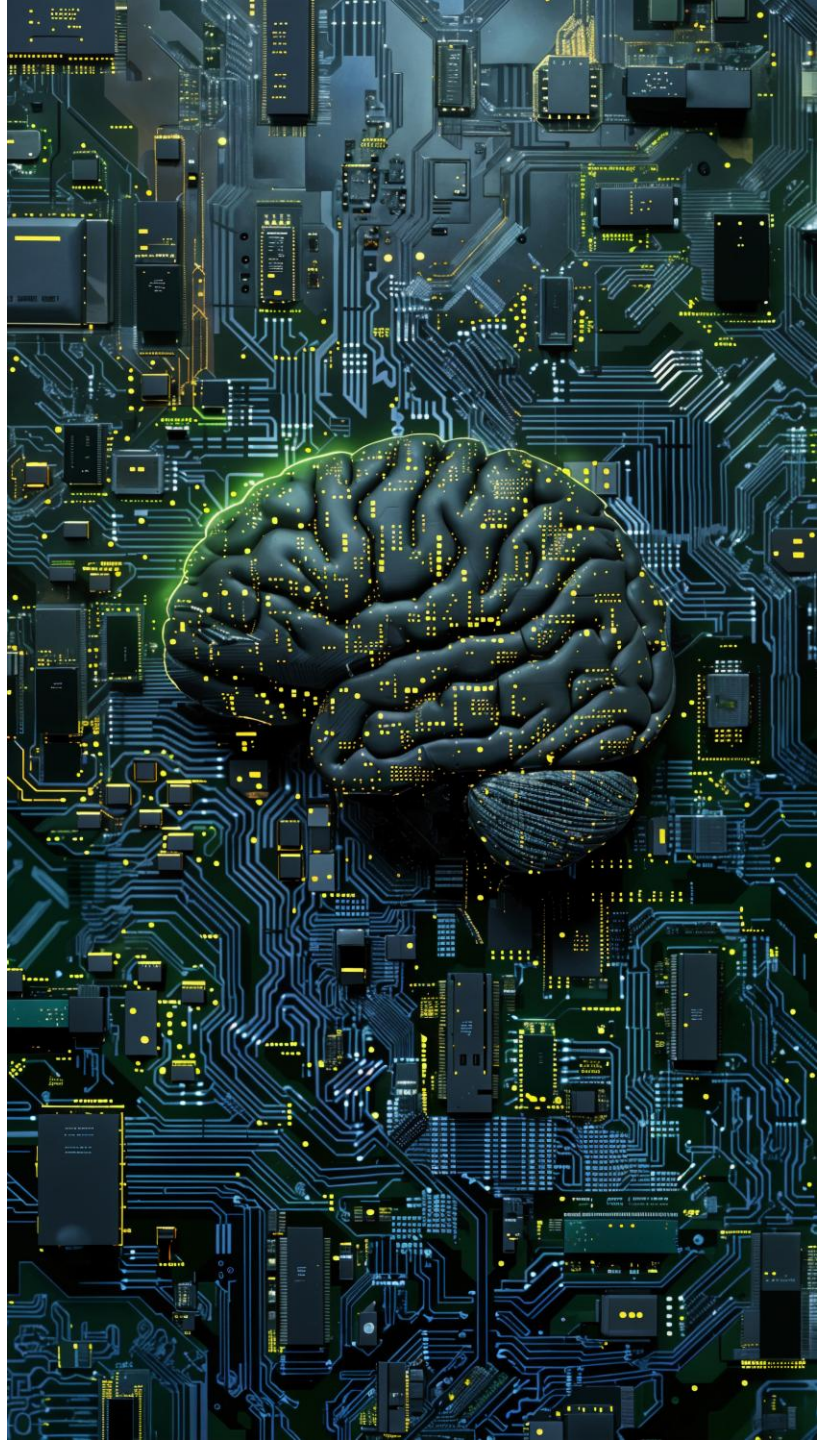
Machine Learning

Week 2 - Regression

SAXION
UNIVERSITY OF
APPLIED SCIENCES



Machine Learning

Machine Learning is the) field of study that gives computers the ability to learn without being explicitly programmed.

