KU x depa
MACHINE LEARNING
Training Course 2022

# Linear Regression

DR. SETHAVIDH GERTPHOL

KU KASETSART UNIVERSITY depa

# Today's Outline

- Linear Regression
- Polynomial Features
- Gradient Descent

# Linear Regression

- Regression: predict the numerical output using input features
- Linear Regression: use this equation for prediction

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- $\hat{y}$ is the predicted value.
- $n$ is the number of features.
- $x_i$ is the $i^{\text{th}}$ feature value.
- $\theta_j$ is the $j^{\text{th}}$ model parameter,

- Note that there are n features, but n+1 model parameters
  - $\theta_0$ is bias, $\theta_i$ is coefficient of $x_i$
- When there is only one feature, the equation has the same form as y = b + ax

# Finding the best thetas (model parameters)

- Which set of thetas is best?
- We must define a cost function and the best thetas are the ones that minimizes the cost
- Mean Square Error is a good cost function
  - Correlates with a metric used to evaluation regression model
  - Easy to optimize

Model parameters

Actual value of that sample

$$\mathrm{MSE}\left(\mathbf{X}, h_{\boldsymbol{\theta}}\right) = \frac{1}{m} \sum_{i=1}^{m} \left( \boldsymbol{\theta}^{\intercal} \mathbf{x}^{(i)} - y^{(i)} \right)^{2}$$

m is the total number of samples

Prediction of one sample

# Solving for thetas

- There is a closed-form solution for MSE cost function called the Normal Equation

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

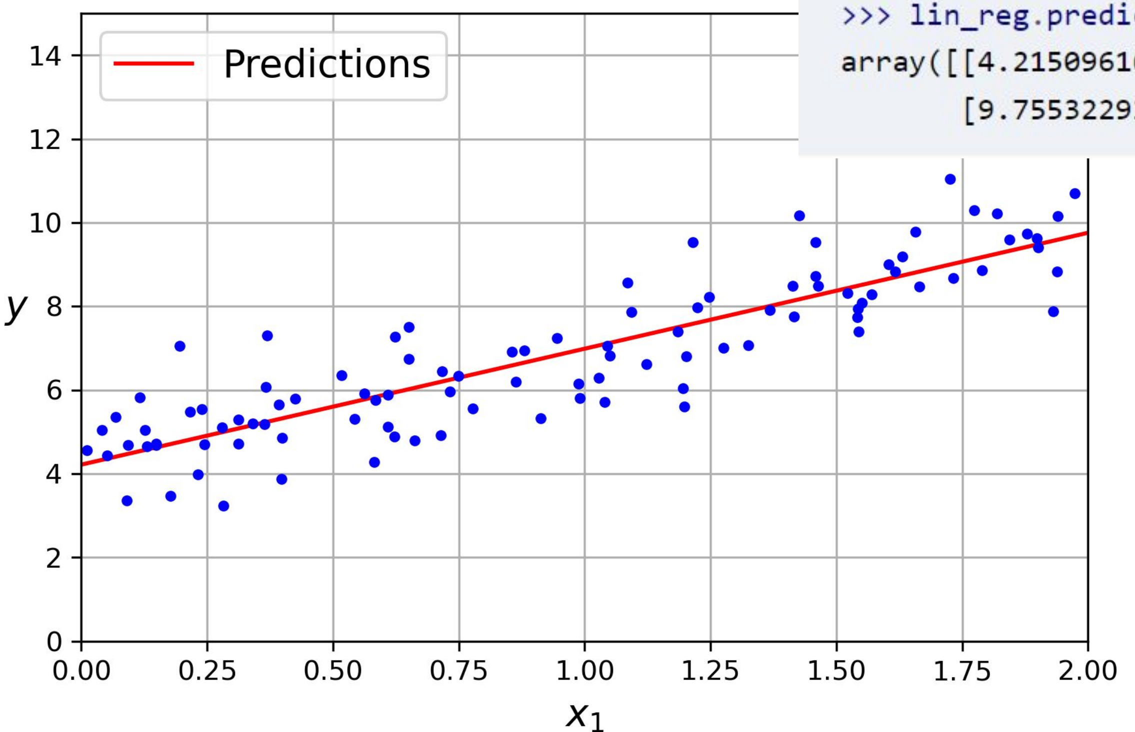
- Sklearn uses SVD (Singular Value Decomposition) to calculate the pseudoinverse matrix
- This is a calculation, so there is no hyperparameter to tune
- No need for feature scaling
- Use one-hot encoding to transform categorical features

