MATH 4/5388: Machine Learning Methods

Module 2: Parametric Regression Models

Instructor: Farhad Pourkamali

Spring 2023

Textbook: Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow (Chapter 4)

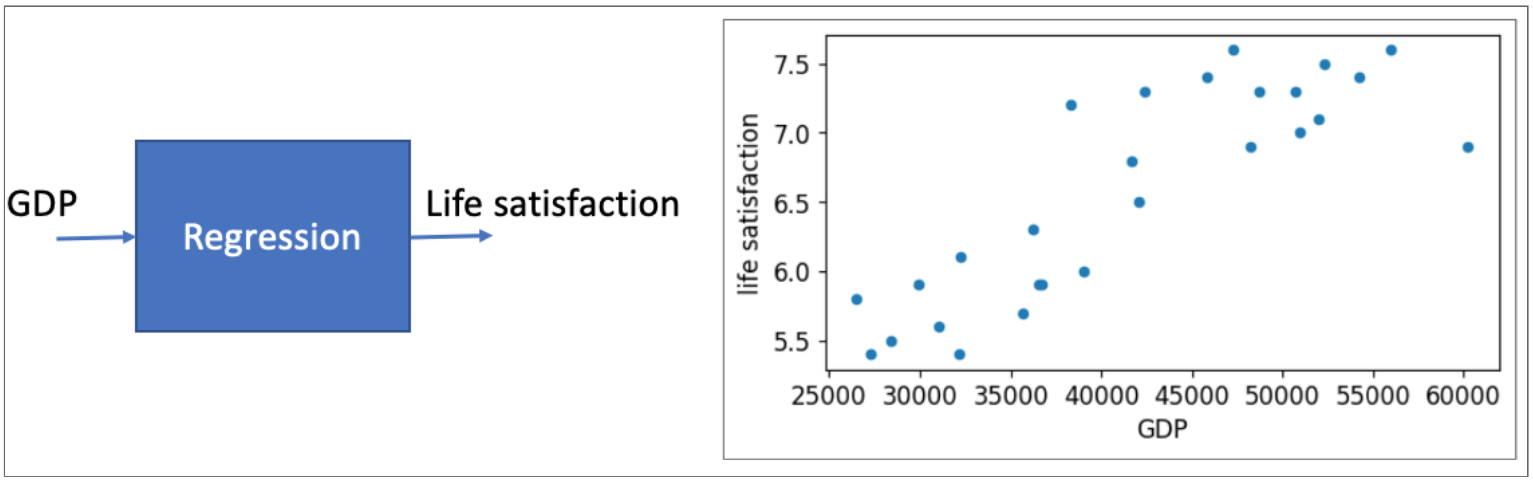
Overview of Module 2

- Linear regression: Problem formulation, assumption, loss function, gradient
    - Normal equation
    - Sklearn implementation
    - Evaluation metrics
    - Gradient descent (GD) and variants
    - Nonlinear extension and regularization
    - Model selection: The process of selecting the proper level of flexibility

Case study: univariate linear regression

- Training data: $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N}$
- Parametric model: $f(x) = \theta_0 + \theta_1 x$
- Objective: Choose θ_0, θ_1 such that $f(x_n)$ is close to y_n
- Mean squared error (MSE):

$$\mathcal{L}(\theta_0, \theta_1) = \frac{1}{N} \sum_{n=1}^{N} \left(y_n - f(x_n)\right)^2$$



Solving the optimization problem

- We'll need the concept of partial derivatives
- To compute $\partial \mathcal{L} / \partial \theta_0$, take the derivative with respect to θ_0, treating the rest of the arguments as constants
- We can show that

$$\frac{\partial \mathcal{L}}{\partial \theta_0} = \frac{-2}{N} \sum_{n=1}^{N} \left(y_n - \theta_0 - \theta_1 x_n \right) = \frac{2}{N} \sum_{n=1}^{N} \left(f(x_n) - y_n \right)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{-2}{N} \sum_{n=1}^{N} \left(y_n - \theta_0 - \theta_1 x_n \right) x_n = \frac{2}{N} \sum_{n=1}^{N} \left(f(x_n) - y_n \right) x_n$$

Gradient

- Extend the notion of derivatives to handle vector-argument functions
  - Given $\mathcal{L} : \mathbb{R}^d \mapsto \mathbb{R}$, where d is the number of input variables

$$\nabla \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_0} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial \theta_{d-1}} \end{bmatrix} \in \mathbb{R}^d$$

- Example from the previous slide ($d = 2$):

$$\nabla \mathcal{L} = \frac{2}{N} \begin{bmatrix} \sum_{n=1}^{N} \big(f(x_n) - y_n\big) \\ \sum_{n=1}^{N} \big(f(x_n) - y_n\big) x_n \end{bmatrix} \in \mathbb{R}^2$$

Compact form of gradient

- Let us define

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_N \end{bmatrix} \in \mathbb{R}^{N \times 2}, \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} \in \mathbb{R}^2, \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^N$$

- Hence, we get

$$\mathbf{X}\boldsymbol{\theta} - \mathbf{y} = \begin{bmatrix} f(x_1) - y_1 \\ \vdots \\ f(x_N) - y_N \end{bmatrix}$$

Compact form of gradient

- The last step is to show that $\nabla \mathcal{L}$ can be written as

$$\frac{2}{N}\mathbf{X}^T\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right) = \frac{2}{N}\begin{bmatrix} 1 & \dots & 1 \\ x_1 & \dots & x_N \end{bmatrix}\begin{bmatrix} f(x_1) - y_1 \\ \vdots \\ f(x_N) - y_N \end{bmatrix}$$
$$= \frac{2}{N}\begin{bmatrix} \sum_{n=1}^{N}\left(f(x_n) - y_n\right) \\ \sum_{n=1}^{N}\left(f(x_n) - y_n\right)x_n \end{bmatrix}$$

- Given this compact form, we can use NumPy to solve the linear matrix equation

$$\underbrace{\mathbf{X}^T\mathbf{X}}_{a}\boldsymbol{\theta} = \underbrace{\mathbf{X}^T\mathbf{y}}_{b}$$

numpy.linalg.lstsq

`linalg.lstsq(a, b, rcond='warn')` [source]

Return the least-squares solution to a linear matrix equation.

