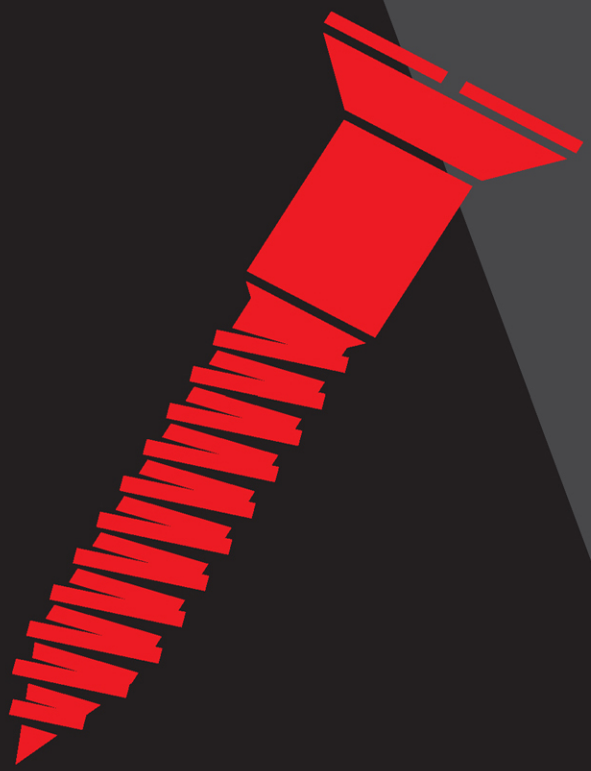
Microsoft

Programming ML.NET



Professional

Dino Esposito
Francesco Esposito

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Microsoft

Programming ML.Net

Dino Esposito
Francesco Esposito

Programming ML.Net

Published with the authorization of Microsoft Corporation by:

Pearson Education, Inc.

ISBN-13: 978-0-13-738365-8

ISBN-10: 0-13-738365-7

Library of Congress Control Number: 2021952995

ScoutAutomatedPrintCode

Editor-in-Chief
Brett Bartow

Executive Editor
Loretta Yates

Sponsoring Editor
Charvi Arora

Development Editor
Rick Kughen

Managing Editor
Sandra Schroeder

Senior Project Editor
Tracey Croom

Copy Editor
Rick Kughen

Indexer
Timothy Wright

Proofreader
Abigail Manheim

Technical Editor
Bri Achtman

Cover Designer
Twist Creative, Seattle

Compositor
codeMantra

To Silvia, Michela and new dreams
— Dino Esposito

To my loved ones, to whom I couldn't help but dedicate a book
— Francesco Esposito

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

This page intentionally left blank

Acknowledgments

FROM DINO:

It's the second time that the two of us, father and son, have written a machine learning book and a lot has changed since our last one, two years ago. In this book, we really joined forces—I put my software experience on the table, and Francesco gave his freshness, energy, and mathematical skills. We learned both how tricky it can be to put machine learning solutions in production and how "easy" it can be to hide those little gems in the folds of "normal" ASP.NET applications.

In the past two years, we achieved other results and, for example, we solidified our grasp of software for professional tennis and expanded to healthcare, agriculture, and customer care. The common denominator is always that one: intelligent software that ends up doing intelligent things. It's not about replacing humans and killing jobs—quite the reverse. It's about replacing tasks with automated procedures, thus freeing humans from boring, automatable tasks and keeping them engaged in more interesting activities.

With Giorgio Garcia-Agreda and Gaetano Guarino, and the entire Crionet crew, we're making our tennis fanatic dreams bigger every day. We are changing the games. With Vito Lanzotti and the KBMS Data Force team, we're making silent history by turning doctors' operating dreams into concrete and applicable artifacts, thus smoothing the way patients receive care. With Salvo Intilisano and Daniel Intilisano of Karma Enterprise, it was really a matter of technical karma. Same mindset, same vision, and same father-and-son business model! Agriculture won't be the same after the project ends, and the bees will be grateful!

The Youbiquitous team is growing, and the business is now spread over multiple pairs of strong shoulders—mainly those of Matteo, Luciano, Martina, Filippo, and Gabriele. Thank you all for taking the time to keep the business running while we were having fun with ML.NET.

Finally, any book is teamwork, and it is our pleasure to call out the names of those who made it ultimately possible. Last but not certainly least! A monumental thank-you goes to Loretta Yates as the acquisition editor, Charvi Arora as the sponsoring editor, Rick Kughen as the development and copy editor, and Bri Achtman as the technical editor.

FROM FRANCESCO:

I'm 23, grown enough to live alone but young enough to feel my heart beating for my grandparents. Sadly enough, the number is smaller than a book ago. A warm thought goes to Grandpa Salvatore and a hug to Grandma Concetta and Grandma Leda: I love you. On an even more personal side, this book is for Gianfranco—friend, business partner, second father, and grandfather. He taught me how to do things right and forgot to teach me how to do it wrong. And this book is also for Michela, who is strong enough to pursue her own way—no matter what—and smart enough to choose a good path!

Introduction

We need men who can dream of things that never were, and ask, "why not?"

—John F. Kennedy, Speech to the Irish Parliament, June 1963

Today, the quest for data scientists is continuous, the data seems to be abundant, and cloud computing power is available. Is it the perfect world for the definitive triumph of machine learning? As we see things, we have all the necessary ingredients to cook up the "applied AI," but we still lack a clear and effective method for combining them.

The purpose of data science is, like the purpose of science, to show that something is possible. Data science, though, doesn't productionize solutions. That's the purpose of another branch of the machine learning universe—data engineering.

