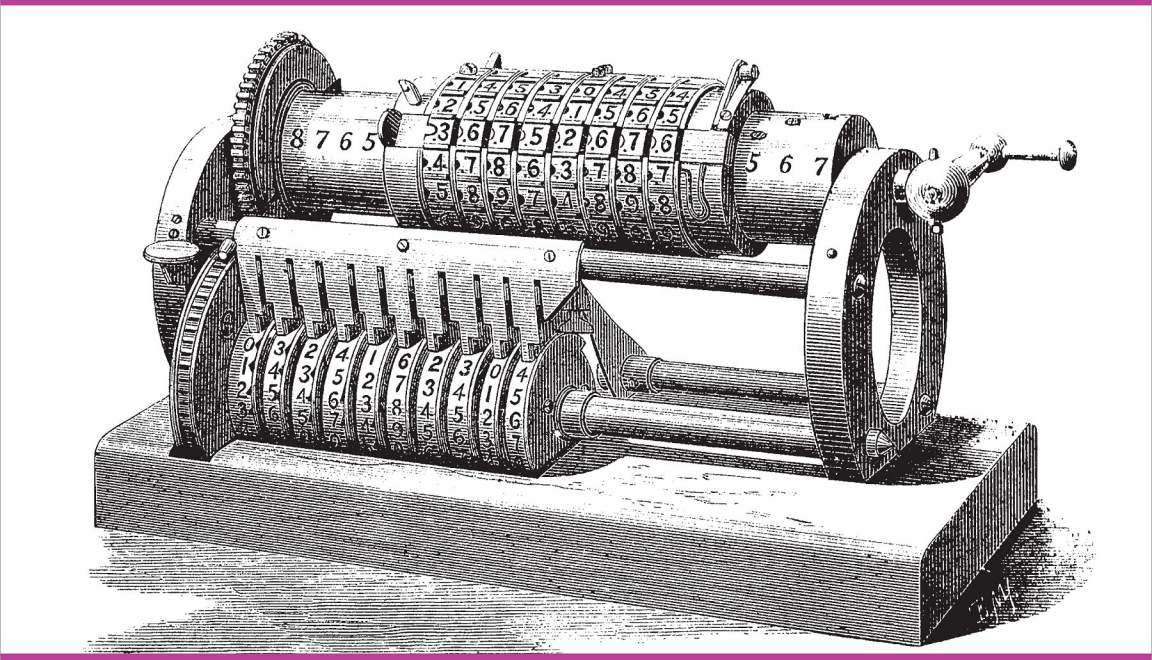
O'REILLY®

Analyzing and Visualizing Data with F#



Tomas Petricek

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Analyzing and Visualizing Data with F#

Tomas Petricek

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Analyzing and Visualizing Data with F#
by Tomas Petricek

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com* .

Editor: Brian MacDonald
Production Editor: Nicholas Adams
Copyeditor: Sonia Saruba
Proofreader: Nicholas Adams

Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

October 2015: First Edition

Revision History for the First Edition
2015-10-15: First Release

978-1-491-93953-6

[LSI]

Table of Contents

Acknowledgements

This report would never exist without the amazing F# open source community that creates and maintains many of the libraries used in the report. It is impossible to list all the contributors, but let me say thanks to Gustavo Guerra, Howard Mansell, and Taha Hachana for their work on F# Data, R type provider, and XPlot, and to Steffen Forkmann for his work on the projects that power much of the F# open source infrastructure. Many thanks to companies that support the F# projects, including Microsoft and BlueMountain Capital.

I would also like to thank Mathias Brandewinder who wrote many great examples using F# for machine learning and whose blog post about clustering with F# inspired the example in Chapter 4. Last but not least, I'm thankful to Brian MacDonald, Heather Scherer from O'Reilly, and the technical reviewers for useful feedback on early drafts of the report.

CHAPTER 1
Accessing Data with Type Providers

Working with data was not always as easy as nowadays. For example, processing the data from the decennial 1880 US Census took eight years. For the 1890 census, the United States Census Bureau hired Herman Hollerith, who invented a number of devices to automate the process. A *pantograph punch* was used to punch the data on punch cards, which were then fed to the *tabulator* that counted cards with certain properties, or to the *sorter* for filtering. The census still required a large amount of clerical work, but Hollerith's machines sped up the process eight times to just one year.[1]

These days, filtering and calculating sums over hundreds of millions of rows (the number of forms received in the 2010 US Census) can take seconds. Much of the data from the US Census, various Open Government Data initiatives, and from international organizations like the World Bank is available online and can be analyzed by anyone. Hollerith's *tabulator* and *sorter* have become standard library functions in many programming languages and data analytics libraries.

1 Hollerith's company later merged with three other companies to form a company that was renamed International Business Machines Corporation (IBM) in 1924. You can find more about Hollerith's machines in Mark Priestley's excellent book, *A Science of Operations (Springer)*.

Making data analytics easier no longer involves building new physical devices, but instead involves creating better software tools and programming languages. So, let's see how the F# language and its unique features like *type providers* make the task of modern data analysis even easier!

Data Science Workflow

Data science is an umbrella term for a wide range of fields and disciplines that are needed to extract *knowledge* from *data*. The typical data science workflow is an iterative process. You start with an initial idea or research question, get some data, do a quick analysis, and make a visualization to show the results. This shapes your original idea, so you can go back and adapt your code. On the technical side, the three steps include a number of activities:

- **Accessing data.** The first step involves connecting to various data sources, downloading CSV files, or calling REST services. Then we need to combine data from different sources, align the data correctly, clean possible errors, and fill in missing values.
- **Analyzing data.** Once we have the data, we can calculate basic statistics about it, run machine learning algorithms, or write our own algorithms that help us explain what the data means.
- **Visualizing data.** Finally, we need to present the results. We may build a chart, create interactive visualization that can be published, or write a report that represents the results of our analysis.

If you ask any data scientist, she'll tell you that *accessing data* is the most frustrating part of the workflow. You need to download CSV files, figure out what columns contain what values, then determine how missing values are represented and parse them. When calling REST-based services, you need to understand the structure of the returned JSON and extract the values you care about. As you'll see in this chapter, the data access part is largely simplified in F# thanks to *type providers* that integrate external data sources directly into the language.

Why Choose F# for Data Science?

There are a lot of languages and tools that can be used for data science. Why should you choose F#? A two-word answer to the question is *type providers*. However, there are other reasons. You'll see all of them in this report, but here is a quick summary:

- **Data access.** With type providers, you'll never need to look up column names in CSV files or country codes again. Type providers can be used with many common formats like CSV, JSON, and XML, but they can also be built for a specific data source like Wikipedia. You will see type providers in this and the next chapter.
- **Correctness.** As a functional-first language, F# is excellent at expressing algorithms and solving complex problems in areas like machine learning. As you'll see in Chapter 3, the F# type system not only prevents bugs, but also helps us understand our code.
- **Efficiency and scaling.** F# combines the simplicity of Python with the efficiency of a JIT-based compiled language, so you do not have to call external libraries to write fast code. You can also run F# code in the cloud with the MBrace project. We won't go into details, but I'll show you the idea in Chapter 3.
- **Integration.** In Chapter 4, we see how type providers let us easily call functions from R (a statistical software with rich libraries). F# can also integrate with other ecosystems. You get access to a large number of .NET and Mono libraries, and you can easily interoperate with FORTRAN and C.

Enough talking, let's look at some code! To set the theme for this chapter, let's look at the forecasted temperatures around the world. To do this, we combine data from two sources. We use the World Bank[2] to access information about countries, and we use the Open Weather Map[3] to get the forecasted temperature in all the capitals of all the countries in the world.

---

2 The World Bank is an international organization that provides loans to developing countries. To do so effectively, it also collects large numbers of development and financial indicators that are available through a REST API at *http://data.worldbank.org/*.

3 See *http://openweathermap.org/*.

Getting Data from the World Bank

To access information about countries, we use the World Bank type provider. This is a type provider for a specific data source that makes accessing data as easy as possible, and it is a good example to start with. Even if you do not need to access data from the World Bank, this is worth exploring because it shows how simple F# data access can be. If you frequently work with another data source, you can create your own type provider and get the same level of simplicity.

The World Bank type provider is available as part of the F# Data library.[4] We could start by referencing just F# Data, but we will also need a charting library later, so it is better to start by referencing FsLab, which is a collection of .NET and F# data science libraries. The easiest way to get started is to download the FsLab basic template from *http://fslab.org/download*.

The FsLab template comes with a sample script file (a file with the `.fsx` extension) and a project file. To download the dependencies, you can either build the project in Visual Studio or Xamarin Studio, or you can invoke the Paket package manager directly. To do this, run the Paket bootstrapper to download Paket itself, and then invoke Paket to install the packages (on Windows, drop the `mono` prefix):

```
mono .paket\paket.bootstrapper.exe
mono .paket\paket.exe install
```

NuGet Packages and Paket

In the F# ecosystem, most packages are available from the NuGet gallery. NuGet is also the name of the most common package manager that comes with typical .NET distributions. However, the FsLab templates use an alternative called Paket instead.

