        return pipeline;
    }
}

```

The code begins with the **using** statements, where the ML.NET core (**Microsoft.ML**) and **Microsoft.ML.Trainers.FastTree** libraries are imported. The **Microsoft.ML.Trainers.FastTree** library contains the algorithm implementation used by the model.

Next, we find the **TestModel** class, also used within **TestModel.consumption.cs**, given that it is a **partial** class. The first method we come across is **RetrainPipeline**. As the comment indicates, this method is responsible for retraining the model once the pipeline has been built. Let's inspect its code.

```

public static ITransformer RetrainPipeline(MLContext mlContext, IDataView
trainData)
{
    var pipeline = BuildPipeline(mlContext);
    var model = pipeline.Fit(trainData);
    return model;
}

```

So, as we can see, this method receives two parameters, an instance of **MLContext** and the training data (**trainData**) of type **IDataView** (used as the input and output of transforms).

The **RetrainPipeline** method returns an object that implements the **ITransformer** interface (responsible for transforming data within an ML.NET model pipeline).

The **RetrainPipeline** method works by building the model's pipeline, in which the different transforms and algorithm(s) that will be used are specified.

With the objects in the pipeline created, the data (**trainData**) is passed to train the model by invoking the **pipeline.Fit** method. Finally, the **model** is returned.

So, the most complex part of the code is what happens within the **BuildPipeline** method. Let's dissect it into smaller chunks to make sense of what is happening.

The **BuildPipeline** method receives as a parameter an **MLContext** instance and returns an object that implements the [IEstimator<ITransformer>](#) interface. The pipeline is built through a series of transformations that get subsequently added using **mlContext.Transforms**. The code begins with the call to:

```
Text.FeaturizeText(inputColumnName:@"text",outputColumnName:@"text")
```

The method transforms the input column strings (text) into numerical feature vectors (integers) that keep normalized counts of words and character [n-grams](#).

Then, a series of **Append** methods are chained to **FeaturizeText**. For every **Append** method, a transform operation or trainer is passed as a parameter, creating the ML.NET pipeline. The first **Append** looks as follows:

```
.Append(mlContext.Transforms.Concatenate(@"Features", new []{@"text"}))
```

What this does is invoke the **Concatenate** method from **mlContext.Transforms** to concatenate the various input columns into a new output column, in this case, **Features**.

The next **Append** looks as follows:

```
.Append(mlContext.Transforms.Conversion.MapValueToKey(  
    outputColumnName:@"target",inputColumnName:@"target")
```

The [MapValueToKey](#) method from **mlContext.Transforms.Conversion** maps the input column (**inputColumnName**) to the output columns (**outputColumnName**) to convert [categorical values](#) into keys.

Moving on, the next **Append** looks as follows. Note this is where the magic happens.

