

# Material Summary: Linear and Logistic Regression

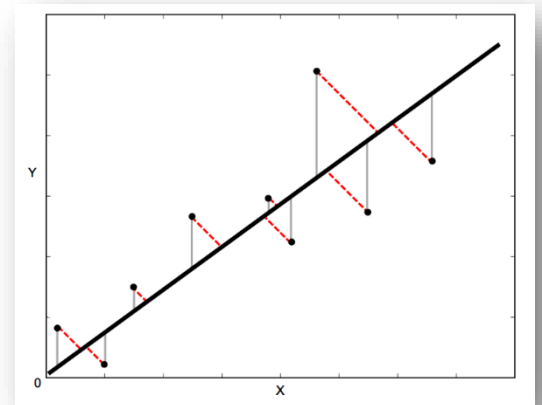
## 1. Linear Regression

### 1.1 Linear Regression Intuition

- Regression – predicting a continuous variable
- Problem statement
  - Given pairs of  $(x; y)$  points, create a model
    - Input  $x$ , output  $y$ ;
    - Under the assumption that  $y$  depends linearly on  $x$  (and nothing else)
- Modelling function
  - $\hat{y} = ax + b$
  - Many samples: for each sample  $(x_1, y_1), \dots, (x_n, y_n)$ :
    - $\tilde{y}_i = ax_i + b, i \in [1; n]$
  - Many variables:  $\hat{y} = a_1x_1 + a_2x_2 + \dots + a_nx_n + b \equiv a^T X + b$ 
    - Trick:  $a_0 \equiv b; x_0 \equiv 1 \Rightarrow \tilde{y} = a_0 \cdot 1 + a_1x_1 + \dots + a_nx_n = a^T x$

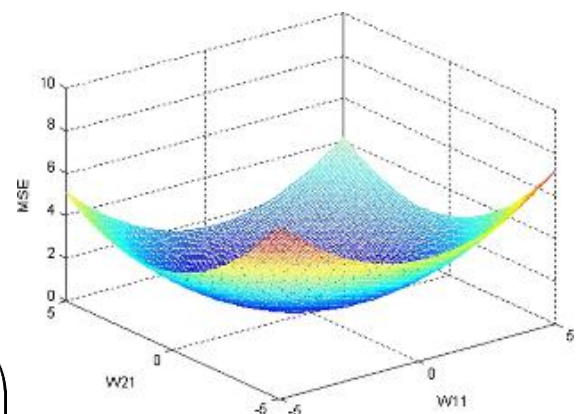
### 1.2 Training

- 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"
  - $J = \frac{1}{n} \sum_{i=1}^n (\tilde{y}_i - y_i)^2$
  - $J$  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  $\arg \min_{a,b} J$



### 1.3 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"
- Iterative algorithm – perform as many times 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
  - $a_1 = a_0 - \alpha \nabla J_a; b_1 = b_0 - \alpha \nabla J_b$
  - $(a_1; b_1)$  are the new coordinates
- Repeat the two preceding steps as needed
  - Usually, we do this for a fixed number of iterations

#### 1.4 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

#### 1.5 Creating a Model

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

```
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_)
```

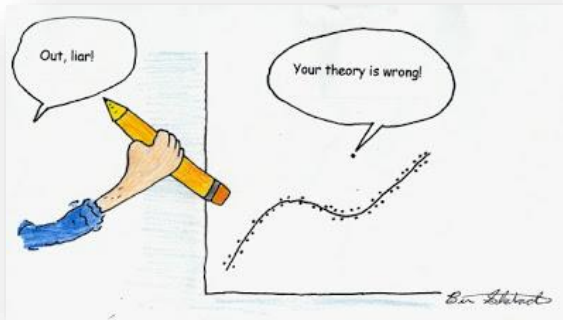
$$\text{Dataset: } y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \end{bmatrix}; X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ 1 & x_{31} & x_{32} & \cdots & x_{3m} \\ 1 & x_{41} & x_{42} & \cdots & x_{4m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

