lab2

May 15, 2024

1 Spark ML

Rémi Pépin, Arthur Katossky, Ludovic Deneuville

1.1 Before you start

- ☐ Download this Jupyter Notebook
- ☐ Follow these instructions

If your Jupyter is in dark mode, some images may not display properly. You can switch to light mode in Settings > search for theme > Selected Theme: JupyterLab Light

1.2 Outline

In this tutorial, we are going to perform exploratory and explanatory analyses of a massive dataset consisting in hundreds of thousands of AirBnB listings, as made available by the Inside AirBnB project.

You will find these listings at this address:

- `s3://ensai-labs-2023-2024-files/lab2/airbnb/` on AWS
- `s3a://ludo2ne/diffusion/ensai/airbnb/` on SSPCloud

1.3 1 Create a Spark session

- ☐ Depending on the **chosen platform**, initialize the Spark session

1.3.1 1.1 Only on SSPCloud

See default configuration on the datalab :

[1]:
```
! cat /opt/spark/conf/spark-defaults.conf
```

```
spark.driver.extraJavaOptions  -Dcom.amazonaws.sdk.disableCertChecking=false
-Dhttp.nonProxyHosts=localhost -Dhttps.nonProxyHosts=localhost
spark.executor.extraJavaOptions  -Dcom.amazonaws.sdk.disableCertChecking=false
-Dhttp.nonProxyHosts=localhost -Dhttps.nonProxyHosts=localhost
spark.kubernetes.authenticate.driver.serviceAccountName jupyter-pyspark-276551
spark.kubernetes.container.image inseefrlab/onyxia-jupyter-
pyspark:py3.12.2-spark3.5.1
spark.kubernetes.driver.pod.name jupyter-pyspark-276551-0
```

```
spark.kubernetes.namespace user-yann223
spark.master k8s://https://kubernetes.default.svc:443
spark.driver.memory 2g
spark.dynamicAllocation.enabled true
spark.dynamicAllocation.executorAllocationRatio 1
spark.dynamicAllocation.initialExecutors 1
spark.dynamicAllocation.maxExecutors 10
spark.dynamicAllocation.minExecutors 1
spark.dynamicAllocation.shuffleTracking.enabled true
spark.executor.memory 2g
spark.hadoop.fs.s3a.bucket.all.committer.magic.enabled true
```

To modify the config :

[2]:
```
import os
from pyspark.sql import SparkSession

spark = (SparkSession
         .builder
         # default url of the internally accessed Kubernetes API
         # (This Jupyter notebook service is itself a Kubernetes Pod)
         .master("k8s://https://kubernetes.default.svc:443")
         # Executors spark docker image: for simplicity reasons, this jupyter␣
 ↪notebook is reused
         .config("spark.kubernetes.container.image", os.environ['IMAGE_NAME'])
         # Name of the Kubernetes namespace
         .config("spark.kubernetes.namespace", os.
 ↪environ['KUBERNETES_NAMESPACE'])
         # Allocated memory to the JVM
         # Stay careful, by default, the Kubernetes pods has a higher limit␣
 ↪which depends on other parameters.
         .config("spark.executor.memory", "4g")
         .config("spark.kubernetes.driver.pod.name", os.
 ↪environ['KUBERNETES_POD_NAME'])
         # dynamic allocation configuration
         .config("spark.dynamicAllocation.enabled","true")
         .config("spark.dynamicAllocation.initialExecutors","1")
         .config("spark.dynamicAllocation.minExecutors","1")
         .config("spark.dynamicAllocation.maxExecutors","5")
         .getOrCreate()
        )
```

[3]:
```
# See the current number of executors (one for now)
!kubectl get pods -l spark-role=executor
```

```
NAME                                   READY   STATUS    RESTARTS   AGE
pyspark-shell-8bf57a8f7b2ebba8-exec-1  1/1     Running   0          11s
```

1.3.2 1.2 Only on AWS

[4]:
```
#Spark session
spark

# Configuraion
spark._jsc.hadoopConfiguration().set("fs.s3.useRequesterPaysHeader","true")
```

1.3.3 1.3 Check spark session

[4]:
```
spark
```

[4]:
```
<pyspark.sql.session.SparkSession at 0x7f25d855e6c0>
```

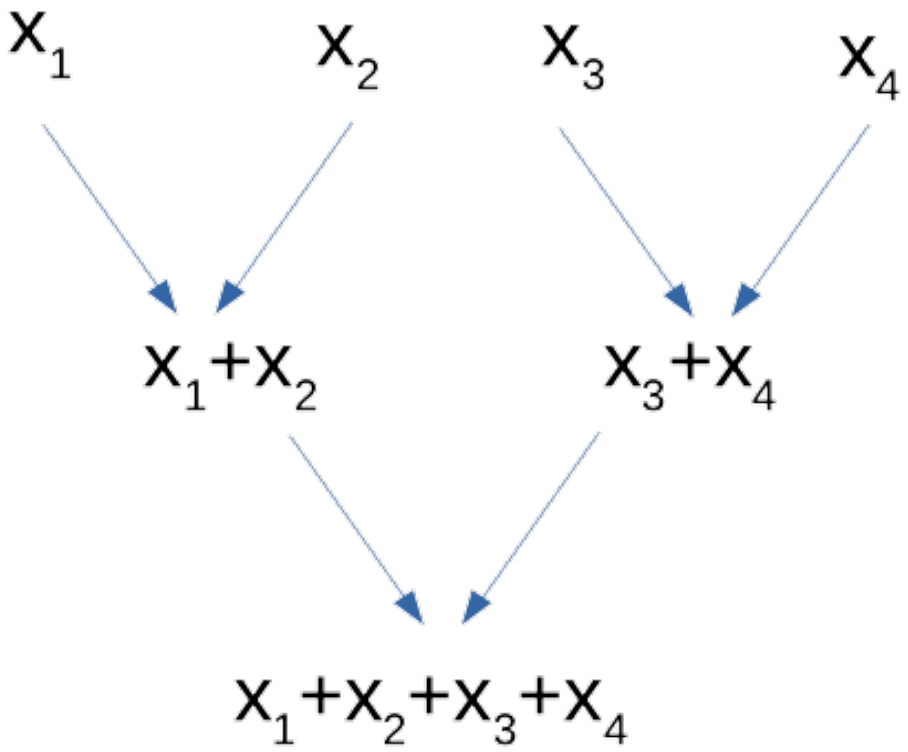
1.4 2 How to distribute elementary statistical tasks?