```
.Append(mlContext.MulticlassClassification.  
    Trainers.OneVersusAll(binaryEstimator:mlContext.  
        BinaryClassification.Trainers.FastTree(new  
            FastTreeBinaryTrainer.Options()  
            {  
                NumberOfLeaves=4,  
                MinimumExampleCountPerLeaf=20,  
                NumberOfTrees=4,  
                MaximumBinCountPerFeature=254,  
                FeatureFraction=1,  
                LearningRate=0.1,  
                LabelColumnName=@"target",  
                FeatureColumnName=@"Features"  
            }  
        ),  
        labelColumnName: @"target"  
    )  
)
```

The **OneVersusAll** method of **mlContext.MulticlassClassification.Trainers** receives a binary estimator (**binaryEstimator**) algorithm instance as a parameter. The one-versus-all technique is a general machine language algorithm that adapts a binary classification algorithm to handle a multiclass classification problem. See “[Transformation to binary](#).”

The binary estimator instance represents the machine learning binary classification task employed by ML.NET that contains the trainers, utilities, and options used by the [FastTree](#) algorithms used for making predictions on the model.

Those [options](#) are then passed to the **FastTree** algorithms (highlighted in blue), predicting a target using a decision tree for binary classification.



Note: I’ve included a link to the official documentation for each option property, so you can look at what each one does and the allowed values for each.

The final part (**labelColumnName: @"target"**) indicates that all predictions done by **FastTree** will be set on the column with the **target** label.

In short, **TestModel.training.cs** describes how the machine learning pipeline for the model works and behaves by specifying the various types of transformers and algorithms used and their sequence.

Summary

Throughout this chapter, we have employed Model Builder to generate a model for our data. Model Builder does a fantastic job of abstracting all the complexity behind writing the logic required to train, test, and consume a model.

As you have seen, the ML.NET classes are challenging to grasp, especially if you don’t have a machine learning background. If we didn’t have Model Builder to do the heavy lifting, we would have to experiment with many different values (for the option properties) and try different combinations or sequences to get the pipeline right. All in all, it would be a daunting and time-consuming activity that would make ML.NET no different from any other machine learning framework—thus, requiring a deeper understanding of machine learning.

By going over the code this way, I have tried to explain the underlying complexity of the Model Builder-generated code in relatively simple terms, so you can get a feeling of what happens behind the scenes, keeping a balance between simplicity and complexity.

For what remains of this book, I’ll go over other ML.NET scenarios and employ Model Builder to generate code for us. We’ll dive into the generated code for those new scenarios and compare how those look to the data classification scenario (binary classification) we just explored to understand differences and commonalities.

Chapter 3 Value Prediction

Quick intro

In the previous chapter, we explored the core aspects of ML.NET. We used Model Builder to create a data classification scenario using binary classification to perform spam detection using a small dataset. By doing that, we could create a model, consume and test it, and explore the behind-the-scenes code generated by Model Builder to train and build the machine learning pipeline.

In this chapter, we'll follow a similar process, but we will work with a **value prediction** Model Builder scenario this time. We'll also create the model, consume and test it, and explain the generated code. We'll then be able to compare both scenarios' differences and how the code differs. So, with that said, let's begin.

Dataset explanation

The [dataset](#) we'll use throughout this chapter is a slightly modified version of the [Boston House Prices](#) dataset. The Boston House prices dataset aims to predict the median price (in 1972) of a house in one of 506 towns or villages near Boston. Each of the 506 data items has 12 predictor variables—11 numeric and 1 Boolean.

If you download and look at the [original dataset](#), you'll notice that although the file has the **.csv** file extension, the values are not comma-separated but are instead separated by spaces. You'll also see the original file doesn't have headers (it doesn't include the column names, even though the columns are described on the webpage).

I edited the original dataset by replacing the white spaces with commas and adding headers (column names) on the first row. The edited dataset, which you can find in my [GitHub repository](#), is the one we will use. So, please download it.

Dataset columns

Let's review the dataset's columns and what each one represents, so you can understand what the values mean when you look at the dataset:

- **CRIM:** Indicates the crime rate by town per capita.
- **ZN:** Reflects the proportion of residential land for lots over 25,000 sq. ft.
- **INDUS:** Specifies the proportion of nonretail business acres per town.
- **CHAS:** Is the Charles River variable (1 if tract bounds river; 0 otherwise).
- **NOX:** Indicates the nitric oxide concentration (parts per ten million).
- **RM:** Indicates the average number of rooms per dwelling.
- **AGE:** Indicates the proportion of owner-occupied units built before 1940.
- **DIS:** Specifies the distances to five employment centers in Boston.

- **RAD**: Indicates the accessibility to highways.
- **TAX**: Indicates the full-value property-tax rate per \$10,000.
- **PTRATIO**: Indicates the pupil-teacher ratio by town.
- **MEDV**: Represents the median value of owner-occupied homes in thousands of dollars.

The goal is to use the **Value Prediction** scenario using the modified dataset to predict the value of **MEDV** based on the values represented by the other columns.

Creating the model

So, with Visual Studio open, let's create a new console application. Click the **File** menu, click on **New**, and then on **Project**.

Select the **Console App** project option and then click **Next**. You'll then be shown the following screen, where you can specify the project's name. I've renamed the project **ValuePredict** (which I suggest you use) and set the project's **Location**.

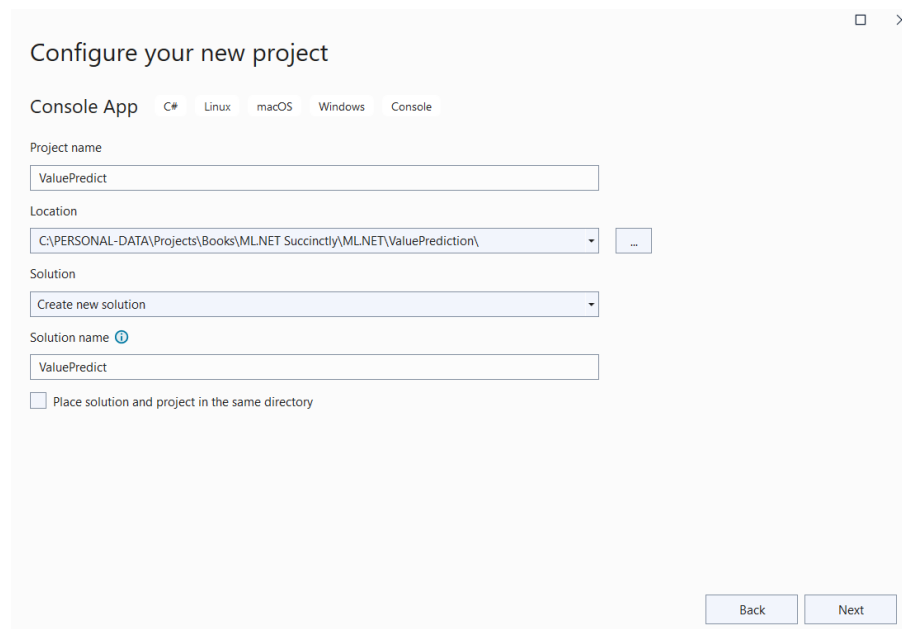


Figure 3-a: The “Configure your new project” Screen (Visual Studio)

To continue to the following screen, click **Next**, and as for the Framework option, any of the options available is fine—I’ll be using **.NET 6.0 (Long Term Support)**. Once done, click **Create**.

At this point, the project has been created and the next step is to add the model. To do that, within the **Solution Explorer**, click on **ValuePredict** to select the app, then right-click, choose **Add**, and then click **Machine Learning Model**.

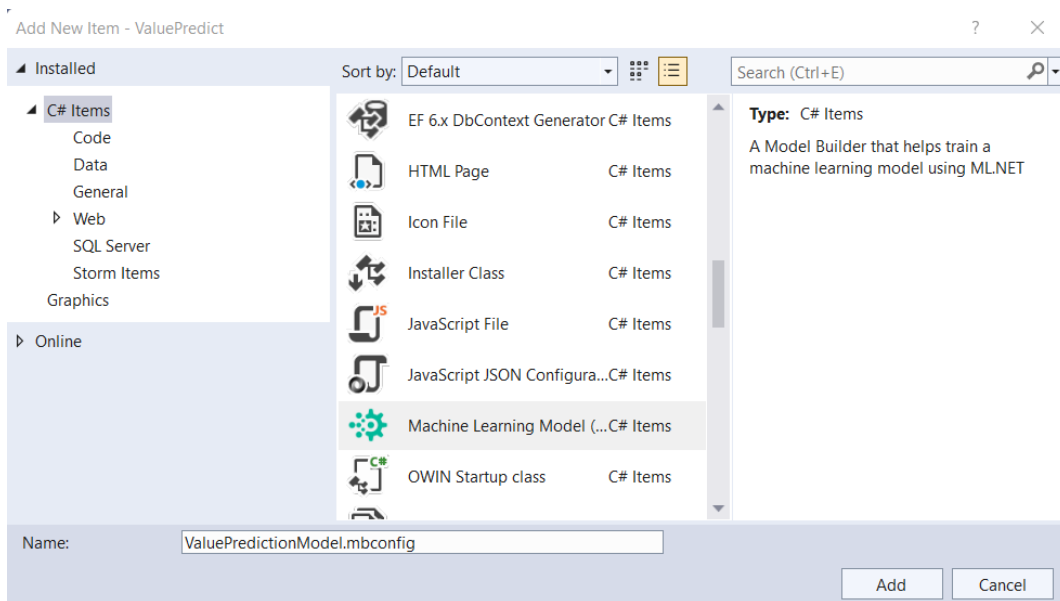


Figure 3-b: Adding a Machine Learning Model (ML.NET)

I'll give it the name **ValuePredictionModel** and then click **Add**. Once done, you'll notice that the **ValuePredictionModel.mbconfig** file has been added to the Visual Studio project, and the Model Builder UI appears.

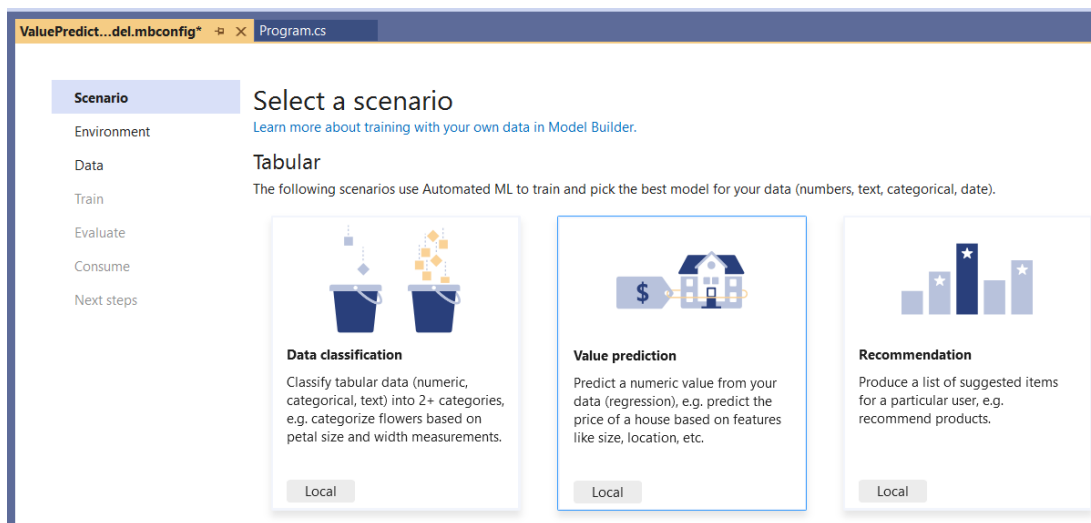


Figure 3-c: Model Builder—Select a scenario (Value prediction)

Let's continue by choosing the **Value prediction** scenario. To do that, click on the **Local** button below the **Value prediction** description.

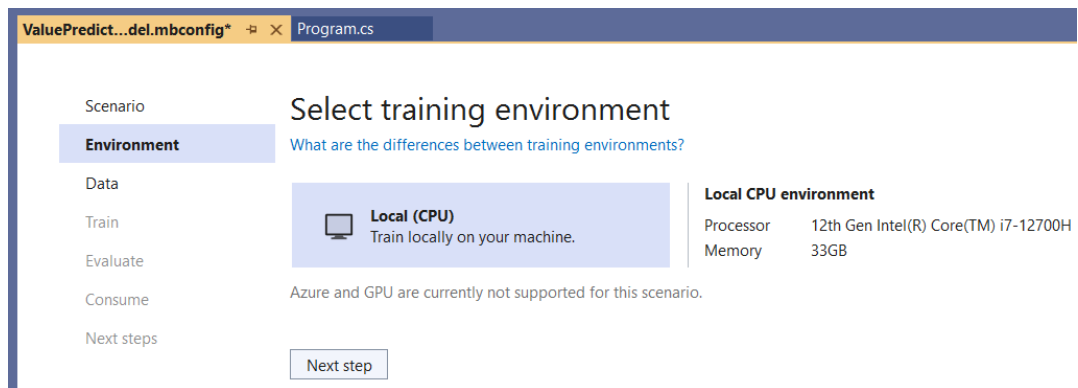


Figure 3-d: Model Builder—Select training environment

To continue, all we have to do is click **Next step**. At this stage, we will add the modified [dataset](#). We can do that by clicking **Browse**, selecting the **housing.csv** file, and then clicking **Open**.

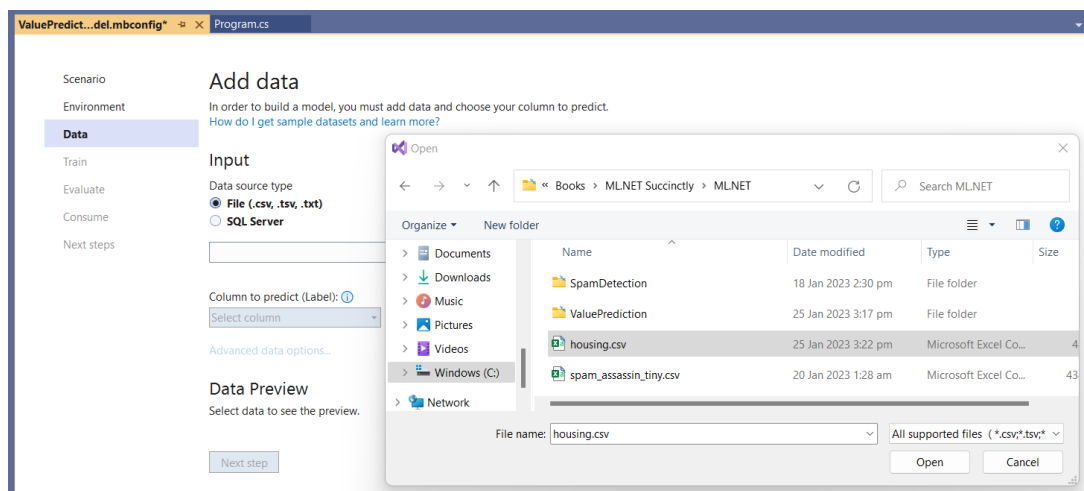


Figure 3-e: Model Builder—Add data (Selecting housing.csv)

Once that has been done, the data will be imported and visible within the Data Preview table.

ValuePrediction...del.training.cs ValuePredicti...consumption.cs **ValuePredict...del.mbconfig*** Program.cs

Scenario
Environment
Data
Train
Evaluate
Consume
Next steps

Add data

In order to build a model, you must add data and choose your column to predict.
[How do I get sample datasets and learn more?](#)

Input

Data source type

☒ **File (.csv, .tsv, .txt)**
☐ **SQL Server**

C:\PERSONAL-DATA\Projects\Books\ML.NET Succinctly\ML.NET\housing.csv [Browse...](#)

Column to predict (Label): ⓘ

MEDV

[Advanced data options...](#)

Data Preview

10 of 506 rows.

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSAT	MEDV
0.00632	18.00	2.310	0	0.5380	6.5750	65.20	4.0900	1	296.0	15.30	396.90	4.98	24.00
0.02731	0.00	7.070	0	0.4690	6.4210	78.90	4.9671	2	242.0	17.80	396.90	9.14	21.60
0.02729	0.00	7.070	0	0.4690	7.1850	61.10	4.9671	2	242.0	17.80	392.83	4.03	34.70
0.03237	0.00	2.180	0	0.4580	6.9980	45.80	6.0622	3	222.0	18.70	394.63	2.94	33.40
0.06905	0.00	2.180	0	0.4580	7.1470	54.20	6.0622	3	222.0	18.70	396.90	5.33	36.20
0.02985	0.00	2.180	0	0.4580	6.4300	58.70	6.0622	3	222.0	18.70	394.12	5.21	28.70
0.08829	12.50	7.870	0	0.5240	6.0120	66.60	5.5605	5	311.0	15.20	395.60	12.43	22.90
0.14455	12.50	7.870	0	0.5240	6.1720	96.10	5.9505	5	311.0	15.20	396.90	19.15	27.10
0.21124	12.50	7.870	0	0.5240	5.6310	100.00	6.0821	5	311.0	15.20	386.63	29.93	16.50
0.17004	12.50	7.870	0	0.5240	6.0040	85.90	6.5921	5	311.0	15.20	386.71	17.10	18.90

[Feedback](#) [Next step](#)

Figure 3-f: Model Builder—Add data (Data Added)

From the **Column to predict (Label)** dropdown, select the **MEDV** column, which indicates the house prices.

With that done, click **Next step** to continue with the training step. To train the model, click **Start training**. Once the model has been trained, you should see an overview of the Training results.

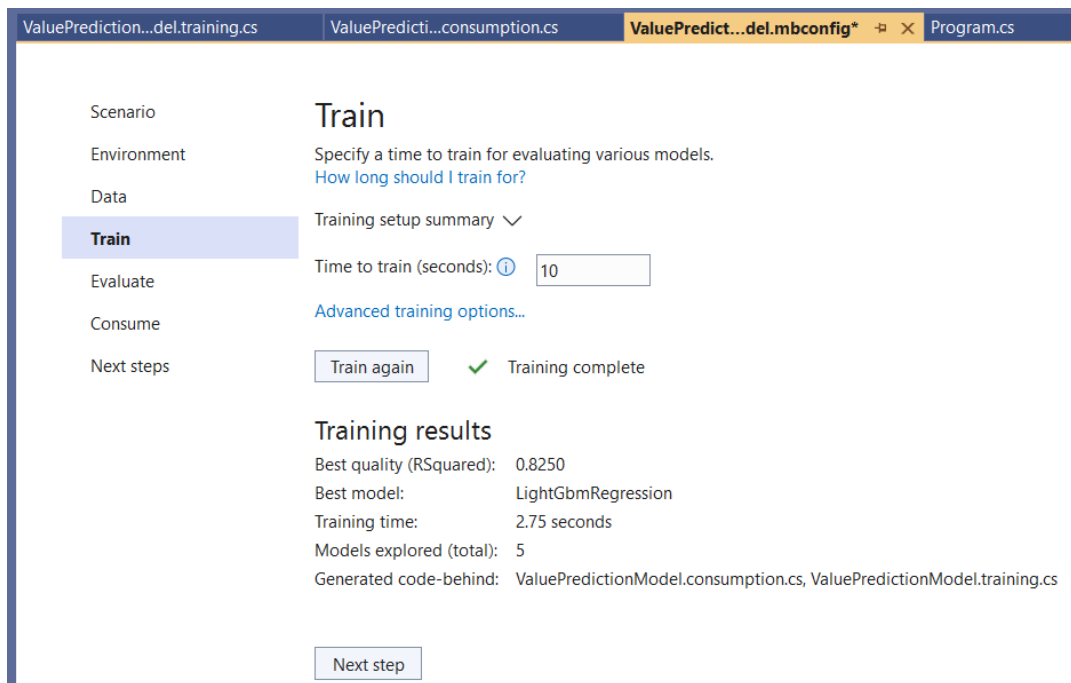


Figure 3-g: Model Builder—Train (Training results)

Notice that the algorithm chosen by Model Builder as the best option for the training data used by this model is [LightGbmRegression](#), based on [decision trees](#).

Next, you can evaluate the model. To do that, click **Next step**; the Evaluate step will be shown.

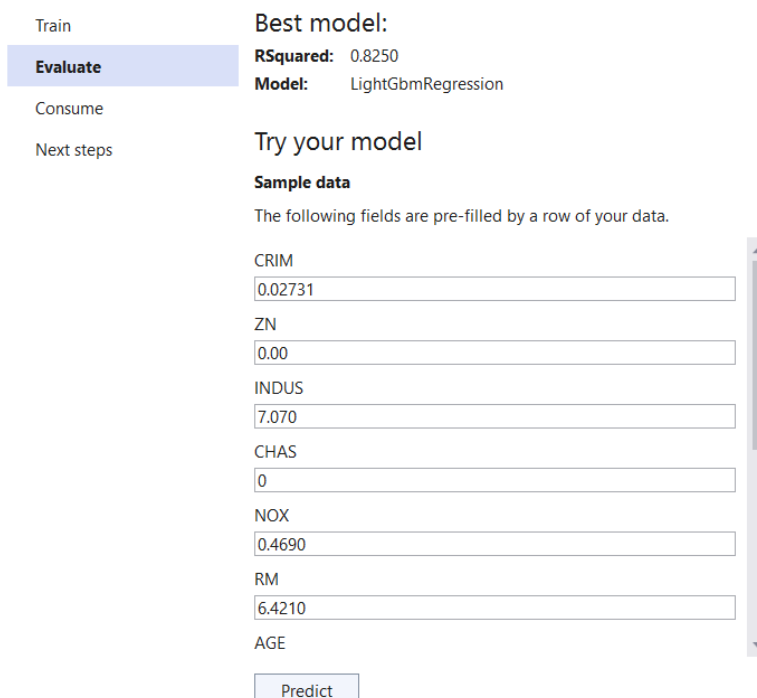


Figure 3-h: Model Builder—Evaluate (Try your model)

Given that we are using a dataset with multiple columns, we are presented with all the input columns available, which we can use to test our newly created model. The test input values are prefilled with data inferred by Model Builder. To view all the test values, scroll down the list. When ready, click **Predict**—this will display the prediction results.

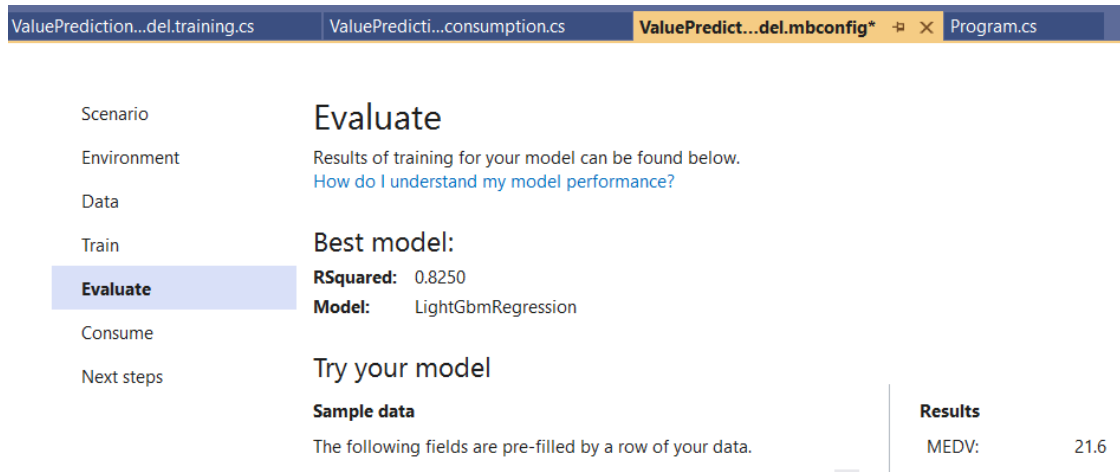


Figure 3-i: Model Builder—Evaluate (Try your model with Results)

Once the prediction value is available, we can continue by clicking on **Next step** to get ready to consume the model.

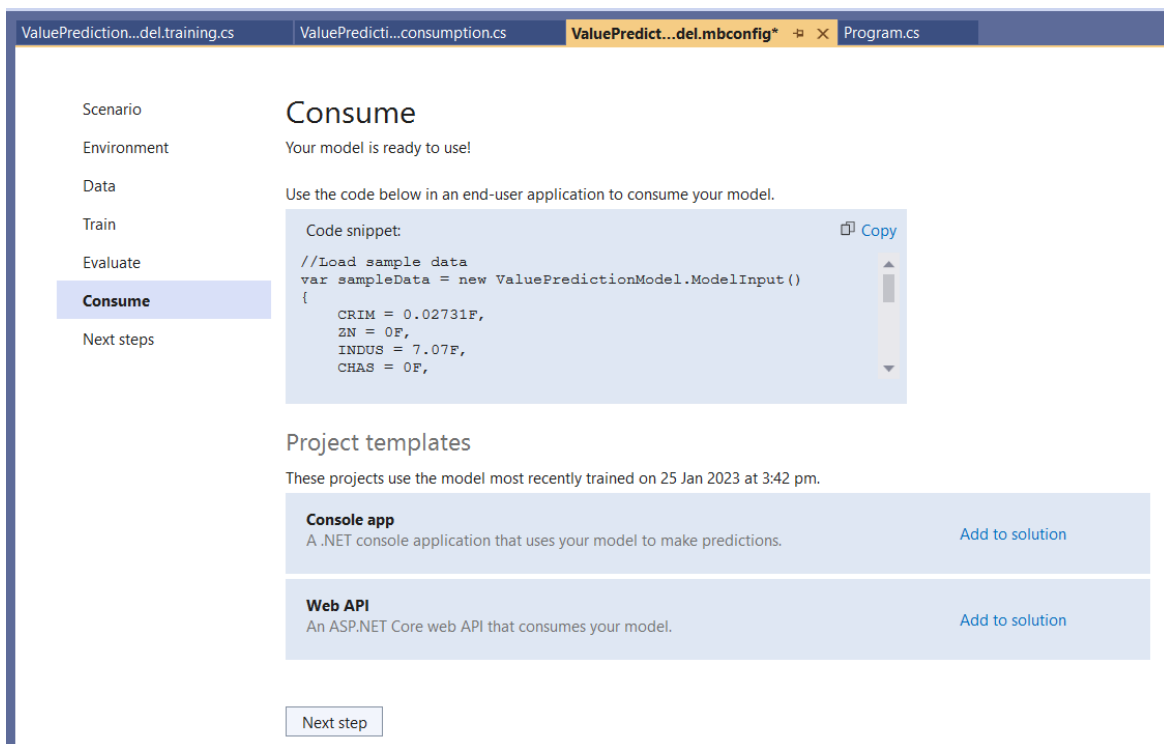


Figure 3-j: Model Builder—Consume

With this scenario, we will do the same thing we did with the spam detection example we previously created: copy the code snippet and consume it directly from the project we already started. But before we copy the code snippet generated by Model Builder, let's go to **Solution Explorer** and open **Program.cs**. The code looks as follows.

Code Listing 3-a: Program.cs

```
internal class Program
{
    private static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Let's replace the preceding code with the following code that uses the code snippet generated by Model Builder and consumes the model. The bold line is not part of the generated code snippet. I've added it manually to see the model's prediction result (**result.Score**), which indicates the predicted house price.

Code Listing 3-b: Program.cs (Modified)

```
using ValuePredict;

internal class Program
{
    private static void Main(string[] args)
    {
        //Load sample data.
        var sampleData = new ValuePredictionModel.ModelInput()
        {
            CRIM = 0.02731F,
            ZN = 0F,
            INDUS = 7.07F,
            CHAS = 0F,
            NOX = 0.469F,
            RM = 6.421F,
            AGE = 78.9F,
            DIS = 4.9671F,
            RAD = 2F,
            TAX = 242F,
            PTRATIO = 17.8F,
            B = 396.9F,
            LSAT = 9.14F,
        };

        //Load model and predict output.
        var result = ValuePredictionModel.Predict(sampleData);
    }
}
```

```

        Console.WriteLine(result.Score.ToString());
    }
}

```

We should see the following output if we run the program by clicking the run button within Visual Studio.