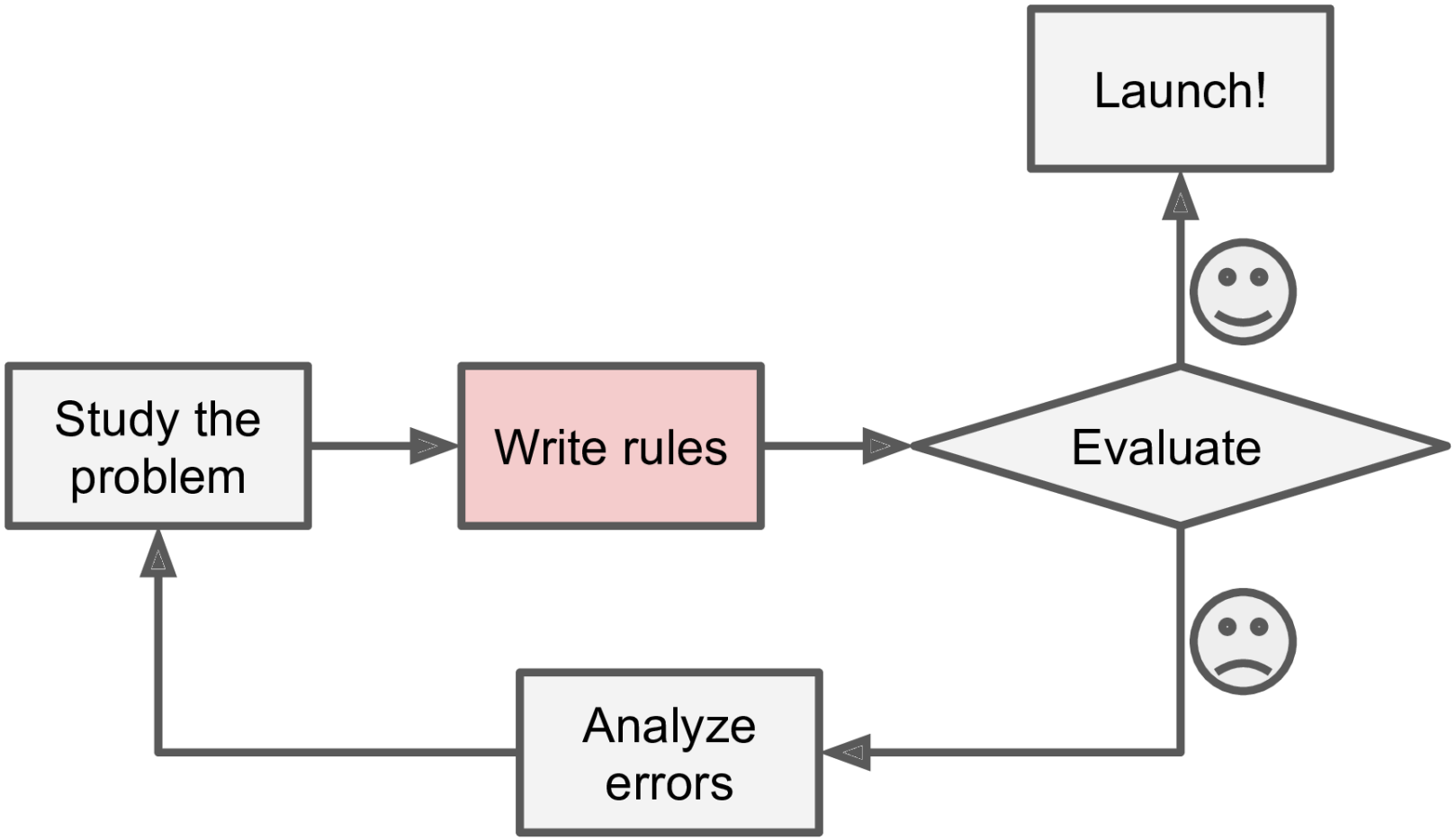
—Arthur Samuel, *1959*

Machine Learning

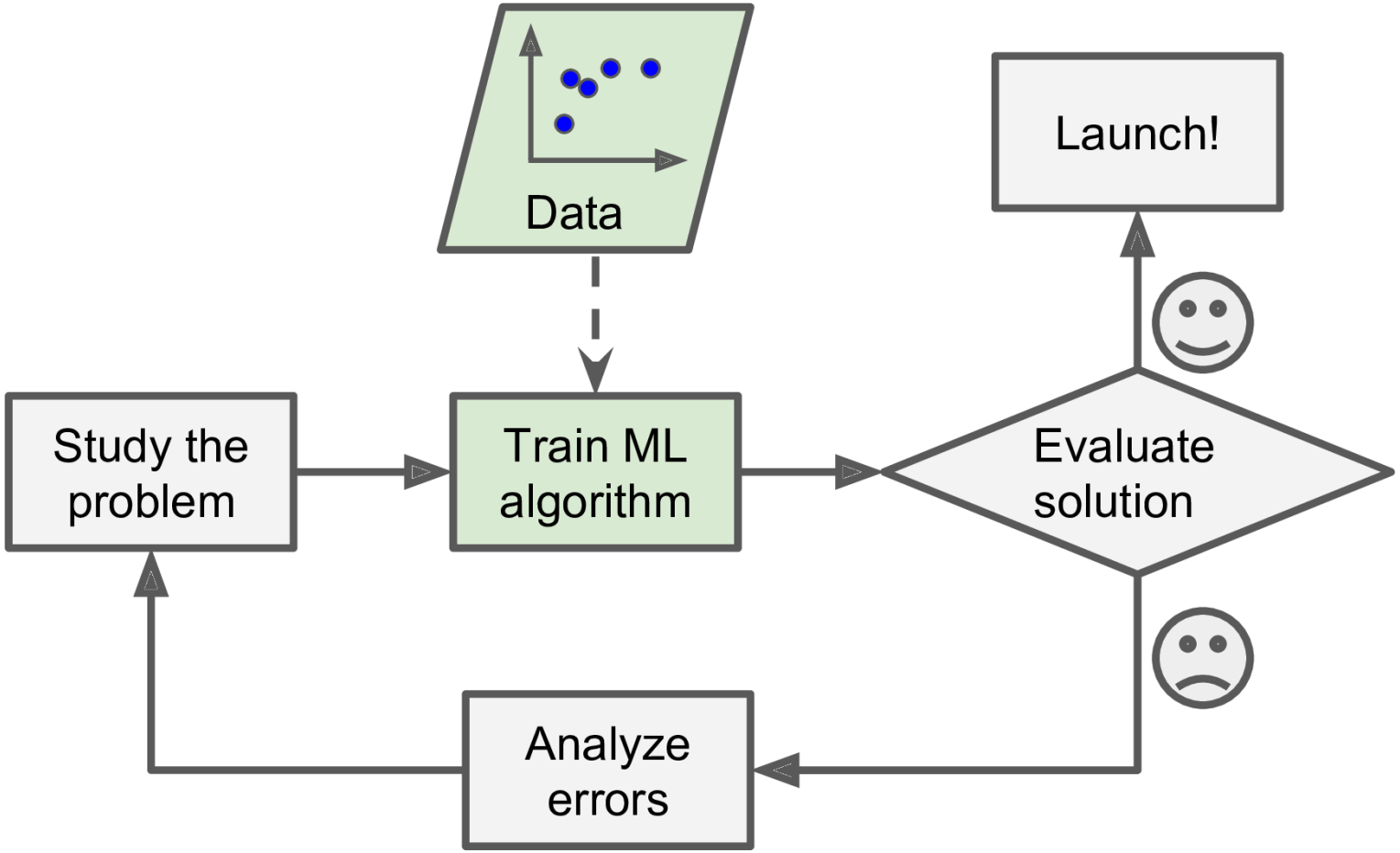
A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