Computes the vector *x* that approximately solves the equation `a @ x = b`. The equation may be under-, well-, or over-determined (i.e., the number of linearly independent rows of *a* can be less than,

In [1]:

```
# GDP data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("https://github.com/ageron/data/raw/main/lifesat/lifesat.csv")

df.head()
```

Out[1]:

	Country	GDP per capita (USD)	Life satisfaction
0	Russia	26456.387938	5.8
1	Greece	27287.083401	5.4
2	Turkey	28384.987785	5.5
3	Latvia	29932.493910	5.9
4	Hungary	31007.768407	5.6

In [2]:

```
X = df['GDP per capita (USD)'].to_numpy()
y = df['Life satisfaction'].to_numpy()

print(X.shape, y.shape)
```

```
(27,) (27,)
```

In [3]:

```
# add the column of all 1's

def add_column(X):
    '''
    add the column of all 1's
    '''
    return np.concatenate(( np.ones((X.shape[0],1)), X.reshape(-1,1)), axis=1)

Xcon = add_column(X)

Xcon.shape
```

Out[3]:

```
(27, 2)
```

In [4]:

```
            # solve the problem

a = np.matmul(np.transpose(Xcon), Xcon)

b = np.matmul(np.transpose(Xcon), y)

theta = np.linalg.lstsq(a, b, rcond=None)[0] # Cut-off ratio for small singular values

print(theta)
```

```
[3.74904943e+00 6.77889970e-05]
```

In [ ]:

```
            # plot the prediction model f

def f(X, theta):
    return np.matmul(X, theta)

plt.rcParams.update({'font.size': 12, "figure.figsize": (6,3)})
plt.scatter(X, y, s=20, label='training data')

X_test = np.array([25000, 60000])
y_test = f(add_column(X_test), theta)
plt.plot(X_test, y_test, 'r--', label='prediction')

plt.legend()
plt.xlabel('GDP')
plt.ylabel('life satisfaction')
plt.show()
```

Linear models for regression

- Given the training data set $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^N$ and an input vector $\mathbf{x} \in \mathbb{R}^D$, the linear regression model takes the form

$$f(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_D x_D = \theta_0 + \boldsymbol{\theta}^T \mathbf{x}$$

- $\boldsymbol{\theta} \in \mathbb{R}^D$: weights or regression coefficients, θ_0: intercept or bias term
- Compact representation by defining $\mathbf{x} = [x_0 = 1, x_1, \ldots, x_D]$ and $\boldsymbol{\theta} = [\theta_0, \theta_1, \ldots, \theta_D]$ in $\mathbb{R}^{D+1}$

$$f(\mathbf{x}) = \boldsymbol{\theta}^T \mathbf{x} = \langle \boldsymbol{\theta}, \mathbf{x} \rangle$$

Loss function for linear regression

- MSE loss function for a linear regression model

$$\mathcal{L}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}\left(y_n - \langle \boldsymbol{\theta}, \mathbf{x}_n \rangle\right)^2 = \frac{1}{N}\|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2$$

where we have

$$\mathbf{X} = \begin{bmatrix} \\ \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \\ \\ \end{bmatrix} \in \mathbb{R}^{N\times(D+1)}, \boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_D \end{bmatrix} \in \mathbb{R}^{D+1}, \ \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} \in \mathbb{R}^{N},$$

- Optimization problem for model fitting/training: $\underset{\boldsymbol{\theta}\in\mathbb{R}^{D+1}}{\operatorname{argmin}}\, \mathcal{L}(\boldsymbol{\theta})$

The Normal equation

- To find the value of $\boldsymbol{\theta}$ that minimizes the MSE, there exists a *closed-form* solution
    - a mathematical equation that gives the result directly
- The gradient takes the form

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N} \mathbf{X}^T \left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y} \right)$$

- Normal equation

$$\boldsymbol{\theta}^* = \left(\mathbf{X}^T \mathbf{X} \right)^{-1} \mathbf{X}^T \mathbf{y}$$

Sklearn implementation

- Documentation page: **https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegress**
    - Parameters: Useful for creating objects
    - Attributes: Estimated coefficients, etc.
    - Methods: Training, prediction, etc.

sklearn.linear_model.LinearRegression

class sklearn.linear_model.**LinearRegression**(*, *fit_intercept=True*, *copy_X=True*, *n_jobs=None*, *positive=False*) [source]

Ordinary least squares Linear Regression.

LinearRegression fits a linear model with coefficients w = (w1, ..., wp) to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation.

