Practical Manual for Data Analysis & Programming for Operations Management

Part B - Optimization, big data tools, and data-analysis

2022-09-04

W. van Wezel, N.D. Van Foreest

University of Groningen, Department of Operations

Table of Contents

1. INTRODUCTION

This document contains the material for the practical work related to optimization and data-analysis of Operations Management problems with Python. Each chapter is roughly mapping one week of the work planned. After week 5, you shall be familiar with the topics discussed, and acquired the practical skills to be able to start the individual assignment.

If you employ Python for optimization or data analysis, you will use external tools for doing number crunching. Usually, this is mostly transparent to you because developers of those external tools provide user friendly ways to use them. In this course, we will use Gurobi for optimization, and Elasticsearch for data analysis. The example code in this manual shows the basic commands to use those tools, **but we rely on you to search on the web for documentation and examples to get a real understanding**. For instance, a search on `Python lp examples` will provide you with lots of useful websites that provide code examples on how to use python to solve LP problems. In fact, by reading (thoroughly!) code examples of others, we learned to program in Python ourselves. First, we looked for simple examples, and then we read harder examples, and so on.

At the end of each week, we will make our solution code available, so that you can compare your solution with ours. Usually, there are multiple ways to solve a problem, so don't worry if your solution is different from ours. However, please **try really hard** to complete the practical work yourself before looking at our code.

2. A Linear Programming Exercise: product-mix optimization

2.1 Get a working python environment with Gurobi

For the practicals, you will have to install a working Python environment and the mathematical optimization solver from Gurobi, which is perhaps the best LP optimization library available, used by nearly every multi-national with optimization problems such as KLM, Schiphol, IBM, and so on). See https://www.gurobi.com/. How to get the Python environment is explained in the manual for the a-practicals (generic programming skills). So here, we assume you have a Python and Pycharm installed on your computer. To install Gurobi and link it to Python, follow these steps:

1. Download Gurobi on https://www.gurobi.com/. On the top of the screen, click 'downloads & licenses', and then click on 'download center'.
2. In the download center, click on 'Gurobi Optimizer'
3. Accept the license
4. Then download the appropriate version of the latest release of the Gurobi Optimizer (this depends whether you are on Windows, MacOs, or linux).
5. You can then install Gurobi.
6. Now go back to the download center, and, in the 'request a license' section, click on 'Academic License'.
7. You need to register an account. Make sure to use your university email address; they might require this to verify you are indeed a student from academia.
8. When you have registered, you can generate a license code.
9. When generated, you can activate this license code on your computer:
    a. Open a command prompt (Windows: click the windows button, type cmd, and press enter; on MacOs: open a terminal).
    b. Copy the command provided by the Gurobi website and press enter. The command should look like this (the numbers will be different for you):
    ```
    grbgetkey 123456-abcde-321232-6456-2342432432
    ```

Note that the license must be activated while connected through the university network.

If you don't get error messages, Gurobi is now installed. We now must arrange for Python to be able to use Gurobi. This requires the installation of the Gurobi Python libraries. The easiest way to do this is start a command prompt or terminal again, and go to the directory where Gurobi is installed. For me this was C:\gurobi903\win64. So after opening the command prompt, I type `cd C:\gurobi903\win64`, and then press enter. This folder should contain a file called `setup.py`. On your command prompt, now type `python setup.py install`. This will install the Gurobi libraries in Python. On a Mac, it might be necessary to specify the Python version, e.g., `python3.8 setup.py install`.

2.2 Running the product mix code

1. Download product_mix.py from Brightspace; the code is also available in Table 1. This code implements a linear program (LP), see Factory Physics Ch 16 for further explanation.
2. Run this code in Pycharm.
3. Read and think about the output.

2.3 Breaking the code

It is extremely useful to become familiar with python's error message. When you are coding, you'll often make mistakes (this is entirely normal; it happens to us all the time). Python's error messages can be enormously helpful in discovering what you did wrong. For this reason, we are going to break our working code, and see what type of errors we get. Read the error trace carefully; for instance, check the line number where the error occurs, and check the error type.

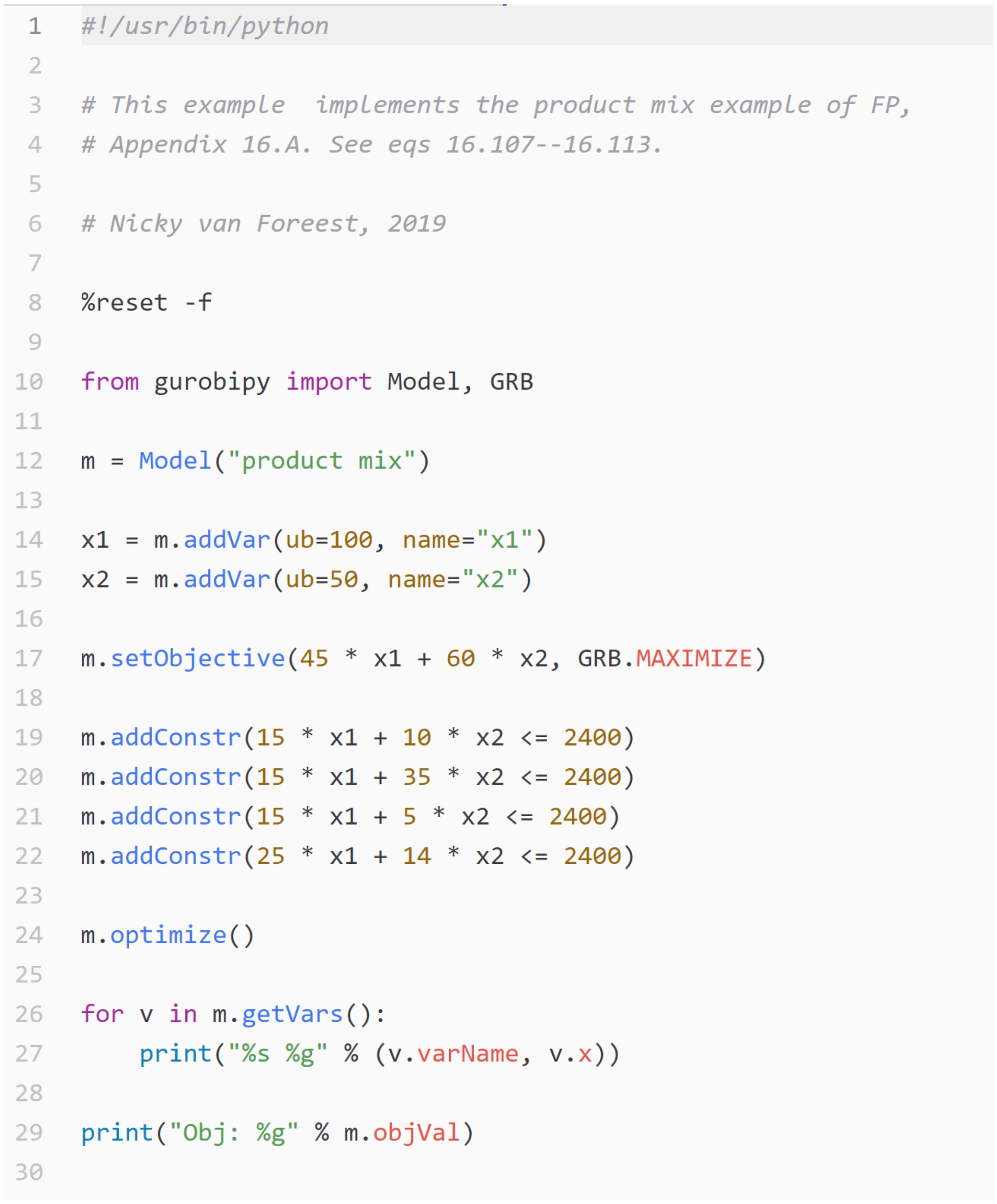
Here are some suggestions to break the code. Do them all, and then try your own mistakes. Read the error message very carefully. Understanding the error messages can save you an enormous amount of time (not just minutes or hours, but days!) After introducing the error, don't forget to repair, i.e., undo, it.

1. Comment Line 10, see Table 1, in your code; that is, put a # in front of the line. (Mind that python is sensitive to indentation, hence, the # should be the very first

character on the line, there should be NOTHING in front of it). What is the error message? Remove the comment sign again.

2. Now comment out Line 12 (and uncomment it after having read the error message).
3. Line 12, put a space in front of the first word, so that everything moves one step to the right. Check the error message, and explain it.
4. Line 12, what would happen if you would put two spaces at the front of the line?
5. Line 12, what would happen if you would put a tab character at the front of the line?
6. Comment out Line 14.
7. In Line 14, change addVar to addvar. (Just change the capital 'V' to 'v').
8. In Line 14, change x1 to x.
9. Comment out Line 17.
10. Comment out Line 19.
11. Comment out Line 21.
12. Line 21, remove the number 2400 so that the line reads like: `m.addConstr(15 * x1 + 10 * x2 <= )`
13. Comment out Line 26.
14. Comment out Line 29. What happens? Explain it.
15. Line 26: replace the line by `for v in:`
16. Introduce your own errors, one by one, and repair again. The more errors you can invent, the better. Once again, reading and understanding python's error messages will save you lots and lots of time, and frustration.

Table 1: product_mix.py



```python
#!/usr/bin/python

# This example  implements the product mix example of FP,
# Appendix 16.A. See eqs 16.107--16.113.

# Nicky van Foreest, 2019

%reset -f

from gurobipy import Model, GRB

m = Model("product mix")

x1 = m.addVar(ub=100, name="x1")
x2 = m.addVar(ub=50, name="x2")

m.setObjective(45 * x1 + 60 * x2, GRB.MAXIMIZE)

m.addConstr(15 * x1 + 10 * x2 <= 2400)
m.addConstr(15 * x1 + 35 * x2 <= 2400)
m.addConstr(15 * x1 + 5 * x2 <= 2400)
m.addConstr(25 * x1 + 14 * x2 <= 2400)

m.optimize()

for v in m.getVars():
    print("%s %g" % (v.varName, v.x))

print("Obj: %g" % m.objVal)

```

2.4 Integer variables, get familiar with the Gurobi documentation.

In the solution you must have seen that the optimal number of products to be produces is a fraction. Check the Gurobi documentation how to enforce that x1 and x2 are integer. (Just search the web. . .) Hint, google for `gurobi addvar integer`.

You should know that optimization problems whose variables are constrained to integers are typically much, much more complicated than problems all of whose variables are continuous. If discrete variables are not necessary, do not include the GRB.INTEGER property! Including this property in the problem specification for Gurobi, i.e., including the integer property in the code, can easily change the computation time from seconds to years for industry size problem. To understand the problem, suppose that the optimal value of x1 is either 75 or 76, and for x2 it is either 35 or 36. To find the optimal integer solution we need to test $2 \times 2 = 2^2 = 4$ alternatives. If we have 100 decisions variables, we need to test about 2^{100} possibilities; this is a huge number, impossible to test in any reasonable amount of time. So be aware!

