Mastering .NET Machine Learning

.NET is one of the widely used platforms for developing applications. With the meteoric rise of machine learning, developers are now keen on finding out how to make their .NET applications smarter using machine learning.

Mastering .NET Machine Learning is packed with real-world examples to explain how to easily use machine learning techniques in your business applications. You will begin with an introduction to F# and prepare yourselves for machine learning using the .NET Framework. You will then learn how to write a simple linear regression model and, forming a base with the regression model, you will start using machine learning libraries available in .NET Framework such as Math.NET, numl, and Accord.NET with examples. Next, you are going to take a deep dive into obtaining, cleaning, and organizing your data. You will learn the implementation of k-means and PCA using Accord.NET and numl libraries. You will be using Neural Networks, AzureML, and Accord.NET to transform your application into a hybrid scientific application. You will also see how to deal with very large datasets using MBrace and deploy machine learning models to IoT devices so that the machine can learn and adapt on the fly.

Who this book is written for

This book is targeted at .NET developers who want to build complex machine learning systems. Some basic understanding of data science is required.

What you will learn from this book

- Write your own machine learning applications and experiments using the latest .NET Framework, including .NET Core 1.0
- Set up your business application to start using machine learning
- Accurately predict the future of your data using simple, multiple, and logistic regressions
- Discover hidden patterns using decision trees
- Acquire, prepare, and combine datasets to drive insights
- Optimize business throughput using Bayes Classifier
- Discover (more) hidden patterns using k-NN and Naive Bayes
- Discover (even more) hidden patterns using k-means and PCA
- Use Neural Networks to improve business decision making while using the latest ASP.NET technologies

Mastering .NET Machine Learning

Master the art of machine learning with .NET and gain insight into real-world applications

Jamie Dixon
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Welcome to Machine Learning Using the .NET Framework'
- A synopsis of the book’s content
- More information on Mastering .NET Machine Learning
About the Author

**Jamie Dixon** has been writing code for as long as he can remember and has been getting paid to do it since 1995. He was using C# and JavaScript almost exclusively until discovering F#, and now combines all three languages for the problem at hand. He has a passion for discovering overlooked gems in datasets and merging software engineering techniques to scientific computing. When he codes for fun, he spends his time using Phidgets, Netduinos, and Raspberry Pis or spending time in Kaggle competitions using F# or R.

Jamie is a bachelor of science in computer science and has been an F# MVP since 2014. He is the former chair of his town’s Information Services Advisory Board and is an outspoken advocate of open data. He is also involved with his local .NET User Group (TRINUG) with an emphasis on data analytics, machine learning, and the Internet of Things (IoT).

Jamie lives in Cary, North Carolina with his wonderful wife Jill and their three awesome children: Sonoma, Sawyer, and Sloan. He blogs weekly at jamessdixon.wordpress.com and can be found on Twitter at @jamie_dixon.
Preface

The .NET Framework is one of the most successful application frameworks in history. Literally billions of lines of code have been written on the .NET Framework, with billions more to come. For all of its success, it can be argued that the .NET Framework is still underrepresented for data science endeavors. This book attempts to help address this issue by showing how machine learning can be rapidly injected into the common .NET line of business applications. It also shows how typical data science scenarios can be addressed using the .NET Framework. This book quickly builds upon an introduction to machine learning models and techniques in order to build real-world applications using machine learning. While by no means a comprehensive study of predictive analytics, it does address some of the more common issues that data scientists encounter when building their models.

Many books about machine learning are written with every chapter centering around a dataset and how to implement a model on that dataset. While this is a good way to build a mental blueprint (as well as some code boilerplate), this book is going to take a slightly different approach. This book centers around introducing the same application for the line of business development and one common open data dataset for the scientific programmer. We will then introduce different machine techniques, depending on the business scenario. This means you will be putting on different hats for each chapter. If you are a line of business software engineer, Chapters 2, 3, 6, and 9 will seem like old hat. If you are a research analyst, Chapters 4, 7, and 10 will be very familiar to you. I encourage you to try all chapters, regardless of your background, as you will perhaps gain a new perspective that will make you more effective as a data scientist. As a final note, one word you will not find in this book is "simply". It drives me nuts when I read a tutorial-based book and the author says "it is simply this" or "simply do that". If it was simple, I wouldn't need the book. I hope you find each of the chapters accessible and the code samples interesting, and these two factors can help you immediately in your career.
What this book covers

Chapter 1, Welcome to Machine Learning Using the .NET Framework, contextualizes machine learning in the .NET stack, introduces some of the libraries that we will use throughout the book, and provides a brief primer to F#.

