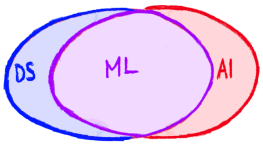
Introduction

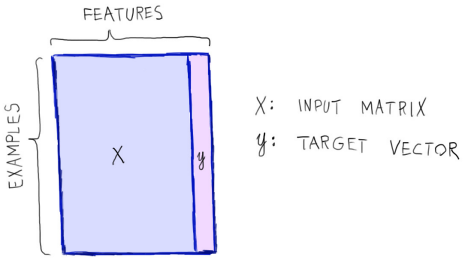
Machine learning (ML), data science (DS) and artificial intelligence (AI) are closely related disciplines.

- Machine learning (https://en.wikipedia.org/wiki/Machine_learning) focuses on creating algorithms that allow computers to learn.
- Artificial intelligence (https://en.wikipedia.org/wiki/Artificial_intelligence), encompasses ML and other technologies (e.g. heuristic search algorithms, robotics) aimed at creating intelligent machines.
- Data science (https://en.wikipedia.org/wiki/Data_science) is a broader field that uses ML and other techniques (e.g. data cleaning, data management) to analyze data and make informed decisions.

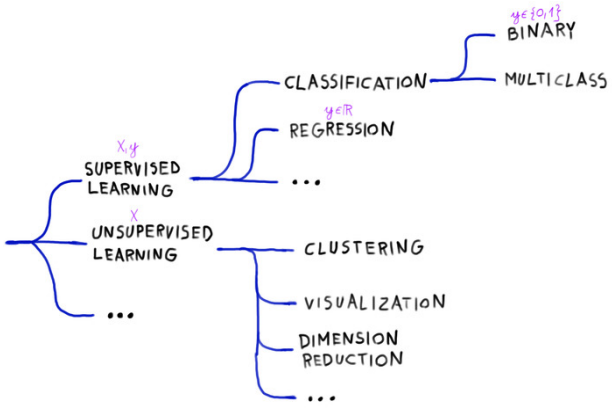


A down-to-earth definition of machine learning

- Imagine a table filled with numbers!
- Machine learning is discovering the relationship between columns of the table.
  - In supervised learning (https://en.wikipedia.org/wiki/Supervised_learning), there is a target column that we aim to estimate based on the other columns.
  - In unsupervised learning (https://en.wikipedia.org/wiki/Unsupervised_learning), there is no designated target column.



A taxonomy of machine learning problems



- Some examples of binary classification problems:
  - Predicting whether an email is spam or not, based on its content, sender, subject, etc.
  - Predicting whether a patient has a certain disease based on their symptoms, test results, medical history, etc.

Traditional machine learning vs. deep neural networks

Machine learning methods can be broadly categorized into two groups: traditional machine learning and deep neural networks. Traditional machine learning includes linear regression (https://en.wikipedia.org/wiki/Linear_regression), logistic regression (https://en.wikipedia.org/wiki/Logistic_regression), support vector machines (https://en.wikipedia.org/wiki/Support_vector_machine), decision trees (https://en.wikipedia.org/wiki/Decision_tree), random forests (https://en.wikipedia.org/wiki/Random_forest), and gradient boosting (https://en.wikipedia.org/wiki/Gradient_boosting). These methods are known for their simplicity, interpretability, and computational efficiency.

On the other hand, deep neural networks (https://en.wikipedia.org/wiki/Deep_learning) are composed of multiple layers of artificial neurons that are trained to learn complex representations of the data. They include convolutional neural networks (https://en.wikipedia.org/wiki/Convolutional_neural_network), recurrent neural networks (https://en.wikipedia.org/wiki/Recurrent_neural_network), generative adversarial networks

(https://en.wikipedia.org/wiki/Generative_adversarial_network), and transformers (https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)). These models have the ability to capture complex patterns in the data. However, deep neural networks are computationally more intensive to train and less interpretable compared to traditional machine learning methods. Additionally, they require larger amounts of training data to achieve optimal performance.

In this semester's Machine Learning course, we will concentrate exclusively on traditional machine learning techniques. For those seeking further knowledge on deep neural networks, the Neural Networks course offered in the following semester may be of interest.

