Prices Optimization and Product Mix

April 14, 2023

1 Prices Optimization and Product Mix

[1]:
```
# Python Language Version
from platform import python_version
print('Python Language Version Used in This Jupyter Notebook:',␣
  ↪python_version())
```

```
Python Language Version Used in This Jupyter Notebook: 3.7.16
```

1.1 Defining the Problem

Smart Moon Tech assembles and tests two smartphone models, Moon1 and Moon2. For the next month, the company wants to decide how many units of each model it will assemble and then test.

No smartphone is in stock since the previous month and as these models will be replaced after this month, the company doesn't want to keep any stock for the following month.

They believe that the most they can sell this month is 600 Moon1 units and 1200 Moon2 units.

Each Moon1 model is sold for 300 and each Moon2 model for 450. The component cost for a Moon1 is 150 and for a Moon2 it is 225.

Labor is required for assembly and testing. There are a maximum of 10,000 hours of assembly and 3,000 hours of testing available. Each working hour for assembly costs 11 and each working hour for testing costs 15. Each Moon1 requires five hours for assembly and one hour for testing. Each Moon2 requires six hours for assembly and two hours for testing.

Smart Moon Tech wants to know how many units of each model it should produce (build and test) to maximize its net profit, but it can't use more man hours than it has available and it doesn't want to produce more than it can sell.

We will optimize prices and mix of Smart Moon Tech products.

1.2 Loading Packages

[2]:
```
# Imports
from pulp import *
```

1.3 Creating the Mathematical Model for the Optimization

1.3.1 Parameters

- Ai = Maximum number of type i smartphones to sell this month, where i belongs to the set {Moon1, Moon2}
- Bi = Sale price of smartphones model type i, where i belongs to the set {Moon1, Moon2}
- Ci = Cost price of component parts for type i smartphones, where i belongs to the set {Moon1, Moon2}
- Di = Assembly labor cost per hour of type i smartphones, where i belongs to the set {Moon1, Moon2}
- Ei = Cost of test labor per hour of smartphones model type i, where i belongs to the set {Moon1, Moon2}
- F = Maximum number of hours of assembly work
- G = Maximum number of hours of test work
- Hf,i = Hours of assembly required to build each model of smartphone type i, where i belongs to the set {Moon1, Moon2}
- Hg,i = Hours of testing required to test each smartphone model type i, where i belongs to the set {Moon1, Moon2}

1.3.2 Decision Variable

- Xi = Type i smartphone numbers to be produced this month, where i belongs to the set {Moon1, Moon2}

1.3.3 Restrictions

- The number of type i smartphones to be produced cannot be negative, that is, Xi >= 0, where i belongs to the set {Moon1, Moon2}.

- The total number of assembly hours cannot be greater than the maximum number of assembly labor hours available.

- The total number of test hours cannot be greater than the maximum test manpower hours available.

- The number of type i smartphones to be produced cannot be greater than the maximum number of type i smartphones to be sold in this month, where i belongs to the set {Moon1, Moon2}.

1.4 Mathematical Model Implementing

1.4.1 Parameters Organizing

[3]:
```
# Maximum number of smartphones to sell this month
A_Moon1 = 600
A_Moon2 = 1200
```

[4]:
```
# Smartphone selling price
B_Moon1 = 300
B_Moon2 = 450
```

[5]:
```
# Cost price of component parts for smartphones
C_Moon1 = 150
C_Moon2 = 225
```

[6]:
```
# Assembly labor cost per hour of smartphones
D_Moon1 = 11
D_Moon2 = 11
```

[7]:
```
# Testing labor cost per hour of smartphones
E_Moon1 = 15
E_Moon2 = 15
```

[8]:
```
# Maximum number of hours of assembly work
F = 10000
```

[9]:
```
# Maximum number of hours of test work
G = 3000
```

[10]:
```
# Hours of assembly required to build each smartphone model
Hfi_Moon1 = 5
Hfi_Moon2 = 6
```

[11]:
```
# Hours of testing required to test each smartphone model
Hgi_Moon1 = 1
Hgi_Moon2 = 2
```