Chapter 2, AdventureWorks Regression, introduces the business that we will use in this book—AdventureWorks Bicycle company. We will then look at a business problem where customers are dropping orders based on reviews of the product. It looks at creating a linear regression by hand, using Math.NET and Accord.NET to solve this business problem. It then adds this regression to the line of business application.

Chapter 3, More AdventureWorks Regression, looks at creating a multiple linear regression and a logistic regression to solve different business problems at AdventureWorks. It will look at different factors that affect bike sales and then categorize potential customers into potential sales or potential lost leads. It will then implement the models to help our website convert potential lost leads into potential sales.

Chapter 4, Traffic Stops – Barking Up the Wrong Tree?, takes a break from AdventureWorks. You will put on your data scientist hat, use an open dataset of traffic stops, and see if we can understand why some people get a verbal warning and why others get a ticket at a traffic stop. We will use basic summary statistics and decision trees to help in understanding the results.

Chapter 5, Time Out – Obtaining Data, stops with introducing datasets and machine learning models and concentrates on one of the hardest parts of machine learning—obtaining and cleaning the data. We will look at using F# type providers as a very powerful language feature that can vastly speed up this process of "data munging".

Chapter 6, AdventureWorks Redux – k-NN and Naïve Bayes Classifiers, goes back to AdventureWorks and looks at a business problem of how to improve cross sales. We will implement two popular machine learning classification models, k-NN and Naïve Bayes, to see which is better at solving this problem.

Chapter 7, Traffic Stops and Crash Locations – When Two Datasets Are Better Than One, returns back to the traffic stop data and adds in two other open datasets that can be used to improve the predictions and gain new insights. The chapter will introduce two common unsupervised machine learning techniques: k-means and PCA.
Chapter 8, Feature Selection and Optimization, takes another break from introducing new machine learning models and looks at another key part of building machine learning models—selecting the right data for the model, preparing the data for the model, and introducing some common techniques to deal with outliers and other data abnormalities.

Chapter 9, AdventureWorks Production – Neural Networks, goes back to AdventureWorks and looks at how to improve bike production by using a popular machine learning technique called neural networks.

Chapter 10, Big Data and IoT, wraps up by looking at a more recent problem—how to build machine learning models on top of data that is characterized by massive volume, variability, and velocity. We will then look at how IoT devices can generate this big data and how to deploy machine learning models onto these devices so that they become self-learning.
Welcome to Machine Learning Using the .NET Framework

This is a book on creating and then using Machine Learning (ML) programs using the .NET Framework. Machine learning, a hot topic these days, is part of an overall trend in the software industry of analytics which attempts to make machines smarter. Analytics, though not really a new trend, has perhaps a higher visibility than in the past. This chapter will focus on some of the larger questions you might have about machine learning using the .NET Framework, namely: What is machine learning? Why should we consider it in the .NET Framework? How can I get started with coding?

What is machine learning?

If you check out on Wikipedia, you will find a fairly abstract definition of machine learning:

"Machine learning explores the study and construction of algorithms that can learn from and make predictions on data. Such algorithms operate by building a model from example inputs in order to make data-driven predictions or decisions, rather than following strictly static program instructions."

I like to think of machine learning as computer programs that produce different results as they are exposed to more information without changing their source code (and consequently needed to be redeployed). For example, consider a game that I play with the computer.
I show the computer this picture and tell it "Blue Circle". I then show it this picture and tell it "Red Circle". Next I show it this picture and say "Green Triangle."

