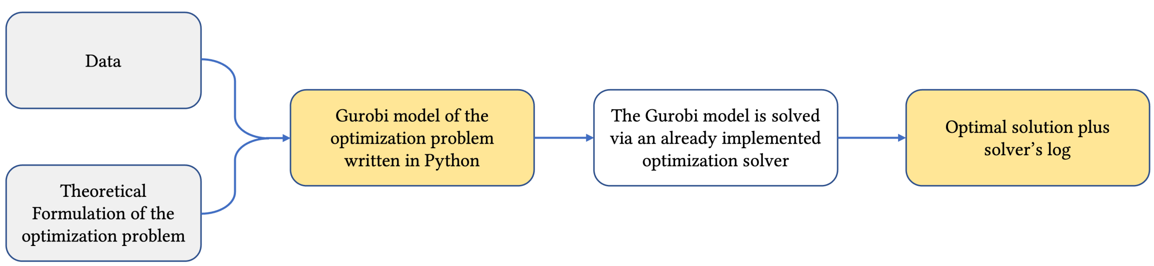
IDS 435 - Optimization via Gurobi (Part 2)

Spring 2022

Table of Contents

A Procedure to Model an Optimization Problem in Gurobi

The main task to use Gurobi in practice is to convert the mathematical formulation of an optimization problem into a Gurobi model that can be solved efficiently.



The Procedure

1. Sets and indices

   Use python `list`, `range`, `arrange`, etc to define index sets to count over decision variables and constraints.

2. Parameters (i.e., data)

   Data should be formatted in a way that is easy to define decision variables, objective function, and constraints in Gurobi. Data can be represented in different formats such as

   - Python lists
   - Python dictionaries
   - Numpy array

- Pandas `dataframe`
- Gurobi multidict

Since Numpy arrays are widely-used data structures in python, we employ them to represent data that later on used in a Gurobi model

3. Decision variables

    - Specify type of the variable (real-valued, binary, integer, etc)
    - Specify if the variable is signed or not (positive or negative)
    - Try to define put related decision variables of the model in an array of variables instead of defining them individually

```
# For loop
for i in range (N):
        model.addVar( ... )

# One line definition
model.addMvar( shape=N)
```

4. Constraints

    - Specify type of a constraint (linear, quadratic, etc)
    - Find an appropriate Gurobi function to model the constraint (oftentimes you can use `addConstrs`)
    - Try to define multiple constraints in a single line of code via python inline for loop

```
# For loop
for i in range (N):
        model.addConstr( ... )

 # Inline for loop (preferred)
model.addConstrs( ... for i in range (N))
```

5. Objective function

    - Define the objective function using data and decision variables
    - Specify if the problem is maximization or minimization

6. Optimize

    - Choose a solver
    - Specify parameters of the solver (i.e., stopping criteria, feasibility tolerance, etc)
    - Solve the Gurobi model

7. Analyze results (*Gurobi solved the model*)

    - Is model well-conditioned (i.e., no numerical issues encountered while optimization)?

- Is the model "normalized"?
- What is an optimal solution?
- What is the optimal value?
- How long did it take to solve the model?

8. Troubleshooting (*Gurobi could not solved the model*)

- Is the issue with the numerical errors?
- Is the issue with solver? Try a different optimization solver. Try to change the parameters of the solver (i.e., feasibility tolerance)?
- Double-check the type of variables and their signs as well as the definition of constraints Gurobi model?

A Toy Example

We use the following optimization problem to illustrate using Gurobi and aforementioned procedure for using Gurobi:

$$\begin{aligned}\min_{x_1,x_2} \quad & -x_1 - x_2 \\ & x_1 + 2x_2 \leq 1 \\ & 2x_1 + x_2 \leq 1 \\ & x_1, x_2 \geq 0\end{aligned}$$

