Models @ MSR

Johannes Härtel – johannes.hartel@vub.be

Last session:

*

Scala, Python, Java, R …

Python

Python

Python

Python & JS

R

Last session:

Typical Workflow

1. Cone repository
2. Extract **data** on the comments.
3. Transform the **data** until it answers your question.

The “**data** structures” used here are somewhat special.

Answering questions might be somehow special.

Typical (Research) Questions (last session)

1. What is the average **number** of comments in Java files?
2. What is the Java **file** with the lowest **number** of comments?
3. What is the Java **package** with the lowest **number** of comments?
4. What is the Java **package** with the lowest **fraction** of comments?

Typical (Research) Questions **(NEW)**

NEW: Does public functionality cause more comments written by developers**?**

Typical (Research) Questions **(NEW)**

NEW: Does public functionality cause more comments written by developers**?**

- Asks for a **relationship** between two variables (we focus on this).
- Asks for **causality** (we will not discuss this).

We **use “models”** that **describe (alternatives) how data is produced**. We compare models and read parts of the models to answer our questions.

Today:

Source ChatGPT:
"What are the different names used for a linear regression?"

1. **Simple Linear Regression:** When there is only one independent variable.
2. **Multiple Linear Regression:** When there are multiple independent variables.
3. **Ordinary Least Squares (OLS) Regression:** OLS is a method used to find the best-fitting linear regression line by minimizing the sum of the squares of the vertical distances (residuals) between observed and predicted values.
4. **Least Squares Regression:** Similar to OLS, this term refers to the minimization of the sum of squared residuals.
5. **Linear Least Squares:** Another term emphasizing the minimization of the sum of squared differences.
6. **Regression Analysis:** A general term for statistical methods used to analyze the relationship between variables, but linear regression specifically focuses on linear relationships.
7. **Linear Modeling:** Describes the process of creating a linear model to represent the relationship between variables.
8. **Regression Modeling:** A broader term that includes various types of regression analysis, with linear regression being one specific type.
9. **Gradient Descent:** This refers to the optimization algorithm used to find the coefficients of the linear regression model.
10. **Line of Best Fit:** A more informal term, especially used in educational settings, to describe the regression line that best fits the data points.
11. **Regression Line:** Refers to the line that best fits the data in a scatterplot.

The terminology is confusing.

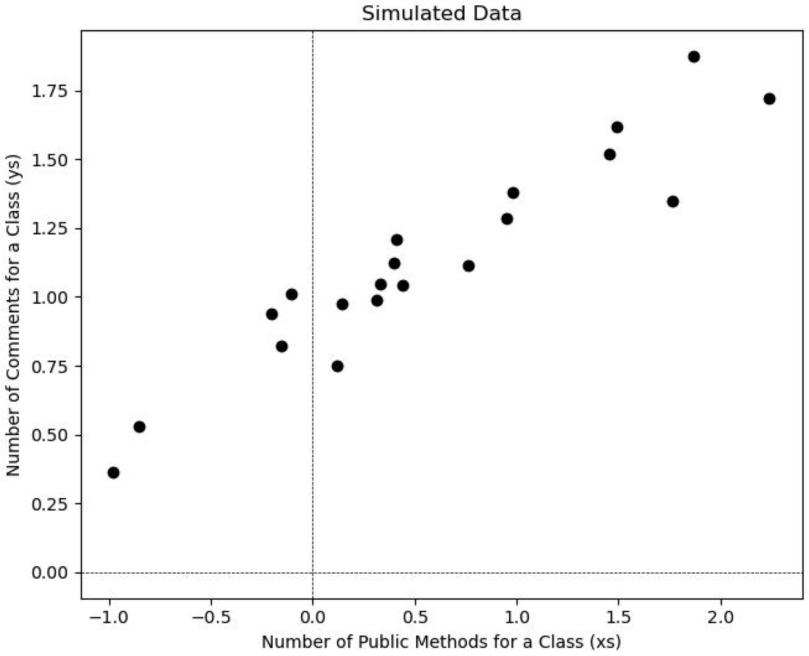
The terminology can be compared to a black box API.

