# Machine Learning: algorithms, Code lecture 3

Data reading and regression

Nicky van Foreest

May 14, 2021

## Contents

## 1 Overview

- Last lecture
  - Gradient descent
  - Kernel methods
  - Examples of bad code, and how to repair
- This lecture focuses on optimization
  - LP solvers and graphs
  - ridge regression
  - lasso
- Next lecture:
  - Recursion
  - Decision trees (which are implemented with recursion)

Recall, the goal of the code lectures (and the set of handouts) is to explain how the toolboxes work and to provide you with working code so that you can see yourself how all works.

## 2 LP (`Pulp`) and graphs (`networkx`)

### 2.1 Intro and motivation

From the theory lectures and DSML you learned that Lasso can only be solved with optimization tools; there are no closed-form solutions to get the $\beta$ directly. As a second ingredient for this lecture, you should know that graphs are used to visualize and analyze certain types of data, for instance Facebook networks, or genes. Hence, it doesn't hurt to see how we can combine optimization and networks to analyze a scheduling problem.

First I will discuss the scheduling problem, then develop the code to analyze it. I find the tools really neat. We will show how to use classes to deal with graphs and optimization problems on a conceptual level. When we come to discussing lasso, you'll see that there are many common points.

### 2.2 Needed software

We will need:

- `graphviz`, a piece of software to visualize graphs
- `networkx`, a python library, to handle (very) large graphs.
- `pulp`, a python library, to solve the LP that results from optimizing the project plan. The API of `pulp` is very similar to that of the optimization library (`CVXPY`), which we will use to tackle the optimization problem involved in lasso.

I installed the software with `pip`.

```
pip install networkx
pip install pulp
```

### 2.3 Projects and critical paths

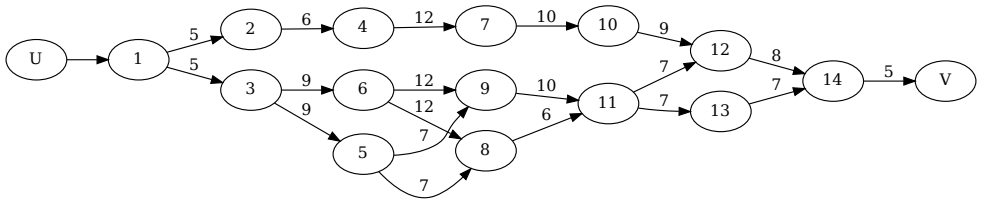
We have a project with tasks. Each task has a duration. Tasks can have predecessor tasks; a predecessor is a task that has be finished before the specific task is allowed to start. (For instance, we have to build the walls before putting the roof on the house.) The problem is to determine the *makespan*: the minimal time to complete the project by taking into account all task durations and predecessor relations.

We implement this as a graph. Each node corresponds to a task, each edge to a duration and a predecessor relation.

Here is an example. (I use the `graphviz` library to plot the graph.) The nodes `U` and `V` correspond to the start and finish of the project.

```
digraph{
    rankdir=LR;
    size="12.3,10.7!";
   "U" -> "1";
   "1" -> "2" [ label = "5" ];
   "2" -> "4" [ label = "6" ];
   "4" -> "7" [ label = "12" ];
   "7" -> "10" [ label = "10" ];
   "10" -> "12" [ label = "9" ];
   "12" -> "14" [ label = "8" ];
   "1" -> "3" [ label = "5" ];
   "3" -> "6" [ label = "9" ];
   "3" -> "5" [ label = "9" ];
   "6" -> "9" [ label = "12" ];
   "6" -> "8" [ label = "12" ];
```

```
    "5" -> "9" [ label = "7" ];
    "5" -> "8" [ label = "7" ];
    "9" -> "11" [ label = "10" ];
    "8" -> "11" [ label = "6" ];
    "11" -> "12" [ label = "7" ];
    "11" -> "13" [ label = "7" ];
    "13" -> "14" [ label = "7" ];
    "14" -> "V" [ label = "5" ];
}
```




## 2.4 Implementation in `networkx`

I'll use the `networkx` library to deal with this graph numerically. I also load `pulp` here because I'll need that below anyway.

```
import networkx as nx
import pulp
```