In [ ]:

```
# Revisit the GDP data

from sklearn.linear_model import LinearRegression

reg = LinearRegression()

reg.fit(X.reshape(-1,1), y) # X should be a 2D array

print(reg.intercept_, reg.coef_)
```

Preprocessing data

- Change raw/original feature vectors into a representation that is more suitable for the downstream estimators/tasks
- The *sklearn.preprocessing* package provides several common utility functions
    - Scaling features to lie between a given minimum and maximum value
    - Removing the mean value and dividing features by their standard deviation
        - features look like standard normally distributed data
- We use the first technique in the next slide

In [ ]:

```
# Revisit the GDP data

from sklearn.preprocessing import MinMaxScaler

minmax = MinMaxScaler()

X_minmax = minmax.fit_transform(X.reshape(-1,1))

reg = LinearRegression()

reg.fit(X_minmax, y) # X should be a 2D array

print(reg.intercept_, reg.coef_)
```

In [ ]:

```
            # Plot the prediction model

plt.rcParams.update({'font.size': 12, "figure.figsize": (6,3)})
plt.scatter(X_minmax, y, s=20, label='training data')

X_test = np.array([25000, 60000]).reshape(-1,1)
X_test_minmax = minmax.transform(X_test)
plt.plot(X_test_minmax, reg.predict(X_test_minmax), 'r--', label='prediction')

plt.legend()
plt.xlabel('GDP (scaled)')
plt.ylabel('life satisfaction')
plt.show()
```

Evaluation metrics for regression problems

- The quality of a regression model can be assessed using various quantities
  - See **https://scikit-learn.org/stable/modules/model_evaluation.html**
- Mean squared error

$$\mathrm{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N_{\text{test}}} \sum_{n=1}^{N_{\text{test}}} \left(y_n - \hat{y}_n\right)^2$$

  - The value you get after calculating MSE is a squared unit of output
  - If you have outliers in the data set, then it penalizes the outliers most

Root Mean squared error (RMSE)

- The output value you get is in the same unit as the required output variable

$$\mathrm{RMSE}(\mathbf{y}, \hat{\mathbf{y}}) = \sqrt{\frac{1}{N_{\text{test}}} \sum_{n=1}^{N_{\text{test}}} (y_n - \hat{y}_n)^2}$$

In [ ]:

```
            from sklearn.metrics import mean_squared_error
y_true = [3, -1, 2, 7]
y_pred = [3, 0, 2, 7]

# If True returns MSE value, if False returns RMSE value.
print('MSE: ', mean_squared_error(y_true, y_pred),
      ', RMSE: ', mean_squared_error(y_true, y_pred, squared=False))
```

R^2 score or the coefficient of determination

- Definition

$$R^2(\mathbf{y}, \hat{\mathbf{y}}) = 1 - \frac{\sum_{n=1}^{N_{\text{test}}} (y_n - \hat{y}_n)^2}{\sum_{n=1}^{N_{\text{test}}} (y_n - \bar{y})^2}, \quad \bar{y} = \frac{1}{N_{\text{test}}} \sum_{n=1}^{N_{\text{test}}} y_n$$

- RSS (Residual Sum of Squares) measures the amount of variability that is left unexplained

$$\text{RSS} = \sum_{n=1}^{N_{\text{test}}} (y_n - \hat{y}_n)^2$$

- Best possible score is 1 and a number near 0 indicates the model does not explain much of the variability in the response

Explained variance score

- Definition

$$explained\_variance(\mathbf{y}, \hat{\mathbf{y}}) = 1 - \frac{Var\{\mathbf{y} - \hat{\mathbf{y}}\}}{Var\{\mathbf{y}\}}$$

- The best possible score is 1.0, lower values are worse
- When the prediction residuals have zero mean, the R^2 score and the Explained variance score are identical

In [ ]:

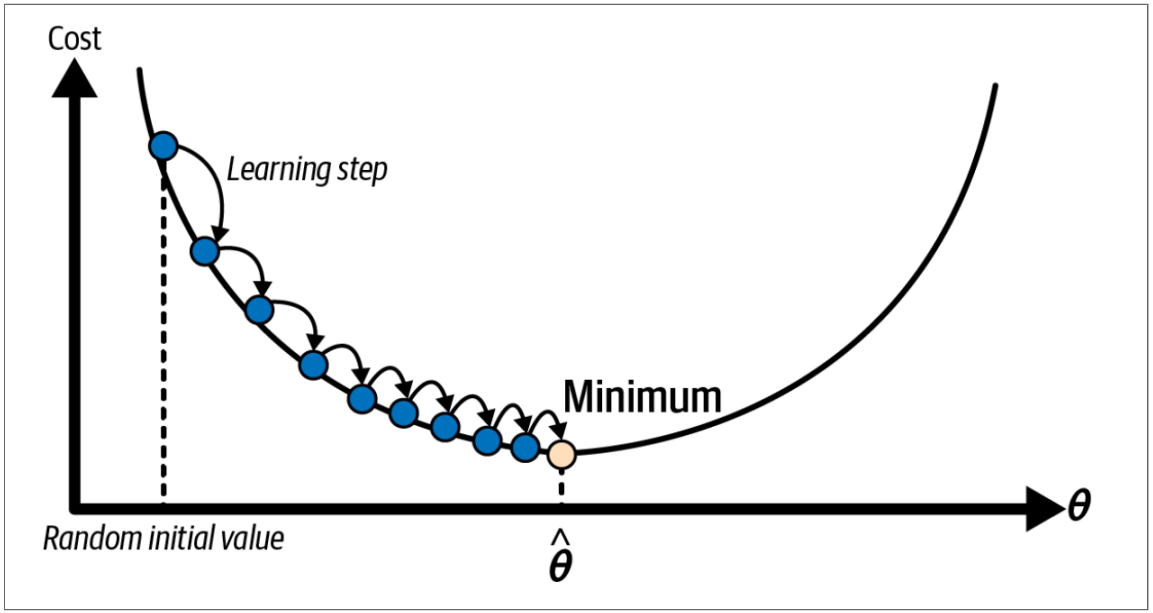
```
from sklearn.metrics import r2_score, explained_variance_score
y_true = [3, -1, 2, 7]
y_pred = [2.9, 0, 2.5, 6.5]
r2_score(y_true, y_pred), explained_variance_score(y_true, y_pred)
```