—Tom Mitchell, 1997

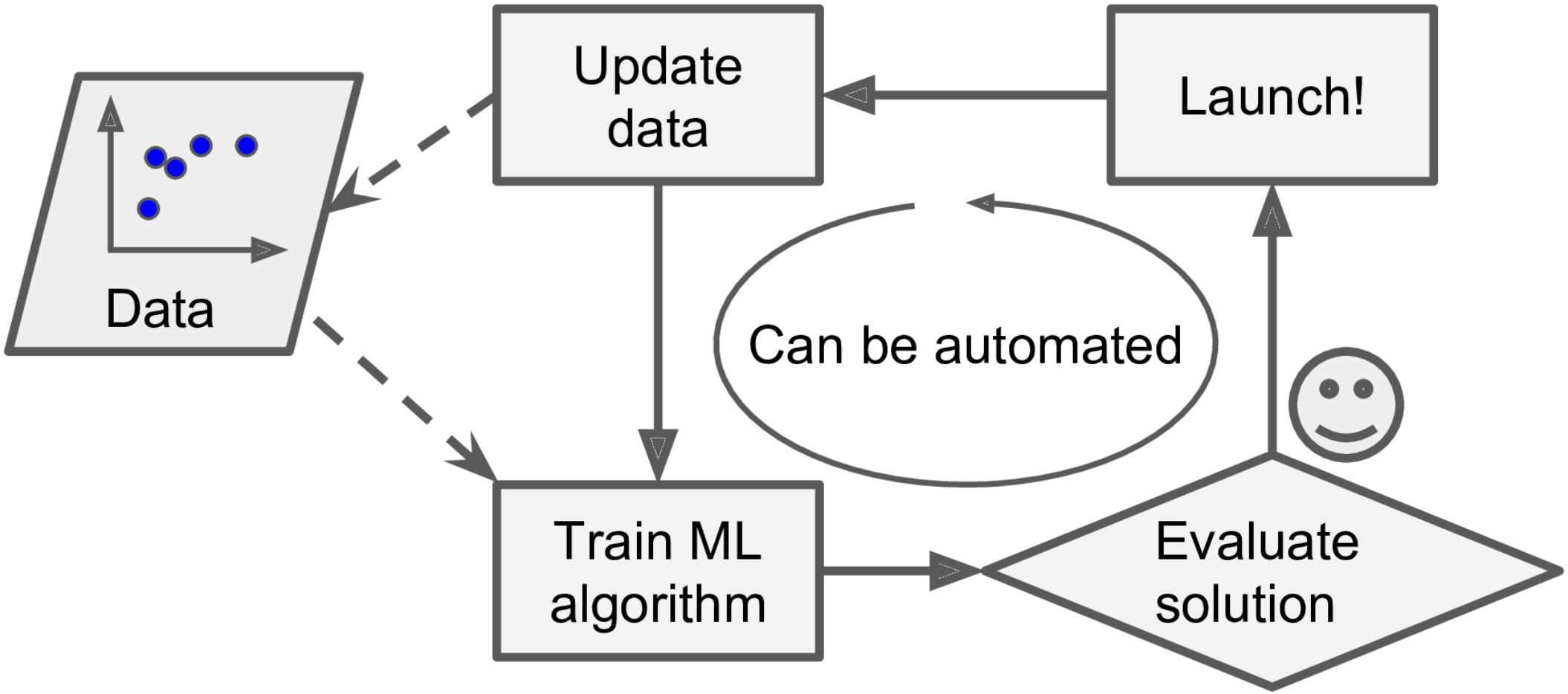
Traditional approach



ML approach


Data
Launch!
Study the problem
Train ML algorithm
Evaluate solution
Analyze errors

ML approach

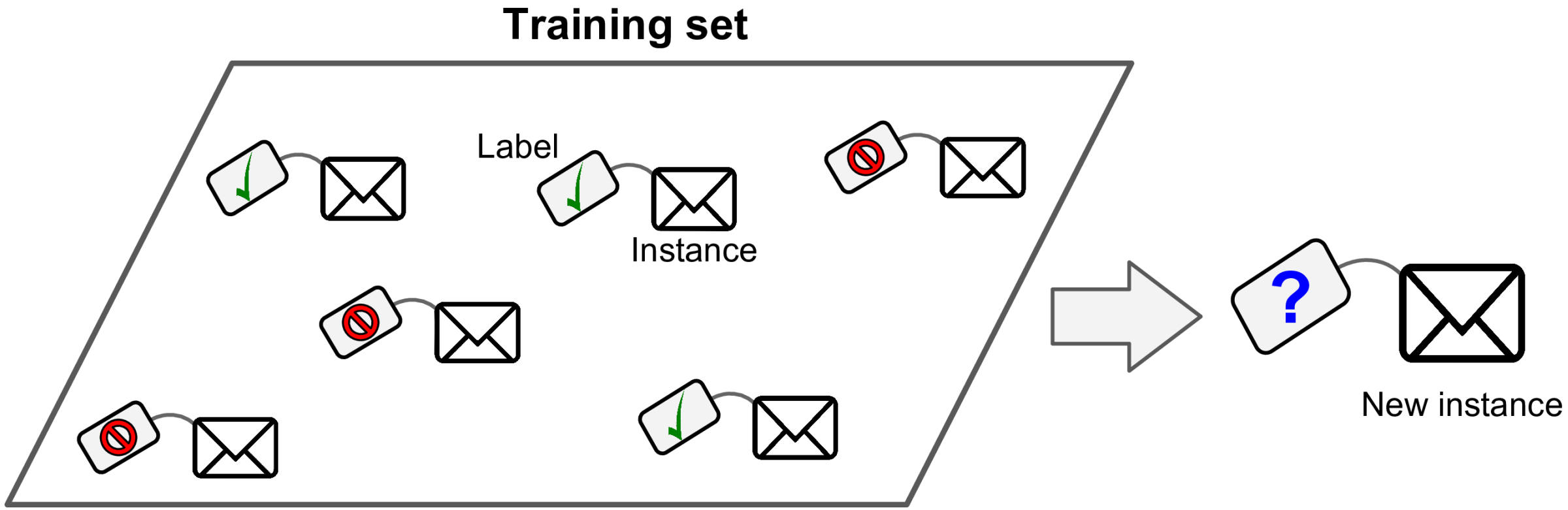

Update data
Launch!
Data
Can be automated
Train ML algorithm
Evaluate solution

ML, what is it?