Companies are wildly looking for data scientists, but the outcome of a good data science team is typically a runnable model whose software quality is often that of a prototype rather than of a production-ready artifact. Algorithms are tightly bound to data, and data must be complete, clean, and balanced. Who's in charge of this part of the job is often unclear, and as a result, the job is often partially done at best. Yet, a data science team disconnected from the rest of the applied AI pipeline that makes it to production is still a due investment for a large organization whose business produces large quantities of data (such as energy utilities, financial institutions, and manufacturing farms). For smaller companies with significantly more limited budgets, the outcome of some applied data science can be cheaper to buy as a service.

From data science to production, there is usually a long way in between and a lot of work on data. Following are a few points to consider:

- How is data stored? On a daily or hourly basis?
- Should data be temporarily copied in some intermediate format?
- What kind of transformation is required for the model to work? How can you automate that?
- How is the performance of the model once deployed to production?
- How often is the model expected to need retrained to stay adherent to live data?
- If retraining is a frequent operation, how to automate any related tasks?

- What about collecting updated datasets, running the training, and deploying the up-to-date model?

The biggest issue we experience with machine learning models traces back to the data employed. In July 2021, the *MIT Technology Review* published an article about the impact of AI in facing the COVID pandemic. The article's bottom line is that many of the problems uncovered by a large review of aptly developed models are linked to the poor data quality that researchers used to develop their tools. Hence, nearly all tools were of nearly no effective use. This leads to a better understanding of the role of data engineering and data quality. Treating data via CSV sparse files is sufficient for probing an idea, but in order to build a robust infrastructure, you need to switch to some database (relational, NoSQL or graph) and some serious query language, and for this purpose, likely move beyond Python and enter into classic programming. Data science is not enough without serious programming and database skills. On the other hand, isn't searching for business-specific insights in data all that you do?

AI in general, and machine learning in particular, are now played as a tradeoff between commodity and direct solutions for vertical problems. Commoditized cloud services offer security, stability, and acceptable quality. They don't cover every possible scenario, though. But they're expanding, and more will expand in the near future.

All this is creating the environment for building the same old software but with more powerful tools. We're not just talking about the primitives of programming languages and classes from some frameworks. We're also talking about intelligent and predictive tools backed by machine learning algorithms and commoditized cloud (or containerized) services.

In this scenario, ML.NET acts as a perfect bridge between data engineering and commoditized data science, and it is fully integrated with the .NET framework. ML.NET commodities come in different forms: built-in algorithms for shallow learning, facilitated access to Azure cloud services, and integration with pre-trained models, such as Keras or TensorFlow networks.

Who Should Read This Book?

In our vision, if you adopt the .NET stack, then ML.NET is the perfect tool to do machine learning, whatever that ultimately means in terms of the internal gears of your chosen algorithms and models.

Hence, this book is for .NET developers willing (or needing) to approach the world of machine learning. It's ideal if you're a software developer adding data science and

machine learning skills to your arsenal. It's ideal if you're a data scientist willing to learn more about software beyond Python. Both categories, though, need to learn more and more about the other.

Who Should Not Read This Book?

This book discusses machine learning through the lenses of ML.NET, which is a platform-specific library. It is tailored more to data engineers and ML engineers than plain data scientists. To clarify, the core responsibilities of an ML engineer are to physically incorporate an externally trained model into client applications and perform the much more delicate task of supervising the building and training of a model based on data science specs. The book discusses the tools for doing this.

If you're not much interested in the actual productionizing of the machine learning solution, this book is probably not the best you can get. It doesn't open your mind to cutting-edge data science techniques, but it teaches you how to start leveraging what the ML.NET team has been doing for years—to integrate simple but effective machine learning solutions in .NET.

Organization of This Book

This book is divided into three sections.

- Chapters 1-3 provide a foundational overview of the library.
- Chapters 4-10 outline the dedicated tasks for data processing, training, and evaluation for common problems, such as regression, classification, ranking, anomaly detection, and more.
- Chapters 11-13 are dedicated to neural networks that might come into play when none of the shallow learning tasks is found to be suitable. Also, we include an overview of neural networks and an example of passport recognition that uses both commoditized Azure cognitive services and a handmade custom Keras network.

Lastly, Appendix A discusses model explainability.

System Requirements

You will need the following hardware and software to complete the practice exercises in this book:

- A computer running Windows 10, Linux, or macOS
- Visual Studio 2019, any edition, or superior; Visual Studio Code
- Internet connection to download software or chapter examples

Code Samples

All the code included in the book, including possible errata and extensions, can be found at MicrosoftPressStore.com/ProgrammingMLNET/downloads.

Errata, updates, & book support

We've made every effort to ensure the accuracy of this book and its companion content. You can access updates to this book—in the form of a list of submitted errata and their related corrections—at:

MicrosoftPressStore.com/ProgrammingMLNET/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit *http://www.MicrosoftPressStore.com/Support*.

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to *http://support.microsoft.com*.

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 2

An Architectural Perspective of ML.NET

"In mathematics, the art of asking questions is more valuable than solving problems."
—Georg Cantor, Doctoral thesis, 1867

As consumers, we continuously experience the pleasant effects of cognitive AI (for example, Amazon, Google, Apple, Microsoft, and Netflix). As people, we hope to see the same power applied to healthcare. In the general enterprise sector, where companies much smaller and rich than the web giants strive, the adoption of AI is slow and steady. This is precisely the feeling that descends from the point of intelligent software. Very few running businesses need the same level of cognitive AI we find in, say, Alexa or Cortana. All running businesses, though, would benefit from smarter features.

What is intelligent software, then?

Any software is statically designed to be aware of the context, but only intelligent software is designed to run while being dynamically aware of the business context. On the other hand, isn't this just what intelligence is supposed to be—the ability to acquire knowledge and turn it into expertise? In a nutshell, intelligence combines cognitive capabilities, including perception, memory, language, and reasoning and uses a specific learning approach to extract, transform, and store information. Turning all this into code requires ad hoc tools that are different from the basic logical equipment provided in any programming language or core framework.