Finally, I show it this picture and ask it "What is this?". Ideally the computer would respond, "Green Circle."

This is one example of machine learning. Although I did not change my code or recompile and redeploy, the computer program can respond accurately to data it has never seen before. Also, the computer code does not have to explicitly write each possible data permutation. Instead, we create models that the computer applies to new data. Sometimes the computer is right, sometimes it is wrong. We then feed the new data to the computer to retrain the model so the computer gets more and more accurate over time—or, at least, that is the goal.

Once you decide to implement some machine learning into your code base, another decision has to be made fairly early in the process. How often do you want the computer to learn? For example, if you create a model by hand, how often do you update it? With every new data row? Every month? Every year? Depending on what you are trying to accomplish, you might create a real-time ML model, a near-time model, or a periodic model. We will discuss the implications and implementations of each of these in several chapters in the book as different models lend themselves to different retraining strategies.

**Why .NET?**

If you are a Windows developer, using .NET is something you do without thinking. Indeed, a vast majority of Windows business applications written in the last 15 years use managed code—most of it written in C#. Although it is difficult to categorize millions of software developers, it is fair to say that .NET developers often come from nontraditional backgrounds. Perhaps a developer came to .NET from a BCSC degree but it is equally likely s/he started writing VBA scripts in Excel, moving up to Access applications, and then into VB.NET/C# applications. Therefore, most .NET developers are likely to be familiar with C#/VB.NET and write in an imperative and perhaps OO style.

The problem with this rather narrow exposure is that most machine learning classes, books, and code examples are in R or Python and very much use a functional style of writing code. Therefore, the .NET developer is at a disadvantage when acquiring machine learning skills because of the need to learn a new development environment, a new language, and a new style of coding before learning how to write the first line of machine learning code.
If, however, that same developer could use their familiar IDE (Visual Studio) and the same base libraries (the .NET Framework), they can concentrate on learning machine learning much sooner. Also, when creating machine learning models in .NET, they have immediate impact as you can slide the code right into an existing C#/VB.NET solution.

On the other hand, .NET is under-represented in the data science community. There are a couple of different reasons floating around for that fact. The first is that historically Microsoft was a proprietary closed system and the academic community embraced open source systems such as Linux and Java. The second reason is that much academic research uses domain-specific languages such as R, whereas Microsoft concentrated .NET on general purpose programming languages. Research that moved to industry took their language with them. However, as the researcher's role is shifted from data science to building programs that can work at real time that customers touch, the researcher is getting more and more exposure to Windows and Windows development. Whether you like it or not, all companies which create software that face customers must have a Windows strategy, an iOS strategy, and an Android strategy.

One real advantage to writing and then deploying your machine learning code in .NET is that you can get everything with one stop shopping. I know several large companies who write their models in R and then have another team rewrite them in Python or C++ to deploy them. Also, they might write their model in Python and then rewrite it in C# to deploy on Windows devices. Clearly, if you could write and deploy in one language stack, there is a tremendous opportunity for efficiency and speed to market.

What version of the .NET Framework are we using?

The .NET Framework has been around for general release since 2002. The base of the framework is the Common Language Runtime or CLR. The CLR is a virtual machine that abstracts much of the OS specific functionality like memory management and exception handling. The CLR is loosely based on the Java Virtual Machine (JVM). Sitting on top of the CLR is the Framework Class Library (FCL) that allows different languages to interoperate with the CLR and each other: the FCL is what allows VB.Net, C#, F#, and Iron Python code to work side-by-side with each other.
Since its first release, the .NET Framework has included more and more features. The first release saw support for the major platform libraries like WinForms, ASP.NET, and ADO.NET. Subsequent releases brought in things like Windows Communication Foundation (WCF), Language Integrated Query (LINQ), and Task Parallel Library (TPL). At the time of writing, the latest version is of the .Net Framework is 4.6.2.

In addition to the full-Monty .NET Framework, over the years Microsoft has released slimmed down versions of the .NET Framework intended to run on machines that have limited hardware and OS support. The most famous of these releases was the Portable Class Library (PCL) that targeted Windows RT applications running Windows 8. The most recent incantation of this is Universal Windows Applications (UWA), targeting Windows 10.

