

# Data Science and Advanced Programming — Lecture 7

## Supervised Machine Learning (Regression)

Simon Scheidegger

Department of Economics, University of Lausanne, Switzerland

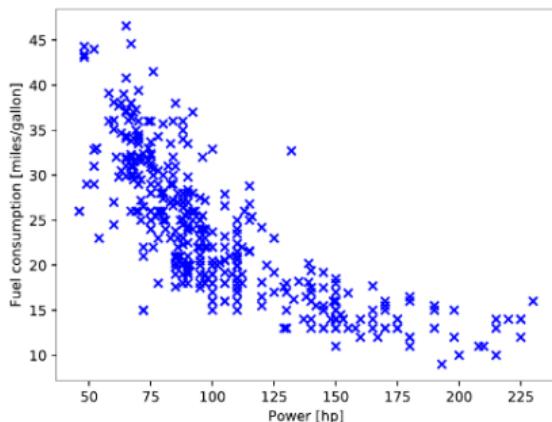
October 27th, 2025 | 12:30 - 16:00 | Internef 263

# Today's Roadmap

1. Supervised Machine Learning — the general idea
2. Linear Regression (1 Variable)
3. Again — Gradient descent
4. Linear Regression (multiple variables)
5. A Probabilistic Interpretation of Linear Regression
6. Polynomial Regression
7. Tuning Model Complexity  
The next two notebooks are supposed to be self-study (no pre-recordings)!
8. [An Introduction to Pandas \(with a Jupyter Notebook\)](#) Pandas\_intro.ipynb
9. [Stock Market Prediction \(with “live data”\) with Linear regression](#)  
see Stock\_prediction\_ML\_Lecture3.ipynb

# 1. Supervised Machine Learning

Supervised methods assume that training data is available from which they can learn to **predict** a target feature based on other features (e.g., fuel consumption of a car as a function of car power [horse power]).

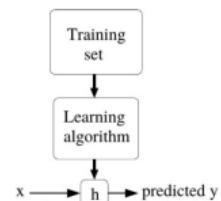


→ Given data like this, how can we learn to predict the fuel consumption of other cars?  
Dataset from [this repository](#).

# Supervised Machine Learning

- ▶  $x(i)$  : “**input**” variables (Horse Power in this example), also called **input features**.
- ▶  $y(i)$  : “**output**” or **target variable** that we are trying to predict (Fuel consumption).
- ▶ A pair  $(x(i), y(i))$  is called a **training example**.
- ▶ The dataset that we'll be using to learn: a list of  $m$  training examples

$$\{(x(i), y(i)); i = 1, \dots, m\}$$



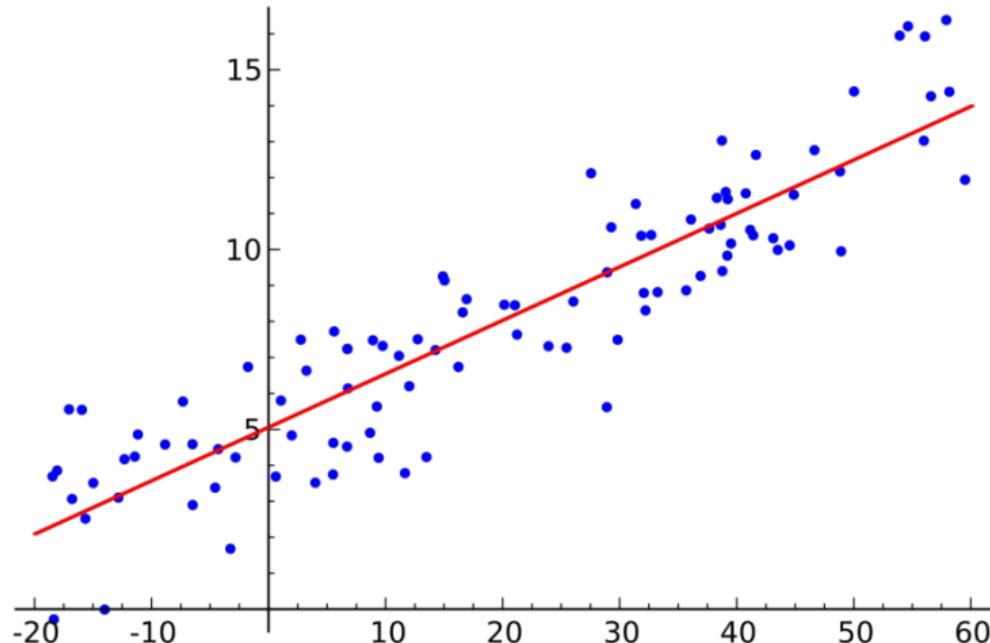
is called a **training set**.

- ▶ Our goal is, given a training set, to learn a function  $h : \mathcal{X} \mapsto \mathcal{Y}$  so that  $h(x)$  is a “**good**” predictor for the corresponding value of  $y$ .
- ▶ For historical reasons, this function  $h$  is called a **hypothesis**.

# Classification versus Regression

- ▶ When the target variable that we're trying to predict is **continuous**, such as in our car example, we call the learning problem a **regression problem**.
- ▶ When  $y$  can take on only a small number of **discrete values** (such as if, given the fuel consumption, we wanted to predict if a car is a SUV or a small city car), we call it a **classification problem**.

## 2. Linear Regression



# Model hypothesis and parameters

- ▶ How can we predict the value of a numerical feature  $y$  based on the value of another numerical feature  $x$ ?
- ▶ First, we need to make some assumption about their relationship, i.e., how  $x$  influences  $y$ .

$$\hat{y} = w_0 + w_1 x$$

- ▶ Our model thus assumes that there is a linear relationship between the two features  $x$  and  $y$ .

→ How can we determine the coefficients  $w_0$  and  $w_1$ ?

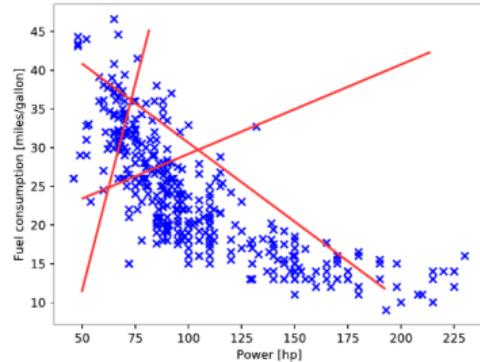
# Assume a Linear Model

Dependent Feature/Variable      Independent Feature/Variable

$$\hat{y} = w_0 + w_1 x$$

Coefficient      Coefficient

- ▶ Different values of  $w_0$  and  $w_1$  correspond to different lines in our plot.
- ▶ Which ones are best?
- ▶ Based on the linear equation that we defined previously, linear regression can be understood as finding **the best-fitting straight line** through the sample points, as shown in the following figure.



# Digression: The UCI ML Dataset Repository

<https://archive.ics.uci.edu/datasets>

**Filters**

Search datasets... 

Keywords 

Data Type 

Subject Area 

Task 

# Features 

# Instances 

Feature Type 

Python 

**Browse Datasets**

 SORT BY # VIEWS, DESC  EXPAND ALL

**Iris**  
A small classic dataset from Fisher, 1936. One of the earliest known datasets used for evaluating classification methods.  
 Classification Tabular 150 Instances 4 Features

**Dry Bean Dataset**  
Images of 13,611 grains of 7 different registered dry beans were taken with a high-resolution camera. A total of 16 features; 12 dimensions and...  
 Classification Multivariate 13.61K Instances 16 Features

**Rice (Cammeo and Osmancik)**  
A total of 3810 rice grain's images were taken for the two species, processed and feature inferences were made. 7 morphological features were q...  
 Classification Multivariate 3.81K Instances 7 Features

**Heart Disease**  
4 databases: Cleveland, Hungary, Switzerland, and the VA Long Beach  
 Classification Multivariate 303 Instances 13 Features

**Raisin**  
Images of the Kecimen and Besni raisin varieties were obtained with CVS. A total of 900 raisins were used, including 450 from both varieties, and...  
 Classification Multivariate 900 Instances 8 Features

**Adult**  
Predict whether income exceeds \$50K/yr based on census data. Also known as "Census Income" dataset.  
 Classification Multivariate 48.84K Instances 14 Features

# Let's look at the Example Data Set

Example by <https://archiveics.uci.edu/ml/datasets/Auto+MPG> and K. Beberich (2017)

- We determine the values of the coefficients  $w_0$  and  $w_1$  based on training data that is available to us.
- Our training data consists of  $n$  data points

$$(x_i, y_i)$$

- in our example those are pairs of power (in hp) and fuel consumption (in mpg) of individual cars

The diagram illustrates the structure of the Auto MPG dataset. It shows three components: 'Attribute Information' (a list of car features and their types), 'demo/auto-mpg.names.txt' (the header file containing feature names), and 'demo/auto-mpg.data.txt' (the data file containing numerical values and car names). Arrows point from the 'Attribute Information' list to both the names and data files.

| Attribute Information: |                                   |                       |  |  |  |  |  |  |
|------------------------|-----------------------------------|-----------------------|--|--|--|--|--|--|
| 1. mpg:                | continuous                        |                       |  |  |  |  |  |  |
| 2. cylinders:          |                                   | multi-valued discrete |  |  |  |  |  |  |
| 3. displacement:       | continuous                        |                       |  |  |  |  |  |  |
| 4. horsepower:         | continuous                        |                       |  |  |  |  |  |  |
| 5. weight:             | continuous                        |                       |  |  |  |  |  |  |
| 6. acceleration:       | continuous                        |                       |  |  |  |  |  |  |
| 7. model year:         | multi-valued discrete             |                       |  |  |  |  |  |  |
| 8. origin:             | multi-valued discrete             |                       |  |  |  |  |  |  |
| 9. car name:           | string (unique for each instance) |                       |  |  |  |  |  |  |

demo/auto-mpg.names.txt

demo/auto-mpg.data.txt

|      |   |       |       |       |      |    |   |                             |
|------|---|-------|-------|-------|------|----|---|-----------------------------|
| 18.0 | 8 | 307.0 | 130.0 | 3504. | 12.0 | 70 | 1 | "chevrolet chevelle malibu" |
| 15.0 | 8 | 350.0 | 165.0 | 3693. | 11.5 | 70 | 1 | "buick skylark 320"         |
| 18.0 | 8 | 318.0 | 150.0 | 3436. | 11.0 | 70 | 1 | "plymouth satellite"        |
| 16.0 | 8 | 304.0 | 150.0 | 3433. | 12.0 | 70 | 1 | "amc rebel sst"             |
| 17.0 | 8 | 302.0 | 140.0 | 3449. | 10.5 | 70 | 1 | "ford torino"               |
| 15.0 | 8 | 429.0 | 198.0 | 4341. | 10.0 | 70 | 1 | "ford galaxie 500"          |

# Recall: Mean, Variance, Standard Deviation

- We define the **mean** of feature  $x$  and  $y$  in our training data as

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$$

- **Variance** of feature  $x$  and  $y$  in our training data is defined as

$$\sigma_x^2 = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 \quad \sigma_y^2 = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2$$

- The values  $\sigma_x$  and  $\sigma_y$  are referred to as the **standard deviation** of features  $x$  and  $y$ .