2.5 Finding the constraints

Ensure that your code is equal to the original code again, so that you have a correctly working example. Let Gurobi solve the problem. Then you should notice that not all demand is covered. By changing the production system it must be possible to increase sales. But. . . , which machine to change, and what to do? In the next couple of exercises we will tackle this problem.

You should observe, and remember, from these problems how easy the numerical analysis becomes with our Python/Gurobi environment. We generate some ideas to extend production capacity. To see the effect of these suggestions, we have to change some numbers in our code, but that is very easy. Then we let Gurobi solve the problem, and we read out the results, and compare the effect on the total revenue. In other words, we (as humans) suggest plans on how to improve things, we let the computer carry out all the boring computations, and then we (as humans) interpret the results.

Now, our plan is to add capacity. But which machine(s) should we update? Which is the best to invest in?

A simple (but rather dumb) way to find out the machine that is the most constraining is as follows. We remove a machine from the set of constraints in lines 16–19, and just check whether the output (i.e., the value of the objective) increases.

1. First comment Line 19, and let Gurobi solve the problem. Check what happens to the solution.
2. Now include Line 19 again, but comment Line 20. Let Gurobi solve the problem and check the solution.
3. And so on.

You should see that machines B and D have an impact on the revenue.

You should know that the above procedure is by far not the smartest. You can use Gurobi to identity the, so-called, active constraints. However, this requires more knowledge of Gurobi, which you can always acquire later.

2.6 Extra minutes for the constraining machines

The learning goal for this and the next session is that you understand how easy it is with computers to quantitatively analyze many different scenarios. As an example, suppose we can buy some overtime, let's say 100 minutes per week on any machine. We would like to know which machine is the best. The next couple of questions are meant to help you get started with asking and answering such questions.

1. Change the constraint on machine B to 2500. Then solve the problem again, with Gurobi. And check the result.
2. What happens if you just add 1000 minutes to machine B? Realize that some of this extra time might be unused. Then this is not-so-smart decision: we pay for personnel, but we are unable to exploit this extra capacity due to the other machines.
3. Undo the changes on machine B, but change machine D's time to 2500. Then solve the problem again, with Gurobi. And check the result.
4. Perhaps it it better to give 50 minutes to machine B and D. What is the result?
5. How many extra minutes are necessary to satisfy all demand? (You don't have to find a general procedure for this problem, just play a bit with the numbers to see the effect.)
6. Another idea is to move some of the total time of machine A to machine B. In other words, we train the personnel that operates machine A to help at machine B, thereby increasing the total available time on machine B (but reducing the

time of machine A). Let's assume that the operators of machine A are less fast when they help at machine B. For instance, if we remove 200 minutes from machine A then this can be converted to just 100 minutes at machine B. Should we consider training machine A's personnel to help at machine B? Finding out is easy: change the 2400 of machine A into 2300, and increase the 2400 to 2450 at machine B. Have Gurobi solve this and interpret the result.

7. Make a graph of the output as a function of the extra capacity added to machine B. For the present, it is ok to make this in Excel. Later you'll learn how to make this with Python tools.

We assume that by now you understand how easy scenario analysis becomes. You should realize that scenario analysis is one of the key skills of managers. Generate good ideas on what to change, analyze the effects of the changes, interpret the results, and implement the best change.

2.7 Moving work from a bottleneck to a non-bottleneck

Another way to shift work from machine B to machine A (or C) is to reduce the production time. The idea is to assume that part of the work of an item can be done at either machine.

1. (Don't forget to bring your code to a pristine state again.)
2. What is the impact of increasing the production time of product 1 at machine A from 15 to 20 minutes, but thereby reducing the production time at station B from 15 to 10 minutes?
3. Should we move time for product 1 to station A or station C?
4. or should we move production time of product 2?
5. Should we move time from station D to station C? Which product?
6. How much time should we try to move? This question is particularly interesting. Suppose we would be able to redesign product 1 so that we can move all its production time at machine B to machine A, so that machine B can be bypassed altogether. Would it be useful to investigate such a design? (If such a redesign does not result in much extra output, it's useless to try. So, with our computations we can first analyze the effect on logistics and revenue before we start thinking about how to do the redesign.)

2.8 Testing

Finally, we need to check whether our model is correct. Note that for a real case it is essential to start with testing your model and implementation. In this practical we did the scenario first, so we sinned a bit. The reason is to show you first how to use computational tools such as Python and Gurobi, and not bore you with testing. But remember, when you use tools to make decisions for real, then always test first. It's easy to make a typo (e.g., type in the wrong number), miss a constraint, and so on, and it is a bit painful when your multi-million decision turns out not to work because you typed in the wrong numbers. . .

1. Remove all machine constraints, by commenting them out, and check that only the demands form the constraints.
2. Include the machine constraints again, but now remove the machine constraints on the demands, by removing the upper bounds indicated by the key word 'ub' in the code. (Check Gurobi's documentation on how to remove the upper bound.) Use Figure FP 16.15 to see what would be the optimal solution for this case, and check that you get the same solution with gurobi.

2.9 Adding Machines

Yet another idea is to buy an extra machine that can take off some of the load of machine D, say. So, we are going to add a machine E to the end of the chain of machines.

1. Copy product_mix.py to product_mix_fifth_machine.py, and use the latter file to implement the changes.
2. Add to the LP a machine E with production times 13 and 7 for products 1 and 2, respectively. Machine E is available for 1500 minutes per week. The production times at station D change now to 12 and 10 respectively.
3. Analyze the quantitative effect of moving work from machine D to machine E. (I see an increase in revenue from 5575 to 6042 by adding a machine E.)

So, based on estimates of the cost of buying and operating such a machine E we can use our computer programs to make good decisions, i.e., whether it's worth the money or not.

2.10 Adding a third product

A marketeer suggests to add a third product to our product portfolio. The estimated production times are 10, 5, 18, 10 on the respective machines; the selling price is $120 and the cost of raw material including labor is $30. The demand is estimated to be 20 products per week. Should we include this product in our portfolio?

1. Copy product_mix.py to product_mix_third_product.py.
2. Add product 3 to the LP and analyze the effect, in particular on the total profit. Does the profit increase when you add product 3?
3. Due to contractual obligations we have to minimally produce 80 items of product 1 and 20 of product 2. Include this in the LP. (tip: in `addVar`, `ub` stands for upper bound. Check the impact.

2.11 Minimizing the capacity required

Here is an interesting challenge: What would be the minimal capacity per machine required to serve all demand? You can build this also as an LP. Here I leave the details to you. The solution becomes available at the end of the week. Tip: what should be your decision variables now?

3 Linear Programming Continued: code organization and inventory control

The code in Table 1 has some problems. It does not scale to large numbers of machines or products because each product and machine has to be included by 'hand'. Moreover, the data is hard-coded in the algorithm itself. It is a much better design pattern to separate data from the algorithm, and make the algorithm generic in the sense that it can scale up to many products or machines. That is what we are going to learn in Section 3.1.

In Section 3.2 we'll organize our code a bit more with functions. With this step we complete the product mix example.

Then, in Section 3.3, we will analyze an inventory control problem with LP. This is a really useful example; over the years students have been applying it at companies to control the production and inventories.

In the next practicals we will no longer take you by the hand, but just give you hints on how to work towards an interesting computer program.

3.1 Separating data from the formal problem specification

We continue extending the case of the previous practical.

1. Create a new file, product_mix_2.py, so that you don't mess up the program in product_mix.py (This one worked, so while developing something new, it is best not to break the stuff that works.)
2. Put demand data in a list, called array `D`. Still use the demand from Factory Physics, appendix 16A. The text files on Brightspace for this week are for 3.2.
3. Put profits in a list `P`.
4. Put the capacity constraints of the machines in a list `C`.

5. Put the production times in a list of lists, `PT`, such that, for instance, `PT[0][1]` corresponds to the production time of product 2 at machine A (recall that python starts counting at 0, hence we subtract 1 from the machine id and product id).
6. Next, we need to make decision variables. Search, for instance, on `gurobi multiple variables example`, for a method similar to `addVar()`, but one suited for more than one variable. Figure out how to make a single compound structure of decision variables via this Gurobi model method call.
7. We now need to add the constraints. A single constraint often refers to multiple decision variables and data elements. In our case, we need to multiply the decision variable with the processing time, and compare the result to the maximum machine capacity. In the previous version of the code, we manually added a constraint for each machine. Now that the machines and products are configurable we need to add the constraints in a loop. Conceptually, we would need something like:

```
for i in range(len(C)):
        m.addConstr(time_used_on_this_machine<=C[i])
```

   where the time used on the machine would be the multiplication of all products produced on that machine multiplied by the amount produced (which is our decision variable `x`). So, within a single constraint, we need to iterate over all products. This is where generators and the Python `sum` function can help us. For example, try the following code:

```
x = sum(2 * n for n in range(5))
print(x)
```

   The code within the `sum` function is called a generator. Track that this code is a shorter version of:

```
x = 0
for n in range(5):
    x = x + 2 * n
print(x)
```

(ps, if you expected the outcome of this code to be 30, please read `https://www.w3schools.com/python/ref_func_range.asp`)

This can also be used when adding constraints. It could look something like:

```
num_machines = len(C)
for i in range(num_machines):
    m.addConstr(sum(PT[i][j] * x[j] <= C[i] for j in range(num_products))
```

But this formulation contains an error. Repair the error, and modify it so that you can use it in your code.

8. You also need the sum function in your `setObjective` call. Complete your program until it runs. Then compare the outcome with your original product_mix.py. The results should match, otherwise there is a mistake somewhere.

3.2 Organize code in functions

The challenge here is to organize the code in a better way, that is, by means of functions. Functions are very useful for a number of reasons. The first is that function names (if properly chosen) act as documentation; a good function name describes what the code in the body does. The function also helps to hide complexity. A programmer just has to read the function name to guess what the function does; the programmer does not necessarily have to read all the code of the function. Finally, a function enables reuse; it is not necessary to copy the code time and again for nearly the same task but with different numbers: just call the function again with new arguments/parameters.

1. Make a new file product_mix_3.py.
2. Give this script the same functionality as the previous one, but now make a function `optimize` to which you pass the data arrays as argument. The output of the function should the optimal profit.
3. Run it to see that it works.

3.3 Inventory control

We will now implement the model of Section 16.2.1 of FP. As said, it is an interesting and practically useful case; students have been using it as part of their master thesis project to control the production and inventory levels at companies. You should realize that in industry problems, there can be like 1000 different product types, the holding costs and profits may depend on product type and time (e.g., discounts in summer time), the inventory positions run in the millions of items. Such cases are impossible to analyze with Excel; programming skills are absolutely necessary here[1].