Some machine learning terms

- feature: one column of the data table,
- example: one row of the data table,
- training: finding the model parameters that fit well to the data,
- prediction: estimating the target value of an input vector, using a trained model,
- training set: a subset of examples that is used for training,
- test set: another, distinct set of examples that is used for evaluating the trained model.

Some Moments from the History of Machine Learning

The Perceptron Algorithm (https://en.wikipedia.org/wiki/Perceptron): In the **1950s**, a psychologist named Frank Rosenblatt (https://en.wikipedia.org/wiki/Frank_Rosenblatt) proposed the Perceptron, a simple artificial neuron model that could solve linear classification problems. Although it was limited in its ability to solve more complex problems, the Perceptron was the first step towards creating more sophisticated machine learning algorithms.

The Dartmouth Workshop (https://en.wikipedia.org/wiki/Dartmouth_workshop): In **1956**, a group of researchers including John McCarthy (https://en.wikipedia.org/wiki/John_McCarthy_(computer_scientist)), Marvin Minsky (https://en.wikipedia.org/wiki/Marvin_Minsky), Nathaniel Rochester (https://en.wikipedia.org/wiki/Nathaniel_Rochester_(computer_scientist)) and Claude Shannon (https://en.wikipedia.org/wiki/Claude_Shannon) gathered at Dartmouth College to discuss the possibilities of thinking machines. This conference is now considered to be the birthplace of the field of artificial intelligence.

The Backpropagation Algorithm (https://en.wikipedia.org/wiki/Backpropagation): In the **1980s**, Geoffrey Hinton (https://en.wikipedia.org/wiki/Geoffrey_Hinton) and his students developed the backpropagation algorithm, which allowed neural networks to learn from their mistakes and improve over time. This breakthrough revolutionized the field of machine learning and paved the way for deep learning, a type of neural network with many layers.

The MNIST Handwritten Digits Dataset (https://en.wikipedia.org/wiki/MNIST_database): In the late **1990s**, researchers at the National Institute of Standards and Technology created a dataset of handwritten digits to use as a benchmark for evaluating machine learning algorithms. This dataset, called MNIST, is still widely used today and has played a major role in advancing the field of computer vision.

The Netflix Prize (https://en.wikipedia.org/wiki/Netflix_Prize): In **2006**, Netflix launched a competition to improve its movie recommendation system. The competition was called the Netflix Prize and offered a million-dollar prize to the team that could produce the best algorithm for predicting which movies a user would like. The winning team, BellKor's Pragmatic Chaos, used a combination of matrix factorization and collaborative filtering to improve the accuracy of Netflix's recommendations.

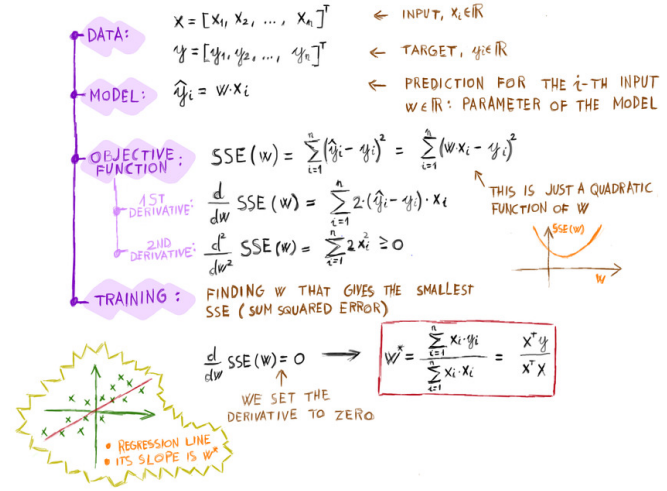
The Deep Learning Revolution (https://en.wikipedia.org/wiki/Deep_learning#Deep_learning_revolution): In **2012**, Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton won the ImageNet image classification challenge by a large margin, using the deep convolutional neural network. In the following years, researchers outperformed state-of-the-art results on several benchmark datasets using deep learning. Deep learning has since become one of the most active areas of research in the field of machine learning.