## Recall: Covariance

- ▶ Covariance  $\text{cov}_{xy}$  measures the degree of joint variability between the two features  $x$  and  $y$

$$\text{cov}_{x,y} = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})$$

- ▶ Large co-variance suggests that the two features vary jointly
  - ▶ a positive value indicates that they tend to deviate from their respective mean in the same direction
  - ▶ a negative value indicates that they tend to deviate from their respective mean in opposite directions.

# Correlation

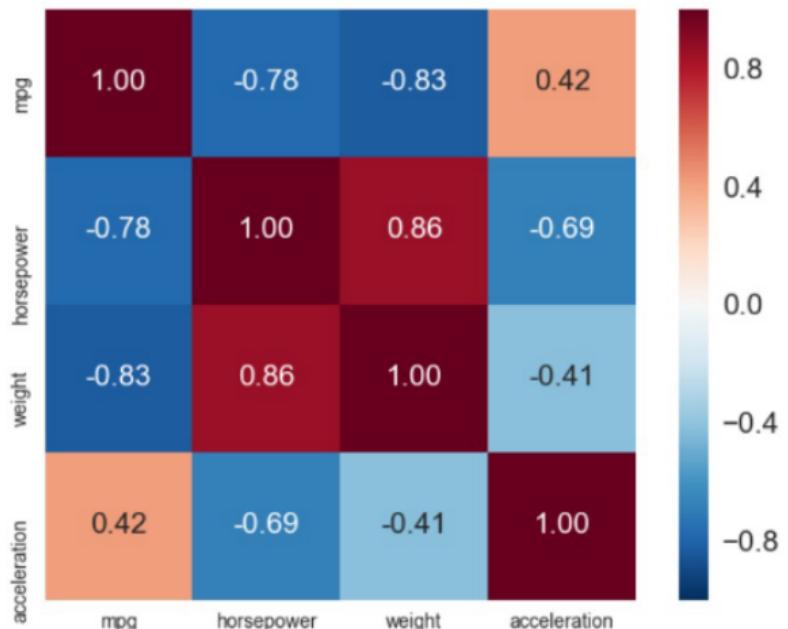
- The **correlation** coefficient (also **Pearson's  $r$** ) is a **normalized measure** of linear correlation between the two features  $x$  and  $y$

$$\text{cor}_{x,y} = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} = \frac{\text{cov}_{x,y}}{\sigma_x \sigma_y}$$

- The correlation coefficient takes values in  $[-1, +1]$ 
  - a value of  $-1$  indicates a negative linear correlation
  - a value of  $0$  indicates that there is no linear correlation
  - a value of  $+1$  indicates a positive linear correlation

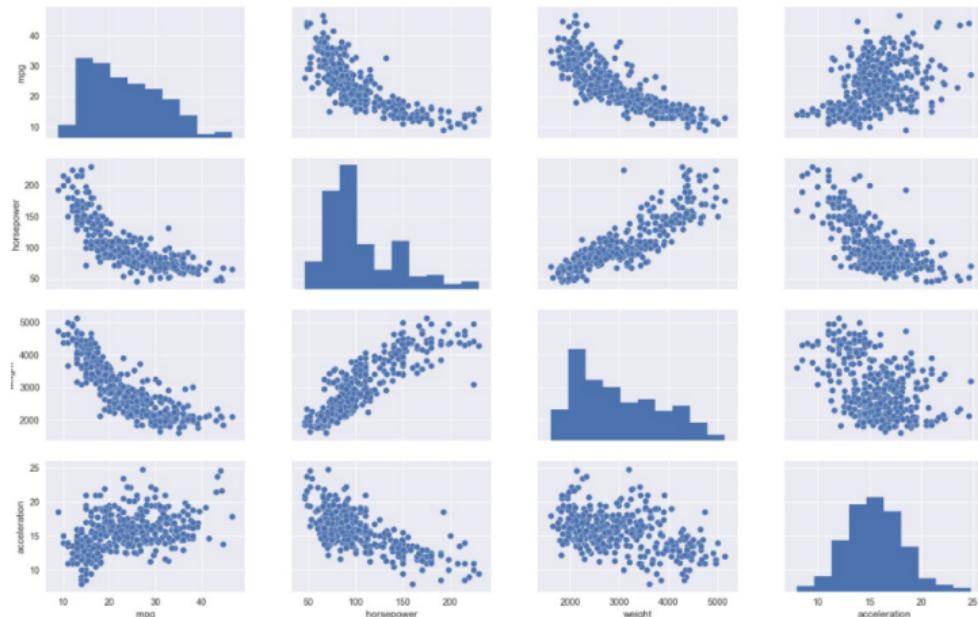
# Example Correlations

[demo/predict\\_prices.py](#)



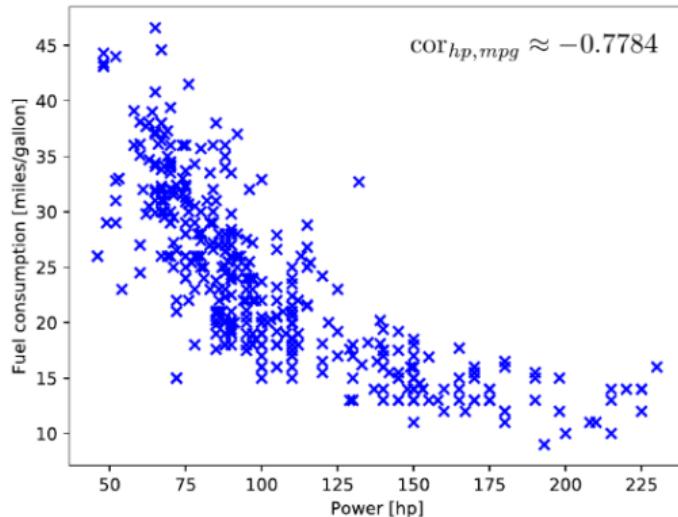
# Basic Analytics — Scatter Plot

[demo/predict\\_prices.py](#)



# Our Data Set

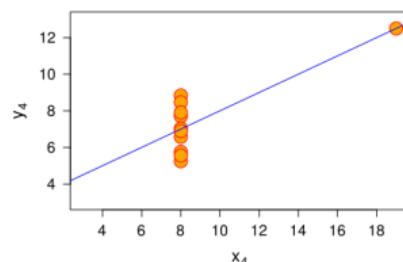
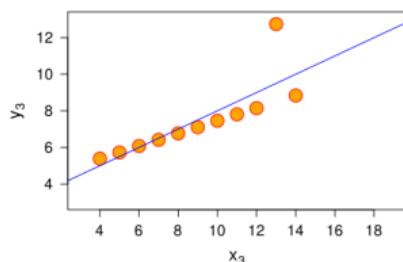
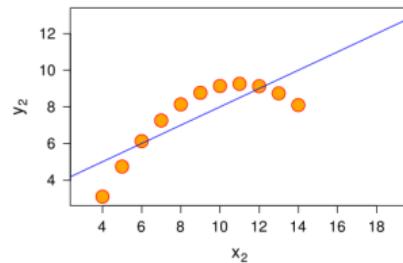
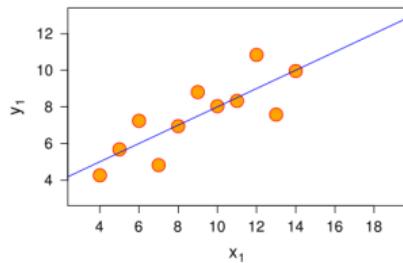
`demo/predict_prices.py`



# Anscombe's Quartet

[https://en.wikipedia.org/wiki/Wikipedia:Featured\\_picture\\\_candidates/Anscombe%27s\\_quartet](https://en.wikipedia.org/wiki/Wikipedia:Featured_picture\_candidates/Anscombe%27s_quartet)

All four datasets have the same mean, variance, correlation coefficient, and optimal regression line.



# Loss function / Cost function

- A Loss/Cost function measures how well our model, for a specific choice of coefficients  $w_0$  and  $w_1$ , describes the training data

*"how much we lose by using the model"*
- The **Residual** for data point  $(x_i, y_i)$  measures how much the observed value  $y_i$  differs from the prediction of our model

$$(y_i - \hat{y}_i) = (y_i - (w_0 + w_1 x_i)) = (y_i - w_0 - w_1 x_i)$$

# Loss function / Cost function

- ▶ Ordinary least squares (OLS) uses the sum of squared residuals (also: sum of squared errors) as a loss function

$$L(w_0, w_1) = \sum_{i=1}^n (y_i - w_0 - w_1 x_i)^2$$

- ▶ Since we're interested in finding the coefficients  $w_0$  and  $w_1$  that minimize the loss, we obtain the optimization problem

$$\arg \min_{w_0, w_1} \sum_{i=1}^n (y_i - w_0 - w_1 x_i)^2$$

# Minimize the Cost Function

Optimal values for the coefficients  $w_0$  and  $w_1$  can be determined analytically in the case of OLS:

- ▶ compute partial derivatives of loss function w.r.t.  $w_0$  and  $w_1$

$$\frac{\partial L}{\partial w_0} = -2 \sum_{i=1}^n (y_i - w_0 - w_1 x_i)$$

$$\frac{\partial L}{\partial w_1} = -2 \sum_{i=1}^n (y_i - w_0 - w_1 x_i) x_i$$

- ▶ identify common zero by solving system of equations

$$\frac{\partial L}{\partial w_0} = 0 \quad \frac{\partial L}{\partial w_1} = 0$$

# The optimal coefficients

→ We obtain the following **closed-form solutions** to compute optimal values of the coefficients based on our data

$$w_0^* = \frac{1}{n} \sum_{i=1}^n y_i - w_1^* \frac{1}{n} \sum_{i=1}^n x_i$$
$$w_1^* = \frac{n \sum_{i=1}^n x_i y_i - (\sum_{i=1}^n x_i) (\sum_{i=1}^n y_i)}{n \sum_{i=1}^n x_i^2 - (\sum_{i=1}^n x_i)^2}$$

## $R^2$ : Coefficient of Determination

- The  $R^2$  coefficient of determination (short: “R squared”) measures how well the determined regression line approximates the data.
- Put differently: how much of the variation observed in the data is explained by it.