Here are the nodes with the task durations, e.g., node 1 has a duration of 5.

```
nodes = (
    (1, 5),
    (2, 6),
    (3, 9),
    (4, 12),
    (5, 7),
    (6, 12),
    (7, 10),
    (8, 6),
    (9, 10),
    (10, 9),
    (11, 7),
    (12, 8),
    (13, 7),
    (14, 5),
)
```

And the edges.

```
edges = [
    (1, 2),
    (2, 4),
    (4, 7),
    (7, 10),
    (10, 12),
    (12, 14),
    (1, 3),
    (3, 6),
    (3, 5),
    (6, 9),
    (6, 8),
    (5, 9),
    (5, 8),
    (9, 11),
    (8, 11),
    (11, 12),
    (11, 13),
    (13, 14),
]
```

I add the nodes and edges to a `DiGraph`, a class of `networkx` to implement directed graphs. I add a tag `p` to each node to indicate the task duration (processing time).

```
cpm = nx.DiGraph()

for n, p in nodes:
    cpm.add_node(n, p=p)

cpm.add_edges_from(edges)
```

Finally, I add a `Cmax` node with job duration `0` and with all other nodes as its predecessor. When the node `Cmax` is finished, the project is finished. Thus, the objective will be to minimize the completion time of `Cmax`, because that will minimize the makespan of the project.

```
cpm.add_node("Cmax", p=0)
cpm.add_edges_from([(j, "Cmax") for j in cpm.nodes()])
```

Remark: I built the above by hand. For real projects, such information is stored in a database, and then we use computer programs to build the graphs.

### 2.5 Solving for the makespan

Now that we have graph, we want to compute the completion time of `Cmax`. For this we model the problem as an LP.

We begin with loading `pulp`, a python library to solve LPs. Then we create a problem, to which we add decision variables, constraints, and an objective.

```
prob = pulp.LpProblem("Critical_Path_Problem", pulp.LpMinimize)
```

As for the decision variables, each job $j$ has a starting time $s_j$ and a completion time $c_j$, for $j = 1, \ldots, n$.

```
all_nodes = [j for j in cpm.nodes()]
s = pulp.LpVariable.dicts("s", all_nodes, 0)  # start
c = pulp.LpVariable.dicts("c", all_nodes, 0)  # completion
```

The 0 (the last attribute) constrains the variables to non-negative values.

Now we implement the constraints. For each job $j$, its completion time $c_j$ must be larger than its starting time $s_j$ plus its processing time $p_j$:

$$c_j \geq s_j + p_j, \quad \text{for } j = 1, \ldots, n.$$

```
for j in cpm.nodes():
    prob += c[j] >= s[j] + cpm.nodes[j]['p']
```

(Observe that the python code is very similar to the maths.)

Next, a job can only start after all its predecessors are finished:

$$s_j \geq s_i + p_i, \text{ for all } i \to j,$$

where we write $i \to j$ to mean that job $i$ precedes job $j$.

```
for j in cpm.nodes():
    for i in cpm.predecessors(j):
        prob += s[j] >= s[i] + cpm.nodes[i]['p']
```

Observe here we that the `DiGraph` class of `networkx` provides us with a function to compute the predecessors. We get it for free!

Finally, a job's completion time must be larger than the completion times of each of its predecessors plus its own processing time:

$$c_j \geq c_i + p_j, \text{ for all } i \to j.$$

```
for j in cpm.nodes():
    for i in cpm.predecessors(j):
        prob += c[j]  >= c[i] + cpm.nodes[j]['p']
```

Observe that the constraint $C_{\max} \geq c_j$ is automatically included by these constraints, as 'Cmax' is a node in the project graph.

The objective function is such that the makespan is minimized and, simultaneously, that the earliest starting times are minimized and latest completion times are maximized. With this we can find the slack of task $j$ as $c_j - s_j - p_j$. Knowing the slack is important: any job with zero slack is *tight*, which means that it's on the *critical path*. Any detail on a tight job will result in a delay of the entire project. Hence, it's essential to manage the tight jobs well, so as to prevent delay.

We therefore define the objective function as

$$\min \left\{ C_{max} + \epsilon \sum_{j=1}^{n} s_j - \epsilon \sum_{j=1}^{n} c_j \right\},$$

where $\epsilon$ is some small number, and $n$ is the number of tasks.

```
eps = 1e-5
prob += (
    c["Cmax"]
    + eps * pulp.lpSum([s[j] for j in cpm.nodes()])
    - eps * pulp.lpSum([c[j] for j in cpm.nodes()])
)
```

Solving is now very simple. We can just call `solve` to have the LP built, and solved.

```
# prob.writeLP("cpmLP.lp")
prob.solve()
pulp.LpStatus[prob.status]
```

Let's check the status:

```
pulp.LpStatus[prob.status]
```

```
Optimal
```

This is good news: the problem is solved to optimality.

