Amit Kulkarni Follow
Nov 9, 2022 · 9 min read · Listen

Sensitivity Analysis Using Python

Learn to apply sensitivity analysis for additional insights from the model

Source: Unsplash | Nathan-Dumlao

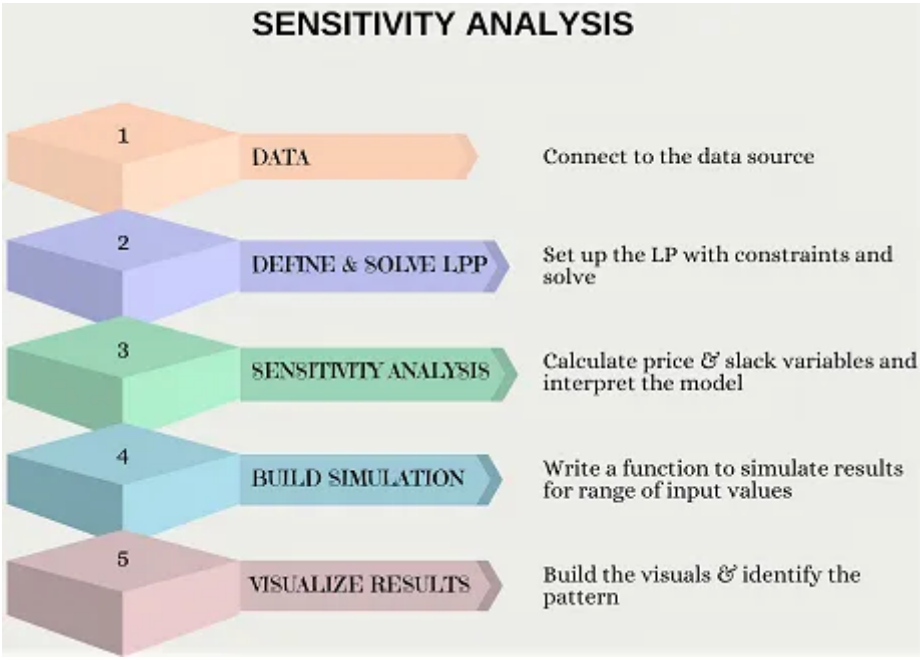
Introduction

In a real-world scenario, an outcome depends on multiple varying input variables and this makes it very difficult [illegible] or make a decision confidently. It is, even more, riskier if the outcome of the analysis has financial

implications. In order to handle such uncertainty in the data and to forecast with fair confidence, sensitivity and simulation analysis are carried out. The sensitivity analysis lets us identify the influential variables and assess the magnitude of the impact of each variable. This is widely used to make data-driven decisions and to test assumptions made during the analysis. The simulations are used to test the model's robustness and identify the threshold values beyond which the outcome will have a positive or negative impact.

In this article, we will define a problem and conduct a sensitivity analysis along with the simulation. We will be learning the below topics in this article

- Defining Linear Programming Problem (LPP)
- Using PuLP for solving the LPP
- What is sensitivity analysis?
- How to interpret the slack and price variables?
- Simulation with itertools and SensitivityAnalyzer
- Visualizing the results



Source: Author

Defining Linear Programming Problem (LPP)

A company manufactures two products — Product A and Product B. Our aim is to maximize profit under the three defined constraints.

Objective Function:

> **Profit = 30 * A + 45 * B**

Constraints:

- **Constraint 1:** 3 * A + 12 * B <= 150
- **Constraint 2:** 4 * A + 3 * B <= 47
- **Constraint 3:** 5 * A + 2 * B <= 60
- A, B >= 0

Using PuLP for solving the LPP

In this article, we will use PuLP which is an open-source package in python. Let's start by importing the libraries.

```
from pulp import *
```

We will translate the problem defined into the code.

Step 1: Initialize the Class. We will use the LpMaximize class for optimization

```
model = LpProblem("Product_Profits",LpMaximize)
```

Step 2: Defining variables

```
A = LpVariable('A', lowBound=0)
B = LpVariable('B', lowBound=0)
```

Step 3: Defining objective — profit on products A and B

```
model += 30 * A + 45 * B
```

Step 4: Define the constraints, we will add three constraints

```
# Constraint 1
model += 3 * A + 12 * B <= 150
# Constraint 2
model += 4 * A + 3 * B <= 47
# Constraint 3
model += 5 * A + 2 * B <= 60
```

Step 5: Solve to optimize

```
model.solve()
print("Model Status:{}".format(LpStatus[model.status]))
print("Objective = ", round(value(model.objective),3))
Result:
Model Status:Optimal
Objective =  617.307
```

Step 6: Let's check for the optimal value of A and B

```
for var in model.variables():
    print(var.name,"=", var.varValue)
Result:
A = 2.9230769
B = 11.769231
```

Understanding sensitivity analysis

We found an optimal solution for our Linear Problem. Here are a few scenarios we will find answers for.