Hence, we will use code to understand it.

Data

Let’s imagine what “supporting data” would look like …

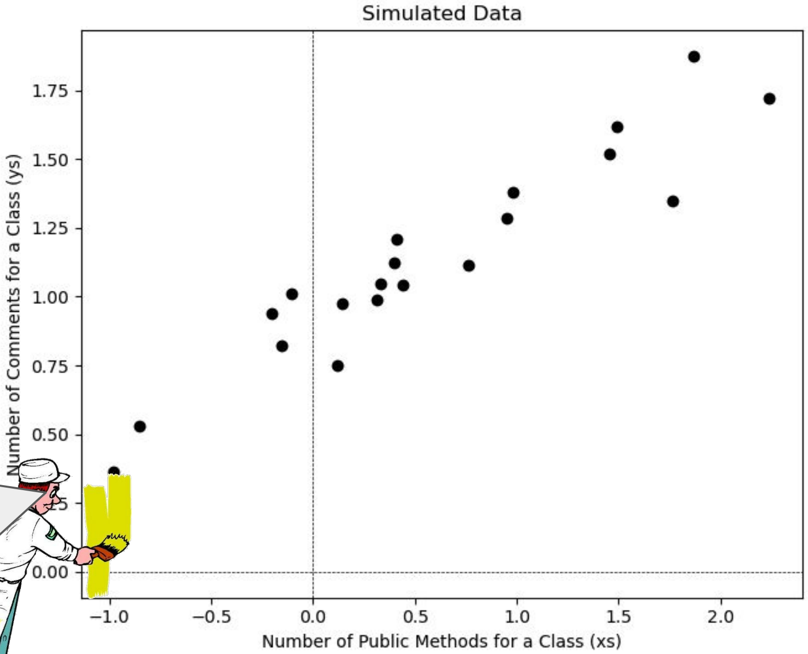
(Does public functionality cause more comments written by developers?)

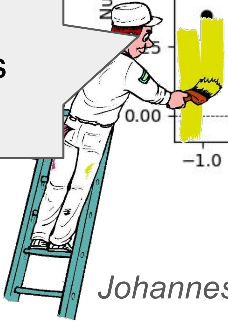


Let’s imagine what “supporting data” would look like …

(Does <u>public functionality</u> cause <u>more comments</u> written by developers?)

I have “**faked**”, **”simulated” or ”invented”** this data using a model. It describes (alternatives) how data is produced. I show it now.





Johannes Härtel – johannes.hartel@vub.be

Programming Demo

(Part 1)

Johannes Härtel – johannes.hartel@vub.be

Find the code in “part1.py”.

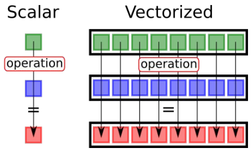
```
# START: Generate fake-data.
n = 20 # number of observations.
xs = np.random.normal(size = n) # random values for x.
mu = 0.9 + xs * 0.4 # mean values for y.
sigma = 0.1

ys = np.random.normal(scale = sigma, size = n) + mu # final output y.

# END: Generate fake-data.
```

What we did is describing how data is produced in terms of code (a “linear model”).

(Code is nice because it can be executed; however, …)

How mathematicians would <u>write</u> our first linear model

(in essence, a model relates variables)



Remember vectorization. I added small vectors signs, but typically, they are avoided. I avoid tensors with rank bigger than 1 (matrices…), but there are used a lot.

$$\overrightarrow{\text{ys}} \sim \text{Normal}(\overrightarrow{\text{mu}}, \text{sigma})$$

$$\overrightarrow{\text{mu}} = \text{alpha} + \overrightarrow{\text{xs}} * \text{beta}$$

How mathematicians would read our first linear model

(in essence, a model relates variables)

Variable ys distributed (symbol ~) normally,
with mean mu, and standard deviation sigma.

$$\overrightarrow{\text{ys}} \sim \text{Normal}(\overrightarrow{\text{mu}}, \text{sigma})$$

$$\overrightarrow{\text{mu}} = \text{alpha} + \overrightarrow{\text{xs}} * \text{beta}$$

Standard Math.

