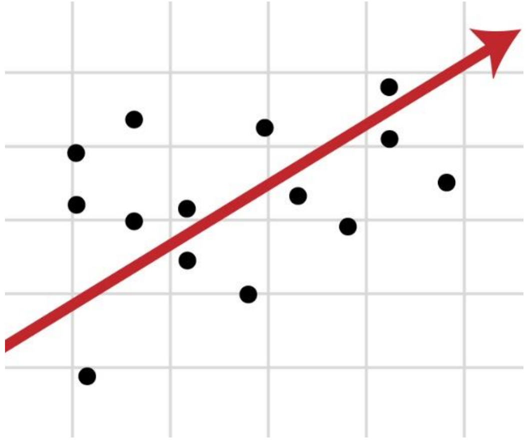
Regression Models

Taking our first steps to modelling data



Yordan Darakchiev

Technical Trainer

iordan93@gmail.com

sli.do

#DataScience

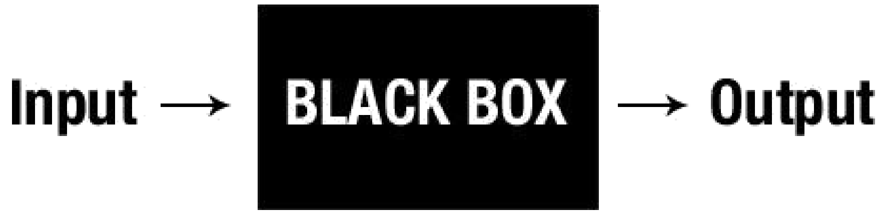
Table of Contents

Problem Overview

Types of modelling

Data Modelling

- As part of the data science process, we want to get a clear idea of what processes generate our data
  - Scientific method: Form a hypothesis and test it
  - Extension: Find a way to understand what's in the data
    - We already did this a lot of times: "mental models" captured our ideas
- A stricter way of modelling
  - Treat the data generating process as a function
    - "Black box"
  - Make some assumptions
  - Create a simplified version of reality under your assumptions
  - Check your model against reality
    - ⇒ Create better and more complex models



A Quick Peek at Machine Learning

- Machine learning is "making computers learn with experience, without being explicitly programmed"
  - Similar to how humans learn
- It's all about models
  - ML follows the same processes as we're going to do
  - ML algorithms are basically "function approximations"
    - Each algorithm does its own thing, i.e., has different assumptions, scope and performance
- It's also about selecting the best model
  - There are many "helper algorithms" to do so – either fully automated or semi-automated
    - Visualization algorithms,
    - Fine-tuning algorithms
    - Model selection algorithms, etc.

A Quick Peek at Machine Learning (2)

- There are a lot of classes of problems
- The most used two
  - **Regression** – model a function which returns a number (i.e., returns a continuous variable)
    - Example: predict the temperature tomorrow
  - **Classification** – model a function which tries to differentiate between two (or more) predefined types of things
    - Example: predict if an image is of a cat or not
- The essence: once we assume a model, **we can make predictions** about function outputs
  - Thus, we can capture patterns in an otherwise unpredictable world

Linear Regression

Predict continuous values…
and torture first-semester students

Linear Regression Intuition

- Regression – predicting a continuous variable
- Problem statement
  - Given pairs of $(x; y)$ points, create a model
    - Under the assumption that y depends linearly on x (and nothing else)
- Linear regression model
  - $y = ax + b$
    - a, b – unknown parameters
    - Example: $y = 2x + 3$
  - Real case: we have many sources of error
    - So, the relationship we observe, cannot be perfect
    - There is some noise added to our data
      - $y = ax + b + \varepsilon$, ε – noise
    - We **don't want** to model the noise, only the "useful function"

Generating Data Points

- Generating a few "ideal" data points is easy

```
x = np.linspace(-3, 5, 10)
y = 2 * x + 3
plt.scatter(x, y)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

- Adding noise – draw from a random distribution

```
y_noise = np.random.normal(size = len(y))
y_with_noise = y + y_noise
plt.scatter(x, y_with_noise)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```

- If we want, we can even configure the "size" of our noise
  - More noise = worse data = less accurate predictions