Here are the earliest starting, completion times, and slacks. All jobs that have zero slack are in the critical path.

```
for j in cpm.nodes():
    print(
        j, s[j].varValue, c[j].varValue, c[j].varValue - s[j].varValue - cpm.nodes[j]['p']
    )
```

```
1 0.0 5.0 0.0
2 5.0 12.0 1.0
3 5.0 14.0 0.0
4 11.0 24.0 1.0
5 14.0 26.0 5.0
6 14.0 26.0 0.0
7 23.0 34.0 1.0
8 26.0 36.0 4.0
9 26.0 36.0 0.0
10 33.0 43.0 1.0
11 36.0 43.0 0.0
12 43.0 51.0 0.0
13 43.0 51.0 1.0
14 51.0 56.0 0.0
Cmax 56.0 56.0 0.0
```

The project completion time is 56 time units.

### 2.6 Summary

Observe how `networkx` helps us to model and solve this scheduling problem on a conceptual level. We have to do astonishingly little ourselves, we can fully concentrate on getting the model correct and on the solution itself, and we don't have to be concerned with any tough (numerical) algorithm.

To show you how little real code we actually need, check `cpm.py` on `github`. This is actually what I would normally program; for me, there is not much need for documentation, as the code tells me what is going on. Of course I need code to enter the data, such as the nodes, task durations and predecessors, and I need code to process the results. These parts of the code can be pretty big at times, BTW. However, as you can see, the engine itself is very short and conceptually clear.

## 3 OLS

Now I want to demonstrate how to solve OLS with the optimization library `CVXPY`. I don't want to use `sklearn` directly, although that would be natural since we are dealing with a data analysis problem. However, `CVXPY` is a generic tool, a tool we can use for many other optimization tools, such as portfolio optimization, inventory control, and so on. So, we learn one tool, but get lots of off-spin.

Below I will use `CVXPY` to deal with lasso, but before trying such a tool on new problems, I prefer to see how it works on things I can check. Hence, let's tackle OLS first.

Installing `CVXPY` took a bit of time (a few minutes) as apparently lots of stuff had to be compiled. Again I use `pip` for this.

```
pip install cvxpy
```

### 3.1 Imports

```
1 import numpy as np
2 from numpy import linalg as LA
3 import cvxpy as cp
4 import matplotlib.pyplot as plt
```

### 3.2 A common API

Below I'll build several predictors: OLS, ridge, lasso. These predictors have some common functionality, which I therefore put in a base class. In particular, I want the predictors to have a similar interface, called an API (Application programming interface). Why is this a good idea? Well, in the first place, it becomes possible to reuse things we learned. We have to learn once to call `solve`, and then it works the same for the rest. Second, we can test the method of a parent class separately. If it works, it will also work for all the derived classes. Hence, we gain in software reliability. We only have to ensure to do it right once. In python this is called the *DRY* concept: Don't Repeat Yourself. Applying this concept makes you a much better

programmer. Third , it makes it easier to replace one type of regressor by another. They all interact in the same way with 'the rest of the world'.

I'll derive a class for OLS, ridge regression and Lasso from the `Preditor` class.

```
class Predictor:
    def __init__(self, X, Y, gamma=0):
        self.X = X
        self.Y = Y
        self.gamma = gamma
        self.beta_0 = 0

    def loss(self, X, Y, beta):
        return np.square(Y - X @ beta).sum()

    def solve(self):
        raise NotImplemented

    def predict(self, x):
        return self.beta_0 + x @ self.beta

    def mse(self, X, Y):
        n, p = X.shape
        return self.loss(X, Y, self.beta).value / n
```

The `solve` method has to be implemented in the derived class.

### 3.3 Regular OLS

With the `Predictor` class, the implementation of an `OLS` class is very short. I just have to build the `solve` method; the rest can be inherited.

```
class OLS(Predictor):
    def solve(self):
        self.beta = LA.solve(self.X.T @ self.X, self.X.T @ self.Y)
```

Thus, I use the regular way to solve an OLS.