How mathematicians would <u>use</u> our first linear model
(in essence, a model relates variables)

“simulate”
(what we did)

$$\vec{ys} \sim \mathrm{Normal}(\vec{mu}, sigma)$$

$$\vec{mu} = alpha + \vec{xs} * beta$$

invent vs. compute vs. have

(Used terminology in practice is a mess. This is my attempt to structure it).

How mathematicians would <u>use</u> our first linear model
(in essence, a model relates variables)

"simulate" (what we did)	"infer" (what we will do next)
$\vec{ys} \sim \text{Normal}(\vec{mu}, sigma)$	$\vec{ys} \sim \text{Normal}(\vec{mu}, sigma)$
$\vec{mu} = alpha + \vec{xs} * beta$	$\vec{mu} = alpha + \vec{xs} * beta$

The yellow stuff is often called a parameter.

invent vs. compute vs. have

(Used terminology in practice is a mess. This is my attempt to structure it).

How mathematicians would <u>use</u> our first linear model
(in essence, a model relates variables)

"simulate"
(what we did)

$$\vec{ys} \sim \mathrm{Normal}(\vec{mu}, sigma)$$
$$\vec{mu} = alpha + \vec{xs} * beta$$

"infer"
(what we will do next)

$$\vec{ys} \sim \mathrm{Normal}(\vec{mu}, sigma)$$
$$\vec{mu} = alpha + \vec{xs} * beta$$

The yellow stuff is often called a parameter.

"predict"
(what we will do some when)

$$\vec{ys} \sim \mathrm{Normal}(\vec{mu}, sigma)$$
$$\vec{mu} = alpha + \vec{xs} * beta$$

$\boxed{\text{invent vs. compute vs. have}}$

(Used terminology in practice is a mess. This is my attempt to structure it).

Obviously, all three types of usage require code (or libraries that do the job for you).

I explain it in terms of code. You are allowed to use libraries.

Where to go next?

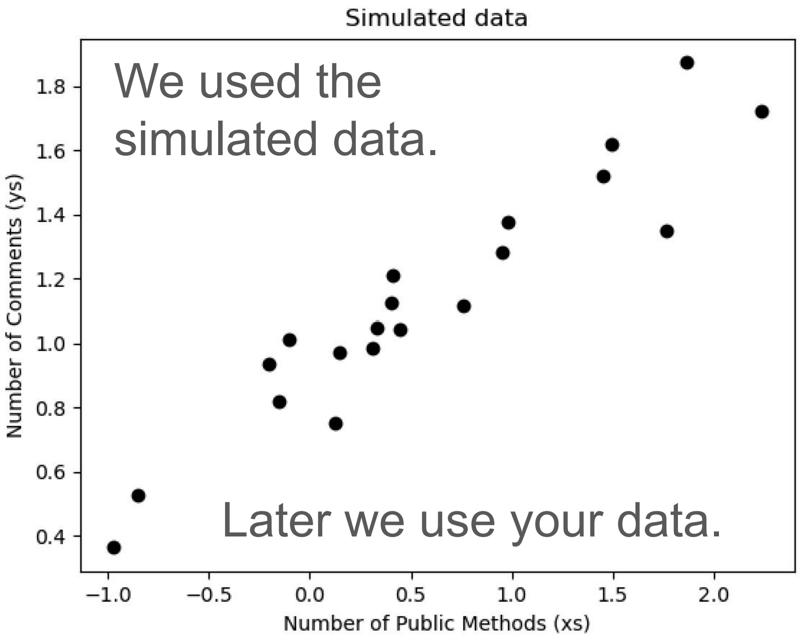
“simulate”
(what we did)

“infer”
(what we will do next)

We go here

“predict”
(what we will do some when)

We focus on the inference


Simulated data
We used the simulated data.
Later we use your data.
Number of Comments (ys)
Number of Public Methods (xs)
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Infer (compute) alpha & beta, having xs and ys
(we will get mu automatically, sigma does not truly matter).