At Connect(); in November 2015, Microsoft announced GA of the latest edition of the .NET Framework. This release introduced the .Net Core 5. In January, they decided to rename it to .Net Core 1.0. .NET Core 1.0 is intended to be a slimmed down version of the full .NET Framework that runs on multiple operating systems (specifically targeting OS X and Linux). The next release of ASP.NET (ASP.NET Core 1.0) sits on top of .NET Core 1.0. ASP.NET Core 1.0 applications that run on Windows can still run the full .NET Framework.


In this book, we will be using a mixture of ASP.NET 4.0, ASP.NET 5.0, and Universal Windows Applications. As you can guess, machine learning models (and the theory behind the models) change with a lot less frequency than framework releases so the most of the code you write on .NET 4.6 will work equally well with PCL and .NET Core 1.0. Saying that, the external libraries that we will use need some time to catch up — so they might work with PCL but not with .NET Core 1.0 yet. To make things realistic, the demonstration projects will use .NET 4.6 on ASP.NET 4.x for existing (Brownfield) applications. New (Greenfield) applications will be a mixture of a UWA using PCL and ASP.NET 5.0 applications.
Why write your own?

It seems like all of the major software companies are pitching machine learning services such as Google Analytics, Amazon Machine Learning Services, IBM Watson, Microsoft Cortana Analytics, to name a few. In addition, major software companies often try to sell products that have a machine learning component, such as Microsoft SQL Server Analysis Service, Oracle Database Add-In, IBM SPSS, or SAS JMP. I have not included some common analytical software packages such as PowerBI or Tableau because they are more data aggregation and report writing applications. Although they do analytics, they do not have a machine learning component (not yet at least).

With all these options, why would you want to learn how to implement machine learning inside your applications, or in effect, write some code that you can purchase elsewhere? It is the classic build versus buy decision that every department or company has to make. You might want to build because:

- You really understand what you are doing and you can be a much more informed consumer and critic of any given machine learning package. In effect, you are building your internal skill set that your company will most likely prize. Another way to look at it, companies are not one tool away from purchasing competitive advantage because if they were, their competitors could also buy the same tool and cancel any advantage. However, companies can be one hire away or more likely one team away to truly have the ability to differentiate themselves in their market.

- You can get better performance by executing locally, which is especially important for real-time machine learning and can be implemented in disconnected or slow connection scenarios. This becomes particularly important when we start implementing machine learning with Internet of Things (IoT) devices in scenarios where the device has a lot more RAM than network bandwidth. Consider the Raspberry Pi running Windows 10 on a pipeline. Network communication might be spotty, but the machine has plenty of power to implement ML models.

- You are not beholden to any one vendor or company, for example, every time you implement an application with a specific vendor and are not thinking about how to move away from the vendor, you make yourself more dependent on the vendor and their inevitable recurring licensing costs. The next time you are talking to the CTO of a shop that has a lot of Oracle, ask him/her if they regret any decision to implement any of their business logic in Oracle databases. The answer will not surprise you. A majority of this book's code is written in F# — an open source language that runs great on Windows, Linux, and OS X.
• You can be much more agile and have much more flexibility in what you implement. For example, we will often re-train our models on the fly and when you write your own code, it is fairly easy to do this. If you use a third-party service, they may not even have API hooks to do model training and evaluation, so near-time model changes are impossible.

Once you decide to go native, you have a choice of rolling your own code or using some of the open source assemblies out there. This book will introduce both the techniques to you, highlight some of the pros and cons of each technique, and let you decide how you want to implement them. For example, you can easily write your own basic classifier that is very effective in production but certain models, such as a neural network, will take a considerable amount of time and energy and probably will not give you the results that the open source libraries do. As a final note, since the libraries that we will look at are open source, you are free to customize pieces of it—the owners might even accept your changes. However, we will not be customizing these libraries in this book.