1. Read FP Section 16.2.1 and the one paragraph intro of Section 16. We use the data of Section 16.2 to check our implementation.
2. Make a list with demand and capacities like so
   ```
   D = [0, 80, 100, 120, 140, 90, 140]
   C = [0, 100, 100, 100, 120, 120, 120]
   ```
   There is a reason for this that is slightly subtle. If you check FP, Eq 16.4, you'll see that to update the inventory level I_1 at time $t = 1$, you'll need the inventory level I_0 at time $t = 0$. Thus, in the implementation, the easiest is let all lists start at time 0 and put zeros where appropriate.
3. Add decision variables for X, I, and S. Note that they need to be 7 long, 6 for the number of periods t = 1,..., 6 , and 1 for period t = 0. Include in the definition of X, S and I that they should be non-negative, i.e., FP Eq. 16.5. Then include also that $X \leq C$ and $S \leq D$, i.e., Eqs 16.2 and 16.3.
4. Add constraints to ensure that $X[0] = 0$, $S[0] = 0$, and $I[0] = I_0$. In the specification of the problem in Section 16.2 $I_0 = 0$.
5. Finally add constraint 16.4 for every t with a for loop. Observe that we need 6 constraints in total. Observe also that if you do have to deal with weekly or daily demand data, for one year, than adding all constraints by typing is not particularly nice. You really need for loops here.

[1] In fact, as a general rule, the analysis of any realistic business problem cannot be carried out with Excel, or it is exceptionally hard (much, much harder than to write a python program and use libraries).

6. Finally, add objective Fp. Eq. 16.1; use `sum` or `quicksum` to make your life easy. Quicksum is a Gurobi specific version of sum, which is faster for optimization models. If you use it, make sure to also import the quicksum library.
7. Solve the problem, and compare your result with the output as shown in Figure 16.5.
8. Think a bit about how your Excel sheet would look like if you would have to deal with yearly data. You should see that it would not fit on screen; in fact, it would look extremely messy.[2]
9. Write a function to read demand and capacity data from the file `inv_control_data_1.txt`. (First open this file with, e.g., Pycharm or notepad. It is best to check the format of an input file before you write code to read it.) Each line contains the demand and the capacity for a specific period. The data is the same as earlier, so testing must be easy. Note that if you read the file as csv and convert it to a list, the individual items in the list will be a string. You need to tell Python explicitly that the values are numbers, for example, `int(record[0])` if you want the values to be rounded to a whole number or `float(record[0])` if the numbers could contain fractions (e.g., 3.8).
10. Then read the info from `inv_control_data_2.txt`; this represents one year of weekly data. Solve the optimization problem. Interpret the results. When, for instance, does inventory start to build up? Are there weeks that we have lost sales? If so, is r sufficiently large? Can you modify the capacity planning to maximize the profit? (This is interesting challenge, and in fact, the challenge in many master thesis projects.)
11. An interesting optional extension is to try to make a graph of the inventory as a function of time. Graphs can be made with the `matplotlib` library.

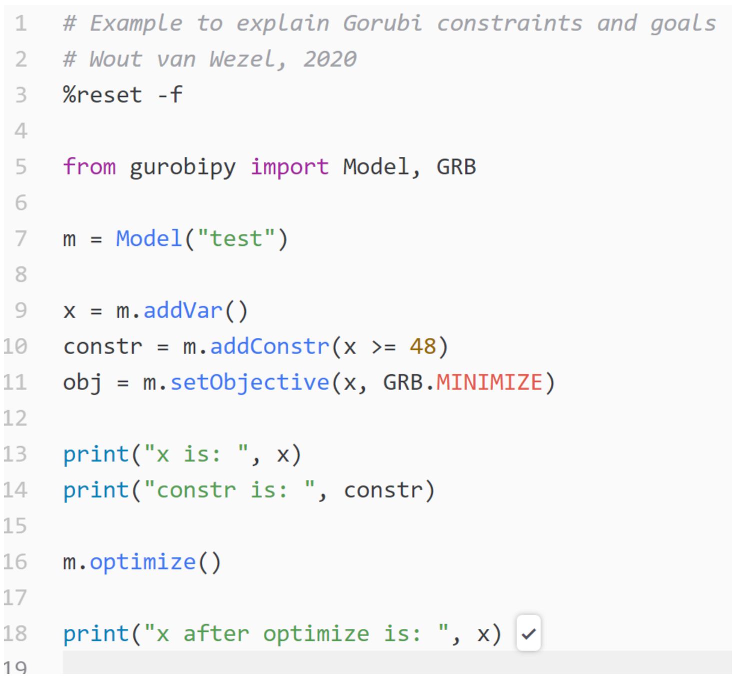
[2] As a general lesson again, Excel files are very hard to maintain and understand in comparison to (Python) programs. My code for this problem consists of some 15 lines, each of which is completely clear by itself.

3.4 *Appendix Practical 2B*

In this phase of the course, going through the mind-shift that is needed to learn programming is challenging. Some (or many) of you will have felt helpless at parts of Practical 2B. This is a struggle that you need to go through. Compare it to learning to drive a car. In your lessons, you don't sit next to your tutor while he/she demonstrates how to drive, with you watching. You would fail miserably during your driving exam. This is the same with programming. It not only is knowledge that you gain, but also a skill, and for that, you need to try and fail, before you understand and succeed.

The assignment of practical B in week 2 was difficult for most of you. Some were able to get a working model, but others might still not comprehend the solution that is posted on Brightspace. We assume that you understand how the demand, profit, capacity, and processing times are modelled as arrays in the code, and that you are able to create for loops to iterate through these arrays. If not, you are really falling behind. If you understand all that, but have trouble grasping the `addConstr` and `setObjective` in the code, please read this appendix carefully. We will explain stepwise how we came to our formulation.

First, let's look at a simple example of an optimization model:



```python
# Example to explain Gorubi constraints and goals
# Wout van Wezel, 2020
%reset -f

from gurobipy import Model, GRB

m = Model("test")

x = m.addVar()
constr = m.addConstr(x >= 48)
obj = m.setObjective(x, GRB.MINIMIZE)

print("x is: ", x)
print("constr is: ", constr)

m.optimize()

print("x after optimize is: ", x)
```

A model is created with one decision variable `x`. The goal is to minimize the value of `x`, with as constraint that it should be equal to, or larger than, 48. Hopefully, you already see the optimal value of `x` in this model: 48.

The powerful element here is that you provide Gurobi with the formula `x >= 48`, without knowing what `x` is at the moment that you specify the constraint. Essentially you tell Gurobi the following when you add a constraint: don't run this yet, but once you determine a value for `x`, please execute this piece of code to determine whether the constraint is violated for that value of `x`. When you run the above program, it will print `x` and `constr`, before running the model (line 13/14). Therefore, the first lines you will see in the output are:

```
x is:  <gurobi.Var *Awaiting Model Update*>
constr is:  <gurobi.Constr *Awaiting Model Update*>
```

This tells us that `x` is an object of the type `gurobi.Var`, `c` is of the type `gurobi.Constr`, and both have no value yet. Rather, they are waiting for the model to run.

So, what happens if we run the model (line 16)? Conceptually the following: Gurubi tries different values for `x`. Each time when it determines a new value for `x`, it runs the code for the constraint. If this code returns `false` for that value of `x`, it concludes that the constraint is violated. It will then try another value for `x`. If this code returns `true`, it concludes that the constraint is not violated. Then, it runs the code to calculate the objective (line 11, which in this case is real simple, because it just returns the value of `x` itself). It remembers the objective value, and then tries other values for `x`, and so on. In the end, it will give you back the `x` for which the object function was the lowest and no constraints were violated.

What is important here is that we apparently give a small program to Gurobi for each constraint and objective that we add to our model. Let's look at a somewhat more complex example. Suppose we have an array with the values 1,2,3, and 4. The constraint is that the sum of the decision variable multiplied by these values must be lower than or equal to 48. So, for `x=2` this would be `1*2 + 2*2 + 3*2 + 4*2 <= 48`, which becomes `20<=48`, which is `true`. So, with `x=2`, the constraint is not violated.

How can we add such a constraint in Gurobi? You should know by now how you can perform the needed calculation using a for loop:

```python
a=[1, 2, 3, 4]

x = 2
sum_of_a = 0
for i in range(len(a)):
   sum_of_a = sum_of_a + a[i] * x

print(sum_of_a)
```

Can we add such code to the constraints engine of Gurobi? Yes, if we first put it in a function, and if we call this function from `addConstr` or `setObjective`:

```python
# Example to explain Gorubi constraints and goals
# Wout van Wezel, 2020
%reset -f

from gurobipy import Model, GRB

def my_sum(arr, x):
  tot = 0
  for i in range(len(arr)):
    tot = tot + arr[i] * x
  return tot

arr = [1,2,3,4]
m = Model("test")
x = m.addVar()
constr = m.addConstr(my_sum(arr, x) >= 48)
obj = m.setObjective(x, GRB.MINIMIZE)

print("x is: ", x)
print("constr is: ", constr)

m.optimize()

print("x after optimize is: ", x)

```

or:

```python
# Example to explain Gorubi constraints and goals
# Wout van Wezel, 2020
%reset -f

from gurobipy import Model, GRB

def my_constraint(arr, x):
  tot = 0
  for i in range(len(arr)):
    tot = tot + arr[i] * x
  return tot >= 48

arr = [1,2,3,4]
m = Model("test")
x = m.addVar()
constr = m.addConstr(my_constraint(arr, x))
obj = m.setObjective(x, GRB.MINIMIZE)

print("x is: ", x)
print("constr is: ", constr)

m.optimize()

print("x after optimize is: ", x)

```

Please look for the differences of these two examples, and make sure that you understand that they are functionally the same.

In optimization models, we often use summation (i.e., when you see $\sum$ in such a model). In Python, we can use the `sum` function for this. In its most basic form, you call this function with an array, and then it will calculate the `sum` of the elements:

```python
print(sum([1,2,3,4]))
```

Another option is to use a generator. This is a Python construct where a loop can be sent to a function. Within this function, the loop is run. So, you can call the `sum` function with a for loop. The sum function recognizes that it did not get an array but a generator as parameter. It will 'run' the generator, and sum the outcome of the calculation as specified.

Here is an example of a generator within a `sum` function:

```
10  a=[1, 2, 3, 4]
11
12  x = 2
13  sum_of_a = 0
14
15  sum_of_a = sum(a[i] * x for i in range(len(a)))
16
17  print(sum_of_a)
```

At line 15, the `sum` function is called with a generator as parameter.