Marketing departments love to generally identify these tools as artificial intelligence, specifically in the machine learning (ML) section. How do we do ML?

Most ML solutions today are built using tools from the Python ecosystem. It's a matter of convenience, however, rather than a matter of technological merit. In this chapter, we introduce the ML.NET platform—the .NET way to machine learning and the core topic of the book. More than just that, though, we will aim at providing an architectural perspective of a generic ML solution and presenting the reasons that, in our opinion, make ML.NET the right thing at the right time.

Life Beyond Python

In the collective imagination, ML is tightly coupled to using the Python programming language. Even from a surface-level look at job descriptions in tons of recruitment posts, this is clear. There are both historical and convenience reasons for languages like Python and C++ to be at the forefront of ML. However, there's no strict business reason or technical argument that prevents .NET and related languages (C# and F#) from being effectively used to build ML models.

Why Is Python So Popular in Machine Learning?

Python is an interpreted and object-oriented programming language created by Guido Van Rossum in the late 1980s with the declared goals of syntax minimalism and readability. The vision of Python as a programming language is that of a small core language engine with a large standard library and an easily extensible interpreter.

Python, which was born in a scientific environment, has become the de facto standard programming language for scientists to practice, explore, and experiment with numbers. In a way, it took the place that Fortran held in the 1960s and '70s. In the beginning, using Python in a hot new scientific field such as machine learning was a natural choice, and over time—given the natural extensibility of the language—it led to the creation of a vast ecosystem of dedicated libraries and tools. In turn, that reinforced the belief that using Python for building computational models was the best option.

Today, most data scientists find Python comfortable to use for machine learning projects, and that is probably because of a combination of the language's simplicity, the available tools, and plenty of examples. As developers, we also find Python comfortable to use to reshape data quickly to find the most appropriate format, test algorithms quickly, and explore different directions.

Once a clear path is outlined, the ML model must be trained and integrated into a runtime environment, its performance with live data must be monitored, and due changes must be applied and redeployed. It's the ML life cycle that is also known as MLOps. When you move away from experimenting with tools and libraries and look just for what enterprises need—working and maintainable code—Python shows structural limits. At the very minimum, it's yet another stack to integrate into .NET or Java stacks, which is how most business applications are written.

NOTE It's difficult to go from ML experimentation (often done in Python and via Notebooks) to deployment. In fact, according to a 2020 State of Enterprise Machine Learning report, only 22 percent of companies using machine learning have successfully deployed a machine learning model into production. See https://bit.ly/3y8BxOH

That's one of the advantages of ML.NET—.NET makes it super easy to bring projects to production.

Taxonomy of Python Machine Learning Libraries

The ecosystem of tools and libraries available in Python can be divided into five main areas: data manipulation, data visualization, numerical computing, model training, and neural networks. It's probably not an exhaustive list because many other libraries exist that perform other tasks and focus on some specific areas of machine learning, such as natural language processing and image recognition.

When using Python, the steps to build a machine learning pipeline are typically performed within the boundaries of a notebook. A notebook is a document created in a specific web or local interactive environment called Jupyter Notebook. (See *https://jupyter.org*.) Each notebook contains a combination of executable Python code, richly formatted text, data grids, charts, and pictures through which you build and share your development story. In some way, a notebook is comparable to a Visual Studio project solution.

In a notebook, you perform tasks such as data manipulation, plotting, and training, and you can use a number of predefined and battle-tested libraries.

Data Manipulation and Analysis

Pandas (*https://pandas.pydata.org*) is a library centered around the `DataFrame` object through which developers can load and manipulate in-memory tabular data. The object can import content from CSV files, text files, and SQL databases, and it provides core capabilities, such as conditional search, filtering, indexing and sorting, data slicing, grouping, and column operations (such as add, remove, and rename). The `DataFrame` object has built-in capability to flexibly reshape and pivot data and merge multiple frames. It also works well with time-series data.

The Pandas library is ideal for data preparation operations. Its integration with interactive notebooks enables you to perform on-the-fly testing of different configurations and data groupings.

Data Visualization

Matplotlib (*https://matplotlib.org*) is a helper library that isn't directly related to any of the common tasks of a machine learning pipeline, but it comes very handy to visually represent data during the various phases of the data preparation step or metrics obtained after evaluating trained models.

In general terms, it's a mere data visualization library built for Python code. It includes a 2D/3D rendering engine and supports common types of graphs, such as histograms, pie charts, and bar charts. Graphs are fully customizable in terms of line styles, font properties, axes, legends, and the like.

Numeric Computing

Because Python is a language that is largely used in scientific environments, it can't be without a bunch of extensions specifically designed for numerical computation. In this area, NumPy and SciPy are popular libraries, though they have slightly different capabilities.

NumPy (*https://www.numpy.org*) focuses on array operations and provides facilities to create, manipulate, and reshape one-dimensional and multidimensional arrays. Also, the library supplies linear algebra, Fourier transform, and random number operations.

SciPy (*https://scipy.org*) extends NumPy with polynomials, file I/O, image and signal processing, and more advanced features such as integration, interpolation, optimization, and statistics.

In the area of scientific computation, another Python library that is worth mentioning is Theano (*https://github.com/Theano/Theano*). Theano evaluates mathematical expressions based on multidimensional arrays, very efficiently making transparent use of the GPU. It also does symbolic differentiation for functions with one or more inputs.

Model Training

Though it was originally designed for data mining, today, scikit-learn (*https://scikit-learn.org*) is a library mainly focused on model training. It provides implementations of popular algorithms for regression, classification, and clustering. Also, scikit-learn provides methods for data preprocessing, such as dimensionality reduction, feature extraction, and normalization.

