Linear and Logistic Regression

Simple, yet powerful predictors

Yordan Darakchiev
Technical Trainer
iordan93@gmail.com

Table of Contents

- sli.do: #regr-models
- Machine learning basics
  - Objective function, cost function, optimization
- Linear regression
  - Problem description, motivation
  - Algorithm
  - Usage
- RANSAC
- Extensions: polynomial regression
- Logistic regression
  - Problem description
  - Algorithm
  - Usage

Linear Regression

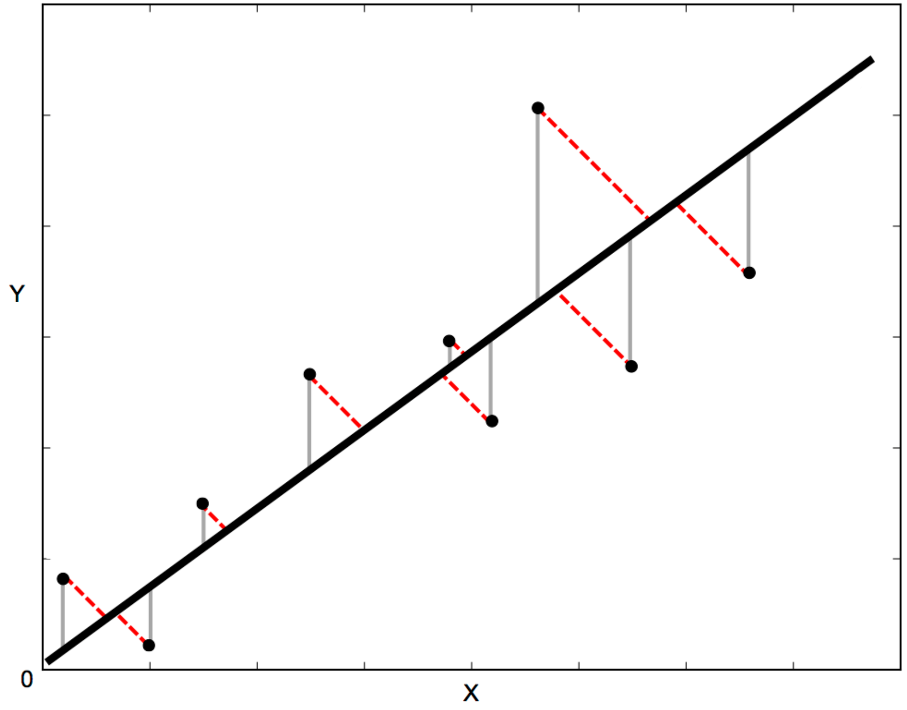
Predict continuous values…
and torture first-semester students

Linear Regression Intuition

- Regression – predicting a continuous variable
- Problem statement
  - Given pairs of $(x; y)$ points, create a model
    - Input x, output y; goal: predict y given x
      - Under the assumption that y depends linearly on x (and nothing else)
- Linear regression model
  - $y = ax + b$
    - **Modelling function**
    - 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”