First Attempt at Modelling

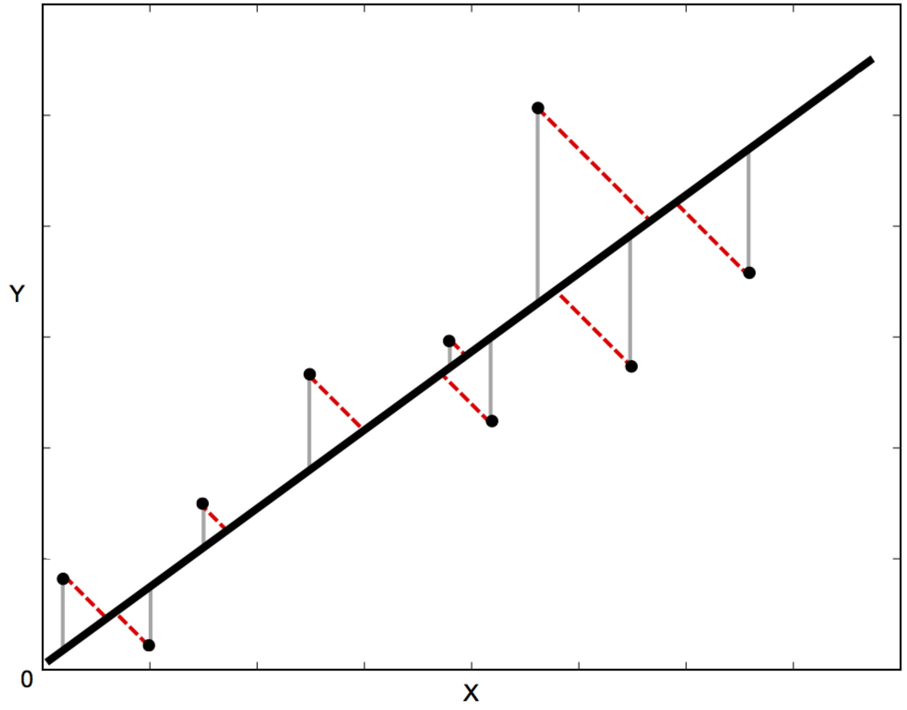
- We know the process was linear
  - Why don't we simply guess a few functions?
    - Remember: what we need to know are the parameters a and b

```
for y_guess in [3 * x + 8, 4 * x + 3, -2 * x]:
  plt.scatter(x, y_with_noise)
  plt.plot(x, y_guess)
  plt.show()
```

- We can see that some functions perform much better than others
  - Idea: the best function lies "closest" to all points
  - Meaning
    - Try to measure the distances from all points to the line
    - See when these distances are smallest
      - This will be the best line

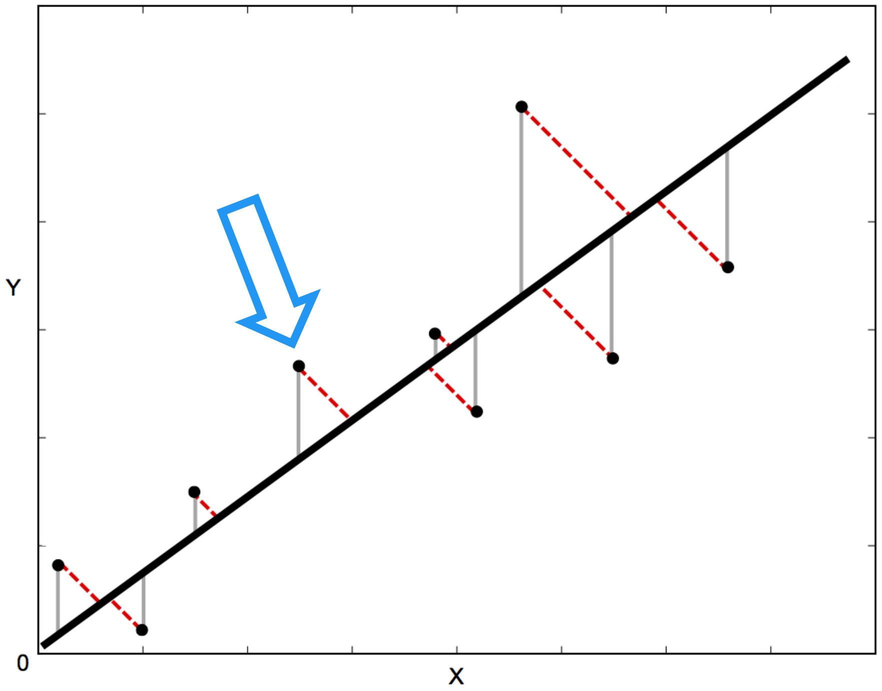
Distances

- By definition, the distance from a point A to the line l is measured on the perpendicular from A to l
  - Red dashed lines
  - This is correct but very computationally expensive
- Another approach: consider vertical distances
  - Gray solid lines
  - Equivalent measures (for our purposes)
    - You can prove it to yourself