Using a generator in our example model, we get the following:

```
# Example to explain Gorubi constraints and goals
# Wout van Wezel, 2020
%reset -f

from gurobipy import Model, GRB

arr = [1,2,3,4]

m = Model("test")

x = m.addVar()
constr = m.addConstr(sum(x * arr[i] for i in range(len(arr))) >= 48)
obj = m.setObjective(x, GRB.MINIMIZE)

print("x is: ", x)
print("constr is: ", constr)

m.optimize()

print("x after optimize is: ", x)

```

This same formulation can be used in the product-mix problem. In the first version of the optimization model (in Practical 1B), data and model were mixed; the constraints were formulated as:

```
m.addConstr(15 * x1 + 10 * x2 <= 2400)
m.addConstr(15 * x1 + 35 * x2 <= 2400)
m.addConstr(15 * x1 + 5 * x2 <= 2400)
m.addConstr(25 * x1 + 14 * x2 <= 2400)
```

We need a constraint per machine, and within the constraint for each machine, we need to sum all task times and compare it the capacity for that machine. In the next, improved, version, the products and machines were configurable through arrays. So we create a loop where we add a constraint for each machine. Similar to the above examples, we can create a function in which we calculate the machine utilization. (Note that if Python encounters a `def` in the program, it does nothing yet. It will only do something with the code in the function, when the function is called from another place in the program).

```
def calc(PT, i, x, num_products):
    tot=0
    for j in range(num_products):
        tot=tot+PT[i][j] * x[j]
    return tot

for i in range(num_machines):
    m.addConstr(calc(PT, i, x, num_products) <= C[i])
```

By now, you should recognize that this can be shortened to:

```
def calc(PT, i, x, num_products):
    return sum(PT[i][j] * x[j] for j in range(num_products))

for i in range(num_machines):
    m.addConstr(calc(PT, i, x, num_products) <= C[i])
```

And finally to:

```
for i in range(num_machines):
    m.addConstr(sum(PT[i][j] * x[j] for j in range(num_products)) <= C[i])
```

From this, you should be able to track that the objective can be set as follows:

```
m.setObjective(sum(P[i]*x[i] for i in range(len(P))), GRB.MAXIMIZE)
```

There is no right or wrong in these different formulations. If you prefer to write calculations in regular for loops in functions, then that is fine. Just make sure that you use function names that explain what is being done there. So don't name the function `calc`. Instead, use a name like `machineUtilization`.

Still, in Gurobi, it is customary to use generators in the formulation of constraints and goals. The advantage is that it aligns nicely with the mathematical formulation of goals and constraints used in models. For example, consider the following objective from Factory Physics:

$$\text{Maximize} \qquad \sum_{t=1}^{\bar{t}} rS_t - hI_t \qquad (16.1)$$

Now compare this to generator based formulation in Python:

```python
m.setObjective(sum(r*S[t]-h*I[t] for t in range(1, len(D))), GRB.MAXIMIZE)
```

Hopefully you recognize the similarity. The correspondence between model and code makes it easier to implement, debug, and change mathematical models in Gurobi.

If you created a Gurobi model in Python, and it does not behave as expected, you can always look at the actual optimization model that is sent to the optimization engine. For example, try the following command with the array based product-mix example after you added constraints and goals:

```python
    m.write('c:/temp/model.lp')
```

This command writes the following LP optimization model to a file:

```
\ Model product mix
\ LP format - for model browsing. Use MPS
Maximize
  45 x[0] + 60 x[1]
Subject To
 R0: 15 x[0] + 10 x[1] <= 2400
 R1: 15 x[0] + 35 x[1] <= 2400
 R2: 15 x[0] + 5 x[1] <= 2400
 R3: 25 x[0] + 14 x[1] <= 2400
Bounds
 x[0] <= 100
 x[1] <= 50
End

```

This, of course, looks really similar to the very first version of our product-mix code where we did not yet use arrays. In this way, if you add constraints and goals through loops, and when you extend your arrays, you can track exactly what happens. So, using this file, you can debug your `addConstr` and `setObjective` calls.

4 Getting Operational with Location Data: make your own Strava Application

In this practical you'll build your own sort of Strava tool (if you don't know what it is, see https://www.strava.com/). Of course, it will not be as slick as the real Strava. However, the app we use will just need a few MB, rather than 100 MB or more for Strava; you don't get unwanted advertisements; you can process your own data, and, on top of this, you'll come to understand how data assembly of routing works.

Although Strava is not really useful in a business context, geo-location tools are very important in transportation companies. Not only for traditional transportation planning and routing, but even more so for new logistic business concepts such as Uber, thuisbezorgd.nl, and the Physical Internet. Now that gps data of all trucks and couriers can be shared real time with planners, such companies get their main competitive advantage from dynamic planning using real time locations of couriers and real time data of traffic congestion. Therefore, understanding and being able to work with geo-location data and tools is very important for a data scientist.

4.1 Computing track length and duration

In our first version of our tracking tool we'll use a library to read the data from a gpx file and we use a library to compute the distance between two gps coordinates. The goal is to build a computer program that computes the total length and duration of a track recorded by an app that runs on your mobile phone.

1. Install an app that can track your routes and can save the data in the standard gpx format. For my iphone I use the app 'Open GPX Tracker'; for Android there must be something similar.
2. Record a route and send it to yourself by mail. If you don't want to track your own route, use the gpx file available on Brightspace. Open the file in Pycharm and look at the structure. It is formatted as Xml, which is a hierarchical structured

file format, primarily used to exchange information between applications. You see it has `lat`, `lon`, and `time` tags. Apparently, the GPX tracker has recorded the gps location every few seconds (see `https://en.wikipedia.org/wiki/Geographic_coordinate_system`).

3. Consider that, using the string parsing functions in Python, you could extract this data yourself. However, others have already created this functionality, so if something exists that is already used by many others and extensively tested, it is better to use that. Therefore, install the `gpxpy` package to read the info from the gpx file. Remember, you can install packages using pip at the terminal in Pycharm: `pip install gpxpy`.
4. The code examples at `https://github.com/tkrajina/gpxpy` show you right away how to use the `gpxpy` package. Copy the code and print for instance the latitude of all points from the gpx file.
5. Once you can print the latitudes of all points, you know you are on the right track. Now remove the print, but try store the points as a list of points (call this list `points`), such that each point in the list is a list itself with the three items `[latitude, longitude, time]`.
6. Once you have the list of points, you should realize that `points[-1]` is the last point, and `points[0]` the first. Now explain that
   ```
   duration = points[-1][2] - points[0][2]
   ```
   is the total duration of the track.
7. It is important to know that duration is a `timedelta` object. With `duration.seconds` you'll get the duration in seconds. With these ideas, write code to print the duration of the track in seconds. Open the gpx file in Pycharm (or some other editor) to check whether your result makes sense; just check the time stamp of the first point in the gpx file and the time stamp of the last point.
8. The next step is to compute the total distance traveled. For this, use the `geopy` package, or some similar package to compute the distance between points with gpx coordinates. (Once again, I did not want to find out how to compute such distances, I just know somebody has solved this, so my problem reduces to finding the right package that does the job for me.)
9. Compute the distance between the first two points as follows. Put

```
from geopy import distance
```

at the top of your program, so that you can use the distance object from the geopy module. Then use the following code to compute the distance:

```
p1, p2 = points[:2]
print(
   "distance between first two points in meters: ",
   distance.distance((p1[0], p1[1]), (p2[0], p2[1])).meters
)
```

10. Once you know how to compute the distance between the first two gps points , you should find a way to add all distances together to compute the total distance.
11. Finally, now that you have the duration and the total distance, it's easy to compute the average speed. Complete the code and print all results.

4.2 Data Analysis

So, now we have the raw data; what can we do with it? We could for instance compute the speeds at each time point, and then find the maximal speed. This turns out to be a bit tricky, so have we to modify the analysis.

The lesson of the section is that, in data science, the transformation of raw data to something that is useful and realistic (i.e., fits in the context), nearly always requires quite a bit of additional effort. In other words, the analysis starts, not ends, with the raw data.

1. Iterate through all segments, and calculate the speed per segment.
2. Compute the minimal and maximal speed; check the internet to how to efficiently find the maximum element in a list. The highest speed in the gpx file from Brightspace should be around 10 km/h.

If you would have used the browser to plot our track on a map, you would have seen that it is a walk around the Duisenberg building. In fact, we recorded it for this course when we walked around the building. Certainly we did not run, so the max speed of about 10 km/h is impossible.

1. One guess is that the time measurements are somewhat irregular. With the above tools you can make an algorithm to compute the time differences between

the successive points and then compute the largest and smallest difference. Is there a large spread in these time differences or not?

2. Can you find a reason on the web about why this problem (of too high speeds) occurs?
3. As an optional challenge, can you find a method to repair this problem? (Hint, the technique is called 'smoothing'; it quickly becomes quite mathematical, and also very interesting.) Optionally, you can watch the following talk about a more advanced way to correct for errors in gps measurement: `https://www.youtube.com/watch?v=9Q8nEA_0ccg`

In this case we have to repair outliers in the speed measurements. Errors in raw data occur always. Some examples are measuring the temperature of a machine to predict a breakdown, website usage statistics (which could be polluted by visits of search engines), and typo's when machine operators must enter data into a system. It is often difficult or impossible to determine whether an outlier is a measurement error or a real outlier. In general, you should make a number of different measurements (such as in our case the max, mean, and min of the speed) to obtain an impression of the quality of your data. Then you should do some (or a lot of) massaging of the raw data to get insights you need.

4.3 Plotting the route

The gps coordinates themselves are of little values for us. Luckily, we can easily plot the route on a map.

1. For this, you can use the `smopy` package, which can be installed with pip.
2. Study the example code on [`https://github.com/rossant/smopy`](https://github.com/rossant/smopy).
3. When initializing the map, you need to specify the approximate part of the map you wish to show. For this, lookup the minimum and maximum latitude and longitude using your program, and use these as parameters when you initialize the map.
4. Another parameter in the map initialization is the zoom level. The website mentions `z=4`. You can increase this depending on the route you walked. With the gpx file on Brightspace, a zoom level of 14 is sufficient.
5. If you wish to show the map a bit bigger, increase the size in the `show_mpl` call.

6. By plotting the individual gps locations as points on the map, you can see the route. To test this functionality, first try to show the first gps location in the gpx file.
7. If this works correctly, iterate and plot all points on the map. If you think the points are too thick, use `ms=3` and `mew=1`.

4.4 Optional extensions

1. A plot of the speeds in a graph (as determined in 4.2) is quite interesting. For instance, Uber eats can figure out how long payments typically take. When the courier is at the door of a client, the speed must be zero.
2. You can determine the color of a plotted point through code, for example:
   ```
   ax.plot(x, y, 'or', color="green", ms=3, mew=1);
   ```
   This can be used to indicate the speed per segment on the map. Try to give different colors for segments where the speed is below 4, between 4 and 7, and above 7 km/h.
3. Other applications are very easy now too. If you want to know your total amount of kilometers walked in a month, just track your walks. Put the tracks that are obtained with walking into a specific directory (to prevent messing up your walking data with your cycling data, for instance). Then, for the total distance, just add the total distances per track (after you corrected the data for suspect measurements of course).

5. Elastic Search: Manipulating and Querying big data - part 1

5.1 Introduction

Computer programs need data. In the past weeks, you used demand data, profit data, and machine capacity data to determine the optimum product mix. First, you added the data directly in the program, for example the demand of products and the capacity of machines in the constraints and goals of an optimization model. Second, you learned how to put the data in lists, so you don't have to change the program logic when the data change; you just change the list. Third, you have imported data that was in files, in our case Comma Separated Values (csv) files. So now you don't have to change your program at all anymore if the underlying data changes. You can just change the csv-file.