In [ ]:

```
import gurobipy as gb
import numpy as np

if __name__ == "__main__":

    """         Step 1. Sets and indices            """
    num_var             = 2
    num_constr          = 2
    var_index           = range(num_var)
    constr_index        = range(num_constr)

    """         Step 2. Parameters                  """
    constr_matrix       = np.array([[1.,2.],
                                    [2.,1.]])
    rhs                 = np.array([1.,1.])


    """         Step 3. Decision variables          """

    model              = gb.Model('Toy Example')
    x                  = model.addMVar(
                                        shape     = num_var,
                                        name      = 'x',
                                        vtype     = gb.GRB.CONTINUOUS,
                                        lb        = 0.,
                                        ub        = gb.GRB.INFINITY )
```

```
"""         Step 4. Constraints                    """
model.addConstrs(gb.quicksum(constr_matrix[i][j]*x[j]  for j in var_inde

"""         Step 5. Objective function             """
model.setObjective(-gb.quicksum(x))

"""         Step 6. Optimize                       """
print('='*100)
model.setParam('Method',2)
model.setParam('Crossover',0)
model.update()
model.optimize()
print('='*100)

"""         Step 7. Analyze results                """
print('The optimal x_1:         \t',        x[0].X)
print('The optimal x_2:         \t',        x[1].X)
print('The optimal value is:    \t',        model.ObjVal)
print('='*100)
```

```
Set parameter Username
Academic license - for non-commercial use only - expires 2022-05-07
================================================================================
========================
Set parameter Method to value 2
Set parameter Crossover to value 0
Gurobi Optimizer version 9.5.1 build v9.5.1rc2 (mac64[rosetta2])
Thread count: 10 physical cores, 10 logical processors, using up to 10 threa
ds
Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xed33d8fe
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [1e+00, 1e+00]
Presolve time: 0.00s
Presolved: 2 rows, 2 columns, 4 nonzeros
Ordering time: 0.00s

Barrier statistics:
 AA' NZ     : 1.000e+00
 Factor NZ  : 3.000e+00
 Factor Ops : 5.000e+00 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0  -8.67927042e-01 -4.61538462e-01  1.51e-01 3.08e-01  2.86e-01     0s
   1  -6.05231787e-01 -6.96010401e-01  0.00e+00 0.00e+00  2.27e-02     0s
   2  -6.66536989e-01 -6.66799107e-01  0.00e+00 0.00e+00  6.55e-05     0s
   3  -6.66666537e-01 -6.66666799e-01  0.00e+00 0.00e+00  6.55e-08     0s
   4  -6.66666667e-01 -6.66666667e-01  0.00e+00 2.22e-16  6.55e-11     0s

Barrier solved model in 4 iterations and 0.01 seconds (0.00 work units)
Optimal objective -6.66666667e-01

================================================================================
========================
The optimal x_1:                 0.3333333332685205
The optimal x_2:                 0.3333333332685205
The optimal value is:            -0.666666666537041
================================================================================
========================
```

Simple Linear Regression

This application is motivated by curve fitting Gurobi example.

Consider a dataset of N points $\{(x^i, y^i) : i = 1, 2, \ldots, N\}$. We want to fit a linear model with intercept β_0 and slope β_1 to approximate response variable y^i using feature x^i. Specifically, we want the following constraints to hold as close as possible:

$$y^i \approx \beta_0 + \beta_1 x^i, \qquad \forall i = 1, 2, \dots, N.$$

For each i, $\beta_0 + \beta_1 x^i$ is the approximate value for y_i. Our objective is to compute β_0 and β such that an "error" is minimized. Below we discuss two metrics of error and implement them in Gurobi.

1. Mean Absolute Deviation (MAD; *linear objective function*):

$$\begin{aligned} \min_{\beta_0,\beta_1} \quad & \frac{1}{N}\sum_{i=1}^{N}(u_i + v_i) \\ & y^i = \beta_0 + \beta_1 x^i + u_i - v_i, && \forall i = 1, 2, \dots, N, \\ & u_i, v_i \geq 0, && \forall i = 1, 2, \dots, N, \\ & \beta_0, \beta_1 \ \text{ unrestricted.} \end{aligned}$$