Paket has a number of benefits that make it easier to use with data science projects in F#. It uses a single `paket.lock` file to keep version numbers of all packages (making updates to new versions easier), and it does not put the version number in the name of the

4 See *http://fslab.org/FSharp.Data*.

folder that contains the packages. This works nicely with F# and the `#load` command, as you can see in the snippet below.

Once you have all the packages, you can replace the sample script file with the following simple code snippet:

```
#load "packages/FsLab/FsLab.fsx"
open FSharp.Data

let wb = WorldBankData.GetDataContext()
```

The first line loads the `FsLab.fsx` file, which comes from the FsLab package, and loads all the libraries that are a part of FsLab, so you do not have to reference them one by one. The last line uses `GetDataContext` to to create an instance that we'll need in the next step to fetch some data.

The next step is to use the World Bank type provider to get some data. Assuming everything is set up in your editor, you should be able to type `wb.Countries` followed by `.` (a period) and get auto-completion on the country names as shown in Figure 1-1. This is not a magic! The country names, are just ordinary properties. The trick is that they are generated on the fly by the type provider based on the schema retrieved from the World Bank.

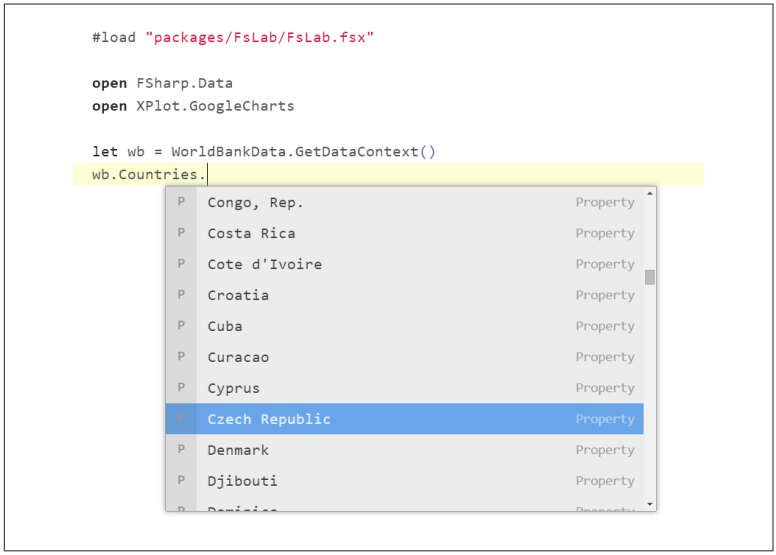


Figure 1-1. Atom editor providing auto-completion on countries

Feel free to explore the World Bank data on your own! The following snippet shows two simple things you can do to get the capital city and the total population of the Czech Republic:

```
wb.Countries.``Czech Republic``.CapitalCity
wb.Countries.``Czech Republic``.Indicators
  .`` CO2 emissions (kt)``.[2010]
```

On the first line, we pick a country from the World Bank and look at one of the basic properties that are available directly on the country object. The World Bank also collects numerous indicators about the countries, such as GDP, school enrollment, total population, CO_2 emissions, and thousands of others. In the second example, we access the CO_2 emissions using the `Indicators` property of a country. This returns a *provided* object that is generated based on the indicators that are available in the World Bank database. Many of the properties contain characters that are not valid identifiers in F# and are wrapped in ``. As you can see in the example, the names are quite complex. Fortunately, you are not expected to figure out and remember the names of the properties because the F# editors provide auto-completion based on the type information.

A World Bank indicator is returned as an object that can be turned into a list using `List.ofSeq`. This list contains values for all of the years for which a value is available. As demonstrated in the example, we can also invoke the indexer of the object using `.[2010]` to find a value for a specific year.

F# Editors and Auto-complete

F# is a statically typed language and the editors have access to a lot of information that is used to provide advanced IDE features like auto-complete and tooltips. Type providers also heavily rely on auto-complete; if you want to use them, you'll need an editor with good F# support.

Fortunately, a number of popular editors have good F# support. If you prefer editors, you can use Atom from GitHub (install the `language-fsharp` and `atom-fsharp` packages) or Emacs with `fsharp-mode`. If you prefer a full IDE, you can use Visual Studio (including the free edition) on Windows, or MonoDevelop (a free version of Xamarin Studio) on Mac, Linux, or Windows. For more

information about getting started with F# and up-to-date editor information, see the "Use" pages on *http://fsharp.org*.

The typical data science workflow requires a quick feedback loop. In F#, you get this by using F# Interactive, which is the F# REPL. In most F# editors, you can select a part of the source code and press Alt+Enter (or Ctrl+Enter) to evaluate it in F# Interactive and see the results immediately.

The one thing to be careful about is that you need to load all dependencies *first*, so in this example, you first need to evaluate the contents of the first snippet (with `#load`, `open`, and `let wb = ...`), and then you can evaluate the two commands from the above snippets to see the results. Now, let's see how we can combine the World Bank data with another data source.

Calling the Open Weather Map REST API

For most data sources, because F# does not have a specialized type provider like for the World Bank, we need to call a REST API that returns data as JSON or XML.

Working with JSON or XML data in most statically typed languages is not very elegant. You either have to access fields by name and write `obj.GetField<int>("id")`, or you have to define a class that corresponds to the JSON object and then use a reflection-based library that loads data into that class. In any case, there is a lot of boilerplate code involved!

Dynamically typed languages like JavaScript just let you write `obj.id`, but the downside is that you lose all compile-time checking. Is it possible to get the simplicity of dynamically typed languages, but with the static checking of statically typed languages? As you'll see in this section, the answer is yes!

To get the weather forecast, we'll use the Open Weather Map service. It provides a daily weather forecast endpoint that returns weather information based on a city name. For example, if we request *http://api.openweathermap.org/data/2.5/forecast/daily?q=Cambridge*, we get a JSON document that contains the following information. I omitted some of the information and included the forecast just for two days, but it shows the structure:

```
{ "city":
   { "id": 2653941,
     "name": "Cambridge",
     "coord": { "lon": 0.11667, "lat": 52.200001 },
     "country": "GB" },
  "list":
   [ { "dt": 1439380800,
       "temp": { "min": 14.12, "max": 15.04 } },
     { "dt": 1439467200,
       "temp": { "min": 15.71, "max": 22.44 } } ] }
```

As mentioned before, we could parse the JSON and then write something like `json.GetField("list").AsList()` to access the list with temperatures, but we can do much better than that with type providers.

The F# Data library comes with `JsonProvider`, which is a parameterized type provider that takes a *sample* JSON. It infers the type of the sample document and generates a type that can be used for working with documents that have the same structure. The sample can be specified as a URL, so we can get a type for calling the weather forecast endpoint as follows:

```
type Weather = JsonProvider<"http://api.openweathermap
  .org/data/2.5/forecast/daily?units=metric&q=Prague">
```

Because of the width limitations, we have to split the URL into multiple lines in the report. This won't actually work, so make sure to keep the sample URL on a single line when typing the code!

The parameter of a type provider has to be a constant. In order to generate the `Weather` type, the F# compiler needs to be able to get the value of the parameter at compile-time without running any code. This is also the reason why we are not allowed to use string concatenation with a `+` here, because that would be an *expression*, albeit a simple one, rather than a *constant*.

Now that we have the `Weather` type, let's see how we can use it:

```
let w = Weather.GetSample()
printfn "%s" w.City.Country
for day in w.List do
  printfn "%f" day.Temp.Max
```

The first line calls the `GetSample` method to obtain the forecast using the sample URL—in our case, the temperature in Prague in

metric units. We then use the F# `printfn` function to output the country (just to check that we got the correct city!) and a `for` loop to iterate over the seven days that the forecast service returns.

As with the World Bank type provider, you get auto-completion when accessing. For example, if you type `day.Temp` and `.`, you will see that the service the returns forecasted temperature for morning, day, evening, and night, as well as maximal and minimal temperatures during the day. This is because `Weather` is a type provided based on the sample JSON document that we specified.

When you use the JSON type provider to call a REST-based service, you do not even need to look at the documentation or sample response. The type provider brings this directly into your editor.

In this example, we use `GetSample` to request the weather forecast based on the sample URL, which has to be constant. But we can also use the `Weather` type to get data for other cities. The following snippet defines a `getTomorrowTemp` function that returns the maximal temperature for tomorrow:

```
let baseUrl = "http://api.openweathermap.org/data/2.5"
let forecastUrl = baseUrl + "/forecast/daily?units=metric&q="

let getTomorrowTemp place =
  let w = Weather.Load(forecastUrl + place)
  let tomorrow = Seq.head w.List
  tomorrow.Temp.Max

getTomorrowTemp "Prague"
getTomorrowTemp "Cambridge,UK"
```

The Open Weather Map returns the JSON document with the same structure for all cities. This means that we can use the `Load` method to load data from a different URL, because it will still have the same properties. Once we have the document, we call `Seq.head` to get the forecast for the first day in the list.

