Econ 425 Week 3

Linear models and regularization

Grigory Franguridi

UCLA Econ

USC CESR

franguri@usc.edu

Example: predicting unemployment

Data:

- y – unemployment rate
- x_1: lagged GDP
- x_2: lagged inflation rate
- x_3: expected inflation rate (from surveys)
- x_4: lagged public spending
- $x_5, x_6, \ldots,$ other economic indicators

Goal: predict unemployment rate y

Linear regression

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p + \epsilon = \boldsymbol{x}'\boldsymbol{\beta} + \epsilon$$

- some features may be highly correlated,
  which may impact performance
- for policy practice, often need to identify which features most affect unemployment
- $\Rightarrow$ may want a **sparse** estimate $\hat{\boldsymbol{\beta}}$ (i.e. with some $\widehat{\beta}_i = 0$),
  while maintaining prediction accuracy;
  especially when p is large
- why sparsity?
  - remove highly correlated features
  - prevent overfitting (i.e. reduce variance)
  - enhance interpretability
- how to achieve sparsity?

Solution: shrinkage

Generic shrinkage (regularized) regression:

$$(\hat{\beta}_0, \hat{\boldsymbol{\beta}}) = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \beta_0 - \mathbf{x}_i' \boldsymbol{\beta})^2 + \lambda J(\boldsymbol{\beta}),$$

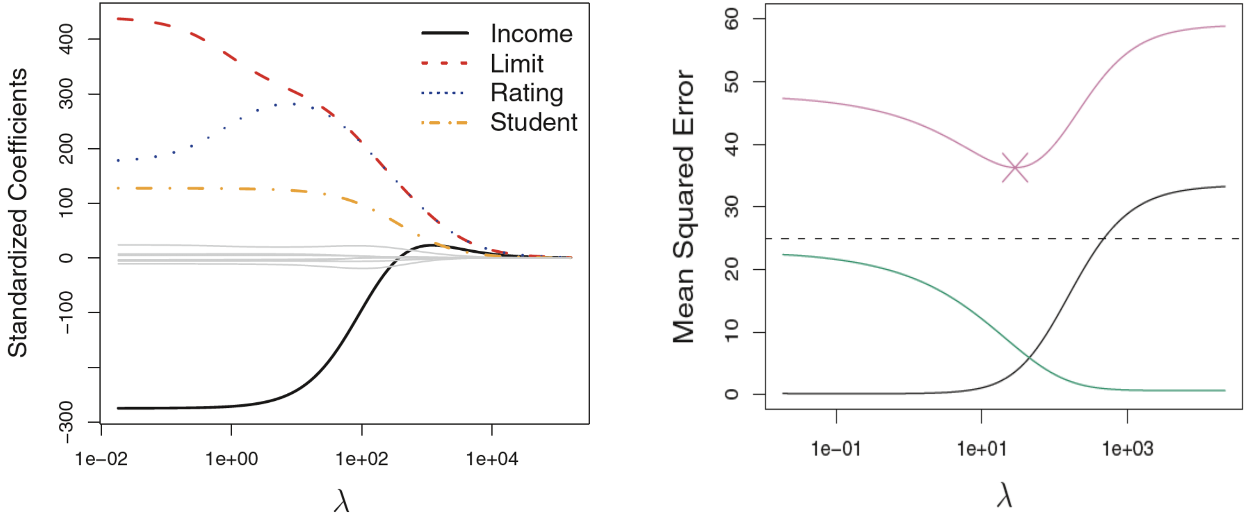
where $J(\cdot)$ is **penalty** (for deviating from desired form of β)

- different $J(\cdot)$'s lead to different shrinkage solutions
- e.g., if we want $\boldsymbol{\beta}$ to have as many zeros as possible, we can define $J(\boldsymbol{\beta}) =$ number of non-zeros in $\boldsymbol{\beta}$ (thresholding)
- after centralization, the problem becomes

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \mathbf{x}_i' \boldsymbol{\beta})^2 + \lambda J(\boldsymbol{\beta})$$

Shrinkage bias

- Generally, any shrinkage estimator $\hat{\beta}_\lambda$ is biased, i.e. $\mathbb{E}\hat{\beta}_\lambda \neq \beta$
- impact of the tuning parameter λ:



Right panel: squared bias (black), variance (green), test error (purple)

Shrinkage I: ridge regression

$$\hat{\beta}_{\lambda}^{ridge} = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (y_i - \mathbf{x}_i' \boldsymbol{\beta})^2 + \lambda \|\boldsymbol{\beta}\|^2,$$

where $\|\boldsymbol{\beta}\|^2 = \sum_{j=1}^{p} \beta_j^2 = \boldsymbol{\beta}'\boldsymbol{\beta}$ is the L_2 (Euclidean) penalty

- shrinks the estimates $\hat{\beta}_{\lambda}^{ridge}$ towards zero
- the tuning parameter $\lambda > 0$ controls the bias-variance trade-off (fitting quality vs shrinkage)
- if $\lambda = 0$, $\hat{\beta}_{\lambda}^{ridge} = \hat{\beta}_{\lambda}^{OLS}$ (unbiased, but no shrinkage)
- if $\lambda \to \infty$, $\hat{\beta}_{\lambda}^{ridge} = 0$ (biased, but extreme shrinkage)

Shrinkage I: ridge regression

Fact:

$$\hat{\beta}_{\lambda}^{ridge} = (\mathbf{X'X} + \lambda \mathbf{I_p})^{\mathbf{-1}}\mathbf{X'y}$$

Proof:

- optimization problem:

$$\hat{\beta}_{\lambda}^{ridge} = \underset{\beta}{\text{argmin}}\ \hat{Q}(\beta) = \underset{\beta}{\text{argmin}}\ (\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) + \lambda\|\beta\|^2$$

- first term:

$$(\mathbf{y} - \mathbf{X}\beta)^T(\mathbf{y} - \mathbf{X}\beta) = \mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\beta - \beta^T\mathbf{X}^T\mathbf{y} + \beta^T\mathbf{X}^T\mathbf{X}\beta$$

- therefore,

$$Q(\beta) = \mathbf{y}^T\mathbf{y} - 2\mathbf{y}^T\mathbf{X}\beta + \beta^T\mathbf{X}^T\mathbf{X}\beta + \lambda\beta^T\beta$$