AlphaGo (https://en.wikipedia.org/wiki/AlphaGo): In **2016**, a machine learning algorithm developed by Google DeepMind, called AlphaGo, defeated Lee Sedol, a 9-dan professional of the game Go, in a best-of-five series. This marked a significant milestone in the development of artificial intelligence and showed that machines could outperform humans in complex, strategic games.

The OpenAI GPT Models (https://en.wikipedia.org/wiki/OpenAI#GPT): In **2018**, OpenAI introduced the GPT (Generative Pretrained Transformer) language models, which were trained on a massive amount of text data and could generate human-like text. The GPT models have been widely adopted and have led to advances in areas such as natural language processing and conversational AI.

Univariate Linear Regression (https://en.wikipedia.org/wiki/Simple_linear_regression)

Linear regression is a simple but still useful learning algorithm that dates back to the 19th century. It was first described by Galton in the 1880s and was later formalized by Fisher in the early 20th century. Despite its age, linear regression remains popular in numerous fields with its implementation constantly evolving in response to technological advancements.



Exercise 1: Perform univariate linear regression on the MLB baseball player data in the baseball.txt (../_data/baseball.txt) file, using height as the input and weight as the target!

In [3]:

```
# Load data.
import numpy as np
data = np.loadtxt("baseball.txt", delimiter=',')
```

In [4]:

```
data
```

Out[4]:

```
array([[188.,  82.],
       [188.,  98.],
       [183.,  95.],
       ...,
       [190.,  93.],
       [190.,  86.],
       [185.,  88.]])
```

In [5]:

```
data.shape
```

Out[5]:

```
(1033, 2)
```

In [6]:

```
# Extract input vector (height) and target vector (weight).
x = data[:,0]
y = data[:,1]
```

In [8]:

```
# Subtract mean.
xm = x.mean()
ym = y.mean()
```

In [9]:

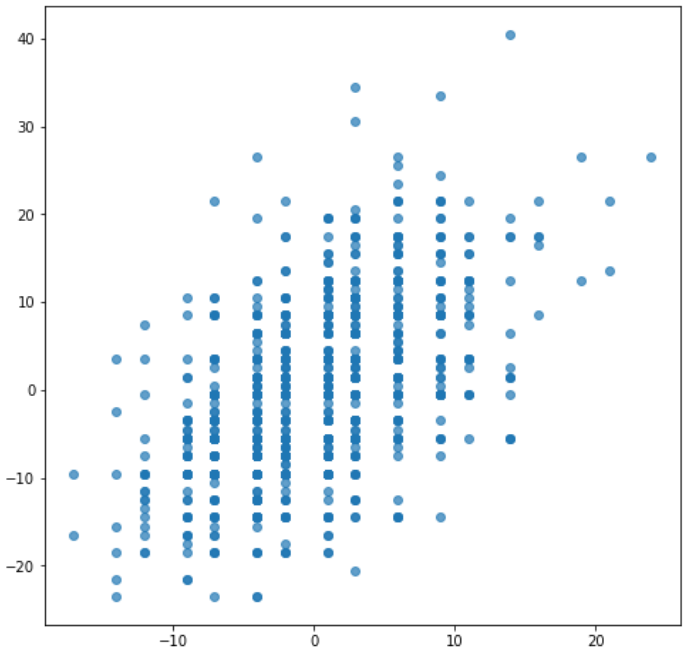
```
x -= xm
y -= ym
```

In [10]:

```
# Prepare a scatter plot of the data.
import matplotlib.pyplot as plt
plt.figure(figsize=(8,8))
plt.scatter(x,y,alpha=0.7)
```

Out[10]:

```
<matplotlib.collections.PathCollection at 0x7f438c1b2580>
```



In [11]:

```
# Train model (compute the optimal w value).
w = (x@y)/(x@x)
w
```

Out[11]:

```
0.8561919786085516
```

In [12]:

```
# Predict the weight of a player who is 190 cm tall.
(190 - xm)*w+ym
```

Out[12]:

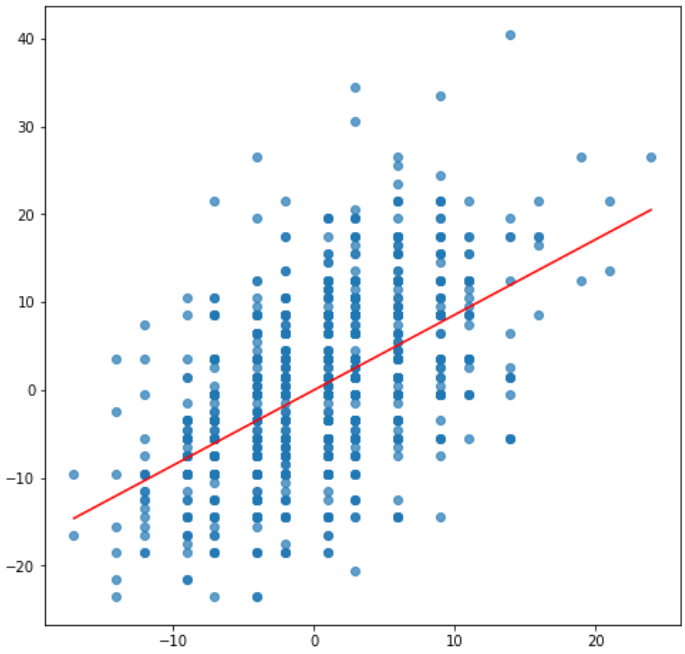
```
94.01062994703861
```

In [15]:

```
# Draw regression line into the scatter plot.
plt.figure(figsize=(8,8))
plt.scatter(x,y,alpha=0.7)
x2 = np.array([x.min(),x.max()])
plt.plot(x2,x2*w, color='red')
```

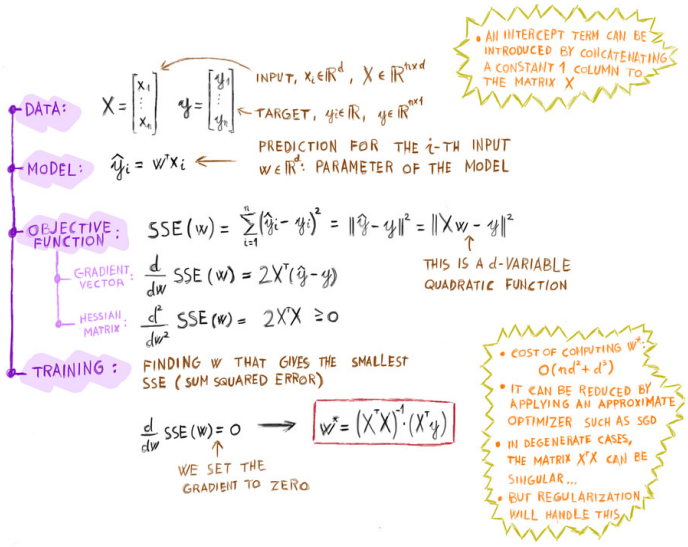
Out[15]:

```
[<matplotlib.lines.Line2D at 0x7f4389f56490>]
```



Multivariate Linear Regression (https://en.wikipedia.org/wiki/Linear_regression)

Multivariate linear regression can be derived from univariate linear regression by adding additional input features to the model. The equation for multivariate linear regression is a extension of the equation for univariate linear regression, with multiple coefficients representing the strength and direction of the relationship between each input feature and the target feature.



Exercise 2: Implement multivariate linear regression on the following artificial problem!

In [16]:

```
import numpy as np

def gen_data(n, d):
    rs = np.random.RandomState(42)
    X = rs.random(size=(n, d)) # input matrix
    v = np.arange(d)  # v = [0, 1, 2, ...]
    y = X @ v + rs.randn(n)      # target vector
    return X, y
```

In [17]:

```
# Generate artificial data set.
X, y = gen_data(1000, 50)
```

In [18]:

```
X.shape
```

Out[18]:

```
(1000, 50)
```

In [21]:

```
# Compute optimal parameter vector.
w = np.linalg.inv(X.T@X) @ (X.T@y)
w
```