In a nutshell, scikit-learn is the Python foundation for shallow learning.

Neural Networks

Shallow learning is an area of machine learning that covers a broad array of fundamental problems such as regression and classification. Outside the realm of shallow learning, there are deep learning and neural networks. For building neural networks in Python, more specialized libraries exist.

TensorFlow (*https://www.tensorflow.org*) is probably the most popular library for training deep neural networks. It is part of a comprehensive framework that can be programmed at various levels. For example, you can use the high-level Keras API to build neural networks or manually build the desired topology and specify via code forward and activation steps, custom layers, and training loops. Overall, TensorFlow is an end-to-end machine learning platform providing facilities also to train and deploy.

Keras (*https://keras.io*) is probably the easiest way to get into the dazzling world of deep learning. It offers a very straightforward programming interface that at least comes in handy for quick prototyping. As mentioned, Keras can be used from within TensorFlow.

Yet another option is PyTorch, available at *https://pytorch.org*. PyTorch is the Python adaptation of an existing C-based library specialized in natural language processing and computer vision. Of the three neural network options, Keras is, by far, the ideal entry point and the tool of choice as long as it can deliver what you're looking for. PyTorch and TensorFlow do the same job of building sophisticated neural networks, but they use different approaches to the task. TensorFlow requires you to define the entire topology of the network before you can train it. In contrast, PyTorch follows a more agile approach and provides a more dynamic method for making changes to the graph. In some ways, their differences can be summarized as "waterfall versus agile." PyTorch is younger and doesn't have TensorFlow's huge community behind it.

End-to-End Solutions on Top of Python Models

With Python, you can easily find a way to build and train a machine learning model. Ultimately, a model is a binary file that must be loaded into a client application and invoked. Usually, a Java or .NET application serves as the client application for an ML model.

There are three main ways to consume a trained model:

- Hosting the trained model in a web service and making it accessible via a REST or gRPC API.
- Importing the trained model as a serialized file in the application and interacting with it through the programming interface provided by the infrastructure it is built upon (for example, TensorFlow or *scikit-learn*). This is possible only if the founding infrastructure provides bindings for the language to which the client application.
- The trained model is exposed via the new universal ONNX format, and the client application incorporates a wrapper for consuming ONNX binaries.

While the web service option is the most commonly used, a direct API that is specific for the client language of choice might seem the fastest way to consume a trained model. There are a couple of aspects to review, however:

- Using a direct API can prevent you from taking advantage of hardware acceleration and network distribution. In fact, if the API is hosted locally, any dedicated hardware (such as a GPU) is up to you. For this reason, if you want to invoke a graph at a very high rate in real-time, then you should consider using an ad hoc, hardware-accelerated cloud host.
- A binding for the specific trained model might not exist for the language of your choice. For example, TensorFlow natively supports Python, C , C++, Go, Swift, and Java.

Invoking a Python (or C++) library from within .NET code is not an unsurmountable technical issue. However, invoking a specific library, such as a trained machine learning model, is usually harder than calling a plain Python or C++ class.

In summary, an ML solution doesn't live on its own and must be framed in the context of an end-to-end business solution. Because many business solutions are based on the .NET stack, it was about time that a platform for training ML models natively in .NET came out. Using ML.NET, you can stay in the .NET ecosystem and don't have to deal with integrating Python into .NET applications.

Introducing ML.NET

First released in the spring of 2019, ML.NET is a free cross-platform and open-source .NET framework designed to build and train machine learning models and host them within .NET applications. See *https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet*.

ML.NET aims to provide the same set of capabilities that data scientists and developers can find in the Python ecosystem, as described earlier. Specifically built for .NET developers (for example, the API reflects common patterns of .NET frameworks and related development practices), ML.NET is built around the concept of the classic ML pipeline: collect data, set the algorithm, train, and deploy. In addition, any required programming steps sound familiar to anybody using the .NET framework and C# and F# programming languages.

The most interesting aspect of ML.NET is that it offers a quite pragmatic programming platform arranged around the idea of predefined *learning tasks*. The library comes equipped to make it relatively easy—even for machine learning newbies—to tackle common machine learning scenarios such as sentiment analysis, fraud detection, or price prediction as if they were just plain programming.

Compared to the pillars of the Python ecosystem presented earlier, ML.NET can be seen primarily as the counterpart of the scikit-learn model building library. The framework, however, also includes some basic facilities for data preparation and analysis that you can find in Pandas or NumPy. ML.NET also allows for the consumption of deep-learning models (specifically, TensorFlow and ONNX). Also, developers can train image classification and object detection models via Model Builder. It is remarkable, though, that the whole ML.NET library is built atop the tremendous power of the whole .NET Core framework.

The ML.NET framework is available as a set of NuGet packages. To start building models, you don't need more than that. However, as of version 16.6.1, Visual Studio also ships the Model Builder wizard that analyzes your input data and chooses the best available algorithm. We return to Model Builder in Chapter 3, "The Foundation of ML.NET."

The Learning Pipeline in ML.NET

A typical ML.NET solution is commonly articulated in three distinct projects:

- An application that orchestrates the steps of any machine learning pipeline: data collection, feature engineering, model selection, training, evaluation, and storage of the trained model
- A class library to contain the data types necessary to have the final model make a prediction once hosted in a client application. Note though that the input and output schemas do not strictly require its own project since these classes can be defined in the same project where training or consumption occurs
- A client application (website or a mobile or desktop application)

The orchestrator can be any type of .NET application, but the most natural choice is to have it coded as a console application.

It is worth noting, though, that this particular piece of code is not a one-off application that stops existing once it has given birth to the model. More often than not, the model has to be re-created many times before production and especially after being run in production. For this reason, the trainer application must be devised to be reusable and easy to configure and maintain.

