Big Data and Data Mining

Supervised Learning

Flavio Bertini

flavio.bertini@unipr.it

Regression

- The regression problem asks to predict a numerical variable's value given the values of other variables
- Easiest example is with two variables x and y:
  - x is the variable in input
  - y is the variable we want to predict

x →	**y**
1	3
2	5
3	7
4	9
5	11

Training data: x is the input data, y is the label data

Learning

Learning task: **what's the mapping from x to y?**

Testing

Try to "learn" from the training data an hypothesis function h

What is h(6)? That is, what's y when x is 6?

x →	**y**
6	?

Regression example 1/2

- We have the data from 10 thousand houses in the U.S.

m=10000

City	Rooms	SQM	Price
Los Angeles	3	130	420000
Los Angeles	2	60	380000
...	...	...	...
Albuquerque	2	140	220000
Albuquerque	3	150	250000

- **Goal**: learn a function (model) that can infer the *Price* given all the other variables, for houses not in the 10 thousand dataset:

X

City	Rooms	SQM
Los Angeles	5	230

y

Price
?

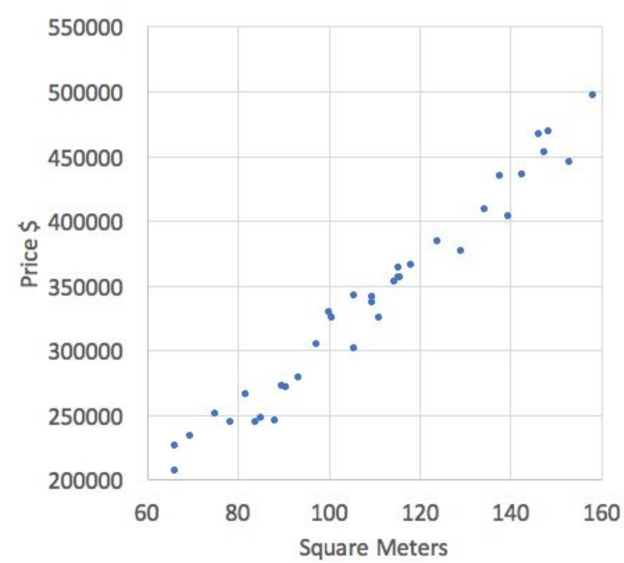
Regression example 2/2

- The houses' prices example is an example of **regression** task:
  - Among the variables there is the output variable (Price)
  - The output variable is the *ground truth*, because it's the actual price of the house
  - The output variable we want to predict is a numerical value

City	Rooms	SQM
Los Angeles	5	230

Price
?

- To visualize things better we will have examples with only two variables (one used as input one as output)
- All the process can be generalized for n variables



Linear regression

- We defined regression as the general task of predicting the real value of a variable using the other variables as input
- The model we want to learn is a function *f* from *n* real values x_i to one real value *y*:

$$y = f(x_1,\dots,x_n)$$

- The learned model is called *hypothesis* function *h*:

$$y = h(x_1,\dots,x_n)$$

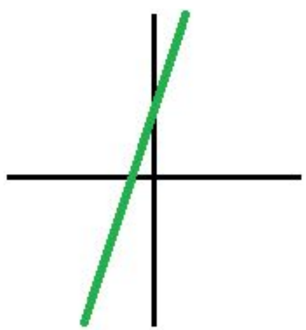
- When ***h* is a linear function** on the input variables x_i, the regression task is called **linear regression**

Linear function

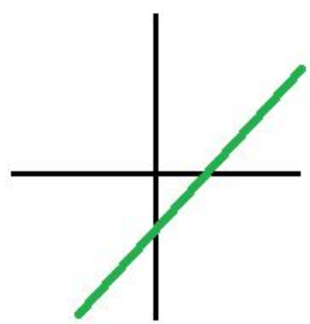
- A linear function is a function of this form:

$$y = f(x_1,\dots,x_n) = w_1x_1 + \dots + w_nx_n + b$$

- When n = 1 (only one input variable), the plotted function has the **shape of a straight line**. When n = 2 it takes the shape of a straight plane etc.
- Straight lines and straight planes are examples of linear functions

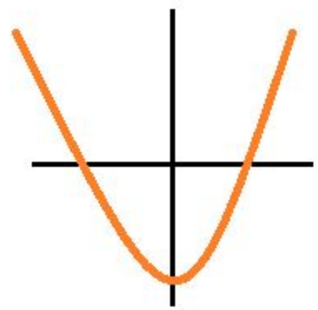


y = 4 + 5x

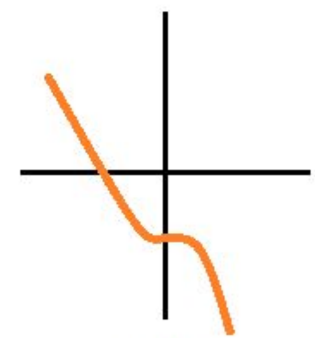


y = x - 3

Linear functions



y = x^2



y = x^3 - 6

Non-linear functions

Linear regression: parameters

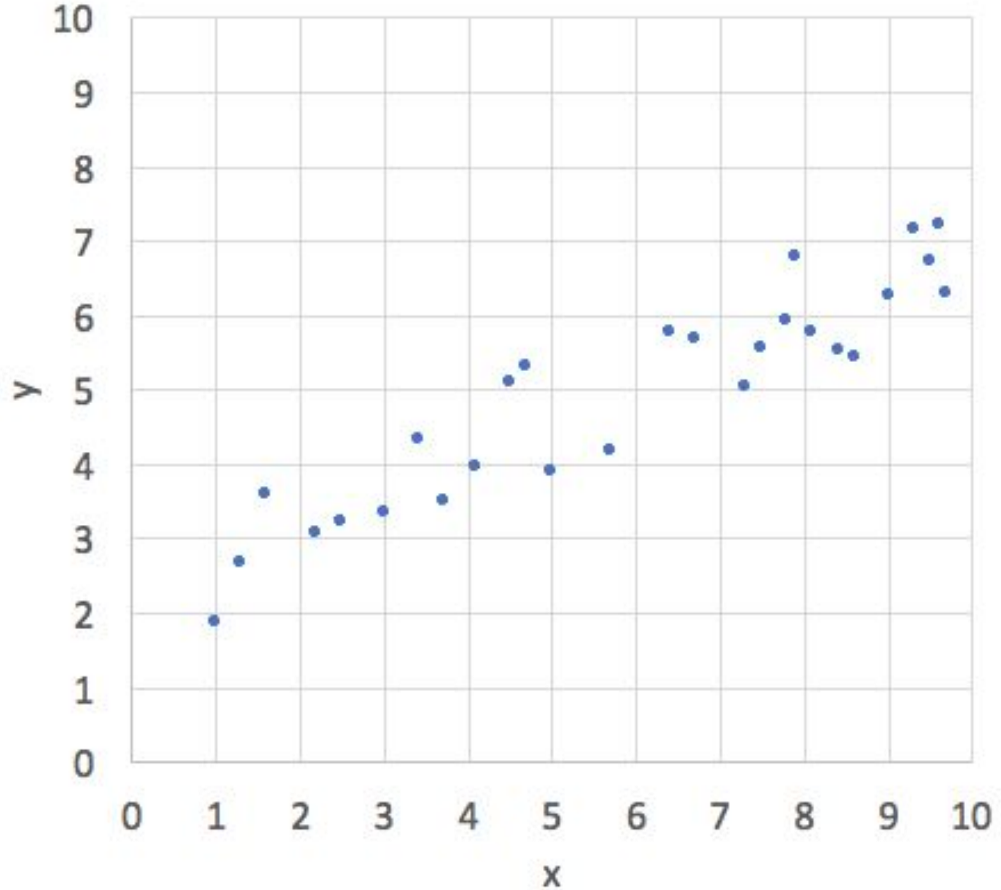
- Recall, a linear function is a function of this form:

$$y = f(x_1,...,x_n) = w_1x_1 + \dots + w_nx_n + b$$

- This means that the function that models our data is already defined!
- What's missing? **What are the parameters we need to "learn"?**
  - The x_i variables are the ones **given in input**, so we already have them (e.g., the square meters of the house)
  - The **w_i coefficients are not known!**
    - In a straight line, the coefficient is the **slope** of the linear function!
    - In ML they are also called **weights**: assuming that you have scaled your variables (e.g. between 0 and 1) they tell you **how much a variable contributes** to the final result (the *y*)
  - The **b** parameter is **also not known**!
    - In a straight line, this is the **y-intercept** of the line: the point where the line intercept the y axis
    - In ML this is also called **bias term** or **bias weight**

Example: data points

- Let's clarify these new notions with an example
- We have plotted some data points, each point is an observation with two variables: x and y



Example: slope 1/3

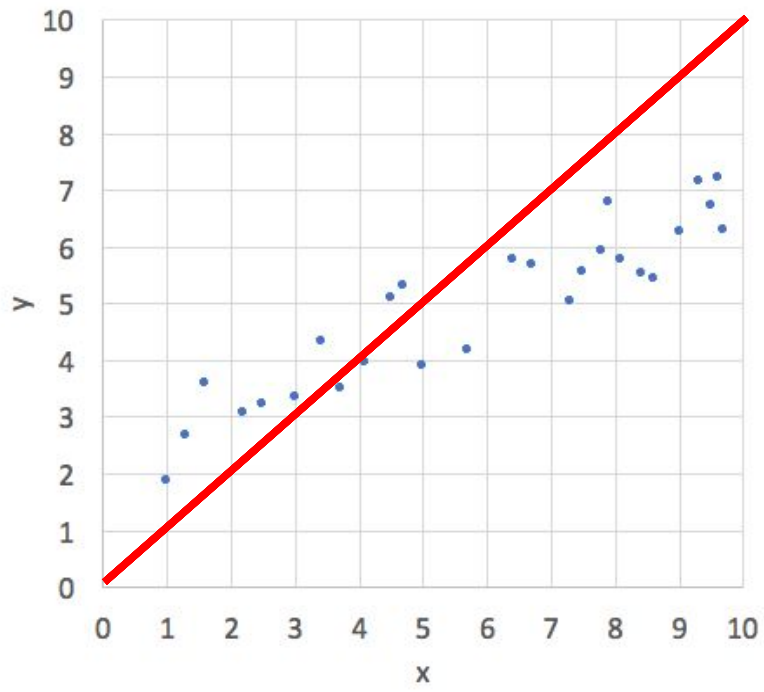
- Let's simulate a learning process: we need to learn the slope on the line
- To keep things simple, we focus on the angle only, setting the y-intercept to 0

1. **Let's try with a slope w = 1**: this means the line will have a 45° slope
2. Having a y-intercept (b parameters) set to 0, the hypothesis function is:

$$y = h(x) = wx = x$$

3. By plotting the line on the figure we can visually evaluate how much this hypothesis match the actual data

- Is this hypothesis good enough?
- We will see later how to **measure precisely the distance between the actual data and the hypothesis**
- But for sure can do better!