The map and reduce principle

When your data is distributed, i.e is spread out across multiple hard disks / memories on different logical or physical machines, it is clearly not possible to load everything in memory to perform some computation. (No computer from the cluster would have enough storage space / memory space to load the full data set, and the exchange of information *between* the nodes of the cluster would take considerable amounts of time.) What can you do then?

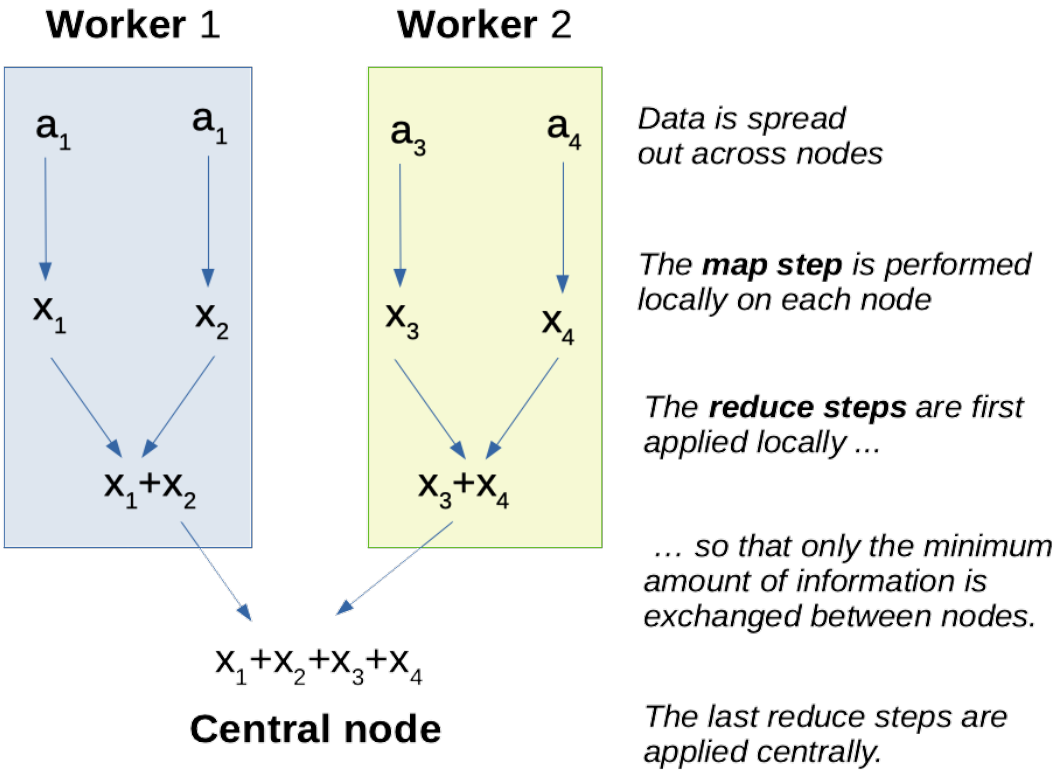
A surprisingly satisfying situation is when your algorithm can be expressed in a `map-and-reduce model`. A `map` step, in computer science, is the equivalent a function in mathematics: from a given entry, return an output. Examples include counting the number of occurrences of a word in a text, squaring some number, subtracting some number, etc. A `reduce` step takes two inputs and produces one input, and can be called recursively onto its own outputs, progressively yielding the final result through a pyramid of `accumulators` (see diagram here under). Popular reduce functions include (pairwise) concatenation of character strings, (pairwise) product, (pairwise) minimum and (pairwise) maximum. But `pairwise addition` is probably the most used reduce function, with the aim goal of performing a complete addition:

Hadoop's MapReduce is the name of what was to become today Apache Spark. The persons behind this framework were among the first to advocate for the map-and-reduce mode in order to achieve efficient parallelisation. Unfortunately, the similarity of the names causes a lot of confusion between the map-and-reduce theoretical model and the concrete Hadoop implementation. I will use "map-and-reduce" to help distinguish the algorithmic concept from the MapReduce program, but this is *not* standard in the literature.



Why is the map-and-reduce scheme so interesting?

Well, say you have n entries and k worker nodes at your disposal. The map operation can always be performed locally on each node, since the transformation does not depend on the rest of the data set. This is an **embarrassingly parallel problem** and we roughly divide the execution time by k. Then, most of the reduce steps can also happen on the worker nodes, until the local data has been completely summarized. This also an k_fold acceleration! Then, there remains only k reduce steps, and since $k \ll n$, this is usually quite negligible, even though the (potentially high) networking costs happen at this step. There is still some cost of task coordination and data exchange, but this usually small compared to the costs of parallelisation.