Getting Started

As simple as it sounds, you can start by creating three such projects manually in Visual Studio and make them look like Figure 2-1. The figure presents three embryonal projects with many files and references missing but with sufficient details to deliver the big picture.

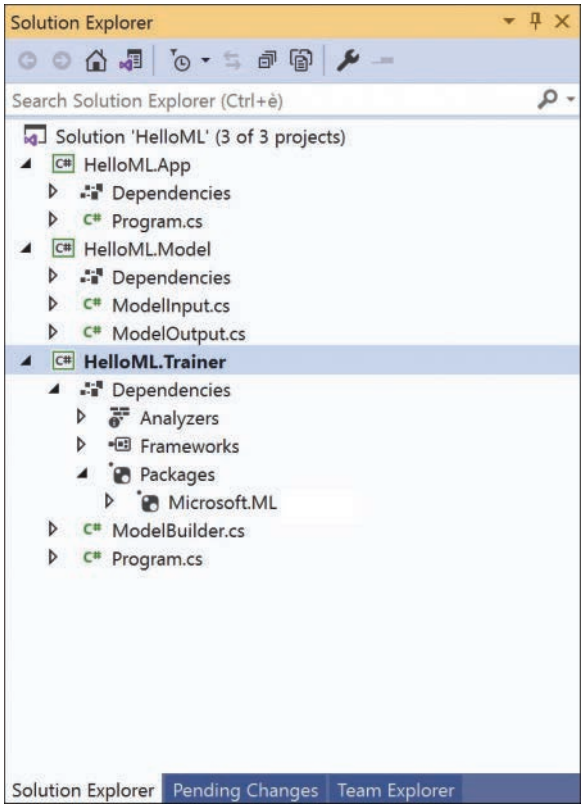


FIGURE 2-1 The skeleton of an ML.NET project in Visual Studio

Aside from the core references to the .NET framework of choice (whether 3.x or 5), the only additional piece you need to bring in is the `Microsoft.ML` NuGet package.

The package is not comprehensive, meaning that depending on what you intend to do, installing more packages might be necessary. However, the package is sufficient to get you started and enables you to experiment with the library. Let's focus on the trainer application and see what it takes to interact with the ML.NET library.

The Pipeline Entry Point

The entry point in the ML.NET pipeline is the `MLContext` object. You use it in much the same way you use the Entity Framework `DBContext` object or the connection object to a database library. You need to have an instance of this class shared across the various objects that participate in the building of the

model. A common practice employed in most tutorials—including the sample code generated by the aforementioned Model Builder wizard—is wrapping the model building workflow in a dedicated class, often just named `ModelBuilder`.

```
public static class ModelBuilder
{
    private static MLContext Context = new MLContext();

    // Main method
    public static void CreateModel(string inputDataFileName, string outputModelFileName)
    {
        // Load data

        // Build training pipeline

        // Train Model

        // QUICK evaluation of the model

        // Save the output model

   }
}
```

The instance of the `MLContext` class is global to the class methods, and the name of the files containing train data and the final output file are passed as arguments. The body of the `CreateModel` method (or whatever name you choose for it) develops around a few steps that involve more specific classes of the ML.NET library for activities such as data transformation, feature engineering, model selection, training, evaluation, and persistence.

Data Preparation

The ML.NET framework can read data from a variety of data sources (for example, a CSV-style text file, a binary file, or any `IEnumerable`-based object) and does that through the services of a few specialized loaders built around a specific interface—the `IDataView` interface, a flexible and efficient way of describing tabular data.

An `IDataView`-based loader works as a database cursor and supplies methods to navigate around the data set at any acceptable pace. It also provides an in-memory cache and methods to write the modified content back to disk. Here's a quick example:

```
// Create the context for the pipeline
Context = new MLContext();

// Load data into the pipeline via the DataView object
var dataView = Context.Data.LoadFromTextFile<ModelInput>(INPUT_DATA_FILE);
```

The sample code loads training data from the specified file and manages it as a collection of `ModelInput` types. Needless to say, the `ModelInput` type is a custom class that reflects the rows of

data loaded from the text file. The snippet below shows a sample `ModelInput` class. The `LoadColumn` attribute refers to the position of the CSV column to which the property binds.

```
public class ModelInput
{
  [LoadColumn(0)]
  public string Month { get; set; }

  [LoadColumn(1)]
  public float Sales { get; set; }
}
```

In the code, there's an even more relevant point to emphasize. The code makes a key assumption that the loaded data is already in a format that is acceptable for machine learning operations. More often than not, instead, some transformation is necessary to perform on top of available data—the most important of which is that all data must be numeric because algorithms can only read numbers. Here's a realistic scenario:

Your customer has a lot of data coming from a variety of data sources, whether timeline series, sparse Office documents, or database tables populated by online web frontends. In its raw format, the data—no matter the amount—might not be very usable. The appropriate format of the data depends on the desired result and the selected training algorithm. Therefore before mounting the final pipeline, a number of data transformation actions might be necessary, such as rendering data in columns, adding ad hoc feature columns, removing columns, aggregating and normalizing values, and adding density wherever possible. Depending on the context, these steps might be accomplished only once or every time the model is trained.

NOTE At first sight, it might seem that integrating data processing in the pipeline is a waste of time and that there's no value in doing it every time the model is built. More in general, instead, it is a matter of trade-off. We're usually talking about large quantities of data, and processing it to some intermediate format may be expensive. On the other hand, if the gap between raw and cleaned data is not that big, transforming data on each build of the model can deliver tremendous flexibility because you can change the transformation parameters as is convenient. It's a pure speed versus flexibility trade-off.