Distances (2)

- Look at a point and its projection
  - x-coordinate: the same
  - y-coordinate
    - Point: we know it from the start
    - Projection: we can calculate it
- Calculating the projection $\tilde{y}$
  - It's whatever the model function produces for x
  - $\Rightarrow \tilde{y} = ax + b$
- Distance becomes a very simple difference
  - $d = y - \tilde{y}$
    - But… now distances can be negative



Distances (3)

- To make distances positive, we can do a lot of things
- Simplest: take the absolute value
  - This is used sometimes
    - Mean absolute error, MAE
  - Although it works quite well, there are a few problems with it
  - E.g. $d = 0$ at the "perfect" line
- Better: square the distance
  - It's also non-negative everywhere but...
    - Is almost always > 0
    - Emphasizes bigger errors more (can be good or bad)
  - This is called mean square error (MSE)
- New definition of distance
  - $d = (y - \tilde{y})^2$

Cost Function

- We want to somehow account for all points
  - We can simply sum all distances to get a measure of "the total distance" from all points to the line
  - Since we can have 4, 10, 100 or 10^9 points, we also need to normalize the error
- The sum of distances now becomes
  - $J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$
  - This is what we call our **total cost function**
    - Beware of **confusing terms**: d is usually called a "loss function", while J is the "(total) cost function"
  - This is an estimation of the total distance
  - **Minimizing this function will produce the best line**

Calculating the Cost Function

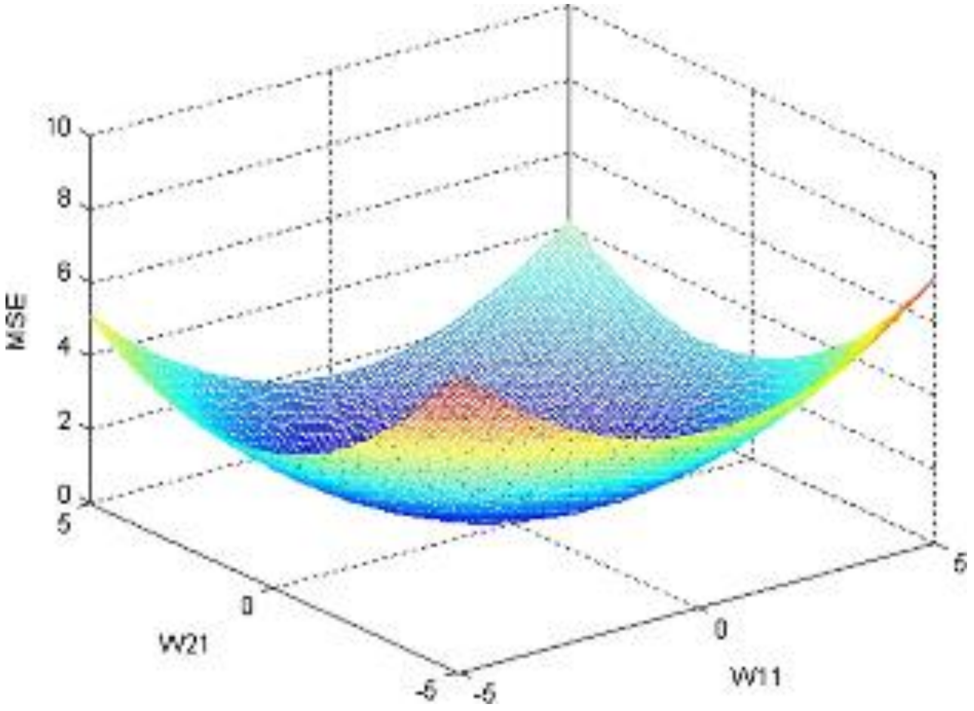
- The code is pretty simple
- Given points x, y and a line with parameters a, b we can simply substitute in the formula above
  - First, for each x, compute $\tilde{y} = ax + b$
  - After that, compute the distances $(y - \tilde{y})^2$
  - Return the sum of all distances, divided by the number of points

```
def calculate_loss(x, y, a, b):
    y_predicted = a * x + b
    distances = (y - y_predicted) ** 2
    return np.sum(distances) / len(x)
```