2. Mean Squared Error (MSE; *quadratic objective function*):

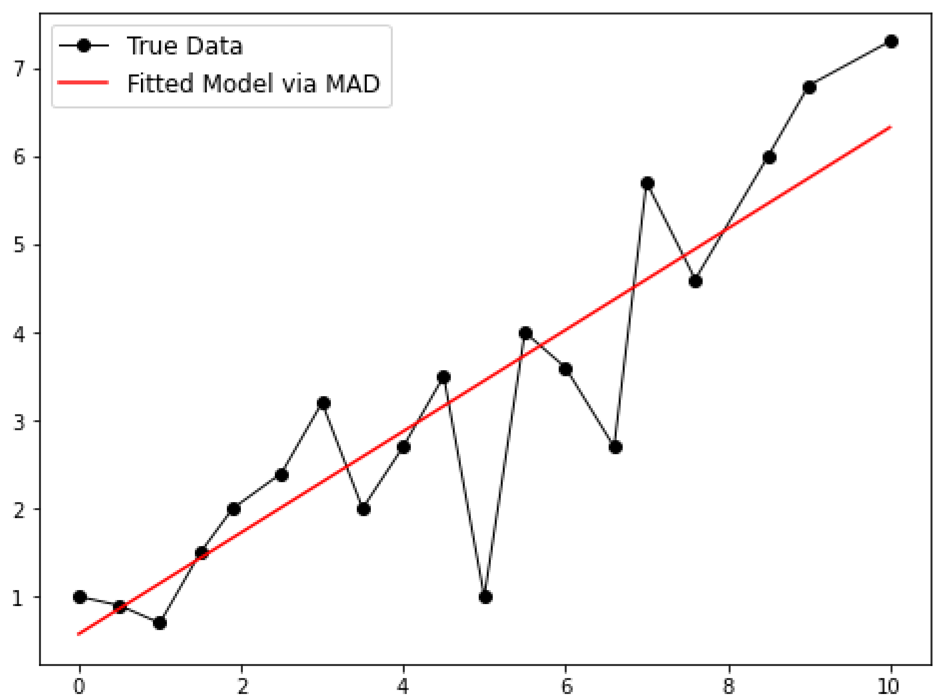
$$\begin{aligned} \min_{\beta_0,\beta_1} \quad & \frac{1}{N}\sum_{i=1}^{N}\varepsilon_i^2 \\ & \varepsilon_i = y^i - \beta_0 - \beta_1 x^i, && \forall i = 1, 2, \dots, N, \\ & \varepsilon_i \ \text{ unrestricted,} && \forall i = 1, 2, \dots, N, \\ & \beta_0, \beta_1 \ \text{ unrestricted.} \end{aligned}$$

Gurobi model for MAD minimization

In [ ]:
```
import numpy as np
import gurobi as gb
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

if __name__ == "__main__":

    """         Step 1. Sets and indices            """
    num_data        = 10
    data_index      = range(num_data)

    """         Step 2. Parameters                  """
    x_train         = np.array([0.,  0.5,    1,      1.5,    1.9,    2.5,
                                     4.5,    5.,  5.5,    6,      6.6,    7,
    
    y_train         = np.array([1.,  0.9,    .7,     1.5,    2.,     2.4,
                                     3.5,    1.,  4,      3.6,    2.7,    5.

    """         Step 3. Decision Variables           """
    model           = gb.Model('MAD')

    ## coef := (\beta_0,\beta)
    coef            = model.addMVar(
```

```
                        shape    = 2,
                        vtype    = gb.GRB.CONTINUOUS,
                        lb       = -gb.GRB.INFINITY,
                        ub       =  gb.GRB.INFINITY )

u               = model.addMVar(
                        shape    = num_data,
                        vtype    = gb.GRB.CONTINUOUS,
                        lb       = 0,
                        ub       = gb.GRB.INFINITY )

v               = model.addMVar(
                        shape    = num_data,
                        vtype    = gb.GRB.CONTINUOUS,
                        lb       = 0,
                        ub       = gb.GRB.INFINITY )

"""         Step 4. Constraints                 """
model.addConstrs(y_train[i] ==  coef[0] + coef[1]*x_train[i] + u[i] - v[

"""         Step 5. Objective function          """
model.setObjective(gb.quicksum(u) + gb.quicksum(v))

"""         Step 6. Optimize                    """
print('='*100)
model.update()
model.optimize()
print('='*100)

"""         Step 7. Analyze results             """
MAD_intercept, MAD_slope = coef[0].X,coef[1].X
print('The optimal intercept:\t', MAD_intercept)
print('The optimal slope:    \t', MAD_slope)
print('The optimal objective value: \t', model.ObjVal)
print('='*100)

fig = plt.figure(figsize=(8,6))
plt.plot(x_train,y_train,marker='o',lw=1,color='black',label='True Data'
plt.plot(x_train,[MAD_intercept + x*MAD_intercept for x in x_train],colo
plt.legend(fontsize=12)
plt.show()


print([np.round(u[i].X,4) for i in data_index])
print([np.round(v[i].X,4) for i in data_index])
```

```
===============================================================================
========================
Gurobi Optimizer version 9.5.1 build v9.5.1rc2 (mac64[rosetta2])
Thread count: 10 physical cores, 10 logical processors, using up to 10 threa
ds
Optimize a model with 10 rows, 22 columns and 39 nonzeros
Model fingerprint: 0x1a0d94ca
Coefficient statistics:
  Matrix range     [5e-01, 4e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [7e-01, 4e+00]
Presolve time: 0.00s
Presolved: 10 rows, 22 columns, 39 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0      handle free variables                          0s
      11    3.3900000e+00   0.000000e+00   0.000000e+00      0s

Solved in 11 iterations and 0.01 seconds (0.00 work units)
Optimal objective  3.390000000e+00
===============================================================================
========================
The optimal intercept:   0.575
The optimal slope:       0.65
The optimal objective value:     3.3899999999999997
===============================================================================
========================
```




```
[0.425, 0.0, 0.0, 0.0, 0.19, 0.2, 0.675, 0.0, 0.0, 0.0]
[0.0, 0.0, 0.525, 0.05, 0.0, 0.0, 0.0, 0.85, 0.475, 0.0]
```