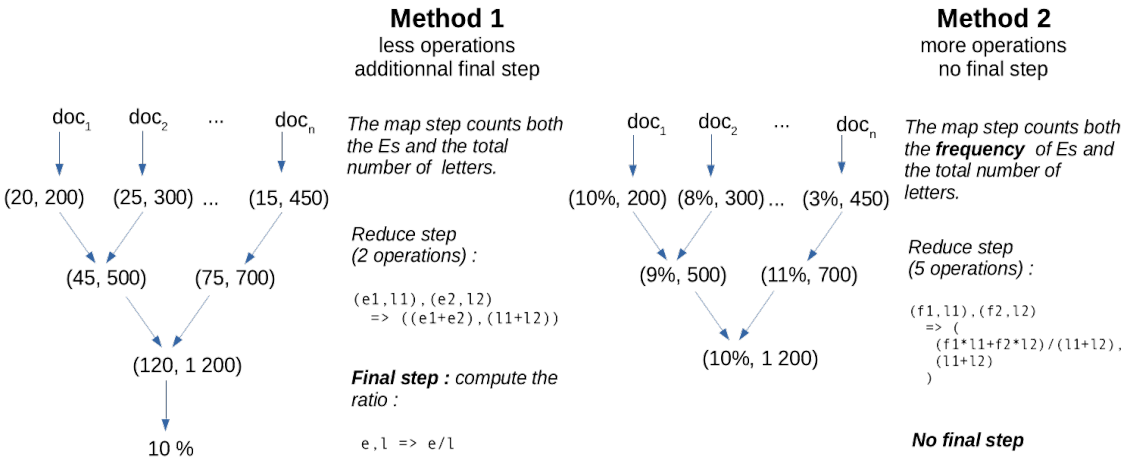
The reduce step

A reduce function is an associative function $f : E \times E \mapsto E$, where associativity means $\forall (a, b, c) \in E^3, f(a, f(b, c)) = f(f(a, b), c)$. This is required because the distribution of data blocks across the nodes is random, and that we want to minimize data transmission between the nodes.

Moreover, f **may or may not be commutative**, in the sense that $f(a, b) = f(b, a)$. If it is the case, such as with addition and multiplication, then the computing may happen in no particular order. This means that the central node need not wait for some partial results to be returned by a belated node. On the contrary, if f is not commutative, (a) the worker nodes must apply the function in a defined order, (b) the central node needs to reduce the intermediate outputs in a defined order, (c) it may have to delay the final reduce steps because of a lingering node.

The reduce function must not be defined on $E = \mathbb{R}$. For instance, in the context where data is a collection of text documents, a word-count function may return accumulator objects looking like: `((word1,count1), (word2,count2))`. Also, the accumulators — that is, the outputs of the each intermediate reduce step — are not necessarily exactly the cumulative version of the final statistic our algorithm outputs! Rather, **accumulators are information-dense, fast-to-compute summary statistics** from which the required final statistics can be obtained.

Imagine you want to count the frequency of the vocal E in English, given a collection of texts. It is faster to count the number of Es as well as the total number of characters than to accumulate directly the frequencies, as shown in this diagram:



Online algorithms

An **online algorithm** is an algorithm with an inner state that can be actualized at low cost for any new arrival of data. A good metaphor is track-keeping of the number of people on a bus: every time a person enters or leaves, you apply ± 1 to the count, without the need to systematically recount everyone. Said otherwise, an online algorithm is any algorithm whose last result can be actualized from new data, at a smaller cost than an alternative algorithm that uses both old and new data from scratch.

It turns out that **respecting the map-and-reduce model gives us online algorithms for free**, where the **inner state** of the algorithm is the output from the last reduce call. Indeed, writing s_{old} and s_{new} the old and new states (the old and new summary statistics), and x_{new} the latest data point, we have:

$$s_{\text{new}} = \text{reduce}(s_{\text{old}}, \text{map}(x_{\text{new}}))$$

Thus, writing an algorithm following the map-and-reduce model gives you both a parallelized batch algorithm and a stream algorithm at once.

Number of passes

So far we have discussed algorithms that require only one map and one reduce functions. But for some statistics, it is not sufficient. For instance, if we want to count the number of texts where the letter E is more common than average, we first have to compute the average frequency in a first pass, then to count the texts where the frequency exceed this number with a second one. We can NOT do this in only one run, since the global average frequency is not known !

Each run is called a **pass** and some algorithms require several passes.

Limits

- Not all statistical algorithms can be expressed according to the map-and-reduce algorithm, and when they can, it may require a significant re-writing compared to the standard algorithms.
- There may be a trade-off between the number of passes, the speed of each map / reduce steps and the volume of data transferred between each reduce step.

1.4.1 2.1 Hands-on 1

- You are given `errors`, a distributed vector of prediction errors `errors = [1, 2, 5, 10, 3, 4, 6, 8, 9]`

☐ Write a map-and-reduce algorithm for computing the **total sum of squares**.
  - You may want to create a Python version of this algorithm, using the `map(function, vector)` and `reduce(function, vector)` functions or you may use lambda-functions
  - You have to import `reduce` from the `functools` module

[8]:
```
from functools import reduce

errors = [1,2,5,10,3,4,6,8,9]

def m_r_1(num_list):
    squared = list(map(lambda x: x**2, num_list))

    sum_r = reduce((lambda x, y: x + y), squared)

    return sum_r

print(m_r_1(errors))
```

```
336
```

☐ Write **two** different map-and-reduce algorithm for computing the ***mean* sum of squares**. *(One may include a final $O(1)$ step.)*

[11]:
```
def alg1(num_list):
    squared = map(lambda x: x**2/len(num_list), num_list)
    sum_r = reduce((lambda x, y: x + y), squared)

    return sum_r

def alg2(num_list):
    squared = map(lambda x: x**2, num_list)
    sum_r_mean = reduce((lambda x, y: x + y), squared)/len(num_list)

    return sum_r_mean

print(alg1(errors))

print(alg2(errors))
```

```
37.33333333333333
37.333333333333336
```

☐ Is the median easy to write as a map-and-reduce algorithm? Why?

[8]:
```
# No
```

☐ Given a (distributed) series of numbers, the variance can be straightforwardly expressed as a two-pass algorithm: (a) in a first pass, compute the mean, then (b) in a second pass, compute the mean of the errors to the mean. Can it be expressed as a one-pass only algorithm? Is it more expensive to compute variance *and* mean instead of the variance alone?