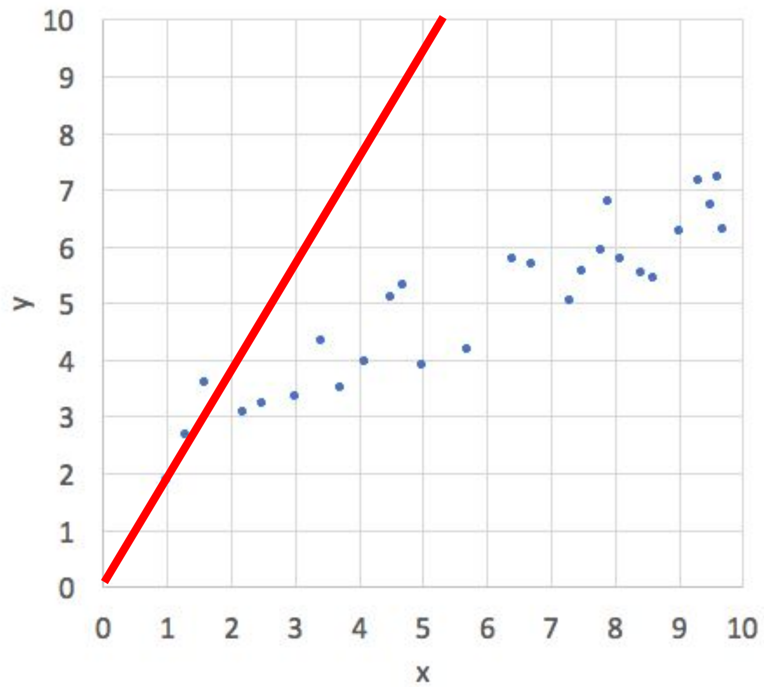
Example: slope 2/3

- Let’s simulate a learning process: we need to learn the slope on the line
- To keep things simple, we focus on the angle only, setting the y-intercept to 0

1. **Now let’s try with a slope w = 2!**
2. Having a y-intercept (b parameters) set to 0, the hypothesis function is:

$$y = h(x) = 2x$$

3. By plotting the line on the figure we can visually evaluate how much this hypothesis match the actual data



- Is this hypothesis better?
- The hypothesis **seems worse than before**

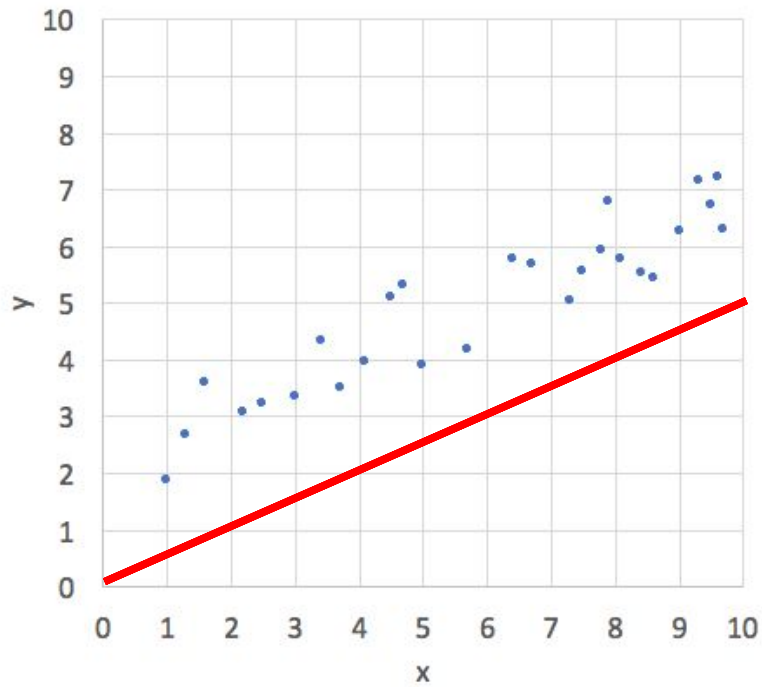
Example: slope 3/3

- Let's simulate a learning process: we need to learn the slope on the line
- To keep things simple, we focus on the angle only, setting the y-intercept to 0

1. **Now let's try with a slope w = 0.5!**
2. Having a y-intercept (b parameters) set to 0, the hypothesis function is:

$$y = h(x) = \frac{1}{2} x$$

3. By plotting the line on the figure we can visually evaluate how much this hypothesis match the actual data



- Is this hypothesis better?
- The hypothesis seems **quite accurate, at least for the slope**
- We should now learn **what's the best value for the bias term *b***

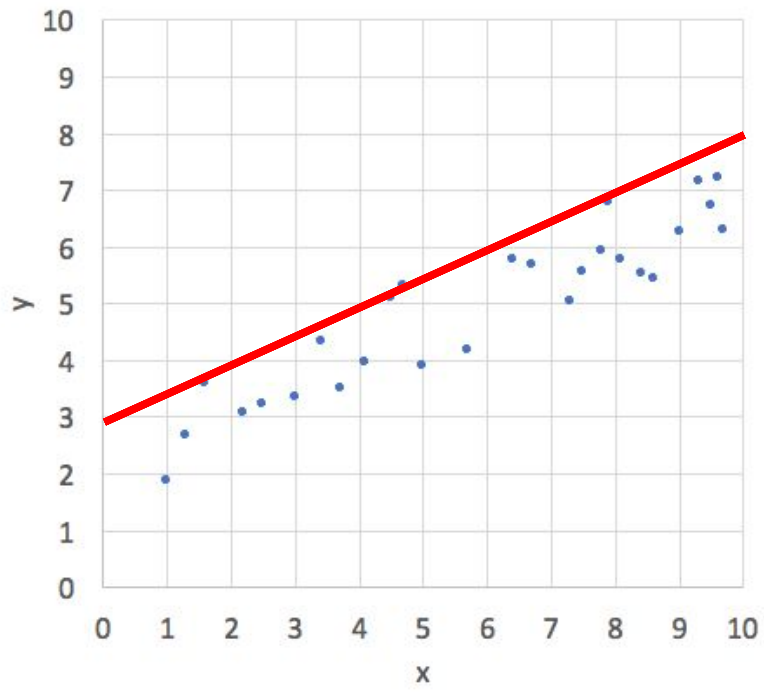
Example: bias term 1/2

- We have now a **good slope but a clearly wrong bias term** (the y intercept parameter)
- Let's try again some reasonable value

1. **Now let's try with a bias b = 3!**
2. Having already set the weight to 0.5, the hypothesis function is:

$$y = h(x) = \frac{1}{2} x + 3$$

3. By plotting the line on the figure we can visually evaluate how much this hypothesis match the actual data



- Are the points approximately around the line?
- **It seems slightly above. We should try a lower value for *b***

Example: bias term 2/2

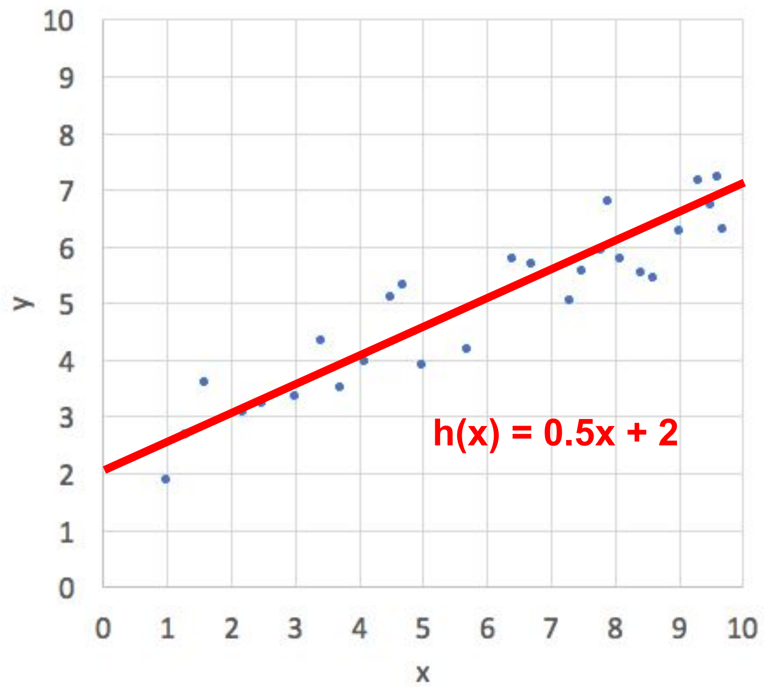
- We have now a **good slope but a clearly wrong bias term** (the y intercept parameter)
- Let's try again some reasonable value