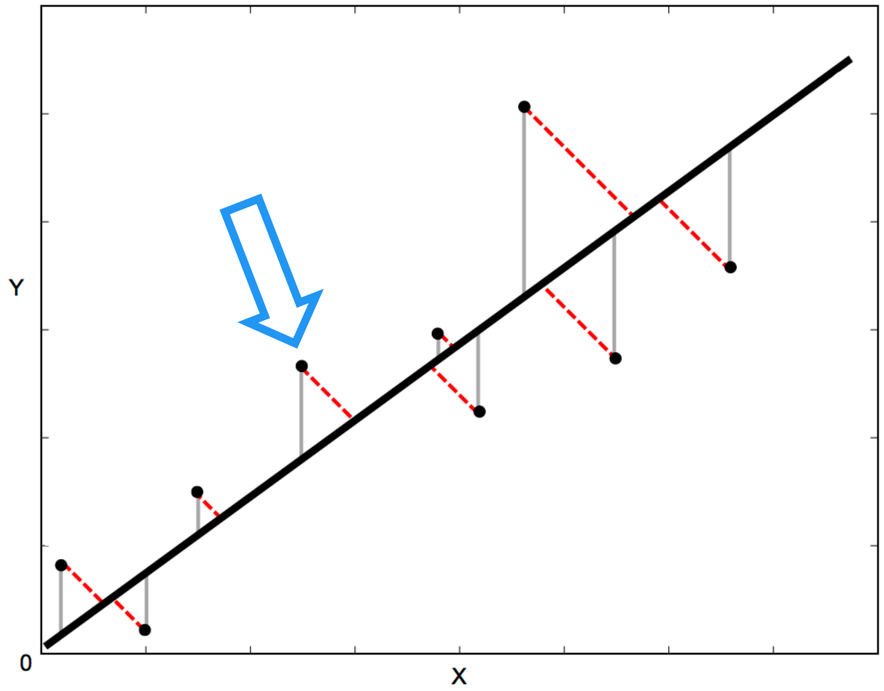
Distances

- 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
- Distance becomes a very simple difference
  - $d = y - \tilde{y};\ \tilde{y} = ax + b$
    - But… now distances can be negative
  - Better: $d = |y - \tilde{y}|$ – mean absolute error (MAE)
    - This is used sometimes but has its drawbacks
  - Another approach: $d = (y - \tilde{y})^2$
    - This is the most widely used function



Loss Function

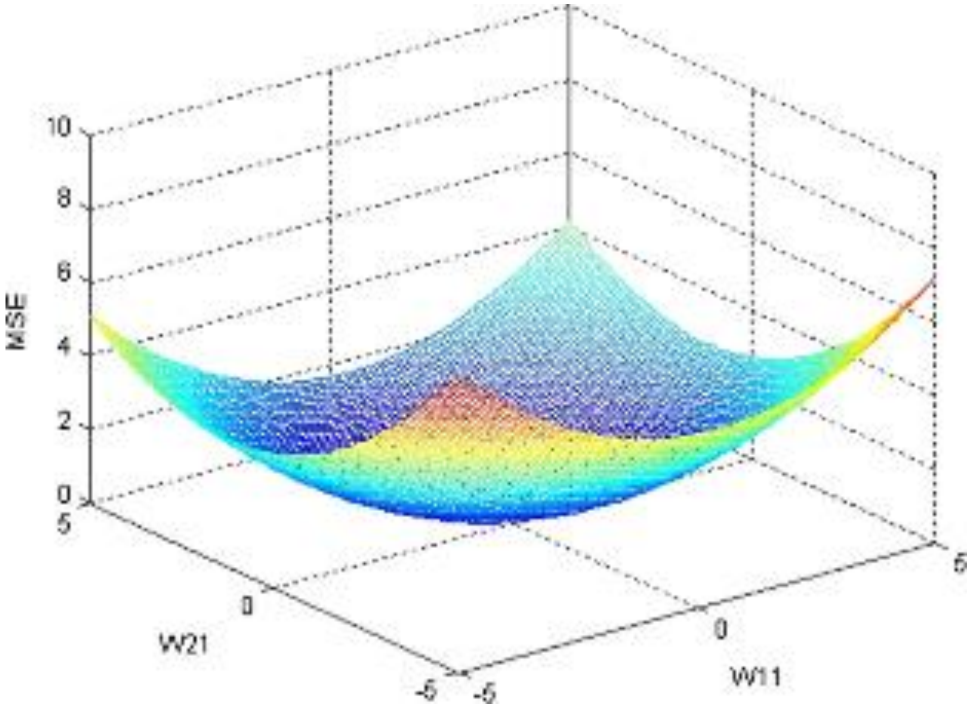
- We want to somehow account for all points
  - We can simply sum all distances to get a measure of "the total distance" from all points to the line
  - Since we can have 4, 10, 100 or 10^9 points, we also need to normalize the error
- The sum of distances now becomes

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

  - This is what we call our **total loss function**
  - This is an estimation of the total distance
  - **Minimizing this function will produce the best line**