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

# Linear Regression in Python

demo/predict\_prices.py

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import linear_model, metrics

### download the original data set -- it has a bunch of NaN's)
#cars = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data',
# header=None,
# sep='\s+')
### Clean data set
cars = pd.read_csv('auto-mpg.data.txt',header=None, sep='\s+')
### Label columns
cars.columns = ['mpg', 'cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model year', 'origin', 'car name']
print(cars.head())

#### Some scatter plots
cols = ['mpg', 'horsepower', 'weight', 'acceleration']
sns.pairplot(cars[cols], size=2.5)
plt.show()

### extract the fuel consumption
y = cars.iloc[:,0].values

### horsepower
X = cars.iloc[:,[3]].values
print(X.size)
```

# Cont.

```
### Plot
g = sns.regplot(x=X, y=y, fit_reg=False)

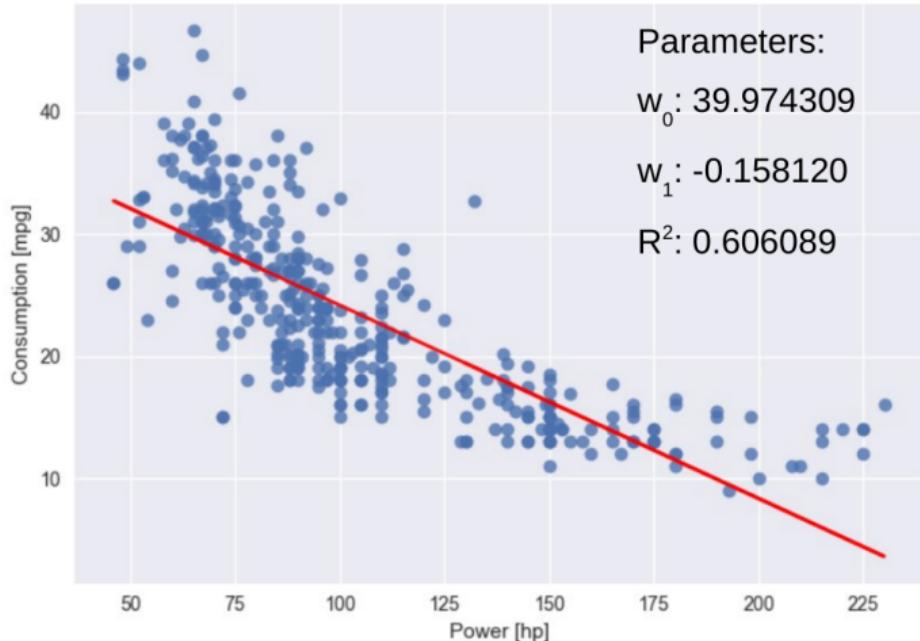
#### Correlation
#cm = np.corrcoef(cars[cols].values.T)
#sns.set(font_scale=1.5)
#hm = sns.heatmap(cm, cbar=True, annot=True, square=True, fmt='.2f', annot_kws={'size': 15}, yticklabels=cols, xticklabels=cols)
#print("correlation", cm)

### Linear Regression
reg = linear_model.LinearRegression()
reg.fit(X,y)
plt.plot(X, reg.predict(X), color='red')

### Labels
plt.xlabel('Power [hp]')
plt.ylabel('Consumption [mpg]')
plt.show()

# Coefficients and R2
print('Parameters:')
print('w0: %f'%reg.intercept_)
print('w1: %f'%reg.coef_[0])
print('R2: %f'%metrics.r2_score(y,reg.predict(X)))
```

# Regression Plot



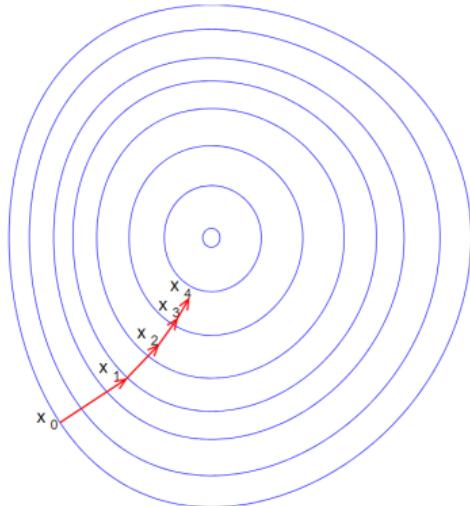
### 3. Recall – Gradient Descent

- ▶ Unfortunately, it is not always possible to determine optimal coefficients for a loss function analytically.
- ▶ Gradient descent is an optimization algorithm that we can use to determine the minimum of a loss function.
- ▶ Basic Idea:
  - ▶ start with a **random choice of the coefficients** (here:  $w_0$  and  $w_1$  )
  - ▶ repeat for a specified number of rounds or until convergence
    - ▶ compute the gradient for this choice of coefficients
    - ▶ **update coefficients based on the gradient**

# Gradient Descent (II)

Intuition: Think of the loss function as a surface on which you want to find the lowest point

- ▶ start your journey at a random position
- ▶ repeat the following
  - ▶ identify direction with steepest descent
  - ▶ walk a few steps in the identified direction



[https://en.wikipedia.org/wiki/  
Gradient\\_descent](https://en.wikipedia.org/wiki/Gradient_descent)

# Gradient Descent in Python

demo/gradient\_descent.py

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k.$$

The gradient descent algorithm is applied to find a local minimum of the function  $f(x) = x^4 - 3x^3 + 2$  with derivative  $f'(x) = 4x^3 - 9x^2$ .

```
cur_x = 6 # The algorithm starts at x=6
gamma = 0.01 # step size multiplier
precision = 0.00001
previous_step_size = 1
max_iters = 10000 # maximum number of iterations
iters = 0 #iteration counter

df = lambda x: 4 * x**3 - 9 * x**2

while previous_step_size > precision and iters < max_iters:
    prev_x = cur_x
    cur_x -= gamma * df(prev_x)
    previous_step_size = abs(cur_x - prev_x)
    iters+=1

print("The local minimum occurs at", cur_x)
#The output for the above will be:
# ('The local minimum occurs at', 2.2499646074278457)
```

# The LMS algorithm (least mean square)

Gradient of a multivariate function (e.g., **the Loss function  $J(\theta)$** ) is defined as the **vector of its partial derivatives**; when evaluated at a specific point, it indicates the direction of steepest ascent. Specifically, let's consider the gradient descent algorithm, which starts with some initial  $\theta$ , and repeatedly performs the update:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta).$$

This update is simultaneously performed for all values of  $j = 1, \dots, n$ . Here,  $\alpha$  is called **the learning rate ( $0 < \alpha \leq 1$ )**. This is a very natural algorithm that repeatedly takes a step in the direction of steepest decrease of  $J$ .

# The LMS Algorithm (II)

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. Let's first work it out for the case of if we have only one training example  $(x, y)$ , so that we can neglect the sum in the definition of  $J$ . We have:

$$\begin{aligned}\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_{\theta}(x) - y)^2 \\&= 2 \cdot \frac{1}{2} (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_{\theta}(x) - y) \\&= (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left( \sum_{i=0}^n \theta_i x_i - y \right) \\&= (h_{\theta}(x) - y) x_j\end{aligned}$$

# The LMS Algorithm (III)

- ▶ For a single training example, this gives the update rule:

$$\theta_j := \theta_j + \alpha \left( y^{(i)} - h_{\theta} \left( x^{(i)} \right) \right) x_j^{(i)}.$$

- ▶ This rule has several properties that seem natural and intuitive.
- ▶ For instance, the magnitude of the update is proportional to the error term  $(y^{(i)} - h_{\theta}(x^{(i)}))$ .
- ▶ Thus, for instance, if we are encountering a training example on which our prediction nearly matches the actual value of  $y^{(i)}$ , then we find that there is little need to change the parameters.
- ▶ In contrast, a larger change to the parameters will be made if our prediction  $h_{\theta}(x^{(i)})$  has a large error (i.e., if it is very far from  $y^{(i)}$  ).

# The LMS Algorithm (IV)

- ▶ This method looks at every example in the entire training set on every step, and is called **batch gradient descent**.
- ▶ Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima.
- ▶ Thus gradient descent always converges to the global minimum as  $J$  is a convex quadratic function.

# Stochastic Gradient Descent (relevant e.g. for Neural Nets)

- ▶ Gradient ascent as a counterpart to maximize functions.
- ▶ Stochastic gradient descent (SGD) does not compute the true gradient, but approximates the gradient based on **a single or few randomly chosen data points in each round**.
- ▶ Gradient can be approximated when the function at hand is non-differentiable or when partial derivatives are expensive to compute.

# SGD (II)

```
Loop {  
    for i=1 to m, {  
         $\theta_j := \theta_j + \alpha (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}$       (for every j).  
    }  
}
```

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only.

Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if  $m$  is large—stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at.

Often, stochastic gradient descent gets  $\theta$  “close” to the minimum much faster than batch gradient descent. (Note however that it may never “converge” to the minimum, and the parameters  $\theta$  will keep oscillating around the minimum of  $J(\theta)$ ; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum) → particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

# SGD (II)

```
Loop {  
    for i=1 to m, {  
         $\theta_j := \theta_j + \alpha (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}$       (for every j).  
    }  
}
```

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only.

Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if  $m$  is large—stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at.

Often, stochastic gradient descent gets  $\theta$  “close” to the minimum much faster than batch gradient descent. (Note however that it may never “converge” to the minimum, and the parameters  $\theta$  will keep oscillating around the minimum of  $J(\theta)$ ; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum) → particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

# SGD (II)