1. **Now let's try with a bias b = 2!**
2. Having already set the weight to 0.5, the hypothesis function is:

$$y = h(x) = \frac{1}{2} x + 3$$

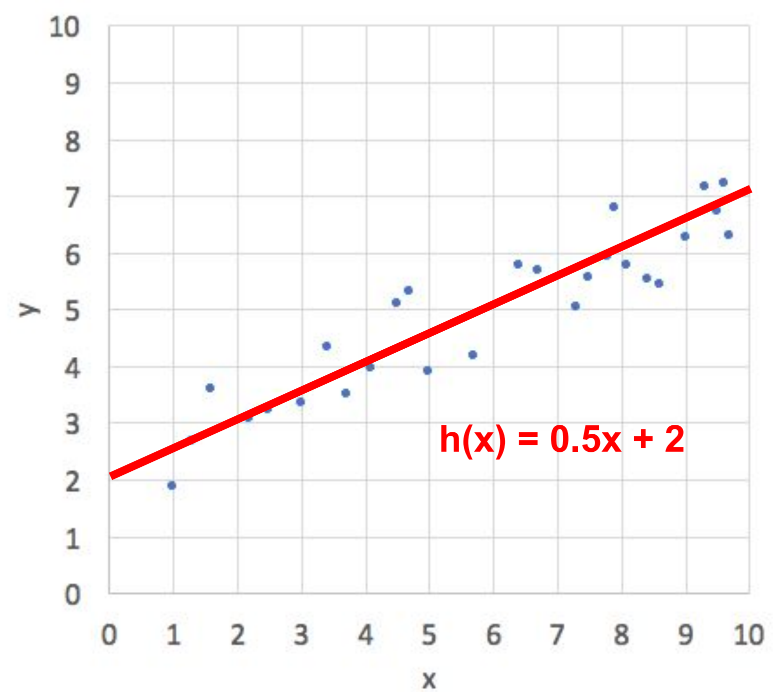
3. By plotting the line on the figure we can visually evaluate how much this hypothesis match the actual data

- **This hypothesis looks quite right!**



Example: observations

- The machine learning algorithm we have simulated is rather naive:
  - It has no precise way to measure how close we are to the optimal hypothesis
  - It makes almost random guesses to generate new hypothesis
- This means that a good machine learning model should have:
  - **A way to measure how good** (or how bad) **an hypothesis is**
  - **A smart algorithm that tunes the weights** until the measure of badness is very low (or conversely the measure of goodness is very high)



Loss function

- In ML, the measure used to evaluate the hypothesis is called a **loss function**
  - Intuitively, a **loss** function measures how **bad** the model is with respect to the actual data
- The most common it's the mean squared error, that is the **average squared distance** between the training data and the hypothesis
  - Mean because we want to consider all the *m* data points in the training set, so we average all the distances
  - Squared to enforce the distance to be non-negative

$$\frac{1}{m}\sum_{i=1}^{m}\big(\underbrace{h(x_i) - y_i}_{\text{Difference between the y predicted using the hypothesis function and the actual y}}\big)^2$$

Arithmetic mean

We square the difference to enforce non-negativity

Optimization

- Now that we have a tool to measure the error of an hypothesis function, we want the error to be the smallest possible
- This means the our ***objective function*** is to find the variables values (weights and bias) that ***minimize*** the error

$$minimize: \quad \frac{1}{m}\sum_{i=1}^{m}(h(x_i) - y_i)^2$$

- Using the *argmin* notation and replacing h(x) with the linear function:

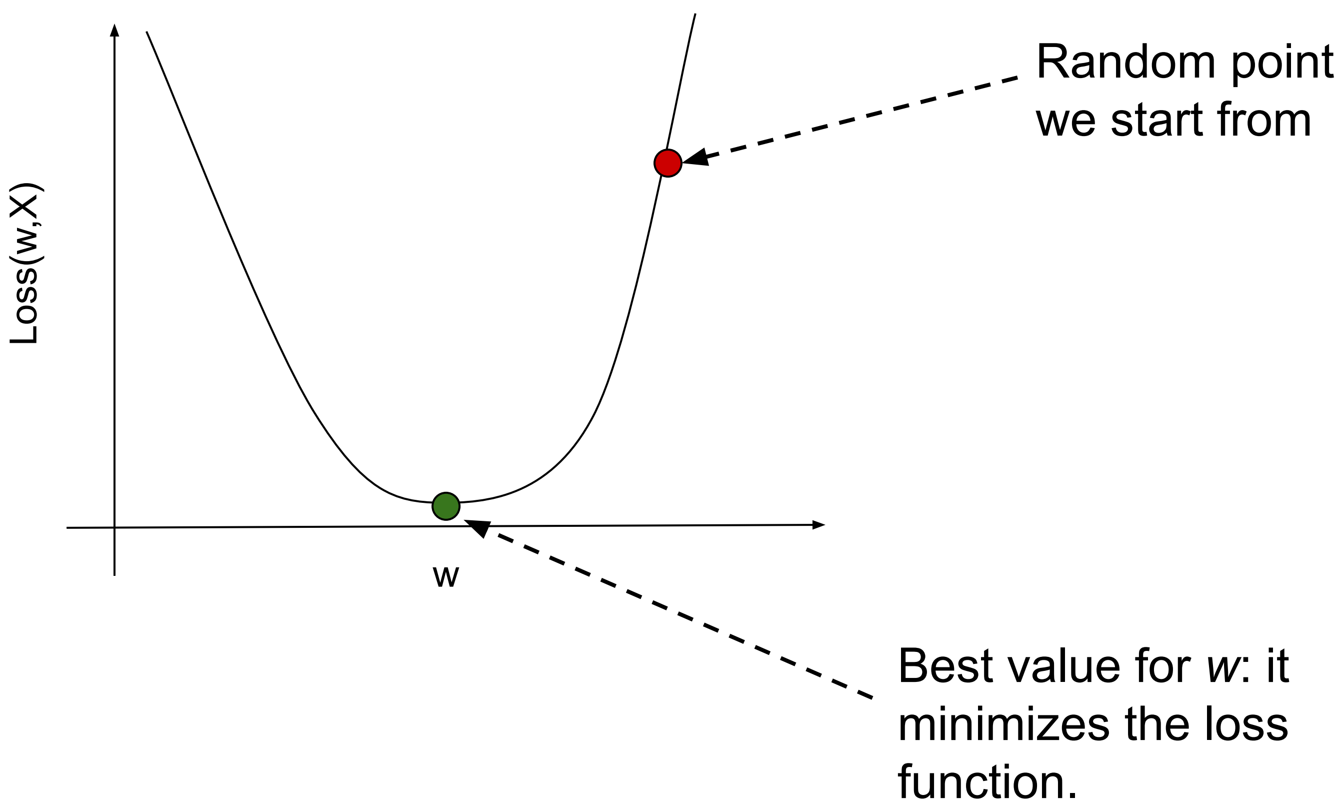
$$\operatorname*{argmin}_{w,b} \frac{1}{m}\sum_{i=1}^{m}((wx_i + b) - y_i)^2$$

Optimization algorithms

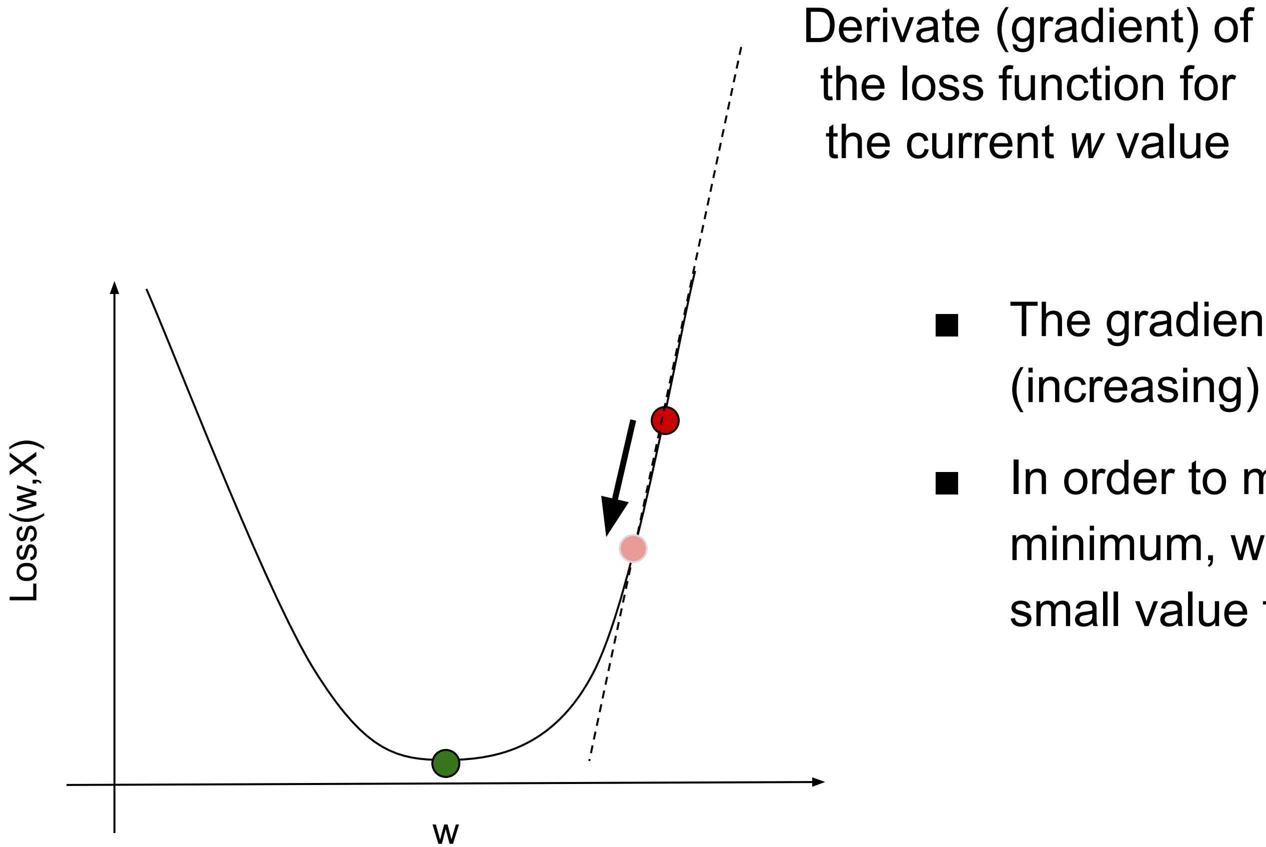
- Most popular and effective optimization algorithm in ML is **gradient descent**
- It **iterates** through 4 steps until the **loss is smaller than a tolerance value:**
  1. Compute the gradient (that is the derivative or slope) of the loss function
  2. The **sign of the gradient** tells us if the loss increase (positive) or decrease (negative) moving to the right. Recall that we want to reach the minimum:
     a. If it increase, we should move the weight to the left (subtract a small value)
     b. If it decrease, we should move to the weight to the right (add a small value)
  3. Update the hypothesis with the new weights
  4. Compute again the loss