Gradient descent (GD)

- Tweak parameters $\boldsymbol{\theta}$ iteratively to minimize the loss function $\mathcal{L}(\boldsymbol{\theta})$
- At each iteration t, perform an update to decrease the loss function

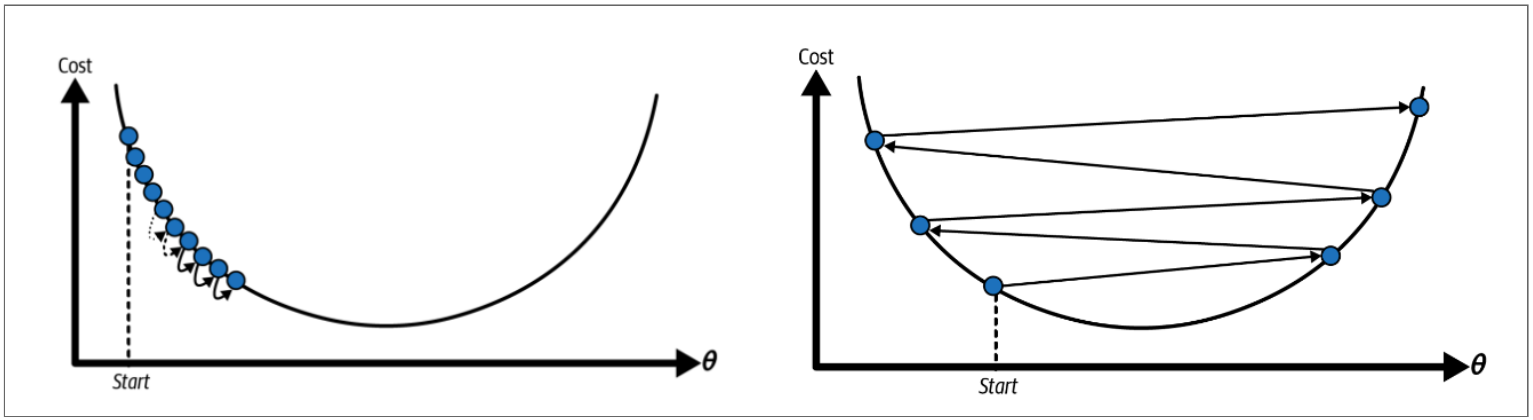
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \nabla \mathcal{L}(\boldsymbol{\theta}_t)$$

where η_t is the step size or learning rate



Learning rate hyperparameter

- A hyperparameter is a parameter that is set *before* the learning process begins
    - These parameters are tunable and directly affect how well a model trains
- If the learning rate is too small, then the algorithm will have to go through many iterations to converge
- The algorithm may diverge when the learning rate is too high



Batch GD for linear regression

- Recall the gradient vector of the loss function

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

- GD step with fixed learning rate

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \mathcal{L}(\boldsymbol{\theta}_t) = \boldsymbol{\theta}_t - \eta \frac{2}{N}\mathbf{X}^T(\mathbf{X}\boldsymbol{\theta}_t - \mathbf{y})$$

  - This formula involves calculations over the full training set $\mathbf{X}$ --> batch or full GD
  - An epoch means one complete pass of the training data set

In [ ]:

```
            # GDP data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("https://github.com/ageron/data/raw/main/lifesat/lifesat.csv")

X = df['GDP per capita (USD)'].to_numpy()
y = df['Life satisfaction'].to_numpy()

def add_column(X):
    '''
    add the column of all 1's
    '''
    return np.concatenate(( np.ones((X.shape[0],1)), X.reshape(-1,1)), axis=1)
```

In [ ]:

```
from sklearn.preprocessing import MinMaxScaler

minmax = MinMaxScaler()

X_minmax = minmax.fit_transform(X.reshape(-1,1))

Xcon = add_column(X_minmax)

print(Xcon[:3])
```

In [ ]:

```
            # Implementation of Batch GD

eta = 0.01  # learning rate
n_epochs = 1000
N = len(Xcon)  # number of instances

np.random.seed(3)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

for epoch in range(n_epochs):
    gradients = 2 / N * Xcon.T @ (Xcon @ theta - y.reshape(-1,1))
    theta = theta - eta * gradients

print(theta)
```