### 3.4 OLS with CVXPY

I also want to build OLS as a convex optimization problem, and then I let `CVXPY` solve it. (I modified the code I found here.)

```python
class OLS_convex(Predictor):
    def loss(self, X, Y, beta):
        return cp.sum_squares(Y - X @ beta)

    def objective(self, beta):
        return self.loss(self.X, self.Y, beta)

    def solve(self):
        n, p = self.X.shape
        beta = cp.Variable(p)
        obj = cp.Minimize(self.objective(beta))
        problem = cp.Problem(obj)
        problem.solve()
        self.beta = np.array(beta.value)
```

I include an `objective` method, because the ridge and lasso regressors will follow the same pattern. Again, note that it suffices to build a `solve` method.

### 3.5 A test

Since class definitions themselves are also objects, we can pass them to test functions, such as I do below.

Here is a subtle point. A class is a type, and not the same as an instance or object of a class. An instance of a class holds actual data within it, whereas a type is merely a template that specifies how its instances should behave. When we write `x = Foo()`, we say `x` is an object or instance of type `Foo`.

To add to the confusion, `Foo` is both a type and an object. That is because `Foo` is an instance of the type `type` - but not of type `Foo`. Understanding the distinction between objects and types is paramount in furthering your programming skills. I admit, it takes some time to understand, but once you do, you will write better code, with less mistakes, and you can yet more adhere to the DRY concept.

```python
def test(method):
    np.random.seed(3)

    n, p = 50, 20
    sigma = 5

    beta_star = np.ones(p)
    beta_star[:10] = np.zeros(10)

    X = np.random.randn(n, p)
    Y = X @ beta_star + np.random.normal(0, sigma, size=n)

    algo = method(X, Y, gamma=1)
    algo.solve()
    print(algo.beta[0])
```

```python
test(OLS)
test(OLS_convex)
```

```
0.7362515517803382
0.7362515517803384
```

```
def test_2(method):
    np.random.seed(3)

    n, p = 50, 20
    sigma = 5

    beta_star = np.ones(p)
    beta_star[:10] = np.zeros(10)

    X = np.random.randn(n, p)
    Y = X @ beta_star + np.random.normal(0, sigma, size=n)

    algo = method(X, Y, gamma=1)
    algo.solve()

    X = np.random.randn(2, p)
    print(algo.predict(X))
```

```
test_2(OLS)
test_2(OLS_convex)
```

```
[-4.31928848  4.15165271]
[-4.31928848  4.15165271]
```

Note that I compare my 'new' tool `CVXPY` against my 'old' tool `numpy.linalg.solve`. Perhaps you find me a bit over cautious, but from experience I can tell you that if you forget (or are too lazy) to do such steps, you can be confronted with very awkward surprises. Such intermediate checks help to locate mistakes that I might make later. In case errors occur later, it's unlikely I have to search for the problem in the points above.

# 4 Ridge regression

## 4.1 Regular Ridge regression

I first build ridge regression as explained in DSML.6.2; this method is direct, i.e., it solves a problem of the form $Ax = b$. If you have read the relevant parts, the code below is evident.

```
class Ridge(Predictor):
    def solve(self):
        n, p = self.X.shape
        A = self.X.T @ self.X + self.gamma * np.eye(p)
        b = self.X.T @ self.Y
        self.beta = LA.solve(A, b)
```

## 4.2 Ridge regression with CVXPY

I adapted the code I found here: cvxpy ridge. Just read the code, it is self explanatory.

```python
class Ridge_convex(Predictor):
    def loss(self, X, Y, beta):
        return cp.pnorm(Y - X @ beta, p=2) ** 2

    def regularizer(self, beta):
        return cp.pnorm(beta, p=2) ** 2

    def objective(self, beta):
        return self.loss(self.X, self.Y, beta) + self.gamma * self.regularizer(beta)

    def solve(self):
        n, p = self.X.shape
        beta = cp.Variable(p)
        obj = cp.Minimize(self.objective(beta))
        problem = cp.Problem(obj)
        problem.solve(solver="SCS")
        self.beta = np.array(beta.value)
```

I got some warnings with the standard solver `ECOS`. A search on `cvxpy warning` helped me resolve this: I called the solver `SCS` instead of `ECOS`. (I don't know about the differences; I just replaced one string for another.)