Gradient descent example 1/6

- For simplicity, we consider only one parameter *w* to be learned
- The loss function with respect to *w* it's a convex function, thus it has a global minimum

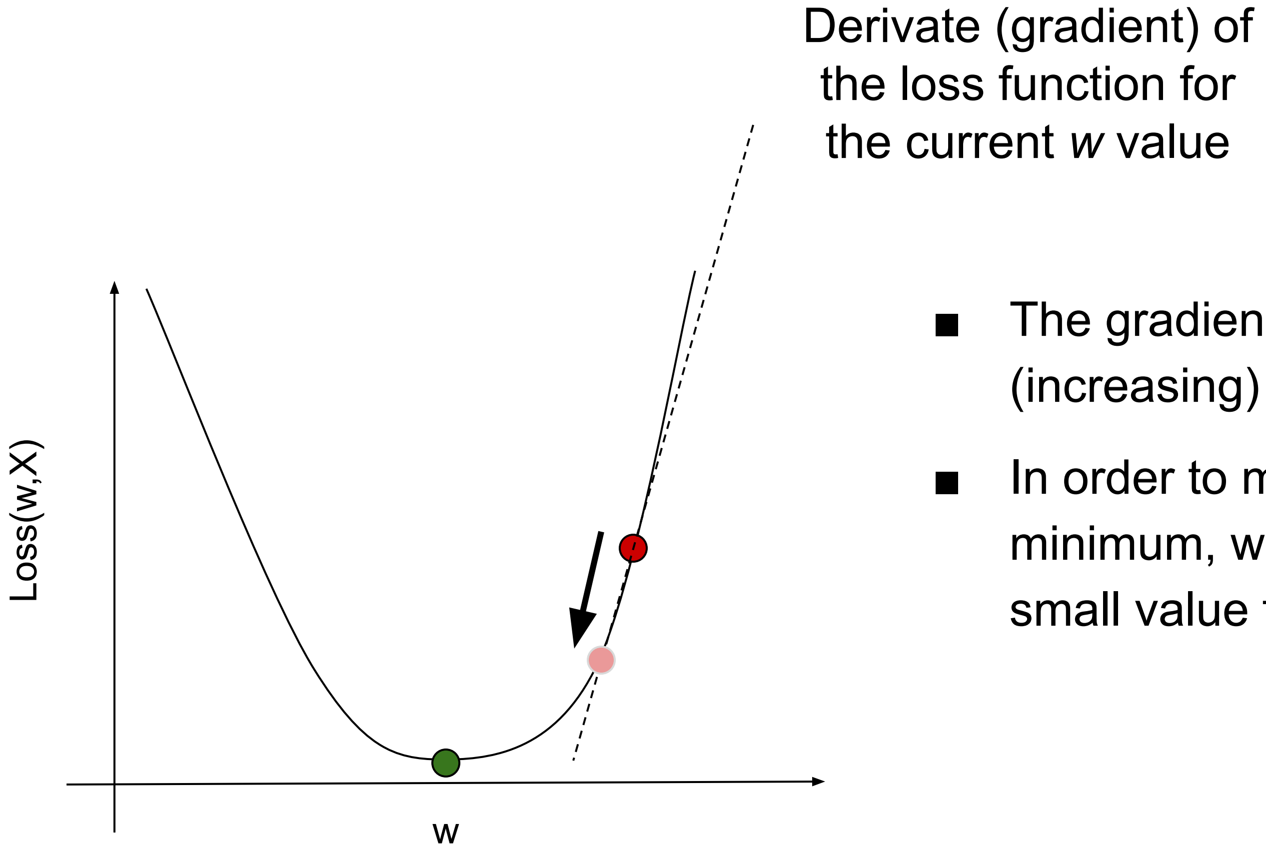


Gradient descent example 2/6



- The gradient is positive (increasing)
- In order to move toward the minimum, we should subtract a small value from *w*

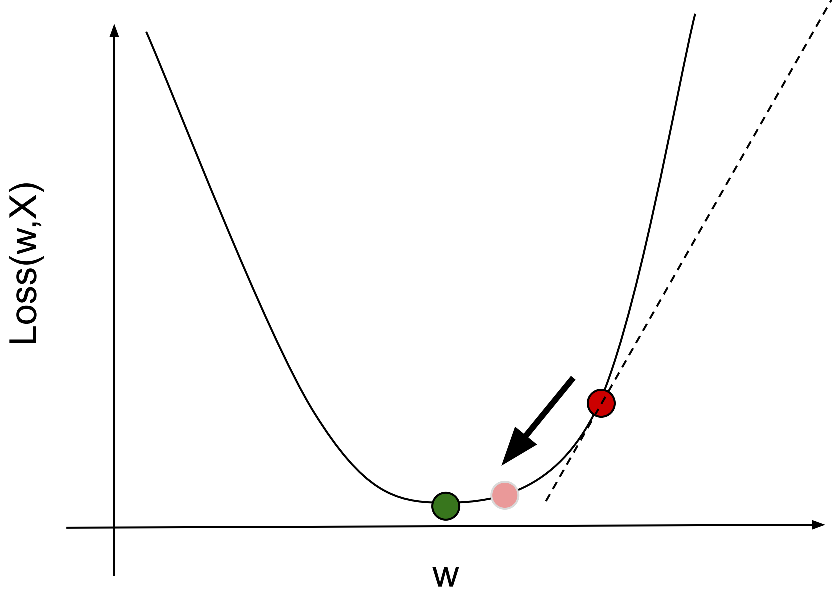
Gradient descent example 3/6



- The gradient is still positive (increasing)
- In order to move toward the minimum, we should subtract a small value from *w*

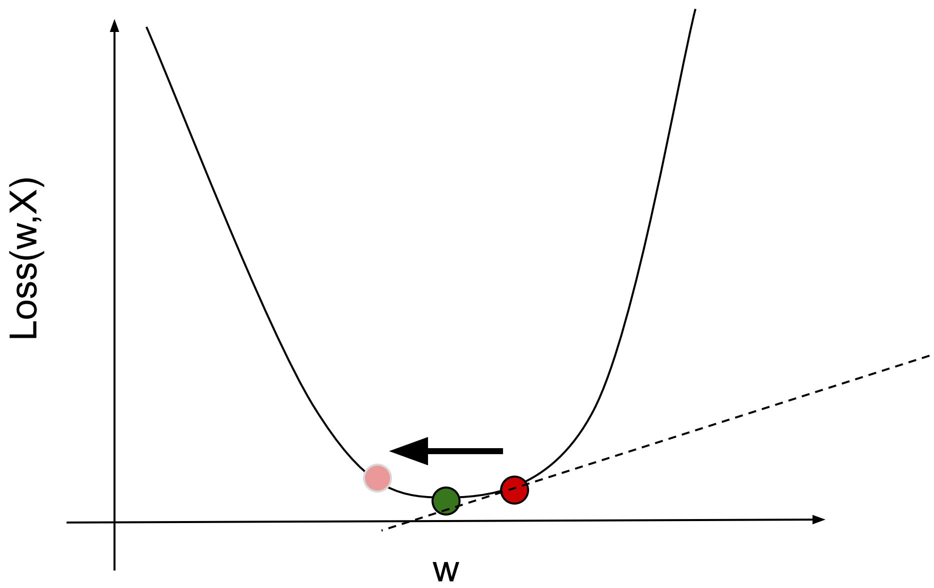
Gradient descent example 4/6

Derivate (gradient) of the loss function for the current *w* value



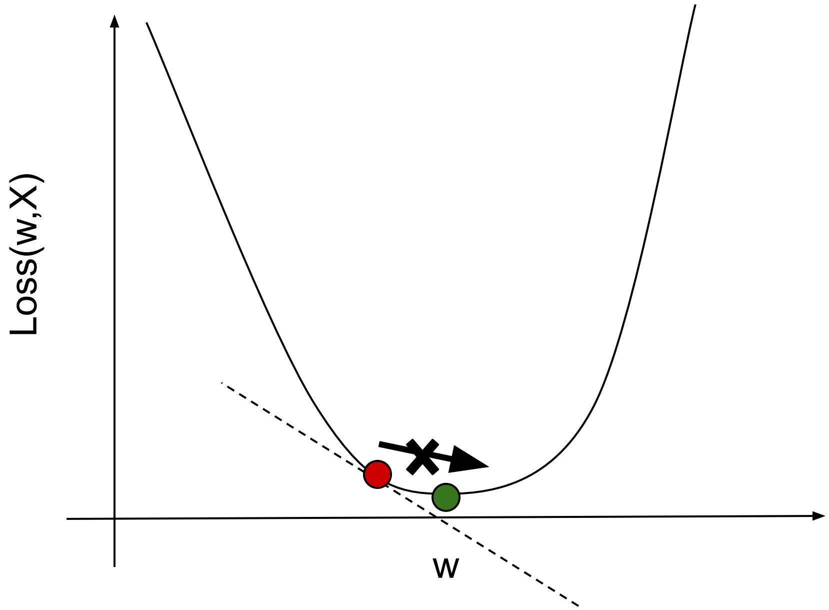
- The gradient is still positive (increasing)
- In order to move toward the minimum, we should subtract a small value from *w*

Gradient descent example 5/6



- The gradient is still positive (increasing)
- In order to move toward the minimum, we should subtract a small value from *w*

Gradient descent example 6/6



- The gradient **now is negative** (decreasing)
- In order to move toward the minimum, we should **add a small value to *w***
- However, we could decide that the loss is **small enough** and **stop our descent iterations**