1.4.2 Creating the Variable for the Optimization Problem

[12]:
```
help(LpProblem)
```

```
Help on class LpProblem in module pulp.pulp:

class LpProblem(builtins.object)
 |  LpProblem(name='NoName', sense=1)
 |
 |  An LP Problem
 |
 |  Methods defined here:
 |
 |  __getstate__(self)
 |
 |  __iadd__(self, other)
 |
 |  __init__(self, name='NoName', sense=1)
 |      Creates an LP Problem
 |
 |      This function creates a new LP Problem  with the specified associated
parameters
```

```
 |
 |      :param name: name of the problem used in the output .lp file
 |      :param sense: of the LP problem objective.                  Either
:data:`~pulp.const.LpMinimize` (default)                 or
:data:`~pulp.const.LpMaximize`.
 |      :return: An LP Problem
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  add(self, constraint, name=None)
 |
 |  addConstraint(self, constraint, name=None)
 |
 |  addVariable(self, variable)
 |      Adds a variable to the problem before a constraint is added
 |
 |      @param variable: the variable to be added
 |
 |  addVariables(self, variables)
 |      Adds variables to the problem before a constraint is added
 |
 |      @param variables: the variables to be added
 |
 |  assignConsPi(self, values)
 |
 |  assignConsSlack(self, values, activity=False)
 |
 |  assignStatus(self, status, sol_status=None)
 |      Sets the status of the model after solving.
 |      :param status: code for the status of the model
 |      :param sol_status: code for the status of the solution
 |      :return:
 |
 |  assignVarsDj(self, values)
 |
 |  assignVarsVals(self, values)
 |
 |  checkDuplicateVars(self)
 |      Checks if there are at least two variables with the same name
 |      :return: 1
 |      :raises `const.PulpError`: if there ar duplicates
 |
 |  checkLengthVars(self, max_length)
 |      Checks if variables have names smaller than `max_length`
 |      :param int max_length: max size for variable name
 |      :return:
 |      :raises const.PulpError: if there is at least one variable that has a
long name
 |
 |  coefficients(self, translation=None)
 |
 |  copy(self)
 |      Make a copy of self. Expressions are copied by reference
 |
 |  deepcopy(self)
 |      Make a copy of self. Expressions are copied by value
 |
 |  extend(self, other, use_objective=True)
 |      extends an LpProblem by adding constraints either from a dictionary
 |      a tuple or another LpProblem object.
 |
 |      @param use_objective: determines whether the objective is imported from
 |      the other problem
 |
 |      For dictionaries the constraints will be named with the keys
 |      For tuples an unique name will be generated
 |      For LpProblems the name of the problem will be added to the constraints
 |      name
 |
 |  fixObjective(self)
 |
 |  getSense(self)
 |
 |  get_dummyVar(self)
 |
 |  infeasibilityGap(self, mip=1)
 |
 |  isMIP(self)
 |
 |  normalisedNames(self)
 |
 |  numConstraints(self)
 |      :return: number of constraints in model
 |
 |  numVariables(self)
 |      :return: number of variables in model
 |
 |  resolve(self, solver=None, **kwargs)
 |      resolves an Problem using the same solver as previously
 |
 |  restoreObjective(self, wasNone, dummyVar)
 |
 |  roundSolution(self, epsInt=1e-05, eps=1e-07)
 |      Rounds the lp variables
 |
 |      Inputs:
 |          - none
 |
 |      Side Effects:
 |          - The lp variables are rounded
 |
 |  sequentialSolve(self, objectives, absoluteTols=None, relativeTols=None,
solver=None, debug=False)
 |      Solve the given Lp problem with several objective functions.
 |
 |      This function sequentially changes the objective of the problem
 |      and then adds the objective function as a constraint
 |
 |      :param objectives: the list of objectives to be used to solve the
problem
 |      :param absoluteTols: the list of absolute tolerances to be applied to
 |         the constraints should be +ve for a minimise objective
 |      :param relativeTols: the list of relative tolerances applied to the
constraints
 |      :param solver: the specific solver to be used, defaults to the default
solver.
 |
 |  setObjective(self, obj)
 |      Sets the input variable as the objective function. Used in Columnwise
Modelling
 |
 |      :param obj: the objective function of type :class:`LpConstraintVar`