Why open data?
Many books on machine learning use datasets that come with the language install (such as R or Hadoop) or point to public repositories that have considerable visibility in the data science community. The most common ones are Kaggle (especially the Titanic competition) and the UC Irvine's datasets. While these are great datasets and give a common denominator, this book will expose you to datasets that come from government entities. The notion of getting data from government and hacking for social good is typically called open data. I believe that open data will transform how the government interacts with its citizens and will make government entities more efficient and transparent. Therefore, we will use open datasets in this book and hopefully you will consider helping out with the open data movement.

Why F#?
As we will be on the .NET Framework, we could use either C#, VB.NET, or F#. All three languages have strong support within Microsoft and all three will be around for many years. F# is the best choice for this book because it is unique in the .NET Framework for thinking in the scientific method and machine learning model creation. Data scientists will feel right at home with the syntax and IDE (languages such as R are also functional first languages). It is the best choice for .NET business developers because it is built right into Visual Studio and plays well with your existing C#/VB.NET code. The obvious alternative is C#. Can I do this all in C#? Yes, kind of. In fact, many of the .NET libraries we will use are written in C#.
However, using C# in our code base will make it larger and have a higher chance of introducing bugs into the code. At certain points, I will show some examples in C#, but the majority of the book is in F#.

Another alternative is to forgo .NET altogether and develop the machine learning models in R and Python. You could spin up a web service (such as AzureML), which might be good in some scenarios, but in disconnected or slow network environments, you will get stuck. Also, assuming comparable machines, executing locally will perform better than going over the wire. When we implement our models to do real-time analytics, anything we can do to minimize the performance hit is something to consider.

A third alternative that the .NET developers will consider is to write the models in T-SQL. Indeed, many of our initial models have been implemented in T-SQL and are part of the SQL Server Analysis Server. The advantage of doing it on the data server is that the computation is as close as you can get to the data, so you will not suffer the latency of moving large amount of data over the wire. The downsides of using T-SQL are that you can't implement unit tests easily, your domain logic is moving away from the application and to the data server (which is considered bad form with most modern application architecture), and you are now reliant on a specific implementation of the database. F# is open source and runs on a variety of operating systems, so you can port your code much more easily.

Getting ready for machine learning

In this section, we will install Visual Studio, take a quick lap around F#, and install the major open source libraries that we will be using.

Setting up Visual Studio

To get going, you will need to download Visual Studio on a Microsoft Windows machine. As of this writing, the latest (free) version is Visual Studio 2015 Community. If you have a higher version already installed on your machine, you can skip this step. If you need a copy, head on over to the Visual Studio home page at https://www.visualstudio.com. Download the Visual Studio Community 2015 installer and execute it.
Welcome to Machine Learning Using the .NET Framework

Now, you will get the following screen:

Select **Custom** installation and you will be taken to the following screen:
Make sure Visual F# has a check mark next to it. Once it is installed, you should see Visual Studio in your Windows Start menu.

**Learning F#**

One of the great features about F# is that you can accomplish a whole lot with very little code. It is a very terse language compared to C# and VB.NET, so picking up the syntax is a bit easier. Although this is not a comprehensive introduction, this is going to introduce you to the major language features that we will use in this book. I encourage you to check out [http://www.tryfsharp.org/](http://www.tryfsharp.org/) or the tutorials at [http://fsharpforfunandprofit.com/](http://fsharpforfunandprofit.com/) if you want to get a deeper understanding of the language. With that in mind, let's create our 1st F# project:

2. Navigate to **File | New | Project** as shown in the following screenshot:
3. When the **New Project** dialog box appears, navigate the tree view to **Visual F# | Windows | Console Application**. Have a look at the following screenshot:

![New Project dialog box](image)

4. Give your project a name, hit **OK**, and the Visual Studio Template generator will create the following boilerplate:

```
1 // Learn more about F# at http://fsharp.org
2 // See the 'F# Tutorial' project for more help.
3
4 [EntryPoint]
5 let main argv =
6     printfn "%A" argv
7     0 // return an integer exit code
```

Although Visual Studio created a `Program.fs` file that creates a basic console .exe application for us, we will start learning about F# in a different way, so we are going to ignore it for now.
5. Right-click in the **Solution Explorer** and navigate to **Add | New Item**.