Testing set

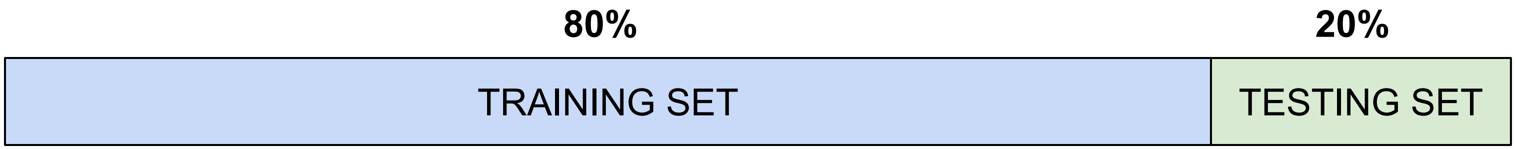
- **Important:** the loss function is made to evaluate the hypothesis **on the training set**! It measures the distance between the training data and the hypothesis
- If the loss is 0, it means that the hypothesis is **perfectly fitting** the training data
- Yet, this hypothesis could be inaccurate **with unseen data**: data that is not in the training set
    - This phenomenon is called ***overfitting***: the model is extremely good (overfitted) on the training data but unable to generalize on new data
- In order to actually test our model, we must use a set of data that the learning algorithm has never seen, that is the ***testing set***

Train/test split

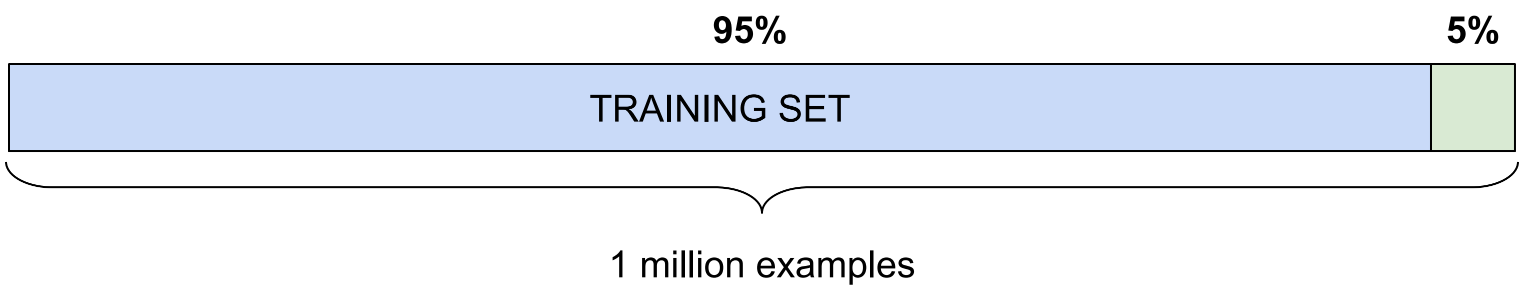
- Usually in a data mining process the data **is not already** splitted into training and testing set
- In splitting between training and testing set you should consider the following:
  - The more data you have in the training the better will perform the learned model
  - The more data you have in the testing the better will be the estimation of accuracy (*e.g.,* if you guess just one example you will have 100% accuracy, but it's not a significant estimation)
- IMPORTANT: both training and testing set **must come from the same distribution**. For example:
  - You can't train on the houses' prices in L.A. and test on the houses in N.Y.
  - You can't train on pictures taken with the smartphone and test it on pictures taken from Google Images

Train/test split ratio

- The rule-of-thumb is to split **80/20** following the Pareto principle: 80 for training and 20 for testing



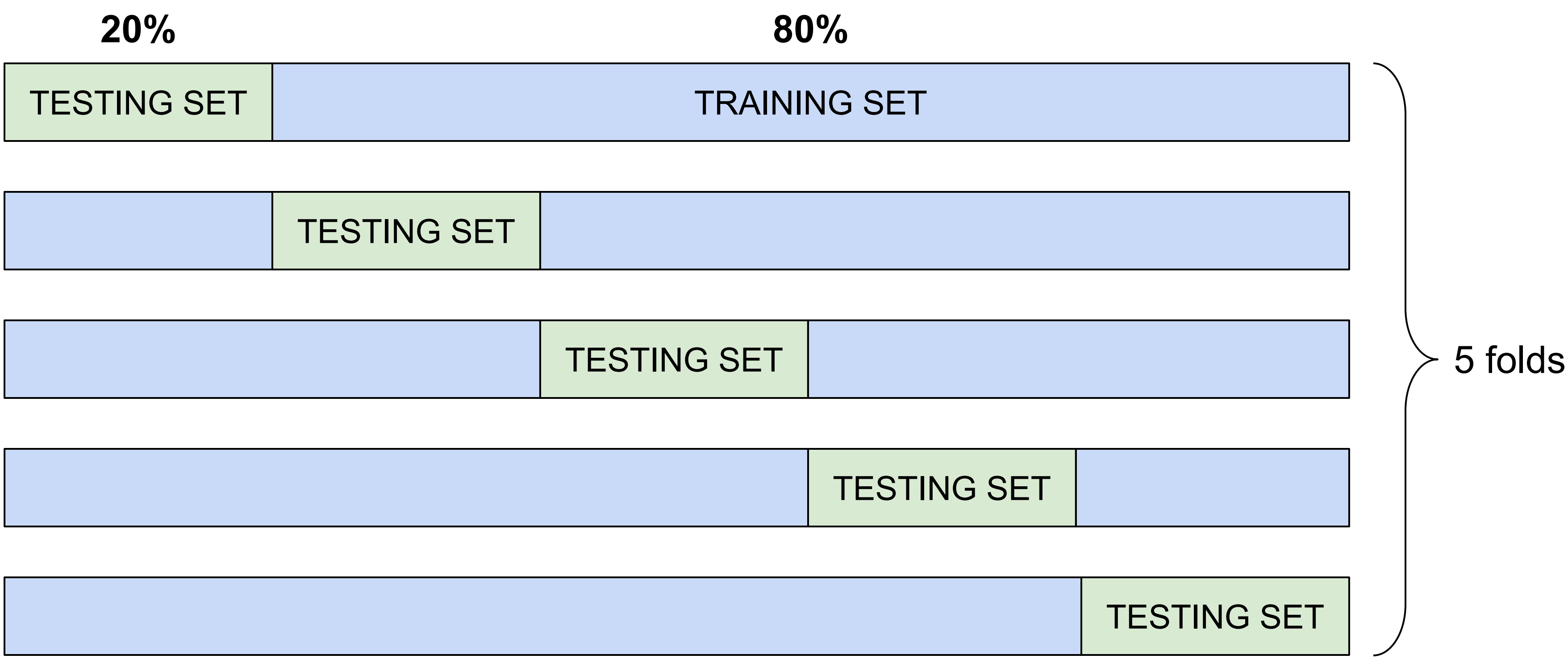
- However, **if you have many examples**, you may decrease the size of testing set to have a more accurate model with a still significant testing set



- What if we want to use **all the data** both for training **and** testing?

Cross-Validation (CV)

- The idea of cross-validation is to rotate over testing/training set split, so that each example will be used for training and for testing
- The fraction of the testing set will determine the number of rotations, also called “folds”. For example, using a testing set of 20%, that is ⅕ , we will have a 5-folds cross validation:



Cross-Validation: limit scenarios

- A **2-fold** cross-validation splits the data 50/50:
  - 1st fold: the first half is used for testing, the other for training
  - 2nd fold: the first half is used for training, the other for testing
- A **m-fold** cross-validation (m is the number of examples in the whole dataset) splits the data 1/m:
  - 1st fold: first example is used for testing, all the rest (m-1 examples) is used for training
  - 2nd fold: second example is used for testing, the first example and all the other examples from the third are used for training
  - …
  - m-th fold: first m-1 examples are used for training, last example is used for testing

  m-fold a.k.a. **Leave-One-Out Cross-Validation**

- Note on CV: **the error of the model will be the average of all the errors**: in the m-fold CV it will be the average of m error values

What about non-linear data?

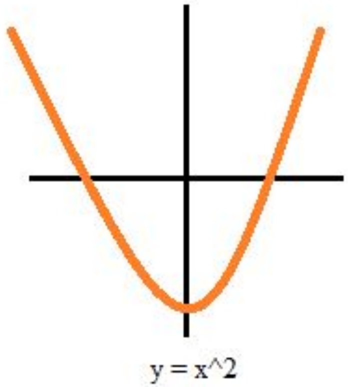
- So far we have seen a regression method that only works if the function from the input variables to the output variable is linear. E.g., with only one input variable x:

$$y = f(x) = wx + b$$

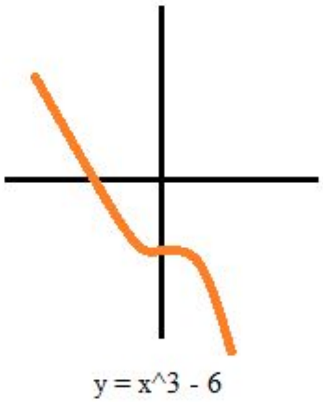
- What if the function is non-linear? The associated expression would be like:

$$y = f(x) = w_1 x + w_2 x^2 + w_3 x^3 \dots$$

with a different weight w for each exponential



Non-linear functions



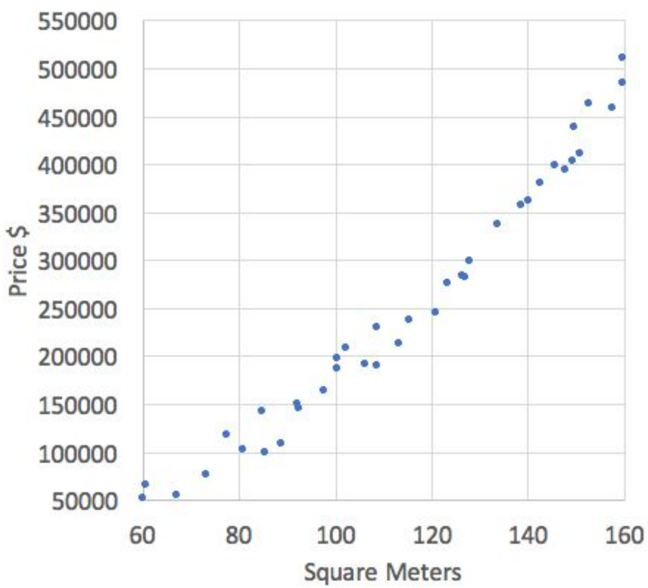
- A simple trick to obtain non-linearity is to artificially generate new features:
  - We start from a feature x_1
  - We can compute the feature x_1^2 and add this new feature to our data
  - We then compute the feature x_1^3 and add it to the data
  - And so on...