[9]:

1.5 3 Application on Airbnb Data

[13]:
```
from pyspark.sql.types import FloatType, IntegerType, DateType
from pyspark.sql.functions import regexp_replace, col

listings_raw = spark.read.parquet("s3a://ludo2ne/diffusion/ensai/airbnb.
 ↪parquet")
```

[14]:
```
listings = (listings_raw
  .withColumn("beds",     listings_raw["beds"    ].cast(IntegerType()))
  .withColumn("bedrooms", listings_raw["bedrooms"].cast(IntegerType()))
  .withColumn("time", listings_raw["last_scraped"].cast(DateType()))
  .withColumn("price", regexp_replace('price', '[$\\,]', '').cast(FloatType()))
  .select("id", "beds", "bedrooms", "price", "city", "time")
  .dropna() # remove lines with missing values
)
```

[15]:
```
listings_raw.cache()
listings.cache()
```

[15]:
```
DataFrame[id: string, beds: int, bedrooms: int, price: float, city: string,
time: date]
```

1.5.1 3.1 Hands-on 2

☐ How many lines do the raw and the formatted datasets have?

[17]:
```
listings_raw.count()
```

[17]:
```
172304
```

[16]:
```
listings.count()
```

[16]:
```
76068
```

☐ How many columns are there?

☐ Can you list all the available columns?

[19]:
```
listings.columns
```

[19]:
```
['id', 'beds', 'bedrooms', 'price', 'city', 'time']
```

Spark SQL's `summary()` method

In Spark SQL, **elementary univariate summary statistics can also be obtained through the `summary()` method**. The `summary()` method takes either the names of the statistics to compute, or nothing, in which case it computes every possible statistics:

```
listings.summary("count", "min", "max").show() # computes the selection of statistics

listings.summary().show() # computes every possible statistics
```

This is a way to incite you to compute all the statistics you want at the same moment : it avoids an extra pass on the data set because all accumulators can be computed simultaneously. You can fin a list of all supported statistics here in PySpark documentation: count, mean, standard-deviation, minimum, maximum, approximate median, approximate first and last quartiles. Null (missing) values will be ignored in numerical columns before calculation.

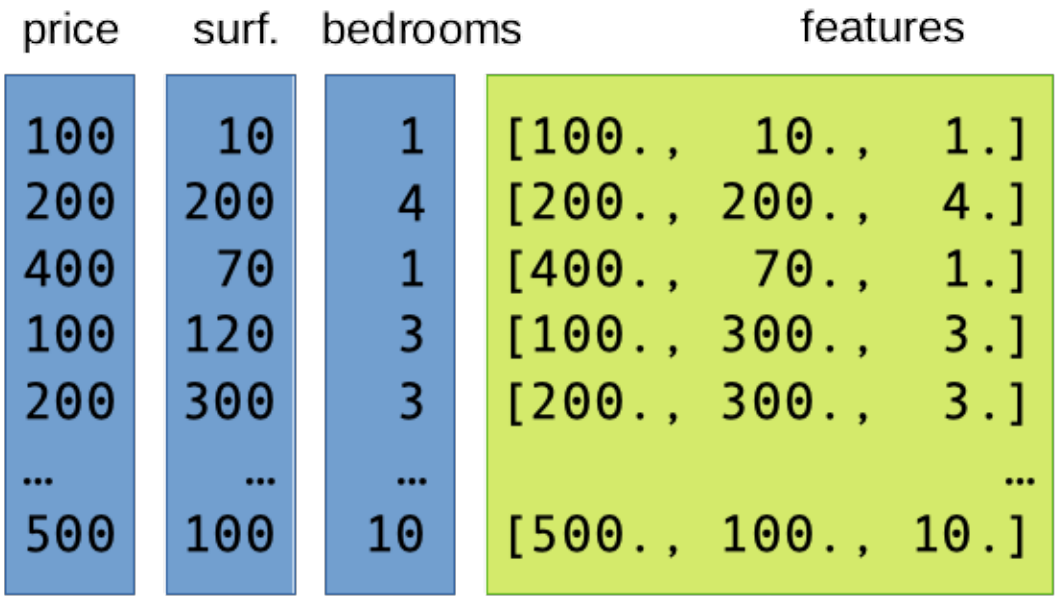
Spark ML

Spark ML is a Spark module that allow us to execute parallelised versions of most popular machine-learning algorithms, such as linear or logistic regression. However, we can also use Spark ML to compute elementaty univariate summary statistics. However the philosophy is quite different, and is worth explaining.

The syntax of Spark ML may feel artificially convoluted ; this not only an impression, it *is* convoluted. However, there are grounds for this situation :

1. Spark ML has been built on top of Spark years into the project, and the core of Spark is not well adapted to machine-learning ;
2. Spark ML is intended for much more advanced treatments than unviariate statistics, and we will see linear regression as an exemple at the end of this tutorial

Step 1: vectorisation. A little counter-intuitively, spark ML operates on a single column of your data frame, typically called `features`. (Features is the word used in the machine-learning community for "variables", see "Vocabulary" section hereunder.) This `features` column has the `Vector` type: each element contains an array of floating-point numbers, representing a subst of the variables from your dataset. The key is that this `features` column is usually redundant with the rest of the data frame: it just ensures the proper conversion from any type we wish (string, integer…) to a standardized numeric format. Indeed, it is often derived from the other columns, as this image illustrates:



price	surf.	bedrooms	features
100	10	1	[100., 10., 1.]
200	200	4	[200., 200., 4.]
400	70	1	[400., 70., 1.]
100	120	3	[100., 300., 3.]
200	300	3	[200., 300., 3.]
…	…	…	…
500	100	10	[500., 100., 10.]