6. When the **Add New Item** dialog box appears, select **Script File**.
The Script1.fsx file is then added to the project.

7. Once Script1.fsx is created, open it up, and enter the following into the file:
   ```fsharp
   let x = "Hello World"
   ```

8. Highlight that entire row of code, right-click and select **Execute In Interactive** (or press **Alt + Enter**):
And the **F# Interactive** console will pop up and you will see this:

![F# Interactive Console](image)

The F# Interactive is a type of REPL, which stands for Read-Evaluate-Print-Loop. If you are a .NET developer who has spent any time in SQL Server Management Studio, the F# Interactive will look very familiar to the Query Analyzer where you enter your code at the top and see how it executes at the bottom. Also, if you are a data scientist using R Studio, you are very familiar with the concept of a REPL. I have used the words REPL and FSI interchangeably in this book.

There are a couple of things to notice about this first line of F# code you wrote. First, it looks very similar to C#. In fact, consider changing the code to this:

```csharp
var x = "Hello World";
```

It would be perfectly valid C#. Note that the red squiggly line, showing you that the F# compiler certainly does not think this is valid.

Going back to the correct code, notice that type of `x` is not explicitly defined. F# uses the concept of inferred typing so that you don't have to write the type of the values that you create. I used the term *value* deliberately because unlike variables, which can be assigned in C# and VB.NET, values are immutable; once bound, they can never change. Here, we are permanently binding the name `x` to its value, *Hello World*. This notion of immutability might seem constraining at first, but it has profound and positive implications, especially when writing machine learning models.
Welcome to Machine Learning Using the .NET Framework

With our basic program idea proven out, let's move it over to a compilable assembly; in this case, an .exe that targets the console. Highlight the line that you just wrote, press Ctrl + C, and then open up Program.fs. Go into the code that was generated and paste it in:

```fsharp
let main argv =
  printfn "%A" argv
  let x = "Hello World"
  0 // return an integer exit code
```

Downloading the example code
You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the SUPPORT tab at the top.
- Click on Code Downloads & Errata.
- Enter the name of the book in the Search box.
- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipget / iZip / UnrarX for Mac
- 7-Zip / PeaZip for Linux

Then, add the following lines of code around what you just added:

```fsharp
// Learn more about F# at http://fsharp.org
// See the 'F# Tutorial' project for more help.
ope System

let main argv =
  printfn "%A" argv
```

---

[14]
let x = "Hello World"
Console.WriteLine(x)
let y = Console.ReadKey()
0 // return an integer exit code

Press the Start button (or hit F5) and you should see your program run:

You will notice that I had to bind the return value from Console.ReadKey() to y. In C# or VB.NET, you can get away with not handling the return value explicitly. In F#, you are not allowed to ignore the returned values. Although some might think this is a limitation, it is actually a strength of the language. It is much harder to make a mistake in F# because the language forces you to address execution paths explicitly versus accidentally sweeping them under the rug (or into a null, but we'll get to that later).

In any event, let's go back to our script file and enter in another line of code:

```fsharp
let ints = [1;2;3;4;5;6]
```

If you send that line of code to the REPL, you should see this:

```fsharp
val ints : int [] = [|1; 2; 3; 4; 5; 6|]
```

This is an array, as if you did this in C#:

```csharp
var ints = new[] {1,2,3,4,5,6};
```

Notice that the separator is a semicolon in F# and not a comma. This differs from many other languages, including C#. The comma in F# is reserved for tuples, not for separating items in an array. We'll discuss tuples later.

Now, let's sum up the values in our array:

```fsharp
let summedValue = ints |> Array.sum
```

While sending that line to the REPL, you should see this:

```fsharp
val summedValue : int = 21
```
There are two things going on. We have the |> operator, which is a pipe forward operator. If you have experience with Linux or PowerShell, this should be familiar. However, if you have a background in C#, it might look unfamiliar. The pipe forward operator takes the result of the value on the left-hand side of the operator (in this case, ints) and pushes it into the function on the right-hand side (in this case, sum).