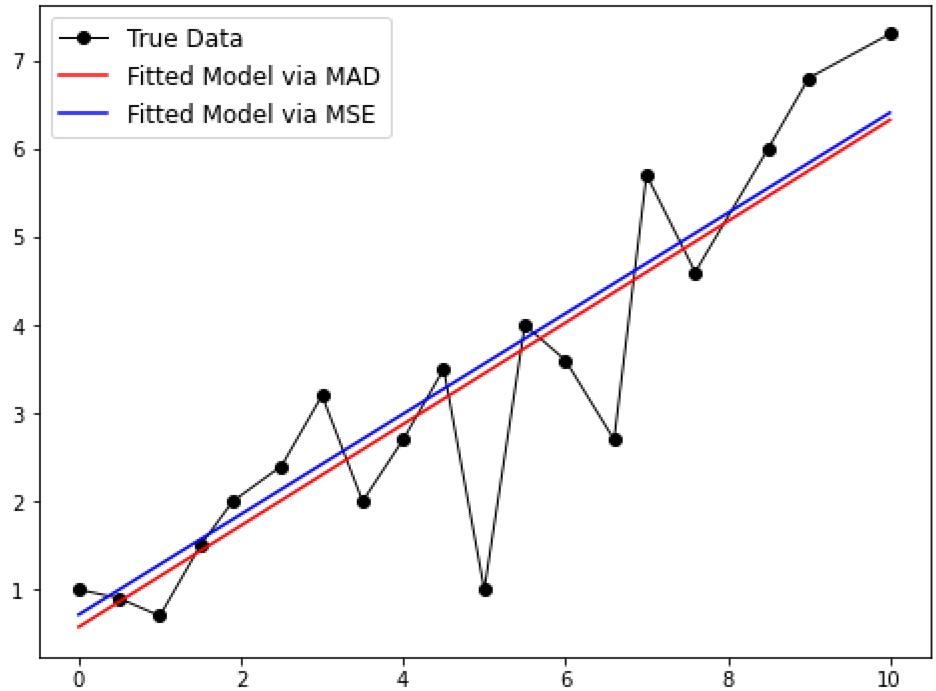
Gurobi model for MSE minimization

In [ ]:
```
import numpy as np
import gurobi as gb
```

```
import matplotlib.pyplot as plt

if __name__ == "__main__":



    """          Step 1. Sets and indices            """
    num_data         = 10
    data_index       = range(num_data)

    """          Step 2. Parameters                  """
    x_train          = np.array([0.,  0.5,    1,      1.5,    1.9,    2.5,
                                 4.5,    5.,  5.5,    6,      6.6,    7,

    y_train          = np.array([1.,  0.9,    .7,     1.5,    2.,     2.4,
                                 3.5,    1.,  4,      3.6,    2.7,    5.

    """          Step 3. Decision variables          """

    model            = gb.Model('MSE')

    ## coef := (\beta_0,\beta)
    coef             = model.addMVar(
                            shape    = 2,
                            vtype    = gb.GRB.CONTINUOUS,
                            lb       = -gb.GRB.INFINITY,
                            ub       =  gb.GRB.INFINITY )

    diff          = model.addMVar(
                            shape    = num_data,
                            vtype    = gb.GRB.CONTINUOUS,
                            lb       = -gb.GRB.INFINITY,
                            ub       = gb.GRB.INFINITY )

    """          Step 4. Constraints                 """
    model.addConstrs(diff[i] == y_train[i] - coef[0] - coef[1]*x_train[i]

    """          Step 5. Objective function          """
    model.setObjective(gb.quicksum(diff[i]*diff[i] for i in data_index))

    """          Step 6. Optimize                    """
    print('='*100)
    model.update()
    model.optimize()
    print('='*100)

    """          Step 7. Analyze results             """
    MSE_intercept, MSE_slope = coef[0].X,coef[1].X
    print('The optimal intercept:\t', MSE_intercept)
    print('The optimal slope:    \t', MSE_slope)
    print('The optimal objective value: \t', model.ObjVal)
    print('='*100)

    fig = plt.figure(figsize=(8,6))
    plt.plot(x_train,y_train,marker='o',lw=1,color='black',label='True Data'
    plt.plot(x_train,[MAD_intercept + x*MAD_intercept for x in x_train],colo
```

```python
    plt.plot(x_train,[MSE_intercept + x*MSE_slope for x in x_train],color='b
    plt.legend(fontsize=12)
    plt.show()
```

```
================================================================================
=========================
Gurobi Optimizer version 9.5.1 build v9.5.1rc2 (mac64[rosetta2])
Thread count: 10 physical cores, 10 logical processors, using up to 10 threa
ds
Optimize a model with 10 rows, 12 columns and 29 nonzeros
Model fingerprint: 0x2ce89fab
Model has 10 quadratic objective terms
Coefficient statistics:
  Matrix range     [5e-01, 4e+00]
  Objective range  [0e+00, 0e+00]
  QObjective range [2e+00, 2e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [7e-01, 4e+00]
Presolve removed 1 rows and 1 columns
Presolve time: 0.00s
Presolved: 9 rows, 11 columns, 27 nonzeros
Presolved model has 10 quadratic objective terms