```
Loop {  
    for i=1 to m, {  
         $\theta_j := \theta_j + \alpha (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}$       (for every j).  
    }  
}
```

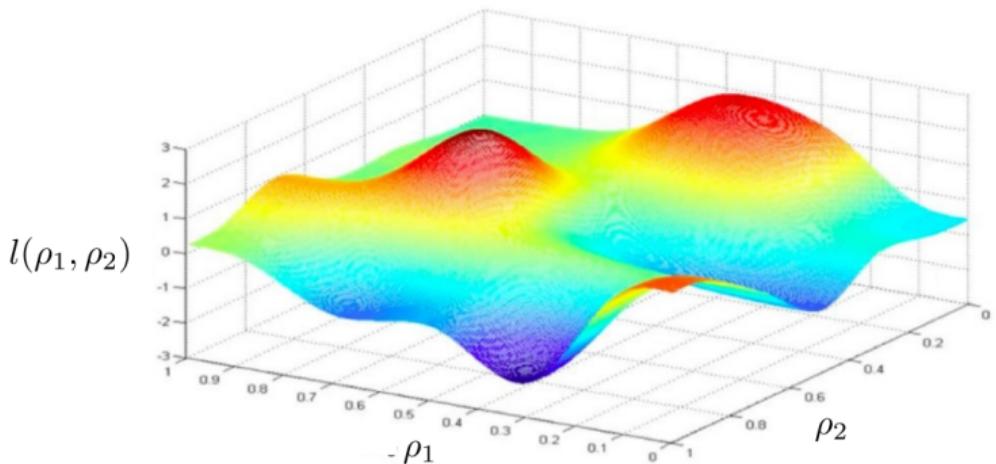
In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only.

Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if  $m$  is large—stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at.

Often, stochastic gradient descent gets  $\theta$  “close” to the minimum much faster than batch gradient descent. (Note however that it may never “converge” to the minimum, and the parameters  $\theta$  will keep oscillating around the minimum of  $J(\theta)$ ; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum) → particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

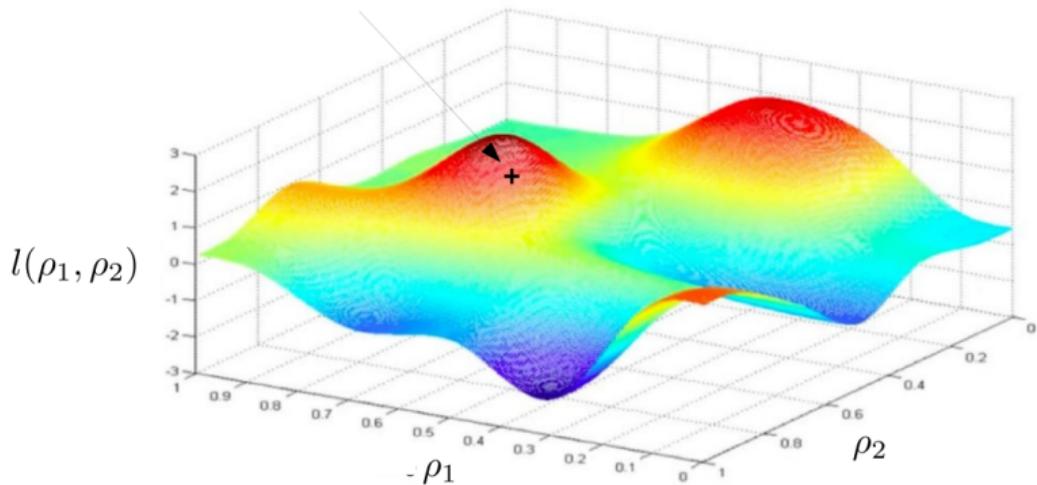
# Gradient Descent in parameter space

$$\rho^* = \operatorname{argmin}_\rho l(\rho)$$



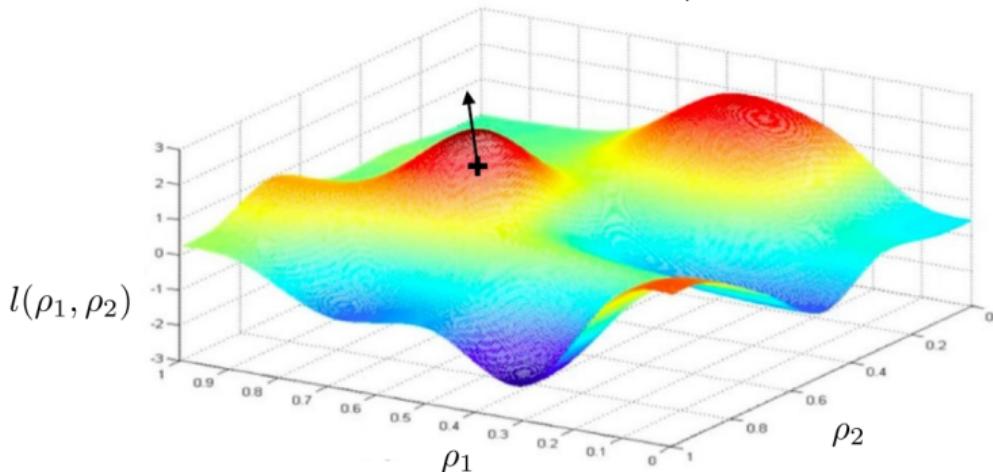
# Loss Optimization

Randomly pick a an initial  $\rho_1$  and  $\rho_2$



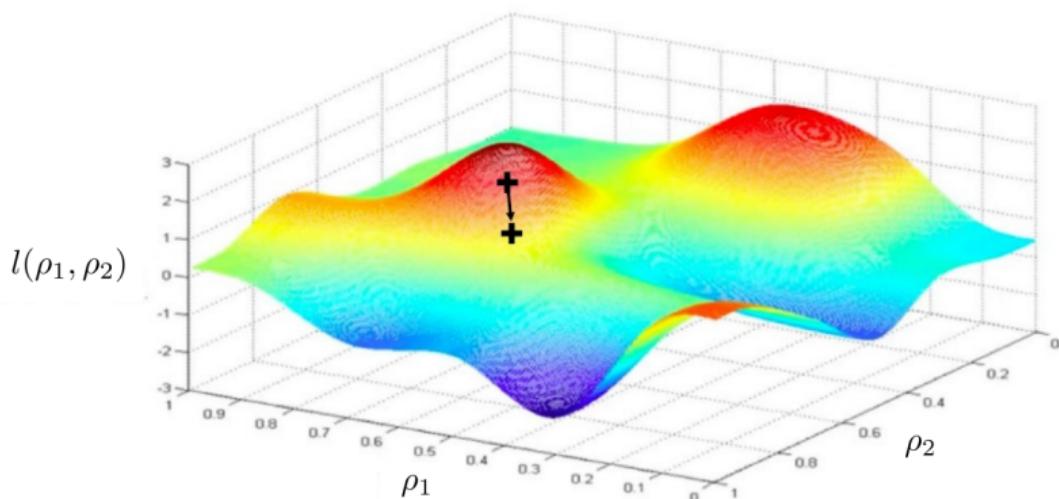
# Loss Optimization

Compute gradient  $\frac{\partial l}{\partial \rho}$  → **Steepest ascent**



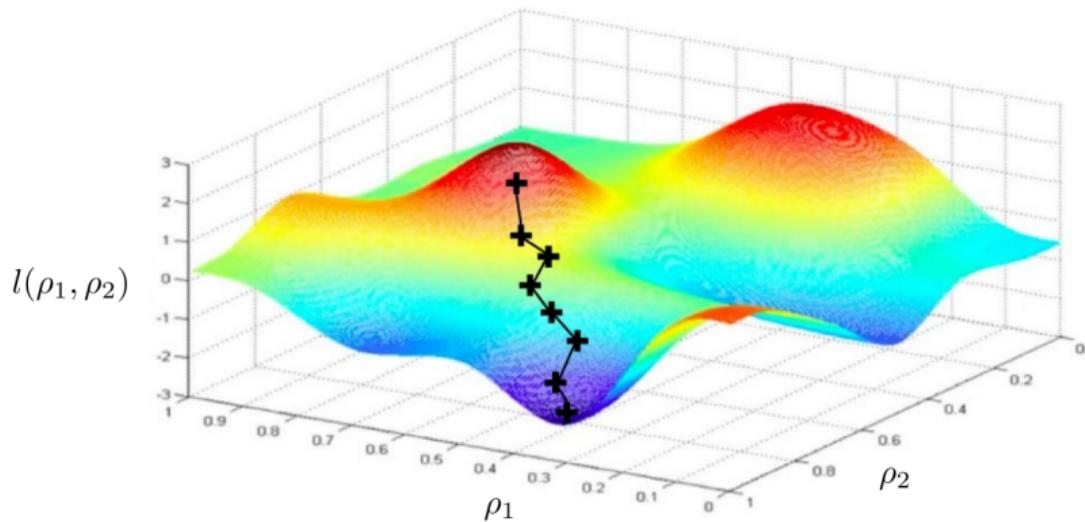
# Loss Optimization

Take a small step in the opposite direction of the gradient.



# Loss Optimization

Repeat until convergence



# Mini Batch Gradient Descent

- ▶ In actual practice we use an approach called **Mini batch gradient descent**.
- ▶ This approach uses random samples but in batches.
- ▶ What this means is that we do **not calculate the gradients for each observation** but for **a group of observations** which results in a faster optimization.
- ▶ A simple way to implement is to **shuffle the observations** and then create batches and then proceed with gradient descent using batches.

# Let's run the Jupyter Notebook

demo/GradientDescent\_StochasticGradientDescent.ipynb



## 4. Regression with multiple variables

- ▶ How can we predict the value of a numerical feature  $y$  based on **multiple other numerical features  $x_1, \dots, x_m$**  ?
- ▶ We assume that there is a **linear relationship** between the target feature and the other features

$$\hat{y} = w_0 + w_1x_1 + \dots + w_mx_m$$

- ▶ Given that we now deal with **multiple data points and multiple features**, it is easier to formulate our optimization problem using matrices and vectors.

# Minimize the Cost Function

The training examples' **input values** in its rows. Also, let  $\vec{y}$  be the  $m$ -dimensional vector containing all the target values from the **training set**:

$$X = \begin{bmatrix} - (x^{(1)})^T - \\ - (x^{(2)})^T - \\ \vdots \\ - (x^{(m)})^T - \end{bmatrix} \quad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

Now, since  $h_{\theta}(x^{(i)}) = (x^{(i)})^T \theta$  (where  $\theta$  is the vector of coefficients), we can easily verify that

$$X\theta - \vec{y} = \begin{bmatrix} (x^{(1)})^T \theta \\ \vdots \\ (x^{(m)})^T \theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} = \begin{bmatrix} h_{\theta}(x^{(1)}) - y^{(1)} \\ \vdots \\ h_{\theta}(x^{(m)}) - y^{(m)} \end{bmatrix}$$

# Minimize the Cost Function

Thus, using the fact that for a vector  $z$ , we have that  $z^T z = \sum_i z_i^2$ :

$$\begin{aligned}\frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) &= \frac{1}{2} \sum_{i=1}^m \left( h_\theta(x^{(i)}) - y^{(i)} \right)^2 \\ \nabla_\theta L(\Theta) &= \nabla_\theta \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) \\ &= \frac{1}{2} \nabla_\theta (\theta^T X^T X \theta - \theta^T X^T \vec{y} - \vec{y}^T X \theta + \vec{y}^T \vec{y}) \\ &= \frac{1}{2} \nabla_\theta \text{tr} (\theta^T X^T X \theta - \theta^T X^T \vec{y} - \vec{y}^T X \theta + \vec{y}^T \vec{y}) \\ &= \frac{1}{2} \nabla_\theta (\text{tr} \theta^T X^T X \theta - 2 \text{tr} \vec{y}^T X \theta) \\ &= \frac{1}{2} (X^T X \theta + X^T X \theta - 2 X^T \vec{y}) \\ &= X^T X \theta - X^T \vec{y}\end{aligned}$$

## Minimize the Cost Function (III)

To minimize  $J$ , we set its **derivatives to zero**, and obtain the normal equations:

$$X^T X \theta = X^T \vec{y}$$

Thus, the value of  $\theta$  that minimizes  $L(\theta)$  is given in closed form by the equation

$$\theta = (X^T X)^{-1} X^T \vec{y}$$

# Multivariate Linear Regression in Python

demo/multi\_par\_reg.py

<https://archive.ics.uci.edu/dataset/9/auto+mpg>

Let's apply this to our car dataset and try to predict fuel consumption based on horsepower and weight.