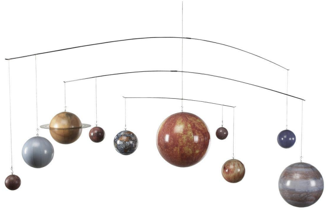
$$\overrightarrow{\mathbf{ys}} \sim \mathbf{Normal}(\overrightarrow{\mathbf{mu}}, \mathbf{sigma})$$

$$\overrightarrow{\mathbf{mu}} = \mathbf{alpha} + \overrightarrow{\mathbf{xs}} * \mathbf{beta}$$

invent vs. compute vs. have

Error Function

A.k.a., a cost function, a (log) **likelihood function**, or whatever…



Vectors ys and mu should be as close as possible.

$$\overrightarrow{ys} \sim \mathbf{Normal}(\overrightarrow{\mathbf{mu}}, \mathbf{sigma})$$

← Error is here (~ symbol).

$$\overrightarrow{\mathbf{mu}} = \underbrace{\mathbf{alpha} + \overrightarrow{\mathbf{xs}} * \mathbf{beta}}$$

← Relation set in stone. No space for any error.

Demo (part 2, onwards)

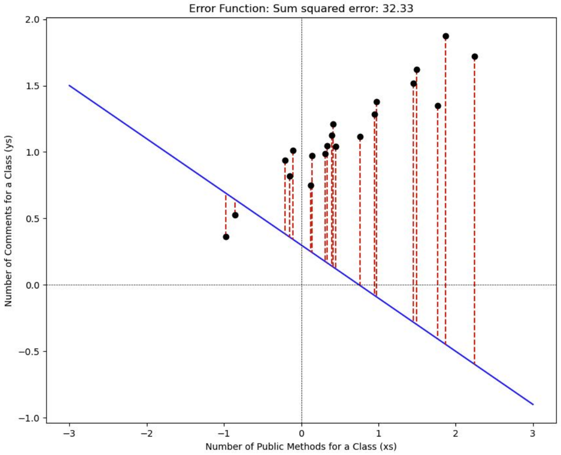
Find the code in “part2.py”.

```
# Define our model's alpha and beta (we need to explore alternatives here).
alpha = 0.3
beta = -0.4

def model(x):
    return alpha + beta * x

# Ceck how good it fits: Calculate the sum squared error on the data.
sum_squared_error = sum(np.power(ys - model(xs), 2))
```

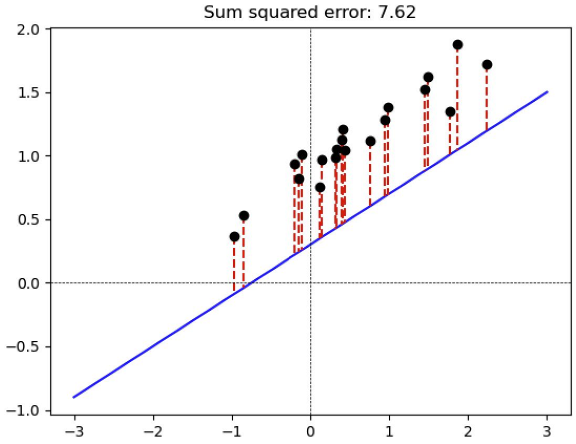
There error for different alphas and betas.

better

(alpha=0.3, beta = -0.4)



(alpha=0.3, beta = 0.4)