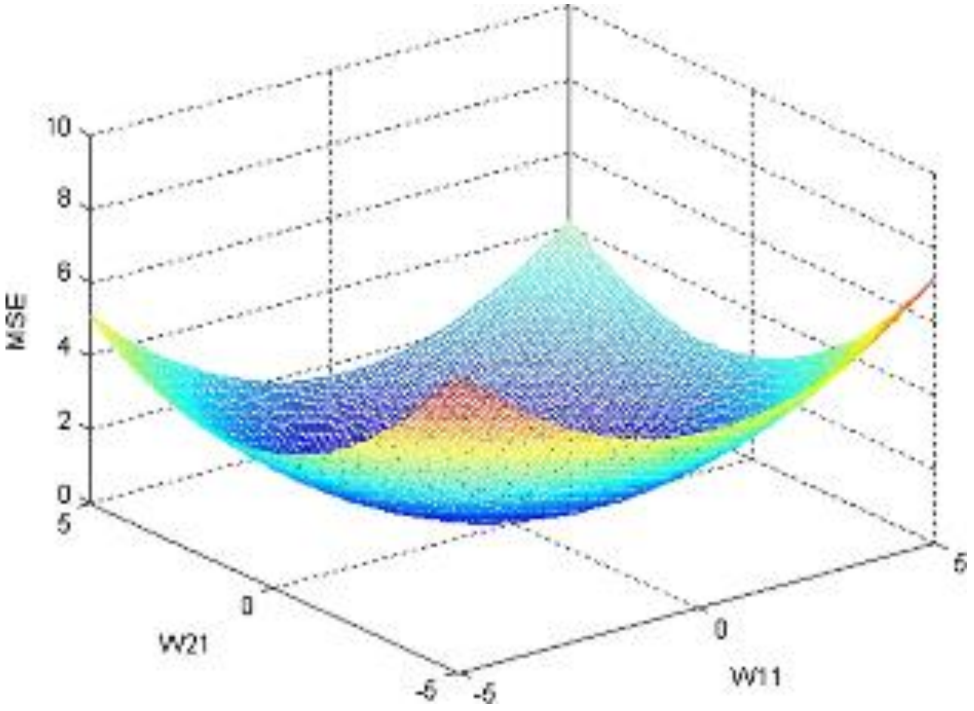
Inspecting the Loss Function

- 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 Loss Function (1)