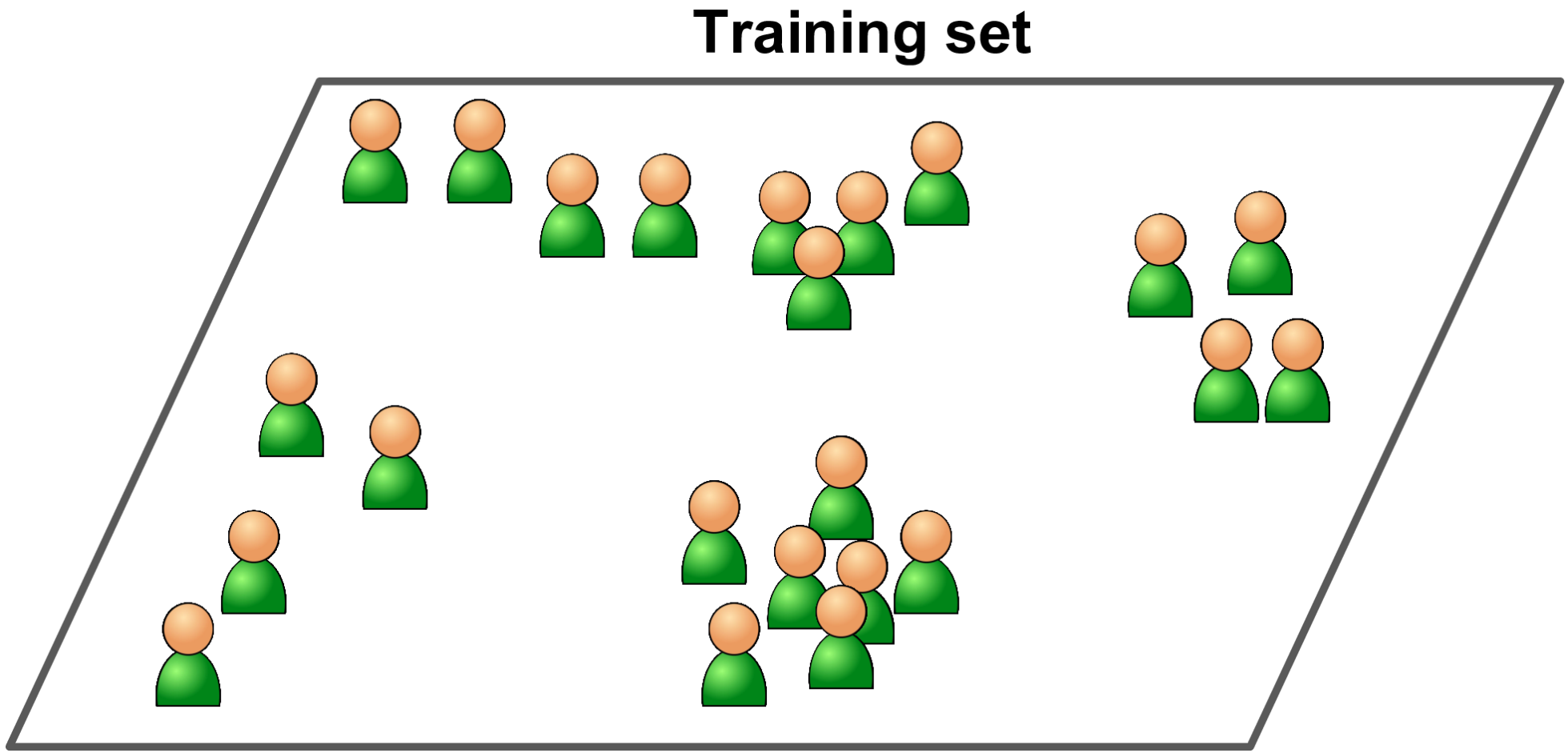
The main division of machine learning:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