Ordering time: 0.00s

Barrier statistics:
 Free vars  : 11
 AA' NZ     : 3.600e+01
 Factor NZ  : 4.500e+01
 Factor Ops : 2.850e+02 (less than 1 second per iteration)
 Threads    : 1

                  Objective                Residual
Iter       Primal          Dual         Primal    Dual     Compl     Time
   0   1.78781526e+00 -1.78781526e+00  3.77e-15 7.83e-01  0.00e+00     0s
   1   1.78685941e+00  1.78685583e+00  1.13e-08 7.83e-07  0.00e+00     0s
   2   1.78685941e+00  1.78685941e+00  2.30e-14 7.82e-13  0.00e+00     0s

Barrier solved model in 2 iterations and 0.01 seconds (0.00 work units)
Optimal objective 1.78685941e+00

================================================================================
=========================
The optimal intercept:   0.7149197447302484
The optimal slope:       0.5692322568168443
The optimal objective value:     1.7868594082382703
================================================================================
=========================
```



Feature Selection in Regression

This application is motivated by Feature Selection for Forecasting Gurobi example.

Preliminaries

Building on the previous example, we study a linear regression problem in which the optimized linear model should only use a small subset of features. In other words, we need to select the best subset of features that their linear combinations have the lowest training error.

Consider training set $\{(x^i, y^i), i = 1, 2, \ldots, N\}$ with N observations, where $x^i = (x^i_1, x^i_2, \ldots, x^i_d)$ and y^i are d-dimensional feature vector and the value of the response variable associated with the i-th observation, respectively.

The goal in linear regression problem is to find coefficients $\beta = (\beta_0, \beta_1, \beta_2, \ldots, \beta_d)$, where β_0 is the intercept and $(\beta_1, \beta_2, \ldots, \beta_d)$ is the slope of the linear model, such that the following approximation errors attain their lowest values:

$$\varepsilon_i := y^i - \left(\beta_0 + \sum_{j=1}^{d} \beta_j x^i_j\right), \qquad \forall i = 1, 2, \ldots, N.$$

Ordinary least squares (OLS). OLS is an optimization problem that allows us to obtain the lowest approximation error discussed above. OLS can be written as follows:

$$\min_{\beta} \ \mathrm{MSE}(\beta) \tag{OLS}$$

where

$$\text{MSE}(\beta) = \frac{1}{N}\sum_{i=1}^{N}\varepsilon_i^2 = \frac{1}{N}\sum_{i=1}^{N}\left[y^i - \left(\beta_0 + \sum_{j=1}^{d}\beta_j x_j^i\right)\right]^2$$

Indirect subset selection via LASSO regression. OLS does not perform the subset selection, that is, allowing only K out of $d+1$ elements of β take a non-zero value. The LASSO regression is a modification of OLS to indirectly perform subset selection. It is given by the following optimization problem:

$$\min_{\beta} \ \text{MSE}(\beta) \quad \text{s.t.} \quad \sum_{j=0}^{d}|\beta_j| \leq T. \tag{LASSO}$$

LASSO is a convex optimization problem. LASSO does not guarantee that its optimal β^* has at most K non-zero elements. That is why we refer to it as an indirect approach for subset selection.