- What will be the impact on the profit if the RHS of constraint 1 is changed from 150 to 160?
- What will be the impact on the profit if the RHS of constraint 2 is changed from 47 to 46 or 45?

- Are there values for constraint 3 where there is no impact on the profit? If yes, then what is the threshold value?

These questions (or similar questions) are important to get a better understanding of the model's performance. We will extract the price and slack values from the model.

```
res = [{'Name':name,'Constraint':const,'Price':const.pi,'Slack':
const.slack}
                for name, const in model.constraints.items()]
print(pd.DataFrame(res))
Result:
  Name     Constraint     Price      Slack
0  _C1  {A: 3, B: 12}  2.307692  -0.000000
1  _C2   {A: 4, B: 3}  5.769231  -0.000000
2  _C3   {A: 5, B: 2} -0.000000  21.846154
```

How to interpret slack and price values?

Price: The price value of constraints 1 is 2.30 and 2 is 5.76. This means if there is a unit change in the RHS of constraints 1 and constraint 2, it will impact the optimal value by 2.30 and 5.76.

Slack: The slack value of constraint 1 and constraint 2 is zero and for constraint 3 it is 21.84 i.e changes in the bandwidth of 21.84 will have no impact on the optimal value. Let's understand this with an example. We will change the RHS of constraint 3 from 60 to 55 and note the impact of this on the resulting value.

Here is the complete code so far.

```
from pulp import *

model = LpProblem("Product_Profits",LpMaximize)
# Define variables
A = LpVariable('A', lowBound=0)
B = LpVariable('B', lowBound=0)
# Define Objetive Function: Profit on Product A and B
model += 30 * A + 45 * B
# Constraint 1
model += 3 * A + 12 * B <= 150
# Constraint 2
model += 4 * A + 3 * B <= 47
# Constraint 3
model += 5 * A + 2 * B <= 55 # <-- the value was changed from 60 to
55
```

```
# Solve Model
model.solve()
print("Model Status:{}".format(LpStatus[model.status]))
print("Objective = ", round(value(model.objective),3))
```

```
Result:
Model Status: Optimal
Objective =  617.308
```

The optimal value of the function has not changed and it is still 617.308. Now let's change the constraint 3 value to 38.

```
Result:
Model Status: Optimal
Objective =  616.667
```

This time optimal value of the function has changed to 616.667. Similarly, for the values of **36**, **35**, and **34**, we get optimal values of **608.33**, **604.16**, and **600.00** respectively. If you have noticed the difference between the optimal values is 4.17. This means with every unit change in the slack value below 38 the optimal value of the function will change by **4.17**

Now, let's try the same with the price value. We will change the RHS of Constraint 1 for different values and keep other constraints the same.

```
# Constraint 1
model += 3 * A + 12 * B <= 149
```

```
Result: For 149
Model Status: Optimal
Objective =  615.000
```

```
Result: For 148
Model Status: Optimal
Objective =  612.692
```

```
Result: For 147
Model Status: Optimal
Objective =  610.385
```

Please note that the difference between the optimal value is 2.30. This means with every unit change in the price value of Constraint 1 below 150, the optimal value of the function will change by **2.30**

Now, We will change the RHS of Constraint 2 for different values and keep other constraints the same.

```
# Constraint 2
model += 4 * A + 3 * B <= 47
```

```
Result: For 46
Model Status: Optimal
Objective =  611.538
```

```
Result: For 45
Model Status: Optimal
Objective =  605.769
```

```
Result: For 44
Model Status: Optimal
Objective =  600.000
```

Please note that the difference between the optimal value is 5.76. This means with every unit change in the price value of Constraint 2 below 47, the optimal value of the function will change by **5.76**