Unfortunately for us, the construction the `features` column is not performed automatically under the hood by Spark, like when doing statistics in R. On the contrary, we have to construct the column explicitly. The `VectorAssembler()` constructor is here for that:

```
from pyspark.ml.feature import VectorAssembler

vectorizer = VectorAssembler(
    inputCols     = ["price", "beds", "bedrooms"], # the columns we want to put in the feature
    outputCol     = "features",                    # the name of the column ("features")
    handleInvalid = 'skip'                         # skip rows with missing / invalid values
)

listings_vec = vectorizer.transform(listings)

# Reminders:
# Spark data sets are immutable: a copy is returned, and the original is unchanged.
# Spark operations are lazy: listings_vec just contains the recipe for building vector column
# but no item of the column is computed unless explicitly asked to.

listings_vec.show(5) # The first 5 values of the features column are computed.
```

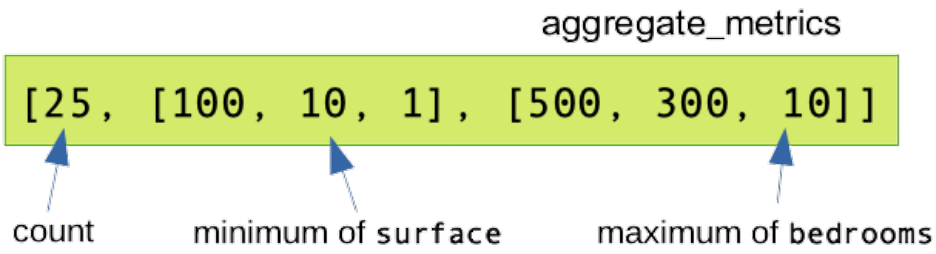
Step 2: summarization. Now that we have a vector column, we can use a `Summarizer` object to declare all the statistics we want to compute, in a similar fashion than with the Spark SQL `summary()` method. The following statistics are known: mean*, sum*, variance*, standard-deviation*, count*, number of non-zero entries, maximum*, minimum*, L2-norm, L1-norm, as can be read in the documentation. *(Stars (*) denote statistics that could also be computed with the `summary()` method. Approximate quartiles are not computed.)* Summarizers are created with the `Summarizer.metrics()` constructor. Here again, you are incited to declare all the summaries at once, so that they can all be computed in one pass:

```
from pyspark.ml.stat    import Summarizer

summarizer = Summarizer.metrics("count", "min", "max")

listings_vec.select( summarizer.summary(listings_vec.features), ).show(truncate=False)
# By default, the output of columns is capped to a maximum width.
# truncate=False prevents this behaviour.
```

This produces the output:



1.5.2 3.2 Hands-on 3

☐ Is `listings.summary()` slower to run than `listings.summary("count", "min", "max")` ? Why?

– You can measure time in Python with this simple template:

```
from timeit import default_timer as t
start = t()
# the thing you want to measure
print("Time:", t()-start)
```

[26]:
```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.stat import Summarizer
from timeit import default_timer as t

start = t()
listings.summary()
print("Time for listings.summary():", t()-start)

start = t()
listings.summary("count", "min", "max")
print("Time for listings.summary('count', 'min', 'max'):", t()-start)
```

```
Time for listings.summary(): 0.11081740446388721
Time for listings.summary('count', 'min', 'max'): 0.03570912592113018
```

☐ Compute the average number of beds per property in Barcelona in four different ways:

– Which method is the fastest?

1. directly with the Spark SQL mean function,
2. using `summary()`,
3. using a `Sumarizer` object
4. locally after you collected the bed columns. *Despite the operation being very common, Spark does **not** provide a simple syntax to collect a column as a local array. A work-around is to use the Pandas package and the `toPandas()` method (documentation). First install Pandas with `!pip install pandas`. Then you can collect a local copy of a dataframe called `df` with: `df_local = df.toPandas()`. A Pandas data frame possesses a `mean()` method, that compute the mean of each column of the data frame: more details are in Pandas' documentation.*

[29]:
```
# Spark SQL mean function
from pyspark.sql.functions import count, countDistinct, approx_count_distinct,␣
 ↪min, max, avg, first, last, sum, sumDistinct

start = t()
listings.select(avg("beds")).show()
print("Time for sql:", t()-start)
```

```
+----------------+
|       avg(beds)|
+----------------+
|2.01415838460325|
+----------------+

Time for sql: 0.20935234054923058
```

[30]:
```
start = t()
listings.summary().show()
print("Time for listings.summary():", t()-start)
```

```
[Stage 21:==============================>                          (1 + 1) / 2]

+-------+------------------+----------------+------------------+-------------
----+---------+
|summary|                id|            beds|          bedrooms|
price|     city|
+-------+------------------+----------------+------------------+-------------
----+---------+
|  count|             76068|           76068|             76068|
76068|    76068|
|   mean|2.796842090852921E7| 2.01415838460325|1.4599568806857022|
139.525464058474|     NULL|
| stddev|1.681550461898421E7|1.560729197626797|0.8792732459703063|359.7987457459
5436|     NULL|
|    min|           10000070|               1|                 1|
8.0|barcelona|
|    25%|        1.3080002E7|               1|                 1|
```

```
55.0|     NULL|
|    50%|          2.83904E7|                1|                1|
85.0|     NULL|
|    75%|        4.3178622E7|                2|                2|
145.0|     NULL|
|    max|             999984|               50|               40|
21000.0| salem-or|
+-------+-------------------+-----------------+-----------------+-------------
----+---------+

Time for listings.summary(): 3.0585638117045164
```

[34]:
```
vectorizer = VectorAssembler(
    inputCols     = ["beds"], # the columns we want to put in the features␣
 ↪column
    outputCol     = "features",                    # the name of the column␣
 ↪("features")
    handleInvalid = 'skip'                         # skip rows with missing /␣
 ↪invalid values
)

listings_vec = vectorizer.transform(listings)

start = t()
summarizer = Summarizer.metrics("mean")
listings_vec.select(summarizer.summary(listings_vec.features),).
 ↪show(truncate=False)
print("Time for summarizer):", t()-start)
```

```
[Stage 24:==============================>                          (1 + 1) / 2]

+--------------------------------+
|aggregate_metrics(features, 1.0)|
+--------------------------------+
|{[2.01415838460327]}            |
+--------------------------------+

Time for summarizer): 1.8343553245067596
```

[41]:
```
#!pip install pandas
import pandas as pd

listings_loc = listings.toPandas()

start = t()
```

```
print(listings_loc["beds"].mean())
print("Time for pandas):", t()-start)
```

```
2.01415838460325
Time for pandas): 0.0006632190197706223
```

The most simple model is often surprisingly difficult to beat!