Example: house prices

- Let's suppose that the price is a non-linear function of the size in square meters
- In this example, the function that relates the price to the size is:

$$\text{Price} = 0*\text{Size} + 20*\text{Size}^2 + 0$$

- We kept the zeros to show all the coefficients and variables
- We can model the above function by augmenting our dataset as follows:



Size (sqm)	Price
60	63193.69799
65.04312529	70911.59045
68.8814367	80082.32417
69.35554098	80602.04965
69.73350476	114729.8332
74.53091304	131513.3546

Size (sqm)	**Size^2**	Price
60	**3600**	63193.69799
65.04312529	**3887.220733**	70911.59045
68.8814367	**4436.82399**	80082.32417
69.35554098	**4764.013352**	80602.04965
69.73350476	**5225.380597**	114729.8332
74.53091304	**5321.441446**	131513.3546

Limits and advanced methods

- There are limits to the previous “trick”:
  - We have to **manually** add features before training
  - We don’t know where to **stop**
    - to which degree of the polynomial? x^3? x^4? … x^{20} ?
    - How about products of features like x_1x_2 or $x_1^2x_2^3x_2^2$?
  - There can be so many features that this would make dimensionality (number of features) to explode
- There are **advanced methods** for regression that do not need to manually add features but have different ways of coping with non-linearity

Classification

- The classification problem ask to predict a categorical variable's value from the values of other variables
- Easiest example is with two variables x and y:
  - x is the variable in input
  - y is the variable we want to predict

x	**y**
1	A
2	A
3	B
4	B
5	A

Training data: x is the input data, y is the label data.

Learning
Learning task: **what's the mapping from x to y?**

Testing
Try to "learn" from the training data an hypothesis function h
What is h(6)? That is, what's y when x is 6?

x	**y**
6	?

Classification example 1/2

- We have again the data from 10 thousand houses in the U.S.

City	Rooms	SQM	Price
Los Angeles	3	130	420000
Los Angeles	2	60	380000
...	...	...	...
Albuquerque	2	140	220000
Albuquerque	3	150	250000

- Goal: learn a function (model) that can infer the City given all the other variables, for houses not in the 10 thousand dataset:

Price	Rooms	SQM
700000	5	230

City
?

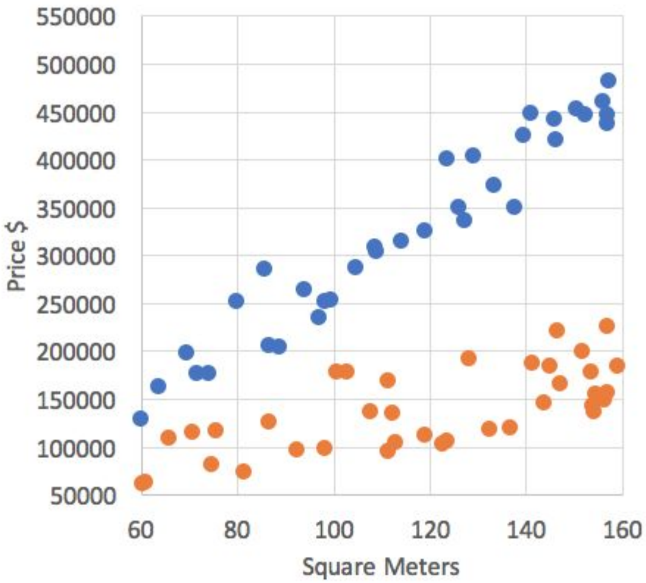
Classification example 2/2

- The houses' city can be an example of **classification** task:
  - Among the variables we choose the output variable (City)
  - The output variable is the *ground truth*, because it's the actual city of the house
  - The output variable we want to predict is a categorical value

Price	Rooms	SQM
700000	5	230

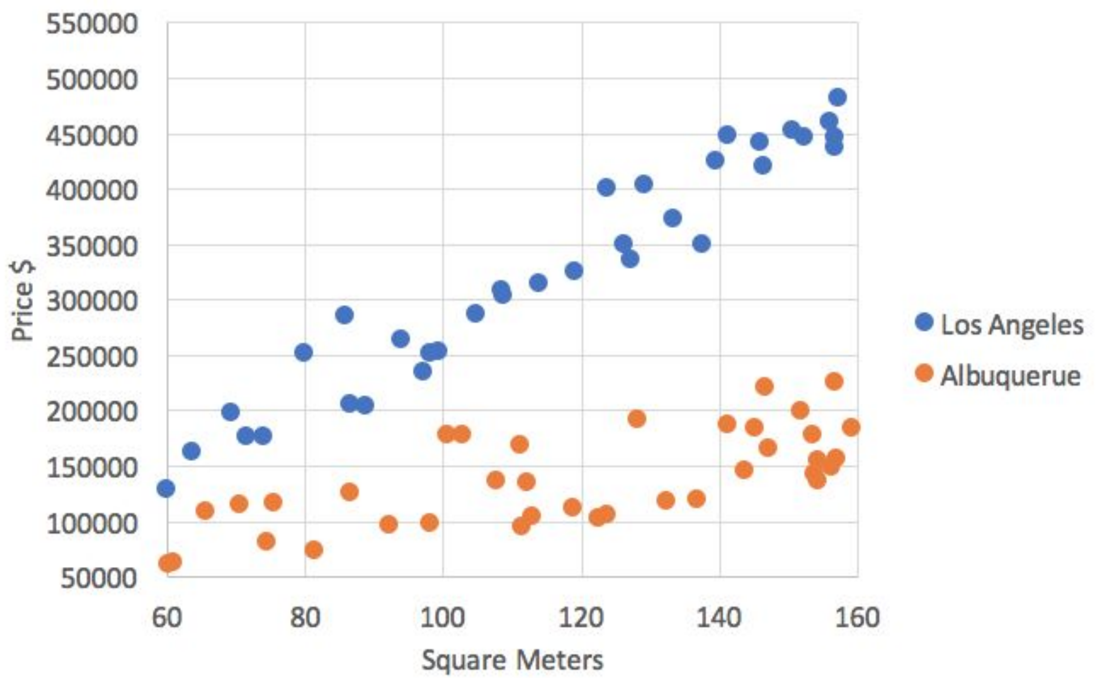
City
?

- To visualize things better we will have examples with two variables (no rooms) as input and one as output
- The categorical output will be visualized using a **different color for each category**



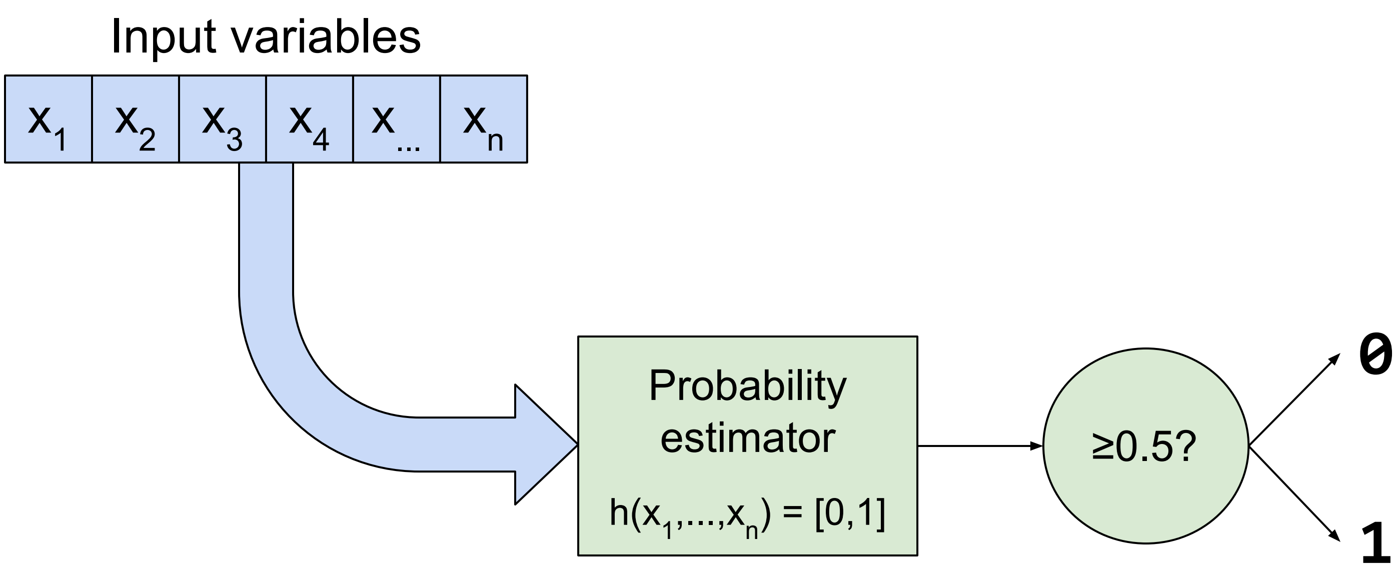
Binary classification

- The binary classification is the task of classifying between two classes. For example:
  - Detecting if an email is spam or not
  - Deciding if an image has a cat in it
  - Predicting if a currency will go up or down
  - Classifying if a house is in L.A. or Albuquerque