### 4.3 A test

I can use the earlier test function right away.

```python
test(Ridge)
test(Ridge_convex)
```

```
0.6839570416892673
0.6837994245619745
```

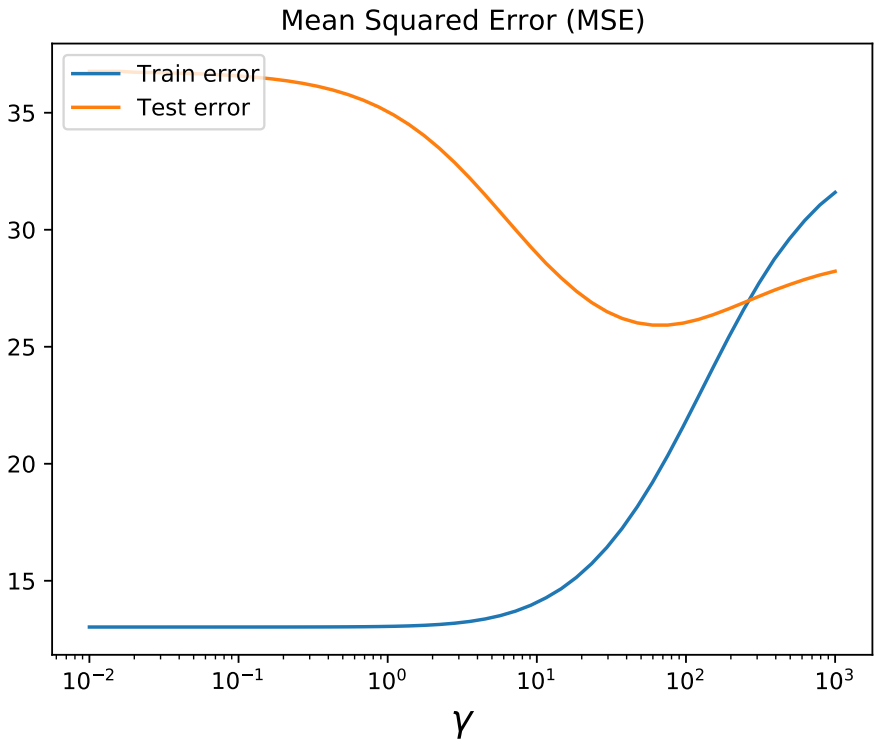
### 4.4 Graphs

Finally we make a graph of how the mse and betas vary as function of the regulation cost $\gamma$.

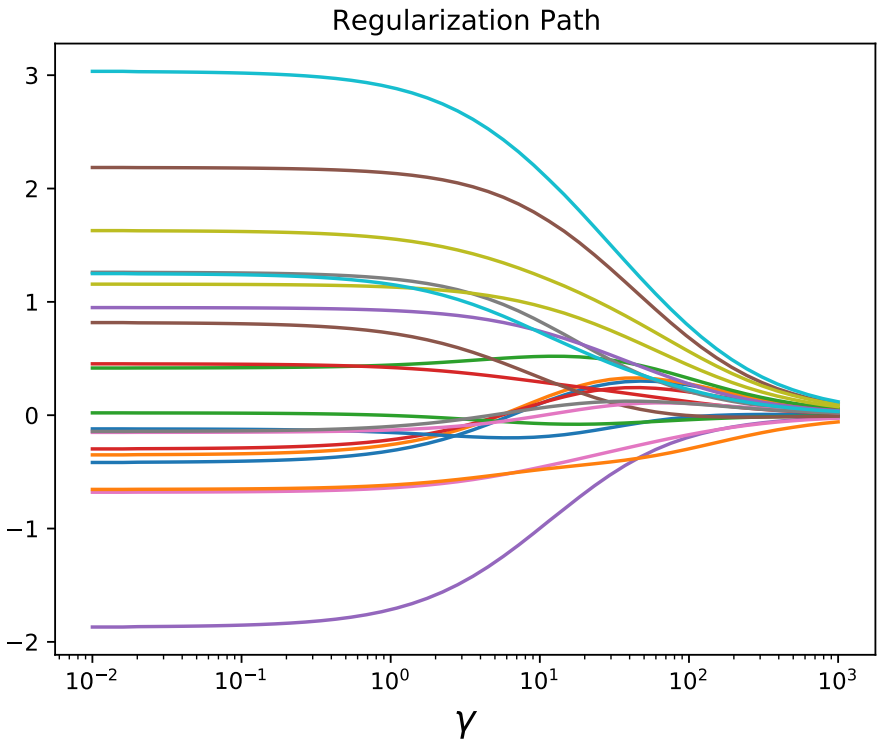
```python
def statistics_plotting(method):
    n, p = 100, 20
    sigma = 5

    beta_star = np.ones(p)
    beta_star[10:] = 0

    np.random.seed(3)
    X = np.random.randn(n, p)
    Y = X @ beta_star + np.random.normal(0, sigma, size=n)

    X_train, Y_train = X[:50, :], Y[:50]
    X_test, Y_test = X[50:, :], Y[50:]

    gamma_values = np.logspace(-2, 3, 50)
    train_errors, test_errors, beta_values = [], [], []
    for g in gamma_values:
        algo = method(X_train, Y_train, gamma=g)
        algo.solve()
        train_errors.append(algo.mse(X_train, Y_train))
        test_errors.append(algo.mse(X_test, Y_test))
        beta_values.append(algo.beta)
    # print(train_errors)

    plt.clf()
    plt.plot(gamma_values, train_errors, label="Train error")
    plt.plot(gamma_values, test_errors, label="Test error")
    plt.xscale("log")
    plt.legend(loc="upper left")
    plt.xlabel(r"$\gamma$", fontsize=16)
    plt.title("Mean Squared Error (MSE)")
    fname = "figures/" + algo.__class__.__name__ + "_mse.pdf"
    plt.savefig(fname)

    plt.clf()
    num_coeffs = len(beta_values[0])
    for i in range(num_coeffs):
        plt.plot(gamma_values, [wi[i] for wi in beta_values])

    plt.xlabel(r"$\gamma$", fontsize=16)
    plt.xscale("log")
    plt.title("Regularization Path")
    fname = "figures/" + algo.__class__.__name__ + "_betas.pdf"
    plt.savefig(fname)
```