```
import pandas as pd
import numpy as np
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

# read cars dataset, a clean data set
cars = pd.read_csv('auto-mpg.data.txt', header=None, sep='\s+')

# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower and weight values
X = cars.iloc[:,[3,4]].values

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(X,y)

# coefficients
reg.intercept_ # 45.640210840177119
reg.coef_ # [-0.04730286 -0.00579416]
```

# Multivariate Linear Regression in Python

```
# compute correlation coefficient
np.corrcoef(reg.predict(X),y) # 0.84046135

# compute mean squared error (MSE)
sum((reg.predict(X) - y)**2) / len(y) # 17.841442442550584
```

Here, the last line computes the mean squared error (MSE) as another widely used measure for assessing the prediction quality of a regression model.

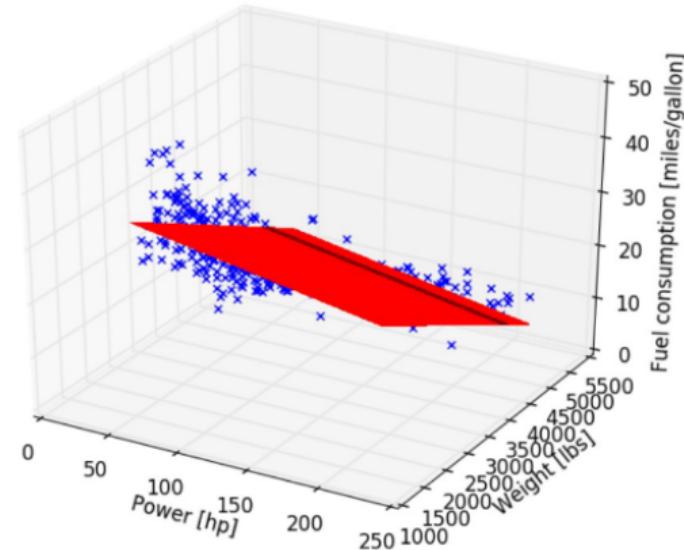
$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

# Inspect the regression — Hyperplane

```
### Plot the hyperplane
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# plot data points
for i in range(0,len(y)):
    ax.scatter(X[i,0], X[i,1], y[i],
               color='blue',
               marker='x')

# plot hyperplane
X0 = np.arange(min(X[:,0]), max(X[:,0]), 25)
X1 = np.arange(min(X[:,1]), max(X[:,1]), 25)
X0, X1 = np.meshgrid(X0, X1)
Z = X0.copy()
n = X0.shape[0]
m = X0.shape[1]
for i in range(0, n):
    for j in range(0, m):
        Z[i,j] = reg.predict([[X0[i,j], X1[i,j]]])
ax.plot_surface(X0, X1, Z, color='red',
                 linewidth=0,
                 antialiased=False)
ax.set_xlabel('Power [hp]')
ax.set_ylabel('Weight [lbs]')
ax.set_zlabel('Fuel consumption [miles/gallon]')
plt.show()
```



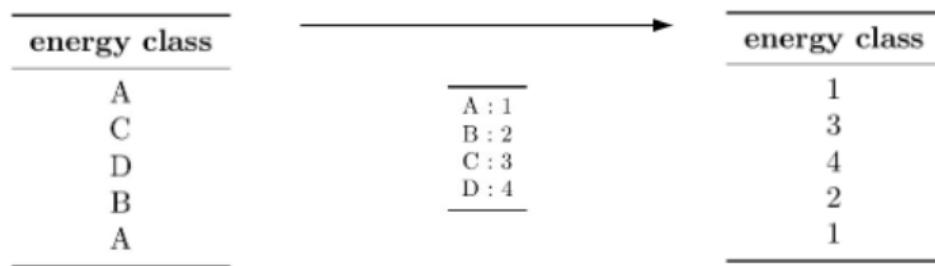
# Handling non-numerical Features

- ▶ How can we **make non-numerical** (i.e., **nominal and ordinal**) features accessible to regression analysis?
- ▶ **Nominal features** (e.g., origin of a car) can be converted **using one-hot encoding**: for each value of the original feature, a binary feature is introduced, indicating whether a data point has the corresponding value for the feature.

| origin |   | origin_1 | origin_2 | origin_3 |
|--------|---|----------|----------|----------|
| 1      | → | 1        | 0        | 0        |
| 3      |   | 0        | 0        | 1        |
| 1      |   | 1        | 0        | 0        |
| 2      |   | 0        | 1        | 0        |

# Handling non-numerical Features

- ▶ **Ordinal features** (e.g., energy efficiency class) can be mapped to integer values preserving their order.



- ▶ Note that this mapping implicitly assumes that the differences between adjacent values are uniform, i.e., have the same magnitude.

# Predict Miles/Gallon with Origin as Feature

demo/multi\_par\_reg\_add\_features.py

```
import pandas as pd
import numpy as np
from sklearn import linear_model, preprocessing

# read cars dataset, a clean data set
cars = pd.read_csv('auto-mpg.data.txt', header=None, sep='\s+')

# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower and weight values, apply one-hot encoding for origin
X = pd.concat([cars.iloc[:,[3,4]], pd.get_dummies(cars[7])], axis = 1).values

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(X,y)

# coefficients
reg.intercept_ # 43.974410233714622
reg.coef_ # [-0.05354417, -0.00484275, -1.2344519 , -0.27333471, 1.50778661]
```

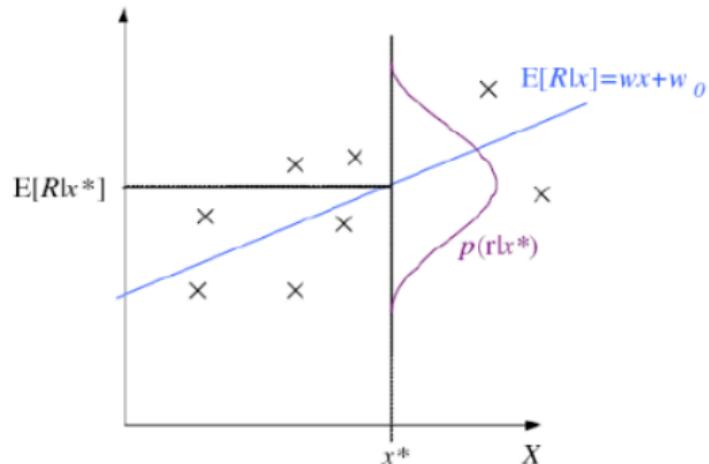
# Predict Miles/Gallon with Origin as Feature