- 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

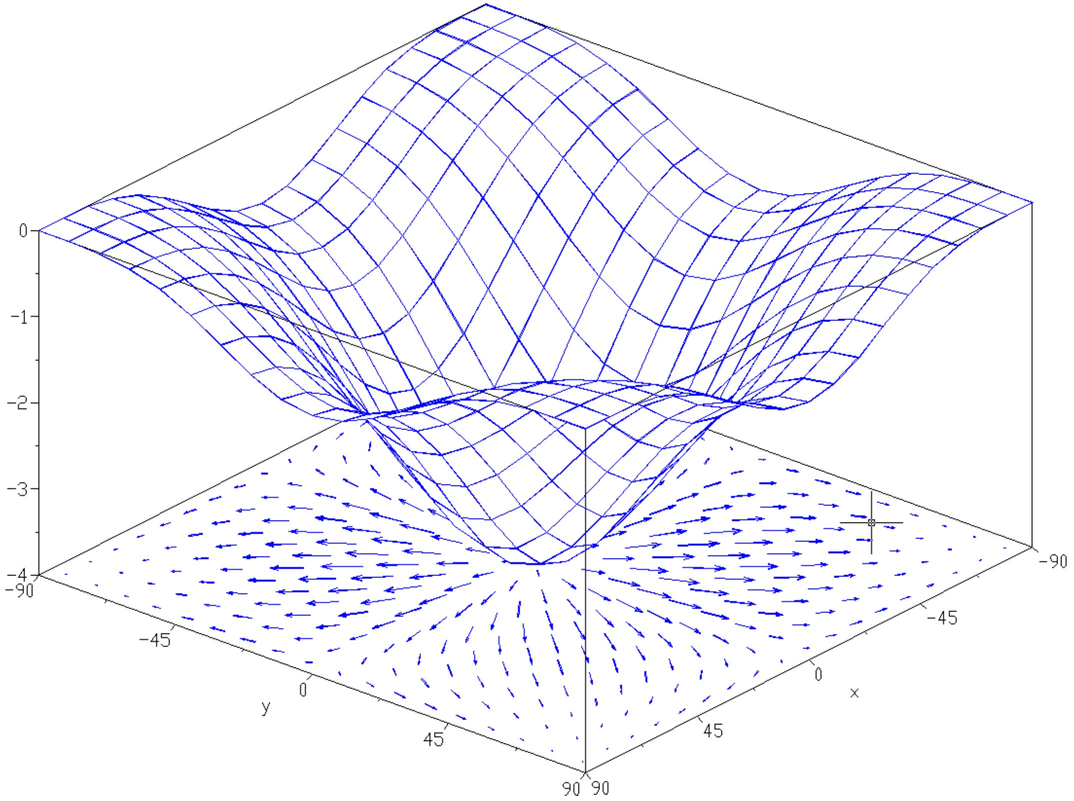


Minimizing the Loss Function (2)

- 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
  - $\frac{\partial J}{\partial a} = -\frac{2}{n}\sum_i x_i(y_i - \tilde{y}_i)$; $\frac{\partial J}{\partial b} = -\frac{2}{n}\sum_i(y_i - \tilde{y}_i)$
- 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
  - $\nabla J = \begin{pmatrix} \frac{\partial J}{\partial a} \\ \\ \frac{\partial J}{\partial b} \end{pmatrix}$
    - This is also math notation – a way to write down the components of the vector

Gradient Descent (4)

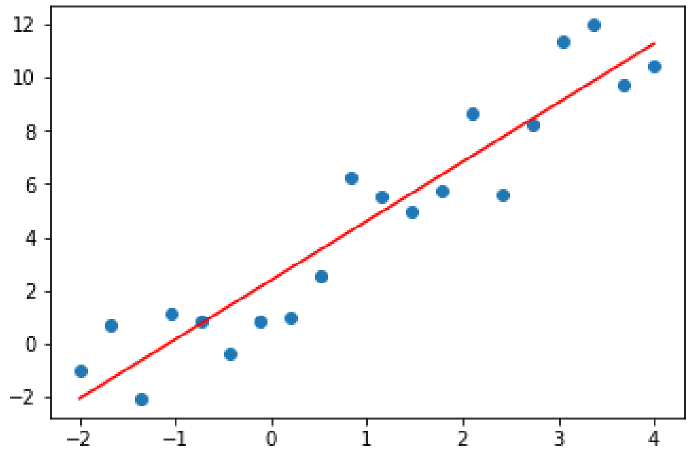
- Let's now get to the real descent
- Iterative algorithm – perform as long as needed
  - Start from some point in the $(a; b)$ space: $(a_0; b_0)$
  - Decide how big steps to take: number α
    - Called "learning rate" in ML terminology
  - Use the current a, b and x, y to compute ∇J
    - $-\nabla J_a$ 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

Results and Interpretation

- Going through the entire process, we now have a line $\tilde{y} = ax + b$ 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

```
from sklearn.linear_model import LinearRegression

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



Lab: Linear Regression on Real Data

- The algorithm can be generalized to more than 2D
  - "Multiple linear regression": $y = \beta_0 . 1 + \beta_{1 \dots m} x$
- Let's use this model to try and predict housing prices (a classical dataset located here)

```
housing.columns = ["crime_rate", "zoned_land", "industry", "bounds_river",
"nox_conc", "rooms", "age", "distance",  "highways", "tax", "pt_ratio",
"b_estimator", "pop_status", "price"]
```

- First, we want to explore the datasets
  - A more thorough exploration is "left as an exercise to the reader"
  - But we want to see what model would be appropriate
    - In addition to usual data analysis techniques, let's plot all correlations between any pair of features

Plotting Correlations

- A correlation matrix is... well, a matrix of all Pearson correlation coefficients

```
housing.corr()
```

  - We can plot it as a "heatmap" for better viewing (using seaborn)

```
import seaborn as sns
sns.heatmap(housing.corr(), annot = True, fmt = ".2f", square = True)
```

- We can also view correlations of attribute pairs
  - We'll select a subset of all attributes for a smaller chart