- ☐ Compute the mean price on the data set as a predictor for an AirBnB listing's price and the total sum of squares. (We will elaborate in the next section.)

[44]:
```
print(listings_loc["price"].mean())

print(m_r_1(listings_loc["price"]))
```

```
139.52547
11328106709.0
```

1.6 4 Regression with Spark ML

A better way to predict prices is to build a regression mode, which in Spark falls under the broad category of machine-learning problems. Regressions thus belong the the `ml` module, often called Spark ML, like the summarizer that we saw just before.

There is an old module called `mllib` that is also called "Spark ML". That can cause confusion.

The `ml` module is built in a distinctive fashion than the rest of Spark. **Firstly** we have seen with `Summarizer` that we can not readily use the columns and that instead **columns have to be first converted to a `Vector` format** with the `VectorAssembler` function.

Secondly, we need to distinguish between two different types of object classes: transformers and estimators classes. **Transformers** are a class of objects representing any process that modifies the dataset, and returns the modified version. It has a **transform()** method. **Estimators** on the other hand are classes of objects representing any process that produces a transformer based on some computed parameters from the data set. It has a **`fit()`** method. It is easier with an example. In the following example, `regressor` is an estimator, and we compute the regression coefficients with the `fit()` method. This produces `model`, the regression model itself, which is of class transformer. Indeed, we can use its `transform()` method to add predictions to the initial dataset.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression

vectorizer = VectorAssembler( # copy-pasted from previous section...
    inputCols     = ["beds", "bedrooms"], # ... but without price
    outputCol     = "features",
    handleInvalid = 'skip'
)

listings_vec = vectorizer.transform(listings)

regressor = LinearRegression(featuresCol="features", labelCol="price")
```

```
model     = regressor.fit(listings_vec)

model.coefficients
model.intercept

listings_pred = model.transform(listings_vec)
listings_pred.show() # model and predictions from the regression
```

Vocabulary

The machine-learning community leaves at the border between computer science and mathematics. They borrow vocabulary from both sides, and it can sometimes be confusing when reading software documentation. Spark's `lib` module uses conventions from this community :

- **label**, rather than "independent variable". This comes from the fact that historically, machine-learning has originated from problems such as image labeling (for instance digit recognition). Even for continuous variables, machine-learners may use "label"
- **features**, rather than "dependent variables" ; the number of features is often dubbed d like dimension (instead of p in statistics)
- machine-learners don't use the word "observation" or "unit" and prefer **row**

Pipelines

If you come to repeat several times the same series of transformations, you may take advantage of the pipeline objects. A **pipeline** is just a collections of steps applied to the same dataset. This helpful when you:

- repeat the same analysis for different regions / periods
- want to control predictions on a new, unseen test set, and ant to apply exactly the same process

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline

vectorizer = VectorAssembler( # same vectorizer as before
    inputCols     = ["beds", "bedrooms"],
    outputCol     = "features",
    handleInvalid = 'skip'
)
regressor = LinearRegression(featuresCol="features", labelCol="price") # same regressor
pipeline  = Pipeline(stages = [vectorizer, regressor]) # ... but now we pack them into a pipel

listings_beij = listings.filter(listings.city=="Beijing")
listings_barc = listings.filter(listings.city=="Barcelona")

model_beij = pipeline.fit(listings_beij) # vectorizer AND regressor are applied
model_barc = pipeline.fit(listings_barc)

print(model_beij.stages[1].coefficients) # model.stages[0] is the first step, model.stages[1]
print(model_beij.stages[1].intercept)
```

```
print(model_barc.stages[1].coefficients)
print(model_barc.stages[1].intercept)
```

1.6.1 4.1 Hands-on 4

☐ Interpret the results of the general regression.

[50]:
```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression

vectorizer = VectorAssembler( # copy-pasted from previous section...
    inputCols     = ["beds", "bedrooms"], # ... but without price
    outputCol     = "features",
    handleInvalid = 'skip'
)

listings_vec = vectorizer.transform(listings)

regressor = LinearRegression(featuresCol="features", labelCol="price")
model     = regressor.fit(listings_vec)

print(model.coefficients)
print(model.intercept)

listings_pred = model.transform(listings_vec)
listings_pred.show() # model and predictions from the regression
```

```
[4.605615925032313,56.5075583633869]
47.7504254834323
+-----+----+--------+-----+-------------+----------+---------+------------------
+
|   id|beds|bedrooms|price|         city|      time| features|
prediction|
+-----+----+--------+-----+-------------+----------+---------+------------------
+
| 3831|   3|       1| 75.0|new-york-
city|2021-12-05|[3.0,1.0]|118.07483162191613|
| 5121|   1|       1| 60.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
| 5136|   2|       2|275.0|new-york-city|2021-12-05|[2.0,2.0]|
169.9767740602707|
| 5178|   1|       1| 68.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
| 5203|   1|       1| 75.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
| 6872|   1|       1| 65.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
```

```
| 6990|   1|       1| 62.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
| 7064|   1|       1| 90.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
| 7097|   2|       1|199.0|new-york-
city|2021-12-05|[2.0,1.0]|113.46921569688382|
| 7750|   2|       2| 96.0|new-york-city|2021-12-05|[2.0,2.0]|
169.9767740602707|
| 8490|   4|       1|140.0|new-york-
city|2021-12-05|[4.0,1.0]|122.68044754694844|
| 9657|   1|       1|175.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
| 9704|   1|       1| 55.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
|10452|   2|       1| 82.0|new-york-
city|2021-12-05|[2.0,1.0]|113.46921569688382|
|11943|   2|       1|150.0|new-york-
city|2021-12-05|[2.0,1.0]|113.46921569688382|
|12192|   1|       1| 40.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
|12343|   1|       1|250.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
|12937|   2|       1|145.0|new-york-
city|2021-12-05|[2.0,1.0]|113.46921569688382|
|12940|   1|       1| 99.0|new-york-
city|2021-12-05|[1.0,1.0]|108.86359977185151|
|13121|   2|       1| 75.0|new-york-
city|2021-12-05|[2.0,1.0]|113.46921569688382|
+-----+----+--------+-----+------------+----------+---------+-----------------
+
only showing top 20 rows
```

☐ Collect the model's R^2. How good is our model?