```
# compute correlation coefficient
np.corrcoef(reg.predict(X),y) # 0.84810338
# compute mean squared error (MSE)
sum((reg.predict(X) - y)**2) / len(y) # 17.057355871889044
```

→ Origin as an additional features reduces mean squared error and allows the model to encode knowledge about fuel efficiency of cars from different origins (U.S.A. least efficient, Japan most efficient).

## 5. A Probabilistic Interpretation of Linear Regression

- When faced with a regression problem, why might linear regression, and **specifically why might the least-squares cost function  $J$ , be a reasonable choice?**
- We look now at a **set of probabilistic assumptions**, under which least-squares regression is derived as a very natural algorithm.



Let us assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)}$$

where  $\epsilon^{(i)}$  is an error term that captures for instance random noise.

# Probability and Regression

Let us further assume that the  $\varepsilon^{(0)}$  are distributed i.i.d (independently and identically distributed) according to a **Gaussian distribution** with **mean zero** and some **variance**  $\sigma^2$ . We can write this assumption as

$$\varepsilon^{(1)} \sim N(0, \sigma^2).$$

i.e., the density of  $\varepsilon^{(0)}$  is given by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}\right)$$

# Probability and Regression (II)

- This implies that

$$p(y^{(i)} | x^{(i)}; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

- The notation “ $p(y^{(i)} | x^{(i)}; \theta)$ ” indicates that this is the distribution of  $y^{(i)}$  given  $x^{(i)}$  and **parameterized by  $\theta$ .**
- Note that we should not condition on  $\theta$  (“ $p(y^{(i)} | x^{(i)}, \theta)$ ”), since  $\theta$  is not a random variable.
- We can also write the distribution of  $y^{(i)}$  as  $y^{(i)} | x^{(i)}; \theta \sim N(\theta^T x^{(i)}, \sigma^2)$ .

# Likelihood Function

- ▶ Given  $X$  (the design matrix, which contains all the  $X^{(i)}$ 's) and  $\theta$ , what is the distribution of the  $y^{(i)}$ 's?
- ▶ The probability of the data is given by  $p(\vec{y} | X; \theta)$  This quantity is typically viewed a function of  $\vec{y}$  (and perhaps  $X$ ), for a fixed value of  $\theta$ .
- ▶ When we wish to explicitly view this as a function of  $\theta$ , we will instead call it the likelihood function:

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y} | X; \theta)$$

## Likelihood Function (II)

Note that by the independence assumption on the  $\varepsilon^{(i)}$ 's (and hence also the  $y^{(i)}$ 's given the  $x^{(i)}$ 's), this can also be written

$$\begin{aligned} L(\theta) &= \prod_{i=1}^m p\left(y^{(i)} \mid x^{(i)}; \theta\right) \\ &= \prod_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right). \end{aligned}$$

# Likelihood Function (III)

- ▶ Now, given this probabilistic model relating the  $y^{(i)}$ 's and the  $x^{(i)}$ 's, what is a reasonable way of choosing **our best guess of the parameters  $\theta$**  ?
- ▶ The principle of **maximum likelihood** says that we should choose  $\theta$  so as to **make the data as high probability as possible** — that is, we should choose  $\theta$  to **maximize  $L(\theta)$** .
- ▶ Instead of maximizing  $L(\theta)$ , we can also **maximize any strictly increasing function of  $L(\theta)$** . In particular, the derivations will be a bit simpler if we instead maximize the **log likelihood  $I(\theta)$**  :

$$\ell(\theta) = \log L(\theta)$$

$$\begin{aligned} &= \log \prod_{i=1}^m \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^m \log \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right) \\ &= m \log \frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{\sigma^2} \cdot \frac{1}{2} \sum_{i=1}^m (y^{(i)} - \theta^T x^{(i)})^2. \end{aligned}$$

# Maximum Likelihood

- ▶ Hence, **maximizing  $I(\theta)$  gives the same answer as minimizing**

$$\frac{1}{2} \sum_{i=1}^m \left( y^{(i)} - \theta^T x^{(i)} \right)^2,$$

which we **recognize to be  $J(\theta)$** , our original least-squares cost function.

- ▶ Under the previous probabilistic assumptions on the data, least-squares regression corresponds to **finding the maximum likelihood estimate of  $\theta$** .
- ▶ This is thus one set of assumptions under which **least-squares regression** can be justified as a very natural method that's just **doing maximum likelihood estimation**.

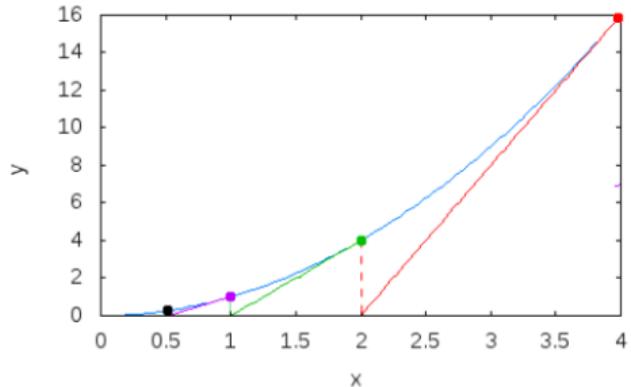
# Newton's Method — another way of maximizing functions

- ▶ Let's now talk about a different algorithm for maximizing e.g.  $\ell(\theta)$ .
- ▶ Let's consider Newton's method for finding a zero of a function.
- ▶ Suppose we have some function  $f: \mathbb{R} \rightarrow \mathbb{R}$ , and we wish to find a value of  $\theta$  such that  $f(\theta) = 0$ . Here,  $\theta$  is a real number.
- ▶ Newton's method performs the following update:

$$\theta := \theta - \frac{f(\theta)}{f'(\theta)}$$

# Newton's Method

Newton's method has a natural interpretation. We can think of it as approximating the function  $f$  via a linear function that is tangent to  $f$  at the current guess  $\theta$ , solving for where that linear function equals to zero, and letting the next guess for  $\theta$  be where that linear function is zero.



[https://en.wikibooks.org/wiki/Calculus/Newton%27s\\_Method](https://en.wikibooks.org/wiki/Calculus/Newton%27s_Method)

## Newton's Method (II)

- ▶ Newton's method gives a way of getting to  $f(\theta) = 0$ .
- ▶ What if we want to use it to maximize some function  $\ell$ ? The maximum of  $\ell$  correspond to points where its first derivative  $\ell'(\theta)$  is zero.
- ▶ So, by letting  $f(\theta) = \ell'(\theta)$ , we can use the same algorithm to maximize  $\ell$ , and we obtain update rule:

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}$$

# Newton's Method for multiple variables

- The generalization of Newton's method to a multidimensional setting (also called the **Newton-Raphson method**) is given by

$$\theta := \theta - H^{-1} \nabla_{\theta} \ell(\theta).$$

- Here,  $\nabla_{\theta} \ell(\theta)$  is, as usual, the vector of partial derivatives of  $\ell(\theta)$  with respect to the  $\theta_i$ 's; and  $H$  is an n-by-n matrix (actually,  $n + 1$ -by-  $n + 1$ , assuming that we include the intercept term) called the Hessian, whose entries are given by

$$H_{ij} = \frac{\partial^2 \ell(\theta)}{\partial \theta_i \partial \theta_j}.$$

- Newton's method typically enjoys **faster convergence** than gradient descent, and requires fewer iterations to get very close to the minimum. One iteration of Newton's can, however, be more expensive than one iteration of gradient descent, since it requires **finding and inverting an n-by-n Hessian**; but so long as  $n$  is not too large, it is usually much faster overall.

# Newton's Method: Pros and Cons

- ▶ When it converges, Newton's method usually converges very quickly and this is its main advantage. However, **Newton's method is not guaranteed to converge** and this is obviously a big disadvantage especially compared to the bisection and secant methods which are guaranteed to converge to a solution (provided they start with an interval containing a root).
- ▶ Newton's method also requires computing values of the derivative of the function in question. This is potentially a disadvantage if the derivative is difficult to compute.
- ▶ The stopping criteria for Newton's method differs from the bisection and secant methods. In those methods, we know how close we are to a solution because we are computing intervals which contain a solution. In Newton's method, we don't know how close we are to a solution. **All we can compute is the value  $f(x)$  and so we implement a stopping criteria based on  $f(x)$ .**
- ▶ Finally, there's no guarantee that the method converges to a solution and we should set a maximum number of iterations so that our implementation ends if we don't find a solution.

# Newton's Method: Pros and Cons

- ▶ When it converges, Newton's method usually converges very quickly and this is its main advantage. However, **Newton's method is not guaranteed to converge** and this is obviously a big disadvantage especially compared to the bisection and secant methods which are guaranteed to converge to a solution (provided they start with an interval containing a root).
- ▶ Newton's method also requires computing values of the derivative of the function in question. **This is potentially a disadvantage if the derivative is difficult to compute.**
- ▶ The stopping criteria for Newton's method differs from the bisection and secant methods. In those methods, we know how close we are to a solution because we are computing intervals which contain a solution. In Newton's method, we don't know how close we are to a solution. **All we can compute is the value  $f(x)$  and so we implement a stopping criteria based on  $f(x)$ .**
- ▶ Finally, there's no guarantee that the method converges to a solution and we should set a maximum number of iterations so that our implementation ends if we don't find a solution.

# Newton's Method: Pros and Cons

- ▶ When it converges, Newton's method usually converges very quickly and this is its main advantage. However, **Newton's method is not guaranteed to converge** and this is obviously a big disadvantage especially compared to the bisection and secant methods which are guaranteed to converge to a solution (provided they start with an interval containing a root).
- ▶ Newton's method also requires computing values of the derivative of the function in question. **This is potentially a disadvantage if the derivative is difficult to compute.**
- ▶ The stopping criteria for Newton's method differs from the bisection and secant methods. In those methods, we know how close we are to a solution because we are computing intervals which contain a solution. In Newton's method, we don't know how close we are to a solution. **All we can compute is the value  $f(x)$  and so we implement a stopping criteria based on  $f(x)$ .**
- ▶ Finally, there's no guarantee that the method converges to a solution and we should set a maximum number of iterations so that our implementation ends if we don't find a solution.

# Newton's Method: Pros and Cons

- ▶ When it converges, Newton's method usually converges very quickly and this is its main advantage. However, **Newton's method is not guaranteed to converge** and this is obviously a big disadvantage especially compared to the bisection and secant methods which are guaranteed to converge to a solution (provided they start with an interval containing a root).
- ▶ Newton's method also requires computing values of the derivative of the function in question. **This is potentially a disadvantage if the derivative is difficult to compute.**
- ▶ The stopping criteria for Newton's method differs from the bisection and secant methods. In those methods, we know how close we are to a solution because we are computing intervals which contain a solution. In Newton's method, we don't know how close we are to a solution. **All we can compute is the value  $f(x)$  and so we implement a stopping criteria based on  $f(x)$ .**
- ▶ Finally, there's no guarantee that the method converges to a solution and we should **set a maximum number of iterations so that our implementation ends if we don't find a solution.**

# Newton's Method: Example

- ▶ Let's write a function called `newton` which takes 5 input parameters:  $f$ ,  $Df$ ,  $x_0$ ,  $\text{epsilon}$  and `max_iter` and returns an approximation of a solution of  $f(x)$  by Newton's method. The function may terminate in 3 ways:
- ▶ If  $\text{abs}(f(x_n)) < \text{epsilon}$  , the algorithm has found an approximate solution and returns  $x_n$ .
- ▶ If  $f'(x_n) == 0$ , the algorithm stops and returns None.
- ▶ If the number of iterations exceed `max_iter`, the algorithm stops and returns None.

# Example Code

demo/newton\_test.py and demo/newton\_solver.py

```
def newton(f,Df,x0,epsilon,max_iter):
    '''Approximate solution of f(x)=0 by Newton's method.

    Parameters
    ----
    f : function
        Function for which we are searching for a solution f(x)=0.
    Df : function
        Derivative of f(x).
    x0 : number
        Initial guess for a solution f(x)=0.
    epsilon : number
        Stopping criteria is abs(f(x)) < epsilon.
    max_iter : integer
        Maximum number of iterations of Newton's method.