```
sns.pairplot(housing[["pop_status", "industry",
  "nox_conc", "rooms", "price"]])
```

  - We can see that most correlations are, indeed, linear

Creating a Model

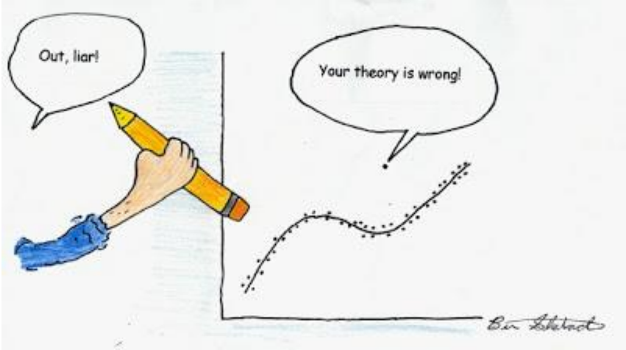
- Modelling is very simple
  - Like in the 2D example

```
housing_model = LinearRegression()
predictor_attributes = housing.drop("price", axis = 1)
housing_model.fit(predictor_attributes, housing.price)
print(housing_model.coef_)
print(housing_model.intercept_)
```

- So what?
  - We might want to predict some prices
  - Let's just pass some random rows and see the result
  - **Note: Never test on the training dataset!**

```
test_houses = housing.sample(10)
predicted = housing_model.predict(
  test_houses.drop("price", axis = 1))
print(predicted)
print(test_houses.price)
```

Regression with Outliers



- As we saw, the data has outliers
  - A few points which are far from the others
- Our goal is to exclude outliers
  - There are several methods
  - One very common – RANSAC (**RAN**dom **SA**mple **C**onsensus)
- Algorithm
  1. Fit a model to a random subsample ("inliers")
  2. Test all data points and include those which are "near" the model
     - Small enough error, tolerance provided by developer
  3. Fit the model again
  4. Estimate the error of the model (difference between first and second)
  5. Iterate steps 1-4 until performance reaches a threshold or number of iterations

Lab: RANSAC on the Housing Dataset

- Usage: similar to the linear regression model

```
from sklearn.linear_model import RANSACRegressor
ransac = RANSACRegressor()
ransac.fit(housing.drop("price", axis = 1), housing.price)
print(ransac.estimator_.coef_, ransac.estimator_.intercept_)
```

- We can also provide parameters, e.g. min number of random samples, max iterations, threshold (to include data points)
  - We can also provide the type of model we want to perform RANSAC on
    - Linear regression by default but we may use other regression models

```
ransac = RANSACRegressor(LinearRegression(), min_samples = 50,
 max_trials = 100, residual_threshold = 5.0)
```

- View inliers and outliers

```
inliers = housing[ransac.inlier_mask_]
outliers = housing[~ransac.inlier_mask_]
plt.scatter(inliers.rooms, inliers.price)
plt.scatter(outliers.rooms, outliers.price)
```

Polynomial Regression

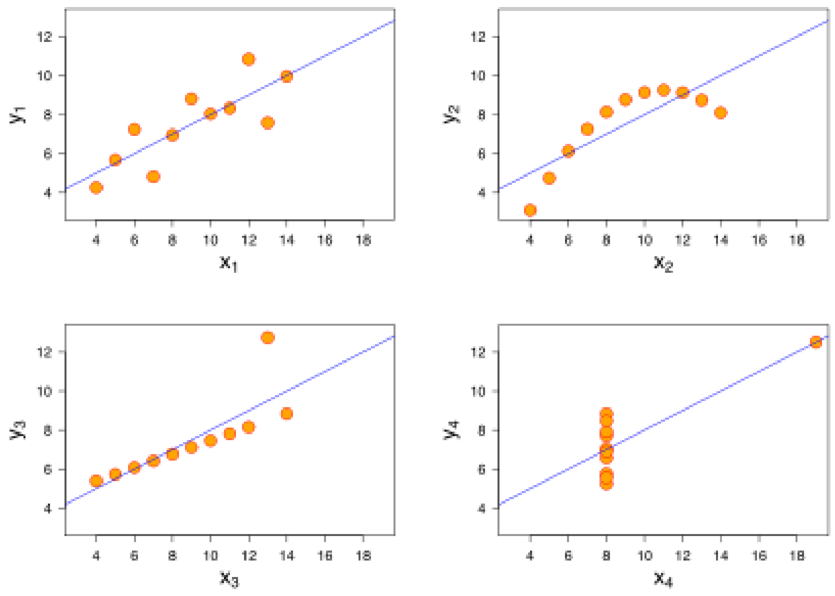
- Extension of the linear regression algorithm
  - We can use the linear regression algorithm to perform polynomial regression (e.g. fitting a quadratic curve)
    - Just precompute the columns
    - Example: if we have columns x, y and z, compute `x * z, y * z, x * z` and perform linear regression on these 6 features
    - Example 2: polynomial terms: multiply x by itself: `x * x`, `x * x * x`, etc.
- This can be achieved easily with scikit-learn