The other new language construct is Array.sum. Array is a module in the core F# libraries, which has a series of functions that you can apply to your data. The function sum, well, sums the values in the array, as you can probably guess by inspecting the result.

So, now, let’s add a different function from the Array type:

```fsharp
let multiplied = ints |> Array.map (fun i -> i * 2)
```

If you send it to the REPL, you should see this:

```fsharp
val multiplied : int [] = [|2; 4; 6; 8; 10; 12|]
```

Array.map is an example of a high ordered function that is part of the Array type. Its parameter is another function. Effectively, we are passing a function into another function. In this case, we are creating an anonymous function that takes a parameter i and returns i * 2. You know it is an anonymous function because it starts with the keyword fun and the IDE makes it easy for us to understand that by making it blue. This anonymous function is also called a lambda expression, which has been in C# and VB.NET since .Net 3.5, so you might have run across it before. If you have a data science background using R, you are already quite familiar with lambdas.

Getting back to the higher-ordered function Array.map, you can see that it applies the lambda function against each item of the array and returns a new array with the new values.

<table>
<thead>
<tr>
<th>ints</th>
<th>Array.map(fun i -&gt; i * 2)</th>
<th>multiplied</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 * 2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2 * 2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3 * 2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>4 * 2</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>5 * 2</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>6 * 2</td>
<td>12</td>
</tr>
</tbody>
</table>
We will be using \texttt{Array.map} (and its more generic kin \texttt{Seq.map}) a lot when we start implementing machine learning models as it is the best way to transform an array of data. Also, if you have been paying attention to the buzz words of map/reduce when describing big data applications such as Hadoop, the word map means exactly the same thing in this context. One final note is that because of immutability in F#, the original array is not altered, instead, multiplied is bound to a new array.

Let's stay in the script and add in another couple more lines of code:

\begin{verbatim}
let multiplyByTwo x =
  x * 2
\end{verbatim}

If you send it to the REPL, you should see this:

\begin{verbatim}
val multiplyByTwo : x:int -> int
\end{verbatim}

These two lines created a named function called \texttt{multiplyByTwo}. The function that takes a single parameter \(x\) and then returns the value of the parameter multiplied by 2. This is exactly the same as our anonymous function we created earlier in-line that we passed into the \texttt{map} function. The syntax might seem a bit strange because of the \texttt{->} operator. You can read this as, "the function \texttt{multiplyByTwo} takes in a parameter called \(x\) of type \(\text{int}\) and returns an \(\text{int}\)."

Note three things here. Parameter \(x\) is inferred to be an \(\text{int}\) because it is used in the body of the function as multiplied to another \(\text{int}\). If the function reads \(x \times 2.0\), the \(x\) would have been inferred as a float. This is a significant departure from C# and VB.NET but pretty familiar for people who use R. Also, there is no return statement for the function, instead, the final expression of any function is always returned as the result. The last thing to note is that whitespace is important so that the indentation is required. If the code was written like this:

\begin{verbatim}
let multiplyByTwo(x) =
  x * 2
\end{verbatim}

The compiler would complain:

\begin{verbatim}
Script1.fsx(8,1): warning FS0058: Possible incorrect indentation: this token is offside of context started at position (7:1).
\end{verbatim}

Since F# does not use curly braces and semicolons (or the end keyword), such as C# or VB.NET, it needs to use something to separate code. That separation is whitespace. Since it is good coding practice to use whitespace judiciously, this should not be very alarming to people having a C# or VB.NET background. If you have a background in R or Python, this should seem natural to you.
Since `multiplyByTwo` is the functional equivalent of the lambda created in `Array.map (fun i -> i * 2)`, we can do this if we want:

```ml
let multiplied' = ints |> Array.map (fun i -> multiplyByTwo i)
```

If you send it to the REPL, you should see this:

```ml
val multiplied' : int [] = [|2; 4; 6; 8; 10; 12|]
```

Typically, we will use named functions when we need to use that function in several places in our code and we use a lambda expression when we only need that function for a specific line of code.