Out[21]:

```
array([-6.27668535e-03,  7.79377042e-01,  2.12007621e+00,  2.91175236e+00,
        3.87036693e+00,  5.10677902e+00,  5.99889354e+00,  6.96317023e+00,
        7.85544458e+00,  9.14518616e+00,  9.91401623e+00,  1.10795175e+01,
        1.18655076e+01,  1.29230253e+01,  1.39493048e+01,  1.50584688e+01,
        1.59630538e+01,  1.70070564e+01,  1.78704537e+01,  1.91984375e+01,
        1.98991713e+01,  2.09711802e+01,  2.20835181e+01,  2.31399674e+01,
        2.40855004e+01,  2.49690066e+01,  2.61665348e+01,  2.69094644e+01,
        2.80304559e+01,  2.89075191e+01,  3.00075384e+01,  3.10977986e+01,
        3.18331446e+01,  3.32259644e+01,  3.40089749e+01,  3.53181296e+01,
        3.59508608e+01,  3.69689455e+01,  3.76996546e+01,  3.89252973e+01,
        4.02508652e+01,  4.11066389e+01,  4.19419945e+01,  4.29471376e+01,
        4.39179113e+01,  4.51380313e+01,  4.60397828e+01,  4.69299566e+01,
        4.78870045e+01,  4.91419825e+01])
```

In [22]:

```
# ...more efficient way
np.linalg.solve(X.T @ X, X.T @ y)
```

Out[22]:

```
array([-6.27668535e-03,  7.79377042e-01,  2.12007621e+00,  2.91175236e+00,
        3.87036693e+00,  5.10677902e+00,  5.99889354e+00,  6.96317023e+00,
        7.85544458e+00,  9.14518616e+00,  9.91401623e+00,  1.10795175e+01,
        1.18655076e+01,  1.29230253e+01,  1.39493048e+01,  1.50584688e+01,
        1.59630538e+01,  1.70070564e+01,  1.78704537e+01,  1.91984375e+01,
        1.98991713e+01,  2.09711802e+01,  2.20835181e+01,  2.31399674e+01,
        2.40855004e+01,  2.49690066e+01,  2.61665348e+01,  2.69094644e+01,
        2.80304559e+01,  2.89075191e+01,  3.00075384e+01,  3.10977986e+01,
        3.18331446e+01,  3.32259644e+01,  3.40089749e+01,  3.53181296e+01,
        3.59508608e+01,  3.69689455e+01,  3.76996546e+01,  3.89252973e+01,
        4.02508652e+01,  4.11066389e+01,  4.19419945e+01,  4.29471376e+01,
        4.39179113e+01,  4.51380313e+01,  4.60397828e+01,  4.69299566e+01,
        4.78870045e+01,  4.91419825e+01])
```

Exercise 3: Plot the time complexity of training the model! \n O(d^2 * n + d^3)

In [24]:

```
def fit_lin_rg(X,y):
    return np.linalg.solve(X.T @ X, X.T @ y)
```

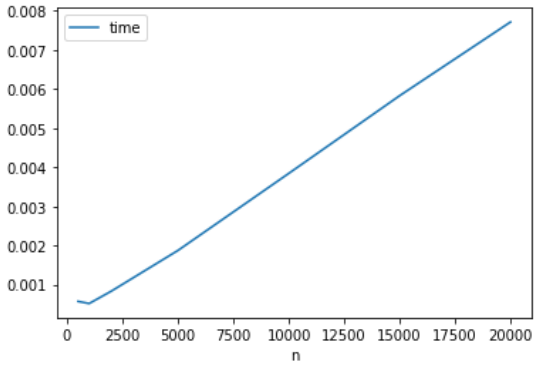
In [36]:

```
import time
import pandas as pd

# How the running time grows with n?
data = []
for n in [500, 1000, 2000, 5000, 10000, 15000, 20000]:
    print(n)
    X, y = gen_data(n,50)
    t0 = time.time()
    fit_lin_rg(X,y)
    t1 = time.time()
    data.append({
        'n':n,
        'time':t1-t0
    })

pd.DataFrame(data).set_index('n').plot()
```

```
500
1000
2000
5000
10000
15000
20000
```