Shrinkage I: ridge regression

- differentiate and set to zero:

$$\frac{\partial Q}{\partial \beta} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\beta + 2\lambda\beta = 0$$

- solve for β:

$$(\mathbf{X}^T\mathbf{X} + \lambda I)\beta = \mathbf{X}^T\mathbf{y}$$

or

$$\hat{\beta}_\lambda^{ridge} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{y}$$

□

Shrinkage I: ridge regression

equivalent formulation:

$$\begin{aligned}\hat{\beta}_\lambda^{ridge} &= \underset{\beta}{\operatorname{argmin}}\ (\mathbf{y} - \mathbf{X}\beta)^{\mathbf{T}}(\mathbf{y} - \mathbf{X}\beta) \\ &\quad \text{subject to } \|\beta\|^2 \le s_\lambda\end{aligned}$$

Shrinkage II: Lasso

$$\hat{\beta}_{\lambda}^{lasso} = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \mathbf{x}_i' \boldsymbol{\beta})^2 + \lambda \|\boldsymbol{\beta}\|_1,$$

where $\|\boldsymbol{\beta}\|_1 = \sum_{j=1}^{p} |\beta_j|$ is the L_1 (absolute) penalty

- lasso $=$ least absolute shrinkage and selection operator
- equivalently,

$$\begin{aligned} \hat{\beta}_{\lambda}^{lasso} &= \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{n} (y_i - \mathbf{x}_i' \boldsymbol{\beta})^2 \\ & \quad \text{subject to } \|\beta\|_1 \leq s \end{aligned}$$

- shrinks many coefficients to exact zeros
- in contrast to ridge, there is no explicit solution
- need to use quadratic programming (QP)

Example: Prostate cancer - background

- Prostate cancer occurs in the prostate, a small walnut-shaped gland in men that produces seminal fluid
- early detection and accurate diagnosis of prostate cancer are crucial for effective treatment and improving patient outcomes
- ML models applied to datasets containing clinical information can assist in predicting the likelihood of prostate cancer based on relevant features

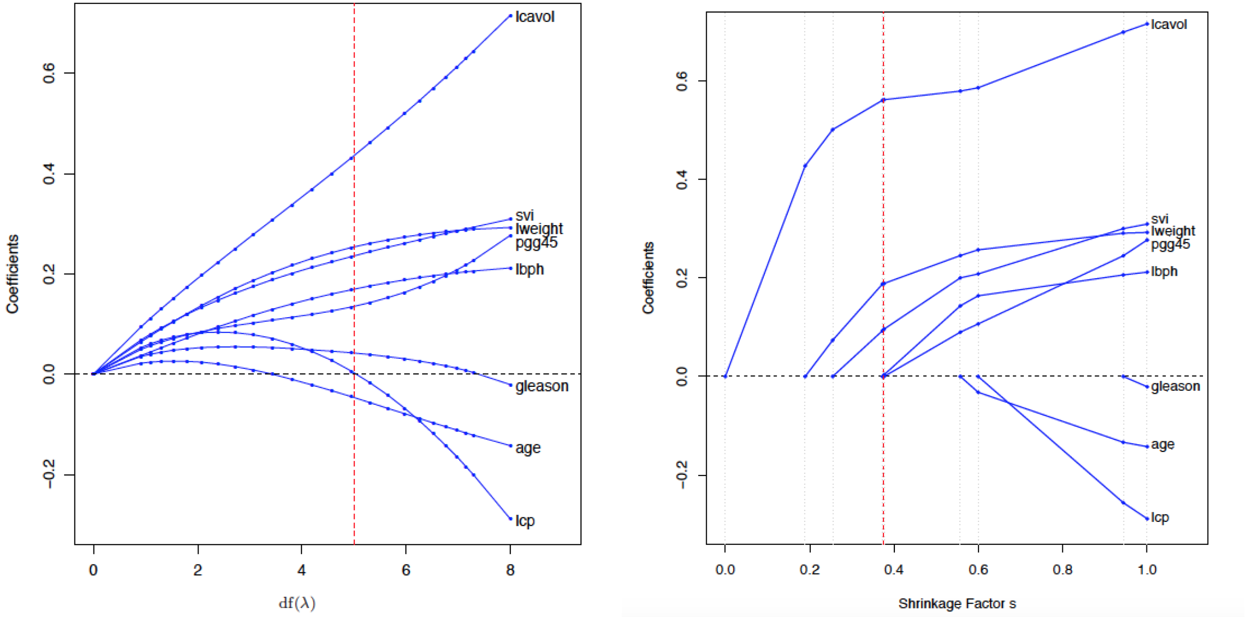
Example: Prostate cancer - background

- Clinical features:
    - age of the patient
    - prostate-specific antigen (PSA) levels in the blood
    - biopsy Gleason scores, which characterize the aggressiveness of prostate cancer cells based on their microscopic appearance
    - other features
- Target variable:
    - presence/absence of prostate cancer
    - in some cases, additional outcomes such as cancer stage

Example: Prostate cancer - experiment

- train logistic regression for prostate cancer prediction
- use ridge or lasso penalty in the logistic regression and plot the coefficients under different regularization parameters