Now let's use it.

```python
statistics_plotting(Ridge_convex)
```




Since you like econometrics, explain this graph!
And a plot of how the sizes of $\beta$ vary as a function of $\gamma$.




### 4.5 Ridge regression without penalizing the constant

When we don't want to include a penalty on the constant $\beta_0$ the construction of the matrices is a bit harder. Here is the code. Read it so that you understand how to tackle the matrix multiplications.

Observe again that I do not include the $n$ term with the constant $\gamma$. Also, the `dum` stores the vector $1'X$ to save time in the other computations.

```
class Ridge_with_constant(Predictor):
    def solve(self):
        n, p = self.X.shape
        A = self.X.T @ self.X
        dum = np.ones((1, n)) @ self.X
        A -= dum.T @ dum / n
        A += self.gamma * np.eye(p)
        b = (self.X.T - dum.T @ np.ones((1, n)) / n) @ self.Y
        self.beta = solve(A, b)
        self.beta_0 = np.ones((1, n)) @ (self.Y - dum @ beta)
```

## 5 Lasso regression

There is not direct method to find the minimum in the objective for lasso; we have to use a tool like CVXPY. I borrowed some ideas from cvxpy.lasso.

## 5.1 Implementation

Given the classes I already built above, there is not much work to do. In fact, the three lines below is all!

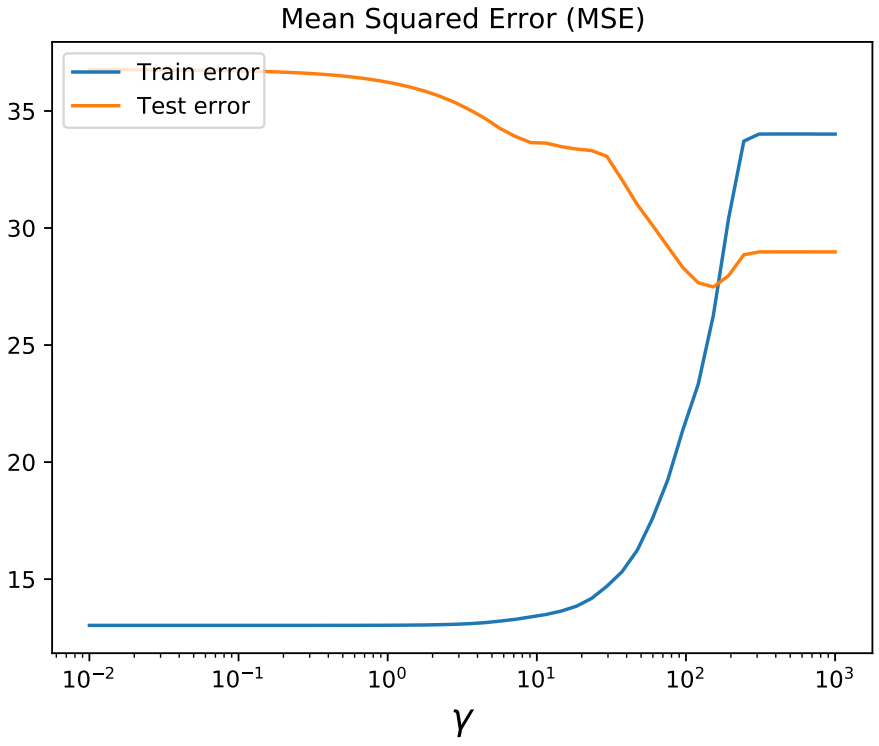
```
class Lasso(Ridge_convex):
    def regularizer(self, beta):
        return cp.norm1(beta)
```