Out[36]:

```
<AxesSubplot: xlabel='n'>
```



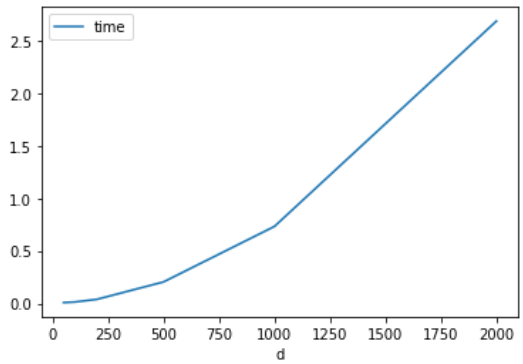
In [38]:

```
# How the running time grows with d?
data = []
for d in [50, 100, 200, 500, 1000, 2000]:
    print(d)
    X, y = gen_data(20000,d)
    t0 = time.time()
    fit_lin_rg(X,y)
    t1 = time.time()
    data.append({
        'd':d,
        'time':t1-t0
    })

pd.DataFrame(data).set_index('d').plot()
```

```
50
100
200
500
1000
2000
```

Out[38]:

```
<AxesSubplot: xlabel='d'>
```



Handling the Bias Term

To include a bias term in d-variate linear regression, one can add an all-ones column to the end of the input matrix, making it $X' = [X \mid 1]$, and append a bias parameter to the end of the parameter vector, $w' = [w \mid \beta]$. This creates a $(d+1)$-variate model is equivalent to a d-variate model with a bias term, because $X'w' = Xw + \beta$

In [39]:

```
# Incorporate a bias term into the previous model.
def gen_data(n, d):
    rs = np.random.RandomState(42)
    X = rs.random(size=(n, d)) # input matrix
    v = np.arange(d)  # v = [0, 1, 2, ...]
    y = X @ v + rs.randn(n)      # target vector
    return X, y
```

In [40]:

```
X, y = gen_data(1000, 50)
```

In [42]:

```
o = np.ones((1000,1))
X_pr = np.hstack([X,o])
w_pr = fit_lin_rg(X_pr,y)
w_pr
```

Out[42]:

```
array([-3.06401464e-03,  7.82222930e-01,  2.12108232e+00,  2.91386326e+00,
        3.87228420e+00,  5.11026964e+00,  6.00067559e+00,  6.96542866e+00,
        7.85673963e+00,  9.14797989e+00,  9.91686530e+00,  1.10816627e+01,
        1.18689718e+01,  1.29260957e+01,  1.39518797e+01,  1.50615225e+01,
        1.59648657e+01,  1.70097186e+01,  1.78728642e+01,  1.92008891e+01,
        1.99009444e+01,  2.09740852e+01,  2.20863362e+01,  2.31414739e+01,
        2.40883621e+01,  2.49715111e+01,  2.61689865e+01,  2.69129379e+01,
        2.80332758e+01,  2.89101312e+01,  3.00091476e+01,  3.11004170e+01,
        3.18356050e+01,  3.32282182e+01,  3.40110034e+01,  3.53203475e+01,
        3.59551113e+01,  3.69712921e+01,  3.77020377e+01,  3.89276599e+01,
        4.02537764e+01,  4.11091654e+01,  4.19436299e+01,  4.29507093e+01,
        4.39199257e+01,  4.51407203e+01,  4.60427084e+01,  4.69329446e+01,
        4.78878303e+01,  4.91438904e+01, -6.19795948e-02])
```

In [43]:

```
w_pr.shape
```

Out[43]:

```
(51,)
```

In [44]:

```
w, beta = w_pr[:50], w_pr[50]
w, beta
```

Out[44]:

```
(array([-3.06401464e-03,  7.82222930e-01,  2.12108232e+00,  2.91386326e+00,
         3.87228420e+00,  5.11026964e+00,  6.00067559e+00,  6.96542866e+00,
         7.85673963e+00,  9.14797989e+00,  9.91686530e+00,  1.10816627e+01,
         1.18689718e+01,  1.29260957e+01,  1.39518797e+01,  1.50615225e+01,
         1.59648657e+01,  1.70097186e+01,  1.78728642e+01,  1.92008891e+01,
         1.99009444e+01,  2.09740852e+01,  2.20863362e+01,  2.31414739e+01,
         2.40883621e+01,  2.49715111e+01,  2.61689865e+01,  2.69129379e+01,
         2.80332758e+01,  2.89101312e+01,  3.00091476e+01,  3.11004170e+01,
         3.18356050e+01,  3.32282182e+01,  3.40110034e+01,  3.53203475e+01,
         3.59551113e+01,  3.69712921e+01,  3.77020377e+01,  3.89276599e+01,
         4.02537764e+01,  4.11091654e+01,  4.19436299e+01,  4.29507093e+01,
         4.39199257e+01,  4.51407203e+01,  4.60427084e+01,  4.69329446e+01,
         4.78878303e+01,  4.91438904e+01]),
 -0.06197959478839162)
```