Direct subset selection via l_0-regression. To directly perform subset selection, we cannot stay in the convex optimization world since couting the number of non-zeros of a vector is not a convex function. The non-convex optimization problem that allows us to perfrom subset selection is given by the following L_0-regression problem:

$$\min_{\beta} \ \text{MSE}(\beta) \quad \text{s.t.} \quad ||\beta||_0 \leq K \tag{l_0-regression}$$

where

$$||\beta||_0 := \text{number of non-zero elements of } \beta.$$

Comparison of OLS, LASSO, and l_0-regression

	OLS	LASSO	l_0-regression
Convex	Yes	Yes	No
Indirect subset selection	No	Yes	Yes
Direct subset selection	No	No	Yes
Constrained?	No	Yes	Yes
How to solve?	`LinearRegression` in sk-learn	`Lasso` in sk-learn	A model in Gurobi

Reformulating l_0-regression

Let's focus on formulating l_0 constraint $||\beta||_0 \leq K$. This constraint can be modeled using the followig constraints that are written via binary variables:

$$z_j := \begin{cases} 1 & \text{if } \beta_j \neq 0 \\ 0 & \text{if } \beta_j = 0 \end{cases}, \qquad \forall j = 0, 1, \ldots, d;$$

$$\sum_{j=0}^{d} z_j = K$$

Overall, we can reformulate l_0-regression problem as the following optimization problem with both continuous and integer variables:

$$\begin{aligned}
\min_{\beta, z} \ & \frac{1}{N} \sum_{i=1}^{N} \left[y^i - \left(\beta_0 + \sum_{j=1}^{d} \beta_j x_j^i \right) \right]^2 \\
& z_j = 0, \quad \text{if} \quad \beta_j = 0, && \forall j = 0, 1, \ldots, d, \\
& z_j = 1, \quad \text{if} \quad \beta_j \neq 0, && \forall j = 0, 1, \ldots, d, \\
& \sum_{j=0}^{d} z_j = K, \\
& \beta_j \ \ \text{unrestricted}, && \forall j = 0, 1, \ldots, d, \\
& z_j \ \ \text{binary}, && \forall j = 0, 1, \ldots, d.
\end{aligned}$$

In [ ]:

```
import numpy as np
import gurobi as gb

def MIQP_version_1(train_X:np.ndarray, train_y:np.ndarray, non_zero_budget:i
    """            Step 1. Sets and indices             """
    num_data,dim     = np.shape(train_X)
    data_index       = range(num_data)      # 1,2, ..., N
    dim_index        = range(dim+1)         # 0,1, ..., d+1

    """            Step 2. Parameters                   """
    # Data comes from the input. Append a column of ones to the
    # feature matrix train_X to account for the intercept
    train_X          = np.concatenate([np.ones((num_data, 1)),train_X], axis=

    """            Step 3. Decision variables           """
    model            = gb.Model('MIQP')
    coef             = model.addMVar(
                            shape     = dim + 1,
                            vtype     = gb.GRB.CONTINUOUS,
                            lb        = -gb.GRB.INFINITY,
                            ub        =  gb.GRB.INFINITY )

    non_zero         = model.addMVar(
                            shape     = dim+1,
                            vtype     = gb.GRB.BINARY)


    """            Step 4. Constraints                  """
```

```
    for j in dim_index:
        model.addConstr((non_zero[j] == 0) >> (coef[j] == 0))

    model.addConstr(gb.quicksum(non_zero)<=non_zero_budget)


    """         Step 5. Objective function          """
    epsilon = [None for _ in data_index]
    for i in data_index:
        epsilon[i] = train_y[i]  - coef@train_X[i,:]


    model.setObjective((1/num_data)*gb.quicksum(epsilon[i]@epsilon[i] for i


    """         Step 6. Optimize                    """
    model.setParam('Seed',       123)
    model.update()
    model.optimize()

    """         Step 7. Analyze results             """
    # we let the user to analyze the results and only return the optimal sol
    opt_coef = np.array([coef[j].X for j in dim_index])
    opt_val  = model.objVal
    return opt_coef, opt_val
```