Here is some advice: think hard about how I organized the code, and compare my code with that of the `CVXPY` site. Hopefully you can see how much work I have been able to suppress by defining good classes. In more general terms, I applied a concept called *abstraction*. I focused on aspects that all predictors have in common. This I put in the `Predictor` class. Then I checked what ridge regression and lasso have in common. As it turns out, quite a lot. That is why the implementation of the lasso class is so small: most of the work is covered in `Ridge_convex`.

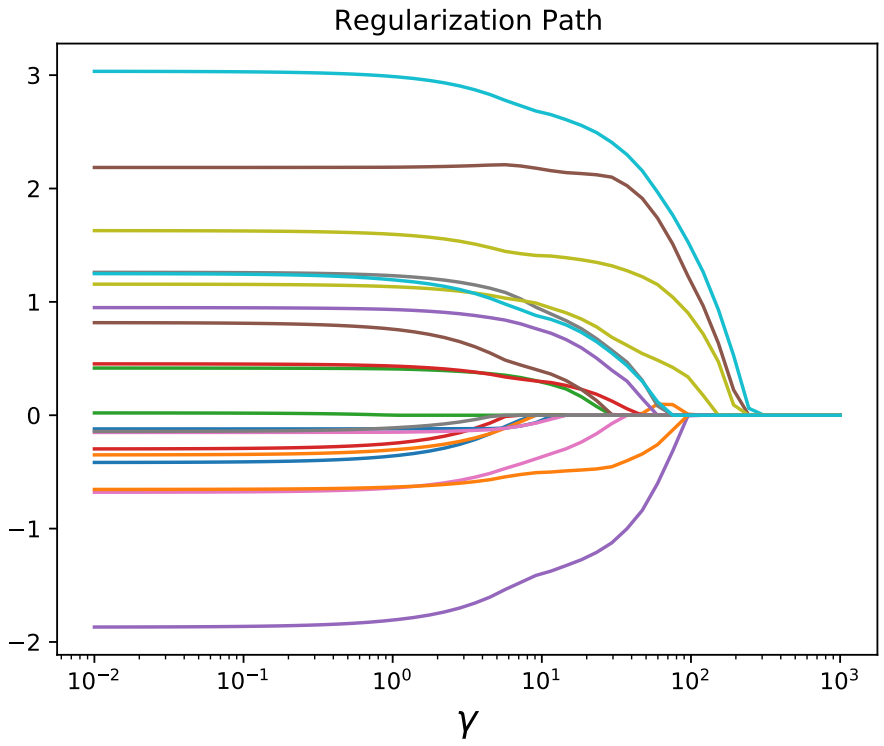
## 5.2 Graphs

Making graphs of the mse and the betas comes down to calling one of the above functions. That's all there is left!

```
statistics_plotting(Ridge_convex)
```



Since you like econometrics, explain this graph!
And a plot of how the sizes of $\beta$ vary as a function of $\$\gamma$.




Recall from the `statistics_plotting()` function that the first 10 betas were 1 and the rest zero, i.e, $\beta_i = 1$ for $i < 10$, and $\beta_i = 0$ for $i = 10, \ldots, p-1$. I have to admit that I am not particularly impressed by the selection of lasso. The test MSE achieves its minimum around $\gamma = 100$. But I have a hard time to learn from the regularization paths what the best $\gamma$ should be. Perhaps I made a mistake somewhere; if so, let me know.