Example: Prostate cancer - results



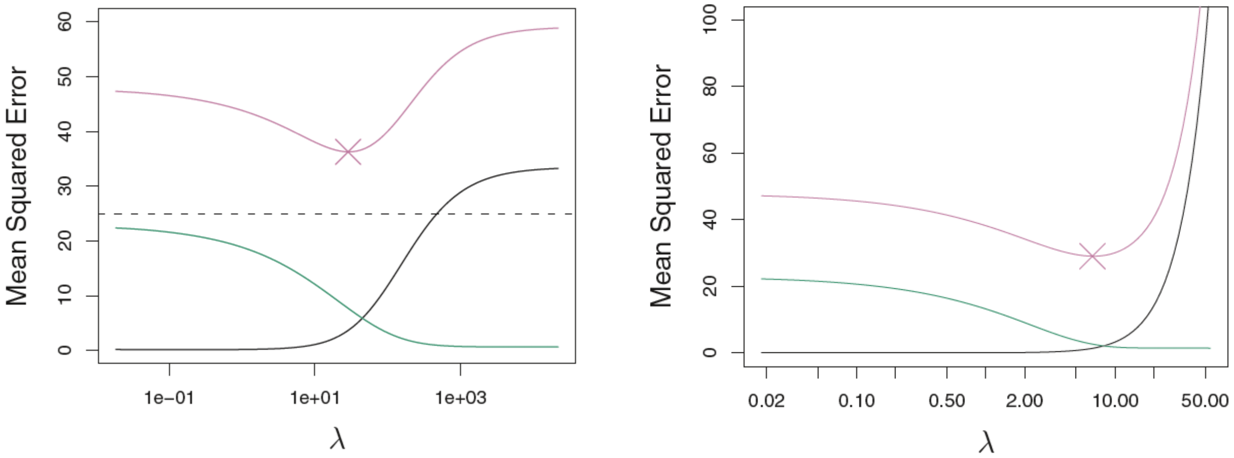
Left: ridge; Right: lasso

Example: Prostate cancer - Interpretation

- Ridge: coefficient are shrunk to zero, but not sparse (i.e. no exact zeros)
- Lasso: coefficients are sparse (exact zeros)

Ridge vs Lasso

- both lasso and ridge shrink coefficients while introducing some bias
- lasso: more sparse and more interpretable models (with only a subset of predictors)
- unclear which one leads to better prediction accuracy in general
- let us compare in a special case (next page)



正交的

Orthogonal case

Let $\underline{n} = p$ and $\mathbf{X} = \underline{\mathbf{I_p}}$. Then

# of obs   \# of params

$\hat{\beta}_{OLS}$: $\operatorname{argmin}_{\beta} \sum_{i=1}^{n} (y_i - \mathbf{x}_i'\boldsymbol{\beta})^2$

- OLS:

Training error = 0 $$\hat{\beta}_j^{ols} = y_j$$

- Ridge:

Shrink to 0 (λ ↑ shrink more)

$$\hat{\beta}_j^{ridge} = y_j/(1+\lambda)$$

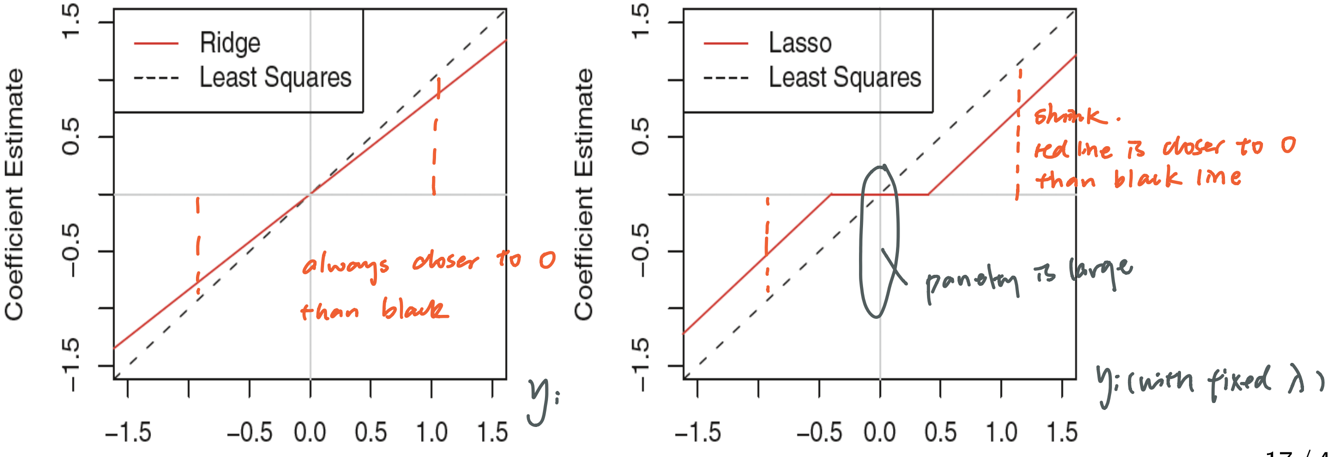
$(XX' + \lambda I_p)^{-1} X'y$

- Lasso: $(a_+ = a$ for $a > 0$; $a_+ = 0$ otherwise)

$$\hat{\beta}_j^{lasso} = \text{sign}(y_j)(|y_j| - \lambda/2)_+$$

panelty range

λ 越大. panelty 越大



Elastic net: ridge + lasso

弹性网络回归.

- leverages benefits of both ridge and lasso
- addresses some limitations of lasso, e.g. its tendency to arbitrarily select one feature among a group of highly correlated features
- introduces an additional hyperparameter to control the mixture of L_1 and L_2 regularization: its penalty function is

$$J(\beta) = \lambda_1 \|\beta\|_1 + \lambda_2 \|\beta\|^2$$

General regularization: bridge

$$\hat{\beta}^{bridge} = \underset{\beta}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{X}\beta\|^{\mathbf{2}} + \lambda \cdot \mathbf{L_r}(\beta),$$

where $L_r(\beta)$ is the r-th power of L_r norm of β
(take limit if $r = 0$ or $r = \infty$)

Examples:

- $L_0(\beta) = \sum_{j=1}^{p} I(\beta_j \neq 0)$ (hard thresholding)
- $L_1(\beta) = \sum_{j=1}^{p} |\beta_j|$ (lasso)
- $L_2(\beta) = \sum_{j=1}^{p} \beta_j^2$ (ridge)
- $L_\infty(\beta) = \max_j |\beta_j|$