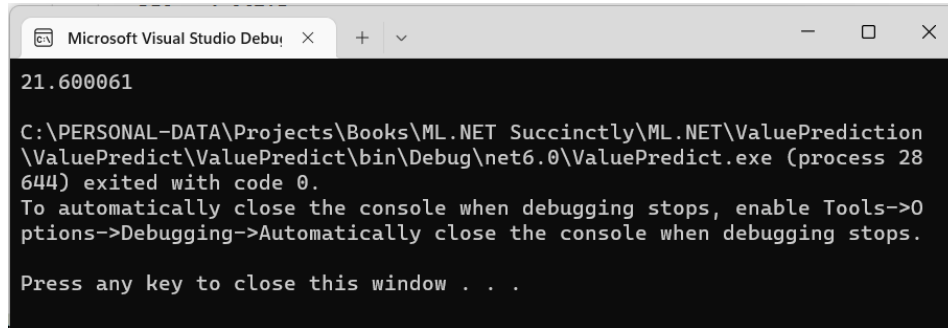


Figure 3-k: The Program Running—Microsoft Visual Studio Debugger

Notice that the output value (21.6) is the same one Model Builder displayed when evaluating the model. Now that we can consume the model correctly, let's switch to **Solution Explorer** and look at the files that Model Builder generated and added to the project.

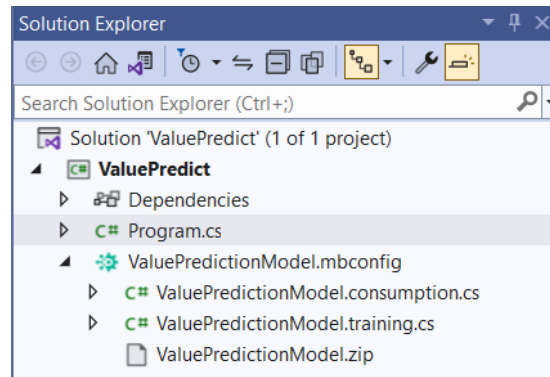


Figure 3-l: The ValuePredictionModel Files Generated by Model Builder—Solution Explorer

As you can see, Model Builder generated three files, ValuePredictionModel.consumption.cs, ValuePredictionModel.training.cs, and ValuePredictionModel.zip.

ValuePredictionModel.consumption.cs

Let's open this file within Solution Explorer to look at the generated code in detail.

Code Listing 3-c: ValuePredictionModel.consumption.cs

```

using Microsoft.ML;

```



```

using Microsoft.ML.Data;

namespace ValuePredict
{
    public partial class ValuePredictionModel
    {
        /// <summary>
        /// model input class for ValuePredictionModel.
        /// </summary>
        #region model input class
        public class ModelInput
        {
            [LoadColumn(0)]
            [ColumnName(@"CRIM")]
            public float CRIM { get; set; }

            [LoadColumn(1)]
            [ColumnName(@"ZN")]
            public float ZN { get; set; }

            [LoadColumn(2)]
            [ColumnName(@"INDUS")]
            public float INDUS { get; set; }

            [LoadColumn(3)]
            [ColumnName(@"CHAS")]
            public float CHAS { get; set; }

            [LoadColumn(4)]
            [ColumnName(@"NOX")]
            public float NOX { get; set; }

            [LoadColumn(5)]
            [ColumnName(@"RM")]
            public float RM { get; set; }

            [LoadColumn(6)]
            [ColumnName(@"AGE")]
            public float AGE { get; set; }

            [LoadColumn(7)]
            [ColumnName(@"DIS")]
            public float DIS { get; set; }

            [LoadColumn(8)]
            [ColumnName(@"RAD")]
            public float RAD { get; set; }

            [LoadColumn(9)]

```

```

    [ColumnName(@"TAX")]
    public float TAX { get; set; }

    [LoadColumn(10)]
    [ColumnName(@"PTRATIO")]
    public float PTRATIO { get; set; }

    [LoadColumn(11)]
    [ColumnName(@"B")]
    public float B { get; set; }

    [LoadColumn(12)]
    [ColumnName(@"LSAT")]
    public float LSAT { get; set; }

    [LoadColumn(13)]
    [ColumnName(@"MEDV")]
    public float MEDV { get; set; }

}

#endregion

/// <summary>
/// model output class for ValuePredictionModel.
/// </summary>
#region model output class
public class ModelOutput
{
    [ColumnName(@"CRIM")]
    public float CRIM { get; set; }

    [ColumnName(@"ZN")]
    public float ZN { get; set; }

    [ColumnName(@"INDUS")]
    public float INDUS { get; set; }

    [ColumnName(@"CHAS")]
    public float CHAS { get; set; }

    [ColumnName(@"NOX")]
    public float NOX { get; set; }

    [ColumnName(@"RM")]
    public float RM { get; set; }

    [ColumnName(@"AGE")]
    public float AGE { get; set; }
}

```

```

        [ColumnName(@"DIS")]
        public float DIS { get; set; }

        [ColumnName(@"RAD")]
        public float RAD { get; set; }

        [ColumnName(@"TAX")]
        public float TAX { get; set; }

        [ColumnName(@"PTRATIO")]
        public float PTRATIO { get; set; }

        [ColumnName(@"B")]
        public float B { get; set; }

        [ColumnName(@"LSAT")]
        public float LSAT { get; set; }

        [ColumnName(@"MEDV")]
        public float MEDV { get; set; }

        [ColumnName(@"Features")]
        public float[] Features { get; set; }

        [ColumnName(@"Score")]
        public float Score { get; set; }
    }

    #endregion

    private static string MLNetModelPath =
        Path.GetFullPath("ValuePredictionModel.zip");

    public static readonly
        Lazy<PredictionEngine<ModelInput, ModelOutput>> PredictEngine =
        new Lazy<PredictionEngine<ModelInput, ModelOutput>>(() =>
            CreatePredictEngine(), true);

    /// <summary>
    /// Use this method to predict <see cref="ModelInput"/>.
    /// </summary>
    /// <param name="input">model input.</param>
    /// <returns><seealso cref=" ModelOutput"/></returns>
    public static ModelOutput Predict(ModelInput input)
    {
        var predEngine = PredictEngine.Value;
        return predEngine.Predict(input);
    }

```

```

    }

    private static PredictionEngine<ModelInput, ModelOutput>
    CreatePredictEngine()
    {
        var mlContext = new MLContext();
        ITransformer mlModel = mlContext.Model.
                                Load(MLNetModelPath, out var _);
        return mlContext.Model.
               CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
    }
}

```

The code begins with the **using** statements that reference the libraries imported and used, **Microsoft.ML** and **Microsoft.ML.Data**.

Following that, we find the **ValuePredictionModel** class, declared as **partial**, given that it resides in both **ValuePredictionModel.*.cs** files. Within the class, we find the **ModelInput** and **ModelOutput** classes. The **ModelInput** class contains the values used as input by the model, and the **ModelOutput** class contains the output values. The **ModelInput** and **ModelOutput** classes contain properties corresponding to the dataset columns we previously explored.

However, the main difference between both classes is that the **ModelOutput** class has the **Features** and **Score** properties that are specific to the model's prediction results.

Then, we find **MLNetModelPath**, which points to the location where the model's metadata is found.

Next is the **PredictEngine** variable, an instance of **Lazy<PredictionEngine<ModelInput, ModelOutput>>**.

To create the instance, the constructor receives as a first parameter a lambda function that instantiates the engine: **() => CreatePredictEngine()**, and the second parameter (**true**) indicates whether the instance can be used by multiple threads (thread-safe).

Next, we have the **Predict** method, which is used for making predictions based on the model, as follows. The **Predict** method receives as a parameter a **ModelInput** instance (**input**), which represents the **input** data for the model. The predicted value (**PredictEngine.Value**) is assigned to the **predEngine** variable. The result of invoking the **predEngine.Predict** method is returned by the **Predict** method.

Then, we find the **CreatePredictEngine** method, which, as its name implies, creates the prediction engine instance.

```

private static PredictionEngine<ModelInput, ModelOutput>
CreatePredictEngine()
{
    var mlContext = new MLContext();

```

```

    ITransformer mlModel = mlContext.Model.Load(MLNetModelPath, out var _);
    return mlContext.Model.
        CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
}

```

This method first creates an instance of **MLContext**, which is assigned to the **mlContext** variable.

Following that, the model is loaded, which is done by invoking the **Load** method from **mlContext.Model**, to which the model's metadata path (**MLNetModelPath**) is passed as a parameter.

The **out var _** parameter represents the **modelInputSchema**. The **Load** method returns the model loaded as an object (**mlModel**).

The prediction engine (**PredictEngine**) is created using the **CreatePredictionEngine** method from **mlContext.Model**, to which the object model (**mlModel**) is passed as a parameter.

Overall, **ValuePredictionModel.consumption.cs** contains the generated code for creating the prediction engine and invoking it—thus, allowing the model consumption.

As you have seen, the pattern used by Model Builder to programmatically generate the code for **ValuePredictionModel.consumption.cs** is almost identical to the code generated for the spam detection use case we previously explored (**TestModel.consumption.cs**). The only difference is the properties of the **ModelInput** and **ModelOutput** classes, which differ between both use cases.

With **ValuePredictionModel.consumption.cs**, the predicted result is assigned to the **Score** property of the **ModelOutput** class, contrary to **TestModel.consumption.cs**, where a target column (label) was used.

ValuePredictionModel.training.cs

Let's move on to the Solution Explorer and open the **ValuePredictionModel.training.cs** file.

Code Listing 3-d: ValuePredictionModel.training.cs

```

using Microsoft.ML.Trainers.LightGbm;
using Microsoft.ML;

namespace ValuePredict
{
    public partial class ValuePredictionModel
    {
        /// <summary>
        /// Retrains model using the pipeline generated as part of the
        /// training process.
        /// For more information on how to load data, see
        /// aka.ms/loaddata.
    }
}

```

```

/// </summary>
/// <param name="mlContext"></param>
/// <param name="trainData"></param>
/// <returns></returns>
public static ITransformer RetrainPipeline(MLContext mlContext,
    IDataView trainData)
{
    var pipeline = BuildPipeline(mlContext);
    var model = pipeline.Fit(trainData);

    return model;
}

/// <summary>
/// Build the pipeline that is used from Model Builder. Use this
/// function to retrain model.
/// </summary>
/// <param name="mlContext"></param>
/// <returns></returns>
public static IEstimator<ITransformer> BuildPipeline(
    MLContext mlContext)
{
    // Data process configuration with pipeline data
    // transformations.
    var pipeline = mlContext.Transforms.ReplaceMissingValues(
        new [] {
            new InputOutputColumnPair(@"CRIM", @"CRIM"),
            new InputOutputColumnPair(@"ZN", @"ZN"),
            new InputOutputColumnPair(@"INDUS", @"INDUS"),
            new InputOutputColumnPair(@"CHAS", @"CHAS"),
            new InputOutputColumnPair(@"NOX", @"NOX"),
            new InputOutputColumnPair(@"RM", @"RM"),
            new InputOutputColumnPair(@"AGE", @"AGE"),
            new InputOutputColumnPair(@"DIS", @"DIS"),
            new InputOutputColumnPair(@"RAD", @"RAD"),
            new InputOutputColumnPair(@"TAX", @"TAX"),
            new InputOutputColumnPair(@"PTRATIO", @"PTRATIO"),
            new InputOutputColumnPair(@"B", @"B"),
            new InputOutputColumnPair(@"LSAT", @"LSAT")
        }
    )

    .Append(mlContext.Transforms.
        Concatenate(@"Features",
            new [] {
                @"CRIM",@"ZN",@"INDUS",@"CHAS",@"NOX",@"RM",
                @"AGE",@"DIS",@"RAD",@"TAX",
                @"PTRATIO",@"B",@"LSAT"
            }
        )
    )
}

```

```

    )
    )

    .Append(mlContext.Regression.Trainers.
        LightGbm(
            new LightGbmRegressionTrainer.Options()
            {
                NumberOfLeaves=4,
                NumberOfIterations=2632,
                MinimumExampleCountPerLeaf=20,
                LearningRate=0.599531696349256,
                LabelColumnName=@"MEDV",
                FeatureColumnName=@"Features",
                ExampleWeightColumnName=null,
                Booster=new GradientBooster.Options()
                {
                    SubsampleFraction=0.999999776672986,
                    FeatureFraction=0.878434202775917,
                    L1Regularization=2.91194636064836E-10,
                    L2Regularization=0.999999776672986
                },
                MaximumBinCountPerFeature=246
            }
        )
    );

    return pipeline;
}
}
}

```

Let's explore each part of the code individually to understand what is happening. First, we find the using statements—the **Microsoft.ML.Trainers.LightGbm** namespace is referenced besides **Microsoft.ML**. The **Microsoft.ML.Trainers.LightGbm** namespace contains the methods used by ML.NET to implement the [LightGbm](#) algorithm.

Then, within the **partial ValuePredictionModel** class, we find the **RetrainPipeline** method. The **RetrainPipeline** method is responsible for retraining the model once the pipeline has been built. This method has two parameters, an instance of [MLContext](#) and the training data (**trainData**) of type [IDataView](#) (used as the input and output of transforms).

The **RetrainPipeline** method returns an object that implements the [ITransformer](#) interface responsible for transforming data within an ML.NET model pipeline. It builds the model's pipeline using different transforms and algorithms.

The call to the **pipeline.Fit** method retrains the model (using **trainData**), returning the retrained model.

On the other hand, the **BuildPipeline** method receives as a parameter an **MLContext** instance and returns an object that implements the [IEstimator<ITransformer>](#) interface.

The pipeline is created using transformations that are added using **mlContext.Transforms**. The first step is to replace any missing values from the model's input, which is done by calling the **ReplaceMissingValues** method from **mlContext.Transforms**.

The **ReplaceMissingValues** method receives an [InputOutputColumnPair](#) array. This array indicates how each model's input columns map to the model's output columns. In this case, the input and output columns have the same names. So, invoking **InputOutputColumnPair(@"CRIM", @"CRIM")** indicates that the **CRIM** input column maps to the **CRIM** output column, and so on.

In short, the **ReplaceMissingValues** method prepares the input data and ensures there aren't missing values for processing. You can specify how to replace missing values, such as using the mean value in a column. The demo code uses the default technique based on the column's data type. In any event, the Boston dataset has no [missing values](#).

Like we saw within the **TestModel.training.cs** file, the **ReplaceMissingValues** method is the first step in creating the pipeline. Following that, we find a series of chained calls to the **Append** method that add more steps to the pipeline.

In the initial **Append** step, we can see the **Concatenate** method from **mlContext.Transforms** being invoked. As you can see, all the dataset columns are concatenated, except the **MEDV** column (which is the value we want the model to predict), to produce the **Features** property. This indicates which input columns will be considered to make the prediction (thus, the name **Features**).

The next step in the pipeline specifies the algorithm to use for training the model, which is done by invoking the **LightGbm** method from **mlContext.Regression.Trainers**.

For the **LightGbm** algorithm, we need to specify a set of configuration options—these are passed as an instance of [LightGbmRegressionTrainer.Options](#).

The great thing about Model Builder is that it can find the best values for the algorithm settings. If we were to do this manually, it would require a lot of trial and error, using different values and seeing which would give the best prediction result. Notice that one of the settings passed to the algorithm is a **GradientBooster.Options** instance, which specifies that the algorithm should implement [gradient boosting](#).