Trainers and Their Categorization

Training is the crucial phase of a machine learning pipeline. The training consists of picking an algorithm, setting in some way its configuration parameters, and running it repeatedly on a given (training) data set. The output of the training phase is the set of parameters that lead the algorithm to generate the best results. In the ML.NET jargon, the algorithm is called the *trainer*. More precisely, in ML.NET, a trainer is an algorithm plus a task. The same algorithm (say, L-BFGS) can be used for different tasks, such as regression or multiclass classification.

Table 2-1 lists some of the supported trainers grouped in a few tasks. We cover ML.NET tasks in more depth throughout the rest of the book and investigate their programming interface.

TABLE 2-1 ML.NET Tasks Related to Training

Task	Description
`AnomalyDetection`	Aims at detecting unexpected or unusual events or behaviors compared to the received training
`BinaryClassification`	Aims at classifying data in one of two categories
`Clustering`	Aims at splitting data in a number of possibly correlated groups without knowing which aspects could possibly make data items related
`Forecasting`	Addresses forecasting problems
`MulticlassClassification`	Aims at classifying data in three or more categories
`Ranking`	Addresses ranking problems
`Regression`	Aims at predicting the value of a data item

Figure 2-2 presents the list of ML.NET task objects as they show up through IntelliSense from the `MLContext` pipeline entry point.

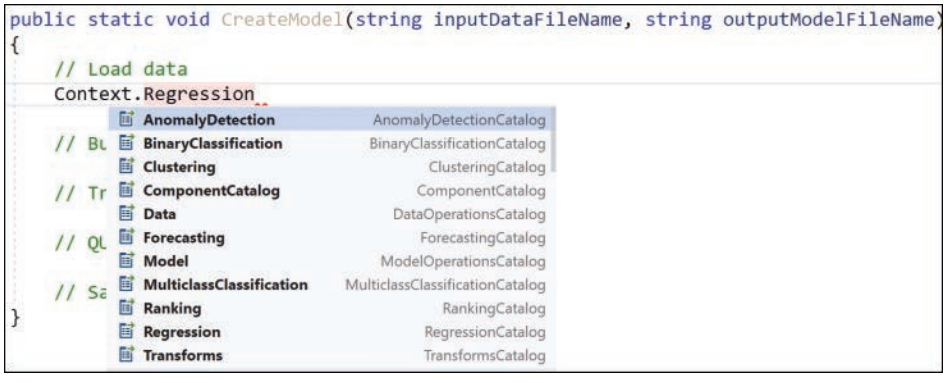


FIGURE 2-2 The list of ML.NET task objects

Each of the task objects listed in Table 2-1 has a `Trainers` property that lists the predefined algorithms supported by the framework. For example, for a prediction task, a good algorithm is the Online Gradient Descent algorithm.

```
var dataProcessPipeline = mlContext.Transforms.Text.FeaturizeText(...);
var trainer = mlContext.Regression.Trainers.OnlineGradientDescent(...);
var trainingPipeline = dataProcessPipeline.Append(trainer);
```

The code snippet selects an instance of the algorithm and then appends it to the data processing pipeline at the end of which the compiled model will come out. This short code contains the essence of the whole ML.NET programming model. This is the way in which the whole pipeline is built step by step and then run.

It is also worth noting that ML.NET supports several specific algorithms for each predefined task. In particular, for the regression task the ML.NET framework also supports the "Poisson Regression" and the "Stochastic Dual Coordinate Ascent" algorithms, and many more can be added to the project at any time through new NuGet packages.

Once the pipeline is built and fully configured, it is ready to run on the provided data. In this regard, the pipeline is a sort of abstract workflow that processes data in a way comparable to how in .NET LINQ queryable objects work on collections and data sets.

Once trained, a model is nothing more than the serialization of a computation graph that represents some sort of mathematical expression or, in some cases, a decision tree. The exact details of the expression depend on the algorithm and to some extent, the nature of the problem's category.

Model Training Executive Summary

Explaining the mechanics of model training is well beyond the scope of this book, which remains strictly focused on the ML.NET framework. However, at least a brief recap of what it means and how it works is in order. For an in-depth analysis, you can easily find online resources as well as books. In particular, you can take a look at our book *Introducing Machine Learning* (Microsoft Press, 2020), in which we mainly focus on the mathematics behind most problems and the algorithmic solutions discovered so far for each class.

Purpose of the Training Phase

Abstractly speaking, an algorithm is the sequence of steps that lead to the solution of a problem. In artificial intelligence, there are two main classes of problems: classification of entities and predictions. In each of these classes, we find several subclasses, such as ranking, forecasting, regression, anomaly detection, image and text analysis, and so forth.

In fact, the output of a machine learning pipeline is a software artifact made of an algorithm (or a chain of algorithms) whose parametric parts (settings and configurable elements) have been adjusted based on the provided training data. In other words, the output of a machine learning pipeline is the instance of an algorithm that, much like the instance of an object-oriented language class, has been initialized to hold a given configuration. The configuration to use for the instance of the algorithm is discovered during the training phase. The schema is outlined in Figure 2-3.

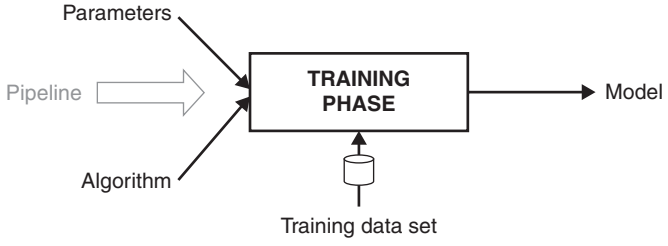


FIGURE 2-3 Overall schema of the training phase in a machine learning process

The Computation Graph

As mentioned, the model is a mathematical black-box—a computation graph—that takes input and computes an output. Input and output are lists of numbers, and in an object-oriented context—as in .NET—they are modeled using classes.

Figure 2-4 presents an abstract and concrete view of how a client application ultimately uses a trained model. Input data flows in, and the gears of the graph crunch numbers and produce a response for the application to deal with.