  - Now that we have a quantifier, we can go back to our three guessed lines and calculate their loss functions
    - It will give us the intuition of what we're dealing with

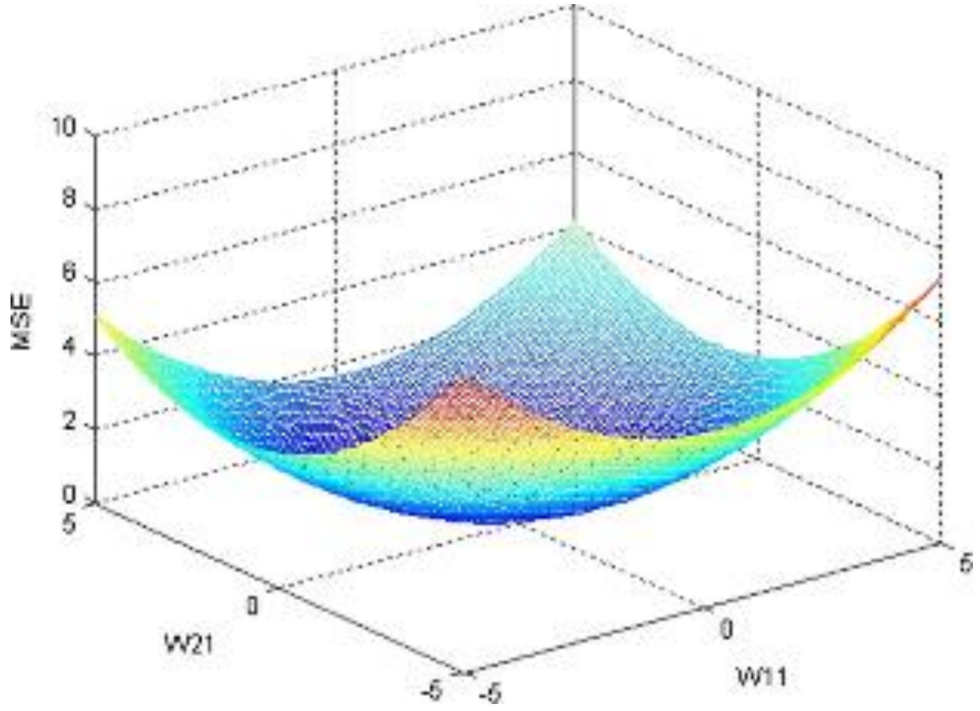
Inspecting the Cost Function

- Note that J does not depend on x and y
  - x and y are already fixed – we don't touch the data at all when we try to model it
  - $\Rightarrow J$ depends only on the line parameters a, b
    - In math jargon, J is a function of a and b: $J = f(a, b)$
- Also note the form of J: it's $(\cdots)^2$
  - This is a paraboloid (3D parabola)
  - See how varying a and b gives us a different output number for J
  - It has exactly one min value
    - And we can see it
  - Our task: find the parameters a, b which make J as small as possible



Minimizing the Cost Function

- Intuition
  - If the plot was a real object (say, a sheet of some sort), we could slide a ball bearing on it
  - After a while, the ball bearing will settle at the "bottom" due to gravity
  - We could measure the position of the ball and that's it :)
- More "nerd speak"
  - This is the same task – we have a gravity potential energy that the ball tries to minimize
    - When it's minimal, the ball remains in stable equilibrium