Linear Regression on Sparse Data

When the input matrix is sparse, meaning it has many zero values, traditional linear regression methods perform poorly. To avoid storage inefficiencies, the input matrix should be represented in a sparse matrix format such as COO, CSR, or CSC. To handle sparse data effectively, alternative training algorithms should be employed that only consider the non-zero locations in the input matrix.

In [1]:

```
import numpy as np
import scipy.sparse as sp

def gen_sparse_data(n, d, density):
    X = sp.random(n, d, density=density, random_state=42) # input matrix
    v = np.arange(d)
    y = X @ v + np.random.RandomState(42).randn(n)   # target vector
    return X, y
```

In [2]:

```
# Generate artificial data set.
X, y = gen_sparse_data(1000, 100, 0.01)
```

In [3]:

```
X
```

Out[3]:

```
<1000x100 sparse matrix of type '<class 'numpy.float64'>'
        with 1000 stored elements in COOrdinate format>
```

In [5]:

```
y
```

Out[5]:

```
array([ 8.31573963e+01, -1.38264301e-01,  6.47688538e-01,  6.95470361e+01,
        8.64440171e+01,  2.21265946e+01,  3.92455292e+01,  7.67434729e-01,
        4.22673463e+00,  5.42560044e-01,  2.18408047e+01,  1.10375192e+01,
        1.03918651e+01,  5.47211613e+01, -1.66547943e+00,  1.97022890e+01,
       -1.01283112e+00,  2.80787047e+01,  6.96266404e-01,  2.23033667e+01,
        1.46564877e+00,  2.30734549e+01,  3.49393826e+01, -1.42474819e+00,
       -5.44382725e-01,  1.10922590e-01,  4.16321594e+01,  3.75698018e-01,
        5.49280248e+00,  3.97282594e+01, -6.01706612e-01,  1.85227818e+00,
       -1.34972247e-02, -1.05771093e+00,  7.60710146e+01,  4.87893739e+00,
        2.08863595e-01,  3.83735573e+01,  8.49340845e+01,  7.63919181e+01,
        9.46626266e+00,  1.71368281e-01,  5.58220928e+01,  6.86964546e+01,
       -1.47852199e+00, -5.28017778e-01,  9.83779634e+00,  1.05712223e+00,
        3.43618290e-01,  2.54551279e-01,  5.83283462e+00, -3.85082280e-01,
        4.78181839e+00,  6.11676289e-01,  2.07980776e+01,  5.18802057e+01,
        4.73186585e+01,  1.97540367e-01,  3.31263431e-01,  9.75545127e-01,
        1.03568355e+01, -1.85658977e-01, -1.10633497e+00, -1.19620662e+00,
        4.12787590e+01,  1.35624003e+00,  5.05158682e+01,  5.77108280e+01,
        3.02452754e+01, -6.45119755e-01,  1.02726922e+02,  5.49761660e+01,
```

In [6]:

```
X.nnz
```

Out[6]:

```
1000
```

In [7]:

```
X.nnz / (X.shape[0] * X.shape[1])
```

Out[7]:

```
0.01
```

In [ ]:

```

```

In [9]:

```
# Compute optimal parameter vector.
n, d = X.shape

A = sp.linalg.LinearOperator((d,d), matvec=lambda v: X.T @ (X @ v))
A
```