# In Sklearn

import

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([4.21509616]), array([[2.77011339]]))
>>> lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

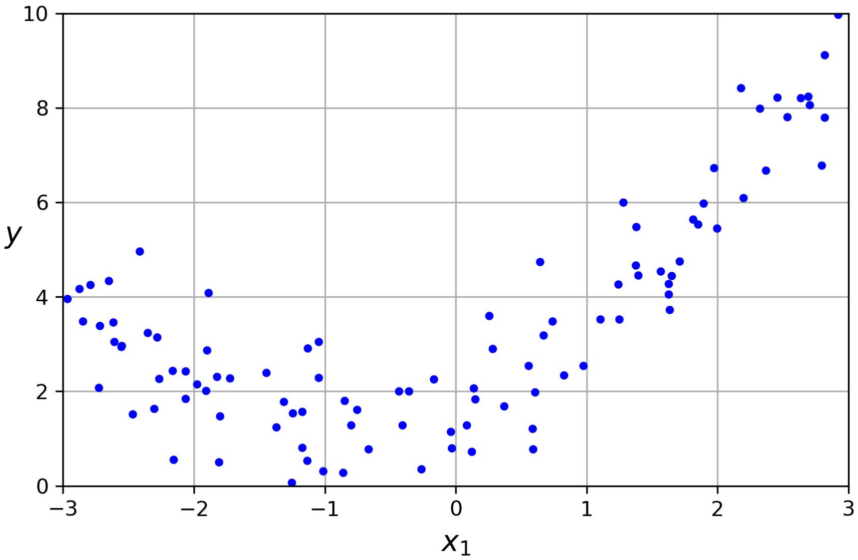
$\theta_0$is in intercept_,
other $\theta_i$ are in coef_




Note that LinearRegression()
adds the bias tutomatically

# Polynomial Regression

- What if the data has underlying polynomial terms?
- Can add polynomial terms to feature matrix and use LinearRegression to fit the new set of features