Once the last **Append** method has been chained, the training **pipeline** is ready and returned by the **BuildPipeline** method.

ValuePredictionModel.training vs. TestModel.training

As previously seen, **ValuePredictionModel.consumption.cs** and **TestModel.consumption.cs** are essentially the same in terms of what they do and achieve. But that's not the case with the **ValuePredictionModel.training.cs** and **TestModel.training.cs** files, which are very different. Let's first explore these differences visually.


```

ValuePredictionModel.training.cs
public static IEstimator<ITransformer> BuildPipeline(MLContext mlContext)
{
    // Data process configuration with pipeline data transformations
    var pipeline = mlContext.Transforms.ReplaceMissingValues(
        new Text
        {
            new InputOutputColumnPair("@CRIM", "@CRIM"),
            new InputOutputColumnPair("@ZN", "@ZN"),
            new InputOutputColumnPair("@INDUS", "@INDUS"),
            new InputOutputColumnPair("@CHAS", "@CHAS"),
            new InputOutputColumnPair("@NOX", "@NOX"),
            new InputOutputColumnPair("@RM", "@RM"),
            new InputOutputColumnPair("@AGE", "@AGE"),
            new InputOutputColumnPair("@DIS", "@DIS"),
            new InputOutputColumnPair("@RAD", "@RAD"),
            new InputOutputColumnPair("@TAX", "@TAX"),
            new InputOutputColumnPair("@PTRATIO", "@PTRATIO"),
            new InputOutputColumnPair("@B", "@B"),
            new InputOutputColumnPair("@LSTAT", "@LSTAT")
        })
        .Append(mlContext.Transforms.Concatenate("Features",
            new Text
            {
                "@CRIM", "@ZN", "@INDUS", "@CHAS", "@NOX", "@RM",
                "@AGE", "@DIS", "@RAD", "@TAX", "@PTRATIO", "@B", "@LSTAT"
            })
        );
    Append(mlContext.Regression.Trainers.LightGbm(
        new LightGbmRegressionTrainer.Options() {
            NumberOfLeaves=4,
            NumberOfIterations=2632,
            MinimumExampleCountPerLeaf=20,
            LearningRate=0.599531696349256,
            LabelColumnName="@MEDV",
            FeatureColumnName="@Features",
            ExampleWeightColumnName=null,
            Booster=new GradientBooster.Options() {
                SubsampleFraction=0.999999776672986,
                FeatureFraction=0.878434202775917,
                L1Regularization=2.91194636064836E-10,
                L2Regularization=0.999999776672986
            },
            MaximumBinCountPerFeature=246
        });
    return pipeline;
}

TestModel.training.cs
public static IEstimator<ITransformer> BuildPipeline(MLContext mlContext)
{
    // Data process configuration with pipeline data transformations
    var pipeline = mlContext.Transforms.Text.FeaturizeText(inputColumnName:@"text", outputColumnName:@"text")
        .Append(mlContext.Transforms.Concatenate(
            @"Features", new I[] {@"text"}))
        .Append(mlContext.Transforms.Conversion.MapValueToKey(
            outputColumnName:@"target", inputColumnName:@"target"))
        .Append(mlContext.MulticlassClassification.Trainers.OneVersusAll(
            new BinaryEstimator
            {
                mlContext.BinaryClassification.Trainers.FastTree(
                    new FastTreeBinaryTrainer.Options() {
                        NumberOfLeaves=4,
                        MinimumExampleCountPerLeaf=20,
                        NumberOfTrees=4,
                        MaximumBinCountPerFeature=254,
                        FeatureFraction=1,
                        LearningRate=0.1,
                        LabelColumnName=@"target",
                        FeatureColumnName=@"Features"
                    }, labelColumnName: @"target")
            })
        .Append(mlContext.Transforms.Conversion.MapKeyToValue(
            outputColumnName:@"PredictedLabel", inputColumnName:@"PredictedLabel"));
    return pipeline;
}

```

Figure 3-m: ValuePredictionModel.training.cs vs. TestModel.training.cs

I've included the **BuildPipeline** method for both files because they differ, as seen in the preceding image; however, the **RetrainPipeline** method is the same for both files. Given that the differences between the files reside only within the **BuildPipeline** method, I've highlighted these differences with different colors.

I have highlighted in **red** the main differences in how the pipeline is built. We can see the number of steps required to create the pipeline within the **ValuePredictionModel.training.cs** file is one less than the number of steps needed to create a pipeline within the **TestModel.training.cs** file.

In **light green**, I have highlighted the common features of the **BuildPipeline** method for both files. Essentially, the only commonality is that both declare and return the **pipeline** variable.

The algorithm to train each of these models is highlighted in **light blue**. The algorithm used by the **TestModel.training.cs** file is **FastTree**, and the one used by **ValuePredictionModel.training.cs** is **LightGbm**.

Highlighted in **light yellow**, we can see the data and feature preparation steps used only within the **ValuePredictionModel.training.cs** file.

By looking at these differences, we can conclude that the differences between these two models, in terms of code, are only three. These differences are down to how the pipelines are built, the data preparation steps, and the algorithm used to train each model.

Summary

We have seen throughout this chapter that a value prediction model solves a different type of prediction problem than a data classification model. Both scenarios are based on using tabular data as input to create a prediction.

Essentially, the main differences between them reside in the type of data used (such as the number of input columns), the steps used in creating each pipeline, and the algorithm applied to generate the prediction.

Nevertheless, the ways both models are retrained and consumed resemble each other, so we can almost certainly assume that this will likely be the pattern for different prediction scenarios.

In the following chapter, we will explore a different scenario using nontabular input data and see whether we can corroborate the exciting findings from this chapter.

Chapter 4 Image Classification

Quick intro

So far, we have explored how to generate ML.NET models using Model Builder to create predictions using data structured as tables with defined input columns. In this chapter, we will step away from tabular data and work with images, an entirely different way of organizing data (as a collection of pixels).

We will focus on creating an **image classification** scenario. Contrary to previous chapters, we will write the code manually instead of initially using Model Builder. Once we have written the code, we will use Model Builder to generate code and compare that to the code we wrote.

I'll leave a vital piece of the solution missing until the end, as this will help you understand how an image classification model has different requirements than what we've seen in previous chapters.

Imaging dataset

We'll use the following [image dataset](#) from the book's GitHub [repository](#). The dataset consists of images from [another GitHub repository](#) and pictures from [Pixabay](#). Within the dataset (which you need to unzip after downloading), you'll find the images that shall be used for training within the sample-images folder. The images to test the model are located within the test-images folder.

The training images have been placed into subfolders (found within the sample-images folder), each representing an image classification category. These subfolders can be seen in the following figure.

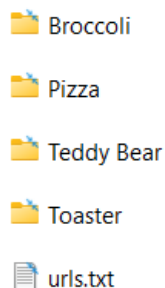


Figure 4-a: Image Classification Category (For Training)

I've also included a urls.txt file within the sample-images folder. This file indicates the sources from where each image was obtained (this [GitHub repository](#) and [Pixabay](#)).

When working with image classification in ML.NET, the images used to train the model must be placed within a specific subfolder structure, where each folder indicates the image category associated with a particular image type.

In other words, the Pizza subfolder contains pizza images, the Broccoli subfolder contains pictures of broccoli, and so on—these subfolders represent image labels.

ImgClass project

As previously mentioned, this time, we will write the code manually for the image classification scenario and then compare it to the structure that Model Builder will produce. We'll use Model Builder later after manually writing all the ImgClass project's code.



Note: *For consistency and ease of following along, the names of the classes, variables, and methods used throughout the code presented in this chapter will be the same as the ones used in previous chapters. It is not a must to follow this convention, but I think it helps with overall understanding.*

To do that, let's create a new Visual Studio project: a console application. Click the **File** menu, click **New**, and then click **Project**.

Select the **Console App** project option and then click **Next**. You'll then be shown the screen where you can specify the project's name. I've renamed the project's default name to **ImgClass** (which I suggest you use) and set the project's **Location**.

Once done, to continue to the following screen, click **Next**, and as for the Framework option, any of the options available is fine—I'll be using **.NET 6.0 (Long Term Support)**. Once done, click **Create**.

You should have the new Visual Studio console application project ready by now. Next, we can add a new C# class file to the project. We can do this by going into **Solution Explorer** and selecting the project name, **ImgClass**. Then, right-click, click **Add**, and then click **Class**.

Let's call the class **ImageClassification.cs** and then click **Add**.

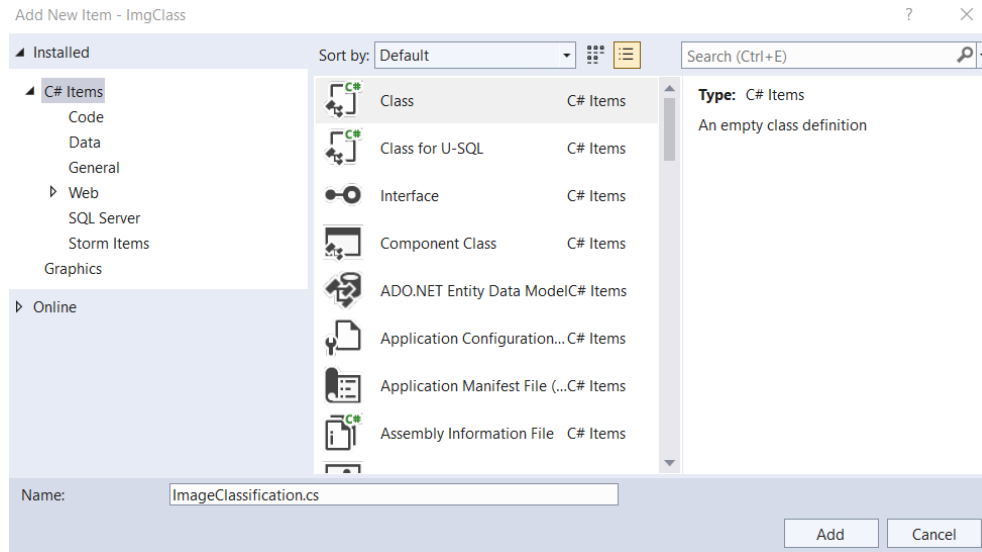


Figure 4-b: Add New Item (ImageClassification.cs for the ImgClass Visual Studio Project)

With the class added to the project, we are ready to start writing the code.

ImageClassification.cs

By default, the newly created ImageClassification.cs file will contain the following code.

Code Listing 4-a: ImageClassification.cs (Newly Created)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ImgClass
{
    internal class ImageClassification
    {
    }
}
```

Let's remove all the **using** statements, as we don't need them—resulting in the following code.

Code Listing 4-b: ImageClassification.cs (Newly Created without the using statements)

```
namespace ImgClass
{
    internal class ImageClassification
    {
    }
```

```
}  
}
```

ImageClassification.cs steps

Before we start making any other changes and adding code, let's understand what steps we need to take to ensure we write all the code we need. There are two main parts to the code we are about to write. One part has to do with the consumption of the model, and the other with training the model. We'll start by writing with the consumption part of the model's logic and later focus on the training part.

Overall, the consumption and training parts have more minor aspects of logic to consider, which can be divided as follows.

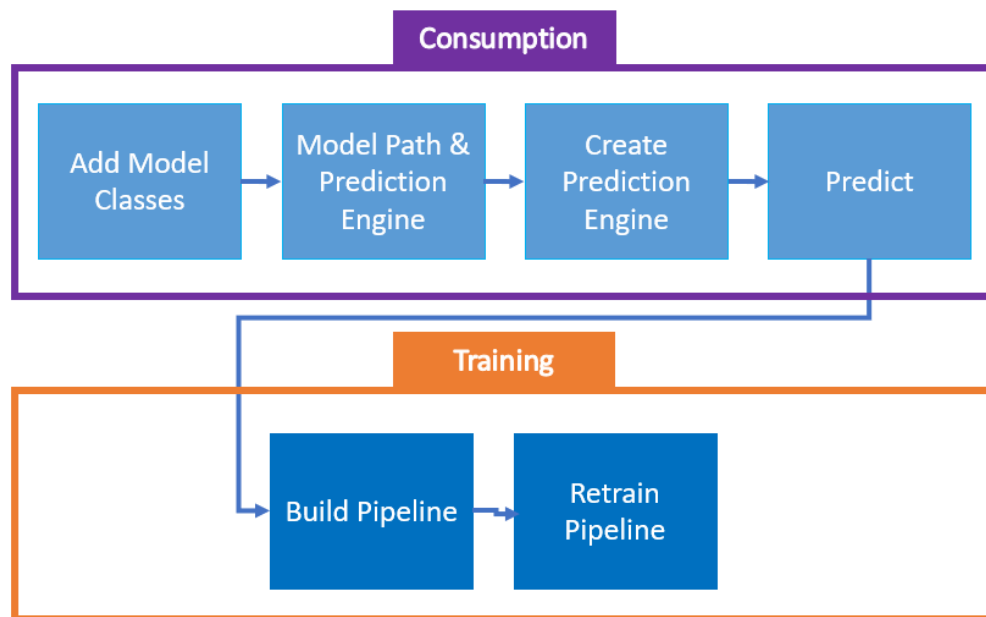


Figure 4-c: Logical Parts of `ImageClassification.cs` Code

From the preceding figure, we can see that within the model's consumption logic, we'll begin by adding the input and output classes required by the model. Following that, we'll add the model's path and the prediction engine. Then, we'll create the prediction engine and write the prediction logic.

Once we have the consumption part of the logic sorted, we'll focus on the training aspect, in which we'll build the training pipeline, and then we'll focus on how to retrain it.

Now that we know the steps needed, let's get started.

Add Model classes

First, we need to change the class from **internal** to **public**, and then we need to declare the **ModelInput** and **ModelOutput** classes. We'll also need to add the reference to **Microsoft.ML.Data**. The changes are highlighted in **bold** in the following code listing.

Code Listing 4-c: ImageClassification.cs (Including the ModelInput and ModelOutput Classes)

```
using Microsoft.ML.Data;

namespace ImgClass
{
    public class ImageClassification
    {
        public class ModelInput
        {
            [LoadColumn(0)]
            [ColumnName(@"Label")]
            public string Label { get; set; }

            [LoadColumn(1)]
            [ColumnName(@"ImageSource")]
            public byte[] ImageSource { get; set; }
        }

        public class ModelOutput
        {
            [ColumnName(@"Label")]
            public uint Label { get; set; }

            [ColumnName(@"ImageSource")]
            public byte[] ImageSource { get; set; }

            [ColumnName(@"PredictedLabel")]
            public string PredictedLabel { get; set; }

            [ColumnName(@"Score")]
            public float[] Score { get; set; }
        }
    }
}
```

Within the **ModelInput** class, we declared the **Label** and **ImageSource** properties. The **Label** property represents the category to which the training image belongs, in other words, the subfolder where the image resides. The **ImageSource** property indicates the file path to the training image.

Within the **ModelOutput** class, we declare the same input properties, **Label**, **ImageSource**, and two extra properties: **PredictedLabel**, and **Score**.

The **PredictedLabel** property will indicate what category a nontraining image will be given once processed by the machine learning algorithm that the model will utilize. The machine learning algorithm will use the **Score** property that the model will utilize to indicate how confident the machine learning algorithm is of the category (**PredictedLabel**) assigned to the image.

Model path and prediction engine

Next, we need to specify the path to the model's metadata (we'll create the metadata file later when running Model Builder), and we also need to declare the instance of the prediction engine. We also need to reference **Microsoft.ML**.

Those changes are highlighted in **bold** in the following code listing.

Code Listing 4-d: ImageClassification.cs (Including the Model Metadata File Path and Prediction Engine)