As mentioned before, F# is statically typed, but we did not have to write any type annotations for the `getTomorrowTemp` function. That's because the F# compiler is smart enough to infer that `place` has to be a string (because we are appending it to another string) and that

the result is `float` (because the type provider infers that based on the values for the `max` field in the sample JSON document).

A common question is, what happens when the schema of the returned JSON changes? For example, what if the service stops returning the `Max` temperature as part of the forecast? If you specify the sample via a live URL (like we did here), then your code will no longer compile. The JSON type provider will generate type based on the response returned by the latest version of the API, and the type will not expose the `Max` member. This is a good thing though, because we will catch the error during development and not later at runtime.

If you use type providers in a compiled and deployed code and the schema changes, then the behavior is the same as with any other data access technology—you'll get a runtime exception that you have to handle. Finally, it is worth noting that you can also pass a local file as a sample, which is useful when you're working offline.

Plotting Temperatures Around the World

Now that we've seen how to use the World Bank type provider to get information about countries and the JSON type provider to get the weather forecast, we can combine the two and visualize the temperatures around the world!

To do this, we iterate over all the countries in the world and call `getTomorrowTemp` to get the maximal temperature in the capital cities:

```
let worldTemps =
  [ for c in wb.Countries ->
      let place = c.CapitalCity + "," + c.Name
      printfn "Getting temperature in: %s" place
      c.Name, getTomorrowTemp place ]
```

If you are new to F#, there is a number of new constructs in this snippet:

- `[ for .. in .. -> .. ]` is a *list expression* that generates a list of values. For every item in the input sequence `wb.Countries`, we return one element of the resulting list.

- `c.Name, getTomorrowTemp place` creates a pair with two elements. The first is the name of the country and the second is the temperature in the capital.
- We use `printf` in the list expression to print the place that we are processing. Downloading all data takes a bit of time, so this is useful for tracking progress.

To better understand the code, you can look at the type of the `worldTemps` value that we are defining. This is printed in F# Interactive when you run the code, and most F# editors also show a tooltip when you place the mouse pointer over the identifier. The type of the value is `(string * float) list`, which means that we get a list of pairs with two elements: the first is a string (country name) and the second is a floating-point number (temperature).[5]

After you run the code and download the temperatures, you're ready to plot the temperatures on a map. To do this, we use the XPlot library, which is a lightweight F# wrapper for Google Charts:

```
open XPlot.GoogleCharts
Chart.Geo(worldTemps)
```

The `Chart.Geo` function expects a collection of pairs where the first element is a country name or country code and the second element is the value, so we can directly call this with `worldTemps` as an argument. When you select the second line and run it in F# Interactive, XPlot creates the chart and opens it in your default web browser.

To make the chart nicer, we'll need to use the F# pipeline operator `|>`. The operator lets you use the fluent programming style when applying a chain of operations or transformations. Rather than calling `Chart.Geo` with `worldTemps` as an argument, we can get the data and *pass* it to the charting function as `worldTemps |> Chart.Geo`.

Under the cover, the `|>` operator is very simple. It takes a value on the left, a function on the right, and calls the function with the value as an argument. So, `v |> f` is just shorthand for `f v`. This becomes more useful when we need to apply a number of operations, because we can write `g (f v)` as `v |> f |> g`.

---

5 If you are coming from a C# background, you can also read this as `List<Tuple<string, float>>`.

The following snippet creates a `ColorAxis` object to specify how to map temperatures to colors (for more information on the options, see the XPlot documentation). Note that XPlot accepts parameters as .NET arrays, so we use the notation `[| .. |]` rather than using a plain list expression written as `[ .. ]`:

```
let colors = [| "#80E000";"#E0C000";"#E07B00";"#E02800" |]
let values = [| 0;+15;+30;+45 |]
let axis = ColorAxis(values=values, colors=colors)

worldTemps
|> Chart.Geo
|> Chart.WithOptions(Options(colorAxis=axis))
|> Chart.WithLabel "Temp"
```

The `Chart.Geo` function returns a chart object. The various `Chart.With` functions then transform the chart object. We use `WithOptions` to set the color axis and `WithLabel` to specify the label for the values. Thanks to the static typing, you can explore the various available options using code completion in your editor.

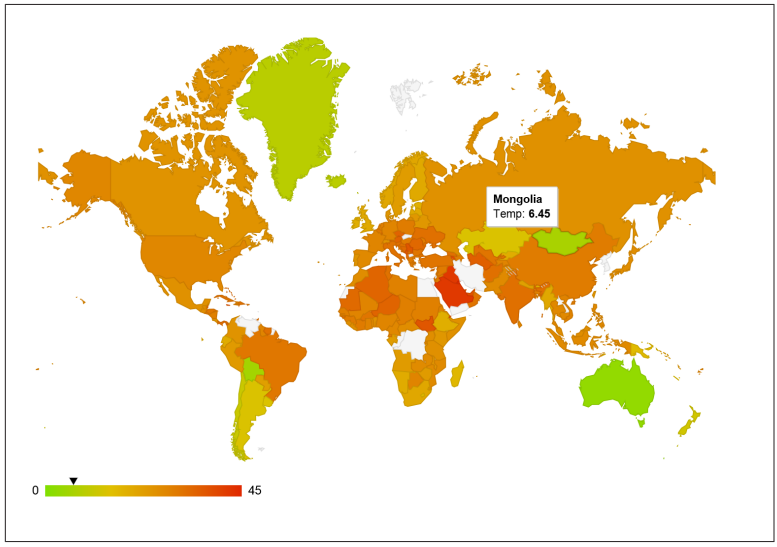


Figure 1-2. Forecasted temperatures for tomorrow with label and custom color scale

The resulting chart should look like the one in Figure 1-2. Just be careful, if you are running the code in the winter, you might need to tweak the scale!

Conclusions

The example in this chapter focused on the *access* part of the data science workflow. In most languages, this is typically the most frustrating part of the *access*, *analyze*, *visualize* loop. In F#, type providers come to the rescue!

As you could see in this chapter, type providers make data access simpler in a number of ways. Type providers integrate external data sources directly into the language, and you can explore external data inside your editor. You could see this with the specialized World Bank type provider (where you can choose countries and indicators in the completion list), and also with the general-purpose JSON type provider (which maps JSON object fields into F# types). However, type providers are not useful *only* for data access. As we'll see in the next chapter, they can also be useful for calling external non-F# libraries.

To build the visualization in this chapter, we needed to write just a couple of lines of F# code. In the next chapter, we download larger amounts of data using the World Bank REST service and preprocess it to get ready for the simple clustering algorithm implemented in Chapter 3.

CHAPTER 2

Analyzing Data Using F# and Deedle

In the previous chapter, we carefully picked a straightforward example that does not require too much data preprocessing and too much fiddling to find an interesting visualization to build. Life is typically not that easy, so this chapter looks at a more realistic case study. Along the way, we will add one more library to our toolbox. We will look at Deedle,[1] which is a .NET library for data and time series manipulation that is great for interactive data exploration, data alignment, and handling missing values.

In this chapter, we download a number of interesting indicators about countries of the world from the World Bank, but we do so efficiently by calling the REST service directly using an XML type provider. We align multiple data sets, fill missing values, and build two visualizations looking at CO_2 emissions and the correlation between GDP and life expectancy.

We'll use the two libraries covered in the previous chapter (F# Data and XPlot) together with Deedle. If you're referencing the libraries using the FsLab package as before, you'll need the following `open` declarations:

```
#r "System.Xml.Linq.dll"
#load "packages/FsLab/FsLab.fsx"
```

1 See *http://fslab.org/Deedle/*.

```
open Deedle
open FSharp.Data
open XPlot.GoogleCharts
open XPlot.GoogleCharts.Deedle
```

There are two new things here. First, we need to reference the `System.Xml.Linq` library, which is required by the XML type provider. Next, we open the `Deedle` namespace together with extensions that let us pass data from the Deedle series directly to XPlot for visualization.

Downloading Data Using an XML Provider

Using the World Bank type provider, we can easily access data for a specific indicator and country over all years. However, here we are interested in an indicator for a specific year, but over all countries. We could download this from the World Bank type provider too, but to make the download more efficient, we can use the underlying API directly and get data for all countries with just a single request. This is also a good opportunity to look at how the XML type provider works.

As with the JSON type provider, we give the XML type provider a sample URL. You can find more information about this query in the World Bank API documentation. The code `NY.GDP.PCAP.CD` is a sample indicator returning GDP growth per capita:

```
type WorldData = XmlProvider<"http://api.worldbank
  .org/countries/indicators/NY.GDP.PCAP.CD?date=2010:2010">
```

As in the last chapter, we had to split this into two lines, but you should have the sample URL on a single line in your source code. You can now call `WorldData.GetSample()` to download the data from the sample URL, but with type providers, you don't even need to do that. You can start using the generated type to see what members are available and find the data in your F# editor.

In the last chapter, we loaded data into a list of type `(string*float) list`. This is a list of pairs that can also be written as `list<string*float>`. In the following example, we create a Deedle series `Series<string, float>`. The series type is parameterized by the type of keys and the type of values, and builds an index based on the keys. As we'll see later, this can be used to align data from multiple series.