# PolynomialFeatures

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

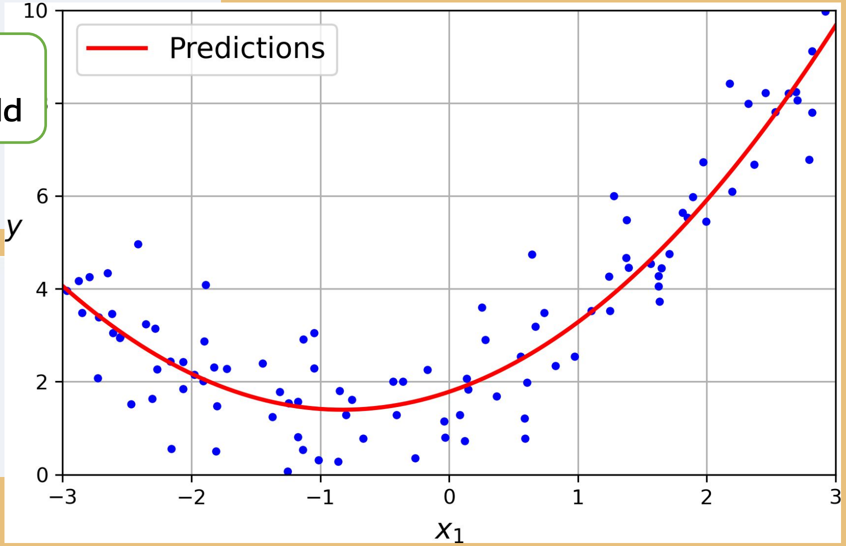
import

No need to add bias because it will be added by LinearRegression

Transform X

Polynomial degrees to add

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([1.78134581]), array([[0.93366893, 0.56456263]]))
```

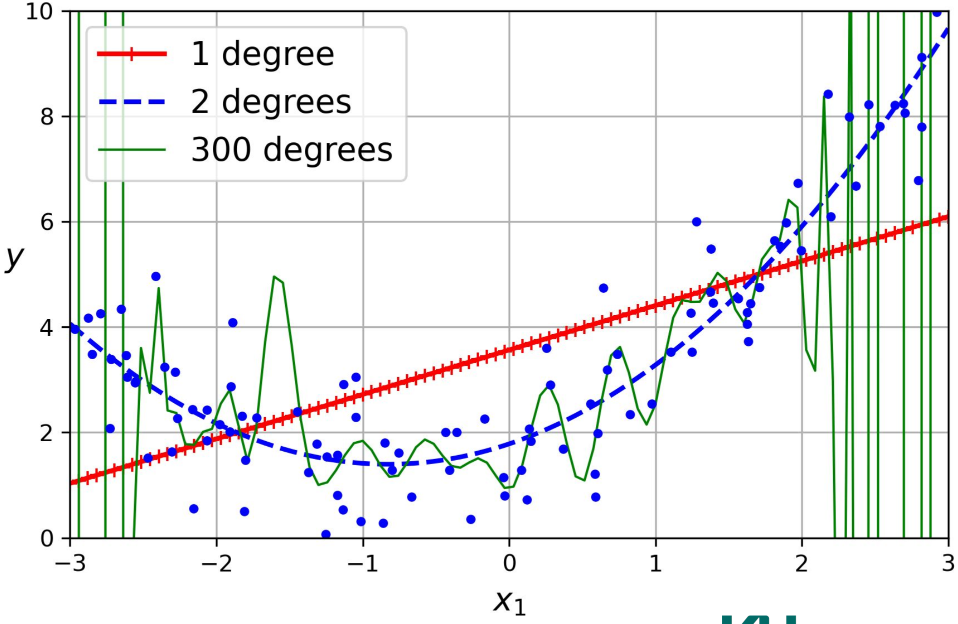



Important: PolynomialFeatures will add all terms from lower degrees and also combination of terms too

x, y with degree=3 will add: $x^3, y^3, x^2, y^2, x^3, xy, xy^2, x^2y$

# What value of degree is best?

- Low degree: model is not flexible enough, may have high bias
  - Bias is a wrong assumption, such as data is linear but in fact it is polynomial
  - Usually leads to underfitting
- High degree: model is too complex, have high variance
  - Varaince is sensitivity to small variations in training data
  - Usually leads to overfitting
- Bias-Variance Trade-off
- Use cross-validation to find best value

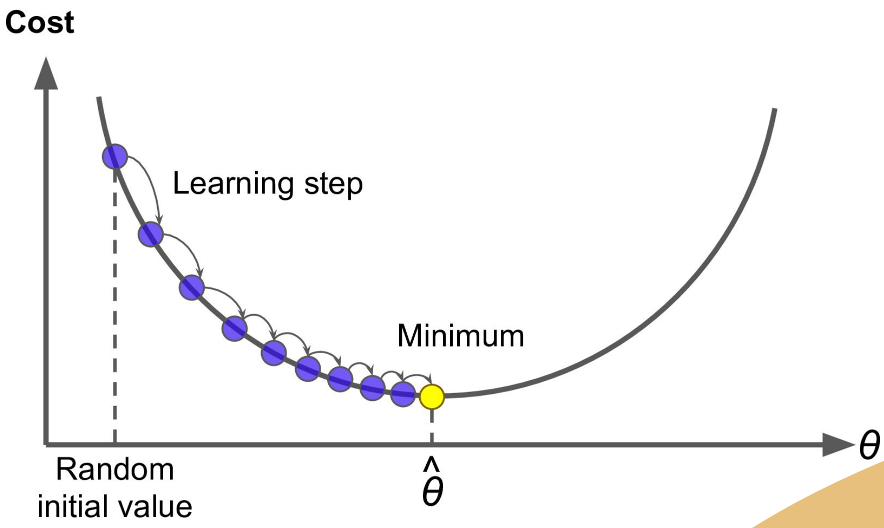


# Evaluation

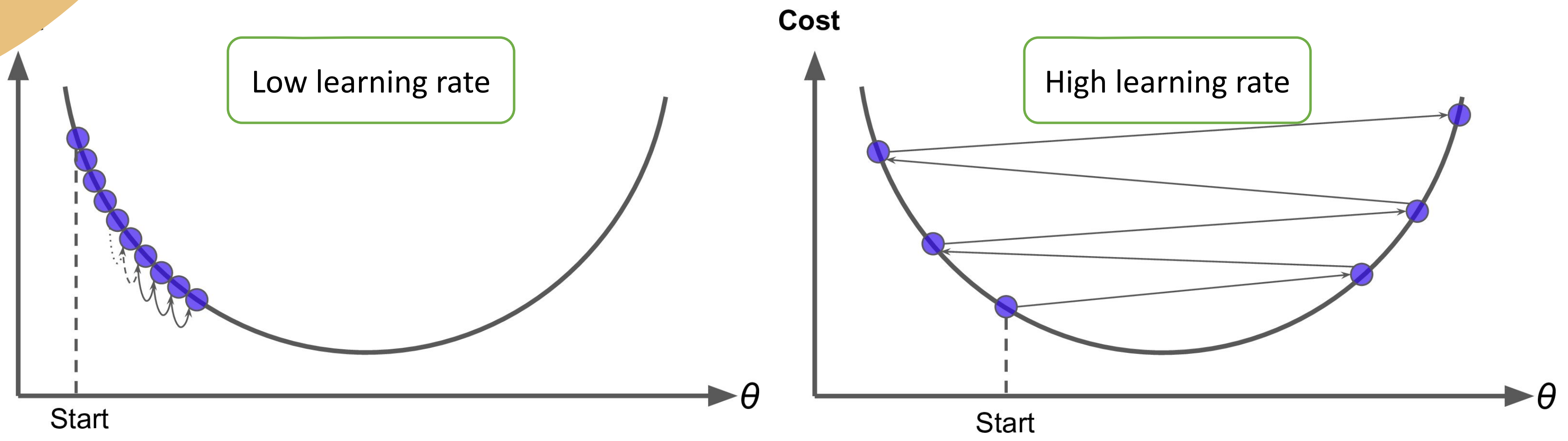
- Metrics that can be used (in sklearn.metrics)
  - mean_absolute_error
  - mean_squared_error
  - mean_absolute_percentage_error
  - r2_score

# Gradient Descent

- Normal Equation and SVD can take a long time with large number of features
- Gradient Descent