```
using Microsoft.ML;  
using Microsoft.ML.Data;  
  
namespace ImgClass  
{  
    public class ImageClassification  
    {  
        public class ModelInput  
        {  
            [LoadColumn(0)]  
            [ColumnName(@"Label")]  
            public string Label { get; set; }  
  
            [LoadColumn(1)]  
            [ColumnName(@"ImageSource")]  
            public byte[] ImageSource { get; set; }  
        }  
  
        public class ModelOutput  
        {  
            [ColumnName(@"Label")]  
            public uint Label { get; set; }  
  
            [ColumnName(@"ImageSource")]  
            public byte[] ImageSource { get; set; }  
  
            [ColumnName(@"PredictedLabel")]  
            public string PredictedLabel { get; set; }  
        }  
    }  
}
```



```

        [ColumnName(@"Score")]
        public float[] Score { get; set; }
    }

    private static string MLNetModelPath =
        Path.GetFullPath("ImgClass.zip");

    public static readonly
        Lazy<PredictionEngine<ModelInput, ModelOutput>>
        PredictEngine = new Lazy
            <PredictionEngine<ModelInput, ModelOutput>>
            (() => CreatePredictEngine(), true);
}

```

As you can see, the process is quite familiar. The file path to the future model's metadata file is called **MLNetModelPath**. We could have chosen to give it another name, but I'll stick to the same variable name used in previous models.

The **MLNetModelPath** variable points to the **ImgClass.zip** file, which is the model's metadata file (which has yet to be created—we'll do that later when creating the model using Model Builder).

Then we declare **PredictEngine** as an instance of **Lazy<PredictionEngine<ModelInput, ModelOutput>>**.

The constructor receives a lambda function that creates the engine as the first parameter: **() => CreatePredictEngine()**, and the second one (**true**) indicates whether the instance can be used by multiple threads and is thread-safe. Here, we also follow the same naming convention and pattern we have witnessed in previous chapters.

So far, everything looks familiar. But there's something slightly different. Have you noticed it yet? If you haven't, don't worry.

The difference (compared to what we have seen in previous chapters) is that, in this case, **ImageClassification** is not a **partial** class. And that's precisely my intention, to put both the model's training and consumption logic into one file.

So far, all the code we've written for **ImageClassification** is related to the consumption aspect of the model's logic. To finalize the consumption aspect, we still need to do two things: create the prediction engine and execute the prediction.

Create prediction engine

Now that we have declared the **PredictEngine** variable, let's create the prediction engine—which we can do with the **CreatePredictEngine** method. The changes are highlighted in **bold** in the following code listing.

Code Listing 4-e: ImageClassification.cs (Including the CreatePredictEngine Method)

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

namespace ImgClass
{
    public class ImageClassification
    {
        public class ModelInput
        {
            [LoadColumn(0)]
            [ColumnName(@"Label")]
            public string Label { get; set; }

            [LoadColumn(1)]
            [ColumnName(@"ImageSource")]
            public byte[] ImageSource { get; set; }
        }

        public class ModelOutput
        {
            [ColumnName(@"Label")]
            public uint Label { get; set; }

            [ColumnName(@"ImageSource")]
            public byte[] ImageSource { get; set; }

            [ColumnName(@"PredictedLabel")]
            public string PredictedLabel { get; set; }

            [ColumnName(@"Score")]
            public float[] Score { get; set; }
        }

        private static string MLNetModelPath =
            Path.GetFullPath("ImgClass.zip");

        public static readonly
            Lazy<PredictionEngine<ModelInput, ModelOutput>>
            PredictEngine = new Lazy
```

```

        <PredictionEngine<ModelInput, ModelOutput>>
        (( ) => CreatePredictEngine(), true);

    private static PredictionEngine<ModelInput, ModelOutput>
        CreatePredictEngine()
    {
        var mlContext = new MLContext();
        ITransformer mlModel =
            mlContext.Model.Load(MLNetModelPath, out var _);
        return mlContext.Model.
            CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
    }
}

```

As you might recall from the previous chapters, the **CreatePredictEngine** method creates an instance of **MLContext** that gets assigned to **mlContext**.

The **Load** method loads the model, as its name implies. To this method, **MLNetModelPath** is passed as a parameter, and the **modelInputSchema** is represented as the **out var _** parameter. Finally, the **Load** method returns the model loaded (**mlModel**).

The prediction engine, previously declared, is finally created when **CreatePredictionEngine** is invoked, passing **mlModel** as a parameter.

Predict

With the prediction engine created, we next need to create the method responsible for predicting the model's results. So, let's do that following the same naming conventions we've been using so far. I've highlighted the changes in **bold** in the following code listing.

Code Listing 4-f: ImageClassification.cs (Including the Predict Method)

```

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

namespace ImgClass
{
    public class ImageClassification
    {
        public class ModelInput
        {
            [LoadColumn(0)]
            [ColumnName(@"Label")]
            public string Label { get; set; }

            [LoadColumn(1)]

```

```

        [ColumnName(@"ImageSource")]
        public byte[] ImageSource { get; set; }
    }

    public class ModelOutput
    {
        [ColumnName(@"Label")]
        public uint Label { get; set; }

        [ColumnName(@"ImageSource")]
        public byte[] ImageSource { get; set; }

        [ColumnName(@"PredictedLabel")]
        public string PredictedLabel { get; set; }

        [ColumnName(@"Score")]
        public float[] Score { get; set; }
    }

    private static string MLNetModelPath =
        Path.GetFullPath("ImgClass.zip");

    public static readonly
        Lazy<PredictionEngine<ModelInput, ModelOutput>>
        PredictEngine = new Lazy
            <PredictionEngine<ModelInput, ModelOutput>>
            (() => CreatePredictEngine(), true);

    private static PredictionEngine<ModelInput, ModelOutput>
        CreatePredictEngine()
    {
        var mlContext = new MLContext();
        ITransformer mlModel =
            mlContext.Model.Load(MLNetModelPath, out var _);
        return mlContext.Model.
            CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
    }

    public static ModelOutput Predict(ModelInput input)
    {
        var predEngine = PredictEngine.Value;
        return predEngine.Predict(input);
    }
}

```

As you can see, the **Predict** method takes an instance of the **ModelInput** class as a parameter and then executes the **Predict** method from **PredictEngine.Value**.

So, with this done, we have completed the consumption part of the image classification code. Next, let's add training logic to the code.

Build the pipeline

To train the model, we need to add the logic to build the image classification pipeline, which we can do in three steps:

- **Convert categorical (string) values into numerical ones**—this is required before executing the training algorithm. In image classification, the input values are numeric (pixels), but in this example, the labels, such as broccoli, are strings that must be converted to integers.
- **Specify the training algorithm.**
- **Convert the key types back to their original (categorical) values.**

So, let's add the code for the **BuildPipeline** method with these steps to the existing code we've written so far. The changes are highlighted in **bold** in the following code listing, and I've assigned a specific color to each step.

Code Listing 4-g: ImageClassification.cs (Including the BuildPipeline Method)

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

namespace ImgClass
{
    public class ImageClassification
    {
        public class ModelInput
        {
            [LoadColumn(0)]
            [ColumnName(@"Label")]
            public string Label { get; set; }

            [LoadColumn(1)]
            [ColumnName(@"ImageSource")]
            public byte[] ImageSource { get; set; }
        }

        public class ModelOutput
        {
            [ColumnName(@"Label")]

```

```

        public uint Label { get; set; }

        [ColumnName(@"ImageSource")]
        public byte[] ImageSource { get; set; }

        [ColumnName(@"PredictedLabel")]
        public string PredictedLabel { get; set; }

        [ColumnName(@"Score")]
        public float[] Score { get; set; }
    }

    private static string MLNetModelPath =
        Path.GetFullPath("ImgClass.zip");

    public static readonly
        Lazy<PredictionEngine<ModelInput, ModelOutput>>
        PredictEngine = new Lazy
            <PredictionEngine<ModelInput, ModelOutput>>
            (() => CreatePredictEngine(), true);

    private static PredictionEngine<ModelInput, ModelOutput>
        CreatePredictEngine()
    {
        var mlContext = new MLContext();
        ITransformer mlModel =
            mlContext.Model.Load(MLNetModelPath, out var _);
        return mlContext.Model.
            CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
    }

    public static ModelOutput Predict(ModelInput input)
    {
        var predEngine = PredictEngine.Value;
        return predEngine.Predict(input);
    }

    public static IEstimator<ITransformer> BuildPipeline(
        MLContext mlContext)
    {
        var pipeline = mlContext.Transforms.Conversion.
            MapValueToKey(
                outputColumnName: @"Label",
                inputColumnName: @"Label")
            .Append(mlContext.MulticlassClassification.
                Trainers.ImageClassification(
                    labelColumnName: @"Label",
                    scoreColumnName: @"Score",

```

```

        featureColumnName: @"ImageSource")
    )
    .Append(mlContext.Transforms.Conversion.
        MapKeyToValue(
            outputColumnName: @"PredictedLabel",
            inputColumnName: @"PredictedLabel")
        );

    return pipeline;
}
}
}

```

Highlighted in **yellow** you'll see the first step in building the pipeline, which is responsible for converting categorical (string) values into numerical ones, in this case, for the **Label** column. This is done by invoking the **MapValueToKey** method. This step is important because the image classification algorithm doesn't work with string values.

Then, the next step (which is added to the pipeline using the **Append** method) is to specify the **training algorithm**, which in this case is done by invoking the **ImageClassification** method from **mlContext.MulticlassClassification.Trainers**.

For the training algorithm, we pass the **Label** column, the column containing the confidence percentage of the result obtained (**Score**), and the column specifying the features of the image (which is the image itself, **ImageSource**).

The **final step** is to take the resultant prediction value (**PredictedLabel**) and convert that from a number to a string label, indicating the image's classification or category determined by the algorithm.

Retrain pipeline

The last part of the puzzle is to create the method responsible for retraining the model. Following the naming conventions used in previous chapters, we'll name this method **RetrainPipeline**. In **bold**, I've highlighted the changes to the existing code in the following code listing.

Code Listing 4-h: ImageClassification.cs (Including the RetrainPipeline Method)

```

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

namespace ImgClass
{
    public class ImageClassification
    {
        public class ModelInput

```

```

{
    [LoadColumn(0)]
    [ColumnName(@"Label")]
    public string Label { get; set; }

    [LoadColumn(1)]
    [ColumnName(@"ImageSource")]
    public byte[] ImageSource { get; set; }
}

public class ModelOutput
{
    [ColumnName(@"Label")]
    public uint Label { get; set; }

    [ColumnName(@"ImageSource")]
    public byte[] ImageSource { get; set; }

    [ColumnName(@"PredictedLabel")]
    public string PredictedLabel { get; set; }

    [ColumnName(@"Score")]
    public float[] Score { get; set; }
}

private static string MLNetModelPath =
    Path.GetFullPath(@"ImgClass.zip");

public static readonly
    Lazy<PredictionEngine<ModelInput, ModelOutput>>
    PredictEngine = new Lazy
        <PredictionEngine<ModelInput, ModelOutput>>
        (() => CreatePredictEngine(), true);

private static PredictionEngine<ModelInput, ModelOutput>
    CreatePredictEngine()
{
    var mlContext = new MLContext();
    ITransformer mlModel =
        mlContext.Model.Load(MLNetModelPath, out var _);
    return mlContext.Model.
        CreatePredictionEngine<ModelInput, ModelOutput>(mlModel);
}

public static ModelOutput Predict(ModelInput input)
{
    var predEngine = PredictEngine.Value;

```



```

        return predEngine.Predict(input);
    }

    public static IEstimator<ITransformer> BuildPipeline(
        MLContext mlContext)
    {
        var pipeline = mlContext.Transforms.Conversion.
            MapValueToKey(
                outputColumnName: @"Label",
                inputColumnName: @"Label")
            .Append(mlContext.MulticlassClassification.
                Trainers.ImageClassification(
                    labelColumnName: @"Label",
                    scoreColumnName: @"Score",
                    featureColumnName: @"ImageSource")
            )
            .Append(mlContext.Transforms.Conversion.
                MapKeyToValue(
                    outputColumnName: @"PredictedLabel",
                    inputColumnName: @"PredictedLabel")
            );

        return pipeline;
    }

    public static ITransformer RetrainPipeline(
        MLContext mlContext, IDataView trainData)
    {
        var pipeline = BuildPipeline(mlContext);
        var model = pipeline.Fit(trainData);