一般还是用这俩。

Bridge penalties

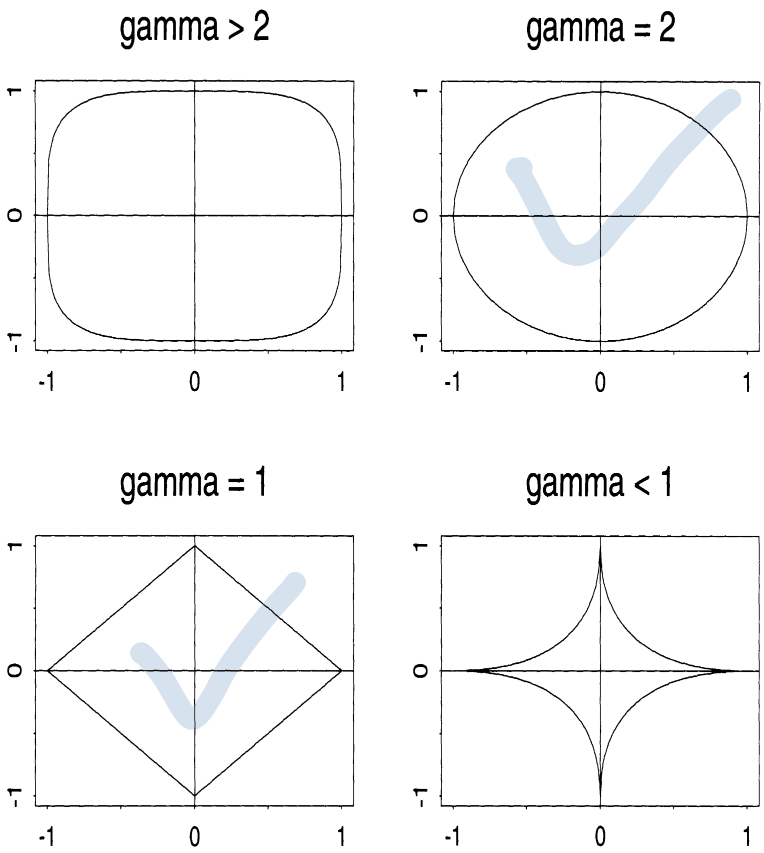
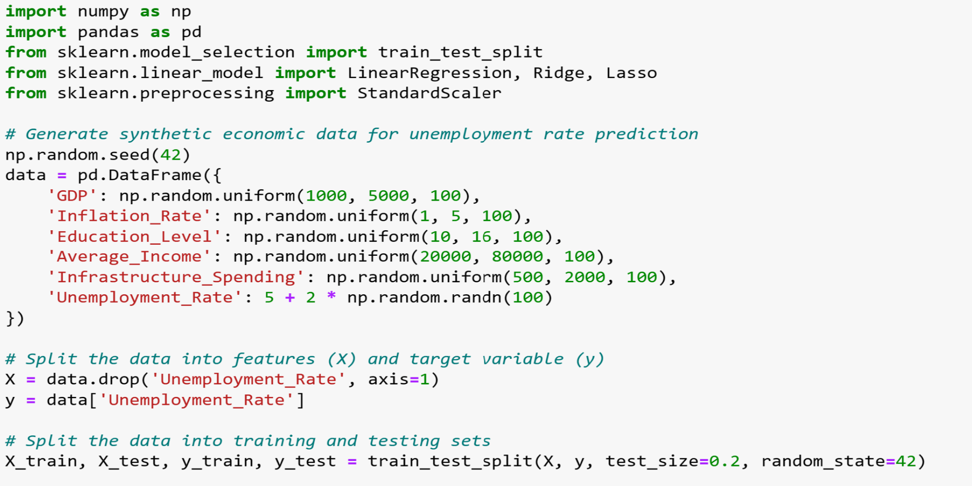


Figure 1. Constrained Areas of Bridge Regressions with $t = 1$.

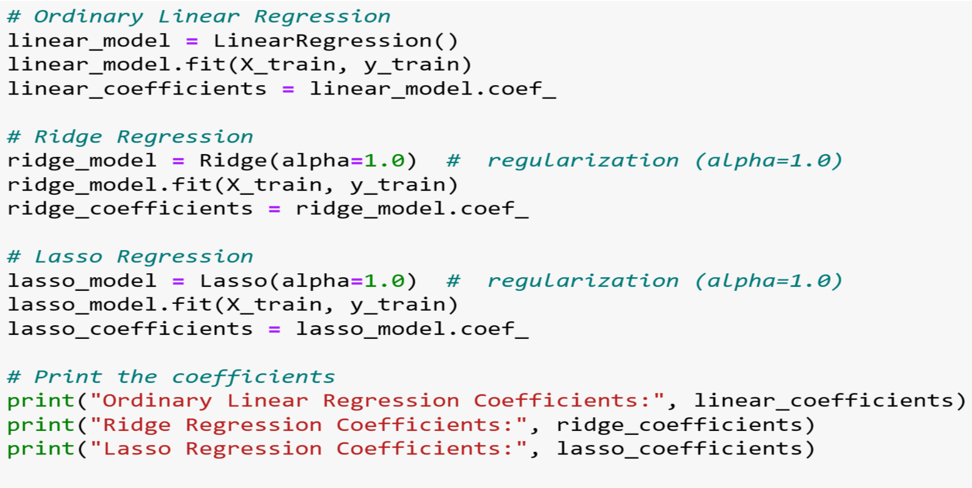
Revisit unemployment rate prediction example

- Let us take a look at the coefficients obtained using OLS, ridge and lasso

Codes: Linear regression vs Ridge vs Lasso

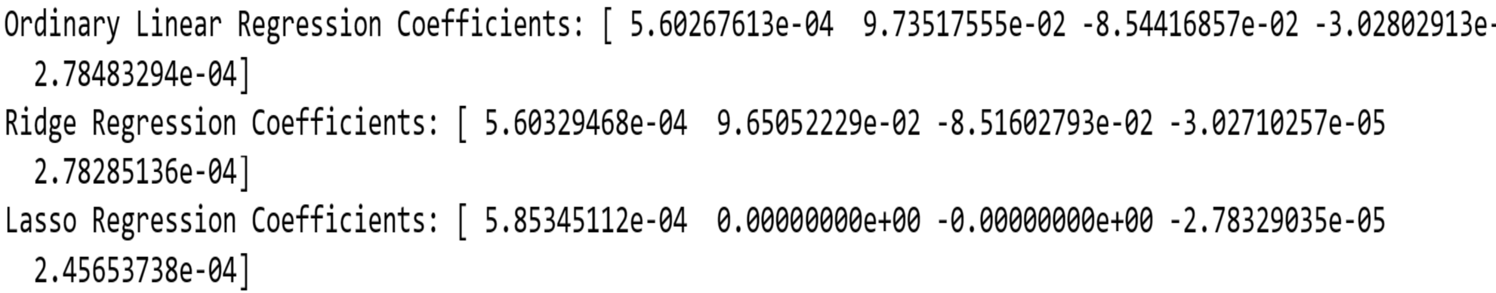


```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.preprocessing import StandardScaler

# Generate synthetic economic data for unemployment rate prediction
np.random.seed(42)
data = pd.DataFrame({
    'GDP': np.random.uniform(1000, 5000, 100),
    'Inflation_Rate': np.random.uniform(1, 5, 100),
    'Education_Level': np.random.uniform(10, 16, 100),
    'Average_Income': np.random.uniform(20000, 80000, 100),
    'Infrastructure_Spending': np.random.uniform(500, 2000, 100),
    'Unemployment_Rate': 5 + 2 * np.random.randn(100)
})

# Split the data into features (X) and target variable (y)
X = data.drop('Unemployment_Rate', axis=1)
y = data['Unemployment_Rate']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```