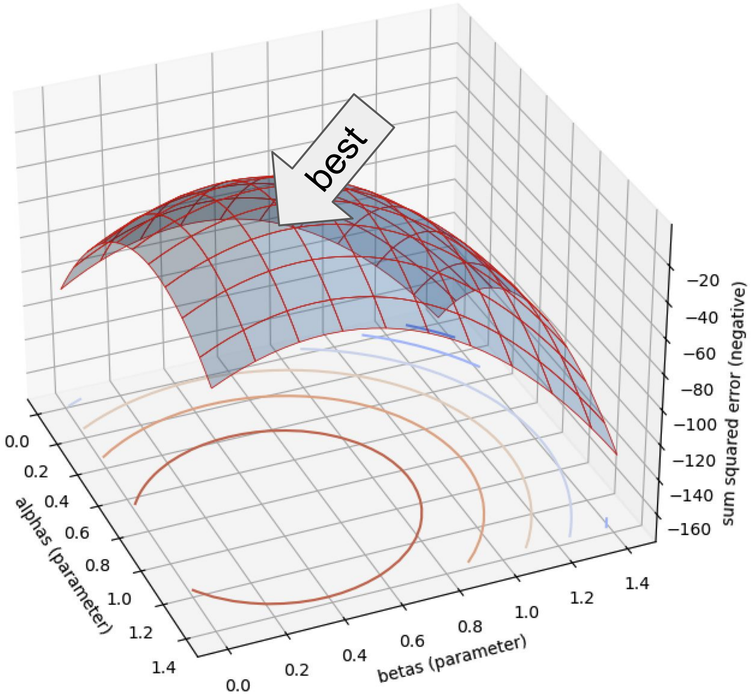


- Black are xs, ys
- Blue is the model
- Red is the error.

Grid search

(See code part 3)

```
# Model fitting by searching (learning) parameters.
for alpha in np.linspace(0, 1.4, grid): # Parameter 1
    for beta in np.linspace(0, 1.4, grid): # Parameter 2
```



We search through combinations of parameters.

Expensive if we search for many parameters.

I chose to plot the negative error, which we need to maximize.

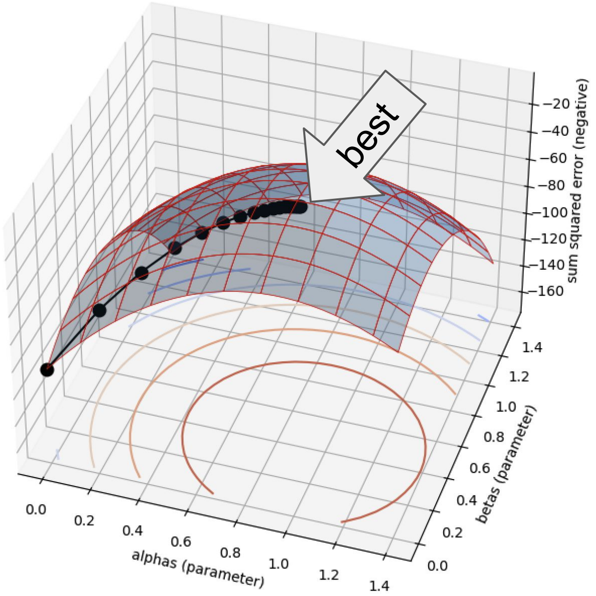
Johannes Härtel – johannes.hartel@vub.be

Gradient Decent

(See code part 4
and 5`

(TensorFlow’s autodiff)

```
[dSQE_dalpha, dSQE_dbeta] = tape.gradient(neg_sum_squared_error, [alpha, beta])

# Update alpha and beta (assign_sub subtracts value from this variable).
alpha.assign_add(0.001 * dSQE_dalpha)
beta.assign_add(0.001 * dSQE_dbeta)
```



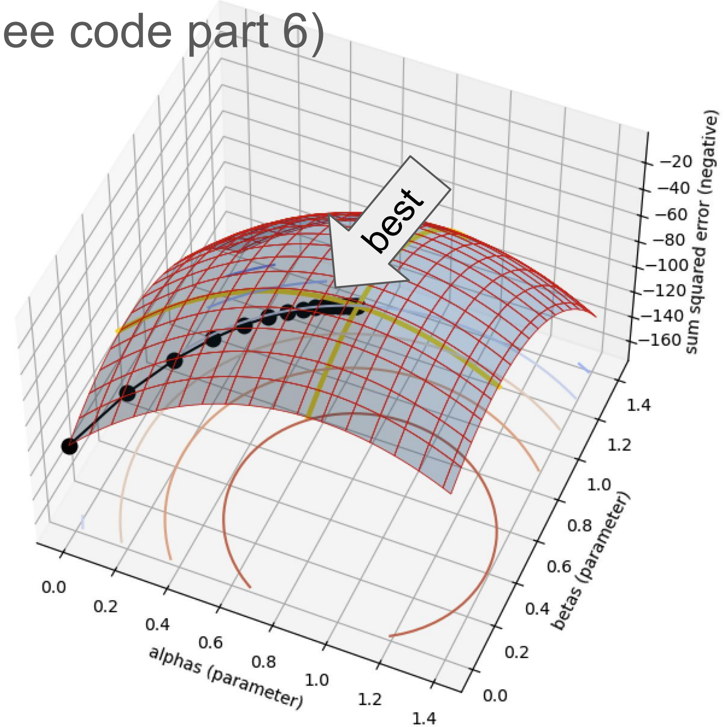
We only search for the peak using the gradient of the error.