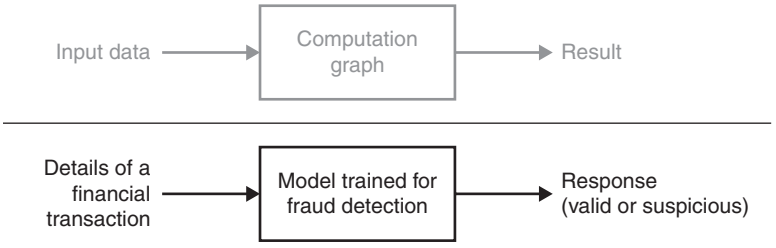


FIGURE 2-4 Overall schema of a trained model being used in production

For example, if you have a model trained to detect possible fraudulent transactions in a financial application, the graph in the model will be called to process a numeric representation of the transaction and produce some values that could be interpreted as to whether the transaction should be approved, denied, or just flagged for further investigation.

Performance of the Model

The term *machine learning* sounds fascinating, but it is not always fully representative of what really happens in a low-level ML framework such as ML.NET or Python's scikit-learn. At this level, the training phase just iteratively processes records in the training data set to minimize an error function.

- The function that produces values–the computation graph—is defined by the selected algorithm.
- The error function is yet another element added to the processing pipeline and also depends, to some extent, on the selected algorithm.
- The error function measures the distance between values produced by the graph on testing data and expected data embedded in the training data set.
- The graph enters the training phase with a default configuration that is updated if the measured error is too large for the desired goal.
- When a good compromise between speed and accuracy is reached, the training ends, and the current configuration of the graph is frozen and serialized for production.

The whole process develops iteratively in the training phase within the ML.NET framework. The steps are summarized in Figure 2-5.

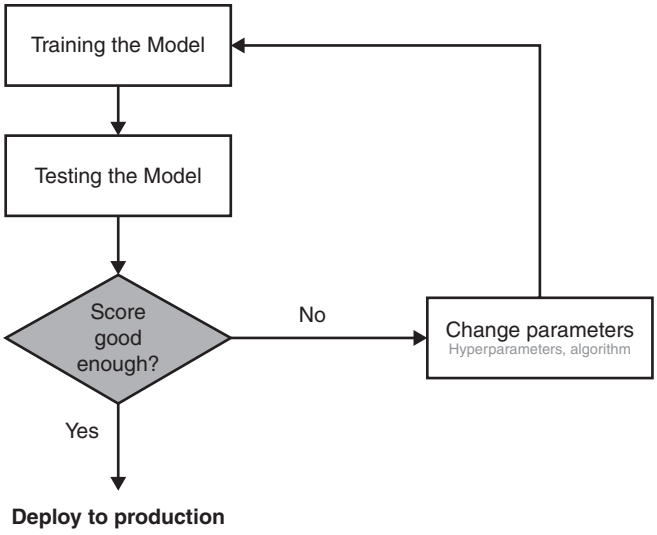


FIGURE 2-5 The generation of an ML model

It should be noted that the evaluation phase depicted here happens within the ML.NET framework and, more in general, within the boundaries of the ML framework of choice. The actual performance evaluated is obtainable on training data with a given set of algorithm parameters (referred to as hyperparameters) and internally computed coefficients.

This is not the same as measuring the performance of the model in production. At this stage, you only measure how good the model is at work on sample data. However, sample data is only expected to be a realistic snapshot of data the model will face once deployed to production. A more crucial evaluation phase will take place later and might even take to rebuilding the model based on different hyperparameters and—why not?—a different algorithm.

In ML.NET, the quality of the model during the training phase is measured using special components called *evaluators*.

A Look at Evaluators

An evaluator is a component that implements a given metric. Evaluation metrics are specific to the class of the algorithm and, in ML.NET, to the ongoing ML task. A good introduction on which evaluator is deemed appropriate for each ML.NET task can be found at

https://docs.microsoft.com/en-us/dotnet/machine-learning/resources/metrics.

A more in-depth discussion about the mathematical reasons that make each metric qualified for a given task can be found in the aforementioned book *Introducing Machine Learning*.

For example, for a prediction problem such as estimating the cost of a taxi ride (and in general for regression and ranking/recommendation problems), a key metric to consider is Squared Loss, also known as Mean Squared Error (MSE). This metric works by measuring how close a regression line is to

test expected values. For each input test value, the evaluator takes the distance between the computed and the expected response, squares it, and then calculates the mean. The squaring is applied to increase the relevance of larger differences.

Interestingly, Model Builder, which is embedded in Visual Studio, does some of the work for you. It first lets you choose the class of the problem (the ML task) and indicate the training data set. Based on that, it automatically selects a few matching algorithms, trains them, and measures the performance according to automatically selected metrics. Then it makes a final call and recommends how you should start coding your machine learning solution.

In general, there are a few things that could possibly go bad in a machine learning project:

- The selected algorithm (or algorithm hyperparameters) might not be the most appropriate to explore the given data set.
- The original data set needs more (or less) column transformation.
- The original data set is too small for the intended purpose.

As an example, Table 2-2 summarizes the scores of various algorithms selected by Model Builder for a prediction (regression) task.

TABLE 2-2 Multiple algorithms (and scores) for a sample regression task

Algorithm	Squared Loss	Absolute Loss	RSquared	RMS Loss
LightGbmRegression	4.49	0.38	0.9513	2.12
FastTreeTweedieRegression	4.70	0.44	0.9491	2.17
FastTreeRegression	4.83	0.41	0.9486	2.19
SdcaRegression	10.52	0.87	0.8845	3.27

After a test run of Model Builder, it shows that all featured algorithms ended up with good marks, but Model Builder ranked them in the order shown, thus suggesting we use the `LightGbmRegression` algorithm based on the evidence provided by metrics. Look in particular at the Squared Loss column. The score is acceptably good for the first three ranked algorithms and significantly worse for `SdcaRegression`. On the other hand, `SdcaRegression` is much faster to train. The golden rule of machine learning is that everything is a trade-off.