```python
# Ordinary Linear Regression
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)
linear_coefficients = linear_model.coef_

# Ridge Regression
ridge_model = Ridge(alpha=1.0)  #  regularization (alpha=1.0)
ridge_model.fit(X_train, y_train)
ridge_coefficients = ridge_model.coef_

# Lasso Regression
lasso_model = Lasso(alpha=1.0)  #  regularization (alpha=1.0)
lasso_model.fit(X_train, y_train)
lasso_coefficients = lasso_model.coef_

# Print the coefficients
print("Ordinary Linear Regression Coefficients:", linear_coefficients)
print("Ridge Regression Coefficients:", ridge_coefficients)
print("Lasso Regression Coefficients:", lasso_coefficients)
```

Results: OLS vs Ridge vs Lasso



```
Ordinary Linear Regression Coefficients: [ 5.60267613e-04  9.73517555e-02 -8.54416857e-02 -3.02802913e-
  2.78483294e-04]
Ridge Regression Coefficients: [ 5.60329468e-04  9.65052229e-02 -8.51602793e-02 -3.02710257e-05
  2.78285136e-04]
Lasso Regression Coefficients: [ 5.85345112e-04  0.00000000e+00 -0.00000000e+00 -2.78329035e-05
  2.45653738e-04]
```

Conclusion:

- Ridge regression slightly **shrinks** the coefficients, while lasso regression makes the fitting coefficients **sparse**

Hyperparameter tuning: how to choose λ

- are the training results the same under different hyperparameters? No.
- $\Rightarrow$ need hyperparameter tuning

Choose λ using CV

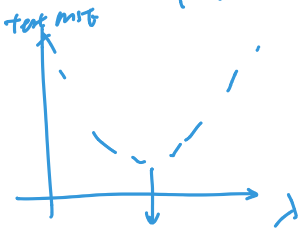
