

# Linear and Logistic Regression

Simple, yet powerful predictors

**Yordan Darakchiev**

Technical Trainer

[iordan93@gmail.com](mailto:iordan93@gmail.com)





sli.do

#MachineLearning

# Table of Contents

- 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

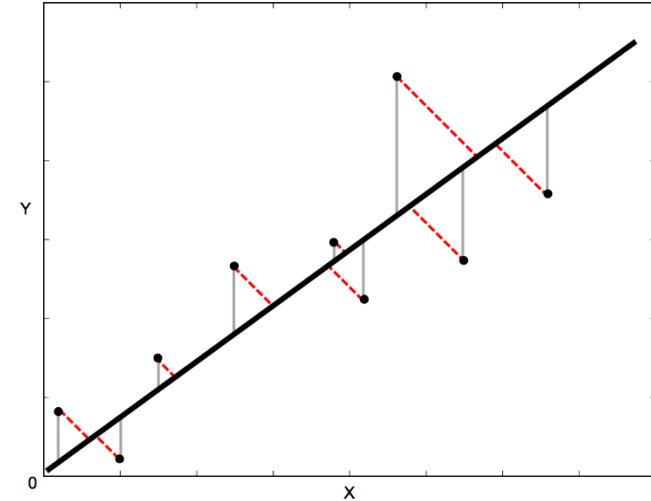
**Predict continuous values...  
and torture first-semester students**

# 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)
- Modelling function
  - $\tilde{y} = ax + b$
  - Many samples: for each sample  $(x_1, y_1), \dots, (x_m, y_m)$ :
    - $\tilde{y}_i = ax_i + b, i \in [1; m]$
  - Many variables:  $\tilde{y} = a_1x_1 + a_2x_2 + \dots + a_nx_n + b \equiv a^T X + b$ 
    - Trick:  $a_0 \equiv b; x_0 \equiv 1 \Rightarrow \tilde{y} = a_0 \cdot 1 + a_1x_1 + \dots + a_nx_n = a^T x$

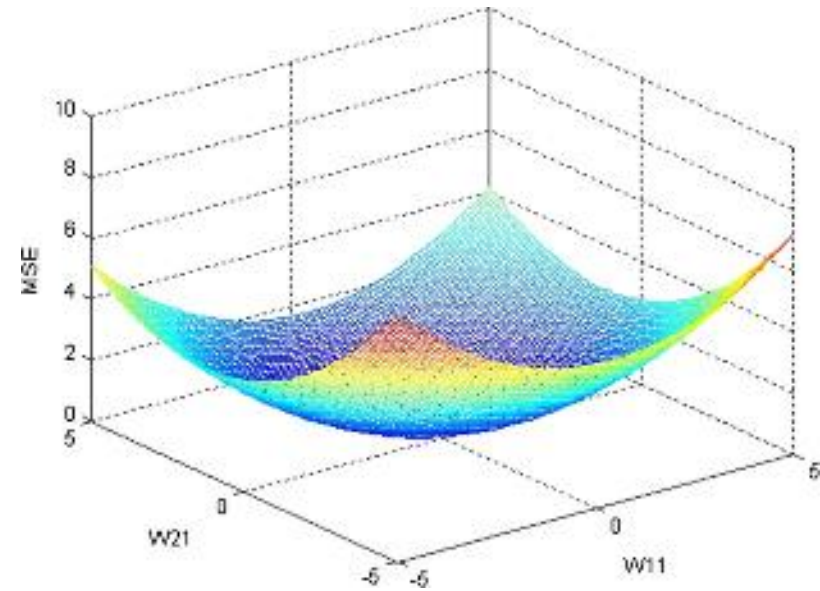
# Training

- Loss function
  - For each sample  $i$ ,  $i \in [1, m]$
  - $d_i = (\tilde{y}_i - y_i)^2$
- Total cost function
  - Also called simply "cost function"
  - $J = \frac{1}{n} \sum_{i=1}^m (\tilde{y}_i - y_i)^2$
  - $J$  depends on  $a, b, x, y$
- Training process
  - Minimize the cost function
    - We're looking for parameters  $a, b$  that lead to  $\min J$
    - Written as  $\arg \min_{a,b} J$



# Gradient Descent

- Input:  $a, b$ ; output  $J$
- Paraboloid (3D parabola)
  - It has exactly one min value
    - And we can see it
- 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 can “simulate” this: **gradient descent**
- Reminder: gradient
  - “Multi-dimensional derivative”



$$\nabla J = \begin{pmatrix} \frac{\partial J}{\partial a} \\ \frac{\partial J}{\partial b} \end{pmatrix}$$