Usually, csv files are generated by systems that extract the relevant data from databases. For example, an ERP system can generate weekly or daily a csv file with demand data and email it to you. Such files, however, simply cannot keep up with the size of many real world business applications. As an example, consider Thuisbezorgd, which, after merging with Just Eat, delivers 400 million meals per year (https://www.parool.nl/nieuws/thuisbezorgd-leverde-in-2019-38-miljoen-maaltijden-aan-de-deur). This is simply too much to put in a simple file. Even more interesting is that they can collect the gps data of their couriers to track their routes, speed, etc., Suppose that the average trip would take 10 minutes, and they would collect the location each 10 seconds, this would amount to 24 billion gps locations per year. You can imagine that it would take a really long time to go through such a file line by line using Python.

Luckily, specialized programs have been developed to handle such amounts of data: database management systems. These programs can store and process huge amounts of data and provide functionality for backups. Companies usually have centralized their database systems to keep the data consistent. So if someone changes data, it becomes immediately visible for all other users. Multiple programs can add and request data in parallel from such databases. For example, order managers enter their orders to the ERP system, machine operators indicate when they start and when they finish production, and you, as production manager, request production progress reports per machine per day.

Interestingly, you can also request the data yourself directly from such databases with Python (and other programming languages). This means that you are not limited to the functionality offered by ERP systems, but that you can also extract data to make analyses yourself.

The centralized storage that databases offer is one of the biggest facilitators of the growth of IT. Usually, companies have expensive computers as database servers, that are used by all computers and systems used in the organization. These servers have huge amounts of memory, fast network connections, arrays of hard disks offering redundancy so nothing is lost if a hard drive fails, and automatic backup facilities. Everything is done to make sure that data is always available in a fast and flexible way, and stored safely so it does not get lost. These databases are known as Relational Databases because they organize data as relations between tables, which are conceptually similar to Pandas frames. Some well-known examples are SQL Server, Oracle, and SQLite. Almost all database systems can be queried through the language SQL, which stands for Structured Query Language.

In the past decade, several developments have led to other kinds of databases next to relational databases, for example:

- Relational databases are not really suitable for full text search in documents (for example, scientific articles, web pages, or emails). The way in which tables are stored makes full text search inefficient.
- Relational databases require a predefined fixed structure of the data. There are many cases, however, where the structure of the data is dynamic.
- Although relational databases can handle quite big databases, they do not scale well to really big data sets that need multiple computers.
- The way in which data is traditionally stored in relational databases makes them less useful for numerical analyses.

To overcome the above problems, other kinds of databases have been developed in the past decade, generally called NoSQL databases (where the 'no' can mean either non-relational databases or not only SQL; https://en.wikipedia.org/wiki/NoSQL). There are several generic types of NoSQL databases, which conceptually differ in the way in which they store the information. Examples of information collectors are Objects, Documents, Tuples, RDF, and Graphs. These databases can overcome one or more of the problems of traditional relational databases, for example:

- Efficient reverse indexing structures to allow efficient full text search.
- Flexibility in the data structure; not necessary to predefine what data needs to be stored.
- Distributing data over multiple machines (the "pieces" of such a data repository are usually called shards, and the machines where they reside are called nodes) makes scaling to practically unlimited size possible.
- By storing data as columns instead of as rows and adding efficient sorted indexes, numerical analysis can be done much more efficient.

Due to these features, NoSQL databases can be hundreds of times faster than relational database. A disadvantage, however, of most NoSQL databases is that they do not offer true ACID. This is an acronym (Atomicity, Consistency, Isolation, Durability) which basically means that database transactions are reliable and consistent. For example, if a NoSQL database is distributing new records over the nodes/shards, there could be a short time where a search query does not yet include those newly added records. Also, if data is distributed over multiple nodes, most systems do not provide accurate results if the data needs to be returned sorted on one of the fields.

In the course Data Analysis and Programming for OM we will use the NoSQL database Elasticsearch that can be downloaded freely from https://elastic.co. Elasticsearch is well suited for full text search and numerical analyses and scales very well to very large databases (think petabytes of data distributed over thousands of dedicated servers). You can install Elasticsearch on your laptop with a small dataset in order to create an integrative python application for analysis, visualization, and optimization. When it works, you can point your program to an instance of Elasticsearch cluster running in the cloud and have the same program running on terabytes of data easily.

In this practical, you will learn how to install and query Elasticsearch, and how to communicate with Elasticsearch through Python. Note that we will not really use Big Data, because you would need a cluster of servers for this. However, the programs you make on your laptop could work exactly the same if scaled to a really big database.

5.2 Installing Elasticsearch

Elasticsearch can be downloaded for free at https://www.elastic.co/downloads/. Just follow the installation instructions. (Select the variant that is for your operating system, Windows, MacOS, or Linux).

Note that Elasticsearch is a standalone program. If you reboot your computer, you have to start it again. The data that you added/imported, however, will still be there.

When installed, Elasticsearch can be accessed for CRUD through a REST interface and talks JSON. Too much jargon? Let's analyze it:

- The data in your database needs to be created, read, updated, and deleted (https://nl.wikipedia.org/wiki/CRUD) by your programs.
- Elasticsearch is a server, which means that it won't do anything by itself. It just waits on other programs for instructions (like a request to add a record or retrieve a record). Like most servers, it offers access through REST (https://en.wikipedia.org/wiki/Representational_state_transfer). In simple terms, programs talk to Elasticsearch with HTTP, just like your web browser talks to a webserver. You can even query Elasticsearch through your web browser (http://localhost:9200/ if you did not change the default configuration during installation).
- Elasticsearch stores and sends data as JSON (https://en.wikipedia.org/wiki/JSON). This is a way in which hierarchical data can be represented such that it is still readable by humans.

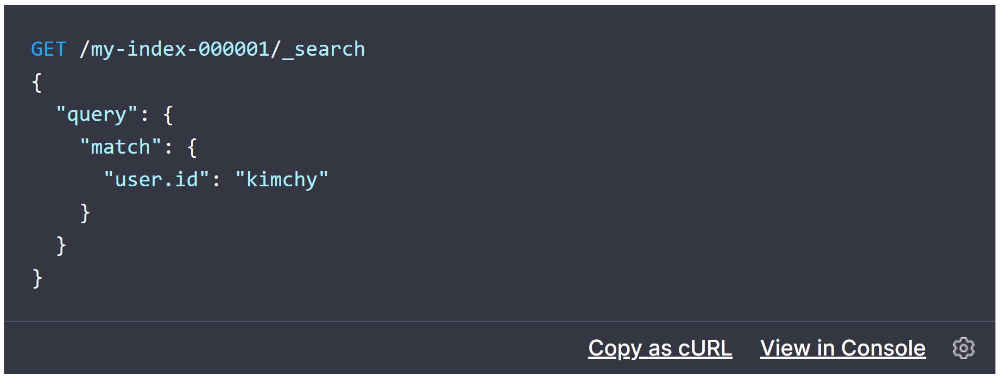
Because Elasticsearch uses these standard protocols, it is very easy to develop components for programming environments (such as Python) to talk to Elasticsearch. To use Elasticsearch in Python you could manually communicate with the server using very specific REST APIs with, for example, the library httplib. However, there are better libraries available that make life easier if you want to work with Elasticsearch in Python:

- elasticsearch
- elasticsearch_dsl
- pandasticsearch

Just install them with the package installer `pip`.

Elasticsearch offers many functionalities, and their website has an extensive manual: https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html

Each function in the manual is explained and demonstrated by examples you can try yourself. The examples are shown like this:



```
GET /my-index-000001/_search
{
  "query": {
    "match": {
      "user.id": "kimchy"
    }
  }
}
```

If you install Kibana (which can also be found on the Elastic website), you can simply click on 'view in console' and you get a page in your browser in which you can execute the query and see the results. So Kibana provides a nice way to test your queries before you implement them in Python.

5.3 Inserting and querying Elasticsearch through Python

The Elasticsearch Python libraries provide easy access to Elasticsearch servers. The following program creates an index (which is the term Elasticsearch uses for a database), inserts a document, queries for a document, and deletes the index:

```
from datetime import datetime
from elasticsearch import Elasticsearch
es = Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])
es.indices.create(index='persons', ignore=400)
es.index(index="persons", id=1, body={"name": "wout", "hobby":
"programming", "age": 49, "timestamp": datetime.now()})
es.index(index="persons", id=2, body={"name": "anna", "hobby":
"netflix", "age": 16, "timestamp": datetime.now()})
result = es.get(index="persons", id=2)
```

```
8. print(result)
9. print(result['_source']['timestamp'])
10.     # es.indices.delete(index='persons', ignore=[400, 404])
```

Run this program. Now let's look at what happens:

1. The standard date time library of Python is imported. It allows us to request the current time, which we use to add a timestamp to the record we add in the database.
2. The Elasticsearch library is imported.
3. Create a new variable (object) that handles the connection to the Elasticsearch database. This database can be on your own computer, but it can also be on another computer. All computers connected to the Internet have an IP address (IP = Internet Protocol) consisting of 4 bytes. Computers communicate to each other using these IP addresses. Normally, you point to other computers with a name, e.g., `www.google.com`. Your computer first translates this to the IP address of the computer of Google to be able to communicate. The `127.0.0.1` in the example code is a special address: it always points to your own computer. You can also use the name `localhost` for the same purpose. If you installed Elasticsearch on a different computer, or you have rented an Elasticsearch instance in the cloud, you need to change the command accordingly. (Note that 'in the cloud' in essence is a fancy term for 'another computer connected to the internet').
4. Create an index with the name 'persons'. In an index you group similar data, analogous to tables in an SQL database. You would, for example, have separate indexes for production characteristics, demand, machine characteristics, etc. If the index specified in the command already exists, Elasticsearch will give you an error with the code 400. With the ignore=400 parameter, you tell it to ignore this error. Otherwise, if an index already exists with this name, executing the code will stop with an error message.
5. With the `index` command you add a new document to the database. An Elasticsearch index contains of one or more documents, and each document has a unique Id. You can update a document by sending the new document with the same Id. The body contains the actual document. In this case, it has fields for name, hobby, and age, and a field 'timestamp' with as value the time at which the records is inserted. The new record is put in the database, and, if there already was a record

with this id, the old one is deleted. The document itself needs to be formatted as JSON.

6. Another record is added.
7. With this command, you ask Elasticsearch to return the document with the Id 2. Elasticsearch returns Json formatted data, which is automatically converted to a Python dictionary.
8. If you print the result of the search query, you'll see some meta-data and the document itself. Carefully analyze the structure!
9. Because it is a dictionary, you can also retrieve specific fields in the document.
10. The index is deleted (and instructed to not give an error message if the index does not exist). However, the line is commented so it is not really executed, otherwise we lose our data.