Out[9]:

```
<100x100 _CustomLinearOperator with dtype=float64>
```

In [10]:

```
A @ np.random.random(100)
```

Out[10]:

```
array([7.85199981, 6.22284795, 2.66380294, 0.5404676 , 3.43693513,
       2.39290994, 1.93339883, 1.78963086, 6.92342463, 2.17248646,
       0.76164501, 2.29251084, 2.13570559, 4.19386299, 7.59619662,
       1.81771501, 5.30015776, 2.95688449, 2.11766347, 3.6963028 ,
       2.71698559, 2.56057553, 6.94201863, 1.08628271, 0.97217213,
       5.19443921, 1.84488771, 2.45718623, 2.12010589, 4.36884965,
       4.12700319, 6.96280876, 3.65546208, 2.44991314, 6.27362133,
       2.74343828, 6.6689392 , 2.2606863 , 5.36668712, 2.40265729,
       3.12105323, 4.91836955, 2.70967492, 1.58585288, 3.39638596,
       1.60125817, 1.04562329, 0.28799912, 0.9645669 , 4.30179147,
       6.07580287, 2.15517405, 4.61354409, 3.9467374 , 2.08764964,
       2.95232561, 7.61396656, 1.32519149, 1.05778667, 4.90474037,
       4.04342699, 3.58982436, 3.63498181, 3.9022192 , 3.81131248,
       2.98056603, 4.12770351, 2.47645163, 2.76678316, 5.02386035,
       1.9375755 , 0.23763414, 3.70393739, 1.83764665, 1.30396617,
       2.31954938, 2.91415677, 5.6046886 , 4.44518914, 2.29868924,
       4.02140675, 1.62461617, 5.93984636, 4.44226253, 2.43759046,
       5.0823062 , 3.71094808, 1.07423217, 6.88570824, 1.95057564,
       2.25649466, 1.95604282, 3.95986926, 5.00507331, 2.39668844,
       2.29705422, 2.60474113, 2.8725407 , 2.82463373, 0.22405277])
```

In [12]:

```
b = X.T @ y
```

In [15]:

```
w = sp.linalg.cg(A, b)[0]
```

In [16]:

```
w
```

Out[16]:

```
array([4.64339975e-02, 3.26604917e-01, 2.29059762e+00, 2.97799922e+00,
       4.43162378e+00, 4.36446042e+00, 6.21333225e+00, 6.55339954e+00,
       7.05355659e+00, 1.03196805e+01, 1.10572575e+01, 1.15858278e+01,
       1.17330927e+01, 1.33788421e+01, 1.37388481e+01, 1.54351381e+01,
       1.57893995e+01, 1.64644072e+01, 1.72860014e+01, 1.92960255e+01,
       2.07635357e+01, 2.11645421e+01, 2.22330136e+01, 2.42350792e+01,
       2.30280771e+01, 2.53374573e+01, 2.57806693e+01, 2.71485673e+01,
       2.78002859e+01, 2.84697530e+01, 3.10263660e+01, 3.07551333e+01,
       3.24995461e+01, 3.37672839e+01, 3.38745444e+01, 3.50316613e+01,
       3.63488936e+01, 3.82325952e+01, 3.76995083e+01, 3.81634254e+01,
       3.99686573e+01, 4.14135827e+01, 4.15741918e+01, 4.23832713e+01,
       4.33668312e+01, 4.48930318e+01, 4.77707684e+01, 4.46614983e+01,
       4.73798992e+01, 4.90847053e+01, 4.98237101e+01, 5.18169738e+01,
       5.11428677e+01, 5.30883577e+01, 5.32198253e+01, 5.48964684e+01,
       5.54724743e+01, 5.81113882e+01, 5.78044648e+01, 5.82857037e+01,
       6.10110395e+01, 6.17879142e+01, 6.14003995e+01, 6.27920372e+01,
       6.41256972e+01, 6.48996134e+01, 6.45764947e+01, 6.74807125e+01,
       6.83645753e+01, 6.91000515e+01, 7.06680225e+01, 7.14375467e+01,
       7.14460225e+01, 7.27029937e+01, 7.41242247e+01, 7.44094776e+01,
       7.61913445e+01, 7.68914070e+01, 7.76538667e+01, 7.96193506e+01,
       7.97402505e+01, 8.03906231e+01, 8.26255295e+01, 8.37349515e+01,
       8.36416654e+01, 8.50963847e+01, 8.59888530e+01, 8.69106689e+01,
       8.80390935e+01, 8.92648847e+01, 9.14780571e+01, 9.08513364e+01,
       9.08726437e+01, 9.32493484e+01, 9.45125753e+01, 9.46143945e+01,
       9.62206304e+01, 9.69323206e+01, 9.84048975e+01, 1.00099921e+02])
```

In [ ]:

```

```