We write a function `getData` that takes a year and an indicator code, then downloads and parses the XML response. Processing the data is similar to the JSON type provider example from the previous chapter:

```
let indUrl = "http://api.worldbank.org/countries/indicators/"

let getData year indicator =
  let query =
   [("per_page","1000");
    ("date",sprintf "%d:%d" year year)]
  let data = Http.RequestString(indUrl + indicator, query)
  let xml = WorldData.Parse(data)
  let orNaN value =
   defaultArg (Option.map float value) nan
   series [ for d in xml.Datas ->
    d.Country.Value, orNaN d.Value ]
```

To call the service, we need to provide the `per_page` and `date` query parameters. Those are specified as a list of pairs. The first parameter has a constant value of `"1000"`. The second parameter needs to be a date range written as `"2015:2015"`, so we use `sprintf` to format the string.

The function then downloads the data using the `Http.RequestString` helper which takes the URL and a list of query parameters. Then we use `WorldData.Parse` to read the data using our provided type. We could also use `WorkldData.Load`, but by using the `Http` helper we do not have to concatenate the URL by hand (the helper is also useful if you need to specify an HTTP method or provide HTTP headers).

Next we define a helper function `orNaN`. This deserves some explanation. The type provider correctly infers that data for some countries may be missing and gives us `option<decimal>` as the value. This is a high-precision decimal number wrapped in an option to indicate that it may be missing. For convenience, we want to treat missing values as `nan`. To do this, we first convert the value into `float` (if it is available) using `Option.map float value`. Then we use `defaultArg` to return either the value (if it is available) or `nan` (if it is not available).

Finally, the last line creates a series with country names as keys and the World Bank data as values. This is similar to what we did in the

last chapter. The list expression creates a list with tuples, which is then passed to the `series` function to create a Deedle series.

The two examples of using the JSON and XML type providers demonstrate the general pattern. When accessing data, you just need a sample document, and then you can use the type providers to load different data in the same format. This approach works well for any REST-based service, and it means that you do not need to study the response in much detail. Aside from XML and JSON, you can also access CSV files in the same way using `CsvProvider`.

Visualizing CO_2 Emissions Change

Now that we can load an indicator for all countries into a series, we can use it to explore the World Bank data. As a quick example, let's see how the CO_2 emissions have been changing over the last 10 years. We can still use the World Bank type provider to get the indicator code instead of looking up the code on the World Bank web page:

```
let wb = WorldBankData.GetDataContext()
let inds = wb.Countries.World.Indicators
let code = inds.``CO2 emissions (kt)``.IndicatorCode

let co2000 = getData 2000 code
let co2010 = getData 2010 code
```

At the beginning of the chapter, we opened Deedle extensions for XPlot. Now you can directly pass `co2000` or `co2010` to `Chart.Geo` and write, for example, `Chart.Geo(co2010)` to display the total carbon emissions of countries across the world. This shows the expected results (with China and the US being the largest polluters). More interesting numbers appear when we calculate the relative change over the last 10 years:

```
let change = (co2010 - co2000) / co2000 * 100.0
```

The snippet calculates the difference, divides it by the 2000 values to get a relative change, and multiplies the result by 100 to get a percentage. But the whole calculation is done over a *series* rather than over individual *values*! This is possible because a Deedle series supports numerical operators and automatically aligns data based on the keys (so, if we got the countries in a different order, it will still work). The operations also propagate missing values correctly. If the

value for one of the years is missing, it will be marked as missing in the resulting series, too.

As before, you can call `Chart.Geo(change)` to produce a map with the changes. If you tweak the color scale as we did in the last chapter, you'll get a visualization similar to the one in Figure 2-1 (you can get the complete source code from *http://fslab.org/report)*.

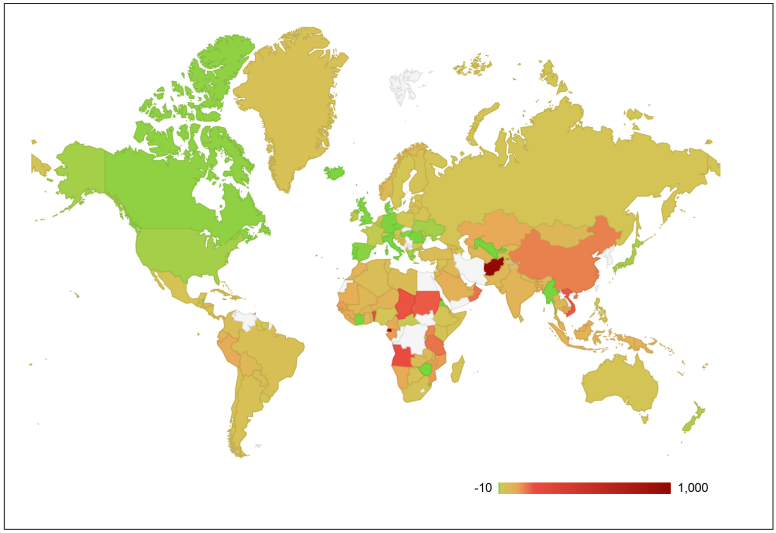


Figure 2-1. Change in CO_2 emissions between 2000 and 2010

As you can see in Figure 2-1, we got data for most countries of the world, but not for all of them. The range of the values is between -70% to +1200%, but emissions in most countries are growing more slowly. To see this, we specify a green color for -10%, yellow for 0%, orange for +100, red for +200%, and very dark red for +1200%.

In this example, we used Deedle to align two series with country names as indices. This kind of operation is useful all the time when combining data from multiple sources, no matter whether your keys are product IDs, email addresses, or stock tickers. If you're working with a time series, Deedle offers even more. For example, for every key from one time-series, you can find a value from another series whose key is the closest to the time of the value in the first series. You can find a detailed overview in the Deedle page about working with time series.

Aligning and Summarizing Data with Frames

The `getData` function that we wrote in the previous section is a perfect starting point for loading more indicators about the world. We'll do exactly this as the next step, and we'll also look at simple ways to summarize the obtained data.

Downloading more data is easy now. We just need to pick a number of indicators that we are interested in from the World Bank type provider and call `getData` for each indicator. We download all data for 2010 below, but feel free to experiment and choose different indicators and different years:

```
let codes =
[ "CO2", inds.``CO2 emissions (metric tons per capita)``
   "Univ", inds.``School enrollment, tertiary (% gross)``
   "Life", inds.``Life expectancy at birth, total (years)``
   "Growth", inds.``GDP per capita growth (annual %)``
   "Pop", inds.``Population growth (annual %)``
   "GDP", inds.``GDP per capita (current US$)`` ]

let world =
  frame [ for name, ind in codes ->
    name, getData 2010 ind.IndicatorCode ]
```

The code snippet defines a list with pairs consisting of a short indicator name and the code from the World Bank. You can run it and see what the codes look like—choosing an indicator from an autocomplete list is much easier than finding it in the API documentation!

The last line does all the actual work. It creates a list of key value pairs using a sequence expression `[ ... ]`, but this time, the value is a series with data for all countries. So, we create a list with an indicator name and data series. This is then passed to the `frame` function, which creates a *data frame*.

A data frame is a Deedle data structure that stores multiple series. You can think of it as a table with multiple columns and rows (similar to a data table or spreadsheet). When creating a data frame, Deedle again makes sure that the values are correctly aligned based on their keys.

Table 2-1. Data frame with information about the world

	CO_2	Univ	Life	Growth	Pop	GDP
Afghanistan	0.30	N/A	59.60	5.80	2.46	561.20
Albania	1.52	43.56	76.98	4.22	-0.49	4094.36
Algeria	3.22	28.76	70.62	1.70	1.85	4349.57
:	…					
Yemen, Rep.	1.13	10.87	62.53	0.90	2.37	1357.76
Zambia	0.20	N/A	54.53	7.03	3.01	1533.30
Zimbabwe	0.69	6.21	53.59	9.77	1.45	723.16

Data frames are useful for interactive data exploration. When you create a data frame, F# Interactive formats it nicely so you can get a quick idea about the data. For example, in Table 2-1 you can see the ranges of the values and which values are frequently missing.

Data frames are also useful for interoperability. You can easily save data frames to CSV files. If you want to use F# for data access and cleanup, but then load the data in another language or tool such as R, Mathematica, or Python, data frames give you an easy way to do that. However, if you are interested in calling R, this is even easier with the F# R type provider.

Summarizing Data Using the R Provider

When using F# for data analytics, you can access a number of useful libraries: Math.NET Numerics for statistical and numerical computing, Accord.NET for machine learning, and others. However, F# can also integrate with libraries from other ecosystems. We already saw

this with XPlot, which is an F# wrapper for the Google Charts visualization library. Another good example is the R type provider.[2]

The R Project and R Type Provider

R is a popular programming language and software environment for statistical computing. One of the main reasons for the popularity of R is its comprehensive archive of statistical packages (CRAN), providing libraries for advanced charting, statistics, machine learning, financial computing, bioinformatics, and more. The R type provider makes the packages available to F#.