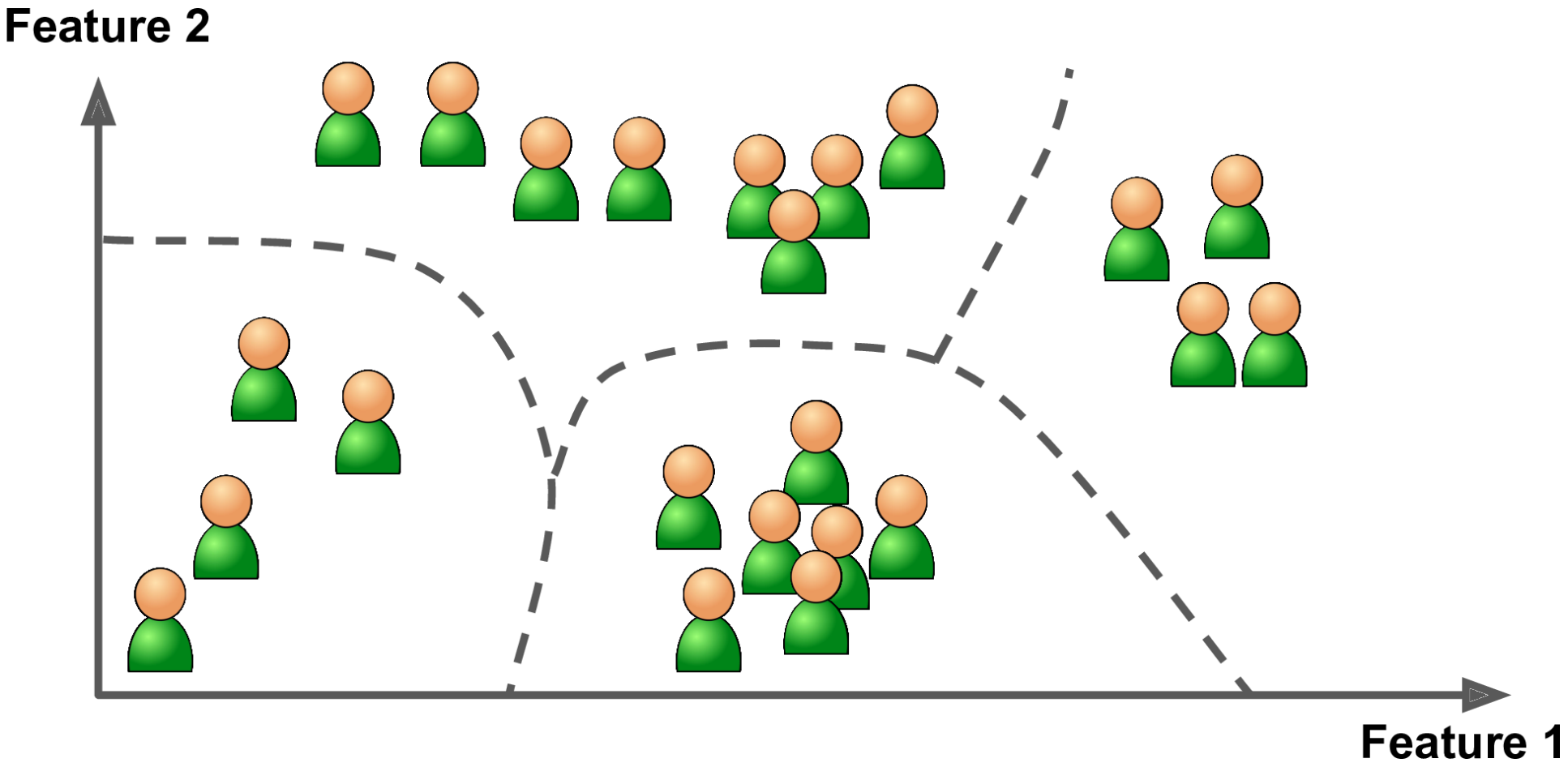
Supervised Learning



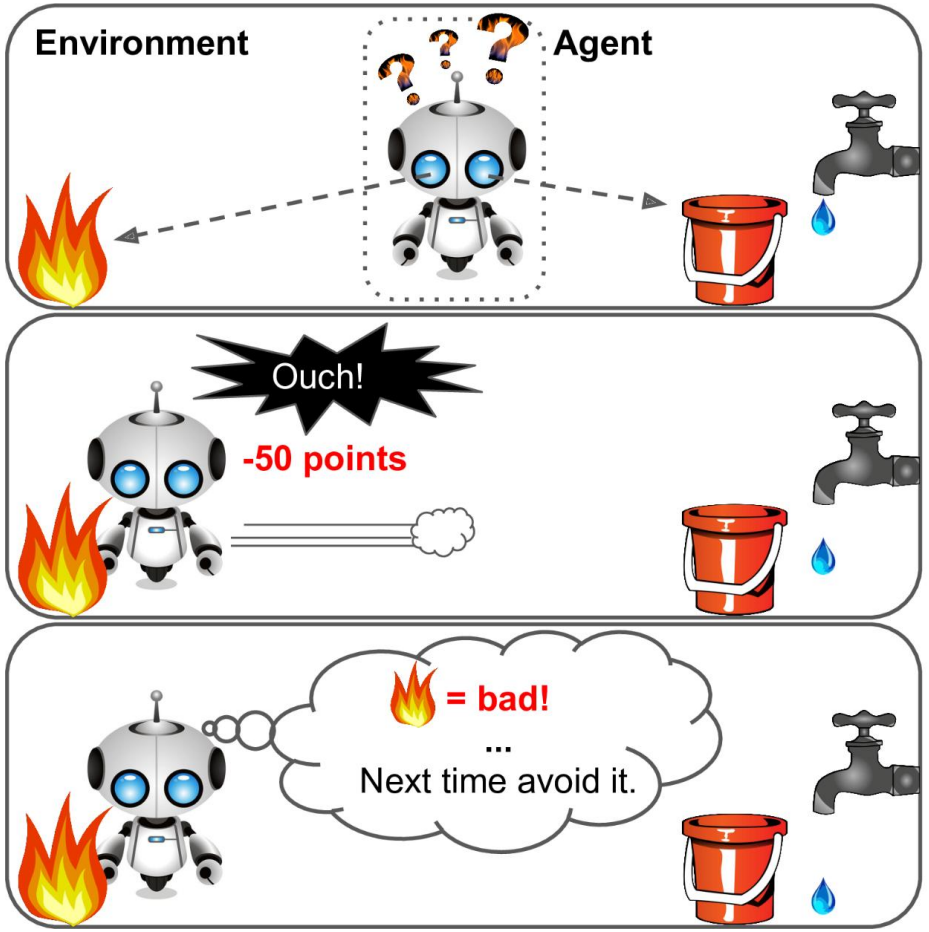
Unsupervised Learning



Unsupervised Learning



Reinforcement Learning

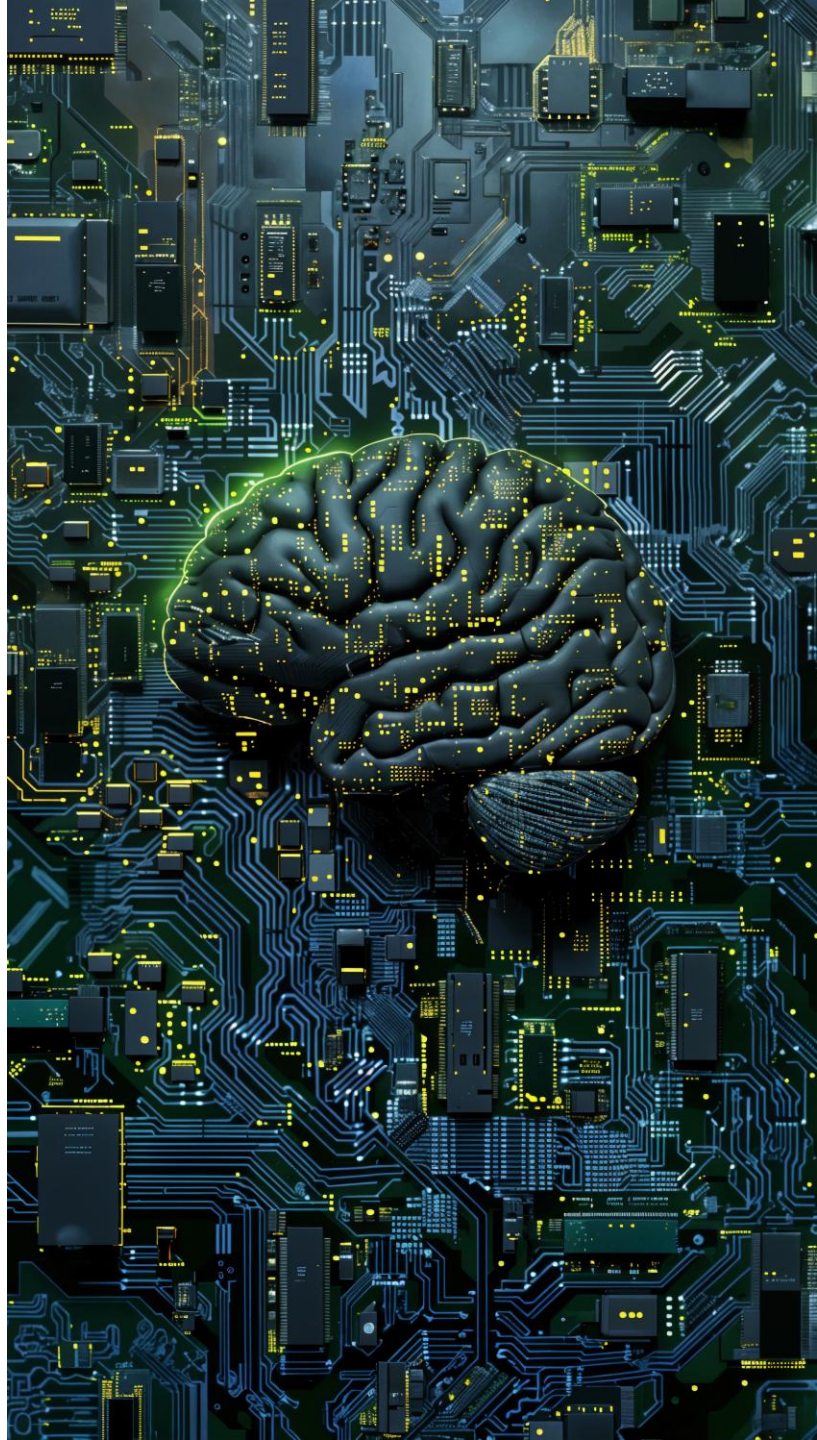


1. Observe
2. Select action using policy
3. Action!
4. Get reward or penalty
5. Update policy (learning step)
6. Iterate until an optimal policy is found