– Models have a `summary` property, that you can explore with `dir(model.summary)`.

[55]:
```
model.summary.r2
```

[55]:
```
0.023645470854636197
```

☐ Repeat the estimation separately for barcelona, brussels and rome.

☐ Are the coefficients stable? *You will build a pipeline object.*

[67]:
```
listings.createOrReplaceTempView("View_listings")

spark.sql("""
SELECT DISTINCT(city) FROM View_listings
""").show()
```

```
listings\
  .select("city")\
  .distinct()\
  .show()
```

```
+-------------+
|         city|
+-------------+
|new-york-city|
|         rome|
|    barcelona|
|     brussels|
|         riga|
|     salem-or|
+-------------+

+-------------+
|         city|
+-------------+
|     brussels|
|         riga|
|     salem-or|
|new-york-city|
|         rome|
|    barcelona|
+-------------+
```

[61]:
```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
from pyspark.ml import Pipeline

vectorizer = VectorAssembler( # same vectorizer as before
    inputCols     = ["beds", "bedrooms"],
    outputCol     = "features",
    handleInvalid = 'skip'
)
regressor = LinearRegression(featuresCol="features", labelCol="price") # same␣
 ↪regressor
pipeline  = Pipeline(stages = [vectorizer, regressor]) # ... but now we pack␣
 ↪them into a pipeline

listings_bruss = listings.filter(listings.city=="brussels")
listings_barc = listings.filter(listings.city=="barcelona")
listings_rome = listings.filter(listings.city=="rome")
```

```
model_bruss = pipeline.fit(listings_bruss) # vectorizer AND regressor are␣
 ↪applied
model_barc = pipeline.fit(listings_barc)
model_rome = pipeline.fit(listings_rome)

print(model_bruss.stages[1].coefficients) # model.stages[0] is the first step,␣
 ↪model.stages[1] the second...
print(model_bruss.stages[1].intercept)

print(model_barc.stages[1].coefficients)
print(model_barc.stages[1].intercept)

print(model_rome.stages[1].coefficients)
print(model_rome.stages[1].intercept)

print("R2 for rome: ",model_rome.stages[1].summary.r2,"\n","R2 for barcelona:␣
 ↪", model_barc.stages[1].summary.r2, "\n", "R2 for brussels: ", model_bruss.
 ↪stages[1].summary.r2)
```

```
[22.01332271072145,20.109348232104463]
25.612245016714546
[11.699388065651194,27.071195447324577]
26.142890673422187
[1.2705666265413038,60.521987958586976]
37.62605657020124
R2 for rome:  0.012507860920676128
 R2 for barcelona:  0.046266345561894906
 R2 for brussels:  0.10868833727612714
```

☐ Are the `fit()` and `transform()` methods called eagerly or lazily?

☐ Check the execution plan with the `explain()` method for lazy evaluations.

[66]:
```
listings.explain()
```

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=true
+- == Final Plan ==
   TableCacheQueryStage 0
   +- InMemoryTableScan [id#0, beds#150, bedrooms#227, price#380, city#74,
time#303]
         +- InMemoryRelation [id#0, beds#150, bedrooms#227, price#380, city#74,
time#303], StorageLevel(disk, memory, deserialized, 1 replicas)
               +- AdaptiveSparkPlan isFinalPlan=false
                  +- Project [id#0, cast(beds#37 as int) AS beds#150,
cast(bedrooms#36 as int) AS bedrooms#227, cast(regexp_replace(price#39, [$\,], ,
1) as float) AS price#380, city#74, cast(last_scraped#3 as date) AS time#303]
                     +- Filter atleastnnonnulls(6, id#0, cast(beds#37 as int),
cast(bedrooms#36 as int), cast(regexp_replace(price#39, [$\,], , 1) as float),
```