If you direct your web browser to http://localhost:9200/persons, you can see the structure Elasticsearch has made based on the records it received through the `es.index` command. It has automatically added the fields you specified, and guessed what data type they are. Study this structure carefully, and determine whether the guesses of Elasticsearch were correct. This is important. For example if you have numeric data, you can do range searches (give me all records of persons where the age is between 20 and 30). If such a field is indexed as text instead of as a number, it cannot do such queries.

You can also query the data itself from your browser, for example:

```
http://localhost:9200/persons/_search?q=programming
```

and

```
http://localhost:9200/persons/_search?q=*flix
```

Tip: If you use Chrome, install a Json formatter (for example: https://github.com/callumlocke/json-formatter). It makes it much easier to read the output.

Performing queries in your browser is easy for testing purposes. More complex queries, however, are easier to do using Kibana and Python, which will be the topic of the next section.

5.4 Querying Elasticsearch through Python

In your previous program, you asked Elasticsearch to return the document with identifier 1 using `es.get(index="persons", id=1)`. In addition to this trivial query, Elasticsearch offers many kinds of queries to let you specify exactly which documents you want to get. Some functionality is discussed here. You can consult the Elasticsearch manuals for more options.

For text fields, you can search for the following:

- Search for words that exist somewhere in the text
- Search for wildcards, for example, searching for `prog*` would result in records containing program, programming, programmer, programmers, etc, and searching for `*flix` gives back all records with words that end with flix.

For numeric fields you can:

- Search for exact values
- Search for ranges (e.g., age between 18 and 80).

For geographic fields, you can:

- Search for records with a gps coordinate within a certain distance of another coordinate. For example, all restaurants that are maximum 10 km distance of your current position.
- Search for all records within a certain gps area determined by a number of points. For example, you can make a polygon with gps points that indicate the boundaries of The Netherlands. Then, feeding this in a query to Elasticsearch, you get all records with a gps coordinate within this shape.

Additionally, you can combine such queries. For example, 'give me all me meal deliverers that currently are within a 2 km radius of the Grote Markt in Groningen that are currently not delivering anything, and that have still a minimum of 30 minutes in their shift left'.

All this functionality is available through Elasticsearch libraries in Python as well. The search requests need to be specified in Json. A basic example of a search command is:

```
from elasticsearch import Elasticsearch

import json

es = Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])
```

```
4. search_body = {
  'size': 100,
  'query': {
    'bool': {
      'must': {
        'term':{
          'hobby': 'netflix'
        }
      }
    }
  }
}
5. result = es.search(index="persons", body=search_body)
6. print(json.dumps(result, indent=4))
```

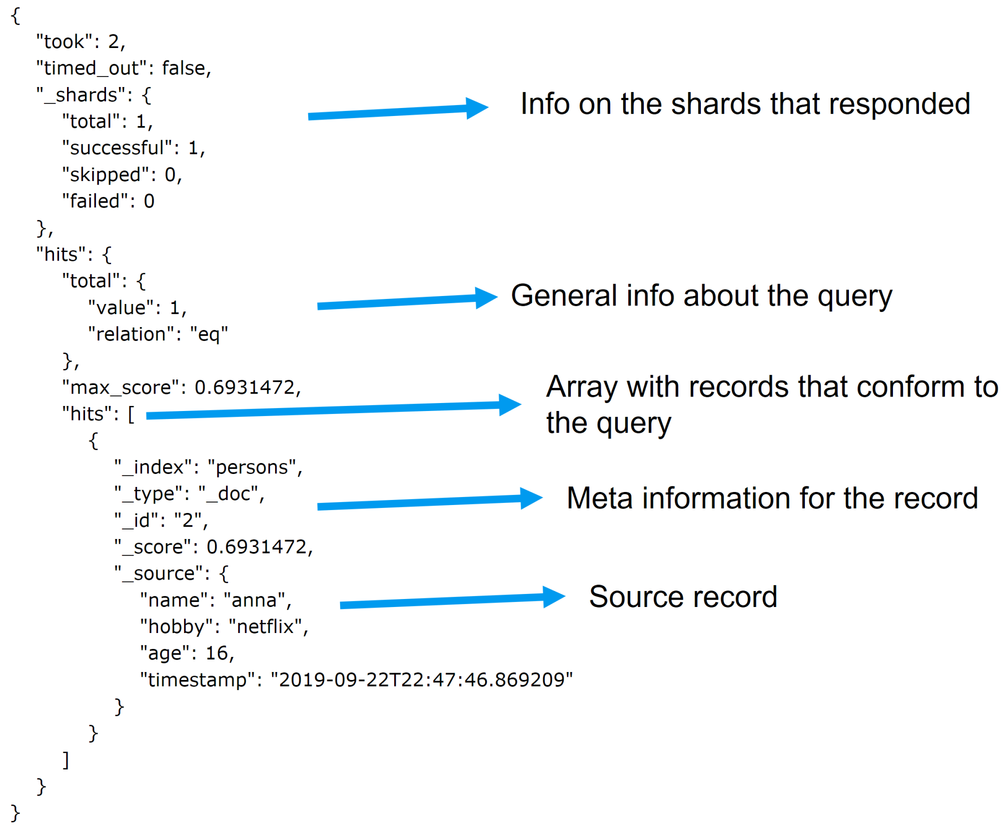
Let's analyze this program step by step:

1. We import the Elasticsearch library.
2. We import the Json library
3. We specify to the library where our Elasticsearch server is running.
4. We specify the search request. This is formatted as a Json structure. First, we ask for a maximum of 100 records using the `size` parameter. If we don't specify it, we get by default 10 records returned. After that, we specify the query itself, in this case a so called boolean query where the field 'hobby' must have the term 'netflix'.
   The boolean operator is only active when we specify more criteria. Then we can use it to specify whether the results must meet all criteria (for example persons having netflix as hobby AND age<20), or should meet at least one of the criteria (netflix as hobby OR age<20).
5. The search request is sent to Elasticsearch. We ask it to run the search query for the index 'persons'.
6. The result is printed. **Tip:** if you print the json output in Pycharm, it is very difficult to read. If you import the json library, you can use:

```
print(json.dumps(result, indent=4))
```

This command pretty-prints the text in the output window, making it much easier to read.

This will show the following output:



```
{
   "took": 2,
   "timed_out": false,
   "_shards": {
      "total": 1,
      "successful": 1,
      "skipped": 0,
      "failed": 0
   },
   "hits": {
      "total": {
         "value": 1,
         "relation": "eq"
      },
      "max_score": 0.6931472,
      "hits": [
         {
            "_index": "persons",
            "_type": "_doc",
            "_id": "2",
            "_score": 0.6931472,
            "_source": {
               "name": "anna",
               "hobby": "netflix",
               "age": 16,
               "timestamp": "2019-09-22T22:47:46.869209"
            }
         }
      ]
   }
}
```

Info on the shards that responded
General info about the query
Array with records that conform to the query
Meta information for the record
Source record

The output not only contains the records you are looking for themselves, but also quite some meta-data. Some explanation of what you see here:

- 'took' is the amount of time it took for Elasticsearch to process your search request. In this case, it took 2 milliseconds, or 2/1000th of a second.
- Elasticsearch spreads the data over shards. Each shard contains a part of the data. This is where Elasticsearch gets most of its speed, because it can send the query to all shards in parallel, and is thereby able to use all the cores of the microprocessor in your laptop. In the output, Elasticsearch also reports whether any shards have failed.
- In the 'hits' section, the records that conform to the query are sent back. It first reports the number of records that match your query, in this case, only 1. By default,

Elasticsearch will report a maximum of 10.000 records. In other words, when it has found 10.000 records, it will stop searching, to avoid really long response times. If you want to know exactly how many record meet your query, you can override this behavior by adding track_total_hits=true to the search request.

- Next to that, it gives the "max_score". By default, records are sorted by relevance (using the following algorithm: https://en.wikipedia.org/wiki/Okapi_BM25). The max_score is the highest score found in all records.
- After that, it will give you another 'hits' section, in this case an array in which all records are returned (you can see it is an array because it starts with a `[` instead of a `{`). Each element in the array again is a hierarchical structure with, for each record, the index in which it was found, the shard, the identifier, and the score of the record relating to the search query. Then, finally, it gives back the record itself in the "_source" section. This is the exact record you sent to Elasticsearch to index.

This is the raw text as sent back by Elasticsearch. The Python library converts it to a dictionary, within which each element respectively is a dictionary itself, or an array, a string value, a numerical value, or a gps coordinate. So `Result` is a dictionary, `Result["hits"]`also is a dictionary, and `Result["hits"]["hits"]` is an array. Hence, if you request:

```
record = Result["hits"]["hits"]
```

you will get a python array containing all hits, where each element in the array is again a dictionary. Finally, within this dictionary you can request the `["_source"]` object. It sounds cumbersome, but because it is always the same structure, you get used to it quite quickly.

To conclude, you can get the number of records that meet your query in the following way:

```
numberOfHits = Result["hits"]["total"]["value"]
```

Because the value (in this case 1) is not enclosed by quotes, it is interpreted as a number, so you can, for example, do calculations with the variable `numberOfHits` (or however you call it).

You can get the first record in the result-set in the following way:

```
sourceRecord = Result["hits"]["hits"][0]["_source"]
```

and the value of the name field in this record as follows:

```
name = sourceRcord["name"]
```

Of course, if you have more records in your results, you can iterate through the array with a `for` loop like mentioned above. Be careful not to use the value reported in `Result["hits"]["total"]["value"]` to iterate through your `Result["hits"]["hits"]` array. As mentioned earlier, by default (if you don't specify the `size` parameter), Elasticsearch returns 10 records. So if 67 records meet your query, but Elasticsearch has only returned 10, you should not iterate over 67 records. We leave it to you to figure out how to determine how many records we can iterate over in a given response of Elasticsearch.

5.5 Practical assignment

In this assignment you will work with a big Json file ("taxi-1000000.json"), which contains the first 1.000.000 taxi trips of New York in 2015. The file is on Brightspace as compressed zip file. Please download and decompress it.

The file has various fields such as pickup and drop-off location, number of passengers, start- and ending time, price, taxes paid, etc. This is an excellent dataset to get familiar with Elasticsearch, as it contains both numeric, time, and geolocation data.

The first task is to create an index called "taxi" and import the file in Elasticsearch in this index. Before starting this assignment, make sure you have enough space on your ssd or hard drive. The unzipped file with taxi trips is approximately 500MB, and a similar amount of space will be used by Elasticsearch.

There are two complications compared to the previous assignment.