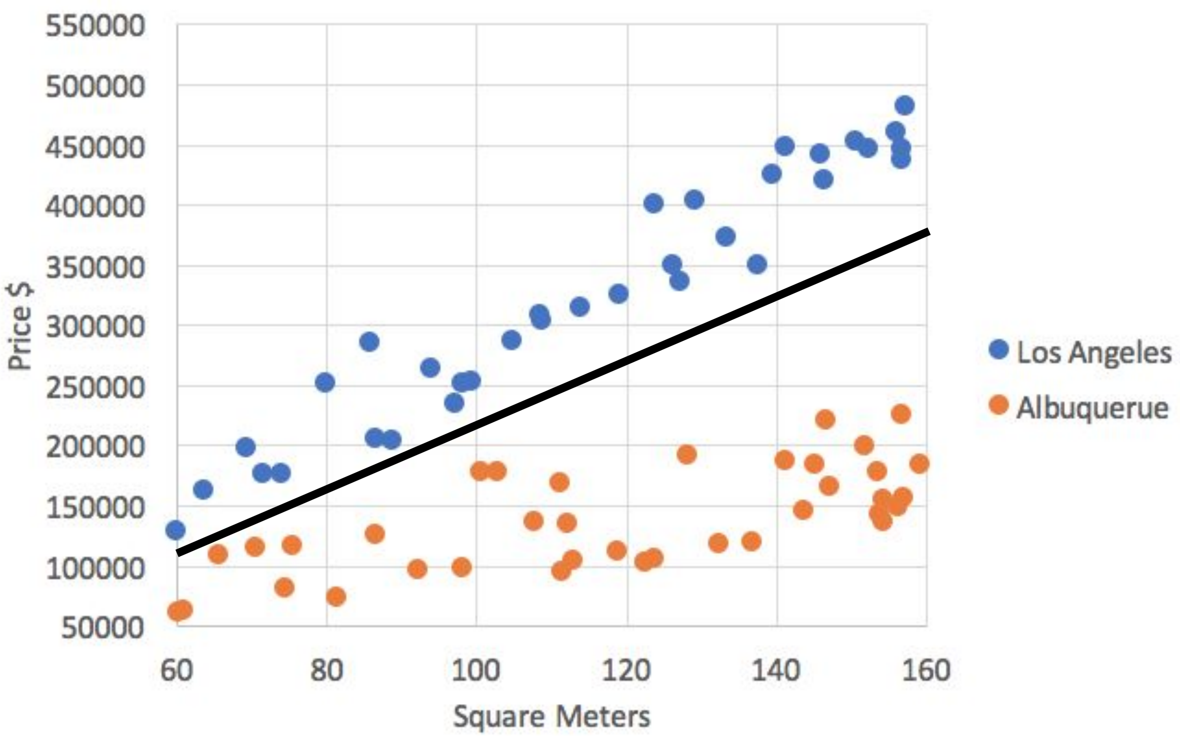
Binary classifier prediction

- A binary classifier prediction works as following:
  - Estimate a score of probability for the first class
  - Output a binary value:
    - 1 if the probability score is greater than or equal to 0.5
    - 0 if the probability score is less than 0.5



Separating the space

- Visually speaking, it is very easy to linearly separate (i.e. with a straight line) the space between Los Angeles houses and Albuquerque houses:



- But how could we **learn** a function that separate the space automatically?

Linear regression for classification?

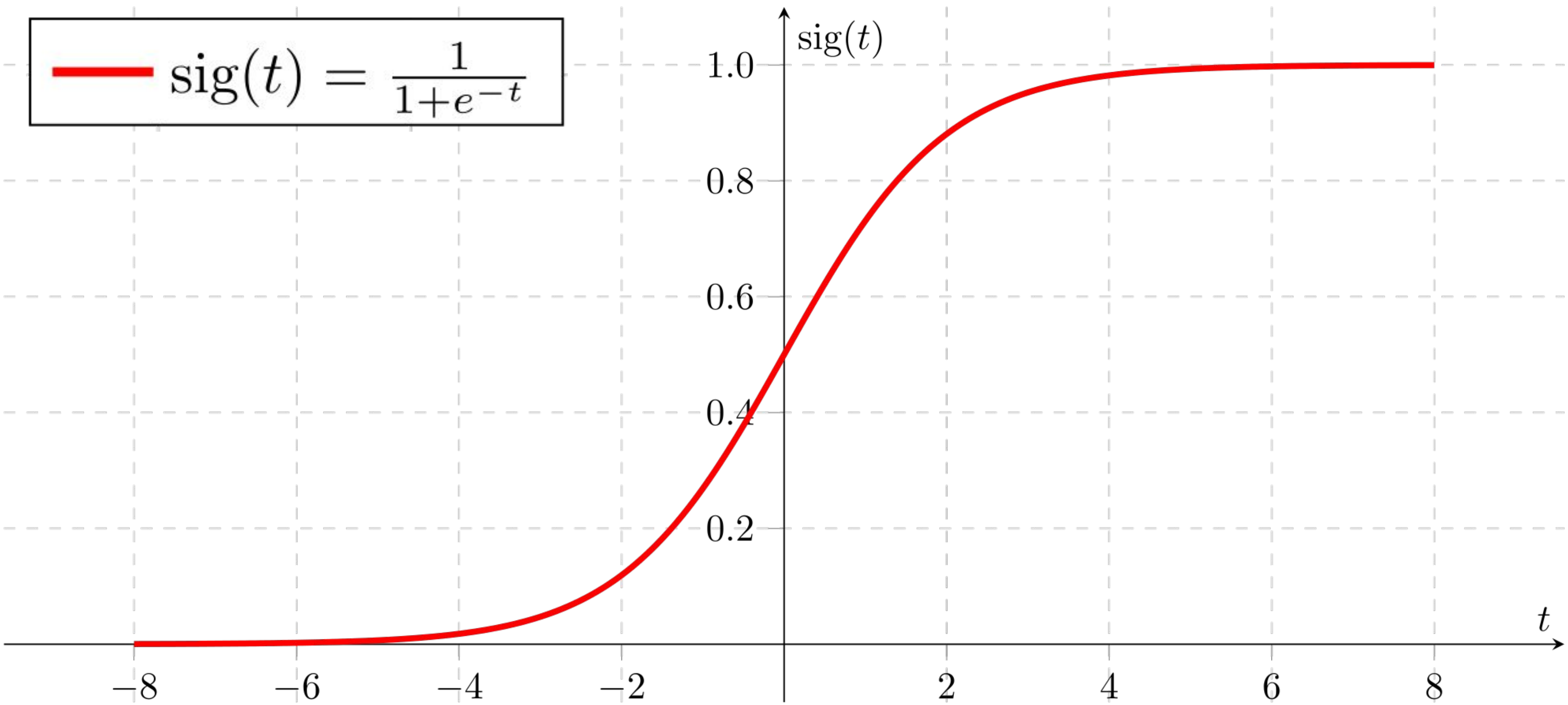
- In the case of classification, our hypothesis is a function that serves as a probability estimator
  - It takes the variables in input (e.g., house size and price)
  - It must outputs **a value between 0 and 1**

- Can we use **linear regression**? After all ...
  - … the output value is actually numerical: 0 or 1
  - … the line that separates our examples is a straight line

- **Problem:** we don't have control over unseen data, so we could have values bigger than 1, or lower than 0
- **Idea:**
  - We use a linear function to combines all the input variables into a value
  - We apply a function to constrain every possible value into a range of [0,1]

Logistic (sigmoid) function

- The ***sigmoid*** function is a special case of the **logistic function**
- Sigmoid functions have domain of **all real numbers**, with return value monotonically increasing **from 0 to 1**



Logistic Regression

- Despite the name, the logistic regression is actually a **classification** method
- Recall the linear regression hypothesis was a linear combination of the input variables, with one weight for each variable:

$$h_{linear}(x_1,...,x_n) = w_1x_1 + \ldots + w_nx_n + b$$

- Its hypothesis function is simply the sigmoid function applied to the linear combination:

$$h_{logistic}(x_1,...,x_n) = sig(w_1x_1 + ... + w_nx_n + b)$$

Logistic Regression: loss function

- Recall that we need a loss function to **tune the weights towards a minimum loss**
- We have a training input x. We consider two cases:
  - **When the true y is 0**: the hypothesis should output 0, anything more than 0 is a loss! So we can use simply the hypothesized score

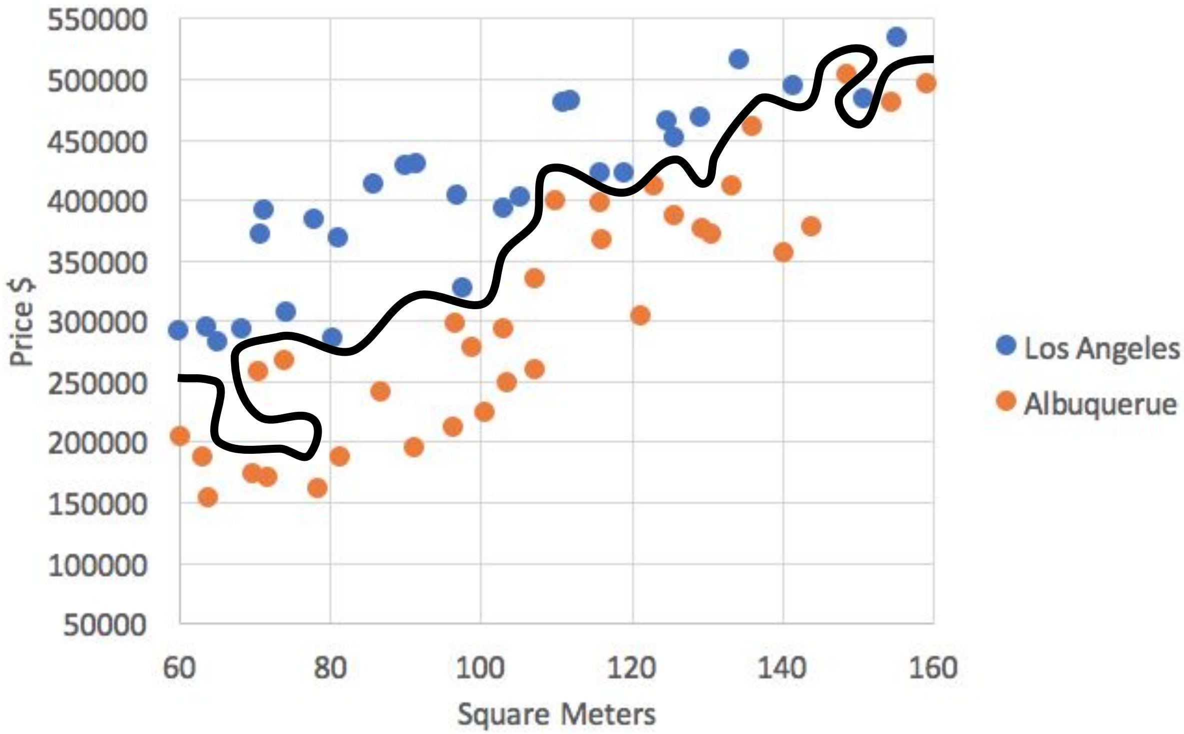
$$loss(x) = h_{logistic}(x)$$

  - **When the true y is 1**: the hypothesis should output 1, anything less than 1 is a loss! So we can use the difference between 1 and the hypothesized score