        return model;
    }
}

```

The **RetrainPipeline** method receives an instance of [MLContext](#) and the training data ([trainData](#)), which is used as the input and output of transforms. This method returns an [object](#) responsible for transforming data within an ML.NET model pipeline. With the objects in the pipeline created, **trainData** is passed to train the model by invoking the **pipeline.Fit** method, and finally, the **model** is returned.

Generating the model

With the code ready, let's use Model Builder to create the model's metadata and generate the model's code to compare it to what we have done.

Going back to **Solution Explorer**, select the **ImgClass** project by clicking on it and then right-clicking. Click **Add** and then click **Machine Learning Model**.

When prompted, change the default name suggestion to **ImgClass.mbconfig** and then click **Add**. When the Select a scenario window opens, scroll down and click the **Local** button under Image classification.

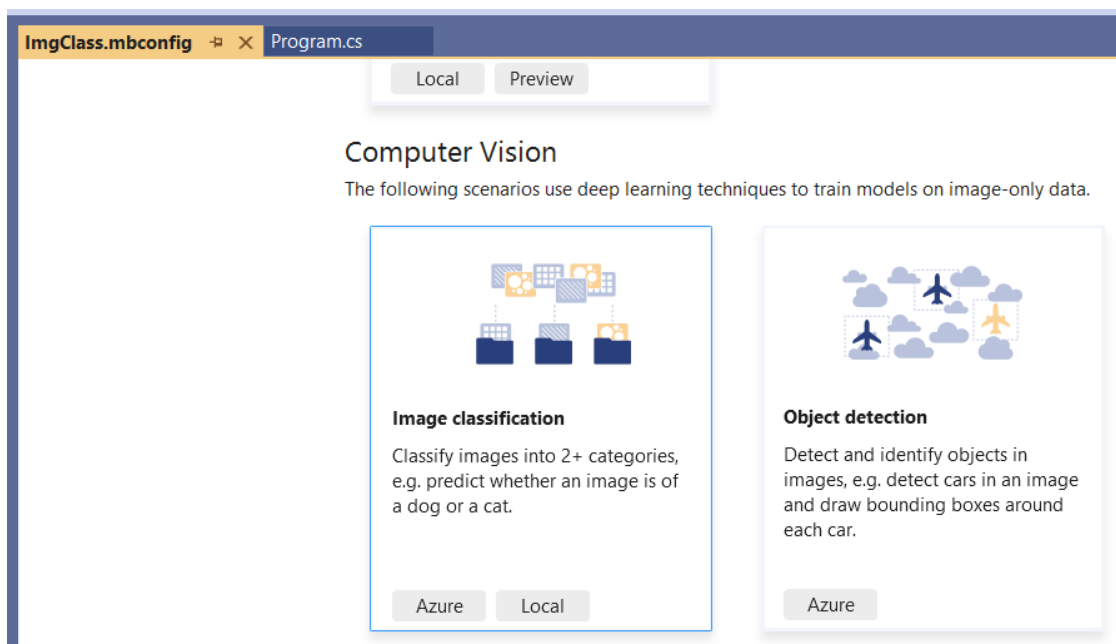


Figure 4-d: Model Builder (Image Classification)

Once done, you'll be shown the Select training environment option. Choose **Local (CPU)** and click **Next step**.

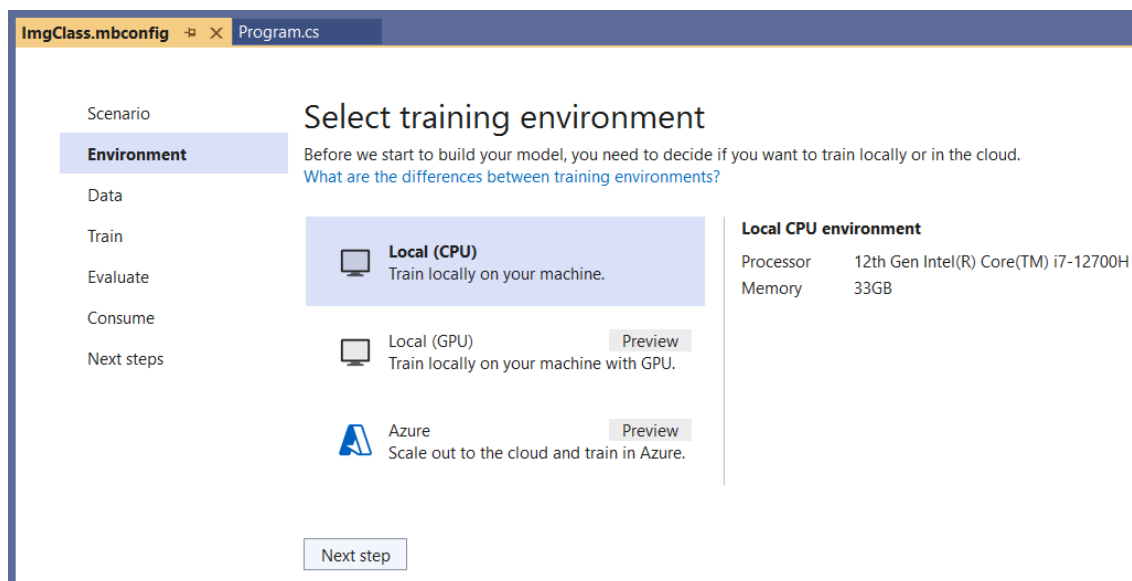


Figure 4-e: Model Builder (Select training environment)

The next step is to provide the dataset: the training images.

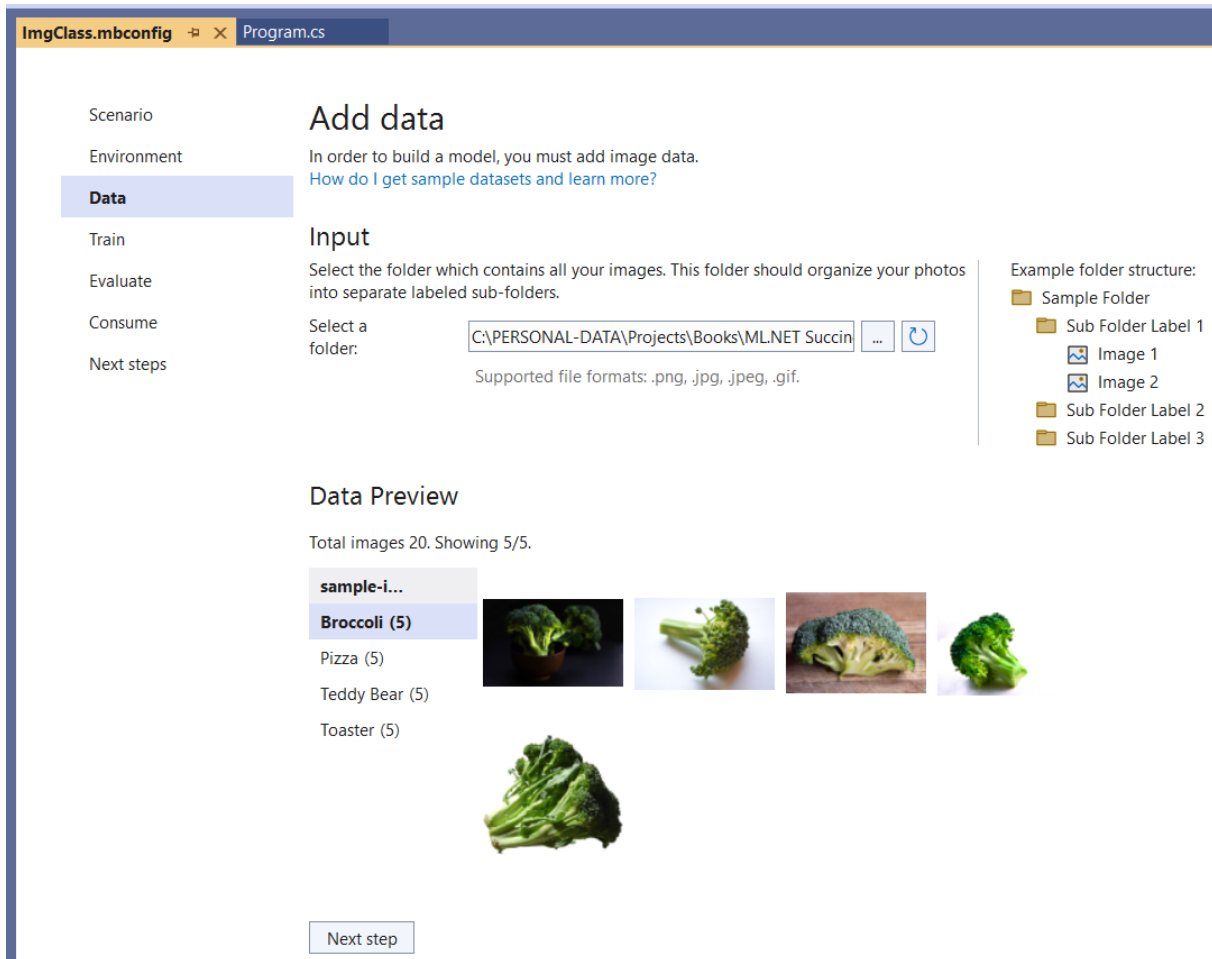


Figure 4-f: Model Builder (Add data)

At this stage, all we need to do is to choose the folder where the training images are located. Then, the images and their different categories will be shown within the Data Preview section. Once done, click **Next step** to train the model.

Within the Train screen, click **Start training** to initiate the training process. Once the training has been finalized, click **Next step** to continue.

Evaluating the model

At this point, the Model Builder has generated the model in the background, and we can test things out by evaluating the model with a test image. Choose one from the dataset provided and the list of sample images and click **Browse an image**.

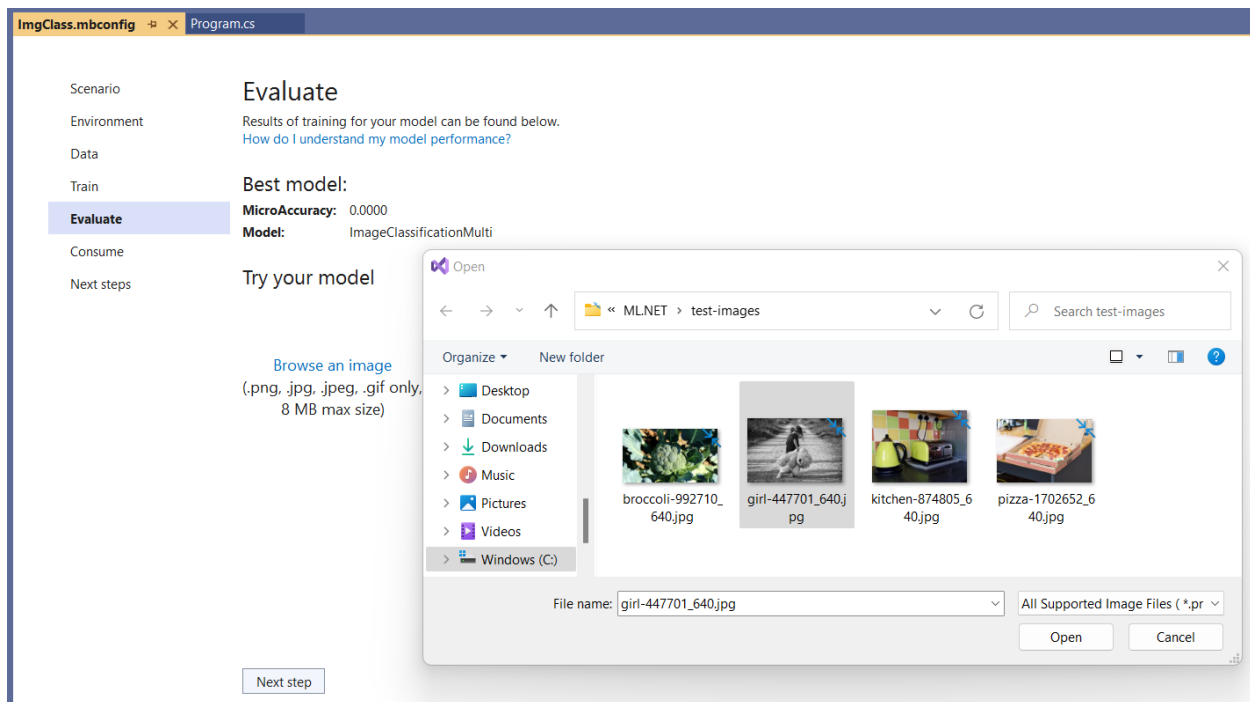


Figure 4-g: Model Builder (Evaluate—Choosing a Test Image)

I will choose the black-and-white photo with the little girl holding a teddy bear and then click **Open**. A few seconds after the image has loaded, Model Builder correctly predicts that the image corresponds to the Teddy Bear category.

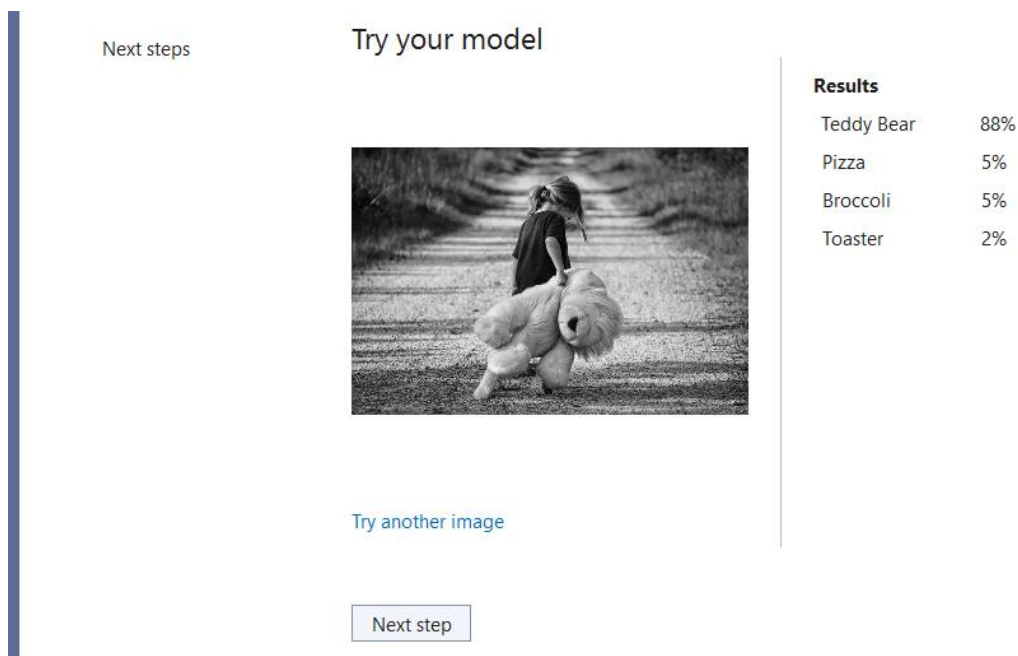


Figure 4-h: Model Builder (Evaluate—Results—Teddy Bear)

The preceding figure shows that the Teddy Bear category has been given a score of 88% compared to the other types. There are two things to which I'd like you to pay close attention. First, we've only trained each image category with five sample images. Second, the sample **Teddy Bear** training images only have a teddy bear and nothing else. However, the test image contains a road, a little girl, and a teddy bear.

So, here's the best part: the test image was correctly predicted to be within the Teddy Bear category despite being only trained with five images per category—this blows me away. If you test the model with other test images that differ from the training images, you'll notice that those predictions will be spot-on.

To do that, click **Try another image**. I will select the image with the light green toaster next to the kettle.

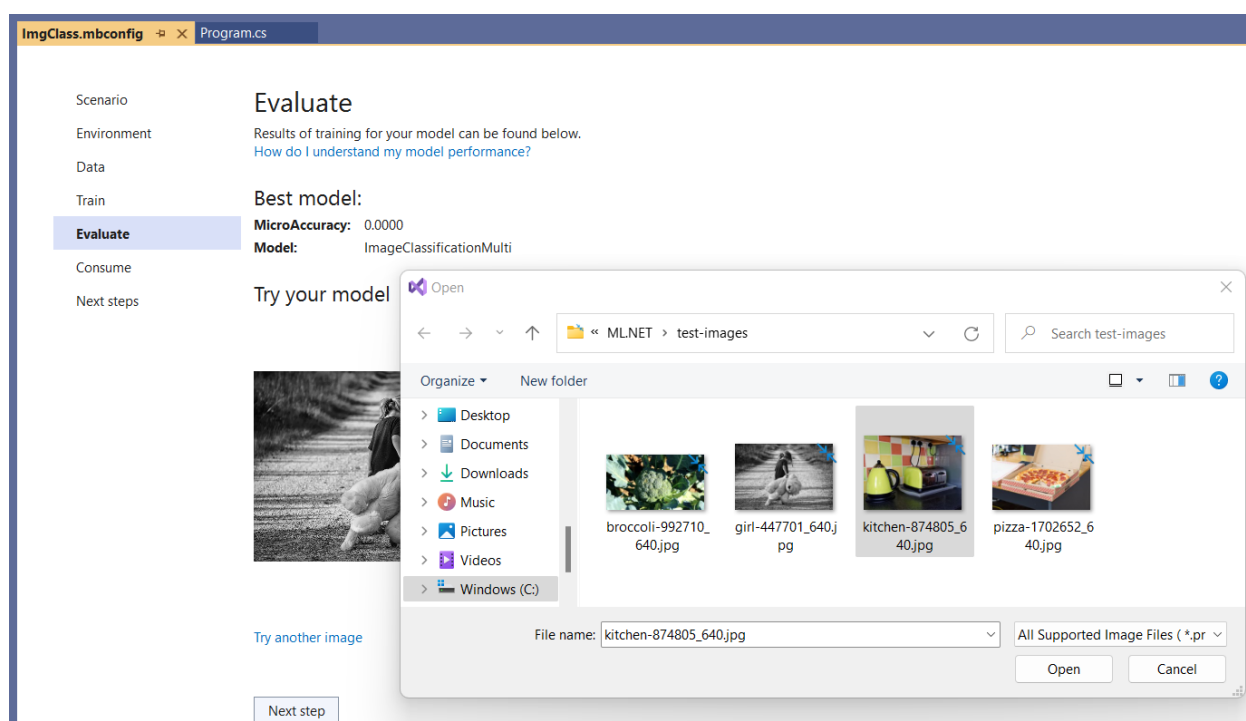


Figure 4-i: Model Builder (Evaluate—Choosing Another Test Image)

Then, click **Open** to see what result we get.

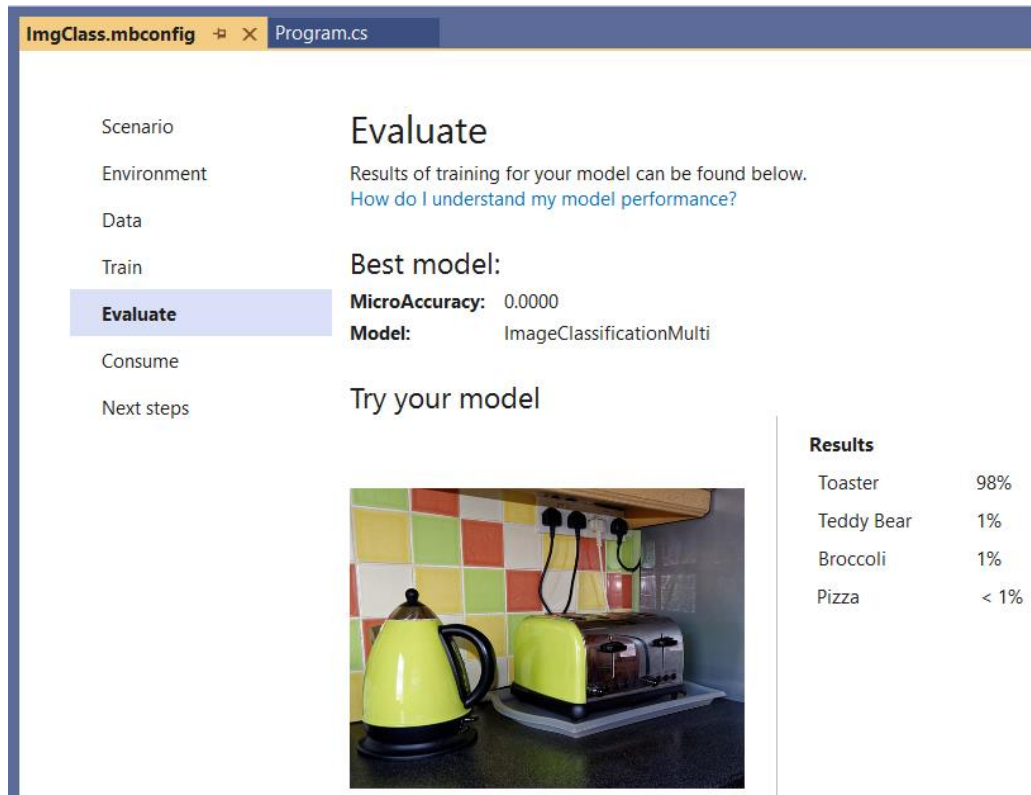


Figure 4-j: Model Builder (Evaluate—Results—Toaster)

After a few seconds, we can see that the image has been correctly labeled as **Toaster**. Since we only used five images per category, I find the results quite impressive, mainly because the test images are not similar to the training images.

Consuming the model and app execution

The last part is to consume the model, and as you know, to do that all we need to do is click on **Next step** and then copy the code snippet.

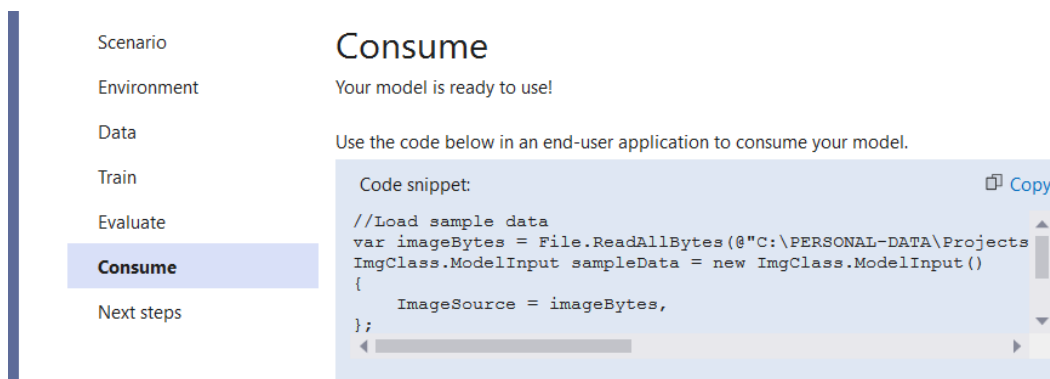


Figure 4-k: Model Builder (Consume)

Let's copy the code snippet, paste it into the **Main** method of **Program.cs**, and adjust it. Following is the modified **Program.cs** code after doing that.

Code Listing 4-i: Modified Program.cs