First, Elasticsearch is quite flexible and if you add new records with fields that were not yet seen before in the index, it just adds those fields. However, fields are configured by default as string or number, whereas we also have GPS locations and date/time fields. If the GPS location is stored as a string (text) instead of numbers, we cannot query Elasticsearch for, for example, all pickups on January 12th within 100 meters of a certain GPS coordinate. For that, it needs to be stored as so called geo-points. So what we will do is first create an index in which we explicitly specify the field types (in Elasticsearch terminology: mappings). To do so, we extend the es.indices.create command to include the mapping:

```
from elasticsearch_dsl import Search

from elasticsearch import Elasticsearch


es = Elasticsearch([{'host':'localhost', 'port': 9200}])


settings = {
    'settings': {
            "number_of_shards" : 3
    },
    'mappings': {
        'properties': {
            'pickup_datetime': {'type': 'date', "format": "yyyy-MM-dd HH:mm:ss" },
            'dropoff_datetime': {'type': 'date', "format": "yyyy-MM-dd HH:mm:ss" },
            'pickup_location': {'type': 'geo_point' },
            'dropoff_location': {'type': 'geo_point'}
        }
    }
}
es.indices.create(index='taxi', body=settings)
print("index created")
```

You can check it by browsing to `http://localhost:9200/taxi`

If we now add a record, it expects the above field types in the source data to be correctly formatted as date/time respectively a GPS location (latitude and longitude). After adding records, we can efficiently query for distance and time.

The second complication is that the Json file you downloaded contains a record (=taxi trip) on each line. You could read the Json file line by line in Python and send it to Elasticsearch. However, this takes a long time, because the overhead of communication between Python and ES is much longer than the actual time it takes for ES to add a single line to the index. A faster way is to do this batch-wise using the Elasticsearch Bulk API. Sending records batch-wise will only give this communication overhead once per batch instead of for each record. A template for this will be available on Brightspace.

When the file has been imported successfully, create a Python program to answer the following question: How many taxi pickups were within 100 meters of the following GPS location:

`Longitude: -73.84300994873047, latitude: 40.71905517578125.`

Tips:

- If you don't know where to start, go back to the basic search example, and make sure you understand each line of code. Search how to do a 'match all' query, and study the structure of the output. The lecture slides show an explanation of the structure of the output of Elasticsearch queries. Then, adapt from there.
- You could of course do a `match_all` query, iterate all records in the result set, and compare the pickup location in each record to the above point using the `geopy distance` function you used in week 3. However, this will be very slow, and Elasticsearch is optimized for these kinds of tasks. So, use a geo_distance query for your search. Google for `geo_distance elasticsearch` for examples how to do this.
- You can find the number of hits in the ["hits"]["total"]["value"] property of the results object (see picture in 5.3)
- As mentioned, Elasticsearch gives back the first 10 records. If you want to see more, add the `size` parameter to the search request. You can also add both the `from` parameter with the `size` parameter and request the records from Elasticsearch in multiple batches.
- As a challenge, plot all pickup points from the query on a map using Smopy or Folium. Experiment with the distance parameter in the query. For example, you can increase the distance to 1000 or 10000 meters. Also, you can consider the `total_amount` field in the color of the point you plot. Then, you can visually determine whether the location has a relation with the price of the taxi trip.

6. BIG DATA - PART 2

Suppose, as an operations manager, you have logged machine status and temperature each second in the past 5 years of your most important machines, resulting in a dataset of 157.680.000 records per machine. Suppose you have stored this data in a csv file, and now want to know the average and standard deviation of machine temperature time in May 2018. For this, you need to go through each line in the CSV file, check if the corresponding data point happened in May 2018, and add the temperature to a Pandas dataframe. When the whole file is checked, you can calculate the required average and standard deviation.

Parsing so many lines can take a very long time. If you import the csv file in Elasticsearch (or skip the csv part altogether and directly add the log lines to Elasticsearch when they happen), querying becomes much faster. In general, simple queries in Elasticsearch take a few milliseconds if it is configured correctly.

So, selecting the data that you want to analyze and doing calculations is something that you prefer to outsource to the database, in this case Elasticsearch. All databases have basic (or even advanced) functionality for calculations. So instead of asking all records, and calculate the average yourself using Pandas, you ask the database to calculate the average of all records. Or you could, for example, ask it to calculate the average per day of the last 365 days. Databases are optimized for these kinds of questions, and the data that you receive and do something with in Python becomes quite small (in the latter case, you receive only 365 numbers, which you can then plot in a graph or use for machine learning purposes). Elasticsearch has many functions for numerical analyses which can work on large datasets, and it is especially strong in so called 'aggregations' (https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations.html). Therefore, Elasticsearch is not only used as full text search engine, but also very popular as a data analytics tool.

In the previous practical you installed Elasticsearch, added records manually, added a large dataset using elasticsearch_loader, and you queried indexes using text search and geo-distance search. Queries can be combined to very complex search structures, which is why Elasticsearch is used on the largest websites worldwide (for example, Ebay, Wikipedia, Netflix, Linked-in, and Stackoverflow). The last few years, however, Elasticsearch also has become one of the most used tools for metrics, statistics, and machine learning. It is now an analytics platform used by many Data Scientists. The most important functionality used for Data Science in Elasticsearch is Aggregations. In this practical you will practice with statistics and aggregations in Elasticsearch.

6.1 Statistics in Elasticsearch

In Practical 4A you worked with Pandas, and you did some basic calculations, such as mean, median, min, and max. In Pandas, working with really large datasets can give problems. It can be slow, or you can run out of memory. Using Elasticsearch, you can also do statistics on large datasets.

Let's try a basic example: calculate the average of the field `total_amount` in our taxi index:

```
from elasticsearch import Elasticsearch
import json

es = Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])
search_body = {
  "size": 0,
  "aggs": {
    "avg_amount": { "avg": { "field": "total_amount" } }
  }
}
result = es.search(index="taxi", body=search_body)
print(json.dumps(result, indent=1))
```

Basically, it is just another request that we send to Elasticsearch, similar to the example in 5.4. However, instead of performing a query, we now request an aggregation, hence the `aggs` (for aggregates) command. In such commands, we always start by giving our requested aggregate a name, in this case `avg_amount`. This name can be anything and is used in the results to indicate which aggregate Elasticsearch is reporting. Therefore, it is best to choose a name that makes sense. Then, we specify which kind of aggregate we need, and for which field. In this case, `avg`, which will return the average for the field `total_amount`. Because we ask Elasticsearch to do a calculation on all documents, we call it a 'metric aggregation'.

Run this and study the results; they are quite obvious.

Other kinds of calculations you can request are, for example, `min`, `max`, `sum`, etc. (see also https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-metrics.html). You can also request more extensive statistics in one command using `stats` or `extended_stats`. Additionally, you can combine queries and metric aggregations. For example, the following program first executes the geo-distance query that you created in practical 4B. The `size` parameter is set to 0, indicating that we are not interested in the records themselves now. Additionally, we request an `extended_stats` aggregation in the `aggr` section, which will only be performed for the records that meet the query:

```
from elasticsearch import Elasticsearch
import json

es = Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])
search_body = {
  "size": 0,
  "query": {
    "bool": {
      "must": {
        "geo_distance": {
          "distance": "100m",
          "pickup_location": {
            "lon": -73.84300994873047,
            "lat": 40.71905517578125
          }
        }
      }
    }
  },
  "aggs": {
    "statistics_amount": { "extended_stats": { "field": "total_amount" } }
  }
}
result =es.search(index="taxi", body=search_body)
print(json.dumps(result["aggregations"]["statistics_amount"], indent=1))
```

So, with this request, we ask the statistics for the field `total_amount`, for taxi trips of which the pickup location is at most 100 meters from the gps coordinate (-73.84300994873047, 40.71905517578125). It gives the following output:

```
{
 "count": 67,
 "min": 3.799999952316284,
 "max": 53.09000015258789,
 "avg": 15.054179038574446,
 "sum": 1008.6299955844879,
 "sum_of_squares": 22569.616444259274,
 "variance": 110.23164040378866,
 "variance_population": 110.23164040378866,
 "variance_sampling": 111.90181677354303,
 "std_deviation": 10.499125697113483,
 "std_deviation_population": 10.499125697113483,
 "std_deviation_sampling": 10.578365505764255,
 "std_deviation_bounds": {
  "upper": 36.052430432801415,
  "lower": -5.944072355652519,
  "upper_population": 36.052430432801415,
  "lower_population": -5.944072355652519,
  "upper_sampling": 36.21091005010295,
  "lower_sampling": -6.102551972954064
 }
```

IMPORTANT: During the practicals we sometimes get a question regarding a query that is not working. When looking at the code, we see queries formatted like this:

```
search_body = {
"size": 0,
"query": {
"bool": {
"must": {
"match_all": {
}
},
"filter": {
"geo_distance": {
"distance": "100m",
"pickup_location": {
"lon": -73.84300994873047,
"lat": 40.71905517578125}}
}
},
"aggs":
{"statistics_amount": { "extended_stats": { "field": "total_amount" }
}
}
}
```

Can you see the error here?

In Python, you always need to be accurate in indenting your code. If you don't indent the code to run in, for example, a for loop, it is a syntax error. However, Elasticsearch queries are seen by Python as strings, and Python itself does not care about indents there. Still, now that the Elasticsearch queries are becoming more complex, make sure you indent your Elasticsearch search body properly. If you don't, your code might run fine, but your queries become difficult to read and adapt. This is especially important when the queries get more extensive and include aggregates. If there is an error in your query, it is almost impossible to debug if it is poorly formatted.

6.2 Bucket aggregations

Bucket aggregations are calculations based on grouping of data. This is a refinement of metric aggregations, where we do the calculations on all the data. We will explain what this means through a simple example.

A company like Zara will record all orders in a big database. This database is used for order management, generic accounting, etc. However, such a database also provides a wealth of information for a data analyst. What would we typically do with such a database? In the previous section, we showed that it is possible to do generic statistics on a big dataset. However, the data analyst is usually not really interested in the average number of orders or the average order amount of the total dataset. Rather, he/she would want to know such averages per region, or per month, or per type of clothing, and probably the combinations of such criteria would be even more interesting. This is where bucket aggregations come into play.

What is a bucket aggregation? Consider the following order data:

Order	**Type**	**Region**	**Price**
1	Trousers	France	10
2	Jacket	Spain	25
3	Socks	France	5
4	Trousers	France	5
5	Socks	Spain	7

A bucket aggregation is to first determine the unique values (called buckets) in a certain column, and then count the number of occurrences for each value. So, if we aggregate the data for Region, we get the following:

Region	**Count**
France	3
Spain	2

We can also aggregate on Type:

Type	**Count**
Trousers	2
Jacket	1
Socks	2

To go a step further, we can also be interested in other data than the count. In this case, the average price could be interesting:

Type	**Count**	**avg(price)**
Trousers	2	7.5
Jacket	1	25
Socks	2	6

Of course, all metric aggregations can be used here, such as the minimum value, the maximum value, the median, etc.