In [ ]:

```
            # impact of epoch and eta
np.random.seed(3)
theta = np.random.randn(2, 1)  # randomly initialized model parameters

X_test = np.array([25000, 60000]).reshape(-1,1)
X_test_minmax = add_column(minmax.transform(X_test))
plt.scatter(X_minmax, y, s=20, label='training data')
eta= 0.01 # 0.001, 0.01, 0.1, 1
for epoch in range(n_epochs):
    gradients = 2 / N * Xcon.T @ (Xcon @ theta - y.reshape(-1,1))
    theta = theta - eta * gradients
    if epoch == 1:
        plt.plot(X_test_minmax[:,1], X_test_minmax@theta , 'b:', label='1 epochs')
    elif epoch == 10:
        plt.plot(X_test_minmax[:,1], X_test_minmax@theta , 'r--', label='10 epochs')
    elif epoch == 100:
        plt.plot(X_test_minmax[:,1], X_test_minmax@theta , 'g-', label='100 epochs')
plt.legend()
plt.xlabel('GDP (scaled)')
plt.ylabel('life satisfaction')
plt.show()
```

Stochastic gradient descent (SGD) for linear regression

- The main problem with GD is that it uses the whole training set at every step
- Consider a minibatch of size $B = 1$ and a selected sample $\mathbf{x}_n^T$ from $\mathbf{X}$ (row vector)

$$\nabla \mathcal{L}(\boldsymbol{\theta}) = \frac{2}{N}\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) \Rightarrow 2\mathbf{x}_n(\mathbf{x}_n^T\boldsymbol{\theta} - y_n)$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - 2\mathbf{x}_n(\mathbf{x}_n^T\boldsymbol{\theta} - y_n)$$

- Given that N is the sample size and B is the batch size, in one epoch we update our model N/B times

In [ ]:

```
n_epochs = 5
t0, t1 = 5, 50  # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

np.random.seed(42)
theta = np.random.randn(2, 1)  # random initialization

for epoch in range(n_epochs):
    for iteration in range(N):
        random_index = np.random.randint(N)
        xi = np.transpose(Xcon[random_index : random_index + 1])
        yi = y[random_index : random_index + 1]
        gradients = 2 * xi @ (xi.T @ theta - yi)  # for SGD, do not divide by N
        eta = learning_schedule(epoch * N + iteration)
        theta = theta - eta * gradients

print(theta)
```

Sklearn implementation of SGD for linear regression

- **https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegresso**
- Parameters
  - max_iter: epochs
  - learning_rate: constant or variable
  - n_iter_no_change: number of iterations with no improvement to wait before stopping fitting

sklearn.linear_model.SGDRegressor

class `sklearn.linear_model.`**`SGDRegressor`**(*loss='squared_error', *, penalty='l2', alpha=0.0001, l1_ratio=0.15, fit_intercept=True, max_iter=1000, tol=0.001, shuffle=True, verbose=0, epsilon=0.1, random_state=None, learning_rate='invscaling', eta0=0.01, power_t=0.25, early_stopping=False, validation_fraction=0.1, n_iter_no_change=5, warm_start=False, average=False*) [source]

In [ ]:

```
            # GDP data
from sklearn.pipeline import Pipeline
from sklearn.linear_model import SGDRegressor

X = df['GDP per capita (USD)'].to_numpy().reshape(-1,1)
y = df['Life satisfaction'].to_numpy()

pipe = Pipeline([('preprocess', MinMaxScaler()),
                 ('reg', SGDRegressor(random_state=42))])

pipe.fit(X, y)

print(pipe['reg'].intercept_, pipe['reg'].coef_)
```

Polynomial regression

- The linear model may not be a good fit for many problems
    - We can improve the fit by using a polynomial regression model of degree d

$$f(x) = \boldsymbol{\theta}^T \phi(x)$$

where $\phi(x) = [1, x, x^2, \ldots, x^d]$

- This is a simple example of feature preprocessing/engineering
    - Benefit: linear function of parameters but nonlinear wrt input features
- We can use **sklearn.preprocessing.PolynomialFeatures** to generate polynomial features
    - Use pipeline in sklearn to assemble several steps (preprocessing + estimator)

In [ ]:

```
# generate simulated data
import numpy as np
np.random.seed(42)
plt.rcParams.update({'font.size': 16, "figure.figsize": (6,4)})

N = 40
X = 6 * np.random.rand(N, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(N, 1)

plt.plot(X, y, "b.")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, 0, 10])
plt.show()
```

In [ ]:

```
            # train polynomial model
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
pipe = Pipeline(steps=[
    ('poly', PolynomialFeatures(degree=2, include_bias=False)),
    ('regr', LinearRegression())])
pipe.fit(X, y) # training
X_new = np.linspace(-3, 3, 100).reshape(100, 1)
y_new = pipe.predict(X_new) # prediction

plt.plot(X, y, "b.")
plt.plot(X_new, y_new, "r-", linewidth=2, label="Predictions")
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.legend(loc="upper left", fontsize=14)
plt.axis([-3, 3, 0, 10])
plt.show()
```

In [ ]:

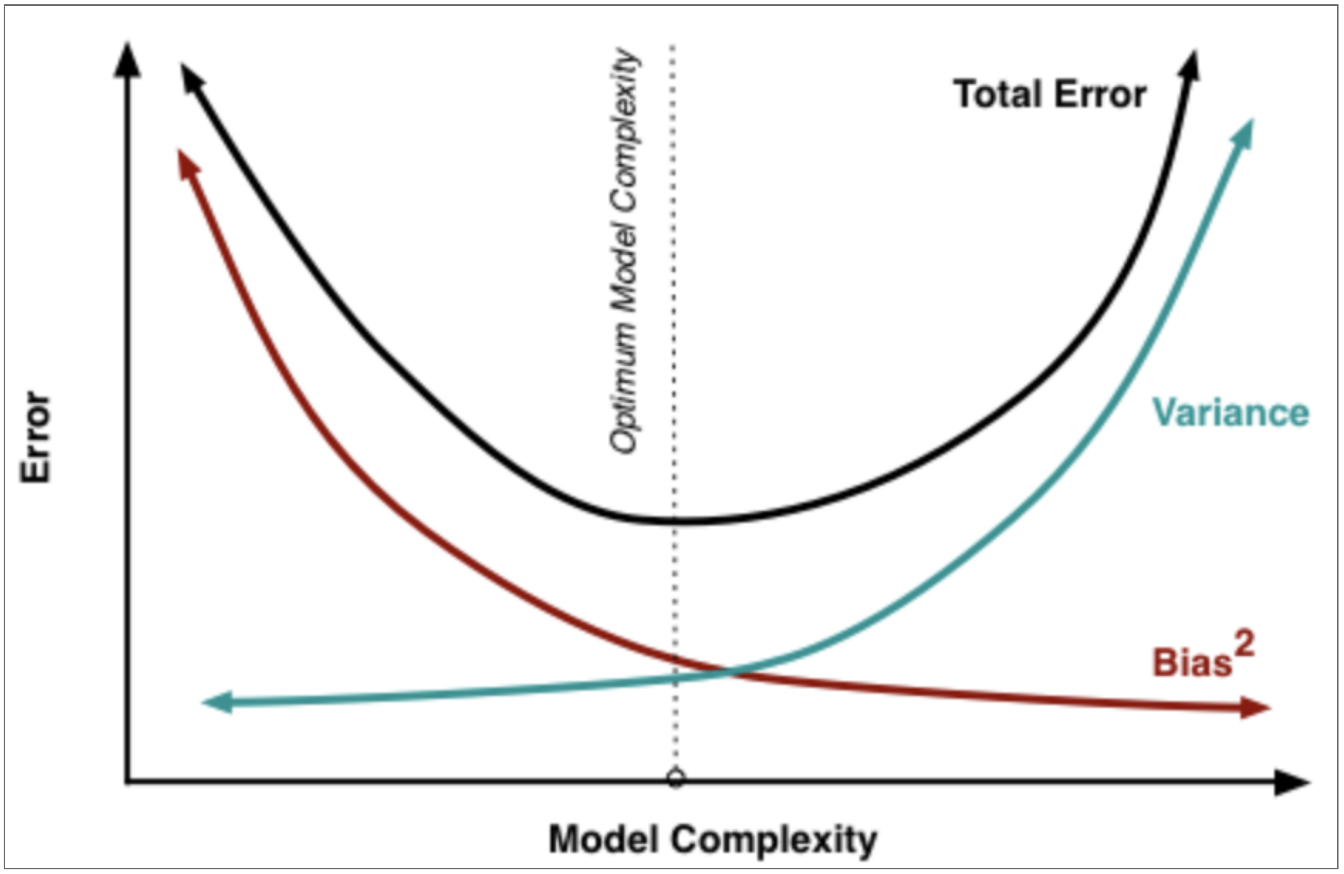
```
            # Compare varying complexity levels
from sklearn.preprocessing import StandardScaler
for style, width, degree in (("g-", 1, 30), ("b--", 1, 2), ("r-+", 1, 1)):
    polybig_features = PolynomialFeatures(degree=degree, include_bias=False)
    std_scaler = StandardScaler()
    lin_reg = LinearRegression()
    polynomial_regression = Pipeline([("poly_features", polybig_features),
                                      ("std_scaler", std_scaler),
                                      ("lin_reg", lin_reg)])
    polynomial_regression.fit(X, y)
    y_newbig = polynomial_regression.predict(X_new)
    plt.plot(X_new, y_newbig, style, label=str(degree), linewidth=width)
plt.plot(X, y, "b.", linewidth=3)
plt.legend()
plt.xlabel("$x$", fontsize=18)
plt.ylabel("$y$", rotation=0, fontsize=18)
plt.axis([-3, 3, -10, 10])
plt.show()
```

The bias-variance tradeoff

- A model's generalization/test error can be expressed as the sum of
    - Bias due to wrong assumptions
        - A high-bias model is likely to underfit the data
    - Variance due to excessive sensitivity to small variations in the training data
        - A high-variance model is likely to overfit the data
    - Irreducible error due to noisiness of the data itself

The bias-variance tradeoff

- As model flexibility increases, training error decreases, but there is a U-shape in test error
- In practice, computing training error is straightforward, but estimating test error is considerably more difficult because no test data are available



Regularization

- Regularization is a way to avoid overfitting by shrinking or simplifying the model

$$\mathcal{L}(\boldsymbol{\theta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\theta}\|_2^2 + \lambda C(\boldsymbol{\theta})$$

- $\lambda \geq 0$ is the regularization parameter (i.e., hyperparameter) and $C(\boldsymbol{\theta})$ is some form of model complexity
- We can quantify complexity using the ℓ_2 regularization formula, i.e., the sum of the squares of all weights

$$\ell_2 \text{ regularization} = \|\boldsymbol{\theta}\|_2^2 = \theta_0^2 + \theta_1^2 + \theta_2^2 + \dots$$