  - Optimization technique
  - Start with random values of thetas and adjust them step-by-step to reduce cost function
  - Evaluating gradient (slope) of cost function at current thetas and follow the greatest descent (steepest slope downward)
  - When gradient is zero, we have found thetas that give minimum cost



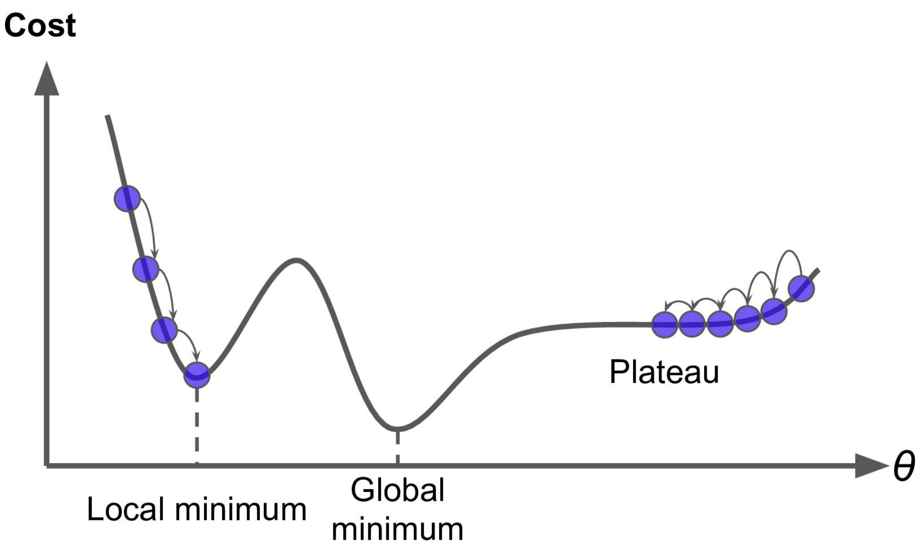
# Learning rate




- How much we adjust thetas during each step of gradient descent
- Trade-off:
  - Low learning rate: small step, take too long to find minimum
  - High learning rate: large step, may go over minimum and diverge

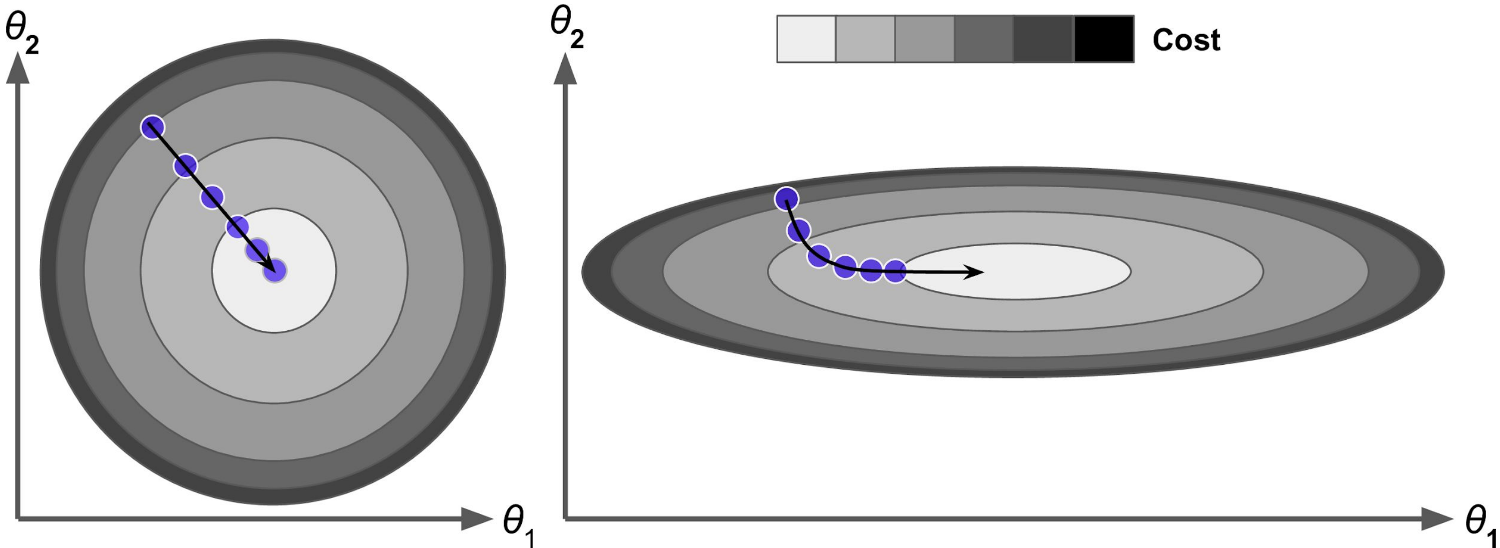
# Local and Global minima

- Not all cost functions have nice smooth graph with one minimum
- Many have several minima, ridges, peaks, plateau
- Gradient descent may stop at local minimum instead of global one
- May not be able to escape local minimum due to peak next to it
- May take a long time to cross plateau



# Feature scaling

- MSE cost function has a nice bowl shape with one minimum
- But if features are on different scale, may take long time to converge
- Should use feature scaling (e.g., StandardScaler) with Gradient Descent



# Batch Gradient Descent

- Calculate partial derivative of each theta on the cost function to find gradient
- Use gradient to adjust thetas