Loves local optimum.

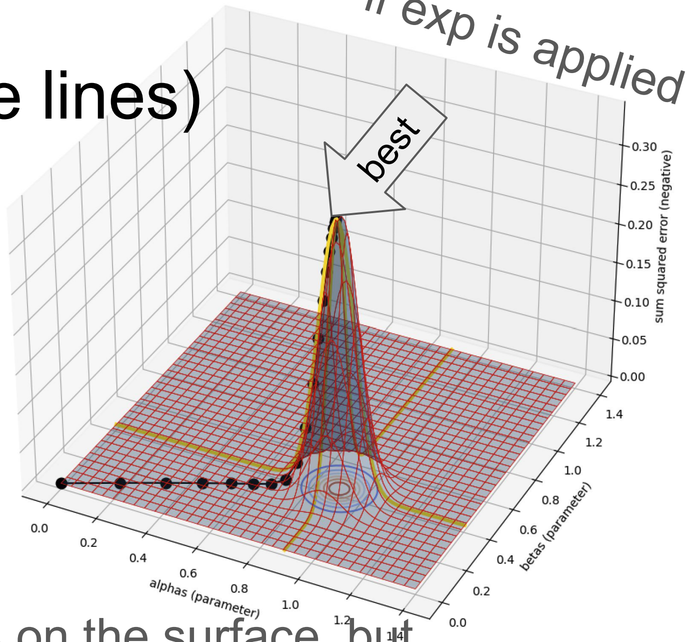
Only report on the peak.

Quadratic Approximation (orange lines)

(See code part 6)



if exp is applied

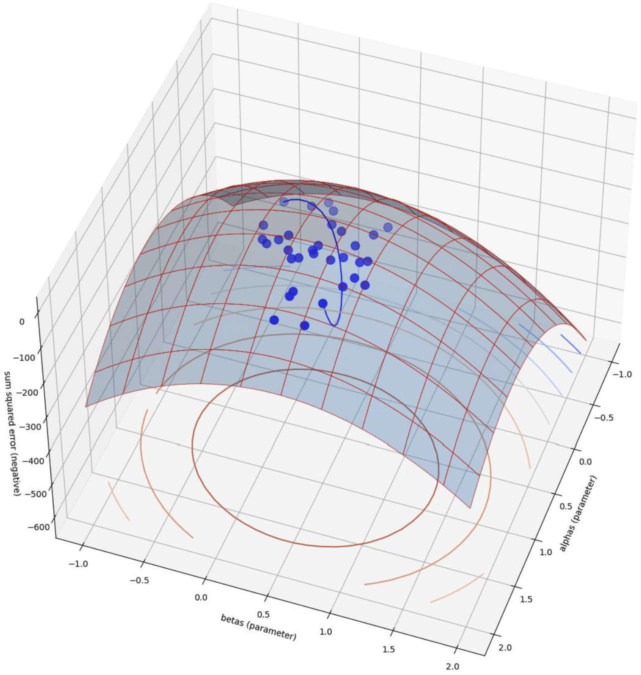


Reports on the surface, but does not work for arbitrary shaped surfaces.

Basis for confidence estimates in many statistic models

Hamiltonian Monte Carlo

(See code part 7)



A particles' movement, influenced by the gradient, and random flicks every n steps.

Approximates arbitrary surfaces.

Basis for modern statistics.

Johannes Härtel – johannes.hartel@vub.be

Summary

- We have seen how to write and use models in different ways.
- We can also use APIs to do this.