 |
 |      Side Effects:
 |          - The objective function is set
 |
 |  setSolver(self, solver=<pulp.apis.coin_api.PULP_CBC_CMD object at
0x000001A49CE88208>)
 |      Sets the Solver for this problem useful if you are using
 |      resolve
 |
 |  solve(self, solver=None, **kwargs)
 |      Solve the given Lp problem.
 |
 |      This function changes the problem to make it suitable for solving
 |      then calls the solver.actualSolve() method to find the solution
 |
 |      :param solver:  Optional: the specific solver to be used, defaults to
the
 |            default solver.
 |
 |      Side Effects:
 |          - The attributes of the problem object are changed in
 |            :meth:`~pulp.solver.LpSolver.actualSolve()` to reflect the Lp
solution
 |
 |  startClock(self)
 |      initializes properties with the current time
 |
 |  stopClock(self)
 |      updates time wall time and cpu time
 |
 |  toDict(self)
 |      creates a dictionary from the model with as much data as possible.
 |      It replaces variables by variable names.
 |      So it requires to have unique names for variables.
 |
 |      :return: dictionary with model data
 |      :rtype: dict
 |
 |  toJson(self, filename, *args, **kwargs)
 |      Creates a json file from the LpProblem information
 |
 |      :param str filename: filename to write json
 |      :param args: additional arguments for json function
 |      :param kwargs: additional keyword arguments for json function
 |      :return: None
 |
 |  to_dict = toDict(self)
 |
 |  to_json = toJson(self, filename, *args, **kwargs)
 |
 |  unusedConstraintName(self)
 |
 |  valid(self, eps=0)
 |
 |  variables(self)
 |      Returns the problem variables
 |
 |      :return: A list containing the problem variables
 |      :rtype: (list, :py:class:`LpVariable`)
 |
 |  variablesDict(self)
 |
 |  writeLP(self, filename, writeSOS=1, mip=1, max_length=100)
 |      Write the given Lp problem to a .lp file.
 |
 |      This function writes the specifications (objective function,
 |      constraints, variables) of the defined Lp problem to a file.
 |
 |      :param str filename: the name of the file to be created.
 |      :return: variables
 |      Side Effects:
 |          - The file is created
 |
 |  writeMPS(self, filename, mpsSense=0, rename=0, mip=1)
 |      Writes an mps files from the problem information
 |
 |      :param str filename: name of the file to write
 |      :param int mpsSense:
 |      :param bool rename: if True, normalized names are used for variables and
constraints
 |      :param mip: variables and variable renames
 |      :return:
 |      Side Effects:
 |          - The file is created
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  fromDict(_dict) from builtins.type
 |      Takes a dictionary with all necessary information to build a model.
 |      And returns a dictionary of variables and a problem object
 |
 |      :param _dict: dictionary with the model stored
 |      :return: a tuple with a dictionary of variables and a
:py:class:`LpProblem`
 |
 |  fromJson(filename) from builtins.type
 |      Creates a new Lp Problem from a json file with information
 |
 |      :param str filename: json file name
 |      :return: a tuple with a dictionary of variables and an LpProblem
 |      :rtype: (dict, :py:class:`LpProblem`)
 |
 |  fromMPS(filename, sense=1, **kwargs) from builtins.type
 |
 |  from_dict = fromDict(_dict) from builtins.type
 |      Takes a dictionary with all necessary information to build a model.
 |      And returns a dictionary of variables and a problem object
 |
 |      :param _dict: dictionary with the model stored
 |      :return: a tuple with a dictionary of variables and a
:py:class:`LpProblem`
 |
 |  from_json = fromJson(filename) from builtins.type
 |      Creates a new Lp Problem from a json file with information
 |
 |      :param str filename: json file name
 |      :return: a tuple with a dictionary of variables and an LpProblem
 |      :rtype: (dict, :py:class:`LpProblem`)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

[13]:
```
# Variable for the problem
problem = LpProblem("ProductMix", LpMaximize)
```

[14]:
```
# Visualize
problem
```

[14]:
```
ProductMix:
MAXIMIZE
None
VARIABLES
```

1.4.3 Defining the Decision Variable for Each Smartphone Model

[15]:
```
help(LpVariable)
```