```
city#74, cast(last_scraped#3 as date))
                        +- InMemoryTableScan [bedrooms#36, beds#37, city#74,
id#0, last_scraped#3, price#39], [atleastnnonnulls(6, id#0, cast(beds#37 as
int), cast(bedrooms#36 as int), cast(regexp_replace(price#39, [$\,], , 1) as
float), city#74, cast(last_scraped#3 as date))]
                              +- InMemoryRelation [id#0, listing_url#1,
scrape_id#2, last_scraped#3, name#4, description#5, neighborhood_overview#6,
picture_url#7, host_id#8, host_url#9, host_name#10, host_since#11,
host_location#12, host_about#13, host_response_time#14, host_response_rate#15,
host_acceptance_rate#16, host_is_superhost#17, host_thumbnail_url#18,
host_picture_url#19, host_neighbourhood#20, host_listings_count#21,
host_total_listings_count#22, host_verifications#23, … 51 more fields],
StorageLevel(disk, memory, deserialized, 1 replicas)
                                    +- *(1) ColumnarToRow
                                       +- FileScan parquet [id#0,listing_url#1,s
crape_id#2,last_scraped#3,name#4,description#5,neighborhood_overview#6,picture_u
rl#7,host_id#8,host_url#9,host_name#10,host_since#11,host_location#12,host_about
#13,host_response_time#14,host_response_rate#15,host_acceptance_rate#16,host_is_
superhost#17,host_thumbnail_url#18,host_picture_url#19,host_neighbourhood#20,hos
t_listings_count#21,host_total_listings_count#22,host_verifications#23,… 51
more fields] Batched: true, DataFilters: [], Format: Parquet, Location:
InMemoryFileIndex(1 paths)[s3a://ludo2ne/diffusion/ensai/airbnb.parquet],
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,listing_ur
l:string,scrape_id:string,last_scraped:string,name:string,description:…
+- == Initial Plan ==
   InMemoryTableScan [id#0, beds#150, bedrooms#227, price#380, city#74,
time#303]
      +- InMemoryRelation [id#0, beds#150, bedrooms#227, price#380, city#74,
time#303], StorageLevel(disk, memory, deserialized, 1 replicas)
            +- AdaptiveSparkPlan isFinalPlan=false
               +- Project [id#0, cast(beds#37 as int) AS beds#150,
cast(bedrooms#36 as int) AS bedrooms#227, cast(regexp_replace(price#39, [$\,], ,
1) as float) AS price#380, city#74, cast(last_scraped#3 as date) AS time#303]
                  +- Filter atleastnnonnulls(6, id#0, cast(beds#37 as int),
cast(bedrooms#36 as int), cast(regexp_replace(price#39, [$\,], , 1) as float),
city#74, cast(last_scraped#3 as date))
                     +- InMemoryTableScan [bedrooms#36, beds#37, city#74, id#0,
last_scraped#3, price#39], [atleastnnonnulls(6, id#0, cast(beds#37 as int),
cast(bedrooms#36 as int), cast(regexp_replace(price#39, [$\,], , 1) as float),
city#74, cast(last_scraped#3 as date))]
                           +- InMemoryRelation [id#0, listing_url#1,
scrape_id#2, last_scraped#3, name#4, description#5, neighborhood_overview#6,
picture_url#7, host_id#8, host_url#9, host_name#10, host_since#11,
host_location#12, host_about#13, host_response_time#14, host_response_rate#15,
host_acceptance_rate#16, host_is_superhost#17, host_thumbnail_url#18,
host_picture_url#19, host_neighbourhood#20, host_listings_count#21,
host_total_listings_count#22, host_verifications#23, … 51 more fields],
StorageLevel(disk, memory, deserialized, 1 replicas)
```

```
                                  +- *(1) ColumnarToRow
                                     +- FileScan parquet [id#0,listing_url#1,scra
pe_id#2,last_scraped#3,name#4,description#5,neighborhood_overview#6,picture_url#
7,host_id#8,host_url#9,host_name#10,host_since#11,host_location#12,host_about#13
,host_response_time#14,host_response_rate#15,host_acceptance_rate#16,host_is_sup
erhost#17,host_thumbnail_url#18,host_picture_url#19,host_neighbourhood#20,host_l
istings_count#21,host_total_listings_count#22,host_verifications#23,… 51 more
fields] Batched: true, DataFilters: [], Format: Parquet, Location:
InMemoryFileIndex(1 paths)[s3a://ludo2ne/diffusion/ensai/airbnb.parquet],
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<id:string,listing_ur
l:string,scrape_id:string,last_scraped:string,name:string,description:…
```

1.7 5 Diving deeper

You are in autonomy for this section. You will find helpful:

- The general Spark documentation for the ml module
- The PySpark documentation

1.7.1 5.1 Hands-on 5

☐ Add a categorical variable to the regression.

[84]:
```
from pyspark.ml.feature import StringIndexer
indexer = StringIndexer(inputCol="city", outputCol="cityIndex",␣
 ↪handleInvalid="skip")
vectorizer = VectorAssembler(
    inputCols=["beds", "bedrooms", "cityIndex"],
    outputCol="features",
    handleInvalid='skip'
)

vec = indexer.fit(listings).transform(listings)
list_vec = vectorizer.transform(vec)
regressor = LinearRegression(featuresCol = "features", labelCol = "price")
model = regressor.fit(list_vec)

print(model.coefficients, model.intercept)
print(model.summary.r2)
```

```
[8.461916981667141,55.79201119050593,-33.975704594548894] 73.75655267109914
0.0331702834936336
```

☐ Compute the p-values of your model as well as confidence intervals for the predictions.

[87]:
```
print(model.summary.pValues)
```

```
std = model.summary.coefficientStandardErrors
coeff = model.coefficients

conf_int = []

for i in range(len(coeff)):
    inte = [coeff[i] - 1.96*std[i], coeff[i] + 1.96*std[i]]
    conf_int.append(inte)

print("95% Confidence interval for the beds variable estimator: ", conf_int[0],␣
 ↪"\n",
      "95% Confidence interval for the bedrooms variable estimator: ",␣
 ↪conf_int[1], "\n",
      "95% Confidence interval for the city variable estimator: ", conf_int[2])
```

```
[2.3127499915176486e-11, 0.0, 0.0, 0.0]
95% Confidence interval for the beds variable estimator:  [5.981226462725395,
10.942607500608887]
 95% Confidence interval for the bedrooms variable estimator:
[51.41579178282355, 60.168230598188316]
 95% Confidence interval for the city variable estimator:  [-36.40836640740081,
-31.54304278169698]
```

☐ Time the regression in different settings and report the results on this shared spreadsheet. How does it scale with the number of listings (n) ? the number of regressors (p) ? the number of nodes in your cluster (k) ? *You will only try a couple of configurations that have not been tested by others. Remember that you can order and revoke nodes from your cluster at any time from the AWS's cluster view, in the hardware tab, on on the CORE line, "resize".*

[28]:
```
# Code here
```

☐ Down-sample your data set to $n = 100000$, while still keeping a few variables. Save it on S3, then download it on your computer. Run the regression locally on your computer in R. In your opinion, is the extra precision (in term of R^2) is worth the extra computation time?

[29]:
```
# Code here
```

1.8 End of the Lab

☐ Export your notebook
- Right click and Download (.ipynb)
- File > Save and Export Notebook > HTML

1.8.1 SSPCloud

☐ Delete the Jupyter-pyspark service
- SSPCloud > My services > Delete

1.8.2 AWS

- ☐ Terminate your cluster
  - On *EMR* service page, click on *Clusters*
  - Select the active cluster and click on `Terminate`

Solution

Solution