    Returns
    ----
    xn : number
        Implement Newton's method: compute the linear approximation
        of f(x) at xn and find x intercept by the formula
        x = xn - f(xn)/Df(xn)
        Continue until abs(f(xn)) < epsilon and return xn.
        If Df(xn) == 0, return None. If the number of iterations
        exceeds max_iter, then return None.
    '''

    # Check for single argument functions
    if not hasattr(f, '__call__') or not hasattr(Df, '__call__'):
        raise TypeError('functions f and Df must be callable')
    if not (isinstance(x0, (int, float)) and
            isinstance(epsilon, (int, float)) and
            isinstance(max_iter, int)):
        raise ValueError('arguments must be numbers')
    if max_iter < 0:
        raise ValueError('max_iter must be non-negative')

    # Initialize x
    x = x0
    for i in range(max_iter):
        f_x = f(x)
        if abs(f_x) < epsilon:
            return x
        if Df(x) == 0:
            return None
        Df_x = Df(x)
        x = x - f_x/Df_x
    return None
```

# Example Code — Cont.

```
'''  
Examples  
-----  
>>> f = lambda x: x**2 - x - 1  
>>> Df = lambda x: 2*x - 1  
>>> newton(f,Df,1,1e-8,10)  
Found solution after 5 iterations.  
1.618033988749989  
'''  
  
xn = x0  
for n in range(0,max_iter):  
    fxn = f(xn)  
    if abs(fxn) < epsilon:  
        print('Found solution after',n,'iterations.')  
        return xn  
    Dfxn = Df(xn)  
    if Dfxn == 0:  
        print('Zero derivative. No solution found.')  
        return None  
    xn = xn - fxn/Dfxn  
print('Exceeded maximum iterations. No solution found.')  
return None
```

# Newton Solver — test 1

```
f = lambda x: x**3 - x**2 - 1
Df = lambda x: 3*x**2 - 2*x
approx = newton(f,Df,1.5,1e-10,10)
```

# Newton solver – divergent example

```
f = lambda x: x**(1/3)
Df = lambda x: (1/3)*x**(-2/3)
approx = newton(f,Df,0.5,1e-2,100)
```

## 6. Polynomial Regression

- ▶ What if the relationship between our **target feature  $y$**  and the independent features is “**more complicated**”?
- ▶ Polynomial regression allows us to estimate the optimal coefficients of a polynomial having degree  $d$

$$h_{\theta} = w_0 + w_1x + w_2x^2 + \dots + w_dx^d$$

- ▶ To estimate the coefficients  $w_i$ , we can pre-compute the values  $x_i$  and treat them just like other numerical features - no other changes are required!
- ▶ Minimize again the cost function

$$\frac{1}{2} \sum_{i=1}^m \left( h_{\theta} \left( x^{(i)} \right) - y^{(i)} \right)^2$$

# Polynomial Regression in Python

demo/poly\_reg.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model, preprocessing

# read cars dataset, a clean data set
cars = pd.read_csv('auto-mpg.data.txt',
                    header=None, sep='\s+')

# extract mpg values
y = cars.iloc[:,0].values

# extract horsepower values
X = cars.iloc[:,[3]].values
X = X.reshape(X.size, 1)

# precompute polynomial features
poly = preprocessing.PolynomialFeatures(2)
Xp = poly.fit_transform(X)

# fit linear regression model
reg = linear_model.LinearRegression()
reg.fit(Xp,y)
```

```
# coefficients
reg.intercept_ # 56.900099702112925
reg.coef_ # [-0.46618963, 0.00123054]

# compute correlation coefficient
np.corrcoef(reg.predict(Xp),y) # 0.82919179 (from 0.77842678)

# compute mean squared error (MSE)
sum((reg.predict(Xp) - y)**2) / len(y)
# 18.984768907617223 ( from 23.943662938603104)

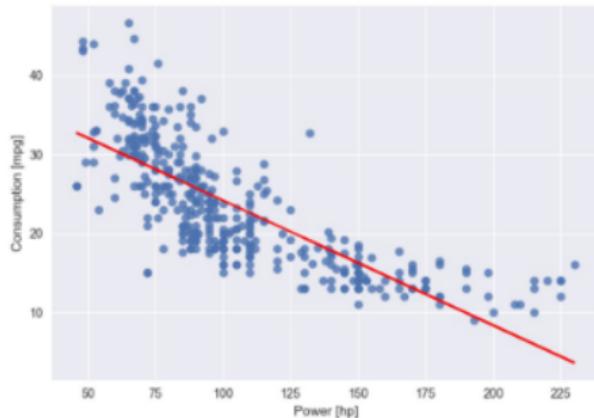
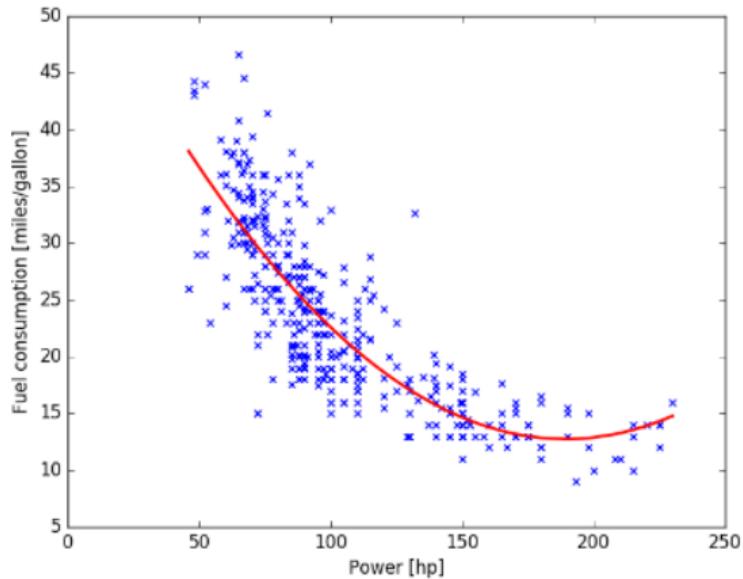
### plot

hp = cars.iloc[:,3].values
mpg = cars.iloc[:,0].values

hps = np.array(sorted(hp))
hps = hps.reshape(hps.size, 1)
hpsp = poly.fit_transform(hps)

plt.scatter(hp, mpg, color='blue', marker='x')
plt.plot(hps, reg.predict(hp), color='red', lw=2)
plt.xlabel('Power [hp]')
plt.ylabel('Fuel consumption [miles/gallon]')
plt.show()
```

# A Much Better Prediction



# Polynomial regression & Probability

See Bishop (2006), Chapter 3 for more details.

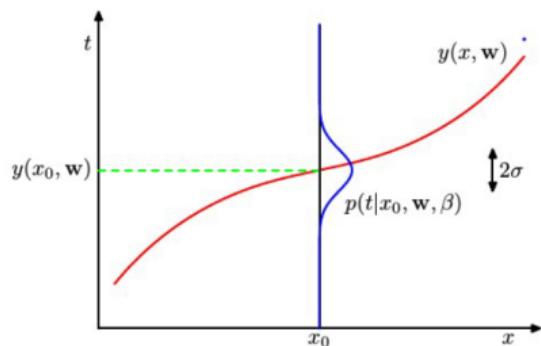
- As before, we assume that the **target variable**  $t$  is given by a deterministic function  $y(x, w)$  with additive Gaussian noise:

$$t = y(\mathbf{x}, \mathbf{w}) + \epsilon$$

where the noise is a zero mean Gaussian random variable with precision (inverse variance)  $\beta$ . Thus we can write

$$p(t | \mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t | y(\mathbf{x}, \mathbf{w}), \beta^{-1}).$$

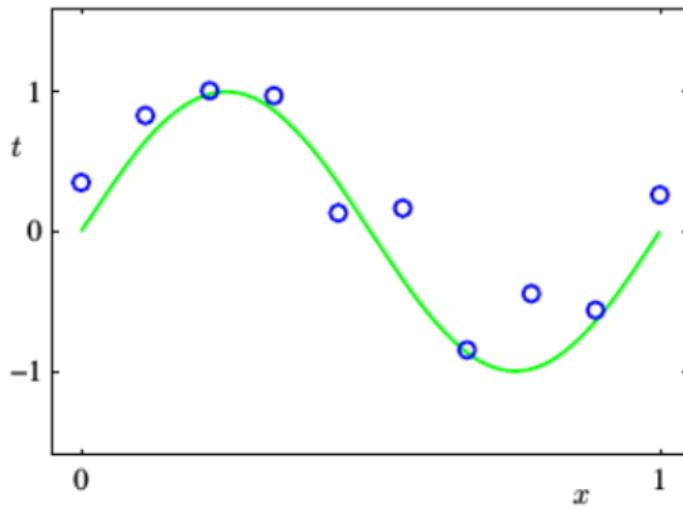
- → to determine the parameters, **maximize again the likelihood**.



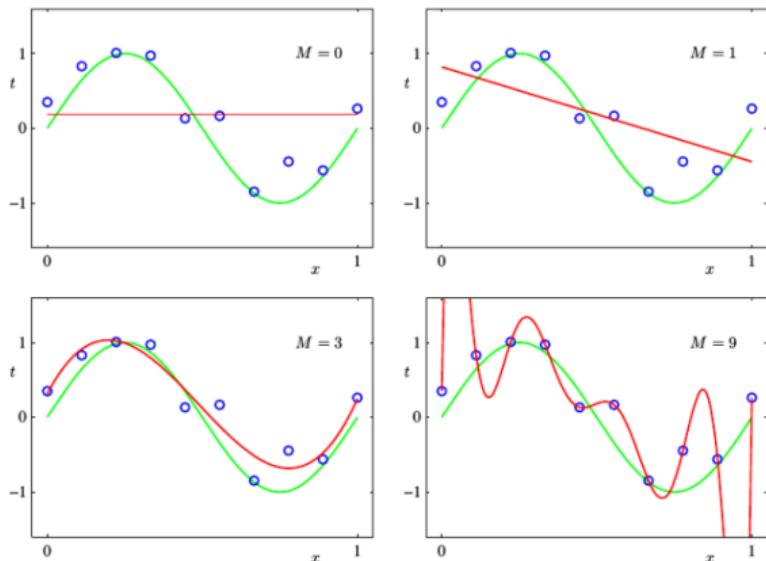
## 7. Tuning Model Complexity

- ▶ Plot of a training data set of  $N = 10$  points, shown as blue circles, each comprising an observation of the input variable  $x$  along with the corresponding target variable  $t$ .
- ▶ The green curve shows the function  $\sin(2\pi x)$  used to generate the data.
- ▶ → Our goal is to predict the value of  $t$  for some new value of  $x$ , without knowledge of the green curve.