- Parameters:  $a = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{bmatrix}$
- Modelling function:  $\tilde{y} = Xa$

#### 1.7 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 (**R**ANdom **S**Ample **C**onsensus)
- Algorithm
  - Fit a model to a random subsample ("inliers")
  - Test all data points and include those which are "near" the model
    - Small enough error, tolerance provided by developer
  - Fit the model again
  - Estimate the error of the model (difference between first and second)
  - Iterate steps 1-4 until performance reaches a threshold or number of iterations



## 1.8 Polynomial Regression

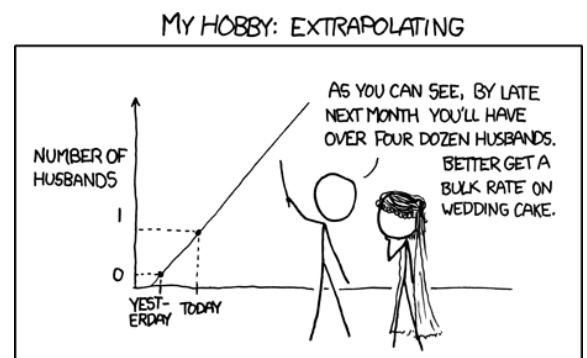
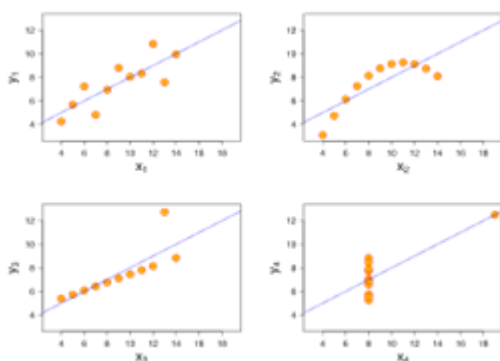
- Extension of the linear regression algorithm
  - We can use the linear regression algorithm to perform polynomial regression (e.g., fitting a quadratic curve)
    - **Just precompute the columns**
    - Example: if we have columns  $x$ ,  $y$  and  $z$ , compute  $x * z$ ,  $y * z$ ,  $x * z$  and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

## 1.9 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)



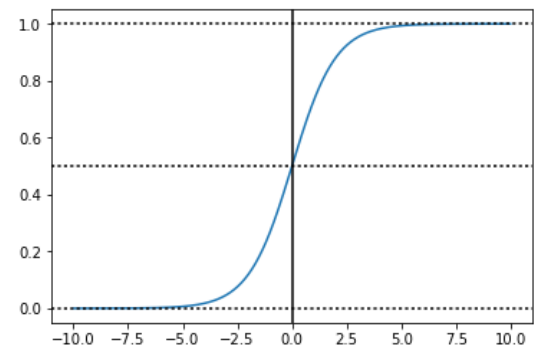
## 2. Logistic Regression

### 2.1 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

### 2.2 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:  $\tilde{y} = Xa$
    - Use the sigmoid function to constrain the output:

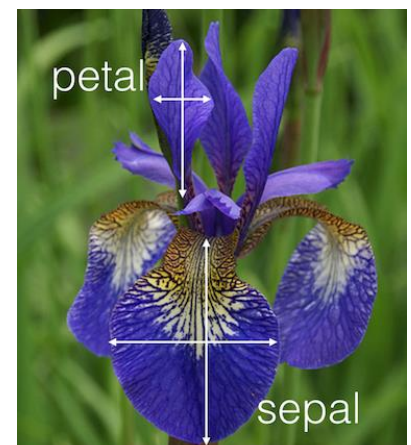


$$\sigma(\tilde{y}) = \frac{1}{1 + e^{-\tilde{y}}} = \frac{1}{1 + e^{-Xa}}$$

- Quantization: if  $\sigma \geq 0.5$  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

### 2.3 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



- Perform logistic regression

```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(C = 1e9)
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

## 2.4 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