- Fast to calculate even with large number of features (thetas)
- Slow with large number of samples

$$\mathrm{MSE}\left(\mathbf{X}, h_{\boldsymbol{\theta}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)^2$$

Cost function (MSE)

$$\frac{\partial}{\partial \theta_j}\mathrm{MSE}(\boldsymbol{\theta}) = \frac{2}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^{\intercal}\mathbf{x}^{(i)} - y^{(i)}\right)x_j^{(i)}$$

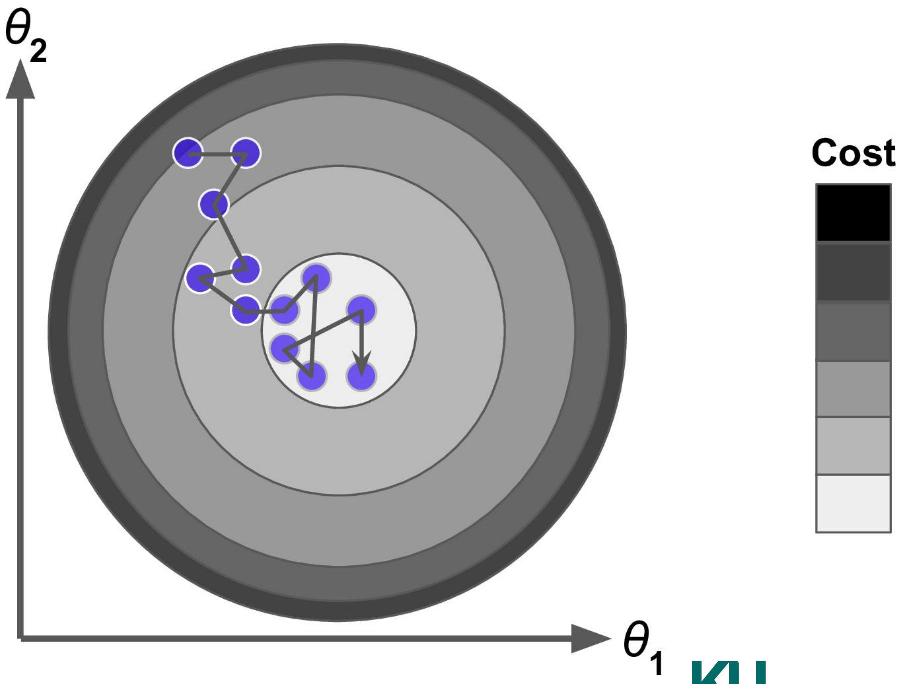
Partial derivative of $\theta_j$

$$\nabla_{\boldsymbol{\theta}}\,\mathrm{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0}\mathrm{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1}\mathrm{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n}\mathrm{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m}\mathbf{X}^{\intercal}\left(\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\right)$$

Matrix form of partial derivative of all thetas

# Stochastic Gradient Descent

- Randomly pick 1 sample, and calculate gradient based on that sample
- Very fast even with large number of samples
- Cost value may bounce around or even go up during one iteration
- Over time it will go near the minimum value, but never stops



# Stopping SGD (Stochastic Gradient Descent)

- Set the number of epochs to optimize
  - epoch: 1 round of of m iterations where m is the number of samples