The R type provider is cross-platform, but it requires a 64-bit version of Mono on Mac and Linux. The documentation explains the required setup in details. Also, the R provider uses your local installation of R, so you need to have R on your machine in order to use it! You can get R from *http://www.r-project.org*.

In R, functionality is organized as *functions* in *packages*. The R type provider discovers R packages that are installed on your machine and makes them available as F# modules. R functions then become F# functions that you can call. As with type providers for accessing data, the modules and functions become normal F# entities, and you can discover them through auto-complete.

The R type provider is also included in the FsLab package, so no additional installation is needed. If you have R installed, you can run the `plot` function from the `graphics` package to get a quick visualization of correlations in the `world` data frame:

```
open RProvider
open RProvider.graphics

R.plot(world)
```

If you are typing the code in your editor, you can use auto-completion in two places. First, after typing `RProvider` and `.` (dot), you can see a list with all available packages. Second, after typing `R` and `.` (dot), you can see functions in all the packages you opened. Also note that we are calling the R function with a Deedle data frame as an argument. This is possible because the R provider

2 See *http://fslab.org/RProvider*.

knows how to convert Deedle frames to R data frames. The call then invokes the R runtime, which opens a new window with the chart displayed in Figure 2-2.

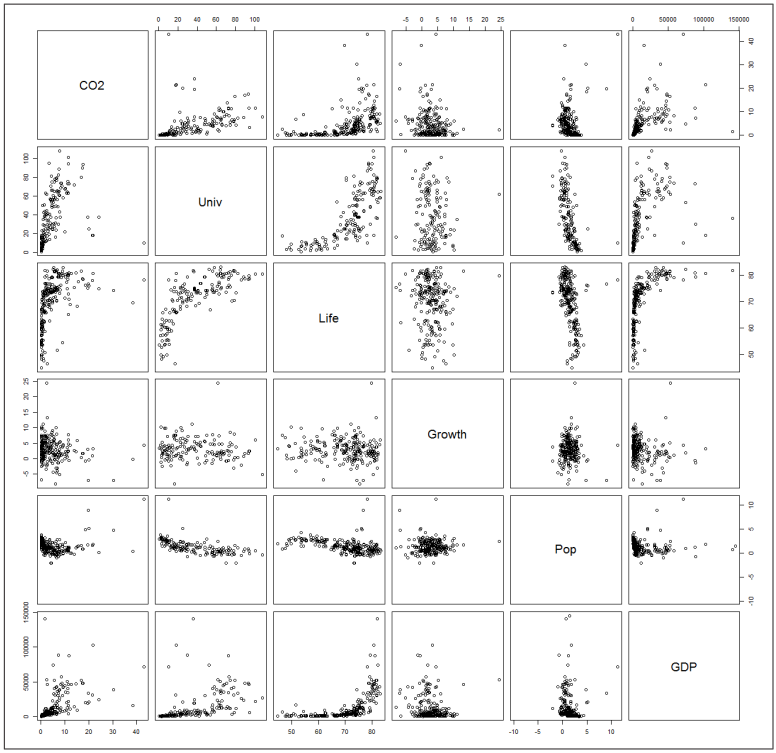


Figure 2-2. R plot showing correlations between indicators

The `plot` function creates a scatter plot for each combination of rows in our input data, so we can quickly check if there are any correlations. For example, if you look at the intersection of the *Life* row and *GDP* column, you can see that there might be some correlation between life expectancy and GDP per capita (but not a linear one). We'll see this better after normalizing the data in the next section.

The `plot` function is possibly the most primitive function from R we can call, but it shows the idea. However, R offers a number of powerful packages that you can access from F# thanks to the R provider. For example, you can use `ggplot2` for producing print-ready charts, `nnet` for neural networks, and numerous other packages for regressions, clustering, and other statistical analyses.

Normalizing the World Data Set

As the last step in this chapter, we write a simple computation to normalize the data in the `world` data frame. As you could see in Table 2-1, the data set contains quite diverse numbers, so we rescale the values to a scale from 0 to 1. This prepares the data for the clustering algorithm implemented in the next chapter, and also lets us explore the correlation between GDP and life expectancy.

To normalize the values, we need the minimal and maximal value for each indicator. Then we can transform a value `v` by calculating `(v-min)/(max-min)`. With Deedle, we do not have to do this for individual values, but we can instead express this as a computation over the whole frame.

As part of the normalization, we also fill missing values with the average value for the indicator. This is simple, but works well enough for us:

```
let lo = Stats.min world
let hi = Stats.max world
let avg = Stats.mean world

let filled =
  world
  |> Frame.transpose
  |> Frame.fillMissingUsing (fun _ ind -> avg.[ind])

let norm =
  (filled - lo) / (hi - lo)
  |> Frame.transpose
```

The normalization is done in three steps:

1. First, we use functions from the `Stats` module to get the smallest, largest, and average values. When applied on a *frame*, the functions return *series* with one number for each column, so we get aggregates for all indicators.
2. Second, we fill the missing values. The `fillMissingUsing` operation iterates over all columns and then fills the missing value for each item in the column by calling the function we provide. To use it, we first transpose the frame (to switch rows and columns). Then `fillMissingUsing` iterates over all countries, gives us the indicator name `ind`, and we look up the average value for the indicator using `avg.[ind]`. We do not need the value of the

first parameter, and rather than assigning it to an unused variable, we use the _ pattern which ignores the value.

3. Third, we perform the normalization. Deedle defines numerical operators between frame and series, such that `filled - lo` subtracts the `lo` series point-wise from each column of the `filled` frame, and we subtract minimal indicator values for each country. Finally, we transpose the frame again into the original shape with indicators as columns and countries as rows.

The fact that the explanation here is much longer than the code shows just how much you can do with just a couple of lines of code with F# and Deedle. The library provides functions for joining frames, grouping, and aggregation, as well as windowing and sampling (which are especially useful for time-indexed data). For more information about the available functions, check out the documentation for the `Stats` module and the documentation for the `Frame` module on the Deedle website.

To finish the chapter with an interesting visualization, let's use the normalized data to build a scatter plot that shows the correlation between GDP and life expectancy. As suggested earlier, the growth is not linear so we take the logarithm of GDP:

```
let gdp = log norm.["GDP"] |> Series.values
let life = norm.["Life"] |> Series.values

let options = Options(pointSize=3, colors=[|"#3B8FCC"|],
  trendlines=[|Trendline(opacity=0.5,lineWidth=10)|],
  hAxis=Axis(title="Log of scaled GDP (per capita)"),
  vAxis=Axis(title="Life expectancy (scaled)"))

Chart.Scatter(Seq.zip gdp life)
|> Chart.WithOptions(options)
```

The `norm.["GDP"]` notation is used to get a specified column from the data frame. This returns a series, which supports basic numerical operators (as used in "Visualizing CO_2 Emissions Change" on page 18) as well as basic numerical functions, so we can directly call `log` on the series.

For the purpose of the visualization, we need just the values and not the country names, so we call `Series.values` to get a plain F# sequence with the raw values. We then combine the values for the X and Y axes using `Seq.zip` to get a sequence of pairs representing the two indicators for each country. To get the chart in Figure 2-3, we

also specify visual properties, titles, and most importantly, add a linear trend line.

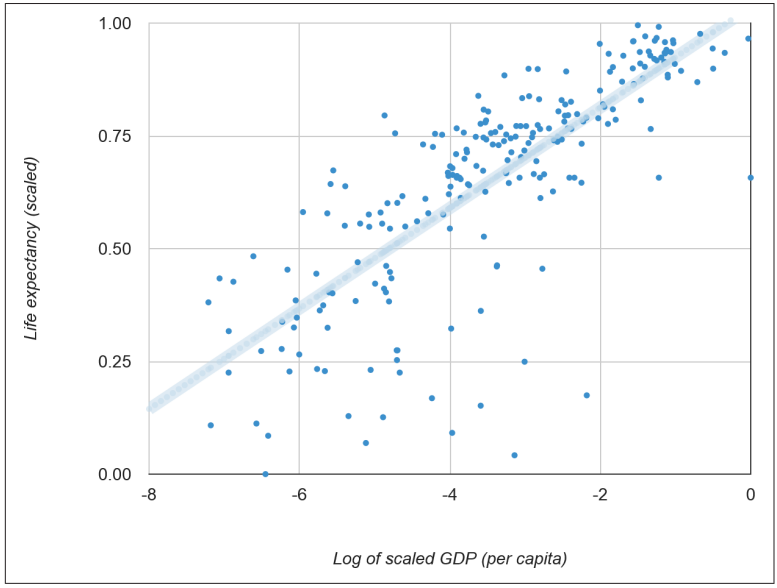


Figure 2-3. Correlation between logarithm of GDP and life expectancy

If we denormalize the numbers, we can roughly say that countries with a life expectancy greater by 10 years have 10 times larger GDP per capita. That said, to prove this point more convincingly, we would have to test the statistical significance of the hypothesis, and we'd have to go back to the R type provider!