Loading Data

In [ ]:
```
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

if __name__ == "__main__":

    housing = fetch_california_housing()

    print(housing['DESCR'])

    X = housing['data']
    y = housing['target']
    train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.20

    # Standardize the features so they have an avg of 0 and a sample var of
    scaler = StandardScaler()
    scaler.fit(train_X)
    train_X_std = scaler.transform(train_X)
    test_X_std  = scaler.transform(test_X)
```

```
.. _california_housing_dataset:

California Housing dataset
--------------------------

**Data Set Characteristics:**

    :Number of Instances: 20640

    :Number of Attributes: 8 numeric, predictive attributes and the target

    :Attribute Information:
        - MedInc        median income in block group
        - HouseAge      median house age in block group
        - AveRooms      average number of rooms per household
        - AveBedrms     average number of bedrooms per household
        - Population    block group population
        - AveOccup      average number of household members
        - Latitude      block group latitude
        - Longitude     block group longitude

    :Missing Attribute Values: None

This dataset was obtained from the StatLib repository.
https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts,
expressed in hundreds of thousands of dollars ($100,000).

This dataset was derived from the 1990 U.S. census, using one row per census
block group. A block group is the smallest geographical unit for which the
U.S.
Census Bureau publishes sample data (a block group typically has a populatio
n
of 600 to 3,000 people).

An household is a group of people residing within a home. Since the average
number of rooms and bedrooms in this dataset are provided per household, the
se
columns may take surpinsingly large values for block groups with few househo
lds
and many empty houses, such as vacation resorts.

It can be downloaded/loaded using the
:func:`sklearn.datasets.fetch_california_housing` function.

.. topic:: References

    - Pace, R. Kelley and Ronald Barry, Sparse Spatial Autoregressions,
      Statistics and Probability Letters, 33 (1997) 291-297
```

Trying function `MIQP_version_1`

In [ ]:
```
if __name__ == "__main__":
```

```
opt_coef, opt_val = MIQP(train_X_std,train_y, non_zero_budget=2)
```

Why do we get error? Aha, we need to do step 8, the Troubleshooting.

Let's rewrite the MSE objective in Gurobi differently to make it work

Recall the definition of error ε_i and OLS that are given by

$$\varepsilon_i := y^i - \left(\beta_0 + \sum_{j=1}^{d} \beta_j x_j^i\right), \qquad \forall i = 1, 2, \ldots, N.$$

We can compactly write the above identities as follows:

$$\varepsilon := Y - \hat{X}\beta$$

where

$$\varepsilon = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \varepsilon_3 \\ \vdots \\ \varepsilon_N \end{bmatrix}_{N\times 1}, \quad Y := \begin{bmatrix} y^1 \\ y^2 \\ y^3 \\ \vdots \\ y^N \end{bmatrix}, \quad \hat{X} := \begin{bmatrix} 1, & x_1^1, & x_2^1, & x_3^1, & \ldots, & x_d^1 \\ 1, & x_1^2, & x_2^2, & x_3^2, & \ldots, & x_d^2 \\ 1, & x_1^3, & x_2^3, & x_3^3, & \ldots, & x_d^3 \\ \vdots, & \vdots, & \vdots, & \vdots, & \ddots, & \vdots \\ 1, & x_1^N, & x_2^N, & x_3^N, & \ldots, & x_d^N \end{bmatrix}_{N\times(d+1)}$$

Using the above matrix form, we can rewrite MSE as the following quadratic form that is a predefined format in Gurobi:

$$\begin{aligned} \mathrm{MSE}(\beta) &= \frac{1}{N}\sum_{i=1}^{N} \varepsilon_i^2 \\ &= \frac{1}{N}(\varepsilon^\top \varepsilon) \\ &= \frac{1}{N}\left((Y - \hat{X}\beta)^\top (Y - \hat{X}\beta)\right) \\ &= \frac{1}{N}\left(\beta^\top \hat{X}^\top \hat{X}\beta - 2Y^\top X\beta + Y^\top Y\right) \\ &= \beta^\top \left(\frac{1}{N}\,\hat{X}^\top \hat{X}\right)\beta - \left(\frac{2}{N}\,Y^\top X\right)\beta + \frac{1}{N}\,Y^\top Y \qquad \text{(quadratic in } \beta) \end{aligned}$$