```
Help on class LpVariable in module pulp.pulp:

class LpVariable(LpElement)
 |  LpVariable(name, lowBound=None, upBound=None, cat='Continuous', e=None)
 |
 |  This class models an LP Variable with the specified associated parameters
 |
 |  :param name: The name of the variable used in the output .lp file
 |  :param lowBound: The lower bound on this variable's range.
 |      Default is negative infinity
 |  :param upBound: The upper bound on this variable's range.
 |      Default is positive infinity
 |  :param cat: The category this variable is in, Integer, Binary or
 |      Continuous(default)
 |  :param e: Used for column based modelling: relates to the variable's
 |      existence in the objective function and constraints
 |
 |  Method resolution order:
```

```
|      LpVariable
|      LpElement
|      builtins.object
|
|  Methods defined here:
|
|  __init__(self, name, lowBound=None, upBound=None, cat='Continuous', e=None)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __ne__(self, other)
|      Return self!=value.
|
|  addVariableToConstraints(self, e)
|      adds a variable to the constraints indicated by
|      the LpConstraintVars in e
|
|  add_expression(self, e)
|
|  asCplexLpAffineExpression(self, name, constant=1)
|
|  asCplexLpVariable(self)
|
|  bounds(self, low, up)
|
|  fixValue(self)
|      changes lower bound and upper bound to the initial value if exists.
|      :return: None
|
|  getLb(self)
|
|  getUb(self)
|
|  infeasibilityGap(self, mip=1)
|
|  isBinary(self)
|
|  isConstant(self)
|
|  isFixed(self)
|      :return: True if upBound and lowBound are the same
|      :rtype: bool
|
|  isFree(self)
|
|  isInteger(self)
|
|  isPositive(self)
|
 |  positive(self)
 |
 |  round(self, epsInt=1e-05, eps=1e-07)
 |
 |  roundedValue(self, eps=1e-05)
 |
 |  setInitialValue(self, val, check=True)
 |      sets the initial value of the variable to `val`
 |      May be used for warmStart a solver, if supported by the solver
 |
 |      :param float val: value to set to variable
 |      :param bool check: if True, we check if the value fits inside the
variable bounds
 |      :return: True if the value was set
 |      :raises ValueError: if check=True and the value does not fit inside the
bounds
 |
 |  toDict(self)
 |      Exports a variable into a dictionary with its relevant information
 |
 |      :return: a dictionary with the variable information
 |      :rtype: dict
 |
 |  to_dict = toDict(self)
 |
 |  unfixValue(self)
 |
 |  valid(self, eps)
 |
 |  value(self)
 |
 |  valueOrDefault(self)
 |
 |  ----------------------------------------------------------------------
 |  Class methods defined here:
 |
 |  dict(name, indices, lowBound=None, upBound=None, cat='Continuous') from
builtins.type
 |
 |  dicts(name, indices=None, lowBound=None, upBound=None, cat='Continuous',
indexStart=[], indexs=None) from builtins.type
 |      This function creates a dictionary of :py:class:`LpVariable` with the
specified associated parameters.
 |
 |      :param name: The prefix to the name of each LP variable created
 |      :param indices: A list of strings of the keys to the dictionary of LP
 |          variables, and the main part of the variable name itself
 |      :param lowBound: The lower bound on these variables' range. Default is
 |          negative infinity
 |      :param upBound: The upper bound on these variables' range. Default is
 |          positive infinity
 |      :param cat: The category these variables are in, Integer or
 |          Continuous(default)
 |      :param indexs: (deprecated) Replaced with `indices` parameter
 |
 |      :return: A dictionary of :py:class:`LpVariable`
 |
 |  fromDict(dj=None, varValue=None, **kwargs) from builtins.type
 |      Initializes a variable object from information that comes from a
dictionary (kwargs)
 |
 |      :param dj: shadow price of the variable
 |      :param float varValue: the value to set the variable
 |      :param kwargs: arguments to initialize the variable
 |      :return: a :py:class:`LpVariable`
 |      :rtype: :LpVariable
 |
 |  from_dict = fromDict(dj=None, varValue=None, **kwargs) from builtins.type
 |      Initializes a variable object from information that comes from a
dictionary (kwargs)
 |
 |      :param dj: shadow price of the variable
 |      :param float varValue: the value to set the variable
 |      :param kwargs: arguments to initialize the variable
 |      :return: a :py:class:`LpVariable`
 |      :rtype: :LpVariable
 |
 |  matrix(name, indices=None, lowBound=None, upBound=None, cat='Continuous',
indexStart=[], indexs=None) from builtins.type
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from LpElement:
 |
 |  __add__(self, other)
 |
 |  __bool__(self)
 |
 |  __div__(self, other)
 |
 |  __eq__(self, other)
 |      Return self==value.
 |
 |  __ge__(self, other)
 |      Return self>=value.
 |
 |  __hash__(self)
 |      Return hash(self).
 |
 |  __le__(self, other)
 |      Return self<=value.
 |
 |  __mul__(self, other)
 |
 |  __neg__(self)
 |
 |  __pos__(self)
 |
 |  __radd__(self, other)
 |
 |  __rdiv__(self, other)
 |
 |  __repr__(self)
 |      Return repr(self).
 |
 |  __rmul__(self, other)
 |
 |  __rsub__(self, other)
 |
 |  __str__(self)
 |      Return str(self).
 |
 |  __sub__(self, other)
 |
 |  getName(self)
 |
 |  setName(self, name)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from LpElement:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  name
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes inherited from LpElement:
 |
 |  expression = re.compile('[\\-\\+\\[\\]\\ \\->/]')
 |
 |  illegal_chars = '-+[] ->/'
```