To make it a bit more complex, but also more useful for the data analyst, we can also make sub-aggregations. We then combine the unique values of different fields. For example, we can determine for each type of clothing the average price per region:

Type	**Region**	**Count**	**avg(price)**
Trousers	France	2	7.5
Jacket	Spain	1	25
Socks	France	1	5

	Spain	1	7

Aggregations can also be combined with queries, for example to filter on minimum price. Usually, the underlying data contains much more features than our simple example. In this case, it could be the home address of the person that bought the product, how much they bought previously, and the average income in the neighborhood where the person lives. Then, aggregations can be made combined with complex queries, for example, give me product types and their average price of all purchases last month in our shop in Groningen by people who live in an area where the average yearly income > E60.000. Playing with queries, aggregations, and calculations can give great insight in data. Doing this with big data sets has taken a huge flight in the past 15 years due to the availability of open source software such as Hadoop, Cassandra, MongoDB, and Elasticsearch.

In the remainder of this section, we will practice with bucket aggregations in Elasticsearch.

To get bucket aggregations, we could manually combine statistics (metric aggregations) with queries. For example, to get the average price per product type, we could simply do metric aggregations for each product type, similar to what we did in section 6.1 for our taxi dataset.

On Brightspace, you can find the code to create an index and add the 5 orders. Run it before you try the following queries.

Below you find an example for the product type 'trousers':

```
from elasticsearch import Elasticsearch
import json
es = Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])
search_body = {
  'size': 0,
  'query': {
    'bool':{
      'must':{
        'term':{
          'type.keyword': 'trousers'
```

```
            }
          }
        }
      },
      'aggs' :{
        "averagePrice" : {
          "avg": {
            "field": "price"
          }
        }
      }
    }
result =es.search(index="products", body=search_body)
print(json.dumps(result, indent=4))
```

Note: you see here we query for `type.keyword` instead of `type`. The '.keyword' can be added if you are interested in the complete value of the field, rather than only one of the words within this field. For example, suppose we also would have the regions 'ireland' and 'northern ireland' in our database. A terms query for 'ireland' in the field 'region' would give back also the records where the region is actually 'northern ireland'. By querying for 'region.keyword', we ask Elasticsearch to only return the records where the complete and exact field value is 'ireland'.

The above query gives the average price of all records of which the type is a trouser. We could repeat this query for the other types (socks, jackets, etc.). However, this approach has several disadvantages. First, we need to know all product types beforehand. Second, because we must do multiple queries, this approach could be slow, especially if we have many product types. Therefore, databases have dedicated functionality for these kinds of analyses. In Elasticsearch they are called `bucket aggregations`. Other terms you could encounter for this functionality is `group by` queries in relational databases, and `facets` in NoSql databases.

The term bucket aggregation can be explained as follows:

- Bucket: the group of documents for which the aggregation is requested

- Aggration: the calculation that you wish to perform on each bucket. If not specified in Elasticsearch, it returns the count, i.e., the number of records in the bucket. However, you can also request calculations on fields within the bucket, e.g., min, max and avg of price.

There are several kinds of buckets in Elasticsearch. The most obvious one is the 'terms' aggregation. It will simply create a bucket for each unique value of a field, for example:

```
from elasticsearch import Elasticsearch
import json

es = Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])
search_body = {
  "size": 0,
  "aggs": {
    "countPerRegion": {
      "terms": {
        "field" : "region.keyword"
      }
    }
  }
}

result = es.search(index="products", body=search_body)
print(json.dumps(result, indent=1))
```

You should get the following output:

```
{
 "took": 0,
 "timed_out": false,
 "_shards": {
  "total": 1,
  "successful": 1,
  "skipped": 0,
  "failed": 0
 },
 "hits": {
  "total": {
   "value": 5,
   "relation": "eq"
  },
  "max_score": null,
  "hits": []
 },
 "aggregations": {
  "countPerRegion": {
   "doc_count_error_upper_bound": 0,
   "sum_other_doc_count": 0,
   "buckets": [
    {
     "key": "france",
     "doc_count": 3
    },
    {
     "key": "spain",
     "doc_count": 2
    }
   ]
  }
 }
}
```

Just like with regular queries, we get some meta-data and a report how many records are in the result-set. However, because we specified `size=0`, we get no records back (the `hits` array is empty), we only get the results of the aggregation request.

Looking at the `aggregations` section in the result, we see the aggregation we requested (countPerRegion). Sometimes, Elasticsearch will approximate the results, because giving back exact results would take too much time. The values for `sum_other_doc_count` and `doc_count_error_upper_bound` indicate to what extent the results are complete. In our case, all records are included, so our results are exact. Subsequently, we see an array with buckets. There we can find our results. The `doc_count` fields indicate that there are three records for France, and two for Spain, similar to our table in section 6.2.

Let's extend this to a more interesting request:

```
from elasticsearch import Elasticsearch
import json

es =Elasticsearch([{'host':'127.0.0.1', 'port': 9200}])

search_body = {
    "size": 0,
    "aggs": {
        "countPerType": {
            "terms": {
                "field" : "type.keyword"
            },
            "aggs": {
                "averagePrice": {
                    "avg": {"field": "price"}
                }
            }
        }
    }
}

result = es.search(index="products", body=search_body)
print(json.dumps(result, indent=1))
```

Here, we ask Elasticsearch to create buckets for the field `type`, and calculate the average price for records in the bucket. The place where you put the second `aggs` is important here! It is put within the first `aggs`, indicating that the calculation should be done separately for each bucket of type.keyword. Indeed, what we get is (skipping the meta-data, which is the same as with the previous query):

```
 "aggregations": {
  "countPerType": {
   "doc_count_error_upper_bound": 0,
   "sum_other_doc_count": 0,
   "buckets": [
    {
     "key": "socks",
     "doc_count": 2,
     "averagePrice": {
      "value": 6.0
     }
    },
    {
     "key": "trousers",
     "doc_count": 2,
     "averagePrice": {
      "value": 7.5
     }
    },
    {
     "key": "jacket",
     "doc_count": 1,
     "averagePrice": {
      "value": 25.0
     }
    }
   ]
  }
 }
```

We now not only get the `doc_count`, but also the requested calculation of the average price per bucket.

The above examples use terms for buckets. Elasticsearch supports various kinds of criteria to create buckets, they are listed at:

https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket.html.

Some useful examples are:

- Range based buckets. You can create buckets for numerical fields, e.g., buckets for price<100, prices between 100 and 200, prices between 200 and 300, and prices above 300. You can manually specify these buckets, or you can let Elasticsearch determine the buckets for you using a so called histogram aggregation.
- Date/time based buckets. In Elasticsearch, you can ask for aggregating on various kinds of time slices. These bucket aggregations are really useful for data science, because they can be used for time series analysis. For example, you can create buckets per day, per hour, per month, per week, etc. Again, each bucket can report the number of records falling into that bucket, but also metrics like average price. This allows you to easily investigate trends in time.

A last topic regarding buckets is the sub-aggregations we discussed in 6.2. Let's try to get the same results from Elasticsearch. What we need to do is to create an aggregation for type, and within that, an aggregation for region (now only giving the `search_body`, leaving out the rest of the program, you can add it yourself to run it):

```
search_body = {
  "size": 0,
  "aggs": {
    "countPerType": {
      "terms": {
        "field" : "type.keyword"
      },
      "aggs": {
        "countPerRegion": {
          "terms": {"field": "region.keyword"}
        }
      }
    }
  }
}
```

What is important here is that we have an `aggs` request within an `aggs` request. Essentially, we ask Elasticsearch to create buckets for `type`, and for each of these buckets, create buckets for `region`.

Indeed what we get as result is:

```
 "aggregations": {
  "countPerType": {
   "doc_count_error_upper_bound": 0,
   "sum_other_doc_count": 0,
   "buckets": [
    {
     "key": "socks",
     "doc_count": 2,
     "countPerRegion": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
       {
        "key": "france",
        "doc_count": 1
       },
       {
        "key": "spain",
        "doc_count": 1
       }
      ]
     }
    },
    {
     "key": "trousers",
     "doc_count": 2,
     "countPerRegion": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
       {
        "key": "france",
        "doc_count": 2
       }
      ]
     }
    },
    {
     "key": "jacket",
     "doc_count": 1,
     "countPerRegion": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
       {
        "key": "spain",
        "doc_count": 1
       }
      ]
     }
    }
   ]
  }
 }
```

At the highest level, you see buckets for socks, trousers, and jacket. Within each bucket, you get buckets for france and spain.

To extend on this, we can of course now also ask for the average price within each bucket:

```
search_body = {
    "size": 0,
    "aggs": {
        "countPerType": {
            "terms": {
                "field" : "type.keyword"
            },
            "aggs": {
                "countPerRegion": {
                    "terms": {"field": "region.keyword"},
                    "aggs": {
                        "averagePrice": {
                            "avg": {"field": "price"}
                        }
                    }
                }
            }
        }
    }
}
```

As you can see, we have now added a metric aggregation under the terms aggregation for region, which gives us:

```
"aggregations": {
  "countPerType": {
   "doc_count_error_upper_bound": 0,
   "sum_other_doc_count": 0,
   "buckets": [
    {
     "key": "socks",
     "doc_count": 2,
     "countPerRegion": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
       {
        "key": "france",
        "doc_count": 1,
        "averagePrice": {
         "value": 5.0
        }
       },
       {
        "key": "spain",
        "doc_count": 1,
        "averagePrice": {
         "value": 7.0
        }
       }
      ]
     }
    },
    {
     "key": "trousers",
     "doc_count": 2,
     "countPerRegion": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
       {
        "key": "france",
        "doc_count": 2,
        "averagePrice": {
         "value": 7.5
        }
       }
      ]
     }
    },
    {
     "key": "jacket",
     "doc_count": 1,
     "countPerRegion": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
       {
        "key": "spain",
        "doc_count": 1,
        "averagePrice": {
         "value": 25.0
        }
       }
      ]
     }
    }
   ]
  }
 }
```

The more complex your queries, the more elaborate the output. Although the output of Elasticsearch is quite easy to understand, it is still difficult to read for a human. However, the output is meant to be read by computer programs, not humans. Luckily, you already know how you can extract relevant information from a dictionary, and report it in more condensed form in your Python programs as tables, graphs, or even on geographic maps.

6.4 Practical assignment

In the previous section, we discussed that date/time buckets can be used to get the aggregate per day, month, etc. You will now do this for the taxi data imported in practical 4B.

1. To start, create an overview of the count per day. Use the `pickup_datetime` field.
   Tip: use a `date_histogram` aggregation instead of a `terms` aggregation, and use the `calendar_interval` parameter.
2. Second, extend your program to calculate the statistics of `total_amount` per day. Study each day. Do you see any illogical results?
3. Remember that your aggregates can be filtered by adding queries. So instead of removing illogical results from your source data, you can also filter them out at query time. So, add a query to your request so your aggregation does not use obvious faulty records anymore, and again calculate the statistics for `total_amount` per day.
   Tip: you can use a `range` query on numerical fields.
4. Lastly, look at the final assignment. In step 1, it shows how you can aggregate over the weekdays (Monday, Tuesday, etc.). Implement it for your taxi data to calculate the statistics per weekday.
   Tip: think about how you should change the code from the final assignment so it considers the `pickup_datetime` field
   Interpret the results. What would `"key": "6"` mean?