- Reduce learning rate in later epochs/iterations
  - Learning schedule: a function to adjust learning rate
- Change in cost:
  - If cost change is less than given threshold over later iterations

# SGD in Sklearn

import

Regularization (not taught)

Starting learning rate (default schedule)

iterations

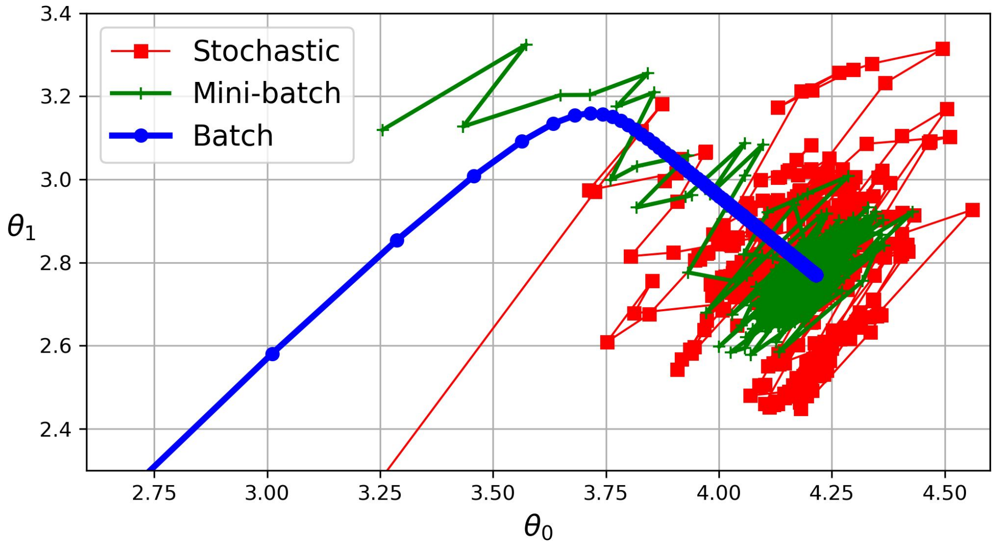
```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                       n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel())  # y.ravel() because fit() expects 1D targets
```

Stop when reach this number of iterations with less than tol change

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.21278812]), array([2.77270267]))
```

# Mini-Batch Gradient Descent

- Calculation of gradient
  - Batch: all samples
  - Stochastic: 1 random sample
  - Mini-batch: random subset of samples
- Progression of thetas in these 3 methods



# Things not taught

- Learning Curve: a technique to evaluate overfitting or underfitting using training and validation result (cross-validation) over iterations
- Regularization: techniques to reduce overfitting by introducing penalty term to cost function
- Regularized Linear Model
  - Ridge Regression
  - Lasso Regression
  - Elastic Net

# Reference


- Aurélien Géron, "Hands-On Machine Learning with Scikit-Learn and TensorFlow", O'Reilly Media, Inc., March 2017.