$$loss(x) = 1 - h_{logistic}(x)$$

Problem: Overfitting 1/2

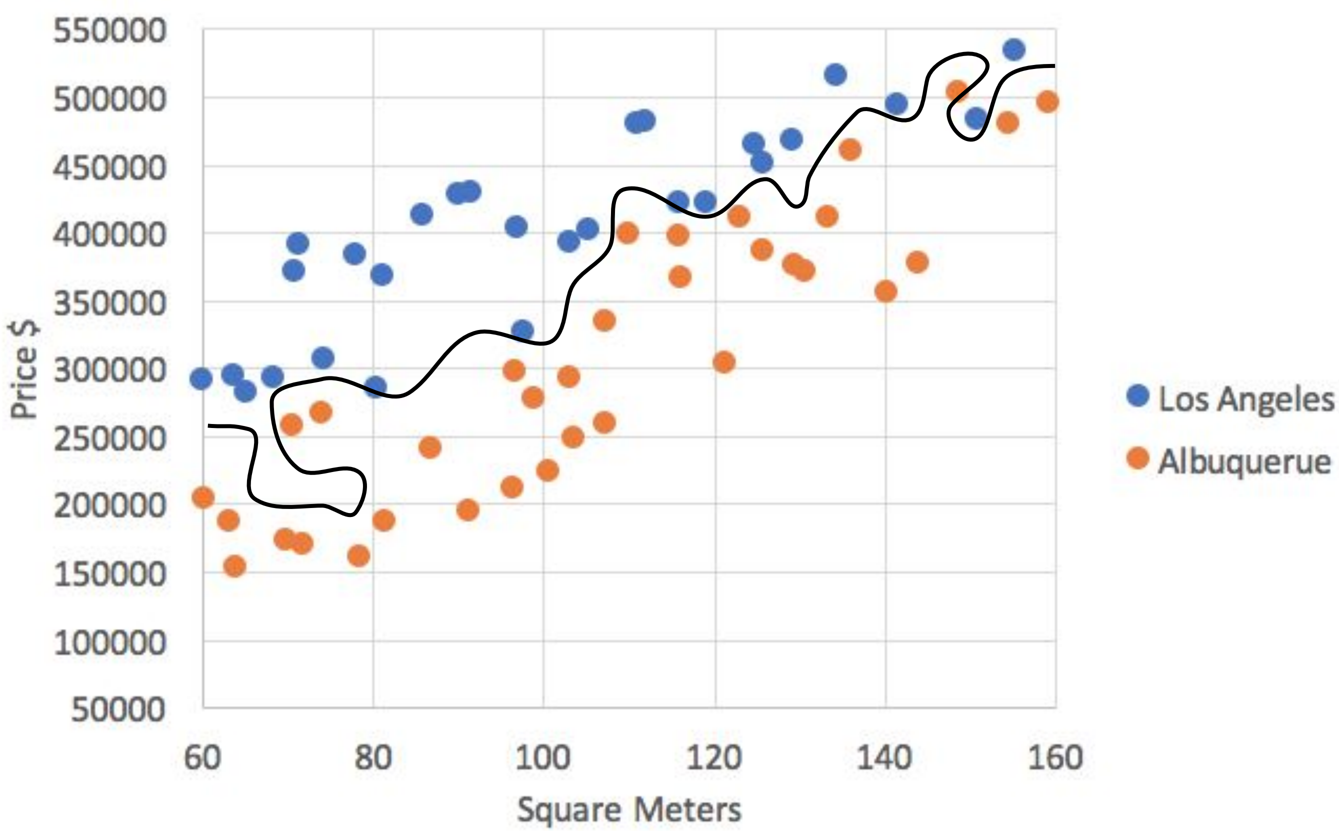
- Using the **same trick** of introducing artificial features using polynomials of different degrees (e.g. $x_1^2 x_2^3 x_2^2 \ldots$), we could train a very complex non-linear function:



- **Is this a good thing?** For sure it's perfect for the training set … what about **unseen data**?

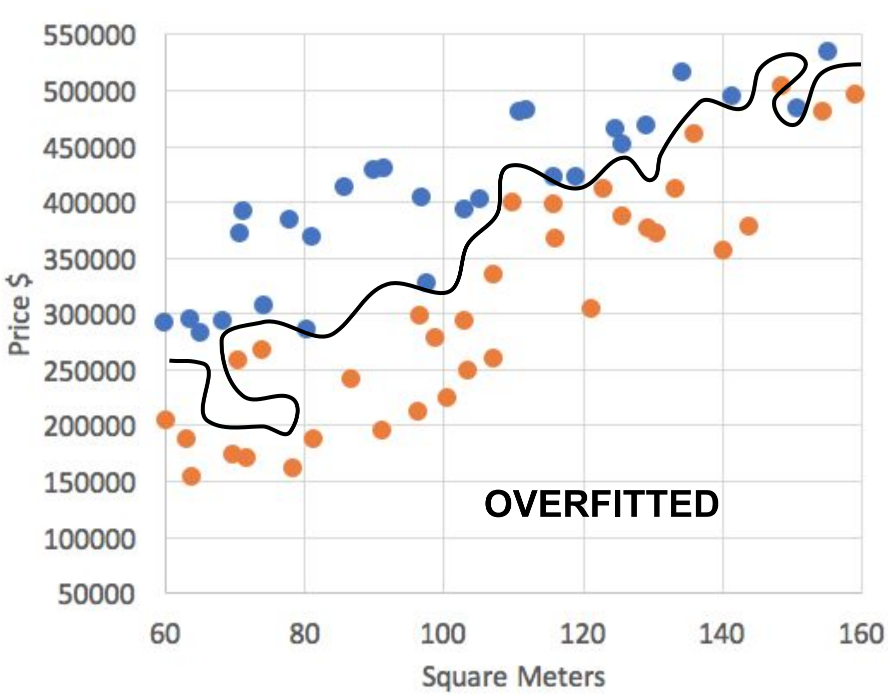
Problem: Overfitting 2/2

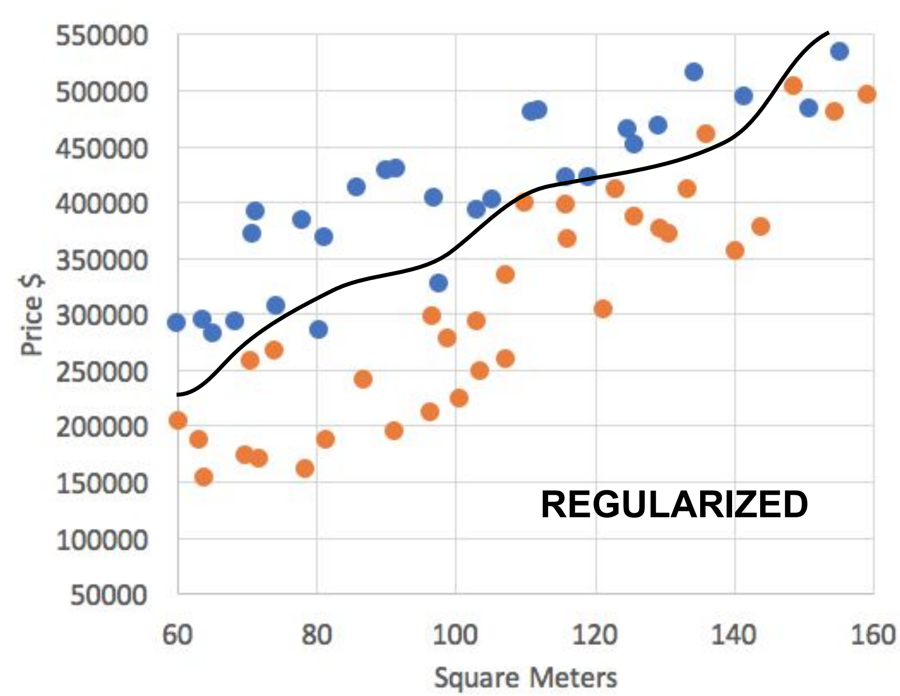
- It turns out it is actually not good: the model is **overly fitted** on the training set the unseen data could fell randomly on one side or the other
- The model should be more robust and **generalized** to work also on unseen data (such as the testing set)



Solution: Regularization

- Regularization is a very effective technique to prevent overfitting
  - It’s a modification of the loss function
  - We add an additional value to the loss function that increases as the value of features weights (w) increase
- What is the effect of regularization?
  - It prevents the features weights (w) to be very large
  - Visually speaking, it will “smooth” the decision boundary line:





Multi-class classification

- Binary classification **can be generalized** to multi-class classification
- There are **two ways** to adapt binary classifiers to a **K-classes** classification problem:
  - **OVO** - **One vs. One**: Each possible pair of classes is taken in consideration to train a different classifier. This approach will **need to train K*(K-1)/2 classifiers**!
  - **OVA** - **One vs. All** (more common): for each class, a different classifiers is trained to distinguish between that class and all the other classes merged together. This approach will **need to train only K classifiers**

References

Stanford Lectures on Machine Learning

Andrew Ng