ℓ_1 regularization or LASSO

- LASSO: Least Absolute Shrinkage and Selection Operator
    - Uses the ℓ_1 norm of weights, instead of ℓ_2

$$\|\boldsymbol{\theta}\|_1 = |\theta_0| + |\theta_1| + |\theta_2| + \ldots$$

- Definition of the ℓ_p norm for a real number $p \geq 1$

$$\|\boldsymbol{\theta}\|_p = \left(\sum_i |\theta_i|^p\right)^{1/p}$$

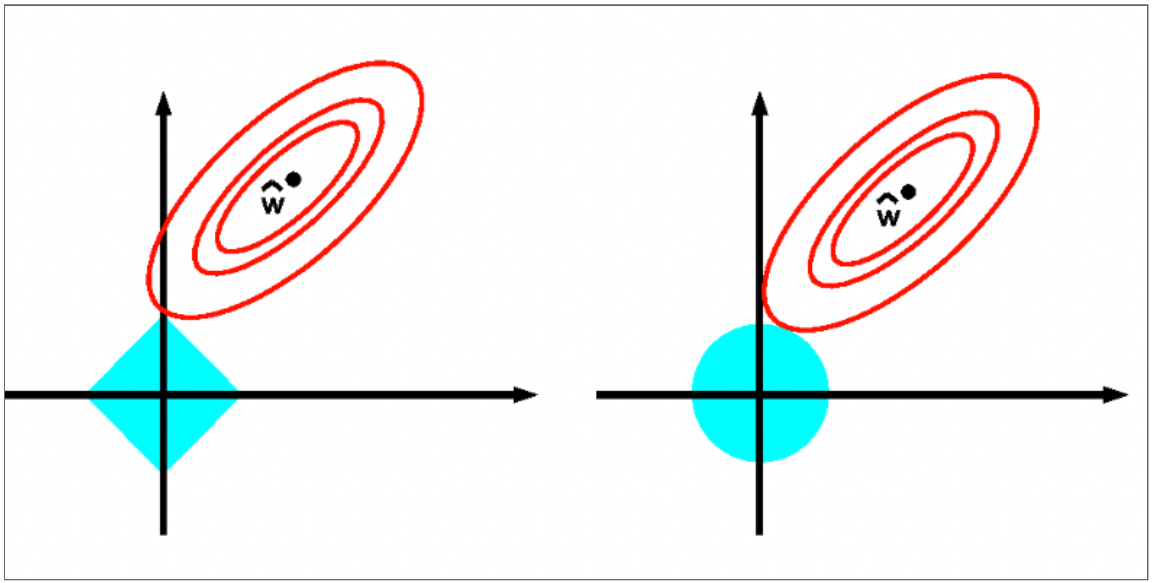
- LASSO tends to eliminate the weights of the least important features (i.e., set them to zero)
    - Next slide provides an intuitive explanation

ℓ_1 vs ℓ_2 regularization

- Write optimization problems in bound constrained forms

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \;\; \text{s.t.} \;\; \|\boldsymbol{\theta}\|_1 \leq B \;\; \text{or} \;\; \|\boldsymbol{\theta}\|_2^2 \leq B$$

- Let us look at the contours of the ℓ_1 and ℓ_2 constrained surfaces
  - Corners of the ℓ_1 ball are more likely to intersect the ellipse than one of the sides



In [ ]:

```
            # Synthetic/simulated data
from sklearn.linear_model import Ridge
np.random.seed(42)
N = 20
X = 3 * np.random.rand(N, 1)
y = 1 + 0.5 * X + np.random.randn(N, 1) / 1.5
X_new = np.linspace(0, 3, 100).reshape(100, 1)
alphas=(0, 10**-5, 1)
for alpha, style in zip(alphas, ("b-", "g--", "r:")): # zip: aggregates them in a tuple
    model = Pipeline([
        ("poly_features", PolynomialFeatures(degree=10, include_bias=False)),
        ("std_scaler", StandardScaler()),
        ("regul_reg", Ridge(alpha=alpha)), # alpha = lambda
                    ])
    model.fit(X, y)
    plt.plot(X_new, model.predict(X_new), style, linewidth=2, label=r"$\alpha = {}$".format(alpha))
plt.plot(X, y, "k.", markersize=10)
plt.legend()
plt.show()
```

In [ ]:

```
            # Lasso
import numpy as np
import pandas as pd

# “Hitters” with 20 variables and 322 observations of major league players
df = pd.read_csv('Hitters.csv', index_col=[0])

# drop missing cases
df = df.dropna()

df.head()
```

In [ ]:

```
# Preprocessing, encode our categorical features as one-hot numeric features

dummies = pd.get_dummies(df[['League', 'Division','NewLeague']])

dummies.head()
```

In [ ]:

```
# Find outputs

y = df['Salary']

y.shape
```

In [ ]:

```
# drop the "Salary" column  and categorical columns

df_input = df.drop(['Salary', 'League', 'Division', 'NewLeague'], axis=1)


# Create all features
X = pd.concat([df_input, dummies[['League_N', 'Division_W', 'NewLeague_N']]], axis=1).to_numpy()

X.shape
```

In [ ]:

```
# Split the data set into train and test set 70/30

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=10)
```

In [ ]:

```
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

In [ ]:
```
X_train
```

Should we use preprocessing?

In [ ]:

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

a = scaler.fit_transform(X_train[:,0:16])

b = scaler.transform(X_test[:,0:16])

X_train = np.concatenate((a, X_train[:,16:]), axis=1)

X_test = np.concatenate((b, X_test[:,16:]), axis=1)

X_train
```

In [ ]:

```
            from sklearn.linear_model import Lasso

alphas = np.linspace(1,300,100)

lasso = Lasso(max_iter=10000)

coefs = []

for a in alphas:
    lasso.set_params(alpha=a)
    lasso.fit(X_train, y_train)
    coefs.append(lasso.coef_)
```