Building a simulation

We have experimented with different values for constraints and noted our observations but this is a tedious process if the number of constraints increases. It will be good to have combinations of constraints and resulting optimal values in a table for ready reference. We will achieve this by wrapping our logic in a function followed by simulating RHS values for each of the constraints and generation of optimal values. Here is the complete code.

```
lst_constraint1 = list(range(140,151,1))
lst_constraint2 = list(range(45,48,1))
lst_constraint3 = list(range(35,39,1))
def sensitivity_table(lst_constraint1, lst_constraint2,
lst_constraint3):

    """_summary_
    Args:
        L1 (List): Range of values of constraint 1
        L2 (List): Range of values of constraint 2
        L3 (List): Range of values of constraint 3

Returns:
        Int: Returns the optimal value i.e profit
    """
```

```
    try:
        # Initialize Class, Define Vars., and Objective
        model = LpProblem("Product_Profits",LpMaximize)

# Define variables
        A = LpVariable('A', lowBound=0)
        B = LpVariable('B', lowBound=0)

# Define Objetive Function: Profit on Product A and B
        model += 30 * A + 45 * B

# Constraint 1
        model += 3 * A + 12 * B <= lst_constraint1

# Constraint 2
        model += 4 * A + 3 * B <= lst_constraint2

# Constraint 3
        model += 5 * A + 2 * B <= lst_constraint3

# Solve Model
        model.solve()

print("Model Status:{}".format(LpStatus[model.status]))
        print("Objective = ", round(value(model.objective),3))

        for var in model.variables():
            print(var.name,"=", var.varValue)
            print(f'"lst_constraint1" = {lst_constraint1},
"lst_constraint2" = {lst_constraint2}, "lst_constraint3" =
{lst_constraint1}')
        res =
[{'Name':name,'Constraint':const,'Price':const.pi,'Slack':
const.slack} for name, const in model.constraints.items()]
        print(pd.DataFrame(res))
        return round(value(model.objective),2)

    except Exception as e:
        print(f'Simulation error: {e}')

res = [(p3, p2, p1, sensitivity_table(p1, p2, p3)) for p3 in
lst_constraint3 for p2 in lst_constraint2 for p1 in lst_constraint1]

df = pd.DataFrame(res, columns= ['Constraint 3','Constraint 2',
'Constraint 1', 'Objective'])
df_pivot = df.pivot(index = 'Constraint 1', columns = ['Constraint
2', 'Constraint 3'], values = 'Objective')
df_pivot
```

Constraint 2	45	46	47	45	46	47	45	46	47	45	46	47	45	46	47
Constraint 3	35	35	35	36	36	36	37	37	37	38	38	38	39	39	39
Constraint 1															
140	573.61	573.61	573.61	577.78	577.78	577.78	581.94	581.94	581.94	582.69	586.11	586.11	582.69	588.46	590.28
141	576.67	576.67	576.67	580.83	580.83	580.83	585.00	585.00	585.00	585.00	589.17	589.17	585.00	590.77	593.33
142	579.72	579.72	579.72	583.89	583.89	583.89	587.31	588.06	588.06	587.31	592.22	592.22	587.31	593.08	596.39
143	582.78	582.78	582.78	586.94	586.94	586.94	589.62	591.11	591.11	589.62	595.28	595.28	589.62	595.38	599.44
144	585.83	585.83	585.83	590.00	590.00	590.00	591.92	594.17	594.17	591.92	597.69	598.33	591.92	597.69	602.50
145	588.89	588.89	588.89	593.06	593.06	593.06	594.23	597.22	597.22	594.23	600.00	601.39	594.23	600.00	605.56
146	591.94	591.94	591.94	596.11	596.11	596.11	596.54	600.28	600.28	596.54	602.31	604.44	596.54	602.31	608.08
147	595.00	595.00	595.00	598.85	599.17	599.17	598.85	603.33	603.33	598.85	604.62	607.50	598.85	604.62	610.38
148	598.06	598.06	598.06	601.15	602.22	602.22	601.15	606.39	606.39	601.15	606.92	610.56	601.15	606.92	612.69
149	601.11	601.11	601.11	603.46	605.28	605.28	603.46	609.23	609.44	603.46	609.23	613.61	603.46	609.23	615.00
150	604.17	604.17	604.17	605.77	608.33	608.33	605.77	611.54	612.50	605.77	611.54	616.67	605.77	611.54	617.31

Source: Author

The above table helps us get values from the combinations of all three constraints we have defined. Here are a few observations.

- If the value of constraint 1 is **150**, constraint 2 is **47**, and constraint 3 is **39**. The result optimal value is **617.31**. Any value below these values will have an impact on the optimal value.
- We saw earlier that a unit change in the value of constraint 3 would have an impact on the value by **4.17** i.e 616.67–612.50 = 4.17 (Yellow boxes).
- Similarly, constraint 2 would have an impact on the value by **5.76** (Green Boxes), and constraint 1 by **2.30** (Red Boxes)

Simulation with itertools

We did quite a bit of coding in the previous section with nested *for loops* and iterations to generate the simulation table.

```
[(p3, p2, p1, sense(p1, p2, p3)) for p3 in lst_constraint3
                for p2 in lst_constraint2
                        for p1 in lst_constraint1]
```