```
namespace ImgClass
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            //Load sample data.
            var imageBytes = File.ReadAllBytes(
                @"C:\test-images\bowl-of-broccoli-2584307_640.jpg");
            ImgClass.ModelInput sampleData = new ImgClass.ModelInput()
            {
                ImageSource = imageBytes,
            };

            //Load model and predict output.
            var result = ImgClass.Predict(sampleData);
        }
    }
}
```

In principle, we should be ready to click the run button within Visual Studio and see the results. Let's give it a try.

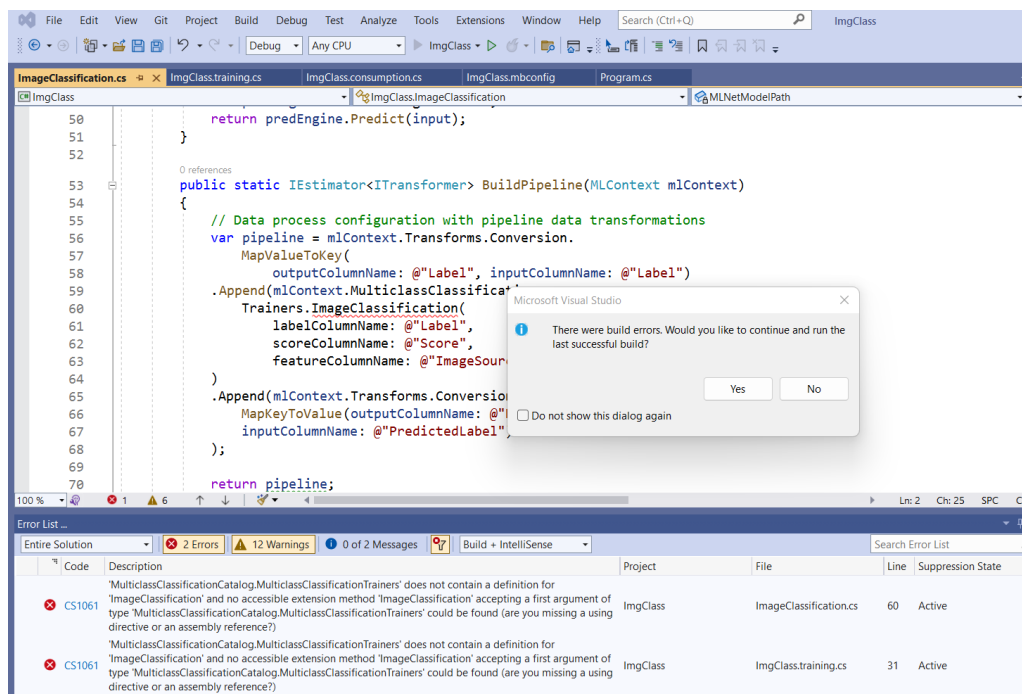


Figure 4-I: Visual Studio—Running the Application (with Errors)

From the preceding figure, we can see that the **ImageClassification** method has been highlighted, and a couple of errors are associated with it.

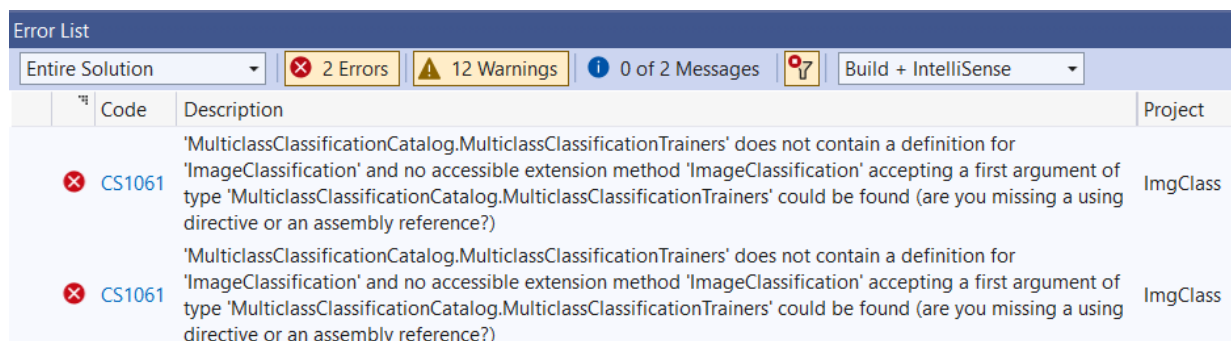


Figure 4-m: Visual Studio Errors

We can see that both errors are the same, indicating that the **ImageClassification** method cannot be found within **mlContext.MulticlassClassification.Trainers**. But why is this? The reason is that we are missing the **LG.Microsoft.ML.Vision** NuGet package, which contains several APIs required for performing image classification using computer vision.

Installing LG.Microsoft.ML.Vision

So, let's add this package. Within **Solution Explorer**, select **ImgClass**, right-click, and then click **Manage NuGet Packages**.

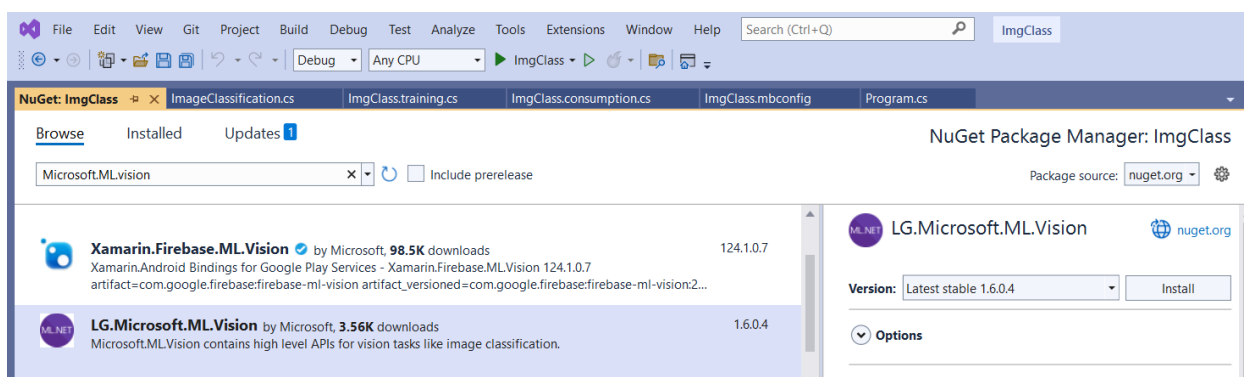


Figure 4-n: NuGet Package Manager (Visual Studio)

To install the **LG.Microsoft.ML.Vision** package, click **Install**.

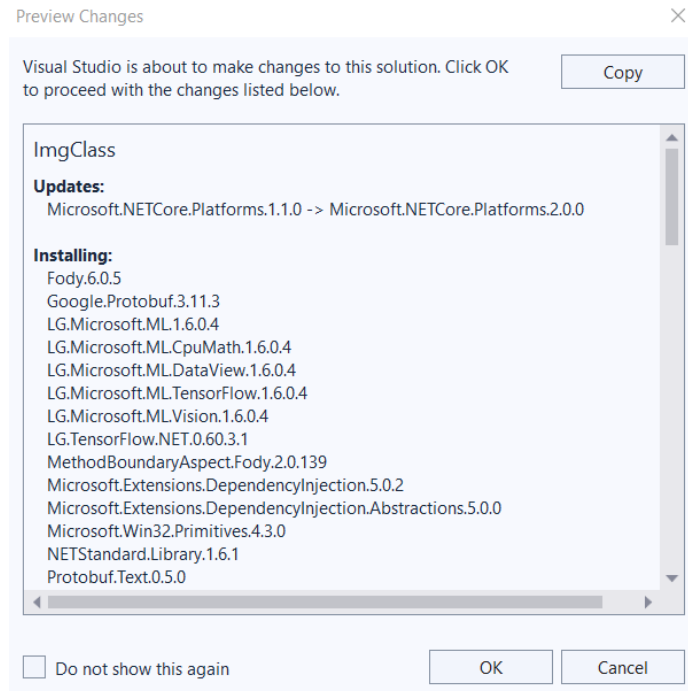


Figure 4-o: Installing the NuGet Package—Preview Changed (Visual Studio)

When prompted, click on **OK** to install the package. As you can see, this package installs a series of other related packages. Before installing these packages, you'll be prompted to accept the license.

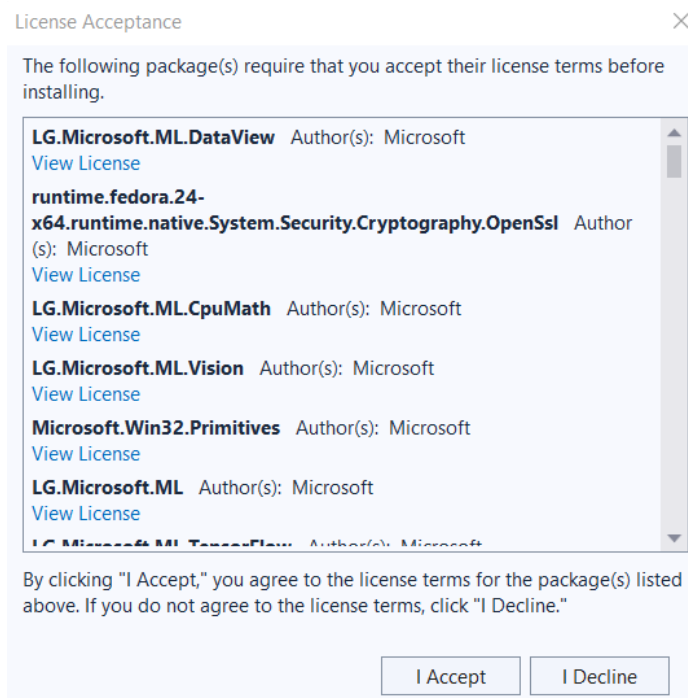


Figure 4-p: License Acceptance (Visual Studio)

At this stage, click on **I Accept** to continue. Once these packages have been installed, you'll notice that the errors are gone. You can verify this by going to **Solution Explorer**, right-clicking **ImgClass**, and clicking **Build**.

Once the build process has succeeded, you can rerun the application by clicking the **Run** button within Visual Studio to see what happens.

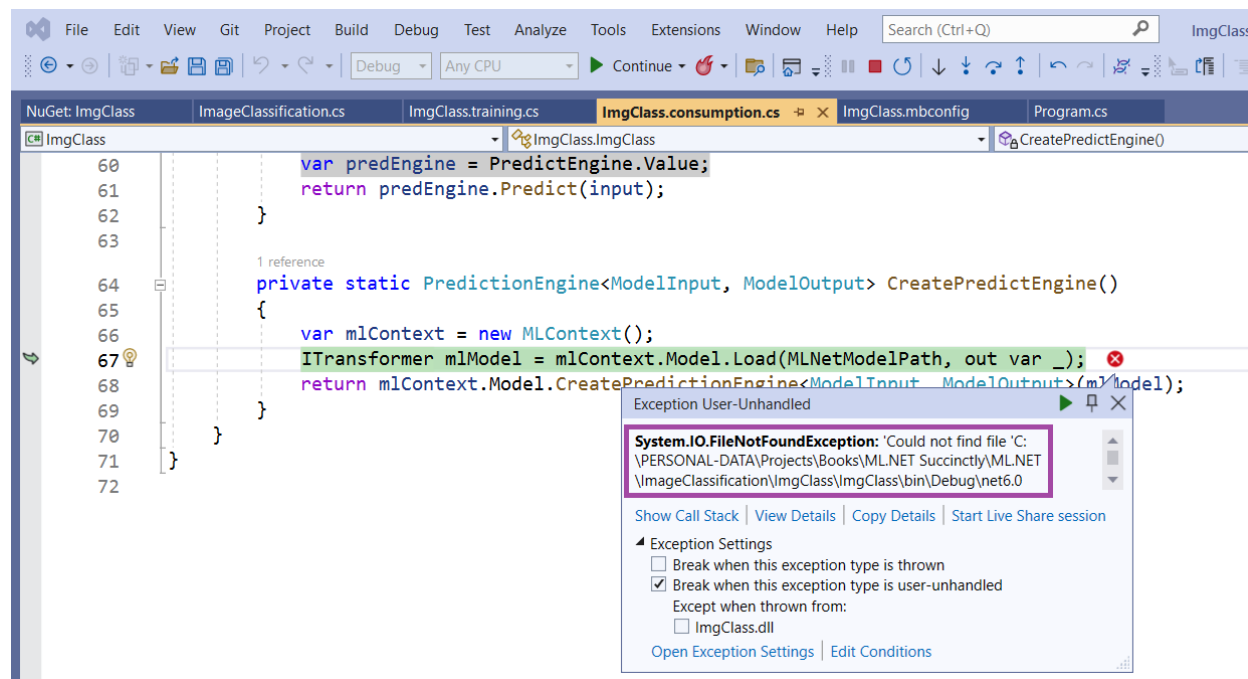


Figure 4-q: Error Executing the Application—Visual Studio

When executing the application, I've run into an error indicating that the **ImgClass.zip** file (the model's metadata) is not in the app's folder.

Fixing the “missing” **ImgClass.zip**

If you run into this error, the solution is simple. First, stop the execution of the application by pressing **Shift+F5**. Then, go to **Solution Explorer** and click on **ImgClass.mbconfig** to expand it. Following that, right-click **ImgClass.zip** and click **Properties**.

With the Properties pane open, change the value of the **Copy to Output Directory** property from **Do not copy** to **Copy if newer**.

Then, click the run button in Visual Studio to execute the application again. In my case, I've run into the following issue.

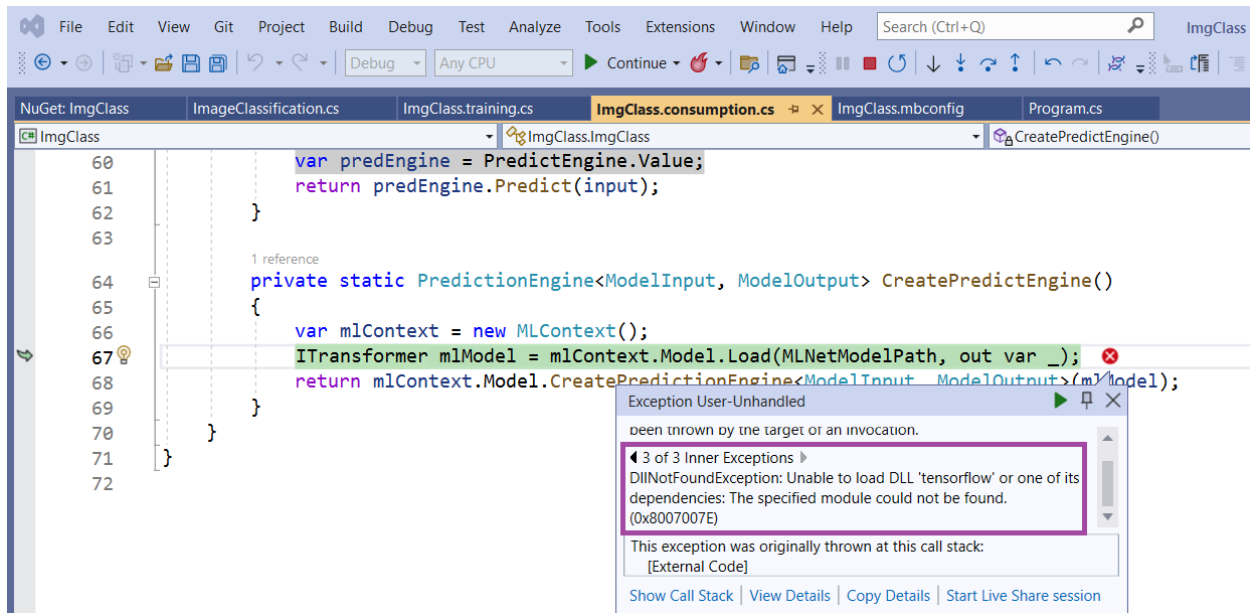


Figure 4-r: Another Error Executing the Application—Visual Studio

Fixing the missing TensorFlow DLL

The preceding error indicates that there's a missing **tensorflow** DLL. [TensorFlow](#) is an open-source and freely available machine learning platform from [Google](#). Internally, ML.NET uses TensorFlow to perform image classification.

To install TensorFlow for ML.NET, open the NuGet Package Manager by selecting the **ImgClass** project within **Solution Explorer**. Right-click, then click **Manage NuGet Packages**, and then click **Install**.

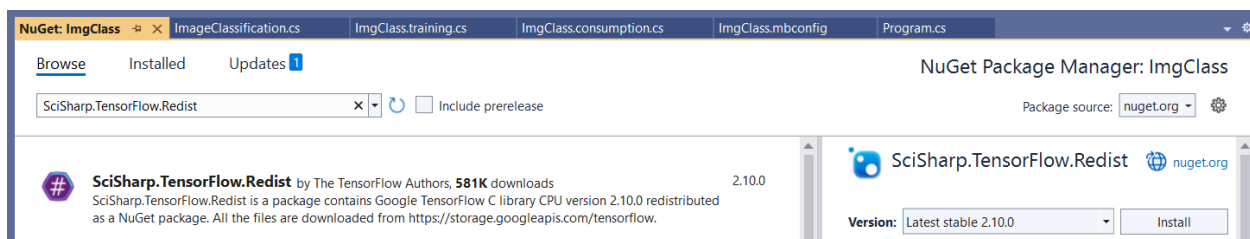
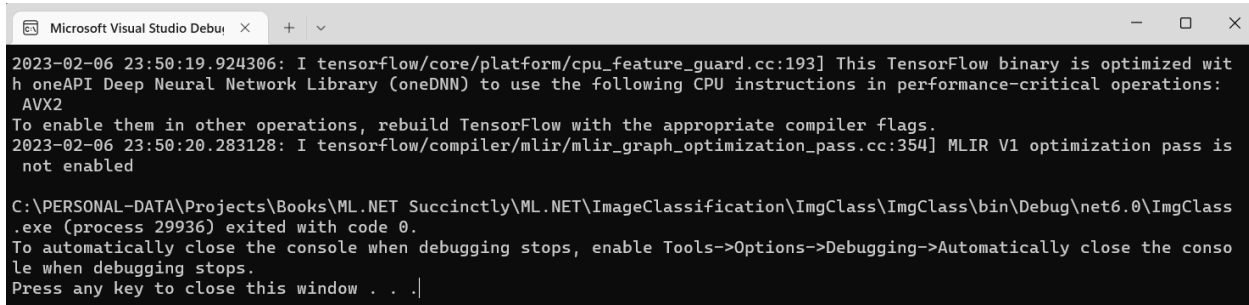


Figure 4-s: The SciSharp.TensorFlow.Redist NuGet Package—Visual Studio

When prompted, click **OK** on the **Preview Changes** window, and later accept any license terms to continue with the installation, if prompted.

Rerunning the app

Once the TensorFlow NuGet package has been installed, we can rebuild the project and rerun the application. Do this by clicking the run button within Visual Studio. After running the app, in my case, I see the following output within the Microsoft Visual Studio Debugger console.

A screenshot of the Microsoft Visual Studio Debugger console window. The window title is "Microsoft Visual Studio Debug Console". The console output shows TensorFlow logs: "2023-02-06 23:50:19.924306: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with the oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2. To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags." followed by "2023-02-06 23:50:20.283128: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:354] MLIR V1 optimization pass is not enabled". Below this, it says "C:\PERSONAL-DATA\Projects\Books\ML.NET Succinctly\ML.NET\ImageClassification\ImgClass\ImgClass\bin\Debug\net6.0\ImgClass.exe (process 29936) exited with code 0." and "To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops." and "Press any key to close this window . . .".