① Pick a grid. say $\lambda \in \{0, 0.01, \cdots 0.050\}$

② For each λ:

2.1 Fill the model

2.2 Compute $\underbrace{\text{test MSE}}_{\text{function of } \lambda}$ using CV

③ draw test MSE



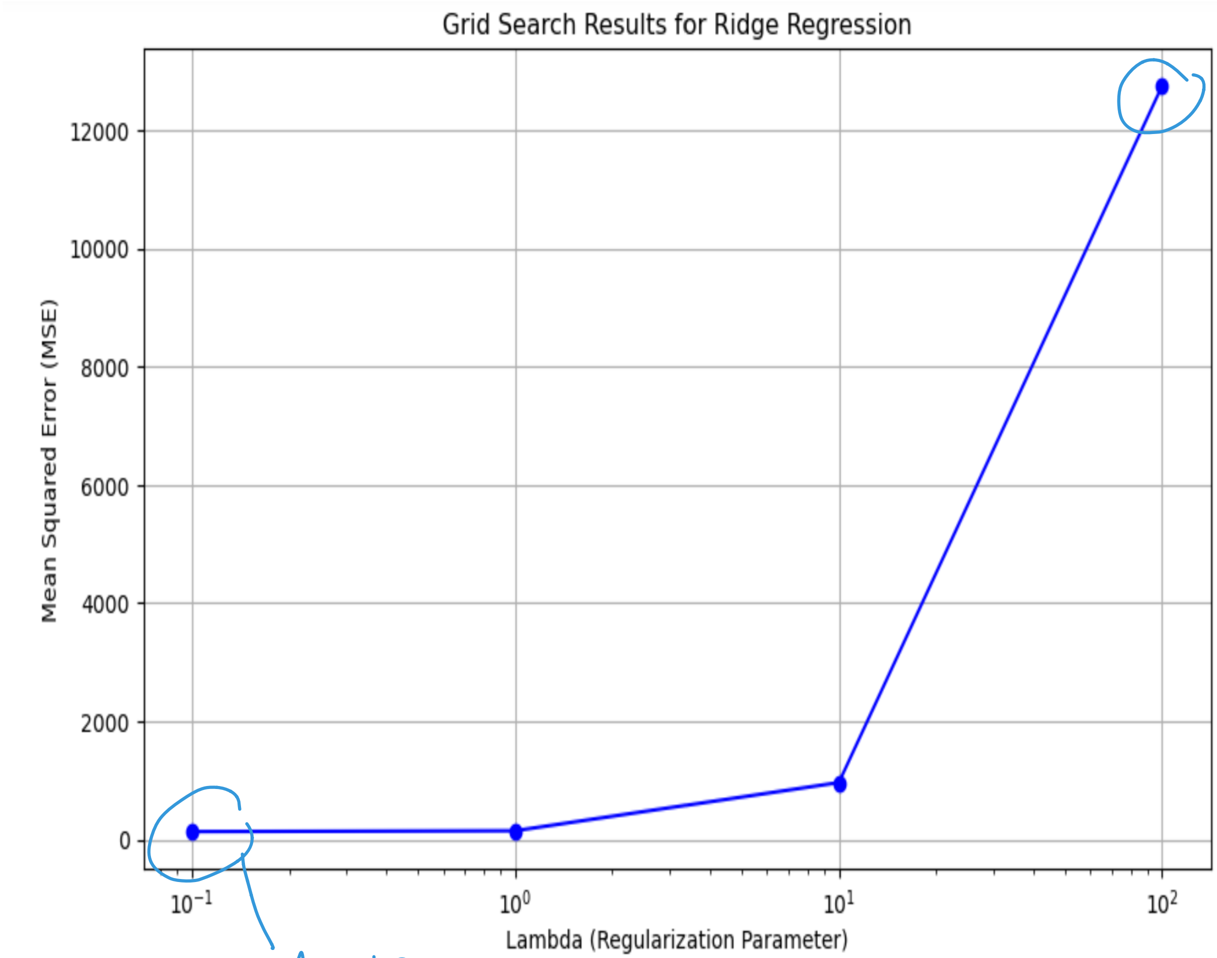
find the minimum point → get λ

Example: ridge

made-up example

- generate a synthetic dataset for regression
- split the data into training and testing sets
- train the ridge regression with different hyperparameters $\lambda = 0.1, 1, 10, 100$
- estimate the test MSE

Example: ridge



too high

choose

一般 $\lambda \in (0,1)$

Optimization basics

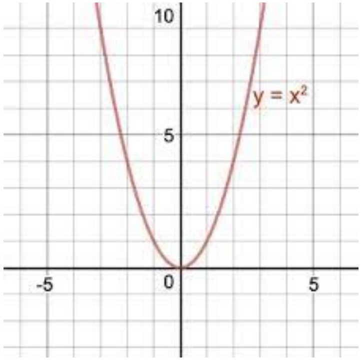
- **Question**: how to minimize a function $f : \mathbb{R} \to \mathbb{R}$ (such as loss function)?
- **Answer**: take the derivative w.r.t. x and set to zero, i.e. solve

$$\frac{d}{dx} f(x) = 0$$

- **Question**: what if there is no explicit solution?
- **Answer**: solve numerically using an optimization algorithm such as gradient descent

Gradient descent: an example

Suppose we want to minimize $f(x) = x^2$



1. randomly choose an initial point $x^{(0)} = 3$
2. calculate the derivative at the point $x^{(0)} = 3$,

$$f'(3) = 6$$

3. update $x^{(1)} = x^{(0)} - \alpha \cdot f'(3)$,
   where α is the step size, aka learning rate
4. repeat the above process until convergence

Gradient descent

- **motivation**: minus gradient is direction of steepest descent of f