# Gradient Descent (2)

- 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  $\alpha$ 
    - Called **learning rate** in ML terminology
  - Use the current  $a, b$  and  $x, y$  to compute  $\nabla J$ 
    - $-\nabla J_a$  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  $\alpha$  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



# Example: Housing Prices

- Multiple linear regression
  - Many predictor variables
- 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

# 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)
```

# Delving Deeper into Matrices

■ Dataset:  $y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$ ;  $X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ 1 & x_{21} & x_{22} & \cdots & x_{2n} \\ 1 & x_{31} & x_{32} & \cdots & x_{3n} \\ 1 & x_{41} & x_{42} & \cdots & x_{4n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$

third observation

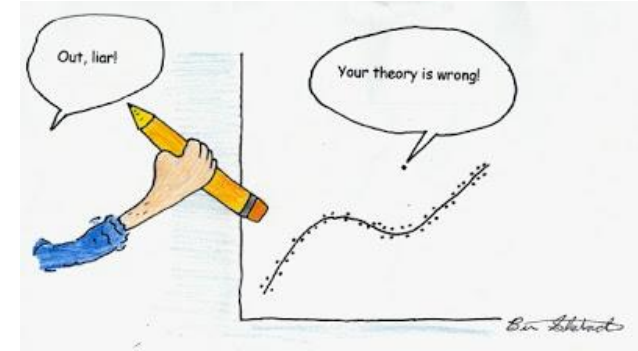
second variable

■ Parameters:  $a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix}$ ,  $a^T = [a_0 \quad a_1 \quad \cdots \quad a_m]$

■ Modelling function:  $\tilde{y} = a^T X$

# 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**mples **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

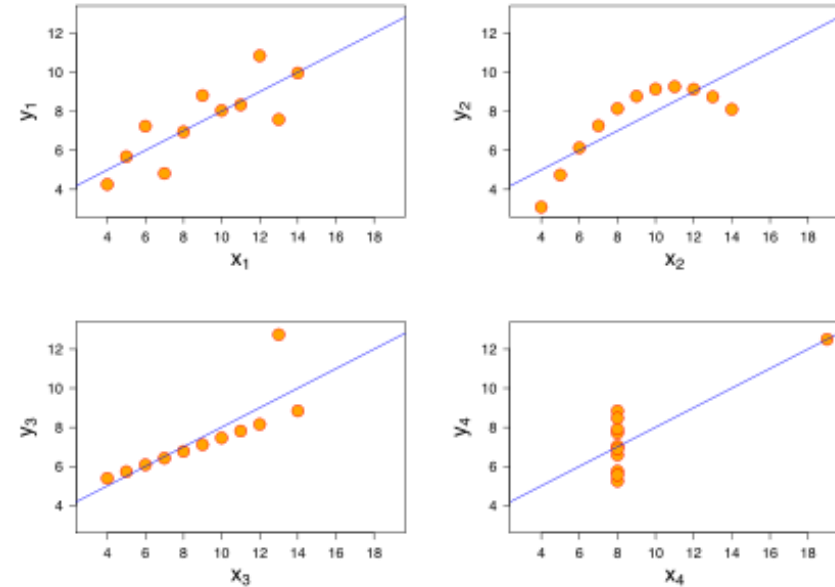
- Extension of the linear regression algorithm
  - We can use the linear regression algorithm to perform polynomial regression (e.g. fitting a quadratic curve)
    - Just precompute the columns
    - Example: if we have columns  $x$ ,  $y$  and  $z$ , compute  $x * z$ ,  $y * z$ ,  $x * x$  and perform 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

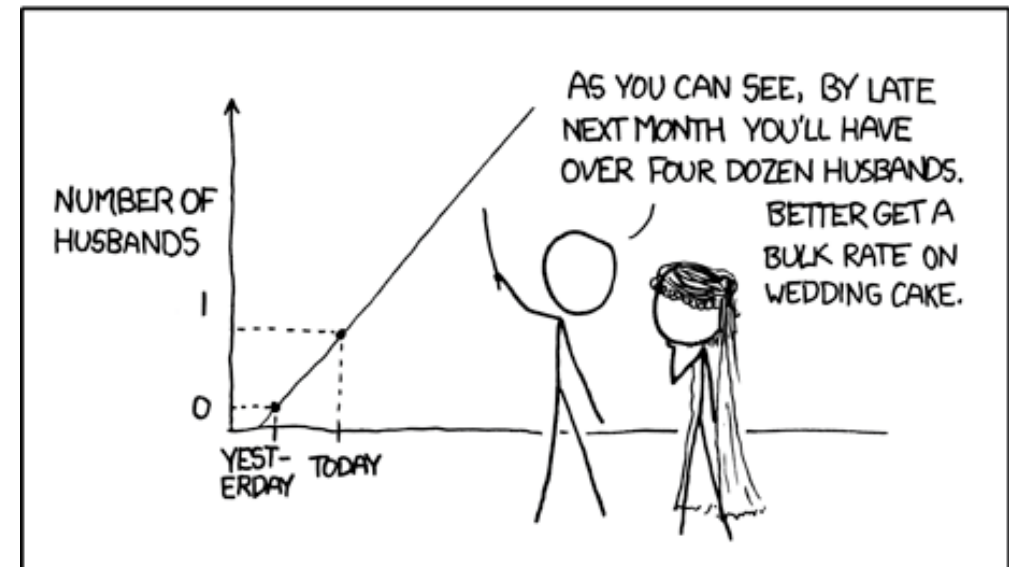
x = np.arange(6).reshape(3, 2)
poly = PolynomialFeatures(2)
x_transformed = poly.fit_transform(x)
print(poly.get_feature_names())
print(poly.n_input_features_)
print(poly.n_output_features_)
# Now we can perform linear regression with x_transformed as the input
```