We had three nested for loops with three constraints and this can become more complex if the number of constraints increases. In this section, we will try to arrive at the same result in a much easier way by using the itertools library. We will use the product function from itertools which will give us all the possible combinations of constraints and optimal values without the hassle of nested loops

```
import itertools
...
[(var[0], var[1], var[2], sense(var[0], var[1], var[2]))
    for var in itertools.product(lst_constraint1, lst_constraint2,
lst_constraint3)]
```

```
[(140, 45, 35, 573.61),
 (140, 45, 36, 577.78),
 (140, 45, 37, 581.94),
 (140, 45, 38, 582.69),
 (140, 45, 39, 582.69)
 ............
 (150, 47, 35, 604.17),
 (150, 47, 36, 608.33),
 (150, 47, 37, 612.5),
 (150, 47, 38, 616.67),
 (150, 47, 39, 617.31)]
```

That came out pretty neat. We can use this list and generate the pivot table.

```
df_res_list = pd.DataFrame(res_list,
                  columns= ['Constraint 3','Constraint 2',
'Constraint 1', 'Objective'])
df_res_list_pivot = df.pivot(index = 'Constraint 1',
                  columns = ['Constraint 2', 'Constraint 3'], values
= 'Objective')
```

Simulation with SenstivityAnalyzer

We simplified our code to arrive at the same result but we still had to iterate and extract values. Now, let us leverage the power of the SensitivityAnalayzer library and simplify the process further. We will create a dictionary of constraints and let the library do all the heavy lifting for us.

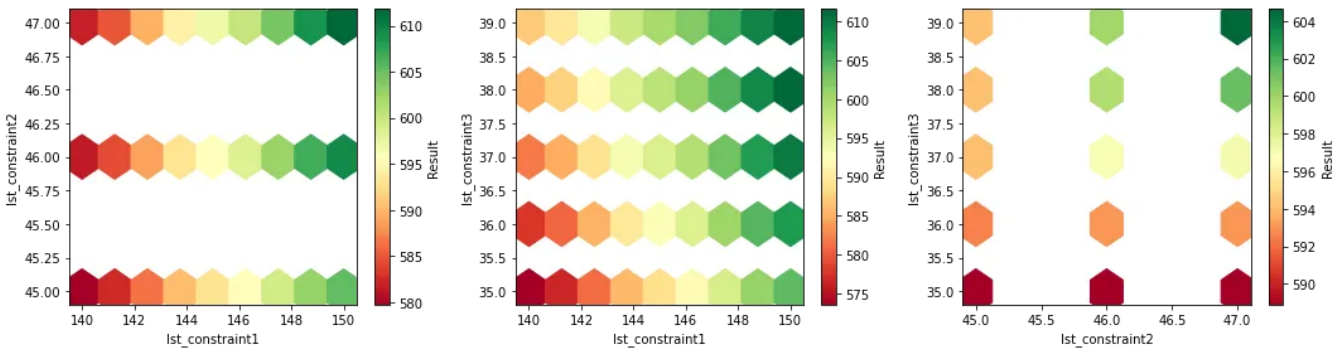
```
from sensitivity import SensitivityAnalyzer
# creating a dictionary of constraints
sensitivity_dict = {
    'lst_constraint1' : lst_constraint1,
    'lst_constraint2' : lst_constraint2,
    'lst_constraint3' : lst_constraint3
}
sa = SensitivityAnalyzer(sensitivity_dict, sensitivity_table)
sa.df
```

```
Result:
lst_constraint1  lst_constraint2  lst_constraint3  Result
```

```
140      45       35       573.610000
140      45       36       577.780000
140      45       37       581.940000
140      45       38       582.690000
140      45       39       582.690000
140      46       35       573.6100
..................................
141      47       35       576.670000
141      47       36       580.830000
141      47       37       585.000000
141      47       38       589.170000
141      47       39       593.330000
142      45       35       579.720000
142      45       36       583.890000
142      45       37       587.310000
```

We can use this list and generate the pivot table. Let us take it a notch higher and visualize the results with just one line of code.

```
plot = sa.plot()
```



Source: Author

Conclusion

The aim of the article was to introduce sensitivity analysis with python, the code and approach can be a template and extended to optimize specific problems. In this article, we started with understanding the sensitivity, followed by defining a Linear Programming Problem (LPP) with constraints that we optimized using the PuLP library. We then deep-dived into exploring the price and slack variable, and its role and analyzed the impact of changes to variables on the optimal value of the function. We then explored ways to generate simulation tables with the python libraries — *itertools* and *SensitivityAnalyzer*.

Some practical applications of sensitivity analysis

- Price optimization of products
- Identify the assumptions and check for tolerance
- Effectively manage resource allocation eg: Manufacturing plants
- Used extensively in the financial industry by economists eg: Portfolio optimization

I hope you liked the article and found it helpful.

You can connect with me — Linkedin

You can find the code for reference — Github

References

https://pypi.org/project/PuLP/

https://pypi.org/project/sensitivity/

https://unsplash.com/