- **General form of GD**: Let f be a p-variate function. Then GD is defined by

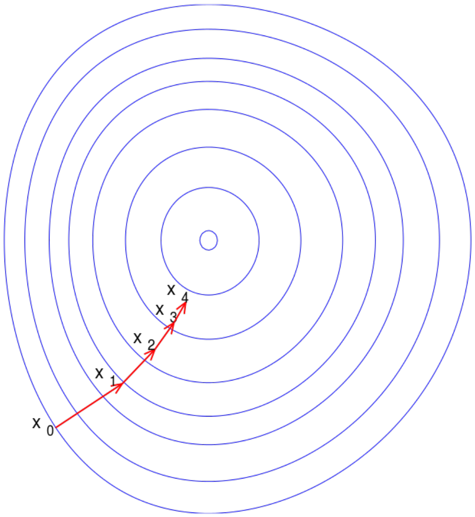
$$\boldsymbol{x}^{(t+1)} = \boldsymbol{x}^{(t)} - \lambda \nabla f(\boldsymbol{x}^{(t)}),$$

where $\nabla f(\boldsymbol{x}) = \left(\frac{\partial}{\partial x_1} f(\boldsymbol{x}), \frac{\partial}{\partial x_2} f(\boldsymbol{x}), \ldots, \frac{\partial}{\partial x_p} f(\boldsymbol{x})\right)$

- **Question**: when (on which iteration t) to stop the GD?
- **Answer**: when $\nabla f(\boldsymbol{x}^{(t)}) \approx \mathbf{0}$ (i.e. close to local minimum)

Applications of gradient descent

- GD is usually employed when (i) f is non-convex, (ii) the minimum cannot be derived analytically, or (iii) $\boldsymbol{x}$ is high-dimensional, e.g., in deep neural networks ((i)+(ii)+(iii))



- if the loss function is convex, then the output of GD is guaranteed to be the optimal solution

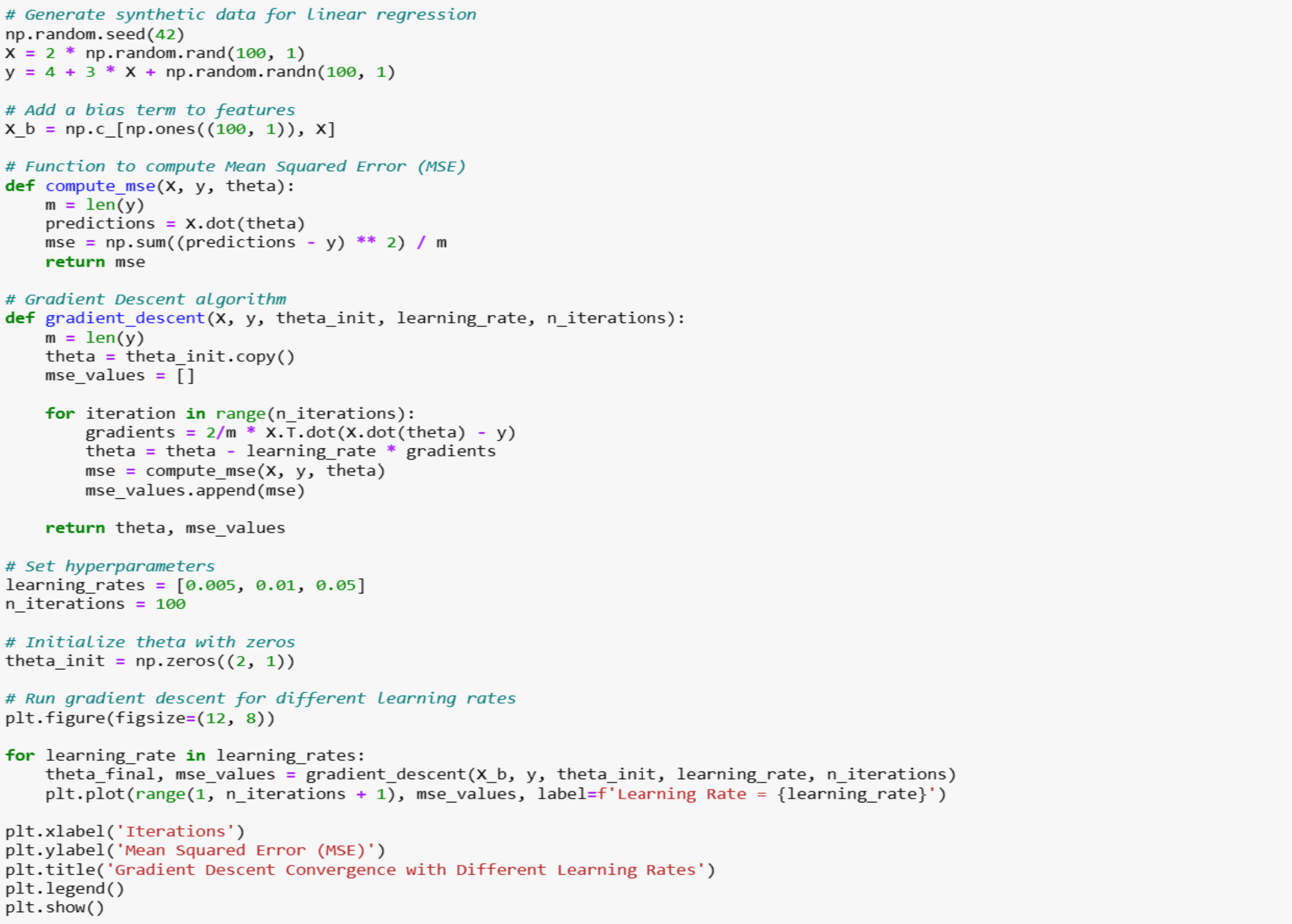
Choosing learning rate in gradient descent

- GD has different convergence rates with different choices of learning rate α
- See the example on the next slide

Choosing learning rate in gradient descent: example

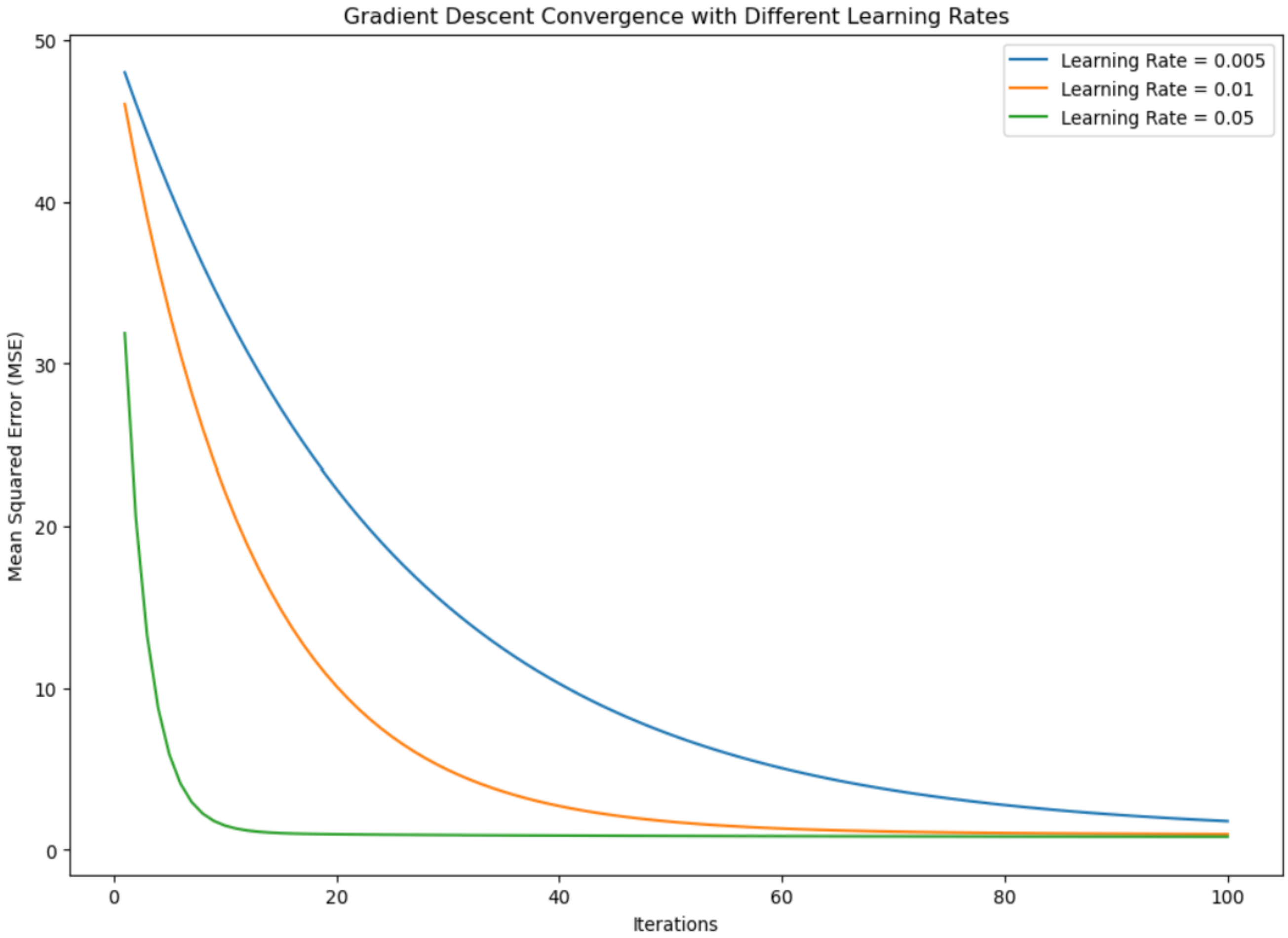
- generate synthetic data for linear regression
- compute the MSE
- use GD to find the minimum
- explore different learning rates $\alpha = 0.005, 0.01, 0.05$ and plot the MSE vs iterations

Learning rate tuning in Gradient Descent: Codes