Conclusions

In this chapter, we looked at a more realistic case study of doing data science with F#. We still used World Bank as our data source, but this time we called it using the XML provider directly. This demonstrates a general approach that would work with any REST-based service.

Next, we looked at the data in two different ways. We used Deedle to print a data frame showing the numerical values. This showed us that some values are missing and that different indicators have very different ranges, and we later normalized the values for further processing. Next, we used the R type provider to get a quick overview of correlations. Here, we really just scratched the surface of what is

possible. The R provider provides access to over 5000 statistical packages which are invaluable when doing more complex data analysis.

In the first two chapters, we used a number of external libraries (all of them available conveniently through the FsLab package). In the next chapter, we shift our focus from *using* to *creating*, and we'll look at how to use F# to implement a simple clustering algorithm.

CHAPTER 3

Implementing Machine Learning Algorithms

All of the analysis that we discussed so far in this report was manual. We looked at some data, we had some idea what we wanted to find or highlight, we transformed the data, and we built a visualization. Machine learning aims to make the process more automated. In general, *machine learning* is the process of building models automatically from data. There are two basic kinds of algorithms. *Supervised* algorithms learn to generalize from data with known answers, while *unsupervised* algorithms automatically learn to model data without known structure.

In this chapter, we implement a basic, unsupervised machine learning algorithm called *k-means clustering* that automatically splits inputs into a specified number of groups. We'll use it to group countries based on the indicators obtained in the previous chapter.

This chapter also shows the F# language from a different perspective. So far, we did not need to implement any complicated logic and mostly relied on existing libraries. In contrast, this chapter uses just the standard F# library, and you'll see a number of ways in which F# makes it very easy to implement new algorithms—the primary way is *type inference* which lets us write efficient and correct code while keeping it very short and readable.

How k-Means Clustering Works

The k-means clustering algorithm takes input data, together with the number *k* that specifies how many clusters we want to obtain, and automatically assigns the individual inputs to one of the clusters. It is *iterative*, meaning that it runs in a loop until it reaches the final result or a maximal number of steps.

The idea of the algorithm is that it creates a number of *centroids* that represent the centers of the clusters. As it runs, it keeps adjusting the centroids so that they better cluster the input data. It is an *unsupervised* algorithm, which means that we do not need to know any information about the clusters (say, sample inputs that belong there).

To demonstrate how the algorithm works, we look at an example that can be easily drawn in a diagram. Let's say that we have a number of points with X and Y coordinates and we want to group them in clusters. Figure 3-1 shows the points (as circles) and current centroids (as stars). Colors illustrate the current clustering that we are trying to improve. This is very simple, but it is sufficient to get started.

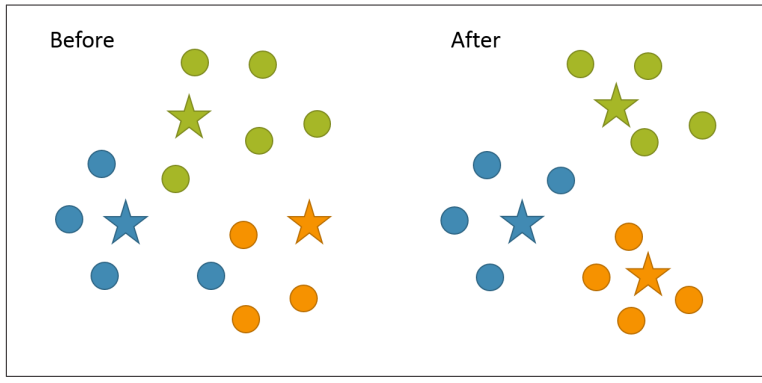


Figure 3-1. Clustering three groups of circles with stars showing k-means centroids

The algorithm runs in three simple steps:

1. First, we randomly generate initial centroids. This can be done by randomly choosing some of the inputs as centroids, or by generating random values. In the figure, we placed three stars at random X and Y locations.

2. Second, we update the clusters. For every input, we find the nearest centroid, which determines the cluster to which the input belongs. In the figure, we show this using color—each input has the color of the nearest centroid. If this step does not change the inputs in any of the clusters, we are done and can return them as the final result.
3. Third, we update the centroids. For each cluster (group of inputs with the same color), we calculate the center and move the centroid into this new location. Next, we jump back to the second step and update the clusters again, based on the new centroids.

The example in Figure 3-1 shows the state before and after one iteration of the loop. In "Before," we randomly generated the location of the centroids (shown as stars) and assigned all of the inputs to the correct cluster (shown as different colors). In "After," we see the new state after running steps 3 and 2. In step 3, we move the green centroid to the right (the leftmost green circle becomes blue), and we move the orange centroid to the bottom and a bit to the left (the rightmost blue circle becomes orange).

To run the algorithm, we do not need any classified samples, but we do need two things. We need to be able to measure the distance (to find the nearest centroid), and we need to be able to aggregate the inputs (to calculate a new centroid). As we'll see in "Writing a Reusable Clustering Function" on page 36, this information will be nicely reflected in the F# type information at the end of the chapter, so it's worth remembering.

Clustering 2D Points

Rather than getting directly to the full problem and clustering countries, we start with a simpler example. Once we know that the code works on the basic sample, we'll turn it into a reusable F# function and use it on the full data set.

Our sample data set consists of just six points. Assuming `0.0, 0.0` is the bottom left corner, we have two points in the bottom left, two in the bottom right, and two in the top left corner:

```
let data =
  [ (0.0, 1.0);  (1.0, 1.0);
```

```
    (10.0, 1.0); (13.0, 3.0);
    (4.0, 10.0); (5.0, 8.0) ]
```

The notation `[ ... ]` is the list expression (which we've seen in previous chapters), but this time we're creating a list of explicitly given tuples.

If you run the code in F# Interactive, you'll see that the type of the `data` value is `list<float * float>`,[1] so the tuple `float * float` is the type of individual input. As discussed before, we need the *distance* and *aggregate* functions for the inputs:

```
let distance (x1, y1) (x2, y2) : float =
  sqrt ((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))

let aggregate points : float * float =
  (List.averageBy fst points, List.averageBy snd points)
```

The `distance` function takes two points and produces a single number. Note that in F#, function parameters are separated by spaces, and so `(x1, y1)` is the first parameter and `(x2, y2)` is the second. However, both parameters are *bound to patterns* that decompose the tuple into individual components, and we get access to the X and Y coordinates for both points. We also included the type annotation specifying that the result is `float`. This is needed here because the F# compiler would not know what numerical type we intend to use. The body then simply calculates the distance between the two points.

The `aggregate` function takes a *list* of inputs and calculates their centers. This is done using the `List.averageBy` function, which takes two arguments. The second argument is the input list, and the first argument is a projection function that specifies what value (from the input) should be averaged. The `fst` and `snd` functions return the first and second element of a tuple, respectively, and this averages the X and Y coordinates.

1 The F# compiler also reports this as `(float * float) list`, which is just a different way of writing the same type.

Initializing Centroids and Clusters

The first step of the k-means algorithm is to initialize the centroids. For our sample, we use three clusters. We initialize the centroids by randomly picking three of the inputs:

```
let clusterCount = 3

let centroids =
  let random = System.Random()
  [ for i in 1 .. clusterCount ->
      List.nth data (random.Next(data.Length)) ]
```

The code snippet uses the `List.nth` function to access the element at the random offset (in F# 4.0, `List.nth` is deprecated, and you can use the new `List.item` instead). We also define the `random` value as part of the definition of `centroids`—this makes it accessible only inside the definition of `centroids` and we keep it local to the initialization code.

Our logic here is not perfect, because we could accidentally pick the same input twice and two clusters would fully overlap. This is something we should improve in a proper implementation, but it works well enough for our demo.

The next step is to find the closest centroid for each input. To do this, we write a function `closest` that takes all centroids and the input we want to classify:

```
let closest centroids input =
  centroids
  |> List.mapi (fun i v -> i, v)
  |> List.minBy (fun (_, cent) -> distance cent input)
  |> fst
```

The function works in three steps that are composed in a sequence using the pipeline `|>` operator that we've seen in the first chapter. Here, we start with `centroids`, which is a list, and apply a number of transformations on the list:

1. We use `List.mapi`, which calls the specified function for each element of the input list and collects the results into an output list.[2] The `mapi` function gives us the value `v`, and also the index `i`

2 If you are familiar with LINQ, then this is the `Select` extension method.

(hence `mapi` and not just `map`), and we construct a tuple with the index and the value. Now we have a list with centroids together with their index.

2. Next, we use `List.minBy` to find the smallest element of the list according to the specified criteria—in our case, this is the distance from the input. Note that we get the element of the previous list as an input. This is a pair with index and centroid, and we use *pattern* `(_, cent)` to extract the second element (centroid) and assign it to a variable while ignoring the index of the centroid (which is useful in the next step).
3. The `List.minBy` function returns the element of the list for which the function given as a parameter returned the smallest value. In our case, this is a value of type `int * (float * float)` consisting of the index together with the centroid itself. The last step then uses `fst` to get the first element of the tuple, that is, the index of the centroid.