```
from sklearn.preprocessing import PolynomialFeatures

x = np.arange(6).reshape(3, 2)
poly = PolynomialFeatures(2)
x_transformed = poly.fit_transform(x)
print(poly.get_feature_names())
print(poly.n_input_features_)
print(poly.n_output_features_)
# Now we can perform linear regression with x_transformed as the input
```

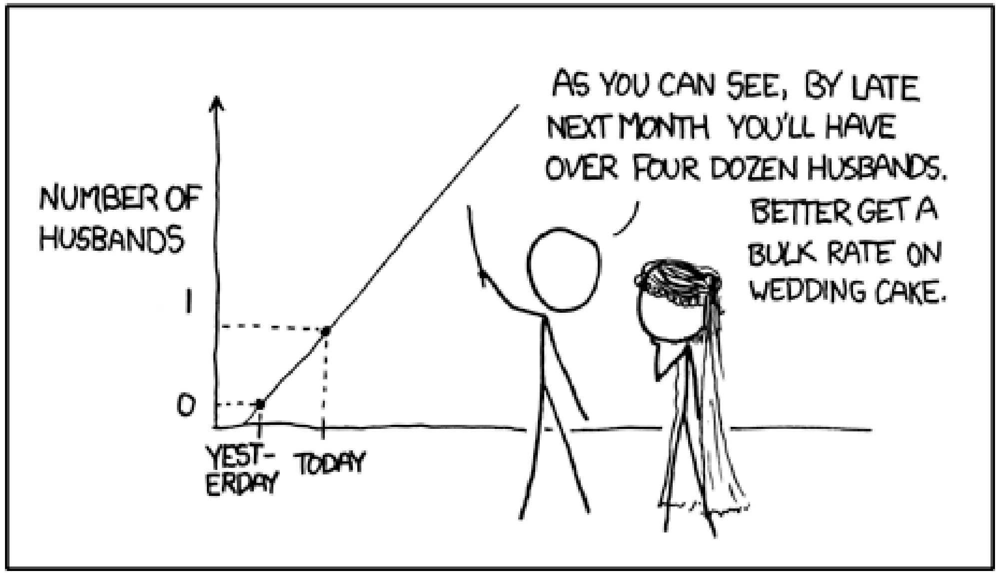
Common Mistakes

- There are two main types of errors we can make while trying regression models
  - Use a **wrong model**
    - Anscombe's quartet



  - **Extrapolate** without knowing (especially if we have interacting features)

MY HOBBY: EXTRAPOLATING



Logistic Regression

Use a regression model to classify

Classification

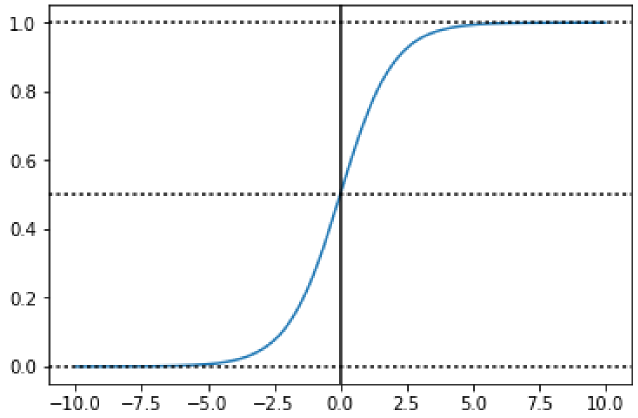
- Predict **one of several known classes**
  - Based on the input parameters
  - Example: classify whether a picture is of a cat or a dog
- Regression and classification make up most of the machine learning problems
- Choosing an algorithm
  - "No free lunch": **no single algorithm** works best
  - It's best to compare some algorithms to select the best for a particular model
    - Also, we might want to tune them first
- Reminder: ML process
  - Select features, choose a performance metric (cost function), choose a classifier, evaluate and fine-tune the performance

Logistic Regression

- Classification algorithm (despite its name)
- Two classes: negative (0) and positive (1)
  - Can be extended to more classes
- How does it work?
  - Linear regression can give us all kinds of values
  - We want to constrain them between 0 and 1
  - Approach
    - Perform linear regression: $\tilde{y} = \beta x$
    - Use the sigmoid function to constrain the output:

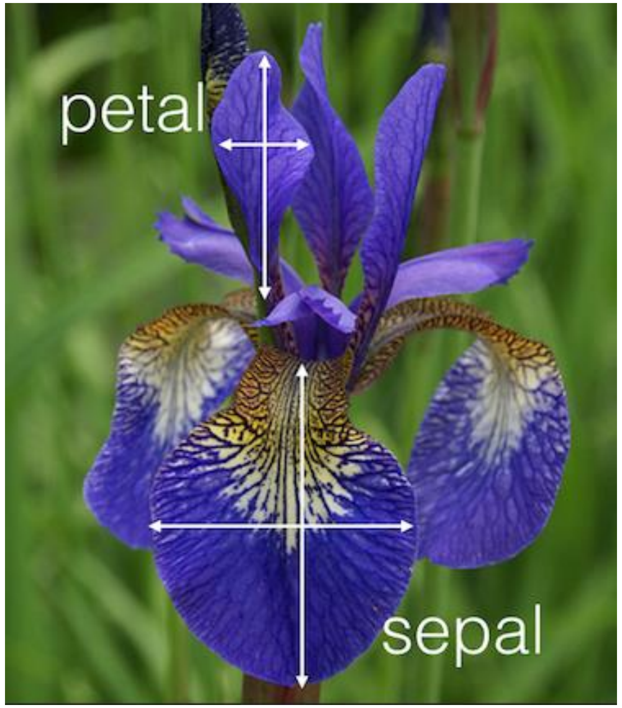