```
# Generate synthetic data for linear regression
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)

# Add a bias term to features
X_b = np.c_[np.ones((100, 1)), X]

# Function to compute Mean Squared Error (MSE)
def compute_mse(X, y, theta):
    m = len(y)
    predictions = X.dot(theta)
    mse = np.sum((predictions - y) ** 2) / m
    return mse

# Gradient Descent algorithm
def gradient_descent(X, y, theta_init, learning_rate, n_iterations):
    m = len(y)
    theta = theta_init.copy()
    mse_values = []

    for iteration in range(n_iterations):
        gradients = 2/m * X.T.dot(X.dot(theta) - y)
        theta = theta - learning_rate * gradients
        mse = compute_mse(X, y, theta)
        mse_values.append(mse)

    return theta, mse_values

# Set hyperparameters
learning_rates = [0.005, 0.01, 0.05]
n_iterations = 100

# Initialize theta with zeros
theta_init = np.zeros((2, 1))

# Run gradient descent for different learning rates
plt.figure(figsize=(12, 8))

for learning_rate in learning_rates:
    theta_final, mse_values = gradient_descent(X_b, y, theta_init, learning_rate, n_iterations)
    plt.plot(range(1, n_iterations + 1), mse_values, label=f'Learning Rate = {learning_rate}')

plt.xlabel('Iterations')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Gradient Descent Convergence with Different Learning Rates')
plt.legend()
plt.show()
```

Learning rate tuning in Gradient Descent: Plots



Exercise: GD on a simple function

- **Problem**: let $f(x) = x^2$. Find the value of x that that minimizes $f(x)$ using GD.

- **How to**:
  - start with an initial guess for x, say $x = 10$
  - use gradient descent to find a value of x that minimizes $f(x)$
  - perform three iterations by hand
  - assume the learning rate $\alpha = 0.1$

Exercise: GD on a simple function

Steps:

- Calculate the gradient: $\frac{df}{dx} = 2x$
- Update rule: $x = x - \alpha \cdot 2x$
- Iterations:
  1. $x = 8.0$
  2. $x = 6.4$
  3. $x = 5.12$
- Conclusion: the value of x keeps going down, eventually converges to $x = 0$

Drawbacks of conventional GD

- **computational intensity**: GD requires the computation of gradients for the entire dataset (recall that the loss function is defined on the whole dataset) to perform a single update of the model parameters
- **memory constraints**: storing the entire dataset in memory for computation can be impractical or impossible with large datasets

Drawbacks of conventional GD

- **redundant calculations**: with real-world data, many samples may be similar or redundant. GD processes the entire dataset in each iteration, leading to redundant calculations that do not significantly contribute to learning
- **convergence speed**: using the entire dataset for each update makes GD slow for large datasets
- **difficulty in escaping local minima**: in high-dimensional and complex error landscapes (common in deep learning), GD can get stuck in local minima or saddle points, especially if the initial parameter values are not optimal

Drawbacks of conventional GD

- How to deal with the above drawbacks?
- **Use stochastic gradient descent (SGD)**

SGD: basics

- **GD** uses the **entire dataset** to compute the gradient
- **SGD** uses only **only a single data point** (or a small batch of data points) chosen at random in each iteration
    1. select a random sample (or a mini-batch of samples) from data
    2. calculate the gradient on this sample
    3. update the model parameters
    4. repeat

$$Q(\theta) = \frac{1}{n} \sum_{1}^{n} (y_i - \theta x_i)^2$$

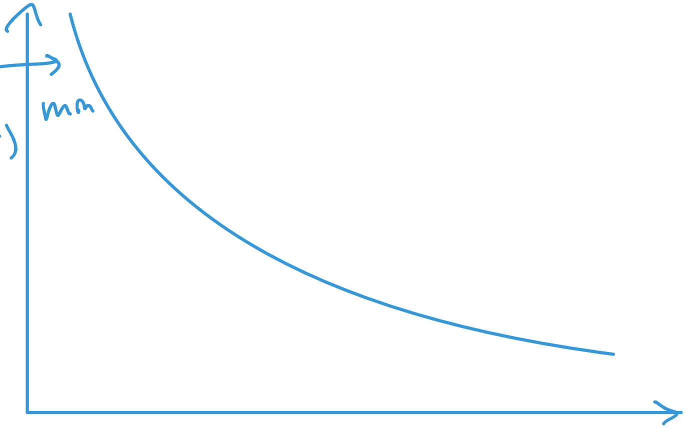
([illegible]):

$$\nabla Q(\theta) = \frac{1}{n} \sum_{1}^{n} 2x_i (y_i - \theta x_i)$$

(SDG):

$$\nabla Q(\theta) = -2x(y - \theta x)$$

where (x.y) is random



SGD: formula

$(t+1)$-th iteration:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \cdot \nabla_\theta Q(\theta^{(t)}; x^{(i)}; y^{(i)})$$

- θ represents the model parameters
- α is the learning rate
- $\nabla_\theta Q(\theta; x^{(i)}; y^{(i)})$ is the gradient of the loss function w.r.t. θ, evaluated at a random sample $(x^{(i)}, y^{(i)})$

Mini-batch SGD

$(t+1)$-th iteration:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \cdot \nabla_\theta Q(\theta^{(t)}; X^{(i:i+n)}, Y^{(i:i+n)})$$

- θ represents the model parameters
- α is the learning rate
- $\nabla_\theta Q(\theta; X^{(i:i+n)}, Y^{(i:i+n)})$ is the gradient of the loss function w.r.t. θ computed over a mini-batch of data points
- $X^{(i:i+n)}$ and $Y^{(i:i+n)}$ are features/labels of the mini-batch, resp., starting from the i-th data point to the $(i+n)$-th data point

Why SGD (for deep learning)?

- **handling big data** typical for deep learning applications
  - SGD does not require the entire dataset to be loaded into memory or used for each parameter update
  - instead, SGD updates parameters using only a small subset of data at a time
- **faster convergence**
  - updates the model parameters more frequently
    $\Rightarrow$ converges faster
  - crucial when training time is critical (e.g., online learning)

Why SGD (for deep learning)?

- **flexibility with mini-batch sizes**: leads to a balance between the computational efficiency of true stochastic updates (using very small batches) and the stability of gradient estimates (using larger batches)
- **generalization and regularization**: the stochastic nature of SGD, where each update is based on a subset of the data, can have a regularizing effect
  - potentially leads to better generalization
- **ability to escape local minima** due to stochasticity, especially with complex, non-convex optimizations typical of deep neural networks