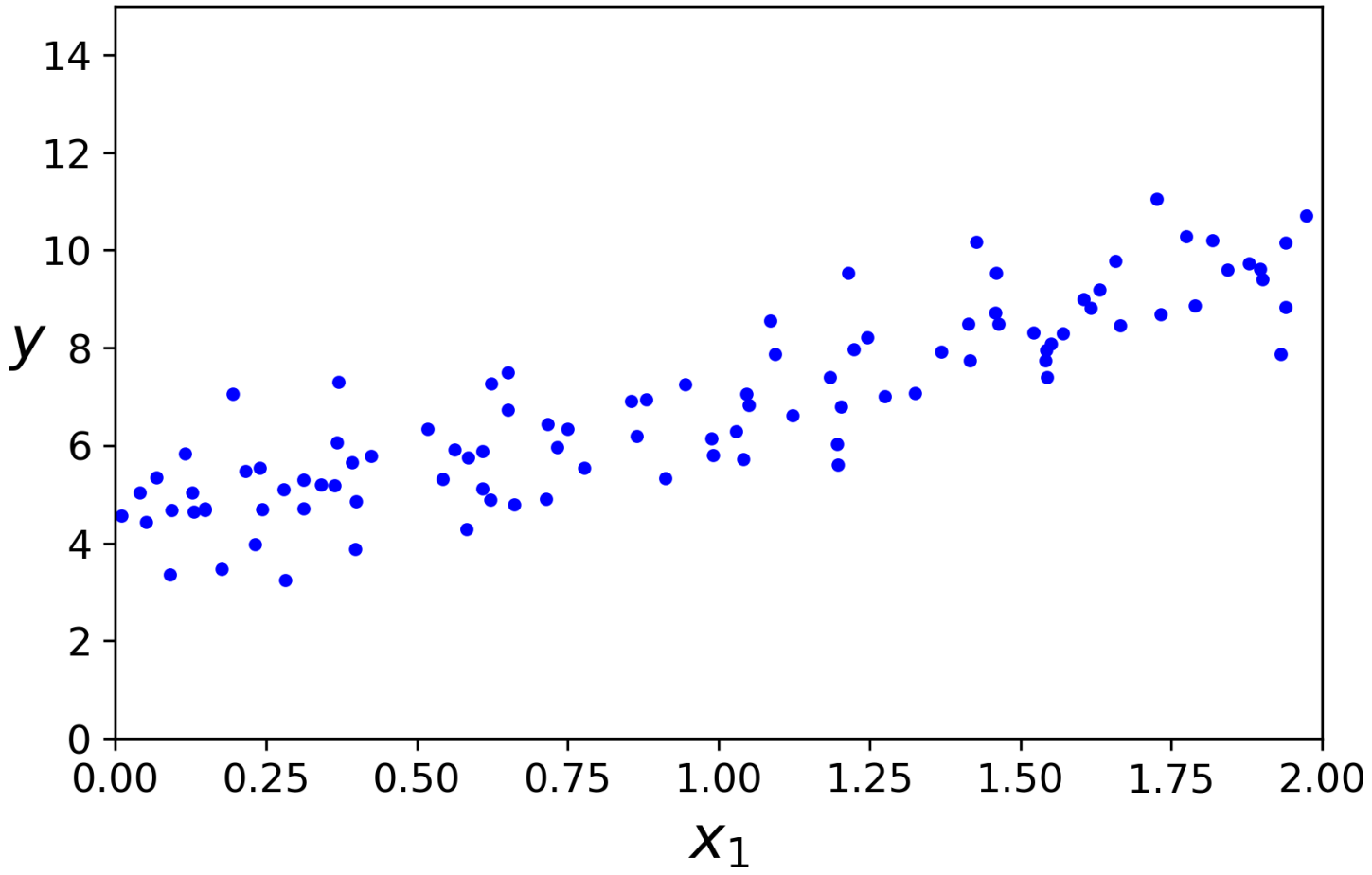
Supervised Learning

Linear regression, gradient descent, learning

SAXION
UNIVERSITY OF
APPLIED SCIENCES



Linear Regression

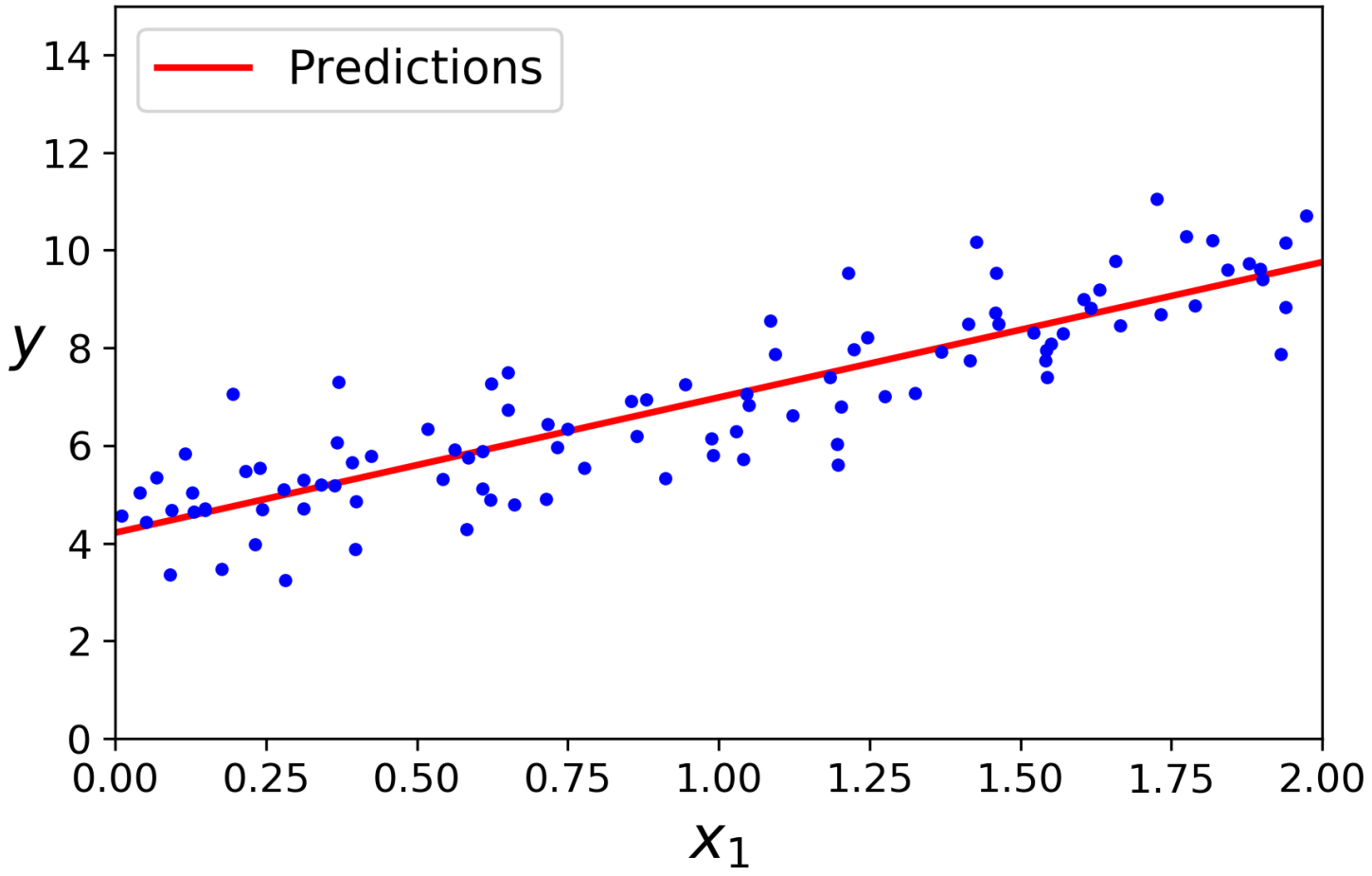


- Observation

x_1 Feature

$$y = ax_1 + b$$

Linear Regression



Model parameters

$$\hat{y} = \theta_1 x_1 + \theta_0$$

Predicted value

Linear Regression

General linear regression model with **n** features:

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

$\hat{y}$ – predicted value
θ_i – *i-th* model parameter
x_i – *i-th* feature

Linear Regression

In vector form (**for a single observation with features $x_1 \ldots x_n$**):

$$\hat{y} = h_\theta(X) = \Theta \cdot X$$

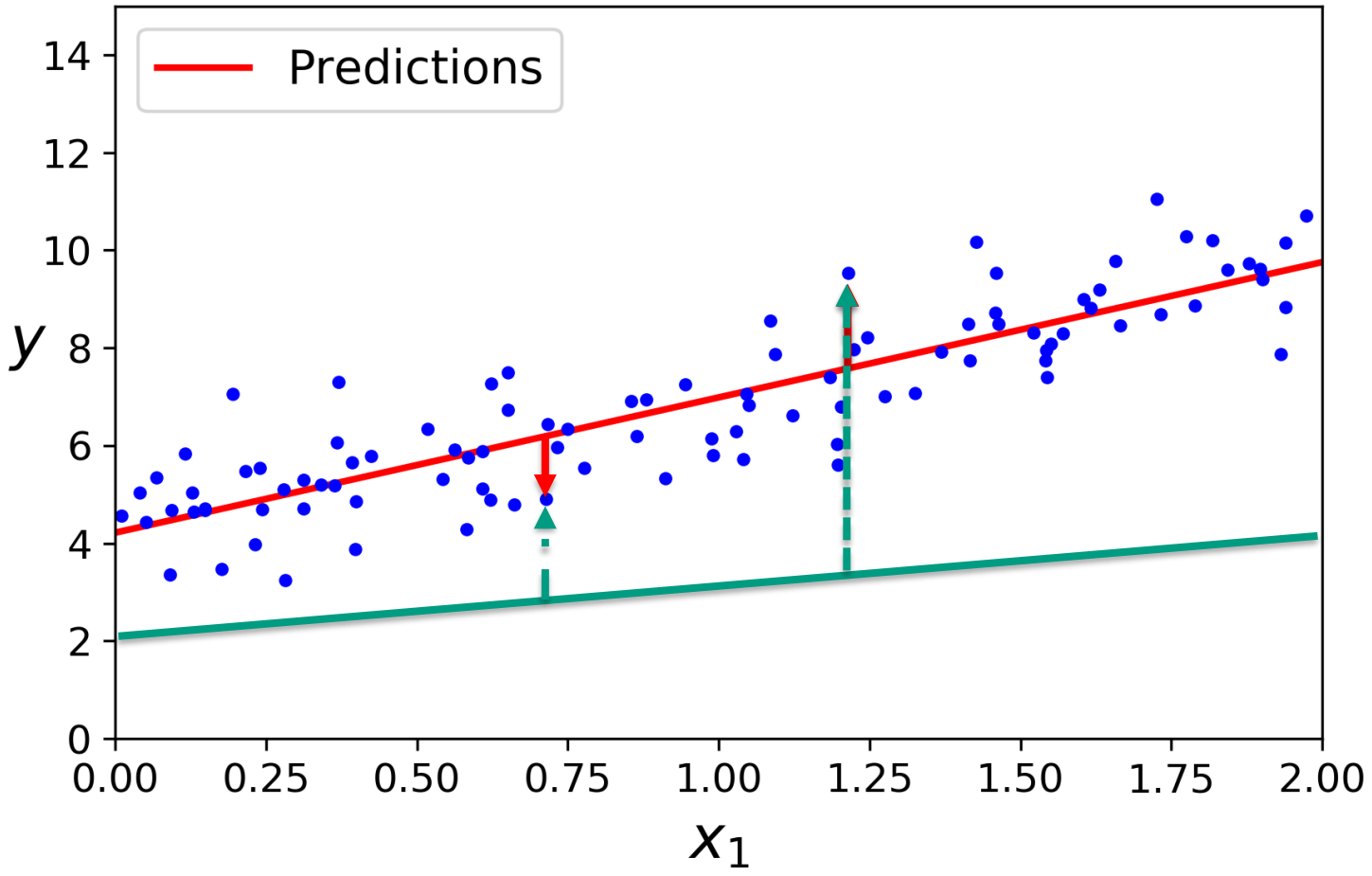
h_θ - hypothesis function

Feature vector

$$\Theta = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \ldots \\ \theta_n \end{pmatrix} \quad X = \begin{pmatrix} 1 \\ x_1 \\ \ldots \\ x_n \end{pmatrix}$$