The one new piece of F# syntax used in this snippet is an anonymous function that is created using `fun v1 .. vn -> e`, where `v1 .. vn` are the input variables (or patterns) and `e` is the body of the function.

Now that we have a function to classify one input, we can easily use `List.map` to classify all inputs:

```
data |> List.map (fun point ->
  closest centroids point)
```

Try running the above in F# Interactive to see how your random centroids are generated! If you are lucky, you might get a result `[0; 0; 1; 1; 2; 2]` which would mean that you already have the perfect clusters. But this is not likely the case, so we'll need to run the next step.

Before we continue, it is worth noting that we could also write `data |> List.map (closest centroids)`. This uses an F# feature called *partial function application* and means the exact same thing: F# automatically creates a function that takes `point` and passes it as the next argument to `closest centroids`.

Updating Clusters Recursively

The last part of the algorithm that we need to implement is updating the centroids (based on the assignments to clusters) and looping until the cluster assignment stops changing. To do this, we write a recursive function `update` that takes the current assignment to clusters and produces the final assignment (after the looping converges). The assignments to clusters is just a list (as in the previous section) that has the same length as our `data` and contains the index of a cluster (between `0` and `clusterCount-1`).

To get all inputs for a given cluster, we need to filter the data based on the assignments. We will use the `List.zip` function which aligns elements in two lists and returns a list of tuples. For example:

```
List.zip [1; 2; 3; 4] ['A'; 'B'; 'C'; 'D'] =
  [(1,'A'); (2,'B'); (3,'C'); (4,'D')]
```

Aside from `List.zip`, the only new F# construct in the following snippet is `let rec`, which is the same as `let`, but it explicitly marks the function as recursive (meaning that it is allowed to call itself):

```
let rec update assignment =
  let centroids =
    [ for i in 0 .. clusterCount-1 ->
        let items =
          List.zip assignment data
          |> List.filter (fun (c, data) -> c = i)
          |> List.map snd
        aggregate items ]
  let next = List.map (closest centroids) data
  if next = assignment then assignment
  else update next

let assignment =
  update (List.map (closest centroids) data)
```

The function first calculates new centroids. To do this, it iterates over the centroid indices. For each centroid, it finds all `items` from `data` that are currently assigned to the centroid. Here, we use `List.zip` to create a list containing items from `data` together with their assignments. We then use the `aggregate` function (defined earlier) to calculate the center of the items.

Once we have new `centroids`, we calculate new assignments based on the updated clusters (using `List.map (closest centroids) data`, as in the previous section).

The last two lines of the function implement the looping. If the new assignment `next` is the same as the previous `assignment`, then we are done and we return the assignment as the result. Otherwise, we call `update` recursively with the new assignment (and it updates the centroids again, leading to a new assignment, etc.).

It is worth noting that F# allows us to use `next = assignment` to compare two arrays. It implements *structural equality* by comparing the arrays based on their contents instead of their reference (or position in the .NET memory).

Finally, we call `update` with the initial assignments to cluster our sample points. If everything worked well, you should get a list such as `[1;1;2;2;0;0]` with the three clusters as the result. However, there are two things that could go wrong and would be worth improving in the full implementation:

- **Empty clusters.** If the random initialization picks the same point twice as a centroid, we will end up with an empty cluster (because `List.minBy` always returns the first value if there are multiple values with the same minimum). This currently causes an exception because the `aggregate` function does not work on empty lists. We could fix this either by dropping empty clusters, or by adding the original center as another parameter of `aggregate` (and keeping the centroid where it was before).
- **Termination condition.** The other potential issue is that the looping could take too long. We might want to stop it not just when the clusters stop changing, but also after a fixed number of iterations. To do this, we would add the `iters` parameter to our `update` function, increment it with every recursive call, and modify the termination condition.

Even though we did all the work using an extremely simple special case, we now have everything in place to turn the code into a reusable function. This nicely shows the typical F# development process.

Writing a Reusable Clustering Function

A nice aspect of how we were writing code so far is that we did it in small chunks and we could immediately test the code interactively to see that it works on our small example. This makes it easy to avoid silly mistakes and makes the software development process

much more pleasant, especially when writing machine learning algorithms where many little details could go wrong that would be hard to discover later!

The last step is to take the code and turn it into a function that we can call on different inputs. This turns out to be extremely easy with F#. The following snippet is *exactly* the same as the previous code—the only difference is that we added a function header (first line), indented the body further, and changed the last line to return the result:

```
let kmeans distance aggregate clusterCount data =

  let centroids =
    let rnd = System.Random()
    [ for i in 1 .. clusterCount ->
        List.nth data (rnd.Next(data.Length)) ]

  let closest centroids input =
    centroids
    |> List.mapi (fun i v -> i, v)
    |> List.minBy (fun (_, cent) -> distance cent input)
    |> fst

  let rec update assignment =
    let centroids =
      [ for i in 0 .. clusterCount-1 ->
          let items =
            List.zip assignment data
            |> List.filter (fun (c, data) -> c = i)
            |> List.map snd
          aggregate items ]
    let next = List.map (closest centroids) data
    if next = assignment then assignment
    else update next

  update (List.map (closest centroids) data)
```

The most interesting aspect of the change we did is that we turned all the inputs for the k-means algorithm into function parameters. This includes not just `data` and `clusterCount`, but also the functions for calculating the distance and aggregating the items. The function does not rely on any values defined earlier, and you can extract it into a separate file and could turn it into a library, too.

An interesting thing happened during this change. We turned the code that worked on just 2D points into a function that can work on *any* inputs. You can see this when you look at the type of the func-

tion (either in a tooltip or by sending it to F# Interactive). The type signature of the function looks as follows:

```
val kmeans :
      distance : ('a -> 'a -> 'b) ->
     aggregate : ('a list -> 'a) ->
  clusterCount : int ->
          data : 'a list
                          -> int list
      (when 'b : comparison)
```

In F#, the `'a` notation in a type signature represents a type parameter. This is a variable that can be substituted for any actual type when the function is called. This means that the `data` parameter can be a list containing any values, but only if we also provide a `distance` function that works on the same values, and `aggregate` function that turns a list of those values into a single value. The `clusterCount` parameter is just a number, and the result is `int list`, representing the assignments to clusters.

The `distance` function takes two `'a` values and produces a distance of type `'b`. Surprisingly, the distance does not have to return a floating point number. It can be any value that supports the `comparison` constraint (as specified on the last line). For instance, we could return `int`, but not `string`. If you think about this, it makes sense—we do not do any calculations with the distance. We just need to find the smallest value (using `List.minBy`), so we only need to compare them. This can be done on `float` or `int`; there is no way to compare two `string` values.

The compiler is not just *checking* the types to detect errors, but also helps you *understand* what your code does by inferring the type.

Learning to read the type signatures takes some time, but it quickly becomes an invaluable tool of every F# programmer. You can look at the inferred type and verify whether it matches your intuition. In the case of k-means clustering, the type signature matches the introduction discussed earlier in "How k-Means Clustering Works" on page 30.

To experiment with the type inference, try removing one of the parameters from the signature of the `kmeans` function. When you

do, the function might still compile (for example, if you have `data` in scope), but it will restrict the type from generic parameter `'a` to `float`, suggesting that something in the code is making it *too* specialized. This is often a hint that there is something wrong with the code!

Clustering Countries

Now that we have a reusable `kmeans` function, there is one step left: run it on the information about the countries that we downloaded at the end of the previous chapter. Recall that we previously defined `norm`, which is a data frame of type `Frame<string, string>` that has countries as rows and a number of indicators as columns. For calling `kmeans`, we need a list of values, so we get the rows of the frame (representing individual countries) and turn them into a list using `List.ofSeq`:

```
let data =
  norm.GetRows<float>().Values |> List.ofSeq
```

The type of `data` is `list<Series<string, float>>`. Every series in the list represents one country with a number of different indicators. The fact that we are using a Deedle series means that we do not have to worry about missing values and also makes calculations easier. The two functions we need for `kmeans` are just a few lines of code:

```
let distance
    (s1:Series<string,float>)
    (s2:Series<string, float>) =
  (s1 - s2) * (s1 - s2) |> Stats.sum

let aggregate items =
  items
  |> Frame.ofRowsOrdinal
  |> Stats.mean
```

The `distance` function takes two series and uses the point-wise `*` and `-` operators to calculate the squares of differences for each column, then sums them to get a single distance metric. We need to provide type annotations, written as `(s1:Series<string,float>)`, to tell the F# compiler that the parameter is a series and that it should use the overloaded numerical operators provided by Deedle (rather than treating them as operators on integers).

The `aggregate` takes a list of series (countries in a cluster) of type `list<Series<string,float>>`. It should return the averaged value that represents the center of the cluster. To do this, we use a simple trick: we turn the series into a frame and then use `Stats.mean` from Deedle to calculate averages over all columns of the frame. This gives us a series where each indicator is the average of all input indicators. Deedle also conveniently skips over missing values.