# 6 General advice

## 6.1 Technical part

- Ridge regression is a problem that can be solved in the form of $Ax = b$.
- Lasso requires optimization tools. In general, solving optimization problems must be slower than solving $Ax = b$, since the latter is a step that is repeatedly done to solve the former. Perhaps optimization is also more prone to numerical issues (but I haven't tested this, so it's a matter of belief). Hence, if you don't have compelling reasons to use Lasso, perhaps it's better to use ridge regression.

## 6.2 Coding

- Defining a good class hierarchy—finding a good level of *abstration*—can save you a lot of typing. Moreover, it helps to understand code much better. Finally, we can reuse lots

of code. Hence, we only have to test code once. All classes that derive from a parent class benefit from this.

- Good python libraries share ideas of the API. For instance, in `pulp` and `CVXPY` we define a problem, and we can call the `solve` method for both. Similar names and APIs make code better understandable, and easier to memorize.
- Avoid (to the extent possible) to learn ten different tools, each with their own quirks, such as AIMS for LPs, Stata (?) for data analysis, yet other tools for convex optimization, and so on. Using many different tools makes you less efficient, and you'll make more mistakes.
- Realize, later in life you don't get paid to use AIMS (or whatever other tools); you get paid to get a job done. So, any time spent on learning ten different tools is (basically) wasted. Your customer will not pay for it, so it's time of yourself not spent on Sunday picnics.

For me:

- I prefer to be lazy and let the computer solve my problems. So, if I have to do something tedious, I often try to develop on an algorithm (an intellectual challenge) and pass the execution of the job (very boring and time consuming) to the computer.
- However, in general I find most (nearly all?) computer things dull, so the less time I have to invest in learning such things, the better. In other terms, I am not a computer scientist or programmer; I am not interested in seeing how ten different computer tools for the same job work.

Best advice: Do as I do :-)

# 7 Exercises

## 7.1 Exercise 1

Read/Browse the web/documentation of the following topics (Just spend a few minutes so that you see what's it about.)

- `networks`
- `https://graphviz.org/` to visualize (huge) graphs
- `networkx and data analysis`
- `networkx and pandas`
- Understand why and when to use classes, for instance here.
- `pulp`, Setting up LPs; search for 'gurobi and python' for the best LP solvers.
- `https://www.cvxpy.org/`. Check the many examples.
- `https://osqp.org/docs/solver/index.html`. This is another solver. Perhaps useful. For instance, they provide an example on portfolio optimization.
- `https://xavierbourretsicotte.github.io/ridge_lasso_visual.html`

### 7.2 Exercise 2

The method `mse` in the `Predictor` class cannot be applied to my implementation of OLS. Why not? (Hence, the implementation of the `Predictor` contains a bug.) Can you repair this?

Hint: the mistake is `value` attribute.

### 7.3 Exercise 3

Can you extend the implementation of lasso to *Fused Lasso*? The objective for fused lasso is this:

$$||Y - X\beta||_2 + \lambda_1||\beta||_1 + \lambda_2 \sum_{i=2}^{p} |\beta_j - \beta_{j-1}|, \tag{1}$$

where $||\cdot||_2$ is the 2-norm, and $||\cdot||_1$ the 1-norm. Check out Wikipedia if you don't know what I mean by this. It's important to know, as these types of norm are also used by `numpy`, `CVXPY`, and other numerical toolboxes.

I haven't tried this, nor tested it. I have no idea whether this additional regularization term works or not. In all honesty (and naive no doubt), I don't see much value in such tweaks. Why would one want to regularize the difference between successive terms of $\beta$? Take $\lambda_1$ big, then $||\beta||$ will be small anyway, so $\beta_j - \beta_{j-1}$ must also be small. Second, why would I care about $\beta_1$ being big, and $\beta_2$ small? If this matters, it must say something about the first and second column of $X$. But why would there be any particular order in the features?

As a general rule: in my own experience, tweaks seldom work. They are not based on fundamental insights (such as regulation which ridge and lasso already capture). Tweaks are mostly (dumb) guess that might work, but just as well might fail. As far as I can see, tweaks are not robust solutions to real problems. For real, practical problems (problems that are typically extremely hard), simply don't use tweaks. However, if you disagree with me, no problem. I can be completely wrong here. Nobody knows the answer here :-)

### 7.4 Exercise 4

Redo the lasso problem above with `sklearn`.