$$\sigma(\tilde{y}) = \frac{1}{1 + e^{-\tilde{y}}} = \frac{1}{1 + e^{-\beta x}}$$

    - Quantization: if $\sigma > 0.5$ return 1, and 0 otherwise
      - Remember that we only need to return 0 or 1
      - We can also use the raw values as probability measures



Lab: Logistic Regression on Real Data

- A classic dataset for classification is the Iris dataset
  - Located here
  - Three classes (setosa, virginica, versicolor)
  - 4 attributes: petal width / height; sepal width / height (all in cm)
    - Some features are highly correlated to the class
  - Explore and inspect the data before modelling



Lab: Logistic Regression on Real Data (2)

- Perform logistic regression

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(C = 1e6)
model.fit(iris_train_data, iris_train_labels)
```

- Test (output classes or probabilities)

```
print(model.predict(iris_test))
print(model.predict_proba(iris_test))
```

- In the model, there's a "mysterious" parameter C
  - Regularization: how powerful the data is (more – next time)
  - A large number means no regularization
    - We just take the data "as-is", with no other constraints

Many Classes

- Two main approaches
  - One-vs-all: several predictors
    - One predictor for each class vs. the others
  - Overall: calculate probabilities of each class
- `scikit-learn` takes care of multiple classes (multinomial logistic regression) by default
  - We don't even need to transform the labels
  - This applies to all algorithms in the library

Summary

- Machine learning basics
  - Objective function, cost function, optimization
- Linear regression
  - Problem description, motivation
  - Algorithm
  - Usage
- RANSAC
- Extensions: polynomial regression
- Logistic regression
  - Problem description
  - Algorithm
  - Usage

Questions?