Now we just need to call the `kmeans` function and draw a chart showing the clusters:

```
let clrs = ColorAxis(colors=[|"red";"blue";"orange"|])
let countryClusters =
  kmeans distance aggregate 3 data

Seq.zip norm.RowKeys countryClusters
|> Chart.Geo
|> Chart.WithOptions(Options(colorAxis=clrs))
```

The snippet is not showing anything new. We call `kmeans` with our new `data` and the `distance` and `aggregate` functions. Then we combine the country names (`norm.RowKeys`) with their cluster assignments and draw a geo chart that uses red, blue, and orange for the three clusters. The result is the map in Figure 3-2.

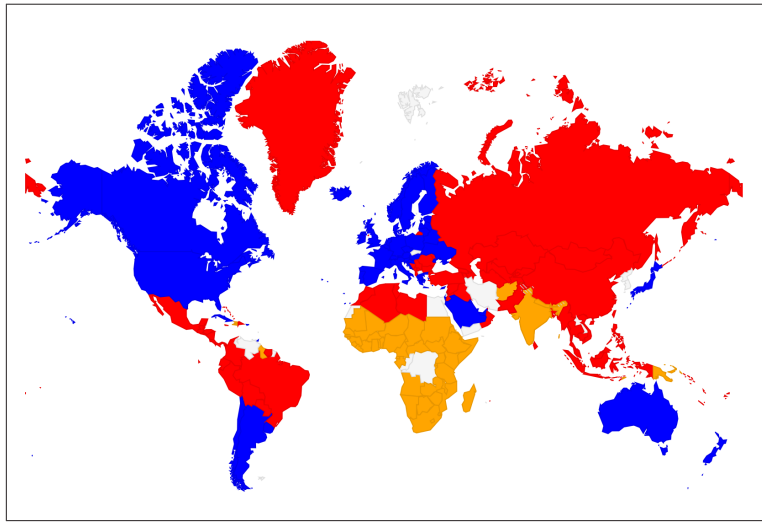


Figure 3-2. Clustering countries of the world based on World Bank indicators

Looking at the image, it seems that the clustering algorithm does identify some categories of countries that we would expect. The next interesting step would be to try understand *why*. To do this, we could look at the final centroids and find which of the indicators contribute the most to the distance between them.

Scaling to the Cloud with MBrace

The quality of the results you get from k-means clustering partly depends on the initialization of the centroids, so you can run the algorithm a number of times with different initial centroids and see which result is better. You can easily do this locally, but what if we were looking not at hundreds of countries, but at millions of products or customers in our database?

In that case, the next step of our journey would be to use the cloud. In F#, you can use the MBrace library,[3] which lets you take existing F# code, wrap the body of a function in the `cloud` computation, and run it in the cloud. You can download a complete example as part of the accompanying source code download, but the following code snippet shows the required changes to the `kmeans` function:

```
let kmeans
  distance aggregate clusterCount
  (remoteData:CloudValue<'T[]>) = cloud {
    let! data = CloudValue.Read remoteData
    // The rest of the function is the same as before
}

kmeans distance aggregator 3 cloudCountries
|> cluster.CreateProcess
```

In the sample, we are using two key concepts from the MBrace library:

- **Cloud computation.** By wrapping the body of the function in `cloud`, we created a cloud computation. This is a block of F# code that can be serialized, transferred to a cluster in the cloud, and executed remotely. Cloud computations can spawn multiple parallel workflows that are then distributed cross the cluster. To start a cloud computation, we use the `CreateProcess` method,

3 Available at *http://www.m-brace.net/*.

which starts the work and returns information about the process running in the cluster.

- **Cloud values.** When running the algorithm on a large number of inputs, we cannot copy the data from the local machine to the cloud every time. The `CloudValue` type lets us create a value (here, containing an array of inputs) that is stored in the cluster, so we can use it to avoid data copying. The `CloudValue.Read` method is used to read the data from the cloud storage into memory.

Using MBrace requires more F# background than we can provide in a brief report, but it is an extremely powerful programming model that lets you take your machine learning algorithms to the next level. Just by adding cloud computations and cloud values, you can turn a simple local implementation into one that runs over hundreds of machines. If you want to learn more about MBrace, the project documentation has all the details and also an extensive collection of samples.[4]

Conclusions

In this chapter, we completed our brief tour by using the F# language to implement the k-means clustering algorithm. This illustrated two aspects of F# that make it nice for writing algorithms:

- First, we wrote the code iteratively. We started by running individual parts of the code on sample input and we could quickly verify that it works. Then we refactored the code into a reusable function. By then, we already knew that the code works.
- Second, the F# type inference helped us along the way. In F#, you do not write types explicitly, so the language is not verbose, but you still get the safety guarantees that come with static typing. The type inference also helps us understand our code, because it finds the most general type. If this does not match your expectations about the algorithm, you know that there is something suspicious going on!

4 Available at *http://www.m-brace.net/programming-model.html*.

Interactive development and type inference will help you when writing any machine learning algorithms. To learn more, you can explore other existing F# projects like the decision tree by Mathias Brandewinder,[5] or the Ariadne project,[6] which implements Gaussian process regression. Writing your own machine learning algorithms is not just a great way to learn the concepts; it is useful when exploring the problem domain. It often turns out that a simple algorithm like k-means clustering works surprisingly well. Often, you can also use an existing F# or .NET library. The next chapter will give you a couple of pointers.

5 Available at *http://bit.ly/decisiontreeblog*.

6 Available at *http://evelinag.com/Ariadne/*.

CHAPTER 4
Conclusions and Next Steps

This brief report shows just a few examples of what you can do with F#, but we used it to demonstrate many of the key features of the language that make it a great tool for data science and machine learning. With type providers, you can elegantly *access* data. We used the XPlot library for *visualization*, but F# also gives you access to the `ggplot2` package from R and numerous other tools. As for *analysis*, we used the Deedle library and R type provider, but we also implemented our own clustering algorithm.

Adding F# to Your Project

I hope this report piqued your interest in F# and showed some good reasons why you might want to use it in your projects. So, what are the best first steps? First of all, you probably should not immediately switch all your code to F# and become the only person in your company who understands it!

A large part of any machine learning and data science is experimentation. Even if your final implementation needs to be in C# (or any other language), you can still use F# to explore the data and prototype different algorithms (using plain F#, R type provider, and the machine learning libraries discussed below).

F# integrates well with .NET and Xamarin applications, so you can write your data access code or a machine learning algorithm in F# and use it in a larger C# application. There are also many libraries

for wrapping F# code as a web application or a web service;[1] and so you can expose the functionality as a simple REST service and host it on Heroku, AWS, or Azure.

Resources for Learning More

If you want to learn more about using F# for data science and machine learning, a number of excellent resources are worth checking out now that you have finished the quick overview in this report.

Report Source Code (fslab.org/report)
: The best way to learn anything is to try it on your own, so download the full source code for the examples from this report and try modifying them to learn other interesting things about the data we've been using, or change the code to load other data relevant to your work!

F# User Groups and Coding Dojos (c4fsharp.net)
: The F# community is very active, and there is likely an F# user group not far from where you live. The Community for F# website is the best place to find more information. It also hosts coding Dojos that you can try completing on your own.

F# Software Foundation (fsharp.org)
: The F# Foundation website is the home of the F# language and is a great starting point if you want to learn more about the language and find resources like books and online tutorials. It also provides up-to-date installation instructions for all platforms.

FsLab Project (fslab.org)
: FsLab is a package that brings together many of the popular data science libraries for F#. We used F# Data (for data access), Deedle and R provider (for data analysis), and XPlot (for visualization). The FsLab website hosts their documentation and other useful resources.

Accord.NET Framework (accord-framework.net)
: Accord.NET is a machine learning library for .NET that works well with F#. In this report, we implemented k-means clustering to demonstrate interesting F# language features, but when solv-

1 See the web guide on the F# Foundation website: *http://fsharp.org/guides/web*.

ing simple machine learning problems, you can often just use an existing library!

MBrace Project (http://www.m-brace.net/)
: MBrace is a simple library for scalable data scripting and programming with F# and C#. It lets you run F# computations in the cloud directly from your F# Interactive with the same rapid feedback that you get when running your F# code locally. Check out MBrace if you are looking at implementing scalable machine learning with F#.

Machine Learning Projects for .NET Developers (Mathias Brandewinder, Apress)
: Finally, if you enjoyed Chapter 3, then Mathias Brandewinder's book is a great resource. It implements a number of machine learning algorithms using F# and also provides more details for some of the libraries used in this report, like the R type provider and F# Data.

About the Author

Tomas Petricek is a computer scientist, book author, and open source developer. He wrote a popular book called *Real-World Functional Programming (Manning)* and is a lead developer of several F# open source libraries. He also contributed to the design of the F# language as an intern and consultant at Microsoft Research. He is a partner at fsharpWorks (*http://fsharpworks.com/*) where he provides training and consulting services.

Tomas recently submitted his PhD thesis at the University of Cambridge, focused on types for understanding context usage in programming languages. His most recent work also includes two essays that attempt to understand programming through the perspective of philosophy of science.