NOTE Another aspect to consider is that once the model goes to production—even with the best metrics—there might still be chances that things go wrong and the predictions are not in line with business expectations. When this happens, the odds are that an inadequate set of data rows was used for training. Inadequate, at least, in comparison to the real data the model was called to manage in production.

Consuming a Trained Model

At the end of the training phase, you have a model that contains instructions on which algorithm to run and which configuration to use. The model file is a zipped file in some serialization format. Note that a universal, interoperable format exists and is the ONNX format. ML.NET supports it.

As is, however, the model is a dead thing. To bring it to life, you need to load it in a runtime environment so that an API can be exposed to invoke the computation from the outside.

Making the Model Callable from the Outside

Once saved to a file—typically a ZIP file—the model is simply the flat description of a computation to be done on some input data. The first step is wrapping it into a framework engine that knows how to deserialize the graph and execute it on some input data.

ML.NET has a tailor-made set of methods ready to use. Here's the skeleton of the code you need to invoke a previously trained model in ML.NET.

```
public ModelOutput RunModel(string modelFileName, ModelInput input)
{
    var ml = new MLContext();
    var model = ml.Model.Load(modelFileName, out var schema);
    var engine = ml.Model.CreatePredictionEngine<ModelInput, ModelOutput>(model);
    return engine.Predict(input);
}
```

The sample function takes the path to the serialized model file and the input data to which a prediction is made. If the model estimates the cost of a taxi ride, then the class `ModelInput` describes the ride for which a quote is required. Typically, you will find that the model uses details such as distance, time of day, type of service requested, traffic conditions, area of the city involved, and whatever else is established. The `ModelOutput` class describes the output of the algorithm used for training. Usually, it's a simple C# class with just a few numeric properties. Here's an example:

```
public class ModelOutput
{
  public double Prediction { get; set; }
}
```

The ML.NET shell code creates an instance of a prediction engine that will carry the task of deserializing and executing the graph and return the calculated value. From the software developer's perspective, invoking an ML model is in no way different from calling a class library method.

Other Deployment Scenarios

Direct embedding of a trained model in the client application is one—and by far the simplest—deployment scenario. There are a couple of potentially sore points to emphasize.

One is the cost of deserializing the model and turning it into an executable computation graph for the runtime environment of choice—in this case, the .NET framework. The other is the (related) cost of setting up a prediction engine. Both operations can be quite expensive to perform if the client application is, say, a web application subject to thousands of calls per second. This is where an artifact like `PredictionEnginePool` comes in handly.

Therefore, the code snippet shown earlier is great for understanding the process but not necessarily good for production. More realistically, a company trains a model to expose a business-critical process as a service to various running software applications. This means that the model should be incorporated in a kind of web service, and proper layers of caching and balancing should be used to ensure proper performance.

In a nutshell, a trained model can be seen as a business black box to be used as a local class library, as a web service, or even as a microservice with its own storage and micro frontend. No option is favorable over the others, but all are feasible options for the architect to choose.

From Data Science to Programming

If you look at the trained model as an autonomous, black-boxed artifact integrated in a given type of software application, you should be able to see also the frontier between data science and programming. Data science contributes the model; programming makes it usable. Both aspects are strictly needed and unavoidable.

A trained model is nothing if not surrounded by a decent programming interface, whether in the form of a class library or a web service. To build an effective model, specific skills are required. First, you need domain expertise. Second, statistics and mathematics and the ability to discern between algorithms and metrics and interpret numbers are required. In extreme cases, the ability to develop new algorithms (including neural networks) or customize existing ones are also required. These skills very rarely belong to developers.

In much the same way, exposing a functional model requires due attention to the overall performance and scalability of the host application and care of the user experience. A taxi ride predictor model ultimately needs numbers to represent any sort of information. But you can hardly expect that people using the app on the go enter their destination through numbers. This is programming work.

In this scenario, ML.NET takes an interesting challenge: enabling developers to code their own machine learning tasks autonomously at least for relatively simple instances of problems and where a sharp precision is not the goal. This is just the ultimate purpose of ML tasks and AutoML—the engine that lies behind Model Builder. In this book, we deeply cover ML tasks but also dedicate a few final chapters to give problems a more real-world perspective. High precision, if necessary, comes at a cost!

Summary

ML.NET is now slated to become the reference platform for machine learning in the .NET space. It is mainly limited to shallow learning and doesn't offer direct support for building neural networks and deep learning (the support is only for consuming existing networks). On the other hand, also in the Python space there are libraries for shallow learning (scikit-learn) and libraries for building neural networks.

It is, instead, much more interesting and promising is the overall approach aimed at making machine learning easy to consume and relatively easy to design for developers. No developers will turn into expert data scientists overnight—not even after digesting the content of this book—but any savvy developer passionate about newer technologies and artificial intelligence would be incredibly comfortable with getting into the dazzling world of machine learning through ML.NET.

We've already mentioned something, but it helps to reinforce the concept: Although Python is quite popular among data scientists, there's no strict reason why machine learning models can't be developed and tested in .NET (or other languages, including Java and Go). It's all about the ecosystem and ease of use. ML.NET relies on the .NET Core infrastructure and Visual Studio.

Let's now go with a simple but not-so-trivial and complete example: taxi fare prediction. The next chapter includes a bit of feature engineering, feature selection, and, more importantly, a client web application.

Index

A

B

C

D

E

F

G

H

I

J-K

L

M

N

Q-R

S

T

U

V

W-X-Y-Z