# 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

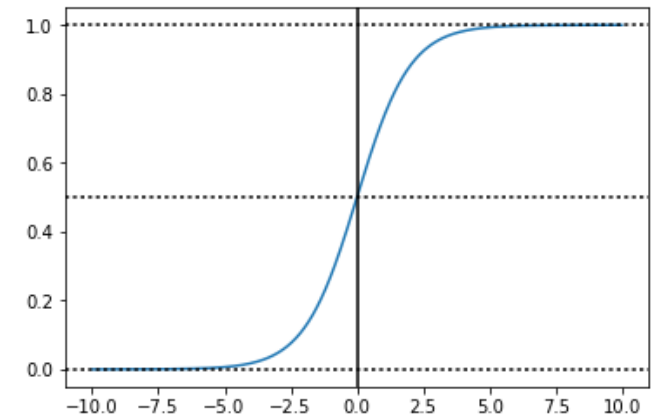


# Classification

- 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

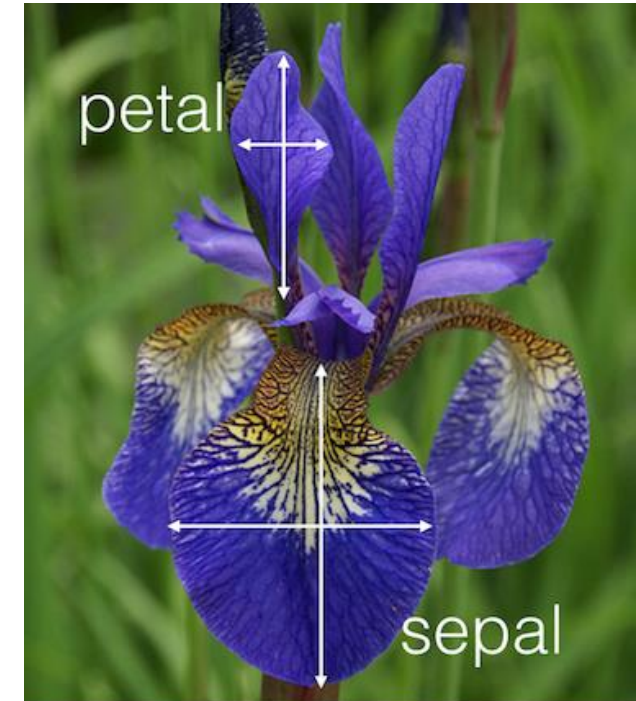
# Logistic Regression

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



# Example: Classifying Iris Flowers

- A classic dataset for classification is the Iris dataset
  - Located [here](#)
  - **3 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



# Example: Classifying Iris Flowers (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

The image features a white background with two blue decorative bars. The top bar is a solid blue strip. The bottom bar is a gradient blue strip that transitions from a lighter blue on the left to a darker blue on the right. The word "Questions?" is centered in a blue, sans-serif font.

Questions?