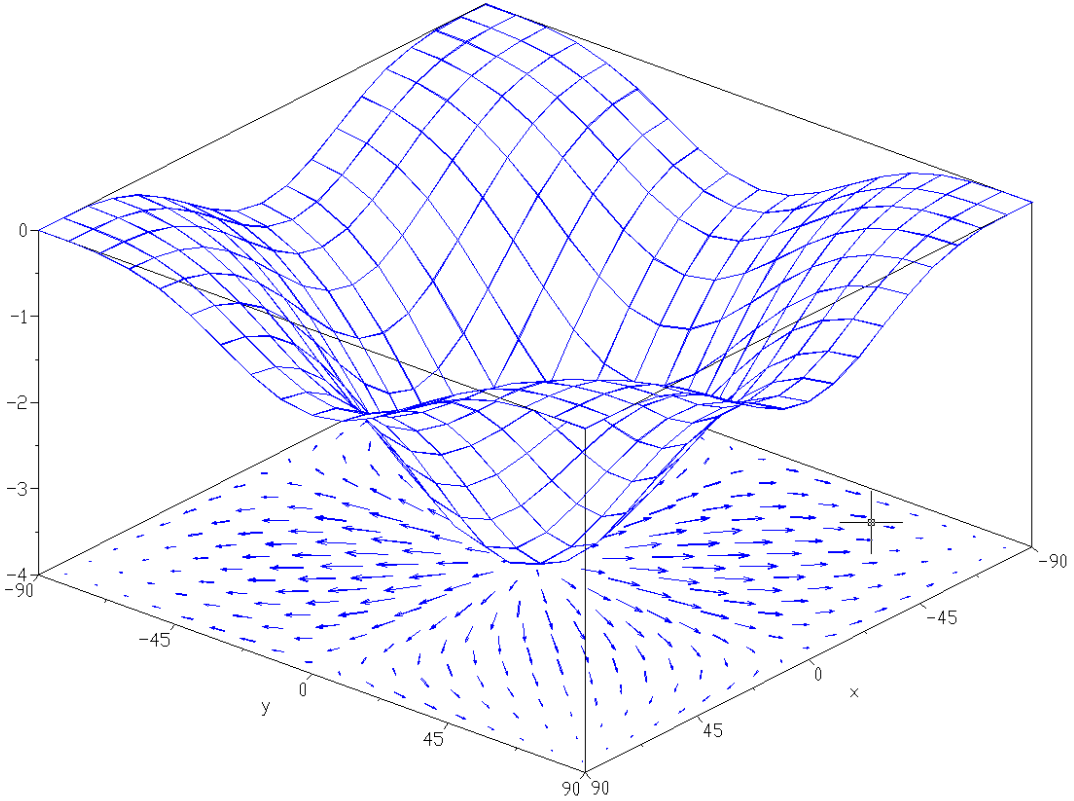


Minimizing the Cost Function (2)

- Turns out, we can also do this using calculus
  - In many dimensions
- We can find the optimal parameters right away
  - Because the function is really simple
  - But we'll stick to another approach because this is what is useful for all other ML tasks
- We'll try to replicate the example with the ball
  - Basically, we'll try to slide (descend) over the function surface until we reach the minimum
  - This method is called **gradient descent**

Gradient Descent

- We know what **descent** is
  - How about **gradient**?
- The gradient (let's call it g for now) is a vector function
  - Like J, g accepts two values a and b
  - g returns a vector which shows where **the steepest ascent** is
  - g is all arrows on the picture
  - Interpretation
    - The length of the vector tells us how steep the maximum is
      - Long vector = very, very steep; short vector = relatively flat
    - The direction of the vector tells us where to go in order to get there



Gradient Descent (2)

- Gradients will almost work
  - Except they show us the highest point, and we're looking for the lowest one
  - Solution: just take the negative gradient $-g$
  - Ascending on $-g$ is the same as descending on g
- This is good now, but how is the gradient defined?
  - We saw from the picture that it’s related to a function
  - The gradient of a function $J(a, b)$ is a vector $g(a, b)$ with the following components
  - $g_a = \frac{\partial J}{\partial a}, g_b = \frac{\partial J}{\partial b}$
  - The ∂ symbol means "partial derivative"
    - If you don't understand this, you only need to know that partial derivatives are quite easy to calculate

Gradient Descent (3)

- Remember that $J = \frac{1}{n}\sum(y_i - \tilde{y}_i)^2$
  - We can prove that
  - $\frac{\partial J}{\partial a} = -\frac{2}{n}\sum x_i(y_i - \tilde{y}_i); \frac{\partial J}{\partial b} = -\frac{2}{n}\sum(y_i - \tilde{y}_i)$
- This can be implemented easily

```
a_gradient = -2 / len(x) * np.sum(x * (y - (a * x + b)))
b_gradient = -2 / len(y) * np.sum(y - (a * x + b))
```

  - Note how this code makes use of `numpy` and its extremely easy operations on arrays
- Now, if we know x, y, a, b we can calculate the gradient vector
  - You'll also see the gradient of J being denoted as ∇J
    - This is simply math notation