Overall, we get the following MIQP:

$$
\begin{aligned}
\min_{\beta,z}\ & \beta^\top \Big(\frac{1}{N}\,\hat{X}^\top \hat{X}\Big)\beta - \Big(\frac{2}{N}\,Y^\top X\Big)\beta + \frac{1}{N}\,Y^\top Y \\
& z_j = 0, \quad \text{if} \quad \beta_j = 0, && \forall j = 0,1,\ldots,d,\\
& z_j = 1, \quad \text{if} \quad \beta_j \neq 0, && \forall j = 0,1,\ldots,d,\\
& \sum_{j=0}^{d} z_j = K, \\
& \beta_j \quad \text{unrestricted}, && \forall j = 0,1,\ldots,d.\\
& z_j \quad \text{binary}, && \forall j = 0,1,\ldots,d.
\end{aligned}
$$

In [ ]:

```
import numpy as np
import gurobi as gb

def MIQP(train_X:np.ndarray, train_y:np.ndarray, non_zero_budget:int,output_
    """          Step 1. Sets and indices            """
    num_data,dim     = np.shape(train_X)
    data_index       = range(num_data)      # 1,2, ..., N
    dim_index        = range(dim+1)         # 0,1, ..., d+1

    """          Step 2. Parameters                  """
    # Data comes from the input. Append a column of ones to the
    # feature matrix train_X to account for the intercept
    train_X          = np.concatenate([np.ones((num_data, 1)),train_X], axis=

    """          Step 3. Decision variables          """
    model            = gb.Model('MIQP')
    coef             = model.addMVar(
                             shape     = dim + 1,
                             vtype     = gb.GRB.CONTINUOUS,
                             lb        = -gb.GRB.INFINITY,
                             ub        =  gb.GRB.INFINITY )

    non_zero         = model.addMVar(
                             shape     = dim+1,
                             vtype     = gb.GRB.BINARY)


    """          Step 4. Constraints                 """

    for j in dim_index:
        model.addConstr((non_zero[j] == 0) >> (coef[j] == 0))

    model.addConstr(gb.quicksum(non_zero)<=non_zero_budget)


    """          Step 5. Objective function          """
    objective = ((coef@(train_X.T@train_X/num_data))@coef
                 - (2*((train_y.T@train_X)/num_data))@coef
                     + (train_y.T@train_y)/num_data)
    model.setObjective(objective)


    """          Step 6. Optimize                    """
```

```
    Troubleshootedmodel.setParam('OutputFlag', output_flag)
    model.setParam('Seed',       123)
    model.update()
    model.optimize()

    """        Step 7. Analyze results             """
    # we let the user to analyze the results and only return the optimal sol
    opt_coef = np.array([coef[j].X for j in dim_index])
    opt_val  = model.objVal
    return opt_coef, opt_val
```

Solving MIQP

In [ ]:
```
if __name__ == "__main__":

    ## MILP using only K features
    print('='*100)
    opt_coef, opt_val = MIQP(train_X_std,train_y, non_zero_budget=5)
    print('='*100)
    print('\nMIQP Optimal Solution: \n', opt_coef)
```

Numerical Comparison of OLS, LASSO, and MIQP

In [ ]:
```
from sklearn.linear_model import LinearRegression,LassoCV
from sklearn.metrics import mean_squared_error as mse

if __name__ == "__main__":
    ## OLS regression using all features
    ols = LinearRegression()
    ols.fit(train_X_std, train_y)

    ## LASSO regression using all features
    lasso = LassoCV(cv=5)
    lasso.fit(train_X_std, train_y)

    ## MILP using only K features
    opt_coef, opt_val = MIQP(train_X_std,train_y, non_zero_bu dget=5,output_
    test_X_std_1      = np.concatenate([np.ones((np.shape(test_X_std)[0], 1)


    print('OLS Testing MSE   :', np.round(mse(test_y, ols.predict(test_X_std
    print('LASSO Testing MSE :', np.round(mse(test_y, lasso.predict(test_X_s
    print('MIQP Testing MSE  :', np.round(mse(test_y, test_X_std_1@opt_coef)

    print('\nOLS Optimal Solution:  \n', np.round(ols.coef_,2))
    print('\nLASSO Optimal Solution:\n', np.round(lasso.coef_,2))
    print('\nMIQP Optimal Solution: \n', np.round(opt_coef,2))
```

In [ ]: