Data Science with Kotlin

Stefano Furi

Jun 01, 2023

CONTENTS

Data Science has rapidly evolved over the years, with numerous programming languages available to perform data analysis tasks. **Python** has become a go-to language for data scientists due to its extensive libraries and tools like NumPy or pandas. On the other hand, **Kotlin**, a language initially designed for Android development, has gradually gained popularity in other domains, including data science recently.

Python for Data Science

Python has been the dominant language for data science for more than twenty years, and there are several reasons why it remains the preferred choice for many data scientist:

- Python has a large and active *community* of developers.
- Extensive libraries and tools, especially for data science.
- Easy to learn and use.
- *Flexibility* due to Python's general purpose nature.

The most used and well known libraries for data science with python are:

- **NumPy**
- **Pandas**
- **Matplotlib**

Kotlin for Data Science

While Kotlin is a relatively new language that has been gaining popularity, the ecosystem of libraries for data-related tasks created by the Kotlin community is rapidly expanding. Even if it is not widely adopted in the data science community, the advantages of using Kotlin include:

- Kotlin is a **statically typed** langauge, improving bug prevention (spotting many errors at compile time!), code quality and performances.
- Concise and expressive syntax, which can improve code readability and maintainability.
- Interoperability with Java, seamlessly integrating with existing Java code and libraries.
- Kotlin's support for functional programming techniques, such as immutability, higher-order functions, and lambdas, can be very useful for data science tasks as it allows for concise and efficient processing of large datasets.

In this document, we will go through three libraries that should cover the Python tools mentioned above:

- **Multik**
- **Dataframe**
- **Lets-Plot**

It's important to notice that Multik and DataFrame are very "young" libraries, meaning that they are not as optimized and supported as Python's data science libraries. The goal of this document is to illustrate and guide the reader on what, why and how Kotlin can be a viable alternative for data analysis tasks, taking advantage of its core features like static typing, functional programming techniques and its maintainability.

Working with Jupyter Notebook

Nowadays it is more and more popular to work with Jupyter Notebooks for Data Science projects. Data visualization is a key aspect about this job, and with a notebook, it's very easy to load, process, manipulate and visualize data.

Each notebook has to connect to a **kernel**, which provides the interpretation/compiling of the code inside a notebook.

There are kernels for most of the python versions, but also kernels that support the R programming language, Ruby and a lot more!

Fortunately, Kotlin Jupyter Kernel provide a kernel that make possible the use kotlin inside a Jupyter Notebook, and it adds support for libraries like Kotlin `DataFrame` and `Lets-Plot` for a proper rendering of Dataframes and Plots respectively.

In the repository linked to this page, the `README.md` contains some summarized instructions for downloading and enabling an environment to work with Jupyter Notebooks and Kotlin inside a notebook using Kotlin Jupyter Kernel.

Note: this document was originally a *website*, so some outputs, especially of Kotlin DataFrame, may be somehow difficult to read. This is due to how the output of those object is rendered (HTML) in a Jupyter Notebook. Whenever it is possible, a call to `print` method is invoked, but keep in mind that this creates a plain text output as in this document.

If you want to see the correct rendering of dataframes and plots interactions, please visit this document website at https://s-furi.github.io/uni-internship.

CHAPTER

ONE

WORKING WITH ARRAYS AND MATRICES

Vectors and matrices are fundamental mathematical objects that play a crucial role in many data science tasks, such as linear algebra, statistics and machine learning. Manipulating and processing these objects is often a critical step, for example in data preprocessing.

Python's **NumPy** is a widely used for scientific computing, and Kotlin **Multik** provides similar set of data structures and functions as NumPy, with some limitations.

In this chapter, we will explore how to create and manipulate N-Dimensional Arrays, perform basic operations and using some built-in functions for more advanced operations.

1.1 Kotlin NDArrays

1.1.1 Mutlik *Engines*

Multik engines are a key feature of this library, which provide a way to perform computation on arrays using different backends.

In the context of Multik, a **backend** is a software component that provides the implementation of *specific array operations* (i.e. a backend might implement the algorithm for matrix multiplication). Multik lets the user select a backend (a default one is loaded when specifying anything), depending on his specific needs and requirements. Currently, there are several Multik engines available, including:

- JVM Engine: default engine, runs on the JVM and provides basic array operations.
- Apache Arrow Engine

Thanks to **Multik** and the standard Collections library, it's easy to create, access and manipulate N-dimensional arrays and matrices.

You can import Multik using Jupyter's magic command `%use`, for importing Multik library and dependencies.

```
%use multik
```

We can then create a simple 1D array as follows:

```
val arr: NDArray<Int, D1> = mk.ndarray(mk[1, 2, 3])
arr
```

```
[1, 2, 3]
```

Notice the type definition of the Array, where we must specify:

- **Type**: Any Kotlin subtype of `Number` (`Boolean` is not permitted).
- **Dimension**: Multik defines 5 `Dimension` types: `D1`, `D2`, `D3`, `D4`, `DN`. `DN` is used in all the cases where the dimension is not known in advance, or when the dimension size is major than 4.

In the same way, we can create a 2 dimensional array, but this time we let the compiler infer the type and dimension

```
val arr = mk.ndarray(mk[mk[1, 2, 3], mk[4, 5, 6]])
arr
```

```
[[1, 2, 3],
[4, 5, 6]]
```

We can also create an array from *Kotlin Collections* **List** or **Set**

```
val arr = mk.ndarray(listOf(1, 2, 3))
arr
```

```
[1, 2, 3]
```

Already from these examples, it's quite easy to spot the similarities with NumPy API, but at the same time, Kotlin's static typing can be a bit tedious at first comparing to NumPy, but the effort will often pay off.

1.1.2 Creation

We will go through simple `NDArrays` creation methods. Most of the times they're very similar to NumPy's methods, so the following examples should be already familiar.

Creating equally spaced arrays with `arange` and `linspace`

```
val a = mk.linspace<Double>(0, 1, 5)
val b = mk.arange<Int>(0, 10, 2)

println("a -> $a")
println("b -> $b")
```

```
a -> [0.0, 0.25, 0.5, 0.75, 1.0]
b -> [0, 2, 4, 6, 8]
```

Generating an array of Zeros and Ones

```
val zrs = mk.zeros<Double>(10)
val ons = mk.ones<Double>(10)
println("Zeros ->$zrs")
println("Ones -> $ons")
```

```
Zeros ->[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
Ones -> [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

We can then map the array to a matrix using the `reshape` method, which takes in input an arbitrary number of dimensions size along which the array will be mapped.

```
mk.zeros<Int>(50).reshape(2, 5, 5) // three dimensional matrix with (z, x, y) = (2, 5,
↪ 5)
```

```
[[[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]],

[[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0],
[0, 0, 0, 0, 0]]]
```

It's also possible to create an `NDArray` providing a **lambda** for constructing it, considering that the matrix will be built upon a `arange` vector.

```
mk.d3array(2, 5, 5) { it % 2 }
```

```
[[[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0]],

[[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1]]]
```

Same as

```
mk.arange<Int>(50).map { it % 2 }.reshape(2, 5, 5)
```

```
[[[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0]],

[[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1],
[0, 1, 0, 1, 0],
[1, 0, 1, 0, 1]]]
```

Arithmetic with NDArrays

In the current version of Multik, an `NDArray` object support all arithmetic operations, enabling what are called as **vectorized** operations. Vectorization is very convenient because enable the user to express batch operations on data without writing any for loops (and also could possibly enhance performances).

```
val arr = mk.ndarray(mk[mk[1.0, 2.0, 3.0], mk[4.0, 5.0, 6.0]])
arr
```

```
[[1.0, 2.0, 3.0],
[4.0, 5.0, 6.0]]
```

```
arr * arr
```

```
[[1.0, 4.0, 9.0],
[16.0, 25.0, 36.0]]
```

```
1.0 / arr
```

```
[[1.0, 0.5, 0.3333333333333333],
[0.25, 0.2, 0.16666666666666666]]
```

```
arr dot arr.transpose()
```

```
[[14.0, 32.0],
[32.0, 77.0]]
```

Warning: Operations among equal shaped arrays and matrices are always allowed. As the current version of Multik (v0.2.0), there is no native support for array **broadcasting**. In *Broadcasting* subsection there is an explanation on how to archieve array broadcasting using tensor algebra provided by Kmath.

Multik supports two separate packages for linear algebra basic operations (package `api.linalg`) and vectorized math operations (package `api.math`)

```
mk.math.cumSum(arr, axis = 1)
```

```
[[1.0, 3.0, 6.0],
[4.0, 9.0, 15.0]]
```

```
val sqrMat = mk.ones<Double>(3, 3)
// Frobenius norm of the matrix (default when calling `norm()`)
mk.linalg.norm(sqrMat, norm = Norm.Fro)
```

```
3.0
```

```
mk.math.sin(arr)
```

```
[[0.8414709848078965, 0.9092974268256817, 0.1411200080598672],
[-0.7568024953079283, -0.9589242746631385, -0.27941549819892586]]
```

The `linalg` package contains also methods for solving linear matrix equations and matrix decomposition methods.

```
// solve the linear system
val a = mk.d2array(3, 3) { it * it }.asType<Double>()
val b = mk.ndarray(mk[54.0, 396.0, 1062.0])

val res = mk.linalg.solve(a, b).map { it.roundToInt() }
println(" x = $res")

// check
(a dot res.asType<Double>()) == b
```

```
 x = [0, 6, 12]
```

```
true
```

`linalg` also offers matrix inversion and `qr` decomposition

```
val X = mk.rand<Double>(5, 5)
val mat = X.transpose().dot(X)
mk.linalg.inv(mat)

val (q, r) = mk.linalg.qr(mat)
q
```

```
[[-0.4455096002684611, 0.7661594686085427, -0.4180257373626529, 0.
↪09998260091335262, -0.17256542859774493],
[-0.47356278816208003, -0.017000898664802536, 0.5693545778223102, 0.
↪6553609275482128, 0.1476030977610939],
[-0.3431030139558301, -0.5628936372258248, -0.6733192686188686, 0.2816843981183238,
↪ 0.18090366869794736],
[-0.5335575506054557, -0.3048919117671395, 0.18219153993040013, -0.
↪4048682348390162, -0.6521082866572192],
[-0.4181580368032221, 0.05387312331511457, 0.12057053080886082, -0.
↪5632416412453933, 0.7003307386576824]]
```

```
r
```

```
[[-3.1981015896243132, -4.619932223138458, -4.3530082252744196, -5.866238542026882,
↪ -4.263853937309269],
[0.0, -0.70967243069802, -1.1920823077725626, -1.1806739744673553, -0.
↪6274909203715776],
[0.0, 0.0, -0.17055525776998118, -0.030301677110024614, 0.04803530411110584],
```

(continues on next page)

(continued from previous page)

```
[0.0, 0.0, 0.0, -0.1517931429191967, -0.29173484960727464],
[0.0, 0.0, 0.0, 0.0, 0.18699845439234958]]
```

For more, visit Multik's documentation for `linalg` and `math`

Indexing and Slicing

There are many ways to select subset of data of and `NDArray`.

For one dimensional arrays is straightforward:

```
val arr = mk.arange<Int>(10)
arr
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
// basic indexing
arr[2]
```

```
2
```

```
// range indexing
println(arr[1..3])
// the behavior is different using kotlin keyword `until`
println(arr[1 until 3])
```

```
[1, 2, 3]
[1, 2]
```

For Multi-Index arrays the *range* operator can be used like:

```
val a = mk.linspace<Int>(0, 20, 10).reshape(5, 2)

// single elemt selection
a[3, 1]
// row selection
a[0]
// range selection left inclusive
a[0..2] // != a[0 until 2] -> left exlusive
// selecting first column of first 3 rows
a[0..2, 0]
```

```
[0, 4, 8]
```

Warning: Slices are **copies** of the source array, unlike NumPy that generates a *view* on the original array.

```
var b = a[0..2, 0]
a[0..2, 0] === b // false!
```

NDArrays does not support boolean indexing, but filters can be applied with the `filter()` method, like every `Collection` in Kotlin's standard library.

For each "indexed functional method" that can be applied to kotlin collections, like `mapIndexed`, `forEachIndexed`, `filterIndexed`, Multik offers the counterpart for multidimensional arrays (of the form `*MultiIndexed()`), where the index is an `intArray()` of the combination of the indices of the current element.

A full list of those methods, and more, can be found int the `ndarray.operations` package.

`inplace` context

Multik provides `inplace` context for operating directly inside an array, modifying it's structure. We can then call the default `math` context for applying *inplace* mathematical transformation on the elements of the array.

```
val a = mk.d1array(10) { it * 10 }.asType<Double>()
val b = mk.arange<Double>(10)

println("Before -> $a")
a.inplace {
    math {
        (this - b) * b
        abs()
    }
}

println("After -> $a")
```

```
Before -> [0.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0]
```

```
After -> [0.0, 9.0, 36.0, 81.0, 144.0, 225.0, 324.0, 441.0, 576.0, 729.0]
```

Note that this approach violates the immutability provided by Kotlin's `val` keyword!

Mathematical and Statistical Methods

Both `api.stat` and `api.math` provides several methods for computing basic statistics like `mean`, `median`, `max`, `argMax`, `sum` and `cumSum`, some of them are callable by the instance method or using the top-level Multik function.

If we want to compute the mean of a matrix, we must specify whether we want to compute the mean for the *whole* matrix, or row/column-wise with `mean()` and `meanD2()` respectively (for higher dimensional matrices, the `meanD*` variant covers dimension up to 4, then the `meanDN` has to be used).

```
val arr = mk.rand<Double>(5, 4)

println("mean of the whole matrix -> ${mk.stat.mean(arr)}")
println("mean across columns -> ${mk.stat.meanD2(arr, 0)}")
println("mean across rows -> ${mk.stat.meanD2(arr, 1)}")
```

```
mean of the whole matrix -> 0.5337780694583083
```

```
mean across columns -> [0.5693749050821688, 0.4882305048890908, 0.709120266412919, 0.3683866014490545]
```

```
mean across rows -> [0.6126098591417105, 0.4506730453629224, 0.60712191472837, 0.5115118187995794, 0.4869737092589591]
```

Example: Random Walks

This example has been taken from the book *Python for Data Analysis*

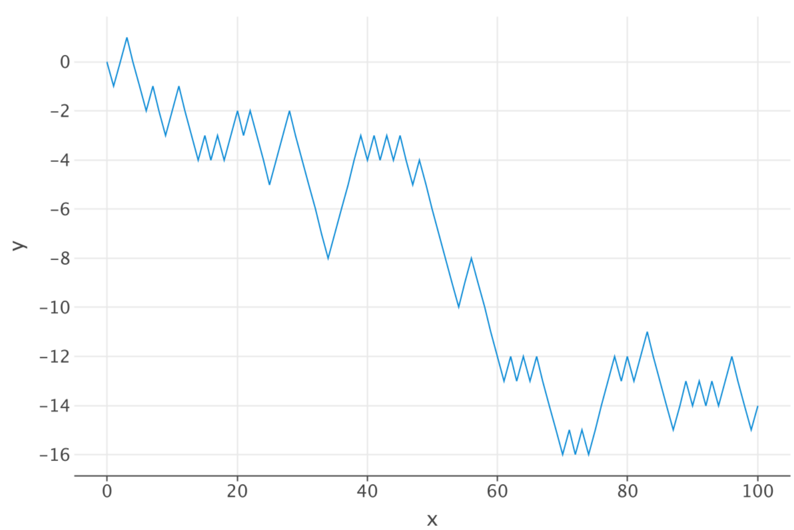
A **simple** pure kotlin implementation can be summarized as follows

```kotlin
import java.util.Random
var position = 0
val walk = mutableListOf(position)
val rand = Random(123456)
val steps = 1000
for (i in 0 until steps) {
    val step = if (rand.nextBoolean()) 1 else -1
    position += step
    walk.add(position)
}
```

We can plot that walk

```
%use lets-plot
```

```kotlin
ggplot() { x=(0..100).toList() ; y= walk.slice(0..100) } +
    geomLine()
```



We can get the same result computing the cumulative sum of the random steps. (note that kotlin `List` can be used for creating one dimensional vector)

```
val nsteps = 1000

val draws = mk.d1array(nsteps) { rand.nextInt(0, 2)}
val steps = mk.d1array(draws.size) { if (draws[it] > 0) 1 else -1 }

val walk = steps.cumSum()
walk.min()
```

```
-20
```

```
walk.max()
```

```
9
```

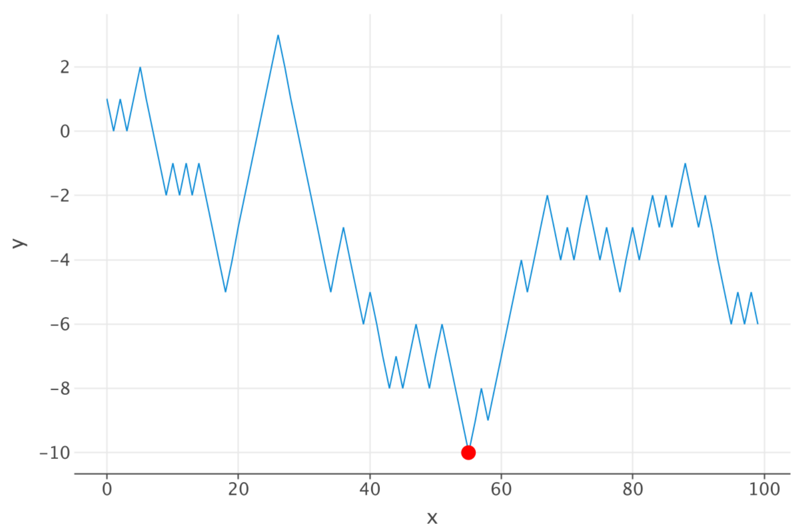
We can get the index of the walk when we cross the 10 steps above or below the origin, even if we have to pass trough a list, because Multik does not support `Boolean` arrays.

```
val first10 = abs(walk).toList().map { it >= 10 }.indexOfFirst { it }
first10
```

```
55
```

And we can plot the first 100 steps similarly as we did before, illustrating where is the point we computed before

```
ggplot() { x = mk.arange<Int>(100).toList() ; y = walk[0..99].toList() } +
    geomLine() +
    geomPoint(x = first10, y = walk[first10], color="red", size=5.0)
```



Multik's `ndarrays` become handy in case we want to simulate many random walks at once, representing them in a matrix, where each row is a walk and each column of that row is a step:

```
val nwalks = 5000
val nsteps = 1000

val draws = mk.d2array(nwalks, nsteps) { rand.nextInt(0, 2)}
val steps = draws.map { if (it > 0) 1 else -1 }
```

```
val walks = mk.math.cumSum(steps, axis = 1)
```

```
walks.min()
```

```
-105
```

```
walks.max()
```

```
107
```

We can get all the runs that reach 50 or -50 using `mapMultiIndexed()` that will preserve the matrix structure.

```
mk.math.maxD2(
    walks.mapMultiIndexed { _, elem -> if (elem >= 50) 1 else 0 }, 1).sum()
```

```
604
```

1.1.3 Broadcasting

Array with different shapes can be broadcasted if this condition is satisfied (from NumPy documentation):

> In order to broadcast, the size of the trailing axes for both arrays in an operation must either be the same size or one of them must be one.

Multik does not have a direct way to broadcast two arrays with different shapes, but **KMath** tensor module provides an implementation in the `DoubleTensorAlgebra` context.

KMath's modules needed for broadcasting are not available using the `%use` magic command, but we can tell Koltin's Jupyter Kernel to import them. KMath also has a set of API for transforming a Multik `NDArray` into a tensor, a more familiar object to Kmath.

We use the `@file:DependsOn` command for importing `core`, `multik` and `tensors` modules of KMath inside the project:

```
@file:DependsOn("space.kscience:kmath-core:0.3.0")
@file:DependsOn("space.kscience:kmath-tensors:0.3.0")
@file:DependsOn("space.kscience:kmath-multik:0.3.0")
```

For being able to use them, we import the needed packages.

```
import space.kscience.kmath.multik.MultikTensor
import space.kscience.kmath.tensors.core.DoubleTensorAlgebra
import space.kscience.kmath.tensors.core.tensorAlgebra
import space.kscience.kmath.tensors.core.withBroadcast
import space.kscience.kmath.tensors.core.DoubleTensor
```

Of course we could have used KMath tensors for all the computation, but KMath tensors are in some way harder to manipulate that Multik's `NDArrays`, so in this example we will see how we can compute broadcasting starting from two `NDArrays`.

Thanks to Kotlin *extensions functions* we can map the + operator between two `MultikTensors` objects to compute broadcasting if needed.

```
fun NDArray<Double, DN>.asMultikTensor(): MultikTensor<Double> = MultikTensor(this)

fun brAdd(a: MultikTensor<Double>, b: MultikTensor<Double>): DoubleTensor? {
    var res: DoubleTensor? = null
    return Double.tensorAlgebra.withBroadcast {
        a + b
    }
}

operator fun MultikTensor<Double>.plus(other: MultikTensor<Double>): DoubleTensor? =
 ↪brAdd(this, other)
```

With those functions, the following code produces the expected output.

```
val a: NDArray<Double, DN> = mk.arange<Double>(40).reshape(2, 4, 5).asDNArray()
val b: NDArray<Double, DN> = mk.arange<Double>(20).reshape(4, 5).asDNArray()

a.asMultikTensor() + b.asMultikTensor()
```

```
DoubleTensor(
   [[[ 0.0      ,  2.0      ,  4.0      ,  6.0      ,  8.0      ],
     [ 1.0e+1   ,  1.2e+1   ,  1.4e+1   ,  1.6e+1   ,  1.8e+1   ],
     [ 0.2e+2   ,  2.2e+1   ,  2.4e+1   ,  2.6e+1   ,  2.8e+1   ],
     [ 0.3e+2   ,  3.2e+1   ,  3.4e+1   ,  3.6e+1   ,  3.8e+1   ]],
    [[ 0.2e+2   ,  2.2e+1   ,  2.4e+1   ,  2.6e+1   ,  2.8e+1   ]],
    [ 0.3e+2   ,  3.2e+1   ,  3.4e+1   ,  3.6e+1   ,  3.8e+1   ]],
   [ 0.4e+2   ,  4.2e+1   ,  4.4e+1   ,  4.6e+1   ,  4.8e+1   ]],
  [ 0.5e+2   ,  5.2e+1   ,  5.4e+1   ,  5.6e+1   ,  5.8e+1   ]]]
)
```

There is no such thing as a `np.newaxis` for specifying the missing dimension to broadcast along. We must reshape the arrays in order to broadcast properly.

Consider this example, where we compute the mean of every row, and we want to add the mean to the original matrix. According to broadcasting rules, if we want to perform broadcasting across axes other than axis 0, the "broadcast dimensions" must be 1 in the smaller array. Being `b` of shape `(4, 5)` and the mean vector is `(, 4)`, we must reshape and insert a "1" in the broadcasting dimension, the second one.

```
val mean = mk.stat.meanD2(b.asD2Array(), 1)
                .reshape(b.shape[0], 1)
                .asDNArray().asMultikTensor()


b.asMultikTensor() + mean
```

```
DoubleTensor(
   [[ 2.0      ,  3.0      ,  4.0      ,  5.0      ,  6.0      ],
    [ 1.2e+1   ,  1.3e+1   ,  1.4e+1   ,  1.5e+1   ,  1.6e+1   ],
```

(continues on next page)

(continued from previous page)

```
    [ 2.2e+1   ,  2.3e+1   ,  2.4e+1   ,  2.5e+1   ,  2.6e+1   ],
    [ 3.2e+1   ,  3.3e+1   ,  3.4e+1   ,  3.5e+1   ,  3.6e+1   ]]
)
```

Notes:

- The `withBroadcast` context is claimed to be *unstable* and could change in the future.
- For large arrays, the chain of conversions can be very slow in performance-critical code.
- No direct ways to re-convert a `DoubleTensor` to a Multik `NDArray` were found: the only way is to manually copy element by element from the `DoubleTensor` to a new `NDArray`.

1.1.4 Conclusions

In this chapter was presented Kotlin's Multik library for working with N-Dimensional Arrays. The set of data structure and methods are very close to NumPy's, which makes it very easy to learn if the developer has a basic knowledge of NumPy. However, NumPy has more that 20 years of development and fine tuning, resulting in one of the best libraries for scientific computation: the better support for linear algebra operation, array broadcasting and the possibility to use multidimensional arrays with *non-numeric* data types, makes it a very flexible and powerful library. On the other hand, Multik can be used with libraries like Kmath and Apache Commons Math, or any other library developed in Java, making it very powerful with the right set of components at the need of the developer (i.e. array broadcasting and more advanced linear algebra can be accomplished with the use of Kmath). Please note that Multik is very efficient with basic mathematical computation, and for a little bit more, Kmath connectors for Multik can really help archiving more difficult tasks (i.e. integration).

Kotlin ecosystem for scientific computation is still young, but with the joint use of the right libraries, a lot can be archived.

CHAPTER

TWO

DATA ANALYSIS AND MANIPULATION TOOLS

When dealing with large datasets, having efficient tools and libraries can greatly improve the speed and accuracy of these tasks. In the following section, it will be presented **Kotlin DataFrame**, a library that is inspired largely by **pandas**, kotlin collections and Krangl. This library tries to integrate the dynamic nature of data, and the static typing and type safeness of Kotlin, through a series of techniques that will be explored later on. As NumPy, **pandas** has been a de-facto standard when dealing with loading, manipulating and presenting data when it comes to data analysis tasks.

In the following section we will see and discuss how Kotlin and Python differs and resemble when involved in handling data with the tools provided by Kotlin DataFrame in contrast to Python's pandas.

2.1 Data handling with Dataframe

Kotlin DataFrame was largely inspired by pandas structures, but despite this, it has his own characteristics that make Data Analysis very understandable and concise, but more importantly, the *type safe*. Most of it's readability comes from functional languages chains of transformations (the data pipeline), and the use of DSL that makes it's syntax closer to natural language.

In this chapter we will cover the basics to work with Kotlin DataFrame's data structures and its basic operations.

2.1.1 Overview

Let's see how a simple workflow can look like using Kotlin DataFrame.

DataFrame is built-in kotlin jupyter kernel, so the magic command `%use` will load the needed packages.

```
%use dataframe
```

For creating a dataframe from scratch, we can follow several approaches, and the most used are:

1. Creating `Column` objects for storing data, and assign columns to a `DataFrame`

```
val names by columnOf("foo", "bar", "baz")
val numbers by columnOf(1, 2, 3)

val df = dataFrameOf(names, numbers)
df.print()
```

```
  names numbers
0   foo       1
1   bar       2
2   baz       3
```

2. Providing a series of `Pair<String, Any>` or a `Map<String, Any>`

```
val df = dataFrameOf(
    "names" to listOf("foo", "bar", "baz"),
    "numbers" to listOf(1, 2, 3)
)

df.print()
```

```
  names numbers
0   foo       1
1   bar       2
2   baz       3
```

3. Providing an `Iterable<Column>`

Note: For more, please refer to DataFrame constructions methods

```
val values = (1..5).map { List(10) { x -> (x - it) * 10 }.toColumn("$it") }

dataFrameOf(values).print()
```

```
    1   2   3   4   5
0 -10 -20 -30 -40 -50
1   0 -10 -20 -30 -40
2  10   0 -10 -20 -30
```

```
3  20  10   0 -10 -20
4  30  20  10   0 -10
5  40  30  20  10   0
6  50  40  30  20  10
```

```
7  60  50  40  30  20
8  70  60  50  40  30
9  80  70  60  50  40
```

We can load a dataset, and compute some statics

```
val df = DataFrame.readCSV(
            "../resources/example-datasets/datasets/stock_px.csv",
            header = listOf("date", "AAPL", "MSFT", "XOM", "SPX"),
            skipLines = 1
```

(continues on next page)

(continued from previous page)

```
        )
df.print(5)
```

```
               date AAPL  MSFT   XOM    SPX
 0 2003-01-02T00:00 7.40 21.11 29.22 909.03

 1 2003-01-03T00:00 7.45 21.14 29.24 908.59
 2 2003-01-06T00:00 7.45 21.52 29.96 929.01

 3 2003-01-07T00:00 7.43 21.93 28.95 922.93
 4 2003-01-08T00:00 7.28 21.31 28.83 909.93
...
```

```
val means = df.groupBy { date.map { it.month } }.mean() // mean for each month, for␣
↪each stock
means.print()
```

```
          date       AAPL      MSFT       XOM          SPX

  0    JANUARY 111.954696 24.842265 57.842873 1173.562486

  1   FEBRUARY 109.541628 23.511686 58.943140 1164.714244

  2      MARCH 114.255829 23.015176 58.378945 1156.397487

  3      APRIL 120.849140 23.710699 60.144301 1186.491237

  4        MAY 126.349468 23.413245 59.813404 1203.199202

  5       JUNE 129.076753 23.191959 59.641598 1191.002268

  6       JULY 133.117884 23.631640 59.988095 1185.564656

  7     AUGUST 137.710754 23.700302 59.219497 1182.175678

  8  SEPTEMBER 142.788387 24.046344 59.703387 1188.879570

  9    OCTOBER 130.348441 24.271774 59.754677 1182.235806

 10   NOVEMBER 122.318712 25.068650 60.125583 1190.728466

 11   DECEMBER 125.574444 25.307193 61.365731 1202.463216
```

```
val yearMeans = df.groupBy { date.map { it.year} }.mean() // compute the means for␣
↪each year
    // mappig each stock and it's value to two separate columns
    .gather { AAPL and MSFT and XOM and SPX}.into("stock", "value")
    .rename { date }.into("year")
yearMeans.print(10)
```

```
   year stock       value
 0 2003  AAPL    9.272619
 1 2003  MSFT   20.595119
```

```
 2 2003   XOM   30.211111
 3 2003   SPX  965.227540
 4 2004  AAPL   17.763889
```

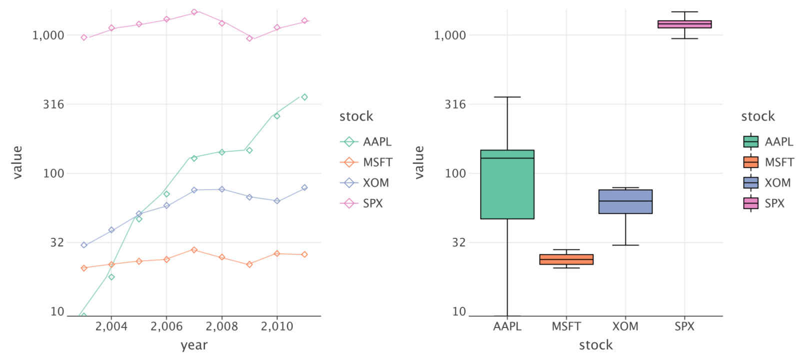
```
 5 2004  MSFT   21.850437
 6 2004   XOM   38.875437
 7 2004   SPX 1130.649444
```

```
 8 2005  AAPL   46.675952
 9 2005  MSFT   23.072421
...
```

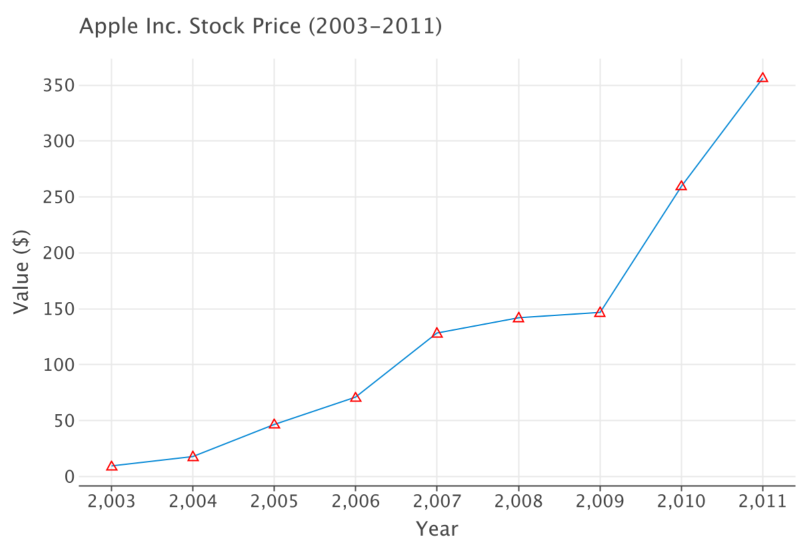
We can then **plot** those statistics with `lets-plot` library (we will explore plotting later in chapter *five*):

```
%use lets-plot
```

```
val p1 = ggplot(yearMeans.toMap()) { x="year" ; y="value" ; color="stock" } +
    geomLine(stat = Stat.identity, position = positionDodge(0.5), alpha = 0.7) +
    geomPoint(size=3.0, shape = 5) +
    scaleYLog10()

val p2 = ggplot(yearMeans.toMap()) { x="stock" ; y="value" } +
    geomBoxplot() { fill = "stock"} +
    scaleYLog10()


val p = gggrid(listOf(p1, p2))
p
```




```
val plt = ggplot(yearMeans.filter { stock == "AAPL"}.toMap() )+
    geomLine(stat = Stat.identity) { x="year" ; y="value" } +
    geomPoint(color="red", size=3.5, shape = 2) { x="year" ; y="value" } +
    ylab("Value ($)") +
    xlab("Year") +
    ggtitle("Apple Inc. Stock Price (2003-2011)")
plt
```



2.1.2 DataFrame Architecture

Working inside a Jupyter Notebook

The strength of Kotlin DataFrame is its ability to conciliate the dynamic nature of data, with Kotlin's strong typing, resulting in a type safe library for working with data. In contrast to pandas, when we compute operations in a dataframe, we know at **compile time** the types of the columns of the dataframe, and their results.

This is true when working with Jupyter Notebooks, because every time a dataframe is loaded and its cell executed, a new `DataSchema` is created for the dataframe specified. The **`DataSchema`** provides a way to define and manage the *metadata* of a `DataFrame`, including columns names and types, and nullable flags. It is used to ensure that the data in a `DataFrame` is consistent and can be processed correctly.

So if we would create a dataframe with one column of names like:

```
val names by columnOf("foo", "bar", "baz")
val df = dataFrameOf(names)
```

The following code is called implicitly (you can see this output using the magic command `%trackExecution`).

```
Executing:
@DataSchema interface _DataframeTypel
val ColumnsContainer< _DataFrameTypel> .names: DataColumn<String> @3vmName (" _
↪DataframeTypel names") get() = this ("names"'] as DataColumn<String>
val DataRow< _DataFrameTypel>.names: String @JvmName (" _DataFrameTypel_names") get)␣
↪= this ("names") as String
val ColumnsContainer‹ DataFrameType1?>.names: DataColumn<String?> @JvmName ("Nullable␣
↪DataFrameTypel names') get() = this ("names"] as DataColumn‹String?>
val DataRow< _DataFrameTypel?>.names: String? @JvmName ("Nullable _DataFrameTypel_
↪names") get () = this ("names") as String?
df.cast<_DataFrameType1>()
```

A custom type of the dataframe is created, that will be used to define the columns containers. For each one of them, the above *extension functions* will be created. Note that this chain of operations is computed even when reading a dataset from file (a full explanation on how data schema is dynamically created, can be found here).

Using this method, each time we execute a cell the new columns types will be defined as above, so that in the next cells, we know at compile time the data type of the columns.

Working inside an IDE

When working inside an IDE, we can ensure type safety in two ways:

- Defining a custom `DataSchema`, and use it inside a **Gradle** project (see documentation for more).
- Defining columns object.

In both ways, the *extension properties API* can be used, providing strong type checking at compile time (we will discuss various column accessors API later).

2.1.3 DataFrame Data Structures

The DataFrame library, defines the following data abstractions:

- `DataColumn`: is a named, typed and ordered collection of elements.
- `DataFrame`: consists of one or several `DataColumns` with unque names and equal size.
- `DataRow` is a single row of a `DataFrame` and provides a single value for every `DataColumn`.

Because we are dealing with structured data, Dataframe provides **hierarchical** data structures using two special types of columns:

- `ColumnGroup` is a group of columns
- `FrameColumn` is a column of dataframes

This makes easy the creation of tree structures among data (very handy when reading `JSON`s files). We can look at `ColumnGroup` and `FrameColumn` as pandas `MultiIndex` objects: they both try to express a hierarchical structure of data.

By nature, data frames are dynamic objects, column labels depend on the input source and also new columns could be added or deleted while wrangling. Kotlin in contrast, is a statically typed language and all types are defined and verified ahead of execution.

For this reason, the Kotlin DataFrame library provide four different ways to access columns:

- String API
- Columns Accessors API
- KProperties API
- Extension Properties API

For detailed usage, refer to the official documentation of column accessors.

The string API is the simplest and **unsafest** of them all. The main advantage is that it can be used at any time, including when accessing new columns in chain calls, so that this call can be made:

```
df.add("age") { ... }
    .sum("age")
```

If you're not working in an Jupyter Notebook, *Column Accessor API* provide type-safe access to columns, but does not ensure that the columns really exist in a particular dataframe. Similarly, when working in an IDE, *KProperties API* is useful when you've already declared classes in you application business logic with fields that correspond columns of a DataFrame.

Otherwise, if you're working inside a notebook, you can use Extension Properties API, which are the safest and convenient to use, with the trade-off of execution speed in the moment of generation.

`DataColumn`

A `DataColumn` object is the equivalent of a pandas `Series` object; both represent a one dimensional array of data with a specific data type. Every `DataColumn` object has a unique type and several data mapped into rows.

As pointed out above, we can create a column object with the `by` keyword

```
val col by columnOf("a", "b", "c")
```

Following this approach, the name of the column is the name we gave to the variable, and we can use the *column accessor* API for better type safety.

Similarly, we can explicitly cast the column to a `Ktype`, and the result will be a `ColumnAccessor`:

```
val col by column<Double>("values")
```

With the `ColumnAccessor`, we can convert it to a `DataColumn` using `withValues` function:

```
val age by column<Int>()
val ageCol1 = age.withValues(15, 20)
val ageCol2 = age.withValues(10..20)
```

`List` and `Set` from the standard library have an *extension function* that can convert the collection to a column with the provided name.

```
val col = List(5) { it * 2 }.toColumn("values")
```

A column can be of three types:

- `ValueColumn`: stores primitives data (by now, the underlying structure is a `List`)
- `ColumnGroup`: stores nested columns
  - For referencing a nested column we can use

```
val name by columnGroup()
val firstName by name.column<String>()
```

- `FrameColumn`: stores a nested `DataFrame`

`DataFrame`

A `DataFrame` represent a list of `DataColumns`. Columns in a `DataFrame` must have **equal size** and **names**.

The simplest way to create a `DataFrame` is using the function `dataFrameOf`

```
val df = dataFrameOf("name", "age")(
    "Alice", 15,
    "Bob", 20,
    "Charlie", 25
)
df.print()
```

```
      name age
0    Alice  15
1      Bob  20
2  Charlie  25
```

For all the methods for building a dataframe, see the official documentation

Unlike pandas, Kotlin DataFrame does not implement an explicit **`Index` object**, meaning that the way we compute operations on `pandas.DataFrame` can be very different when working with Kotlin `DataFrame` object. The nature of the dataframe is quite different between the two libraries, and both of them has its pros and cons.

In the following sections we will see most of the operations that could be made on top of `DataFrame`.

2.1.4 Operations Overview

As said before, data transformations pipelines are designed in functional style so that the whole processing can be represented as a sequential chain of operations. `DataFrames` are immutable, and every operations return a copy of the object instance *reusing* underlying data structures as much as possible.

Operations can be divided in three categories:

- **General Operations**: all basic operations that can be called on a dataframe (e.g. `schema()`, `sum()`, `move()`, `map()`, `filter()`, ...)
- **Multiplex Operations**: more complex operations that does not return a new `DataFrame` immediately, instead they provide an intermediate object that is used for further configurations. Every multiplex operation follows the schema:
  1. Use a column selector to select target columns
  2. Additional configuration functions
  3. Terminal function that returns the modified `DataFrame`

- Most of these operations end with `into` or `with`, and the following convention is used:
  - `into` defines column names for storing the result.
  - `where` defines row-wise data transformation.
- **Shortcut Operations**: shortcut for more general operations (e.g. `rename` is a special case for `move`, `fillNA` is a special case for `update`, …)

Essential Functionalities

This section will walk you through the fundamental mechanics of interacting with the data contained in `DataFrames`.

Indexing, Selection and Filtering

```
val obj = dataFrameOf(
    "letters" to listOf("a", "b", "c", "d"),
    "nums" to listOf(0.0, 1.0, 2.0, 3.0)
)
obj.print()
```

```
  letters nums
0       a  0.0
1       b  1.0
2       c  2.0
3       d  3.0
```

We can access by row with

```
obj[1].print()
```

```
{ letters:b, nums:1.000000 }
```

And we can access by columns with:

```
obj["nums"].print()
```

```
  nums
0  0.0
1  1.0
2  2.0
3  3.0
```

We can select multiple columns with the usual notation:

```
obj["nums", "letters"].print()
```

```
  nums letters
0  0.0       a
1  1.0       b
2  2.0       c
3  3.0       d
```

We can select also ranges (note that the second boundary of the range is *included*)

```
obj[0..2].print() // obj[0 until 3]
```

```
  letters nums
0       a  0.0
1       b  1.0
2       c  2.0
```

The `[...]` operator calls the `get()` method, so:

```
obj[0] == obj.get(0) // true
obj["nums"] == obj.getColumn(1) == obj.getColumn("nums") // true
```

Unlike python, kotlin does not provide filtering inside square brackets, but it offers the filter method that can be more understandable

```
obj.filter { nums.toInt() % 2 == 0 }.print()
```

```
  letters nums
0       a  0.0
1       c  2.0
```

instead of python's:

```
obj[(obj["nums"] % 2 == 0)]
```

Arithmetic Operations

Unlike `pandas.DataFrame`, operations between dataframes in kotlin are not defined by default.

We can still compute row-wise operations with the `update()` method:

```
obj.update { nums }.with { it * 100 }.print()
```

```
  letters  nums
0       a   0.0
1       b 100.0
2       c 200.0
3       d 300.0
```

But the following code will not compile:

```
// obj + obj
```

Function Application and Mapping

Much like NumPy's `ufunc` element wise operations, every Kotlin collection (and then DataFrame `DataColumn`) is provided with the `map` method. In case of a dataframe, we can apply a function for each *column* with the `map` operator.

Most of the time, when we want to compute some row-wise operations, the methods `update()` and `convert()` suit perfectly our needs.

```
val frame = dataFrameOf("b", "d", "e").randomDouble(3)
    .update { colsOf<Double>() }.with { it * 7 }

frame.print()
```

```
         b        d        e
0 5.614869 5.623290 6.741218
1 6.996338 1.677955 0.896543
```

```
2 4.709310 0.802725 6.352933
```

We can now apply a function with `map` on one, some or all columns and get its result as a `List`. Additionally, it's possible to store the result of the function application to a new column using the method `mapToColumn()`. Lastly, the `mapToFrame()` map the function applications along multiple columns inside a new dataframe.

```
frame.map { Math.ceil(it.b) }
```

```
[6.0, 7.0, 5.0]
```

```
frame.mapToColumn("ceiling_b") { Math.ceil(it.b) }.print()
```

```
  ceiling_b
0       6.0
1       7.0
2       5.0
```

```
frame.mapToFrame {
    "new_b" from b ;
    d gt 1.0 into "b_gt_1"
}.print()
```

```
     new_b b_gt_1
0 5.614869   true
1 6.996338   true
2 4.709310  false
```

2.1.5 Conclusions

In this chapter, we have introduced Kotlin DataFrame, a library that provides working with tabular data with all the advantages that the Kotlin languages provides. Coming from Python's pandas, its behavior is very similar and intuitive, but the major differences are:

- DataFrame ensures type safety at compile time (especially when working with Jupyter Notebook).
- Lack of an `Index` object to manipulate (i.e. having a `DateTimeIndex` for indexing).
- pandas use of NumPy ensures optimized arithmetic operations (through vectorization), whereas Kotlin DataFrame does not use an optimized library for representing vectors and matrices, but only Kotlin standard collections.

While pandas offers a more comprehensive set of features and is a more mature library, Kotlin DataFrame offers a useful subset of its capabilities. In the following chapters, we will explore further the capabilities of Kotlin DataFrame and demonstrate its use in various data analysis tasks, from data cleaning to grouping strategies. Overall, Kotlin DataFrame is a valuable addition to the toolkit of any data analyst or scientist working in Kotlin.

CHAPTER

THREE

DATA LOADING AND STORAGE

Accessing data is the first prerequisite when dealing with Data Science. A library that handles data should be able to both read and write data on file system in various file formats. The libraries for data manipulation discussed until now, Kotlin DataFrame and Python's pandas, provides those functionalities.

Koltin DataFrame provides more basic functions than pandas (as said before, DataFrame is quite young and currently under development), especially when it comes to file format. In the current version (v0.10) data reading and writing is supported in four file formats: CSV, JSON, Excel Spreadsheets format and Apache Arrow. On the other hand, pandas offers support for all four of them and even more (see docs for a full list of supported file formats). Note that both DataFrame and pandas can read from file system and URLs.

You can find all the references for DataFrame I/O in the user guide, and plenty of documentation about pandas reading and writing data in this section.

CHAPTER

FOUR

DATA CLEANING AND PREPARATION

Data cleaning and preparation are essential steps in any data analysis project, as data is often messy, incomplete or inconsistent. In order to perform accurate analysis and modeling, it is necessary to clean and preprocess data, which involves tasks such as removing duplicates, filling in missing values and converting data types.

With DataFrame and Kotlin's language features, a high-level, flexible, and fast set of tools enable the analyst to manipulate data into the desired form. All those tools also integrate seamlessly with Kotlin's functional programming capabilities, resulting into a very powerful tool for the accomplishment of various data analysis task.

4.1 Data Preparation with Kotlin

In the following chapter we will see the basic operation for preparing data for further analysis. Most of the covered Kotlin DataFrame's functions are very similar to what pandas counterparts.

```
%use dataframe
```

4.1.1 Handling `null` values

In many dataset and data analysis application, **missing data** occurs commonly.

Thanks to Kotlin's nullable values, we can have a column of a nullable type like:

```
val col by columnOf<String?>("a", "b", null)
col.print()
```

```
   col
0    a
1    b
2 null
```

We can then print for each row if it is null or not, similar to pandas `dataframe.isnull()`:

```
col.map { it.isNullOrEmpty() }.print()
```

```
    col
0 false
```

```
1 false
2  true
```

The great advantage in using Kotlin in this kind of situation is the fact that we have a complete control on how we can handle missing data. Unlike python, kotlin's provides out of the box methods for handling null values, possibly without raising a `NullPointerException` if the developer keeps the context safe with the use of *safe call operators* (`?.`) or explicit null checking.

Moreover, Dataframe offers a series of method for filtering or filling `null` values.

```
val df = dataFrameOf(
    "0" to listOf(1.0, null, null, 2.0),
    "1" to listOf(3.5, 6.0, 4.0, null),
    "2" to listOf(1.0, null, 9.6, 10.0)
)

df.print()
```

```
     0    1    2
0  1.0  3.5  1.0
1 null  6.0 null
2 null  4.0  9.6
3  2.0 null 10.0
```

By default, the method `dropNulls()` drops all the *rows* that contain a null value.

```
df.dropNulls().print()
```

```
    0   1   2
0 1.0 3.5 1.0
```

It is important to notice that Dataframe provides three methods for dropping possible null values:

- `dropNull()`: drops every row with a `null` value
- `dropNaNs()`: drop rows with `Double.NaN` or `Float.NaN` values
- `dropNA()`: removes rows with `null`, `Double.NaN` or `Float.NaN` values

For each method, we can choose which columns we want to check for nulls, for example:

```
df.dropNA(whereAllNA = true).print() // removes rows where ALL values are null
```

```
     0    1    2
0  1.0  3.5  1.0
1 null  6.0 null
2 null  4.0  9.6
3  2.0 null 10.0
```

```
df.dropNA("0").print() // dropping all rows that has null in "0" column
```

```
    0    1    2
0 1.0  3.5  1.0
1 2.0 null 10.0
```

```
// remove rows where col "0" and "2" have null or NaN
df.dropNA(whereAllNA = true) { "0" and "2" }.print()
```

```
      0    1    2
0   1.0  3.5  1.0
1  null  4.0  9.6
2   2.0 null 10.0
```

Instead of dropping null values, there could be the need to fill in missing values. Just like pandas, Dataframe offers similar API.

```
var df = dataFrameOf("a", "b", "c").randomDouble(7)
df.print()
```

```
          a        b        c
0  0.472536 0.737483 0.760267
1  0.716778 0.834889 0.700041
```

```
2  0.208427 0.276698 0.021607
3  0.225339 0.262087 0.411242
4  0.817689 0.401514 0.972627
```

```
5  0.661688 0.059996 0.297823
6  0.741784 0.420173 0.536911
```

```
// dummy Double.NaN filing
df = df.update { a }.at(1..4).with { Double.NaN }
  .update { b }.at(1, 2).with { Double.NaN }
df.print()
```

```
          a        b        c
0  0.472536 0.737483 0.760267
1       NaN      NaN 0.700041
```

```
2       NaN      NaN 0.021607
3       NaN 0.262087 0.411242
4       NaN 0.401514 0.972627
```

```
5  0.661688 0.059996 0.297823
6  0.741784 0.420173 0.536911
```

```
df.fillNA { all() }.withZero().print()
```

```
          a        b        c
0  0.472536 0.737483 0.760267
1  0.000000 0.000000 0.700041
```

```
2 0.000000 0.000000 0.021607
3 0.000000 0.262087 0.411242
4 0.000000 0.401514 0.972627
```

```
5 0.661688 0.059996 0.297823
6 0.741784 0.420173 0.536911
```

pandas offers the filling method `ffill` or `bfill`, that fills missing values with the next or preceding row's value.

We can simulate that behavior with:

```
df.fillNA { all() }.perRowCol { row, col -> row.prev()?.get(col) }.print()
```

```
         a        b        c
0 0.472536 0.737483 0.760267
1 0.472536 0.737483 0.700041
```

```
2      NaN      NaN 0.021607
3      NaN 0.262087 0.411242
4      NaN 0.401514 0.972627
```

```
5 0.661688 0.059996 0.297823
6 0.741784 0.420173 0.536911
```

The example below does not consider the new values that are computed during the before computations. In case we want to fill ALL missing values in that column with the first non null preceding row, we must specify the column we want to modify, and use the method `newValue()`.

```
df.fillNA { a }.with { prev()?.newValue() }.print()
```

```
         a        b        c
0 0.472536 0.737483 0.760267
1 0.472536      NaN 0.700041
```

```
2 0.472536      NaN 0.021607
3 0.472536 0.262087 0.411242
4 0.472536 0.401514 0.972627
```

```
5 0.661688 0.059996 0.297823
6 0.741784 0.420173 0.536911
```

With `fillNA` (or `fillNulls` or `fillNaNs`) you can pass any kind of function inside the `with` construct, for example the row mean (remember to pass `skipNA = true` when computing the mean):

```
df.fillNaNs{ colsOf<Double>() }
    .perCol { it.mean(skipNA = true) }.print()
```

```
         a        b        c
0 0.472536 0.737483 0.760267
1 0.625336 0.376251 0.700041
```

```
2 0.625336 0.376251 0.021607
3 0.625336 0.262087 0.411242
4 0.625336 0.401514 0.972627
```

```
5 0.661688 0.059996 0.297823
6 0.741784 0.420173 0.536911
```

4.1.2 Data Transformation

Data transformation includes filtering, cleaning, converting and updating values.

Thanks to the underlying usage of Koltin's collections for storing data, most collections' manipulations methods can be called on columns or rows.

Suppose we have a dataframe containing data about meat and their quantities

```
val df = dataFrameOf("food", "ounces")(
    "bacon", 4,
    "pulled pork", 3,
    "bacon", 12,
    "pastrami", 6,
    "corned beef", 7.5,
    "bacon", 8,
    "pastrami", 3,
    "honey ham", 5,
    "nova lox", 6,
)
df.print()
```

```
          food ounces
0        bacon      4
1  pulled pork      3
2        bacon     12
```

```
3     pastrami      6
4  corned beef    7.5
5        bacon      8
6     pastrami      3
```

```
7    honey ham      5
8     nova lox      6
```

We could map each meat with it's corresponding animal

```
val meatToAnimal = mapOf(
    "bacon" to "pig",
    "pulled pork" to "pig",
    "pastrami" to "cow",
    "corned beef" to "cow",
    "honey ham" to "pig",
    "nova lox" to "salmon"
)
```

Using mapToFrame let us perform column wise operations, creating a new dataframe with the provided set of instructions.

Mapping the whole column food with the corresponding animal, is the same as applying a map function to a Kotlin collection.

```
df.mapToFrame{
    food.map{ meatToAnimal[it] } into "animal"
    +food
    +ounces
}.print()
```

```
  animal        food ounces
0    pig       bacon      4
1    pig pulled pork      3
```

```
2    pig       bacon     12
3    cow    pastrami      6
4    cow corned beef    7.5
```

```
5    pig       bacon      8
6    cow    pastrami      3
7    pig   honey ham      5
```

```
8 salmon    nova lox      6
```

Notice that the three map methods, offers three ways to yield the result of the mapping of the row expression provided.

- map returns a List from the provided **row expression**.
- mapToColumn returns a new Column from the provided **row expression**.
- mapToFrame returns a new DataFrame from the provided **column mappings**. Here we can specify to keep the old columns in the new frame with the +<col_name> operator.

It is advised to learn more about DataRows for understanding which useful methods DataRows offers for creating *row expressions* or *row conditions*.

```
df.map { meatToAnimal[it.food]}
```

```
[pig, pig, pig, cow, cow, pig, cow, pig, salmon]
```

```
df.mapToColumn("animal") { meatToAnimal[it.food] }.print()
```

```
  animal
0    pig
1    pig
2    pig
3    cow
4    cow
5    pig
6    cow
7    pig
8 salmon
```

In python, the API is very similar:

```
df['animal'] = df['food'].map(lambda x: meat_to_animal[x])
```

Replacing Values

Value replacing can be accomplished with the use of `update` or `convert` methods. They differs only for the return type: `update` modify data in each row, but keeping it's type; `convert` modify data in each row, possibly changing it's type.

```
val values by columnOf(1.0, -999.0, 2.0, -999.0, -1000.0, 3.0)
val df = dataFrameOf(values)
df.update { values }.where { it == -999.0 }.withZero().print()
```

```
   values
0     1.0
1     0.0
2     2.0
3     0.0
4 -1000.0
5     3.0
```

Detecting and Filtering Outliers

Sometimes it's very useful to sport and filter out outliers before doing some computation.

Consider a Dataframe with some randomly distributed data:

```
val df = dataFrameOf("a", "b", "c", "d")
    .randomDouble(1000)
    .update { colsOf<Double>() }.with { it * 5}
df.print(5)
```

```
         a        b        c        d
0 1.513110 2.542493 1.473468 0.408357
```

```
1 2.926424 1.445217 1.021359 2.601634
2 4.217417 0.428976 1.996013 0.071969
```

```
3 4.641088 1.359385 0.760510 2.518607
4 3.326521 3.055039 1.258175 3.238394
...
```

```
df.describe().print()
```

```
  name   type count unique nulls      top freq      mean       std       min ␣
↪median      max
```

```
0     a Double  1000   1000     0 1.513110    1 2.526729 1.422892 0.000444 2.
↪488111 4.993321
```

```
1     b Double  1000   1000     0 2.542493    1 2.517847 1.403702 0.006065 2.
↪497022 4.997296
```

```
2     c Double  1000   1000     0 1.473468    1 2.490796 1.446345 0.000696 2.
↪442725 4.996507
```

```
3     d Double  1000   1000     0 0.408357    1 2.439161 1.430172 0.009126 2.
↪435353 4.999208
```

Suppose we want to find all values exceeding 4.5. Unlike pandas *boolean indexing*, with Dataframe we will apply a row-wise filter.

Note: With pandas we simply would have made:

```
df[(df > 3).any(1)]
```

```
df.filter { it.values().any { it as Double > 4.5 } }.print(5)
```

```
         a        b        c        d
0 4.641088 1.359385 0.760510 2.518607
```

```
1 4.032011 4.840225 4.370720 3.624690
2 0.382613 3.553842 3.962862 4.786169
```

```
3 1.269150 3.498156 4.681507 0.186666
4 2.231474 2.363167 4.552983 4.735957
...
```

Now let's remap all values > 4.5, to be inside that range.

```
df.update { colsOf<Double>() }
    .where { it > 4.5 }
    .withValue(4.5).print(5)
```

```
         a        b        c        d
0 1.513110 2.542493 1.473468 0.408357
```

```
1 2.926424 1.445217 1.021359 2.601634
2 4.217417 0.428976 1.996013 0.071969
```

```
 3 4.500000 1.359385 0.760510 2.518607
 4 3.326521 3.055039 1.258175 3.238394
...
```

4.1.3 Recap

Dataframe power comes from `update` and `convert` values, which they can let us perform any kind of row-wise transformation on data. Those two clauses allows performing a pre filtering operation before updating cell values, using the `where` method and providing a row condition. A more precise manipulation of the single cell values can be archived with `perCol` or `perRowCol` methods, allowing more complex expressions to be written.

Being the underlying structure of `DataColumns` kotlin's collections, all the standard methods (like `map`, `filter`, all `iterable` properties and so on) of them can be called and used when manipulating rows or columns.

More real world examples can be found in the examples section.

CHAPTER

FIVE

DATA WRANGLING

> Data wrangling is the process of transforming and mapping data from one "raw" data form into another format that is more appropriate for analytics and visualization. The goal of data wrangling is to assure quality and usefulness of data.

Data scientists' goal is not just to analyze data, but to gain insights that can drive business decisions. This requires data that is clean, complete, and well-organized. Data wrangling enables us to achieve this by transforming data from a raw or semi-structured form into a more structured format that can be analyzed efficiently.

In this chapter we will explore data wrangling tools such as joining, combining and reshaping dataframes.

5.1 Data Wrangling with Kotlin

In this chapter we will explore some tools and strategies that Kotlin DataFrame offers for all the tasks involved in data wrangling, alongside their python's counterparts.

As always, we will import Kotlin DataFrame with the magic command:

```
%use dataframe
```

Before digging into data wrangling techniques that DataFrame offers, there is one Column type that has not been covered extensively yet: `ColumnGroup`.

5.1.1 `ColumnGroup` and `FrameColumn`

They are a special kind of columns that contains a series of column (in `ColumnGroups`) or a `DataFrame`.

The power of those structures is the ability to store and organize data in a **hierarchical** way. This is essential when dealing with JSON serialization and deserialization.

Dealing with *"nested"* objects can also occur very often when using grouping and pivoting operations (discussed in next chapter), and a minimum comprehension is required before dealing with those operations.

Let's consider a Dataframe of people with the following informations:

```
val name by columnOf(
    "Woody Allen",
    "Bob Dylan",
    "Charlie Chaplin",
    "John Coltrane",
```

(continues on next page)

(continued from previous page)

```
    "Bob Marley",
    "Linus Torvalds",
    "Charlie Parker",
)
val age by columnOf(15, 45, 20, 30, 15, 22, 57)
val city by columnOf(
    "Rome",
    "Moscow",
    "Tirana",
    "Sarajevo",
    "Cesena",
    null,
    "Kyoto",
)

val weight by columnOf(55, 70, null, 80, null, null, 90)
val isDied by columnOf(false, false, true, true, true, false, true)

val people = dataFrameOf(name, age, city, weight, isDied)
people.print()
```

```
              name age     city weight isDied
0      Woody Allen  15     Rome     55  false

1        Bob Dylan  45   Moscow     70  false
2  Charlie Chaplin  20   Tirana   null   true

3    John Coltrane  30 Sarajevo     80   true
4       Bob Marley  15   Cesena   null   true

5   Linus Torvalds  22     null   null  false
6   Charlie Parker  57    Kyoto     90   true
```

Creating a group of columns is pretty straightforward:

```
people.group { age and city }.into("group").print()
```

```
              name                     group weight isDied

0      Woody Allen     { age:15, city:Rome }     55  false

1        Bob Dylan   { age:45, city:Moscow }     70  false

2  Charlie Chaplin   { age:20, city:Tirana }   null   true

3    John Coltrane { age:30, city:Sarajevo }     80   true

4       Bob Marley   { age:15, city:Cesena }   null   true
```

```
5  Linus Torvalds                { age:22 }   null  false
```

```
6  Charlie Parker    { age:57, city:Kyoto }     90   true
```

We can also create a nested column, for example, splitting the name in a `firstName` and a `lastName` column:

```
val groupedDf = people.split { name }.by(' ').inward("firstName", "lastName")
groupedDf.print()
```

	name	age	city	weight	isDied
0	{ firstName:Woody, lastName:Allen }	15	Rome	55	false
1	{ firstName:Bob, lastName:Dylan }	45	Moscow	70	false
2	{ firstName:Charlie, lastName:Chaplin }	20	Tirana	null	true
3	{ firstName:John, lastName:Coltrane }	30	Sarajevo	80	true
4	{ firstName:Bob, lastName:Marley }	15	Cesena	null	true
5	{ firstName:Linus, lastName:Torvalds }	22	null	null	false
6	{ firstName:Charlie, lastName:Parker }	57	Kyoto	90	true

Using the `inward()` method splits the columns into the provided column names, nesting the inside the original column, creating a `ColumnGroup`.

```
groupedDf.name.javaClass
```

```
class org.jetbrains.kotlinx.dataframe.impl.columns.ColumnGroupImpl
```

We can always access the fields of the `ColumnGroup` with the `.` notation

```
groupedDf.name.firstName.print()
```

```
  firstName
0     Woody
1       Bob
2   Charlie
3      John
4       Bob
5     Linus
```

```
6   Charlie
```

As said above, most of the time we will have to deal with these nested structures when using `pivot` or `groupBy` methods. We can, for example, pivot the table to create columns that contains a `DataFrame`: `FrameColumns`

```
groupedDf.pivot{ name.firstName }
```

No outputs are returned because the dataframe is now a `Pivot` object, and it should be a temporary object before applying an aggregate function or other manipulations. We will cover `pivot` and `groupBy` extensively in the *chapter 7*.

These nested structures can resemble to a `pandas.MultiIndex`: they both express the concept of organizing data in a **hierarchical** way.

DataFrame multilevel structures differs from pandas because they do not have an explicit concept of `Index`, and operations like `pandas.dataframe.stack()/unstack()` would make no sense. In some ways that result can be accomplished with some trickery, but DataFrame's `ColumnGroup` or `FrameColumn` are not intended to substitute `pandas.MultiIndex`, even if they're goal is very similar.

5.1.2 Working with Multiple DataFrames

DataFrame provides three methods for operating with multiple `DataFrames`:

- `add`: adds new **columns** to the `DataFrame`.
- `concat`: returns the **union** of the provided `DataFrames`.
- `join`: SQL-like join of two `DataFrames` by **key** columns.

we already have seen an application of the `add` method, but it is possible to add multiple columns all at once:

```
groupedDf
    .convert { weight }.toDouble()
    .dropNA { weight }
    .add {
    "year of birth" from 2023 - age
    age gt 18 into "is adult"
    "details" {
         "weight"<Double>() / 6.35 into "weight (approx. stones)"
        "full name" from { name.firstName + " " + name.lastName }
    }
}.print()
```

```
                                        name age     city weight isDied year of birth␣
↪is adult                                      details

0    { firstName:Woody, lastName:Allen }  15     Rome   55.0  false          2008␣
↪    false { weight (approx. stones):8.661417, f...

1      { firstName:Bob, lastName:Dylan }  45   Moscow   70.0  false          1978␣
↪     true { weight (approx. stones):11.023622, ...

2  { firstName:John, lastName:Coltrane }  30 Sarajevo   80.0   true          1993␣
↪     true { weight (approx. stones):12.598425, ...

3 { firstName:Charlie, lastName:Parker }  57    Kyoto   90.0   true          1966␣
↪     true { weight (approx. stones):14.173228, ...
```

When applying concat, it concatenates the rows of the provided DataFrames or DataColumns.

```
val df1 = dataFrameOf("a", "b", "c").fill(5) { it }
val df2 = dataFrameOf("a", "b", "c").fill(2) { it  - 10 }

df1.concat(df2).print()
```

```
    a   b   c
0   0   0   0
1   1   1   1
2   2   2   2
3   3   3   3
4   4   4   4
```

```
5 -10 -10 -10
6  -9  -9  -9
```

When concatenating dataframes with different column keys, the result is like a *full join* in the database world, where non matching values of the two collections are filled with null.

```
val df1 = dataFrameOf("a", "b", "c").fill(10) { it }
val df2 = dataFrameOf("a", "c", "d").fill(2) { it - 10 }

df1.concat(df2).print()
```

```
     a    b   c    d
 0   0    0   0 null
 1   1    1   1 null
 2   2    2   2 null
```

```
 3   3    3   3 null
 4   4    4   4 null
 5   5    5   5 null
 6   6    6   6 null
```

```
 7   7    7   7 null
 8   8    8   8 null
 9   9    9   9 null
10 -10 null -10  -10
```

```
11  -9 null  -9   -9
```

We can also use the concat method providing a List object

```
listOf(df1, df2).concat().print()
```

```
     a    b   c    d
 0   0    0   0 null
```

(continues on next page)

(continued from previous page)

```
 1   1    1   1 null
 2   2    2   2 null

 3   3    3   3 null
 4   4    4   4 null
 5   5    5   5 null
 6   6    6   6 null

 7   7    7   7 null
 8   8    8   8 null
 9   9    9   9 null
10 -10 null -10  -10

11  -9 null  -9   -9
```

The `concat` method is similar to `pandas.concat` method, with the difference that in pandas you can specify which *axis* to merge, having the `Index` object that can provide a merging key when choosing `axis=0`. On the other hand, When using `axis=1`, merging two pandas `DataFrames` will produce a similar result to what Kotlin `DataFrame` provides.

If we want to use sql like `join` operations, we can use the `join` method provided by Kotlin `DataFrame`.

`join`'s method signature is the following:

```
join(otherDf, type = JoinType.Inner) [ { joinColumns } ]
```

Having the join columns as optional, and the default join is set to Inner (only matched columns from left and right `DataFrames`).

```
val df1 = dataFrameOf("a", "b", "c").fill(10) { it }
val df2 = dataFrameOf("a", "c", "d").fill(2) { it }
```

```
df1.join(df2).print()
```

```
  a b c d
0 0 0 0 0
1 1 1 1 1
```

We can specify which column to match with the `match` DSL keyword

```
df1.join(df2, type = JoinType.Full) { a match right.a }.print()
```

```
  a b c   c1    d
0 0 0 0    0    0
1 1 1 1    1    1
2 2 2 2 null null
3 3 3 3 null null

4 4 4 4 null null
5 5 5 5 null null
```

(continues on next page)

(continued from previous page)

```
6  6  6  6 null null
7  7  7  7 null null
8  8  8  8 null null
```

```
9  9  9  9 null null
```

And you can sport that any column that matched during the join, but not included in the `joinColumn` clause, are duplicated with a new column key.

The `match` keyword is used in all those cases where we can apply the join because of matching row values, but the columns keys differs by name.

Consider the next example:

```
people.print()
```

```
                name age     city weight isDied
0      Woody Allen  15     Rome      55   false
```

```
1        Bob Dylan  45   Moscow      70   false
2  Charlie Chaplin  20   Tirana    null    true
```

```
3    John Coltrane  30 Sarajevo      80    true
4       Bob Marley  15   Cesena    null    true
```

```
5   Linus Torvalds  22     null    null   false
6   Charlie Parker  57    Kyoto      90    true
```

and let's suppose we have a new dataset with new data that can be joined with the previous one

```
val newPeopleDf = people.head(2) // pick just the first two
        .rename("name").into("fullName") // renameing join column
        .add("stonesWeight") { // add new dummy column
            weight!! / 6.35
        }
        .select("fullName", "stonesWeight")

newPeopleDf.print()
```

```
     fullName stonesWeight
0 Woody Allen     8.661417
1   Bob Dylan    11.023622
```

We can use the `match` keyword for specifying which columns to use for the join operation:

```
people.join(newPeopleDf, type = JoinType.Left) { name match right.fullName }.print()
```

```
                name age     city weight isDied stonesWeight
```

```
0     Woody Allen  15     Rome     55  false     8.661417
1       Bob Dylan  45   Moscow     70  false    11.023622
2 Charlie Chaplin  20   Tirana   null   true         null
3   John Coltrane  30 Sarajevo     80   true         null
4      Bob Marley  15   Cesena   null   true         null
5  Linus Torvalds  22     null   null  false         null
6  Charlie Parker  57    Kyoto     90   true         null
```

There are handy shortcuts for specifying which type of join we want to perform for each kind of join. The previous code can be rewritten to:

```
people.leftJoin(newPeopleDf) { name match right.fullName }.print()
```

```
             name age     city weight isDied stonesWeight
0     Woody Allen  15     Rome     55  false     8.661417
1       Bob Dylan  45   Moscow     70  false    11.023622
2 Charlie Chaplin  20   Tirana   null   true         null
3   John Coltrane  30 Sarajevo     80   true         null
4      Bob Marley  15   Cesena   null   true         null
5  Linus Torvalds  22     null   null  false         null
6  Charlie Parker  57    Kyoto     90   true         null
```

See the full reference for supported types of join.

5.1.3 Reshaping and Pivoting

Reshaping and Pivoting a datasets is a very common operation that is being made during Data Analysis, and Kotlin DataFrame provides a series of methods that can help the developer in the creation of different views of the same `DataFrame`.

The most common operations that are used when pivoting and reshaping a dataset, are `pivot` and `groupBy`, and very often they're used chained together for distributing data along rows or columns.

We will run all the examples with the `macrodata` dataset

```
val df = DataFrame.readCSV("../resources/example-datasets/datasets/macrodata.csv")
df.print(3)
```

```
  year quarter  realgdp realcons realinv realgovt realdpi   cpi    m1 tbilrate␣
↪unemp     pop infl realint
```

```
0 1959       1 2710.349   1707.4 286.898  470.045  1886.9 28.98 139.7     2.82  ␣
↪5.8 177.146 0.00    0.00
```

```
1 1959       2 2778.801   1733.7 310.859  481.301  1919.7 29.15 141.7     3.08  ␣
↪5.1 177.830 2.34    0.74
```

```
2 1959       3 2775.488   1751.8 289.226  491.260  1916.4 29.35 140.5     3.82  ␣
↪5.3 178.657 2.74    1.09
```

```
...
```

Consider the following example:

```
val longFormat = df.groupBy { year and quarter }
    .values { realgdp and infl and unemp }
    .gather { realgdp and infl and unemp }.into("item", "value")
    .explode("value")

longFormat.print(10)
```

```
  year quarter    item    value
0 1959       1 realgdp 2710.349
1 1959       1    infl    0.000
```

```
2 1959       1   unemp    5.800
3 1959       2 realgdp 2778.801
4 1959       2    infl    2.340
```

```
5 1959       2   unemp    5.100
6 1959       3 realgdp 2775.488
7 1959       3    infl    2.740
```

```
 8 1959       3   unemp    5.300
 9 1959       4 realgdp 2785.204
...
```

In Kotlin DataFrame, `groupBy` operation is not only used for grouping and aggregating data, but it can be very useful for rearranging data. `groupBy` takes a list of columns to group by, and produces a `DataFrame` where each group key is placed in a distinct row, with its associated group (a `FrameColumn`).

If we run the code above row by row, we can see how the result `DataFrame` has been formed:

```
df.head(4) // using head just to limit the output
    .groupBy { year and quarter }.print()
```

```
  year quarter                                    group

 0 1959       1 [1 x 14] { year:1959, quarter:1, real...

 1 1959       2 [1 x 14] { year:1959, quarter:2, real...

 2 1959       3 [1 x 14] { year:1959, quarter:3, real...

 3 1959       4 [1 x 14] { year:1959, quarter:4, real...
```

On the other hand, if we call `pivot`, all the provided columns will be the column keys of the resulting `DataFrame`, creating another group of columns.

So for example, if we want to pick only all the data from 1995 to 2000, compute the mean of `unemp` variable, and display one column for each year, we can use the `pivot` method.

```
df.filter { year >= 1995 && year <= 2000 }.pivot { year }.mean { unemp }.print()
```

```
{ 1995:5.625000, 1996:5.400000, 1997:4.950000, 1998:4.475000, 1999:4.225000,
 2000:3.950000 }
```

After a `pivot` or `groupBy` operation, we can use the method `values` for selecting only some columns of the group.

```
df.head(4)
    .groupBy { year and quarter }
    .values { realgdp and infl and unemp }.print()
```

```
  year quarter    realgdp   infl unemp
 0 1959       1 [2710.349]  [0.0] [5.8]

 1 1959       2 [2778.801] [2.34] [5.1]
 2 1959       3 [2775.488] [2.74] [5.3]

 3 1959       4 [2785.204] [0.27] [5.6]
```

Now, if we want to display data in the so called *long format*, with each row containing the year, quarter, item name and value, we have to make item's columns to be mapped as rows.

With the help of the `gather` we can map a set of columns to two columns: "key" containing **names** of the original columns and "value" containing **values** of the original columns.

In a certain way, `gather` is the opposite of `pivot`, that splits rows of a dataframe and groups them horizontally into new columns.

If we apply `gather`, the result will be:

```
df.head(4)
    .groupBy { year and quarter }
    .values { realgdp and infl and unemp }
    .gather { realgdp and infl and unemp }
    .into("item", "value").print()
```

```
   year quarter    item      value
  0 1959       1 realgdp [2710.349]
```

```
  1 1959       1    infl      [0.0]
  2 1959       1   unemp      [5.8]
```

```
  3 1959       2 realgdp [2778.801]
  4 1959       2    infl     [2.34]
```

```
  5 1959       2   unemp      [5.1]
  6 1959       3 realgdp [2775.488]
```

```
  7 1959       3    infl     [2.74]
  8 1959       3   unemp      [5.3]
```

```
  9 1959       4 realgdp [2785.204]
 10 1959       4    infl     [0.27]
```

```
 11 1959       4   unemp      [5.6]
```

Lastly, the square brackets suggest us that the `value` column contains a series of list, and we can flatten it with the `explode` method.

In contrast to Kotlin DataFrame, pandas has more "ad-hoc" methods for both pivoting and reshaping datasets. Having the `Index` object, every `DataFrame` has the ability to easily `reindex` itself, or using the `pivot` operation to swap the order of both index and columns, specifying which will be the columns of values. For example, in pandas the pipeline we created above could be translated as follows:

```
df.pivot_table(index=['year', 'quarter'], values=['realgdp', 'infl', 'unemp']) \
    .stack() \
    .reset_index() \
    .rename(columns={ "level_2": "item", 0: "value"})
```

Where the `pivot_table` method is used to rearrange `Index` objects and value columns (it's a more generalized version of `pivot` method). In pandas, `stack` and `unstack` operations are very useful when collapsing several columns in one (like DataFrame's `gather`), or when distributing data contained in one column across several (like DataFrame's `pivot`). These operations, combined with the indices manipulations techniques (like `reset/set_index`, `reindex` (works also with columns)), makes pandas more powerful and more precise when it comes to data wrangling. Moreover, pandas `Index` object can be of multiple types, for example `CategoricalIndex`, `DatetimeIndex`, `PeriodIndex`, `MultiIndex`. In this example, a `PeriodIndex` would have fit the data perfectly, because we can create a range of dates from a year and a quarter: `pd.PeriodIndex(year=df['year'], quarter=df['quarter'], name='date')`.

We now understand that the biggest difference between pandas and Kotlin DataFrame is the presence of an explicit Index Object that let us perform reshaping of the dataframe in a more precise way.

Note that the operation after `stack` are used only to recreate the example, the `Series` created with `stack` is perfectly usable as is.

5.1.4 Conclusions

In the chapter we have explored how Kotlin DataFrame has a full support for data wrangling tasks. The differences with pandas are sometimes remarkable, and our way to compute some operations can be very different between the two platforms. Anyhow, with a little bit of practicing, a lot of what can be done in pandas can be archived with the use of Kotlin DataFrame.

CHAPTER

SIX

DATA VISUALIZATION

During the data analysis process, data visualization is one of the most important tasks. Making an effective informative visualization, may be useful in the exploratory process, but it can also be the end goal!

Python's most popular **plotting** library is `Matplotlib`, and it will be compared with Kotlin's `Lets-Plot` library, which is largely based on the API that `ggplot2` (*R* open source plotting library) offers.

The approach of ggplot and Lets-Plot is to use a series of **layers** for building the figure, whereas Matplotlib uses an object oriented approach, where all the customization is made on the figure itself.

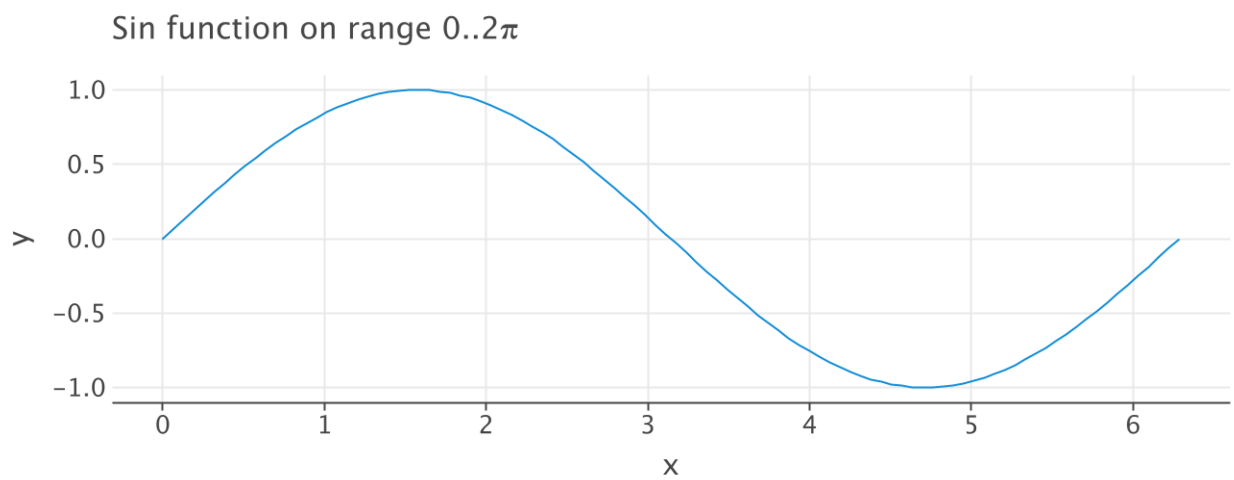
6.1 Visualizing Data in Kotlin

As DataFrame and Multik, **Lets-Plot** library can be imported with the magic command:

```
%use multik
%use dataframe
%use lets-plot
```

For creating a `Plot` object, we invoke the function `ggplot()`, which accepts a `Map<*, *>` and a generic aesthetic mapping.

```
val x = mk.linspace<Double>(0.0, 2.0 * kotlin.math.PI, 100)
val y = mk.math.sin(x)

val points = mapOf<String, Any>(
    "x" to x.toList(),
    "y" to y.toList()
)

val p = ggplot(points) { x= "x" ; y = "y" } +
    geomLine() +
    ggtitle("Sin function on range 0..2π") +
    ggsize(650, 250)
p
```



In Lets-Plot, the main difference with python's Matplotlib, is the creation of the plot by **layers**. In Kotlin, thanks to the possibility to overload the + operator, we create the figure with a chain of additions on top of the `Plot` object. In the example below, we created

- The plot providing a series of points, and the mapping of map's keys to plot axes.
- The layer of the line.
- A layer for the title
- A layer for configuring the dimensions of the figure.

This approach seems a little bit more expensive than with python:

```
x = np.linspace(0, 2 * np.pi, 100)
plt.plot(x, np.sin(x))
```

but the benefits of layers can be seen in more complicated examples.

Note: Matplotlib encourages the use of the "Object Oriented" APIs, so the following code would be preferred:

```
x = np.linspace(0, 2 * np.pi, 100)

fig, ax = plt.subplots()
ax.plot(x, np.sin(x))
plt.show()
```

See matplotlib blog for more information.

```

```

6.1.1 Lets-Plot Architecture

As already said, a plot is composed by one or more *Layers*. Each layer is responsible for creating the objects painted on the "canvas" and each one contains:

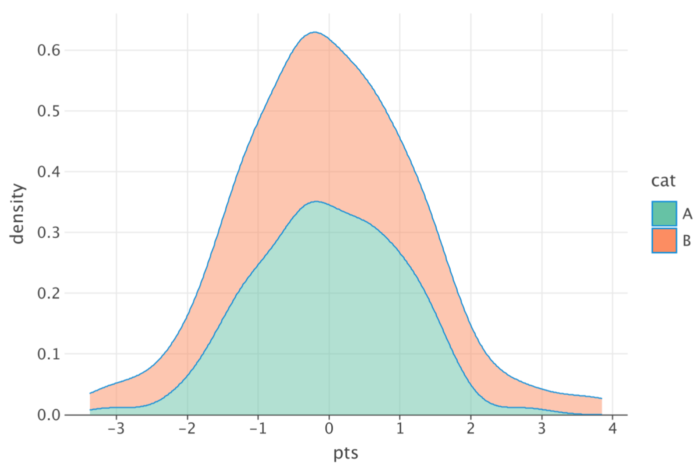
- **Data**: the set of data specified for all layers, or one dataset per layer.
- **Aesthetic Mapping**: describe how variables in the dataset are mapped to the visual properties of the layer.
- **Geometric Object**: a geometric object that represents a particular type of chart.
- **Statistical Transformation**: computes a statistical summary on the raw input data.
- **Positional Adjustment**: method used to compute the final coordinates of geometry.

Geometric Objects

They are responsible for drawing in the plot. All the functions that are of the type `geomXxx()` create a new layer that draws the data. Every `geom` object has its own default parameters and behavior, see the documentation for understanding what the desired plot does or require.

The `geom` package contains some `statXxx()` methods which also create a plot layer: sometimes is more natural to use `statXxx()` objects instead of `geomXxx()` to add a new plot layer.

```
val rand = java.util.Random(123)
val dataset = mapOf(
    "pts" to List(100) { rand.nextGaussian() } + List(100) { 1.5 * rand.
↪nextGaussian() },
    "cat" to List(100) { "A" } + List(100) { "B" }
)
val p = ggplot(dataset)

p + statDensity(alpha = 0.5) { x="pts" ; fill="cat" }
```



stat

`stat` can be added as an argument to a geometric object to define statistical transformation. The `Stat` object contains all the statistical transformations that can be applied to a dataset, and it can be used like `geomXxx(stat = Stat.identity)`.

We can apply a statistical transformation like `bin`, `density`, `count`, `smooth` and more.

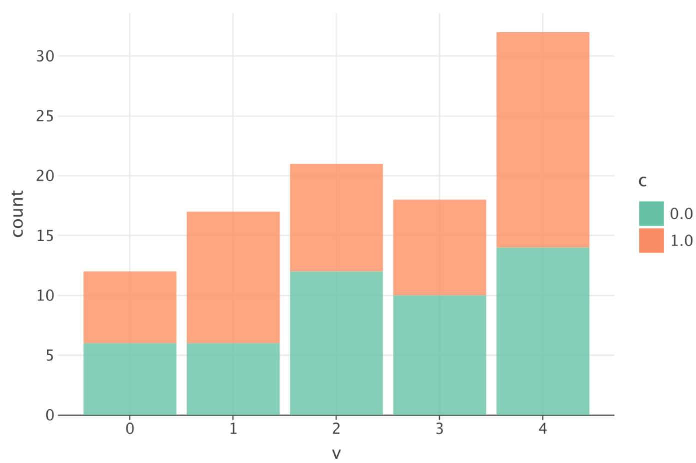
Position

It's possible to adjust the position of data, especially in all those cases where data overlaps.

We also introduce `ggbunch` that let us draw multiple plots in the same figure.

Consider this dataset and it's corresponding bar plot:

```
val data = mapOf(
    "v" to List(100) { rand.nextInt(5) },
    "c" to List(100) { rand.nextInt(2) }
)

val p0 = ggplot(data) +
    geomBar(alpha = 0.8) { x = "v"; fill=asDiscrete("c") }
p0
```



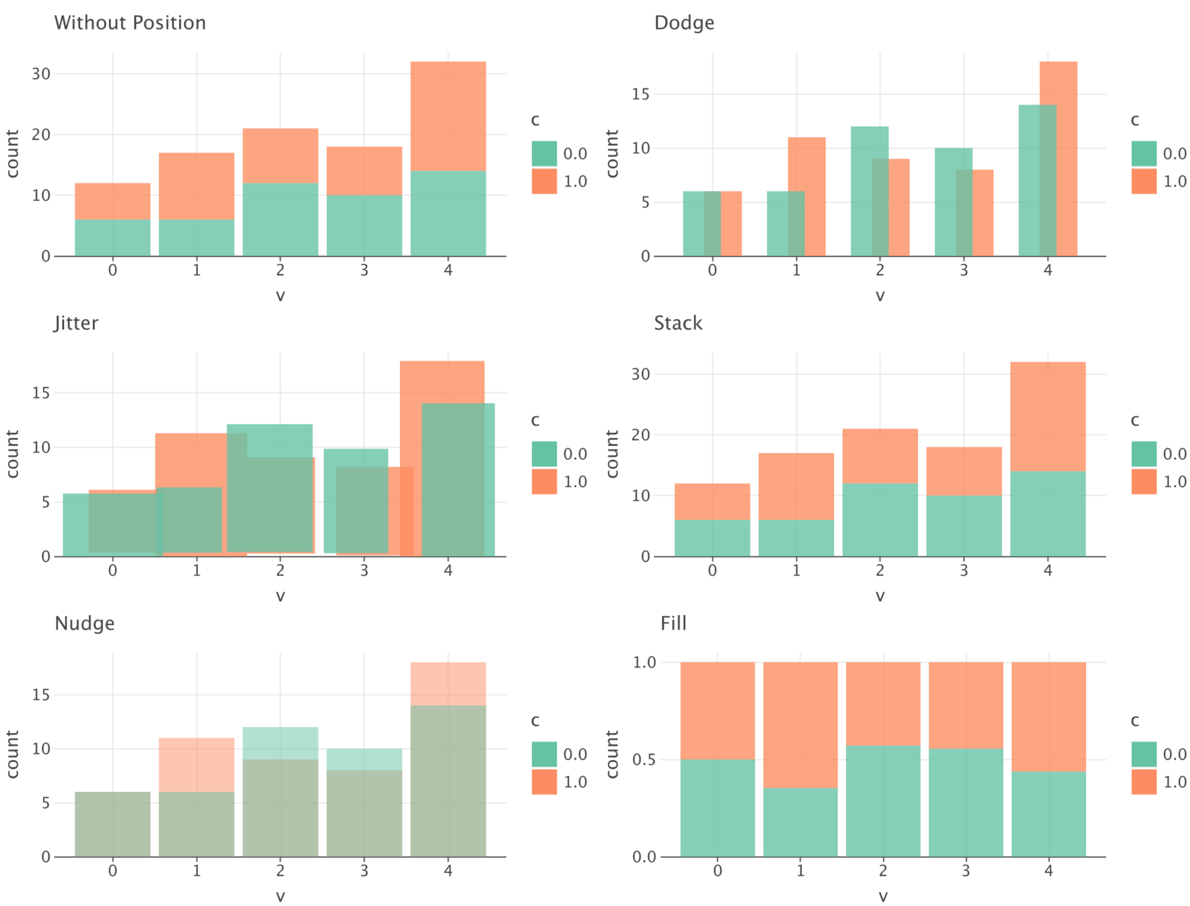
We can now set the position of the data to better visualize data:

```
val p1 = ggplot(data) +
    geomBar(alpha = 0.8, position = positionDodge(0.5)) { x = "v"; fill = asDiscrete(
↪"c") }

val p2 = ggplot(data) +
    geomBar(alpha = 0.8, position = positionJitter(0.2) ) { x = "v"; fill =␣
↪asDiscrete("c") }

val p3 = ggplot(data) +
    geomBar(alpha = 0.8, position = positionStack() ) { x = "v"; fill = asDiscrete("c
↪") }
```

(continues on next page)

(continued from previous page)

```
val p4 = ggplot(data) +
    geomBar(alpha = 0.5, position = positionNudge() ) { x = "v"; fill = asDiscrete("c
↪") }

val p5 = ggplot(data) +
    geomBar(alpha = 0.8, position = positionFill() ) { x = "v"; fill = asDiscrete("c
↪") }

val allplt = GGBunch()
    .addPlot(p0 + ggtitle("Without Position"), 0, 0, 500, 250)
    .addPlot(p1 + ggtitle("Dodge"), 500, 0, 500, 250)
    .addPlot(p2 + ggtitle("Jitter"), 0, 250, 500, 250)
    .addPlot(p3 + ggtitle("Stack"), 500, 250, 500, 250)
    .addPlot(p4 + ggtitle("Nudge"), 0, 500, 500, 250)
    .addPlot(p5 + ggtitle("Fill"), 500, 500, 500, 250)
allplt
```



Features

The entire plot can be provided with additional *features layers*. The features can be grouped in the following categories:;

- **Scale**: enables choosing a `scale` for each mapped variable, depending on its attributes. With scales, we can tweak things like, the axis labels, legends keys, aesthetics (like the fill color) and so on.
- **Coordinate System**: determine how x and y aesthetics combine, to position elements in the plot. (i.e. for overriding default axes ratio we can use `coordFixed(ratio = 2)`).
- **Legend**: we can customize the legend (i.e. the number of columns) by using the `guide` methods, or the `guide` argument inside a `scale` method. The location of the legend can be tweaked with `theme`'s methods.
- **Sampling**: we can pick samples of the dataset (sampling is applied *after* `stat` transformations), and if the dataset exceeds a certain threshold, sampling is applied automatically (the `samplingNone` value disables any sampling for the given layer). See the `sampling` documentation for more.

6.1.2 Integration with Kotlin DataFrame

As you might have already seen, `DataFrame` objects has the `toMap()` method, making plotting a dataframe a trivial task. Let's see an example on how we can integrate all the libraries that we have seen all together for computing and showing the log difference of two variables.

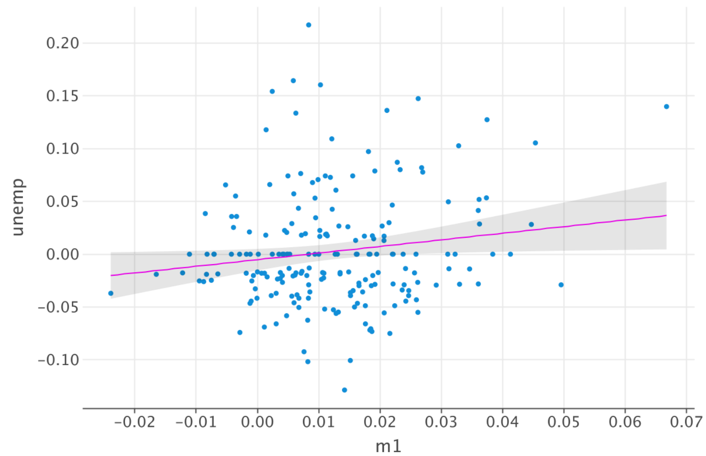
```
val df = DataFrame.readCSV("../resources/example-datasets/datasets/macrodata.csv")[
↪"cpi", "m1", "tbilrate", "unemp"]
df.print(5)
```

```
     cpi    m1 tbilrate unemp
 0 28.98 139.7     2.82   5.8
 1 29.15 141.7     3.08   5.1
```

```
 2 29.35 140.5     3.82   5.3
 3 29.37 140.0     4.33   5.6
 4 29.54 139.6     3.50   5.2
...
```

We select `m1` and `unemp` variables and make a scatter plot with a regression line (`geomSmooth()`)

```
// python equivalent: `np.log(df).diff().dropna()`
val trans_data =  df.columns()
    .map {
        val data = it.toList()
        mk.dnarray<Double, D1>(intArrayOf(data.size)) { data[it] as Double }
    }.map { mk.math.log(it).toList() }
    .mapIndexed { idx, x ->
        x.zipWithNext { a, b -> b - a }
            .toColumn(df.columnNames()[idx])
    }.toDataFrame().dropNA()

ggplot(trans_data["m1", "unemp"].toMap()) { x="m1"; y="unemp" } +
    geomPoint() + geomSmooth()
```



The same result with Matplotlib would not be as easy as in Kotlin, unless *Seaborn* would be used.

6.1.3 Conclusions

In this chapter we have explored Lets-Plot library, a very powerful tool for visualizing data. This library and ggplot's APIs are used in a lot of different programming languages, so a solid knowledge of this library can be portable on other platforms.

Its behavior is quite different from Python's Matplotlib, but if you know what you can get with Matplotlib, you can easily find a solution using Lets-Plot!

This chapter wasn't intended to cover all capabilities of this library, but a basic understanding on how to build plots stacking together a series of layers, the ability to customize every aspect of it, and how easy it is to plot data from a `DataFrame`. The documentation provided from the Lets-Plot team is very good, with plenty of examples for every geometric objects, kind of plots, scales and much more.

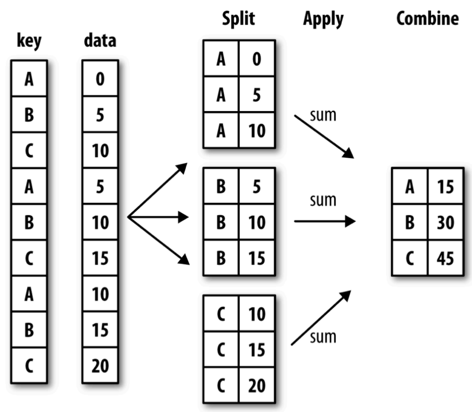
CHAPTER

SEVEN

DATA AGGREGATION AND GROUP OPERATIONS

Another important tool for data analysis, is the ability to group and categorize data, for computing some statistics with aggregate operations to each group, or maybe pivoting a table for statistical visualization purposes.

Both pandas and Kotlin DataFrame follow the **split-apply-combine** strategy ([Wic11]).



This strategy involves breaking down a dataset into smaller subsets, performing a specific analysis or transformation on each subset, and then combining the result into a single output.

In the following section we will see in depth how Kotlin DataFrame compute grouping operations, data aggregation and table pivoting in comparison with pandas.

7.1 Data Grouping and Aggregation with Kotlin

```
%use dataframe
```

Kotlin DataFrame provides a very similar interface to the one supplied by pandas. The methods that we will use more will be `groupBy()` and `pivot()`, that we have already discussed in a previous chapter.

Having no such thing as `pandas.Index`, with Kotlin DataFrame the structure of the tabular data is more flexible, but in the same time it lacks all the power that a *labeled* index or multi index could give. The tabular object represented by a Kotlin `DataFrame` is slightly different from the pandas one, and so we have to think differently sometimes when it comes to replicate some pandas behaviors, especially when pivoting and grouping by a variable.

7.1.1 Split

The "Split" operation in both pandas and DataFrame, is performed when invoking the `groupBy` or `pivot` operation. The result of those methods are not a dataset, but a temporary data structure that needs an aggregate operation to combine all the groups that are been created.

Let's consider the following dataset:

```
val rand = java.util.Random(100)
val letters = sequenceOf("A", "B", "C")

val df = mapOf(
    "letter" to List(10) { letters.shuffled().find { true } },
    "num" to List(10) { rand.nextGaussian() }
).toDataFrame()

df.print()
```

```
  letter       num
0      B  0.624629
1      A -0.858192
2      C  0.676221
3      B  1.126394
```

```
4      B  1.437662
5      C  1.492079
6      A -1.598537
7      B -0.077561
8      C  1.991766
```

```
9      C  0.270969
```

If we call a `groupBy` with the `letter` column, we will *split* the dataset into three groups, one for each letter. In Kotlin DataFrame, we can print easily the content of the grouping operation, whereas in python we have to loop through the content of each group, because calling `groupby` will not compute anything except some intermediate data about the group key.

```
df.groupBy { letter }.print()
```

```
  letter   group
0      B [4 x 2]
1      A [2 x 2]
2      C [4 x 2]
```

And we can see all the groups that are being formed. The `groupBy` method accept any column expression provided, so the following will be still legit:

```
df.groupBy { letter.map { it == "A"} }.print()
```

```
  letter   group
0  false [8 x 2]
1   true [2 x 2]
```

This is very useful when manipulating `date` objects and we want to group by year/month/day.

```
val dates = listOf(
    LocalDate(1998, 1, 2),
    LocalDate(1999, 11, 21),
    LocalDate(1999, 3, 12),
    LocalDate(1998, 6, 22),
    LocalDate(1998, 12, 25),
    LocalDate(1999, 12, 24),
    LocalDate(1999, 2, 9),
    LocalDate(1999, 6, 24),
    LocalDate(1998, 1, 20),
    LocalDate(1999, 12, 20),
).toColumn("date")

val dfd = df.add(dates)
```

```
dfd.groupBy { date.map { it.year } }.print()
```

```
  date   group
0 1998 [4 x 3]
1 1999 [6 x 3]
```

`groupBy` accepts any number of column names to group by with.

```
dfd.groupBy { letter and date.map { it.year } }.print()
```

```
  letter date                                        group

0      B 1998                                      [3 x 3]

1      A 1999                                      [2 x 3]

2      C 1999                                      [3 x 3]

3      B 1999 [1 x 3] { letter:B, num:-0.077561, da...

4      C 1998 [1 x 3] { letter:C, num:1.991766, dat...
```

We can also override the name of the group key column in the resulting dataframe using the `named` keyword.

Before computing aggregation or reduction operations, we can apply some **transformation** to the groups, like sorting, or computing other groups modifications operations. Among those methods, there's the `concat` method, that unions all data groups of `groupBy` into original `DataFrame` preserving new order of rows produced by grouping.

A very common pattern, is to use the concatenation of `groupBy` and `pivot` operations:

```
dfd.groupBy { date.map { it.year } named "year" }.pivot { letter }.values().print()
```

```
  year                               letter
0 1998 { B:[0.6246292191371761, 1.1263938263...
```

```
1 1999 { B:[-0.07756122563035872], C:[0.6762...
```

The result we have computed are not directly usable: as said before, the split-apply-combine strategy require that grouped data are aggregated and reduced with an application of a combining operation.

7.1.2 Aggregation (Apply and Combine)

With grouped data, we can compute a large variety of operations on groups. The most common operations are defined in Kotlin DataFrame library, like `mean()`, `sum()` or `count()`.

For example, if we take the dataframe we grouped above, we can compute the mean of the values inside each group:

```
val groupedDf = dfd.groupBy { date.map { it.year } named "year" }
    .pivot { letter }

groupedDf.mean().print()
```

```
  year                                letter
0 1998               { B:1.062895, C:1.991766 }
```

```
1 1999 { B:-0.077561, C:0.813090, A:-1.228365 }
```

The same exact result can be archived in python with a very similar instruction:

```
dfd.groupby([df['date'].dt.year, 'letter'])\
    .mean() \
    .unstack('letter')
```

Note that the `unstack` operation is used to move "letters" from the `MultiIndex` object created with "year", to columns. It is also possible to call `pivot_table` instead of `unstack`, specifying what will be in place of indices, columns keys and values.

Sometimes is useful to flatten the group result, in favor of a better handling of indices. We can use `flatten()`, which affects *every* groups of the dataframe, or `ungroup()` for selecting only one group.

```
groupedDf.mean().ungroup("letter").print()
```

```
  year         B        C          A
0 1998  1.062895 1.991766      null
```

```
1 1999 -0.077561 0.813090 -1.228365
```

Every time that we create a `GroupBy` or `Pivot` object, and after applying the needed transformations, we can aggregate or reduce all data with several functions all computed in the same time, inside the `aggregate` function. Each operation can then be mapped in the corresponding column.

If we want to compute the sum, the mean and the number of rows of each group, we can write:

```
groupedDf.aggregate {
        sum { num } into "sum"
        mean { num } into "mean"
```

(continues on next page)

(continued from previous page)

```
        count() into "count"
    }.ungroup("letter").print()
```

```
    year                                        B                                       ␣
↪       C                                        A
```

```
  0 1998 { sum:3.188685, mean:1.062895, count:3 } { sum:1.991766, mean:1.991766,␣
↪count:1 }                                        { }
```

```
  1 1999 { sum:-0.077561, mean:-0.077561, coun... { sum:2.439269, mean:0.813090,␣
↪count:3 } { sum:-2.456729, mean:-1.228365, coun...
```

And this approach can be more straightforward than pandas use of `agg` function. If we consider the example above, a similar behavior can be computed in python as follows:

```
df.groupby([df['date'].dt.year, 'letter']) \
    .agg([('sum', 'sum'), ('mean', 'mean'), ('count', 'count')]) \
    .unstack('letter')
```

Where inside `agg` function, we specify the name of the column and the function we want to apply.

Both in python and Kotin, inside the aggregation function a custom function can be called. For example let's consider this function:

```
fun peakToPeak(arr: List<Double>): Double = arr.max() - arr.min()
```

We can the call the function inside the aggregation method:

```
groupedDf.aggregate { peakToPeak(num.toList()) }.print()
```

```
    year                                letter
  0 1998              { B:0.813033, C:0.000000 }
```

```
  1 1999 { B:0.000000, C:1.221110, A:0.740346 }
```

And the same thing can be done with pandas `agg` easily with:

```
def peak_to_peak(arr):
    return arr.max() - arr.min()

groupedDf.agg(peak_to_peak)
```

Kotlin DataFrame is very similar in behavior with pandas, and the transition between the two platform is very natural because of the same strategies implementations.

apply

In python, the `agg` function performs row-wise operations, whereas the `apply` function is a more general purpose. In Kotlin we are more limited with the use of `aggregate`, but with some turnarounds, we can stille get the same results, effortlessly.

Consider this python function:

```
def top(df, n=5, column):
    return df.sort_values(by=column)[-n:]
```

That returns the top `n` rows with the largest value in `column`.

With `apply`, we can call this operation to every group:

```
df.groupby(df['year'].dt.year, 'letter').apply(top, n=1, 'num')
```

and the "dataframe" operation is computed along each groups.

The same behavior can be accomplished in kotlin with:

```
fun top(df: AnyFrame, col: String, n: Int = 5): AnyFrame = df.sortByDesc(col).head(n)
```

```
groupedDf.aggregate { top(it, col="num", n=5) }.print()
```

```
  year                                    letter
0 1998 { B:[3 x 3], C:[1 x 3] { letter:C, nu...
```

```
1 1999 { B:[1 x 3] { letter:B, num:-0.077561...
```

Where the function invocation is a little more explicit than the panda's one.

In contrast to Kotlin DataFrame, pandas `groupby` and `apply` function are way more powerful when grouping and applying functions when slicing data up into buckets with bins or by sample quantiles (using the function `cut` and `qcut`). Kotlin DataFrame does not provide a way to handle `Categorical` object easily as in pandas, but some turnarounds can be made in order to emulate the behavior of pandas.

7.1.3 Pivot Tables and Cross Tabulation

The pandas method `pivot_table` summarizes the joint use of `groupBy` and `pivot` in Kotlin Dataframe:

- The list of values specified in the pivot table, are the ones used for compute an aggregate function in Kotlin DataFrame's `aggregate` function body.
- The `index` list provided to `pivot_table`, is represented by the list of column names inside the Kotlin DataFrame `groupBy` method.
- The `columns` list is represented by the list of column names inside the Kotlin DataFrame's `pivot` function.
- The `aggfunc` parameter in Kotlin is specified inside the `aggregate` function body.

Let's consider some examples with the `tips` dataset. (More about this dataset in the examples at the end of this document)

```
val tips = DataFrame.readCSV("../resources/example-datasets/datasets/tips.csv")
tips.print(5)
```

```
   total_bill  tip    sex smoker day   time size
 0      16.99 1.01 Female  false Sun Dinner    2
```

```
 1      10.34 1.66   Male  false Sun Dinner    3
 2      21.01 3.50   Male  false Sun Dinner    3
```

```
 3      23.68 3.31   Male  false Sun Dinner    2
 4      24.59 3.61 Female  false Sun Dinner    4
```

```
...
```

```
tips.groupBy { day and smoker }.sortBy { day and smoker }.mean().print()
```

```
    day smoker total_bill      tip     size
 0  Fri  false  18.420000 2.812500 2.250000
```

```
 1  Fri   true  16.813333 2.714000 2.066667
 2  Sat  false  19.661778 3.102889 2.555556
```

```
 3  Sat   true  21.276667 2.875476 2.476190
 4  Sun  false  20.506667 3.167895 2.929825
```

```
 5  Sun   true  24.120000 3.516842 2.578947
 6 Thur  false  17.113111 2.673778 2.488889
```

```
 7 Thur   true  19.190588 3.030000 2.352941
```

In python we could have accomplished the same result with the use of pivot table as follows:

```
tips.pivot_table(index=['day', 'smoker'])
```

and by default, the aggregation function used is the mean of the data.

Note that in the following examples we are using the `sortBy` method for sticking to the output that pandas would provide.

We could now create a more sophisticated pivot table with this python snippet:

```
tips['tips_pct'] = tips['tip'] / tips['total_bill']
tips.pivot_table(['tips_pct', 'size'], index=['time', 'day'],
                 columns='smoker')
```

And in Kotlin that pivot table can be created with:

```
tips.add("tips_pct") { tip / total_bill }
    .groupBy { time and day }.sortBy { time and day }
    .pivot { smoker }
    .aggregate {
        mean("tips_pct") into "tips_pct"
        mean { size } into "size"
    }.print()
```

```
     time  day                                   smoker

 0 Dinner  Fri { true:{ tips_pct:0.165347, size:2.22...

 1 Dinner  Sat { true:{ tips_pct:0.147906, size:2.47...

 2 Dinner  Sun { true:{ tips_pct:0.187250, size:2.57...

 3 Dinner Thur { false:{ tips_pct:0.159744, size:2.0...

 4  Lunch  Fri { true:{ tips_pct:0.188937, size:1.83...

 5  Lunch Thur { true:{ tips_pct:0.163863, size:2.35...
```

So a deep understanding of the pandas `pivot_table` method, can really helps a lot when it comes to learning how to pivot and rearrange the dataset using Kotlin DataFrame!

7.1.4 Conclusion

This process of data aggregation and the application of the split-apply-combine strategy can help both data cleaning as well as modeling or statistical analysis work, and as we have seen in this chapter, Kotlin DataFrame offers almost the same pandas capabilities when it come to those operations, especially when pivoting tables.

CHAPTER

EIGHT

DATA ANALYSIS EXAMPLES

In the following chapter, we will go through the different behaviors of Kotlin and Python when it's time to make some data analysis. (References to the examples, and little explanation of what we want to look for (goals of statistical/exploratory analysis))

- *Kotlin: Tips Dataset*
- *NBA Data Analysis*

8.1 Kotlin: Tips Dataset

The *tips* dataset is a data frame with 244 rows and 7 variables, which represents some tipping data where one waiter recorded information about each tip he received over a period of few months working in one restaurant.

(short intro about the dataset)

For this example, we will import the following packages

```
%use multik
%use dataframe
%use lets-plot
```

Let's load the "Tips" dataset, and show it's first 5 rows:

```
val tips = DataFrame.readCSV("../resources/example-datasets/datasets/tips.csv")
tips.print(5)
```

```
   total_bill  tip    sex smoker day   time size
 0      16.99 1.01 Female  false Sun Dinner    2
```

```
 1      10.34 1.66   Male  false Sun Dinner    3
 2      21.01 3.50   Male  false Sun Dinner    3
```

```
 3
```

```
 23.68 3.31   Male  false Sun Dinner    2
 4      24.59 3.61 Female  false Sun Dinner    4
...
```

The dataset has 7 variables:

- `total_bill` in dollars
- `tip` in dollars
- `sex` of the bill payer
- `smokers` whether there were smokers in the party
- `day` of the week
- `time` time of day
- `size`: people at the party

During the loading of the dataset, some values could have been mapped to a wrong datatype (e.g. `Date` can be loaded as `String` if not well formatted).

With the `schema()` method it's possible to see how values have been parsed.

```
tips.schema()
```

```
total_bill: Double
tip: Double
sex: String
smoker: Boolean
day: String
time: String
size: Int
```

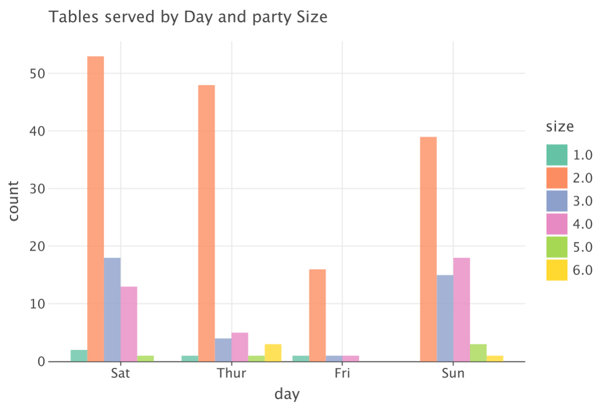
We can analyze some statistics of categorical data (`String` and `Boolean` columns):

```
tips.describe { colsOf<String>() and colsOf<Boolean>() }.print()
```

	name	type	count	unique	nulls	top	freq	min	median	max
0	sex	String	244	2	0	Male	157	Female	Male	Male
1	day	String	244	4	0	Sat	87	Fri	Sun	Thur
2	time	String	244	2	0	Dinner	176	Dinner	Dinner	Lunch
3	smoker	Boolean	244	2	0	false	151	false	false	true

There are four categorical variables in the Tips dataset as seen above. For a better visualization of those data, we can make plots for visualizing for example the number of people for each day of the week.

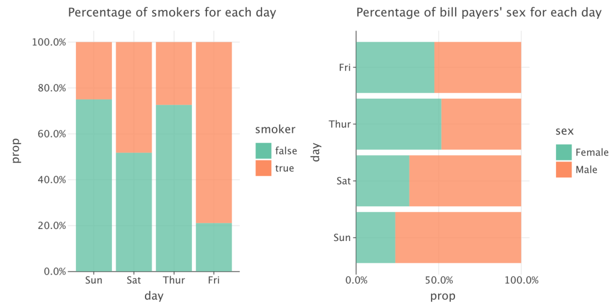
```
val tablesPlt = ggplot(tips.select { day and size }.sortBy("size").toMap()) { x = "day
↪" } +
    geomBar(stat = Stat.count(), position=positionDodge(), alpha=0.8 ) {
        y = "..count.." ; fill=asDiscrete("size")
    } +
    ggtitle("Tables served by Day and party Size")

tablesPlt
```



- Fridays are the quietest days. Saturdays are the busiest followed by Sundays, meaning that there are more customers in the weekend.
- The most common party size is by far *2*, and there are very a few lone diners.

Note: The `tooltips` option is used to customize the tooltip when you hover the mouse on the graph (available only on the web).

```
val p1 = ggplot(tips.select { day and smoker}.toMap()) { x = "day" } +
    geomBar(
        stat = Stat.count(),
        position = positionFill(),
        alpha = 0.8,
        tooltips = layerTooltips("smoker")
            .format("..prop..", ".1%")
            .line("perc. |@..prop..")

    ) { y = "..prop.."; fill = "smoker"} +
    scaleYContinuous(format=".1%") +
    ggtitle("Percentage of smokers for each day")

val p2 = ggplot(tips.select { day and sex }.toMap()) { x = "day" } +
    geomBar(
        stat = Stat.count(),
        position = positionFill(),
        alpha = 0.8,
        tooltips = layerTooltips("sex")
            .format("..prop..", ".1%")
            .line("perc. |@..prop..")
        ) { y = "..prop.." ; fill = "sex" } +
    coordFlip() +
    scaleYContinuous(format=".1%") +
    ggtitle("Percentage of bill payers' sex for each day")

val plt = GGBunch().addPlot(p1, 0, 0, 400, 400).addPlot(p2, 400, 0, 400, 400)
plt
```

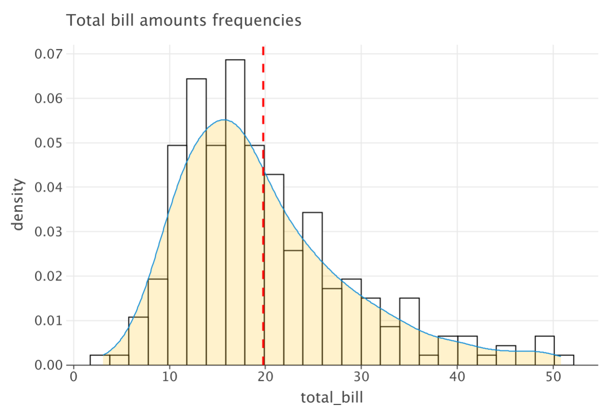


It's very easy now to notice that:

- There are almost equal numbers of male and female that pay the bill in the weekday, but the number of male increases at the weekend.
- The percentage of non smokers is most of the time major that the total percentage, but in the day with least people in the restaurant (Friday), most of them are smokers.

Let's analyze now *quantitative variables*: `total_bill` and `tips`.

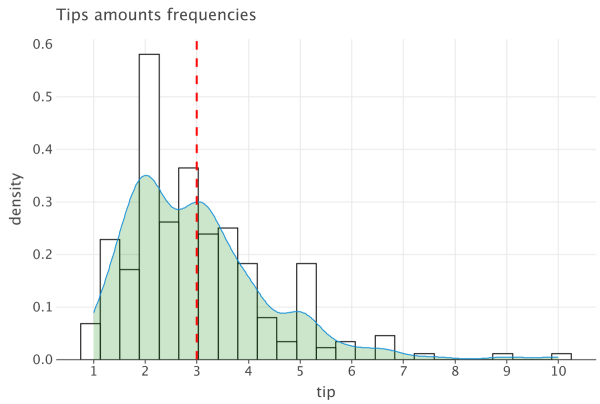
```
ggplot(tips.toMap()) { x = "total_bill" } +
    geomHistogram(bins = 25, fill="white", color="black") { y = "..density.." } +
    geomArea(stat = Stat.density(), fill = "orange", alpha = 0.2) +
    geomVLine(xintercept = tips.total_bill.mean(), color="red", linetype = "dashed",␣
↪size = 1.0) +
    ggtitle("Total bill amounts frequencies")
```



This histogram shows that the average bill amount falls inside the range from 10 to 25 dollars, with it's mean located at about 20 dollars (red dashed line at 19.8).

We can make the same plot, but with `tips` instead

```
ggplot(tips.toMap()) { x = "tip" } +
    geomHistogram(bins = 25, fill="white", color="black") { y = "..density.." } +
    geomArea(stat = Stat.density(), fill = "dark-green", alpha = 0.2) +
    geomVLine(xintercept = tips.tip.mean(), color="red", linetype = "dashed", size =␣
↪1.0) +
    ggtitle("Tips amounts frequencies")
```



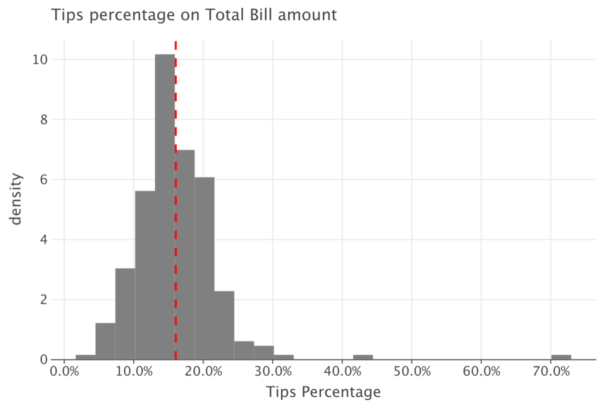
As shown above, the tips peak is at about two dollars, while the mean is right about at three dollars.

It would be more interesting to see the distribution of the tips in relation to its total bill.

```
var data = tips.add("tip_pct") { tip / total_bill }
data.print(5)
```

```
   total_bill  tip    sex smoker day   time size  tip_pct

 0      16.99 1.01 Female  false Sun Dinner    2 0.059447

 1      10.34 1.66   Male  false Sun Dinner    3 0.160542

 2      21.01 3.50   Male  false Sun Dinner    3 0.166587

 3      23.68 3.31   Male  false Sun Dinner    2 0.139780

 4      24.59 3.61 Female  false Sun Dinner    4 0.146808
...
```

```
ggplot(data.toMap()) { x="tip_pct" } +
    geomHistogram(
        bins = 25,
        fill="gray",
        tooltips = layerTooltips("tip_pct")
            .format("tip_pct", ".1%")
    ) { y = "..density.." } +
    geomVLine(
        xintercept = data.tip_pct.mean(),
        linetype = "dashed",
        color = "red",
        size = 1.0,
    ) +
    scaleXContinuous(format = ".1%") +
    xlab("Tips Percentage") +
    ggtitle("Tips percentage on Total Bill amount")
```



We can see that the peak is at about 15% of the total bill. We can spot also some outliers, and let's see their details in the dataframe.

```
data.sortBy { tip_pct.desc() }.print(5)
```

```
   total_bill  tip    sex smoker day   time size  tip_pct

 0       7.25 5.15   Male   true Sun Dinner    2 0.710345

 1       9.60 4.00 Female   true Sun Dinner    2 0.416667

 2       3.07 1.00 Female   true Sat Dinner    1 0.325733

 3      11.61 3.39   Male  false Sat Dinner    2 0.291990

 4      23.17 6.50   Male   true Sun Dinner    4 0.280535
...
```

It can also be interesting to analyze the amount of money spent by each person inside a group

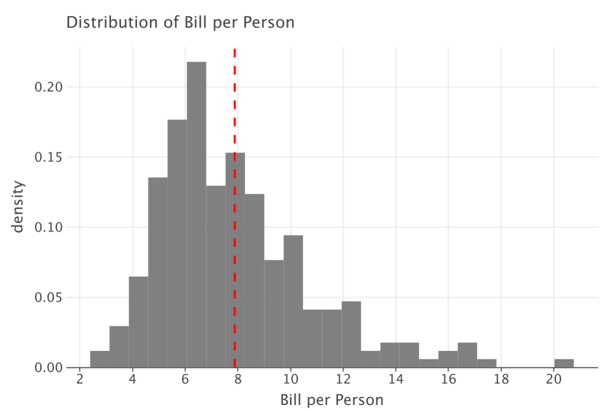
```
// adding Bill Per Person col
data = data.add("bill_pp") { total_bill / size }
data.print(5)
```

```
   total_bill  tip    sex smoker day   time size  tip_pct   bill_pp

 0      16.99 1.01 Female  false Sun Dinner    2 0.059447  8.495000

 1      10.34 1.66   Male  false Sun Dinner    3 0.160542  3.446667

 2      21.01 3.50   Male  false Sun Dinner    3 0.166587  7.003333

 3      23.68 3.31   Male  false Sun Dinner    2 0.139780 11.840000
```

```
 4      24.59 3.61 Female  false Sun Dinner    4 0.146808  6.147500
...
```

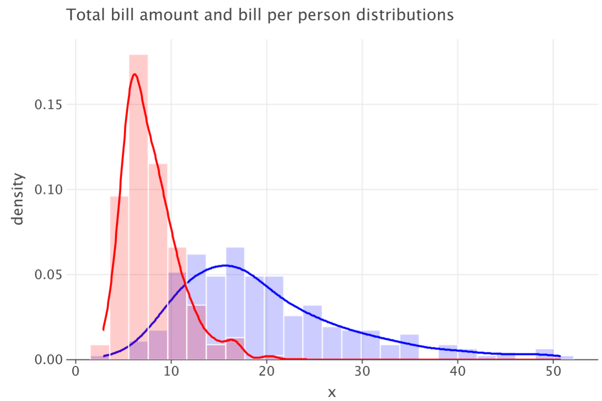
And similarly as above:

```
ggplot(data.toMap()) { x="bill_pp" } +
    geomHistogram(
        bins = 25,
        fill="gray",
    ) { y = "..density.." } +
    geomVLine(
        xintercept = data["bill_pp"].cast<Double>().mean(),
        linetype = "dashed",
        color = "red",
        size = 1.0,
    ) +
    xlab("Bill per Person") +
    ggtitle("Distribution of Bill per Person")
```



It can be useful to see the bill per person with `total_bill` in the same plot.

```
ggplot(data.toMap()) +
    geomHistogram(
        bins=25, fill="blue",
        color="white", alpha=0.2) {
            x="total_bill" ; y="..density.."
    } +
    geomLine(stat = Stat.density(), color="blue", size=1.0) {
        x = "total_bill"
    } +
    geomHistogram(bins=25, fill="red", color="white", alpha=0.2) {
        x="bill_pp" ; y="..density.."
    } +
    geomLine(stat = Stat.density(), color="red", size=1.0) { x="bill_pp"} +
    ggtitle("Total bill amount and bill per person distributions")
```



We want to see if there is correlation with smokers, group size and tip percentage:

```
val smokersData =
    data.groupBy { size }
        .pivot { smoker }
        .mean { tip_pct }
        .sortBy { size }

smokersData.print()
```

```
  size                            smoker
0    1 { false:0.159829, true:0.274755 }

1    2 { false:0.164996, true:0.166706 }
2    3 { false:0.149671, true:0.157543 }

3    4 { false:0.147604, true:0.142036 }
4    5 { false:0.178415, true:0.086116 }

5    6                { false:0.156229 }
```

In order to easily process data for plotting, we rearrange data as follows

```
val data = smokersData.flatten().gather("false", "true").into("smoker", "tip_pct")
data.print()
```

```
   size smoker  tip_pct
 0    1  false 0.159829
 1    1   true 0.274755
 2    2  false 0.164996

 3    2   true 0.166706
 4    3  false 0.149671
 5    3   true 0.157543
 6    4  false 0.147604

 7    4   true 0.142036
 8    5  false 0.178415
```

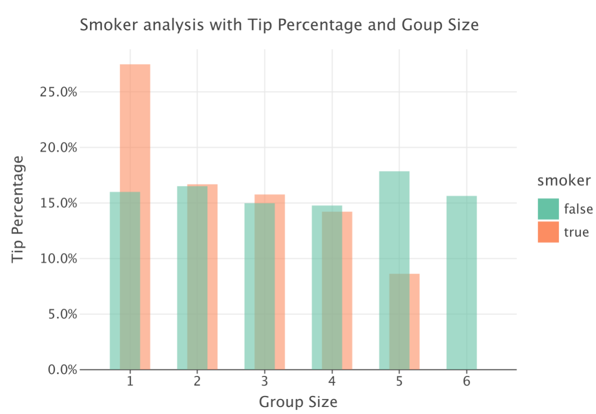
(continues on next page)

(continued from previous page)

```
 9     5   true 0.086116
10     6  false 0.156229
```

```
11     6   true      null
```

```
ggplot(data.toMap()) { x = "size" ; y="tip_pct" } +
    geomBar(
        stat = Stat.identity,
        position=positionDodge(0.3),
        alpha=0.6,
        tooltips = layerTooltips("smoker", "tip_pct")
            .format("tip_pct", ".1%")
    ) { fill="smoker" } +
    ylab("Tip Percentage") +
    scaleYContinuous(format=".1%") +
    xlab("Group Size") +
    ggtitle("Smoker analysis with Tip Percentage and Goup Size")
```



We can see that smoker's tip percentage is generally lower that non smoker's. Even on Friday, the day with most smokers, the tips of non-smokers people are higher.

8.2 NBA Data Analysis

The dataset used, available at https://www.kaggle.com/datasets/justinas/nba-players-data, contains over than two decades of information about NBA players.

```
%use dataframe
%use lets-plot
```

```
val raw_df = DataFrame.readCSV("../resources/example-datasets/datasets/all_seasons.csv
↪")
raw_df.columnNames()
```

```
[untitled, player_name, team_abbreviation, age, player_height, player_weight,␣
↪college, country, draft_year, draft_round, draft_number, gp, pts, reb, ast, net_
↪rating, oreb_pct, dreb_pct, usg_pct, ts_pct, ast_pct, season]
```

As showed above, the dataset includes demographic variables like age, height, weight and place of birth, as well as biographical details such as the team they played for, draft year and round. Additionally, it contains basic box score statistics such as games played, average number of points, rebounds, assists, etc.

Let's look through the data types that has been interpreted by DataFrame:

```
raw_df.schema()
```

```
untitled: Int
player_name: String
team_abbreviation: String
age: Double
player_height: Double
player_weight: Double
college: String?
country: String
draft_year: String
draft_round: String
draft_number: String
gp: Int
pts: Double
reb: Double
ast: Double
net_rating: Double
oreb_pct: Double
dreb_pct: Double
usg_pct: Double
ts_pct: Double
ast_pct: Double
season: String
```

At a first sight we notice that the column `untitled` is useless, so we will remove it. We also notice that it would be more convenient to store ages as `Int` instead of `Double`. For all the columns concerning **draft**, the type is string because of Undrafted players, so we will keep the type `String` for convenience.

Let's apply this chain of changes and assign the new dataframe to a new object

```
val df = raw_df.remove { untitled }
    .convert { age }.toInt()
```

As the type suggests, some `college` rows are missing; we can check how many of them are missing with

```
// check if any record are missing
df.describe().filter { it["nulls"] != 0 }.select("name", "nulls").print()
```

```
      name nulls
 0 college     5
```

8.2.1 Data Analysis

Drafts

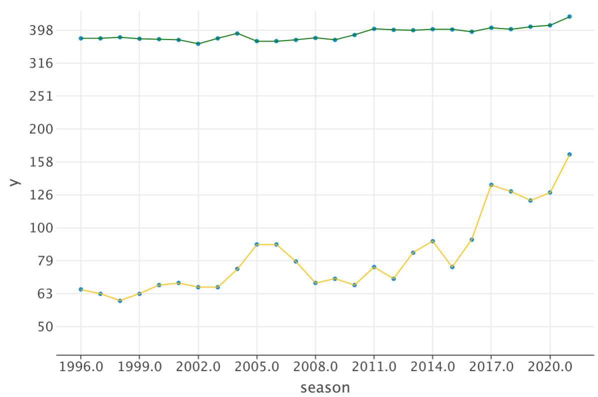
Let us analyze players drafted and undrafted for each season.

```
val drafts = df.groupBy { season.map { it.split('-')[0] } }.aggregate {
    count { draft_year != "Undrafted" } into "drafted"
    count { draft_year == "Undrafted" } into "undrafted"
}.convert { season }.toInt()

drafts.print(5)
```

```
   season drafted undrafted
 0   1996     376        65
 1   1997     376        63

 2   1998     379        60
 3   1999     375        63
 4   2000     374        67
...
```

We can visualize this difference over the years

```
val drafted = geomPoint() { y="drafted" } +
    geomLine(color="darkgreen") { y="drafted" }

val undrafted =  geomPoint() { y="undrafted"} +
    geomLine(color="orange") { y="undrafted"}

ggplot(drafts.toMap()) { x =  "season" } + drafted + undrafted +
    scaleXContinuous(breaks = (1996..2021 step 3).toList()) +
    scaleYLog10()
```



We can see that the number of drafted players are three times more than undrafted players in each season. There is an increase in the trend of undrafted players in 2017-18 season, because that was the year when the "two way contract" rule applied, which help undrafted players secure deals with NBA franchises.

Height and Weight

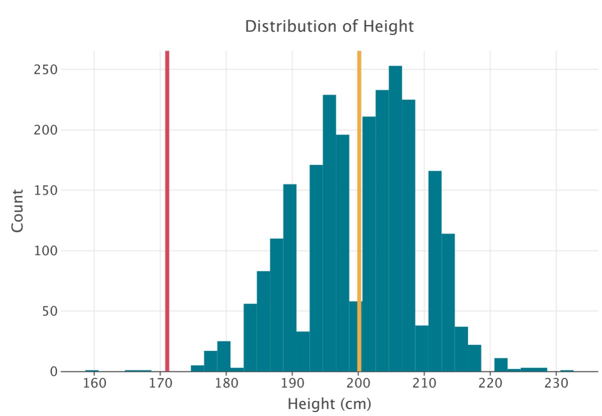
We can summarize player's physical data with the Body Mass Index (BMI) value for each player. Before of this, we must track a player changes during the years, so we will compute an average of weight and height.

```
val physical_data = df.select { player_name and player_height and player_weight }
    .groupBy { player_name }.mean()
    .add("BMI") { player_weight / (Math.pow(player_height / 100, 2.0))}

physical_data[0..5].print()
```

	player_name	player_height	player_weight	BMI
0	Dennis Rodman	199.390000	97.522280	24.529975
1	Dwayne Schintzius	217.170000	123.603820	26.207900
2	Earl Cureton	205.740000	95.254320	22.503352
3	Ed O'Bannon	203.200000	100.697424	24.387706
4	Ed Pinckney	205.740000	108.862080	25.718116
5	Eddie Johnson	200.660000	97.522280	24.220451

We can then plot the distribution of height, adding the global male height average of 171 cm.

```
ggplot(physical_data.toMap()) { x="player_height"} +
    geomHistogram(binWidth = 2, fill="#00798c") +
    geomVLine(xintercept = 171, size = 2.0, color="#d1495b") +
    geomVLine(xintercept = physical_data.player_height.mean(),
              size = 2.0, color="#edae49") +
    labs(title="Distribution of Height", x="Height (cm)", y="Count") +
    theme(title = elementText(hjust = 0.5))
```

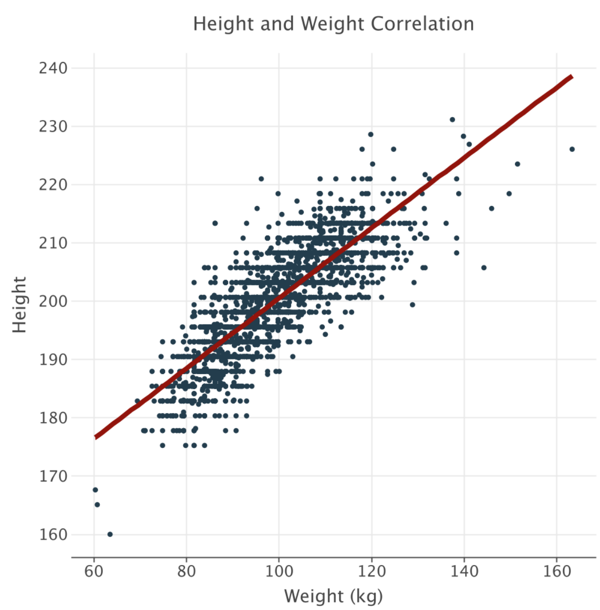


Where the red line is the male average height, and the golden one is the NBA average height.

It can be useful to see how's the correlation between weight and height, and we can compute it with Pearson's correlation coefficient, computed as: $\rho_{X,Y} = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \overline{X})^2 \sum_{i=1}^{n}(Y_i - \overline{Y})^2}}$

```
val corrHeightWeight = ggplot(physical_data.toMap()) { x="player_weight" ; y="player_
↪height"} +
    geomPoint(color = "#233d4d") +
    labs(title = "Height and Weight Correlation", x = "Weight (kg)", y = "Height") +
    theme(title = elementText(hjust = 0.5)) +
    ggsize(500, 500)

// Adding correlation
val correlation = physical_data.corr { player_weight }
        .with { player_height }["player_height"].values().toList()[0]

println("Correlation: $correlation")
corrHeightWeight +
    geomSmooth(method = "lm", deg = 1, color ="#92140c", size = 2.0, se = false)
```

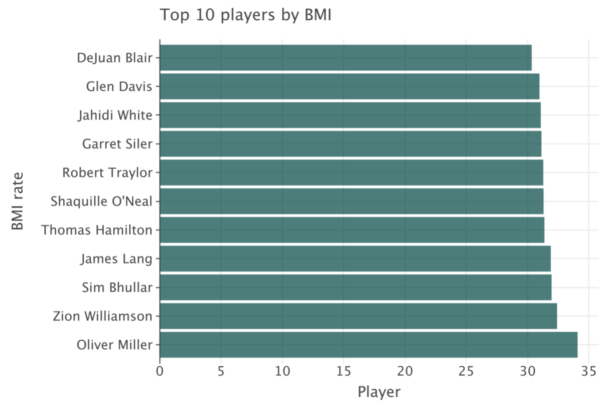
```
Correlation: 0.8210705060051193
```



We can determine that height and weight are fairly strong correlated variables.

Let's see now the top 10 players with highest BMI:

```
val topBMI = physical_data.sortBy { BMI.desc() }[0..10]

ggplot(topBMI.toMap()) { x = "player_name" ; y = "BMI" } +
    geomBar(stat = Stat.identity, fill = "#004643", alpha=0.7) +
    coordFlip() +
    labs(title = "Top 10 players by BMI", x = "BMI rate", y = "Player")
```



According to ourworldindata.org, 95% of male height lie between 163cm to 193cm. With the average of 203 cm, most of NBA Players are on 5% of entire population with height above 193cm.

We can then get the highest and the shortest player ever in the NBA:

```
physical_data.minBy { player_height }.concat(
    physical_data.maxBy { player_height}
).print()
```

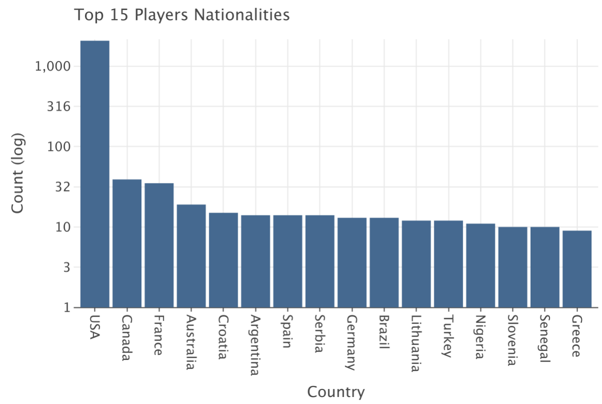
	player_name	player_height	player_weight	BMI
0	Muggsy Bogues	160.02	63.502880	24.799612
1	Gheorghe Muresan	231.14	137.438376	25.725143

Players Nationalities

Being the NBA USA's professional basketball league, most of the players are from North America. We can create a frame and a plot visualizing each year how many new players were from USA and how many of them are foreigners.

Let's first visualize top 15 countries.

```
val topCountries = df.distinctBy { player_name }
    .select { player_name and country }
    .groupBy { country }
    .count()
    .sortBy { "count"<Int>().desc() }

ggplot(topCountries[0..15].toMap()) { x = "country" ; y="count" } +
    geomBar(stat = Stat.identity, fill = "#456990") +
    scaleYLog10() +
    labs(title = "Top 15 Players Nationalities", x = "Country", y = "Count (log)")
```

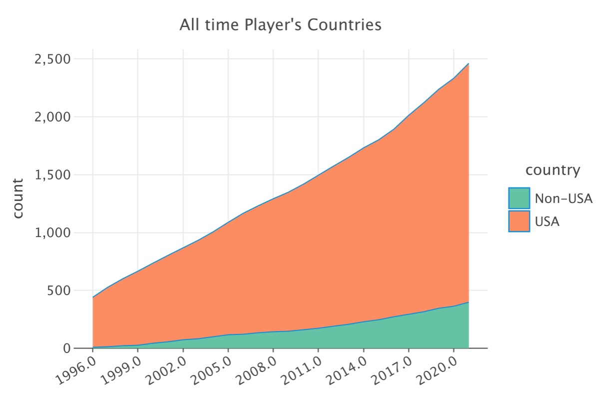


We can see during the years how many USA players have been vs. how many foreign players.

```
val yearNationalities = df.distinctBy { player_name }
    .groupBy { season.map { it.split('-')[0] } }
    .aggregate {
        count { country != "USA" } into "Non-USA"
        count { country == "USA" } into "USA"
    }.cumSum()
    .gather("Non-USA", "USA").into("country", "count")
    .convert { season }.toInt()

yearNationalities.tail(5).print()
```

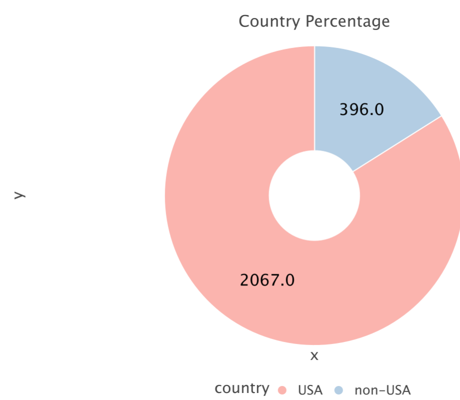
```
   season country count
0    2019     USA  1894
1    2020 Non-USA   363
2    2020     USA  1970

3    2021 Non-USA   396
4    2021     USA  2067
```

```
ggplot(yearNationalities.toMap()) { x="season"; y="count"} +
    geomArea(stat = Stat.identity) { fill="country"} +
    scaleXContinuous(breaks = (1996..2021 step 3).toList()) +
    theme(title = elementText(hjust = 0.5)) +
    ggtitle("All time Player's Countries")
```



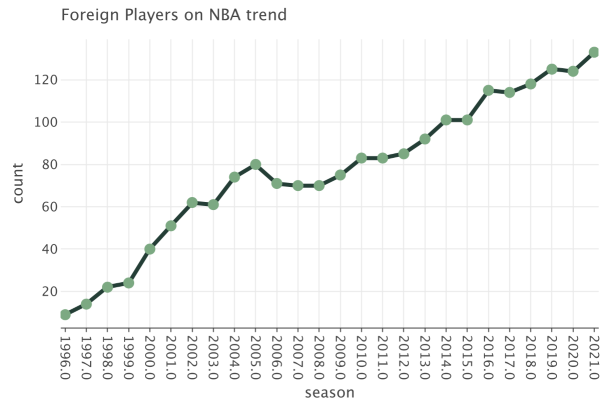
And we can plot the overall percentage of USA and foreign players.

```
val countNations = df.distinctBy { player_name }
    .groupBy { country.map { it == "USA" } }.count()
    .convert("country").with { if ("country"<Boolean>()) "USA" else "non-USA" }
    .toMap()

ggplot(countNations) +
    geomPie(stat = Stat.identity,
            size = 30, stroke = 1, strokeColor = "white", hole = 0.3,
            labels = layerLabels().line("@count").size(16),
    ) { slice = "count" ; fill = "country" } +
    theme(
        line = elementBlank(),
        axis = elementBlank(),
        title = elementText(hjust=0.5)
    ).legendPositionBottom() +
    scaleFillBrewer(palette = "Pastel1") +
    ggtitle("Country Percentage")
```



And finally, we can visualize foreign players trend since 1996

```
val foreignersCount = df.groupBy { season.map { it.split('-')[0] } }
                        .count { country != "USA" }
                        .convert { season }.toInt().toMap()

ggplot(foreignersCount) { x = asDiscrete("season") ; y = "count" } +
    geomLine(color = "#243e36", size = 2.0) +
    geomPoint(size = 5.0, color="#7ca982") +
    ggtitle("Foreign Players on NBA trend")
```



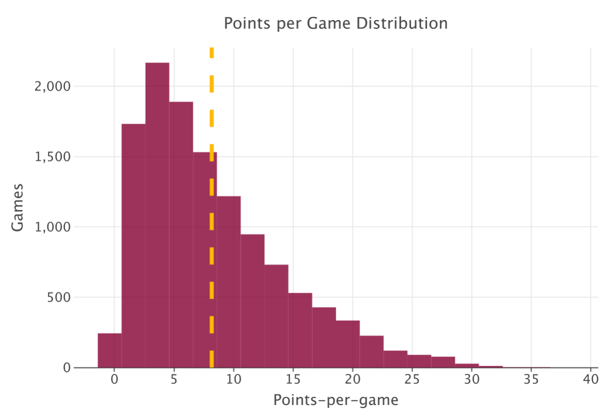
Not surprisingly, as the USA Basketball League, North American players are still dominating the NBA, with the USA only at 84%, but the number of the foreign players is increasing progressively. Even if they are the minority of the league, since 2019 to today (2023) NBA's Most Valuable Player prize has been won by foreigners!

Players Statistics

In this section we will go through in game statistics for evaluating a player excellence, analyzing points, assists and rebounds per game.

Points Per Game

```
ggplot(df.toMap()) { x="pts" } +
    geomHistogram(binWidth = 2, fill = "#840032", alpha=0.8) { y="..count.." } +
    geomVLine(xintercept = df.pts.mean(), color = "#fcba04",
              size = 2.0, linetype = "dashed") +
    labs(title="Points per Game Distribution",
         x="Points-per-game",
         y="Games") +
    theme(title = elementText(hjust=.5))
```



We can see that exceptional performances are above 25 points per game.

We can write a simple `quantile` function to extract the top 1% and 10% of point per game performances.

```
fun quantile(perc: Double=0.99, data: List<Double>): List<Double> =
    data.sortedDescending()
        .subList(0, (perc * data.size).toInt())
```

```
val ppgQuantile = (1..10).map {
    quantile(it.toDouble() / 100.0, df.pts.toList()).average()
}

val ppgDf = dataFrameOf(
    "Percentile" to (99 downTo 90).map { it.toDouble() / 100 },
    "PPG" to ppgQuantile
)

ppgDf.print()
```

```
  Percentile       PPG
0       0.99 28.452846
1       0.98 26.763821
2       0.97 25.462602
```

```
3       0.96 24.484756
4       0.95 23.699187
5       0.94 23.049593
6       0.93 22.472938
```

```
7       0.92 21.960772
8       0.91 21.492954
9       0.90 21.061707
```

Let's rank now the top 10 players to have a point-per-game statistic in the top 1%, and the highest number of seasons played with the team.

```
df.filter { pts >= ppgDf.PPG[0] }
    .update { season }.with { it.split('-')[0] }
    .convert { season }.toInt()
    .groupBy { player_name }.aggregate {
        count() into "Seasons"
        mean { pts } into "Avg PPG"
    }.sortBy { "Seasons"<Int>().desc() }[0..10].print()
```

```
        player_name Seasons   Avg PPG
 0     James Harden       5 31.780000
```

```
 1    Allen Iverson       4 31.550000
 2      Kobe Bryant       4 31.375000
```

```
 3     LeBron James       4 30.350000
 4     Kevin Durant       3 30.666667
```

```
 5   Michael Jordan       2 29.150000
 6 Shaquille O'Neal       2 29.200000
```

```
 7  Carmelo Anthony       2 28.800000
 8    Stephen Curry       2 31.050000
```

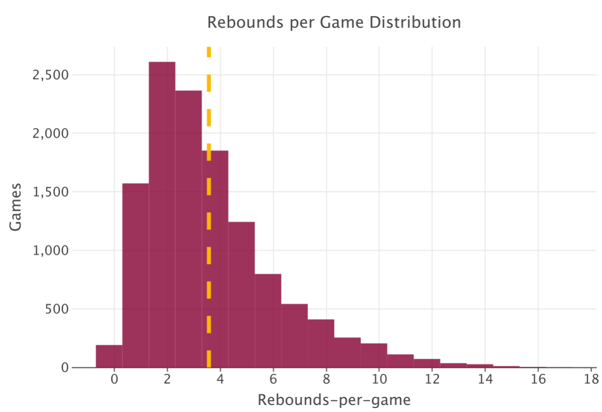
```
 9   Damian Lillard       2 29.400000
10     Bradley Beal       2 30.900000
```

From the above dataframe, James Harden has the highest number of seasons averaging 31.78 points per game. Now we have a clearer picture of what characterized an excellent scorer.

Rebounds Per Game

Similarly as above, we can understand what values of rebounds per game characterize the best rebounder in the league.

```
ggplot(df.toMap()) { x="reb" } +
    geomHistogram(binWidth = 1, fill = "#840032", alpha=0.8) { y="..count.." } +
    geomVLine(xintercept = df.reb.mean(), color = "#fcba04",
              size = 2.0, linetype = "dashed") +
    labs(title="Rebounds per Game Distribution",
         x="Rebounds-per-game",
         y="Games") +
    theme(title = elementText(hjust=.5))
```



On average, an NBA player take 3 to 4 rebounds per game.

We can find the best 1% to 10% rebound per games values

```
val rebQuantile = (1..10).map {
    quantile(it.toDouble() / 100.0, df.reb.toList()).average()
}

val rebDf = dataFrameOf(
    "Percentile" to (99 downTo 90).map { it.toDouble() / 100 },
    "reb" to rebQuantile
)

rebDf.print()
```

```
  Percentile       reb
0       0.99 12.929268
1       0.98 11.934959
2       0.97 11.302168
```

```
3       0.96 10.832927
4       0.95 10.440163
5       0.94 10.100949
6       0.93  9.805343
```

```
7       0.92  9.541870
8       0.91  9.306052
9       0.90  9.092439
```

The players that falls into the top 1% rebounder, for the highest number of seasons are

```
df.filter { reb >= rebDf.reb[0] }
    .update { season }.with { it.split('-')[0] }
    .convert { season }.toInt()
    .groupBy { player_name }.aggregate {
        count() into "Seasons"
        mean { reb } into "Avg RPG"
    }.sortBy { "Seasons"<Int>().desc() }[0..10].print()
```

```
         player_name Seasons   Avg RPG
 0    Andre Drummond       7 14.585714
```

```
 1    DeAndre Jordan       6 14.083333
 2     Dwight Howard       5 13.960000
```

```
 3     Dennis Rodman       3 15.133333
 4       Ben Wallace       3 13.866667
```

```
 5     Kevin Garnett       3 13.600000
 6        Kevin Love       3 14.166667
```

```
 7       Rudy Gobert       3 13.900000
 8   Jayson Williams       2 13.550000
```

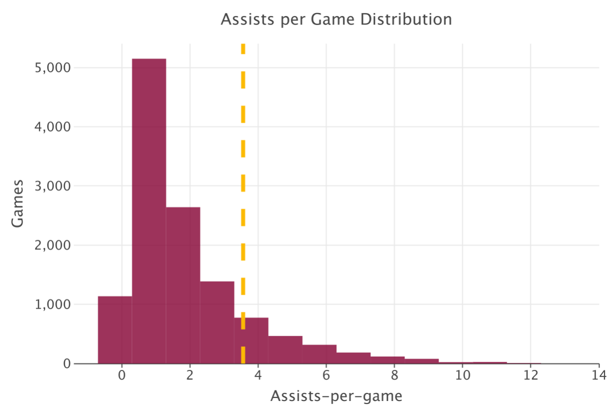
```
 9   Dikembe Mutombo       2 13.800000
10 Hassan Whiteside       2 13.800000
```

We can see that Andre Drummond is the most consistent rebounder, but Dennis Rodman is the best rebounder since 1996.

Assists Per Game

Lastly, we will cover Assists per Game.

```
ggplot(df.toMap()) { x="ast" } +
    geomHistogram(binWidth = 1, fill = "#840032", alpha=0.8) { y="..count.." } +
    geomVLine(xintercept = df.reb.mean(), color = "#fcba04",
              size = 2.0, linetype = "dashed") +
    labs(title="Assists per Game Distribution",
         x="Assists-per-game",
         y="Games") +
    theme(title = elementText(hjust=.5))
```



The mean value is from 3 to 4 assists per game, but the most common values are from one to two.

As above, we compute the quantiles from 1 to 10% top assists per game.

```
val astQuantile = (1..10).map {
    quantile(it.toDouble() / 100.0, df.ast.toList()).average()
}

val astDf = dataFrameOf(
    "Percentile" to (99 downTo 90).map { it.toDouble() / 100 },
    "ast" to astQuantile
)

astDf.print()
```

```
  Percentile      ast
0       0.99 9.568293
1       0.98 8.679268
2       0.97 8.083740
```

```
3       0.96 7.635366
4       0.95 7.274797
5       0.94 6.971951
6       0.93 6.708595
```

```
7       0.92 6.474085
8       0.91 6.262782
9       0.90 6.070244
```

And the players with the highest number of seasons averaging the 1% quartile are:

```
df.filter { ast >= astDf.ast[0] }
    .update { season }.with { it.split('-')[0] }
    .convert { season }.toInt()
    .groupBy { player_name }.aggregate {
        count() into "Seasons"
        mean { ast } into "Ast RPG"
    }.sortBy { "Seasons"<Int>().desc() }[0..10].print()
```

```
           player_name Seasons   Ast RPG
 0         Chris Paul        9 10.500000

 1         Steve Nash        8 10.937500
 2        Rajon Rondo        6 10.883333

 3         Jason Kidd        5 10.140000
 4  Russell Westbrook        5 10.700000

 5     Deron Williams        4 10.500000
 6          John Wall        4 10.125000

 7       James Harden        3 10.766667
 8     Rod Strickland        2 10.200000

 9       John Stockton       1 10.500000
10        Mark Jackson       1 11.400000
```

Chris Paul is the most consistent of all players from 1996 to today when it comes to assists per game, where Rajon Rondo has the best assist per game season:

```
df.sortBy { ast.desc() }.select { player_name and ast }[0].print()
```

```
{ player_name:Rajon Rondo, ast:11.700000 }
```

8.2.2 College Ranking

The last section will summarize the above statics (points, rebounds, assists)

Let's create a college ranking based on player's total games played in NBA

```
val careerGames = df.groupBy { player_name }.sum { gp }

val college_rank = careerGames.join(df) { player_name match right.player_name }
    .select { player_name and college and gp }
```

(continues on next page)

(continued from previous page)

```
    .distinctBy { player_name }
    .rename { gp }.into("total_games")
    .groupBy { college }.sum("total_games")
    .sortByDesc("total_games")
    .filter { college != "None" }.add("rank") { index() }
```

```
college_rank[0..10].print()
```

```
           college total_games rank
 0        Kentucky       23051    0
 1            Duke       20584    1
 2  North Carolina       19723    2
 3            UCLA       16444    3
 4         Arizona       15569    4
 5          Kansas       15269    5
 6     Connecticut       13552    6
 7    Georgia Tech       10409    7
 8         Florida       10402    8
 9        Michigan        9516    9
10           Texas        9345   10
```

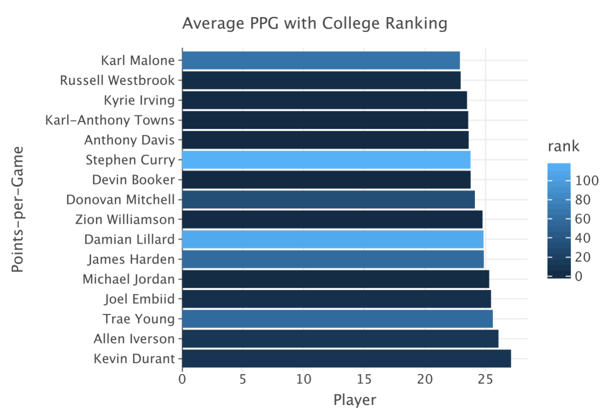
We can then plot player's best **points-per-game** season, showing the rank of the college he comes from

```
val bestScorer = df.groupBy { player_name }.mean { pts }.sortBy { pts.desc() }
                    .join(
                        df.distinctBy { player_name }
                            .select { player_name and college }
                    ) { player_name match right.player_name }
                    .filter { college != "None" }
                    .join(
                        college_rank.select { college and rank }
                    ) { college match right.college }


bestScorer[0..10].print()
```

```
          player_name       pts        college rank
 0      Kevin Durant 27.100000          Texas   10
 1     Allen Iverson 26.064286     Georgetown   12
```

2	Trae Young	25.600000	Oklahoma	58
3	Joel Embiid	25.450000	Kansas	5
4	Michael Jordan	25.300000	North Carolina	2
5	James Harden	24.853846	Arizona State	59
6	Damian Lillard	24.830000	Weber State	110
7	Zion Williamson	24.750000	Duke	1
8	Donovan Mitchell	24.120000	Louisville	33
9	Devin Booker	23.771429	Kentucky	0
10	Stephen Curry	23.761538	Davidson	116

```
val tooltipOptions = layerTooltips()
                        .line("college|@college")
                        .line("rank|@rank ")


ggplot(bestScorer[0..15].toMap()) { x="player_name" ; y="pts" } +
    geomBar(stat = Stat.identity, tooltips = tooltipOptions) { fill="rank" } +
    coordFlip() +
    labs(title = "Average PPG with College Ranking",
         x = "Points-per-Game",
         y = "Player")
```



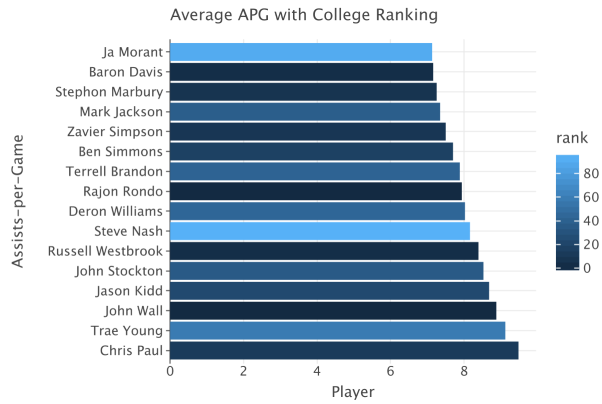
We can see that the higher the rank of the college is, the higher is the number of players in the top 15 scorer.

Let's see if this is true also for rebounds and assists.

```
val bestAst = df.groupBy { player_name }.mean { ast }.sortBy { ast.desc() }
                    .join(
                        df.distinctBy { player_name }
                            .select { player_name and college }
                    ) { player_name match right.player_name }
                    .filter { college != "None" }
                    .join(
                        college_rank.select { college and rank }
                    ) { college match right.college }

bestAst[0..10].print()
```

	player_name	ast	college	rank
0	Chris Paul	9.482353	Wake Forest	13
1	Trae Young	9.125000	Oklahoma	58
2	John Wall	8.880000	Kentucky	0
3	Jason Kidd	8.682353	California	23
4	John Stockton	8.528571	Gonzaga	35
5	Russell Westbrook	8.392857	UCLA	3
6	Steve Nash	8.161111	Santa Clara	94
7	Deron Williams	8.025000	Illinois	44
8	Rajon Rondo	7.937500	Kentucky	0
9	Terrell Brandon	7.883333	Oregon	42
10	Ben Simmons	7.700000	Louisiana State	17

```
ggplot(bestAst[0..15].toMap()) { x="player_name" ; y="ast" } +
    geomBar(stat = Stat.identity, tooltips = tooltipOptions) { fill="rank" } +
    coordFlip() +
    labs(title = "Average APG with College Ranking",
         x = "Assists-per-Game",
         y = "Player")
```



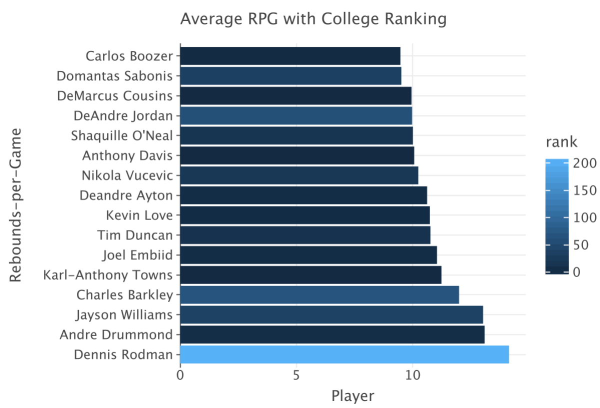
When it comes to top 15 player's assists per game, the college ranking distribution is very similar to the point per game one, with the top 5 with an average ranking of 25.

```
val bestReb = df.groupBy { player_name }.mean { reb }.sortByDesc { reb }
                    .join(
                        df.distinctBy { player_name }
                            .select { player_name and college }
                    ) { player_name match right.player_name }
                    .filter { college != "None" }
                    .join(
                        college_rank.select { college and rank }
                    ) { college match right.college }

bestReb[0..10].print()
```

	player_name	reb	college	rank
0	Dennis Rodman	14.150000	Southeastern Oklahoma State	204
1	Andre Drummond	13.100000	Connecticut	6
2	Jayson Williams	13.033333	St. John's (NY)	38
3	Charles Barkley	12.000000	Auburn	68
4	Karl-Anthony Towns	11.242857	Kentucky	0
5	Joel Embiid	11.050000	Kansas	5
6	Tim Duncan	10.768421	Wake Forest	13
7	Kevin Love	10.742857	UCLA	3
8	Deandre Ayton	10.625000	Arizona	4

9	Nikola Vucevic	10.245455	Southern California	22
10	Anthony Davis	10.070000	Kentucky	0

```
ggplot(bestReb[0..15].toMap()) { x="player_name" ; y="reb" } +
    geomBar(stat = Stat.identity, tooltips = tooltipOptions) { fill="rank" } +
    coordFlip() +
    labs(title = "Average RPG with College Ranking",
         x = "Rebounds-per-Game",
         y = "Player")
```



Dennis Rodman is the best rebounder since 1996, and in his case, the college ranking did not matter. For the rest of the top 15, the college ranking is fairly low among best rebounders (max. 68).

CHAPTER

NINE

BIBLIOGRAPHY

BIBLIOGRAPHY

[Wic11] Hadley Wickham. The Split-Apply-Combine Strategy for Data Analysis. *Journal of Statistical Software*, 40(1):1–29, April 2011. URL: http://www.jstatsoft.org/v40/i01.