Gradient Descent (4)

- Let's now get to the real descent
- Iterative algorithm – perform as long as needed
  - Start from some point in the $(a; b)$ space: $(a_0; b_0)$
  - Decide how big steps to take: number α
    - Called "learning rate" in ML terminology
  - Use the current a, b and x, y to compute ∇J
    - $-\nabla J_a$ tells us how much to move in the a direction in order to get to the minimum
    - Similar for $-\nabla J_b$
  - Take a step with size α in each direction
    - $a_1 = a_0 - \nabla J_a; b_1 = b_0 - \nabla J_b$
    - $(a_1; b_1)$ are the new coordinates
  - Repeat the two preceding steps as needed
    - Usually, we do this for a fixed number of iterations

Gradient Descent Code

- Gradient descent step

```
def perform_gradient_descent(x, y, a, b, learning_rate):
    a_gradient = -2 / len(x) * np.sum(x * (y - (a * x + b)))
    b_gradient = -2 / len(y) * np.sum(y - (a * x + b))
    new_a = a - a_gradient * learning_rate
    new_b = b - b_gradient * learning_rate
    return (new_a, new_b)
```

- Entire process: 1000 iterations

```
model_a, model_b = -10, 20 # Start points; can be anywhere
alpha = 0.01 # Learning rate
for step in range(1001):
    model_a, model_b = perform_gradient_descent(
        data_x, data_y, model_a, model_b, alpha)
    if step % 100 == 0:
        error = calculate_loss(data_x, data_y, model_a, model_b)
        print("Step {}: a = {}, b = {}, J = {}".format(
            step, model_a, model_b, error))
print("Final line: {} * x + {}".format(model_a, model_b))
```

Results and Interpretation

- Going through the entire process, we now have a line $y = ax + b$ which describes our data in the best way
  - We could plot the evolution of J to see that it always decreases
    - If it doesn't, this indicates a problem with our algorithm
- This was a lot of work
  - Thankfully, there are libraries that hide away all that complexity for us
  - `scikit-learn` is the most popular of them
    - Arguably, the most popular of the `scikits` as well
  - Also, generalizes trivially to more dimensions

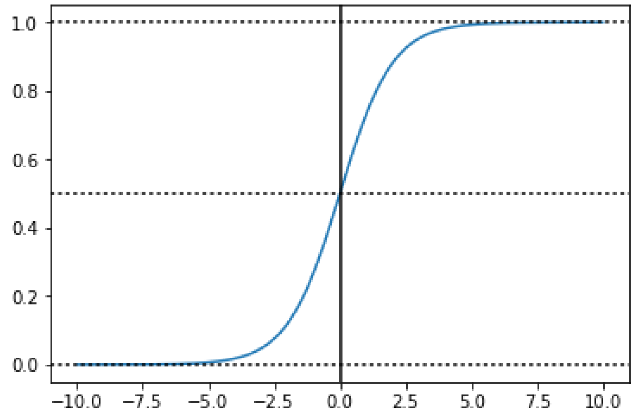
```python
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(data_x.reshape(-1, 1), data_y)
print(model.coef_, model.intercept_)
```

Logistic Regression

Use a regression model to classify

Logistic Regression

- The name is a bit misleading
  - This is used for classification
- Two classes, 0 and 1
  - Can generalize to more classes using a "trick"
- A function to discriminate: **sigmoid**
  - $x < 0 \Rightarrow y = 0; x > 0 \Rightarrow y = 1$
  - We'll look at the implementation later
- Loss function
  - Similar to the linear regression cost function
- Gradient descent
- Usage in `scikit-learn`




```
from sklearn.linear_model import LogisticRegression
```

Overview of the Process

- We dealt mainly with the modelling part
  - It's only a piece of the puzzle
- Many algorithms to choose from
  - Each with its own features and drawbacks
- Many ways to test that we're on a correct path
- The end result depends mainly on
  - The person working on the dataset
  - The data quality
  - Less prominent but also worth mentioning
    - Data size (bigger is usually better)
    - Data acquisition and sampling processes

Summary

- Problem overview
  - Regression, classification
  - Machine learning: putting it all together
- Linear regression
  - Motivation, derivation, usage
  - More involved example
- Logistic regression
  - Motivation, usage

Questions?