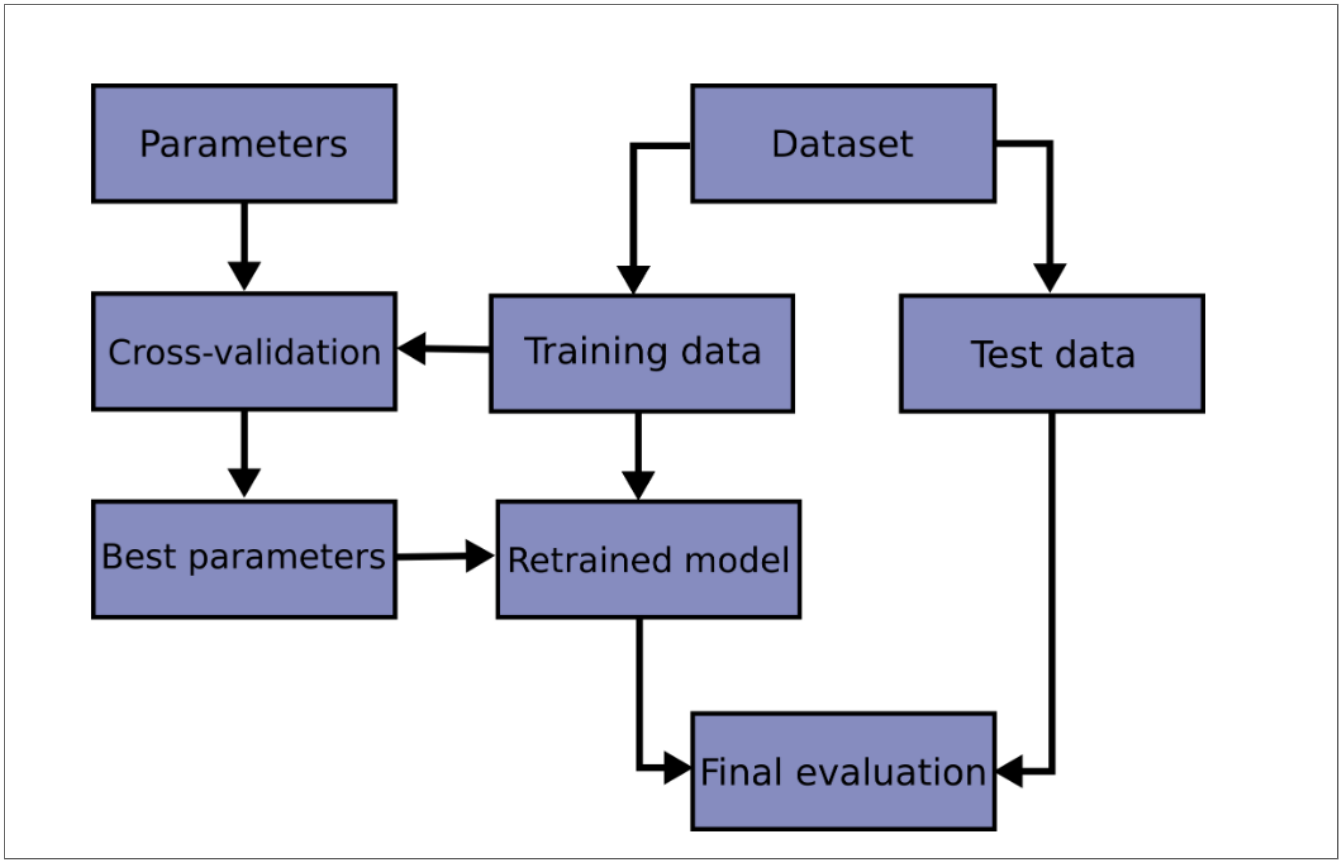
In [ ]:

```
            plt.rcParams.update({'font.size': 15, "figure.figsize": (14,5)})
max_vals = [item.max() for item in coefs]
min_vals = [item.min() for item in coefs]
zero_vals = [np.sum(item == 0) for item in coefs]
plt.subplot(121)
plt.plot(alphas, max_vals, 'b--')
plt.plot(alphas, min_vals, 'r--')
plt.xlabel('alpha')
plt.ylabel('max/min values')

plt.subplot(122)
plt.plot(alphas, zero_vals)
plt.xlabel('alpha')
plt.ylabel('number of zeros')
plt.show()
```

Model selection

- Model selection is the process of selecting the best one by comparing and validating with various parameters
    - **https://scikit-learn.org/stable/modules/cross_validation.html**



In [ ]:

```
# Exhaustive search over specified parameter values for an estimator.

from sklearn.model_selection import GridSearchCV

parameters = {'alpha': np.linspace(1,300,100)}

lasso = Lasso(max_iter=10000)

reg = GridSearchCV(lasso, parameters, scoring='r2')

reg.fit(X_train, y_train)

reg.best_params_
```

In [ ]:

```
            # Set the best alpha

lasso_best = Lasso(alpha=reg.best_params_['alpha'], max_iter=10000)

lasso_best.fit(X_train, y_train)

lasso_best.coef_
```

In [ ]:

```
            # R2 for training and test sets

from sklearn.metrics import r2_score

pred_train = lasso_best.predict(X_train)
r2_train = r2_score(y_train, pred_train)
print('R2 training set', round(r2_train, 2))

# Test data
pred_test = lasso_best.predict(X_test)
r2_test = r2_score(y_test, pred_test)
print('R2 test set', round(r2_test, 2))
```