```
Microsoft Visual Studio Debug Console
2023-02-06 23:50:19.924306: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with the oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2. To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-02-06 23:50:20.283128: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:354] MLIR V1 optimization pass is not enabled
C:\PERSONAL-DATA\Projects\Books\ML.NET Succinctly\ML.NET\ImageClassification\ImgClass\ImgClass\bin\Debug\net6.0\ImgClass.exe (process 29936) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Figure 4-t: The App Running—Microsoft Visual Studio Debugger

From the output, you can see that TensorFlow has been invoked, and the application has been successfully executed. However, we don't know the result of the model's prediction. To find that out, let's go back to **Program.cs** and add an extra line of code to display the prediction result (**PredictedLabel**). The changes are highlighted in **bold** in the following code listing.

Code Listing 4-j: Modified Program.cs (With the Results Displayed)

```
namespace ImgClass
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            //Load sample data.
            var imageBytes = File.ReadAllBytes(
                @"C:\test-images\bowl-of-broccoli-2584307_640.jpg");
            ImgClass.ModelInput sampleData = new ImgClass.ModelInput()
            {
                ImageSource = imageBytes,
            };

            //Load model and predict output.
            var result = ImgClass.Predict(sampleData);

            Console.WriteLine(
                 $"Category {result.PredictedLabel}");
        }
    }
}
```

If we rerun the application, we'll see the following result.

```

Microsoft Visual Studio Debug
2023-02-07 00:02:29.369431: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-02-07 00:02:29.630094: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:354] MLIR V1 optimization pass is not enabled
Category Broccoli
C:\PERSONAL-DATA\Projects\Books\ML.NET Succinctly\ML.NET\ImageClassification\ImgClass\ImgClass\bin\Debug\net6.0\ImgClass.exe (process 13776) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 4-u: The App Running (PredictedLabel Shown)—Microsoft Visual Studio Debugger

Great—that worked! Nevertheless, notice that within `Program.cs`, we are using the code that Model Builder generated, not the `ImageClassification.cs` code we wrote. The reason is that the code that exists in the `Main` method of `Program.cs` invokes `ImgClass` and not `ImageClassification`.

So, to finally test the code we wrote, let's change any occurrences of `ImgClass` found within the `Main` method to `ImageClassification`.

Code Listing 4-k: Modified `Program.cs` (Changed All `ImgClass` Occurrences to `ImageClassification`)

```

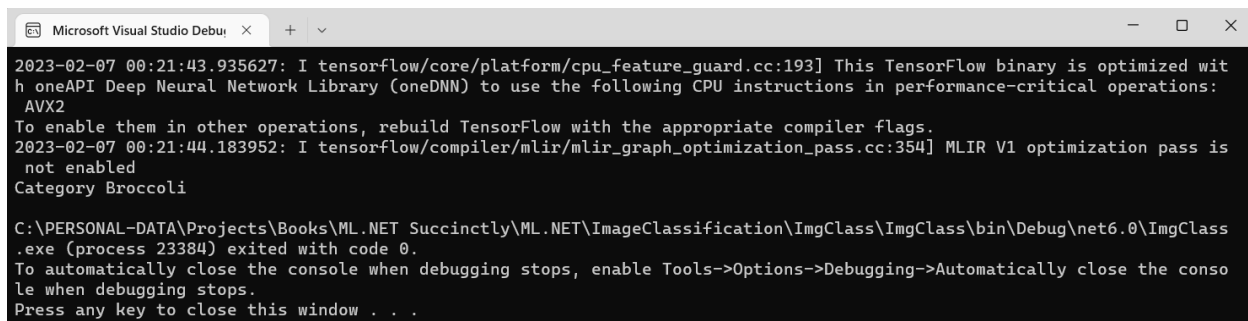
namespace ImgClass
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            //Load sample data.
            var imageBytes = File.ReadAllBytes(
                @"C:\test-images\bowl-of-broccoli-2584307_640.jpg");
            ImageClassification.ModelInput sampleData =
                new ImageClassification.ModelInput()
            {
                ImageSource = imageBytes,
            };

            //Load model and predict output.
            var result = ImageClassification.Predict(sampleData);

            Console.WriteLine(
                $"Category {result.PredictedLabel}");
        }
    }
}

```

If we rerun the program, we should see the same result.



```
Microsoft Visual Studio Debug Console
2023-02-07 00:21:43.935627: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with
oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:
AVX2
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
2023-02-07 00:21:44.183952: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:354] MLIR V1 optimization pass is
not enabled
Category Broccoli

C:\PERSONAL-DATA\Projects\Books\ML.NET Succinctly\ML.NET\ImageClassification\ImgClass\ImgClass\bin\Debug\net6.0\ImgClass
.exe (process 23384) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

Figure 4-v: The App Running (PredictedLabel Shown)—Microsoft Visual Studio Debugger

Indeed, the result is the same. At this stage, you might realize that using the same naming conventions within the `ImageClassification.cs` file was probably worth it because all we had to do was swap `ImgClass` with `ImageClassification`.

Summary and final thoughts

We've reached the end of this chapter and the book. As you noticed, this chapter was structured differently than the previous ones. The structural change of this final chapter was done on purpose. I wanted to take you on the road less traveled by hitting a few bumps along the way but still ending up with the same outcome.

I think the goal was achieved. By focusing on the code logic first and then using Model Builder to complete what we started, we stepped out of the ML.NET comfort zone. Interestingly, we ran into a few missing pieces and learned how to correct them by following the process differently.

This lesson didn't require diving too deep into ML.NET classes but provided valuable insights into using this library. Hopefully, you agree with me. Nevertheless, although we did get a good idea of the features, ease of use, and power behind ML.NET, we just scratched the surface of what is possible.

In the future, if any of the machine learning scenarios using ML.NET that we touched upon piqued your interest, additional goodies are awaiting you. There are other scenarios that Model Builder covers out of the box that we could not cover, as doing that would probably require a couple of other *Succinctly* books.

Furthermore, as we saw in this final chapter, ML.NET integrates and uses capabilities and features from other machine learning frameworks, such as TensorFlow. If using external machine learning frameworks in combination with ML.NET excites you, there's a wealth of learning opportunities waiting.

To conclude, machine learning is not easy. However, thanks to the great work of the engineering team behind ML.NET at Microsoft, it is now accessible to any .NET developer. I think ML.NET is a significant step forward in empowering any .NET developer to embrace the world of machine learning, which every day becomes more prevalent and essential.

Until next time, never stop learning, take care, and thank you for reading.