```
|
|  trans = {32: 95, 43: 95, 45: 95, 47: 95, 62: 95, 91: 95, 93: 95}
```

[16]:
```
# Define the variables
x_Moon1 = LpVariable("Moon1 Units", 0, None, LpInteger)
x_Moon2 = LpVariable("Moon2 Units", 0, None, LpInteger)
```

[17]:
```
# Print
print(x_Moon1)
print(x_Moon2)
```

```
Moon1_Units
Moon2_Units
```

1.4.4 Implementing the Objective Function

[18]:
```
# Revenue
revenue = (x_Moon1 * B_Moon1) + (x_Moon2 * B_Moon2)
revenue
```

[18]:
```
300*Moon1_Units + 450*Moon2_Units + 0
```

[19]:
```
# Assembly Cost
assembly_cost = (x_Moon1 * Hfi_Moon1 * D_Moon1) + (x_Moon2 * Hfi_Moon2 *␣
 ↪D_Moon2)
assembly_cost
```

[19]:
```
55*Moon1_Units + 66*Moon2_Units + 0
```

[20]:
```
# Test Cost
test_cost = (x_Moon1 * Hgi_Moon1 * E_Moon1) + (x_Moon2 * Hgi_Moon2 * E_Moon2)
test_cost
```

[20]:
```
15*Moon1_Units + 30*Moon2_Units + 0
```

[21]:
```
# Components Cost
components_cost = (x_Moon1 * C_Moon1) + (x_Moon2 * C_Moon2)
components_cost
```

[21]:
```
150*Moon1_Units + 225*Moon2_Units + 0
```

[22]:
```
# Profit = Revenue - Assembly Cost - Test Cost - Component Cost
problem += revenue - assembly_cost - test_cost - components_cost
problem
```

[22]:
```
ProductMix:
MAXIMIZE
80*Moon1_Units + 129*Moon2_Units + 0
```

```
VARIABLES
0 <= Moon1_Units Integer
0 <= Moon2_Units Integer
```

1.4.5 Restrictions

[23]:
```
# Maximum Number of Assembly Hours
problem += (x_Moon1 * Hfi_Moon1) + (x_Moon2 * Hfi_Moon2) <= F,"Maximum Number␣
 ↪of Assembly Hours"
```

[24]:
```
# Maximum Number of Test Hours
problem += (x_Moon1 * Hgi_Moon1) + (x_Moon2 * Hgi_Moon2) <= G,"Maximum Number␣
 ↪of Test Hours"
```

[25]:
```
# Production less than or equal to demand for the Moon1 model
problem += x_Moon1 <= A_Moon1,"Production less than or equal to demand for the␣
 ↪Moon1 model"
```

[26]:
```
# Production less than or equal to demand for the Moon2 model
problem += x_Moon2 <= A_Moon2,"Production less than or equal to demand for the␣
 ↪Moon2 model"
```

[27]:
```
# Final problem
problem
```

[27]:
```
ProductMix:
MAXIMIZE
80*Moon1_Units + 129*Moon2_Units + 0
SUBJECT TO
Maximum_Number_of_Assembly_Hours: 5 Moon1_Units + 6 Moon2_Units <= 10000

Maximum_Number_of_Test_Hours: Moon1_Units + 2 Moon2_Units <= 3000

Production_less_than_or_equal_to_demand_for_the_Moon1_model: Moon1_Units
 <= 600

Production_less_than_or_equal_to_demand_for_the_Moon2_model: Moon2_Units
 <= 1200

VARIABLES
0 <= Moon1_Units Integer
0 <= Moon2_Units Integer
```

1.4.6 Solving the Optimization problem

```
[28]: # Optimization
      problem.solve()
```

```
[28]: 1
```

```
[29]: # Maximized Profit
      print("Maximized Profit:", value(problem.objective))
```

```
Maximized Profit: 199600.0
```

```
[30]: # Number of Moon1 Model Units to be Produced
      print("Number of Moon1 Model Units to be Produced:", problem.variables()[0].
        ↪varValue)
```

```
Number of Moon1 Model Units to be Produced: 560.0
```

```
[31]: # Number of Moon2 Model Units to be Produced
      print("Number of Moon2 Model Units to be Produced:", problem.variables()[1].
        ↪varValue)
```

```
Number of Moon2 Model Units to be Produced: 1200.0
```

1.5 Conclusion

The company Moon Smart Tech must produce 560 Moon1 units and 1200 Moon2 units to reach the maximum profit of 199,600.

2 End