There is another minor thing to note. I used the tick notation for the value multiplied when I wanted to create another value that was representing the same idea. This kind of notation is used frequently in the scientific community, but can get unwieldy if you attempt to use it for a third or even fourth (`multiplied''`) representation.

Next, let’s add another named function to the REPL:

```ml
let isEven x =
    match x % 2 = 0 with
    | true -> "even"
    | false -> "odd"

isEven 2
isEven 3
```

If you send it to the REPL, you should see this:

```ml
val isEven : x:int -> string
```

This is a function named `isEven` that takes a single parameter `x`. The body of the function uses a pattern-matching statement to determine whether the parameter is odd or even. When it is odd, then it returns the string `odd`. When it is even, it returns the string `even`.

There is one really interesting thing going on here. The match statement is a basic example of pattern matching and it is one of the coolest features of F#. For now, you can consider the match statement much like the switch statement that you may be familiar within R, Python, C#, or VB.NET, but we will see how it becomes much more powerful in the later chapters. I would have written the conditional logic like this:

```ml
let isEven' x =
    if x % 2 = 0 then "even" else "odd"
```
But I prefer to use pattern matching for this kind of conditional logic. In fact, I will attempt to go through this entire book without using an if...then statement.

With isEven written, I can now chain my functions together like this:

```fsharp
let multipliedAndIsEven =
    ints
    |> Array.map (fun i -> multiplyByTwo i)
    |> Array.map (fun i -> isEven i)
```

If you send it to REPL, you should see this:

```fsharp
val multipliedAndIsEven : string [] =
    ["even"; "even"; "even"; "even"; "even"; "even"]
```

In this case, the resulting array from the first pipe `Array.map (fun i -> multiplyByTwo i)` gets sent to the next function `Array.map (fun i -> isEven i)`. This means we might have three arrays floating around in memory: `ints` which is passed into the first pipe, the result from the first pipe that is passed into the second pipe, and the result from the second pipe. From your mental model point of view, you can think about each array being passed from one function into the next. In this book, I will be chaining pipe forwards frequently as it is such a powerful construct and it perfectly matches the thought process when we are creating and using machine learning models.

You now know enough F# to get you up and running with the first machine learning models in this book. I will be introducing other F# language features as the book goes along, but this is a good start. As you will see, F# is truly a powerful language where a simple syntax can lead to very complex work.

**Third-party libraries**

The following are a few third-party libraries that we will cover in our book later on.

**Math.NET**

Math.NET is an open source project that was created to augment (and sometimes replace) the functions that are available in `System.Math`. Its home page is http://www.mathdotnet.com/. We will be using Math.NET's Numerics and Symbolics namespaces in some of the machine learning algorithms that we will write by hand. A nice feature about Math.NET is that it has strong support for F#.
Accord.NET
Accord.NET is an open source project that was created to implement many common machine learning models. Its home page is http://accord-framework.net/. Although the focus of Accord.NET was for computer vision and signal processing, we will be using Accord.NET extensively in this book as it makes it very easy to implement algorithms in our problem domain.

Numl
Numl is an open source project that implements several common machine learning models as experiments. Its home page is http://numl.net/. Numl is newer than any of the other third-party libraries that we will use in the book, so it may not be as extensive as the other ones, but it can be very powerful and helpful in certain situations. We will be using Numl in several chapters of the book.

Summary
We covered a lot of ground in this chapter. We discussed what machine learning is, why you want to learn about it in the .NET stack, how to get up and running using F#, and had a brief introduction to the major open source libraries that we will be using in this book. With all this preparation out of the way, we are ready to start exploring machine learning.

In the next chapter, we will apply our newly found F# skills to create a simple linear regression to see if we can help AdventureWorks improve their sales.
Where to buy this book

You can buy Mastering .NET Machine Learning from the Packt Publishing website. Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers. Click here for ordering and shipping details.