See Bishop (2006), Chapter 1 and 3 for more details.



# Overfitting



Polynomial Curve Fitting of polynomials having various orders  $M$ , shown as red curves, fitted to the data set shown above.

# Overfitting (II)

- So far, we've assessed the prediction **quality of our model based on the same data that we used for training.**
- This is **a very bad idea**, since we can not accurately**measure how well our model works for previously unseen data points** (e.g., for cars not in our dataset)
- Our model may over-fit to the training data and lose its ability to make predictions.

→ Next, we'll see some best practices for evaluating machine learning models

# Overfitting (III)

- ▶ Overfitting occurs when our model describes the training data very accurately, but fails to make predictions for previously unseen data points.
- ▶ When the number of features is large in comparison to the number of data points available for training, over-fitting is likely to occur.
- ▶ In that case, we learn a model that uses many features, and is thus more complex, but fails to generalize.

# Occam's Razor

- ▶ Occam's razor is a logical principle attributed to the medieval philosopher **William of Occam (or Ockham)**.
- ▶ The principle states that **one should not make more assumptions than the minimum needed**. This principle is often called the **principle of parsimony**.
- ▶ It underlies all scientific modeling and theory building. **It admonishes us to choose from a set of otherwise equivalent models of a given phenomenon the simplest one.**
- ▶ In any given model, Occam's razor helps us to “**shave off**” those concepts, variables or constructs that are not really needed to explain the phenomenon.
- ▶ By doing that, developing the model will become much easier, and there is less chance of introducing inconsistencies, ambiguities and redundancies.

# How to Avoid Overfitting

- ▶ To avoid over-fitting, it is good practice to **assess the quality of a model based on test data that must not be used** for training the model.
- ▶ The key idea is to split the available data (randomly) into **training**, **validation**, and **test data**.

# Splitting the Data

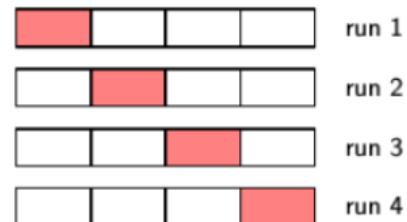
One common approach to reliably assess the quality of a machine learning model and avoid over-fitting is to randomly split the available data into

- ▶ **training data (~ 70% of the data)** used for determining optimal coefficients.
- ▶ **validation data ( 20% of the data) used for model selection** (e.g., fixing degree of polynomial, selecting a subset of features, etc.)
- ▶ **test data ( 10% of the data)** used to measure the quality that is reported.

# Model Selection: k-Fold Cross Validation

`demo/k-fold_cross_validation.py`

- ▶ Another common approach, especially suitable when only limited data is available, is **k-fold cross-validation.**
- ▶ Data is (randomly) split into  $k$  folds of (about) equal size.
- ▶  **$k$  rounds of training and validation** are performed, in which
  - ▶  $(k-1)$  folds serve as training data
  - ▶ one fold serves as validation/test data



In the end the mean of the quality measure (e.g., MSE) is reported as an estimate of the overall quality.

# Cross-validation in Python

demo/k-fold\_cross\_validation.py

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import linear_model,
metrics, preprocessing, metrics, model_selection

### Clean data set
cars = pd.read_csv('auto-mpg.data.txt', header=None, sep='\s+')

### Label columns
cars.columns = ['mpg', 'cylinders',
                'displacement', 'horsepower',
                'weight', 'acceleration', 'model year',
                'origin', 'car name']
print (cars.head())

### Some scatter plots
cols = ['mpg', 'horsepower', 'weight', 'acceleration']
sns.pairplot(cars[cols], size=2.5)
# plt.show()

### extract the fuel consumption
y = cars.iloc[:,0].values

### horsepower
X = cars.iloc[:,[3]].values
# Compute Polynomial Features (e.g., horsepower^2)
poly = preprocessing.PolynomialFeatures(2)
X = poly.fit_transform(X)
```

```
#5-fold Cross-validation
kf = model_selection.KFold(n_splits=5, shuffle=True)
mses = []
for train_index, test_index in kf.split(X):

    #Split into training and test data
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    #Linear regression
    reg = linear_model.LinearRegression()
    reg.fit(X_train,y_train)

    #Print Parameters
    print("Parameter: ")
    print('w0: %f' %reg.intercept_)
    print('w1: %f' %reg.coef_[0])
    print('w2: %f' %reg.coef_[1])

    # MSE
    mse = sum((y_test - reg.predict(X_test))**2.0)/len(y_test)
    print("MSE: %f" %mse)
    mses.append(mse)

print("MSE (Average): %f" %(sum(mses)/len(mses)))
```

# Regularization — Ridge Regression

One technique that is often used to **control the over-fitting** phenomenon in such cases is that of **regularization**, which involves **adding a penalty term to the error function** in order to discourage the coefficients from reaching large values.

The simplest such penalty term takes the form of a sum of squares of all of the coefficients, leading to a modified error function of the form

$$\frac{1}{2} \sum_{n=1}^N \{y(x_n, w) - t_n\}^2 + \frac{\lambda}{2} \|w\|^2 \quad \text{Ridge Regression}$$

where  $\|w\|^2 \equiv w^T w = w_0^2 + w_1^2 + \dots + w_M^2$  and the coefficient  $\lambda$  governs the relative importance of the regularization term compared with the sum-of-squares error term.

# Ridge Regression in Python

demo/ridge\_regression.py

We consider our car dataset and learn coefficients for polynomial ridge regression with degree 5 based on a subset of 10 randomly chosen cars.

```
import pandas as pd
import numpy as np
from sklearn import linear_model, preprocessing
import matplotlib.pyplot as plt
import random

# read cars dataset, a clean data set
cars = pd.read_csv('auto-mpg.data.txt', header=None, sep='\s+')

# random sample of 20 cars
sample = random.sample(range(0, len(cars)), 10)
out_of_sample = list(set(range(0, len(cars))) - set(sample))

# extract mpg values for cars in sample
y = cars.iloc[sample, 0].values
y_oos = cars.iloc[out_of_sample, 0].values

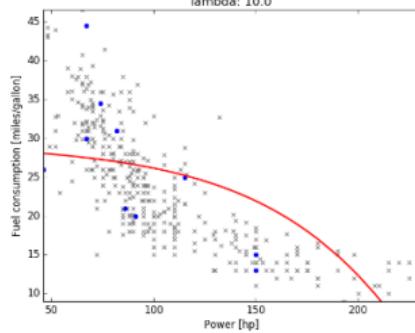
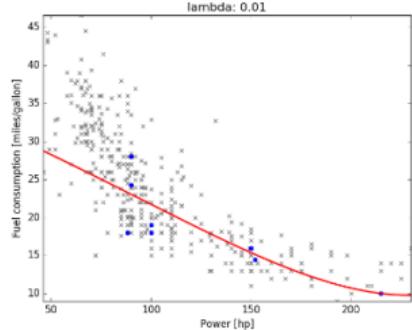
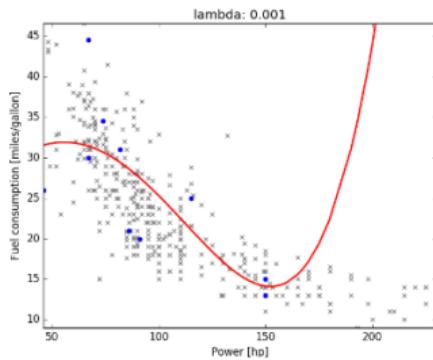
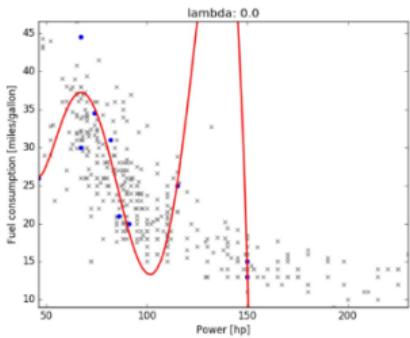
# extract horsepower values for cars in sample
X = cars.iloc[sample, [3]].values
X.reshape(X.size, 1)

# precompute polynomial features for degree 5
poly = preprocessing.PolynomialFeatures(5)
Xp = poly.fit_transform(X)
```

# Cont.

```
for lmbd in [0.0, 0.0001, 0.0001, 0.001, 0.01, 0.1, 1.0, 10.0]:  
    # fit linear regression model  
    reg = linear_model.Ridge(alpha=lmbd, normalize=True)  
    reg.fit(Xp,y)  
    # plot fitted function  
    hp = cars.iloc[:,3].values  
    mpg = cars.iloc[:,0].values  
    hps = np.array(sorted(hp))  
    hps = hps.reshape(hps.size, 1)  
    hpsp = poly.fit_transform(hps)  
    plt.title("lambda: " + str(lmbd))  
    plt.scatter(hp, mpg, color='gray', marker='x')  
    plt.scatter(X, y, color='blue', marker='o')  
    plt.plot(hps, reg.predict(hp), color='red', lw=2)  
    plt.xlabel('Power [hp]')  
    plt.ylabel('Fuel consumption [miles/gallon]')  
    plt.xlim([min(hp), max(hp)])  
    plt.ylim([min(mpg), max(mpg)])  
    plt.show()
```

# Ridge Regression in Python

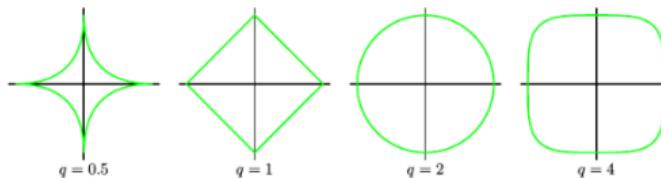


# Regularization in General

- A more general regularizer is sometimes used, for which the regularized error takes the form

$$\frac{1}{2} \sum_{n=1}^N \{ t_n - w^T \phi(\mathbf{x}_n) \}^2 + \frac{\lambda}{2} \sum_{j=1}^M |w_j|^q$$

- The case of  $q = 1$  is known as the LASSO in the statistics literature. It has the property that if  $\lambda$  is sufficiently large, some of the coefficients  $w_j$  are driven to zero, leading to a sparse model in which the corresponding basis functions play no role.



Contours of the regularization term in the equation for various values of the parameter  $q$ .

# Some Literature for this lecture

***Python Machine Learning***

S. Raschka

PACKT Publishing, 2017

***Pattern Recognition and Machine Learning***

C.M. Bishop

Springer, 2006