Parameter vector

LR: fitting the model

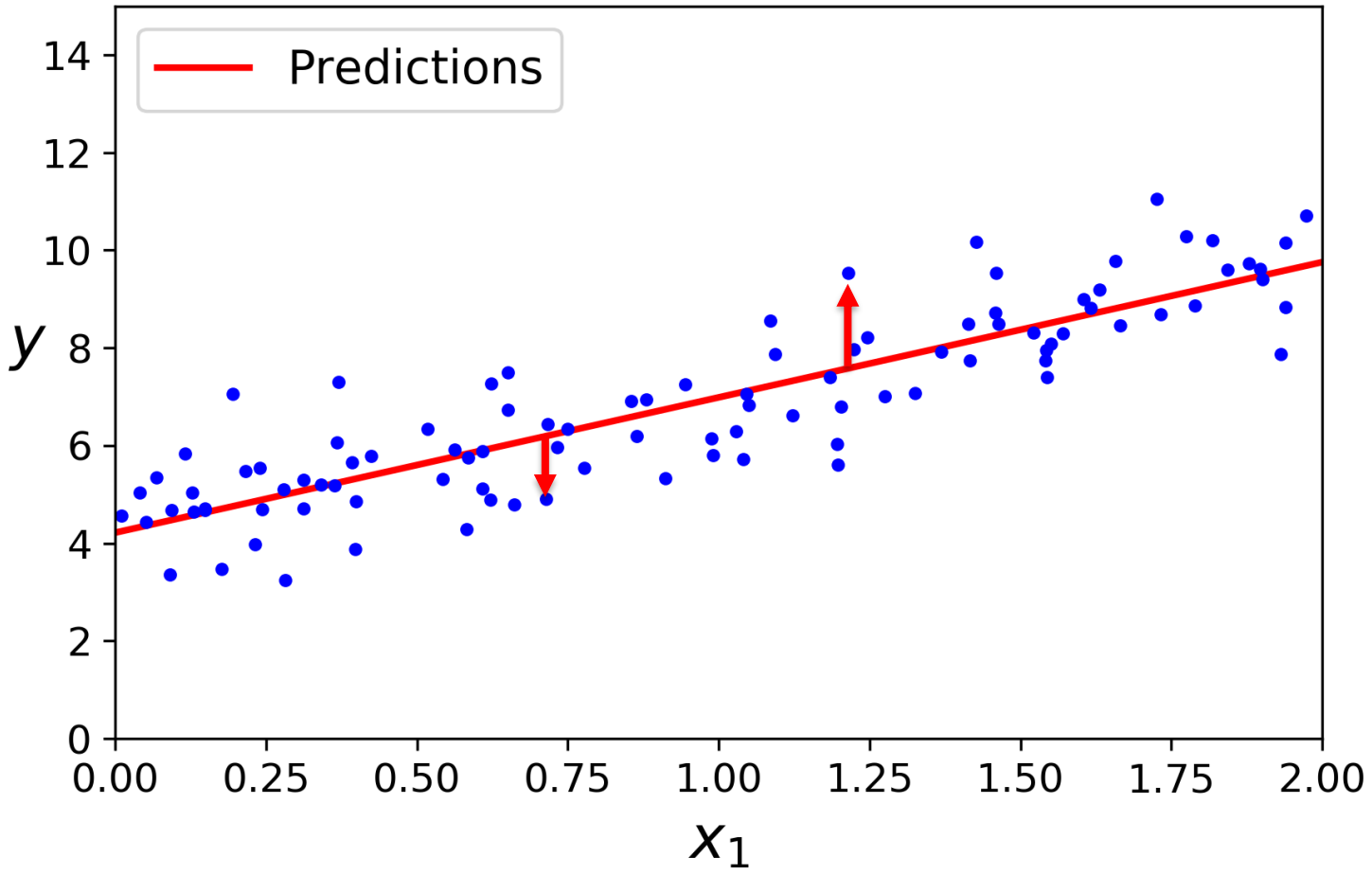


$$\hat{y} = \theta_1 x_1 + \theta_0$$

$$(\hat{y}^{(j)} - y^{(j)})$$

Predicted value **Observed value**

LR: fitting the model

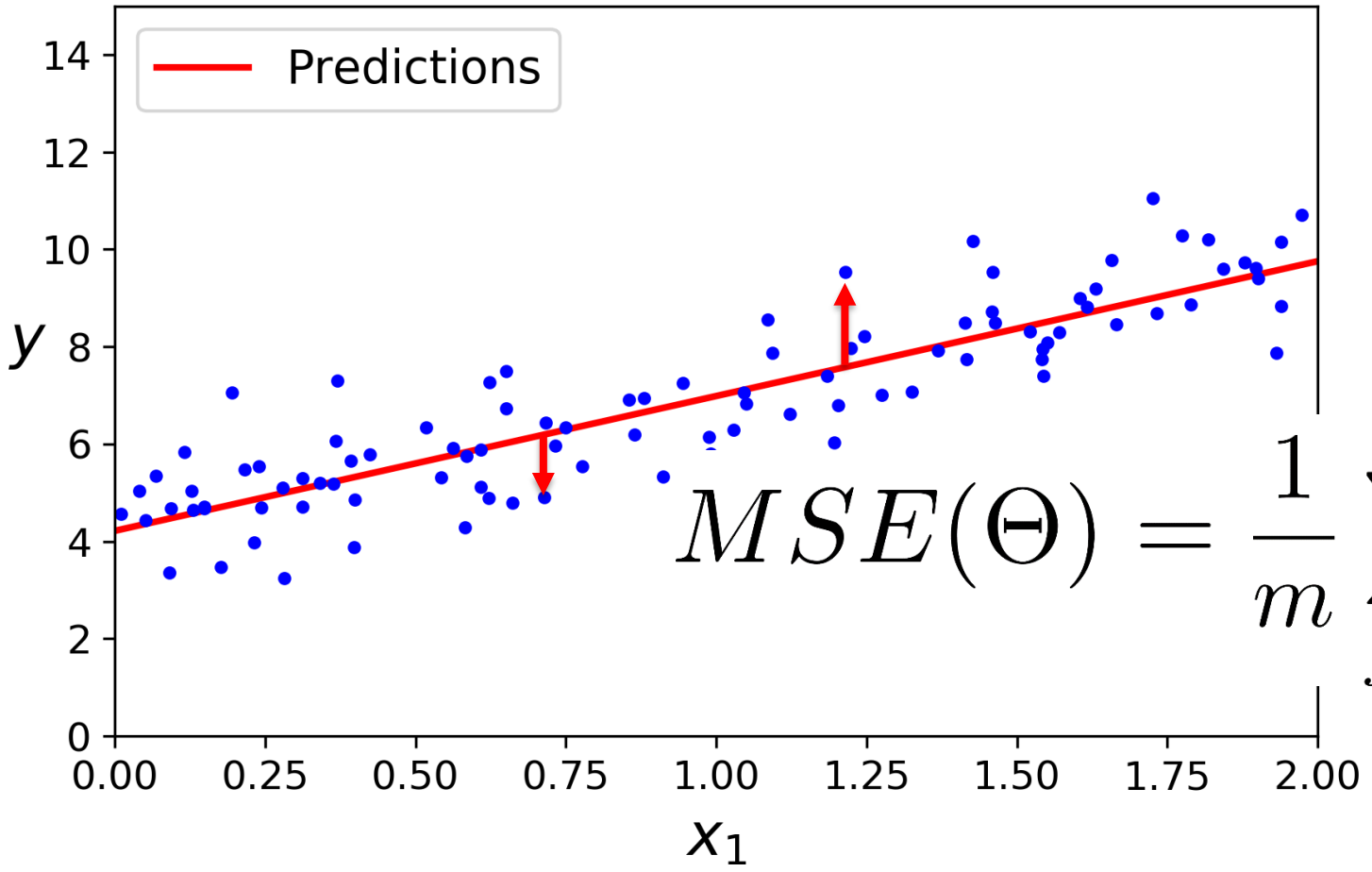


$$\hat{y} = \theta_1 x_1 + \theta_0$$

$$(\hat{y}^{(j)} - y^{(j)})^2$$

Predicted value **Observed value**

LR: Mean Squared Error (MSE)



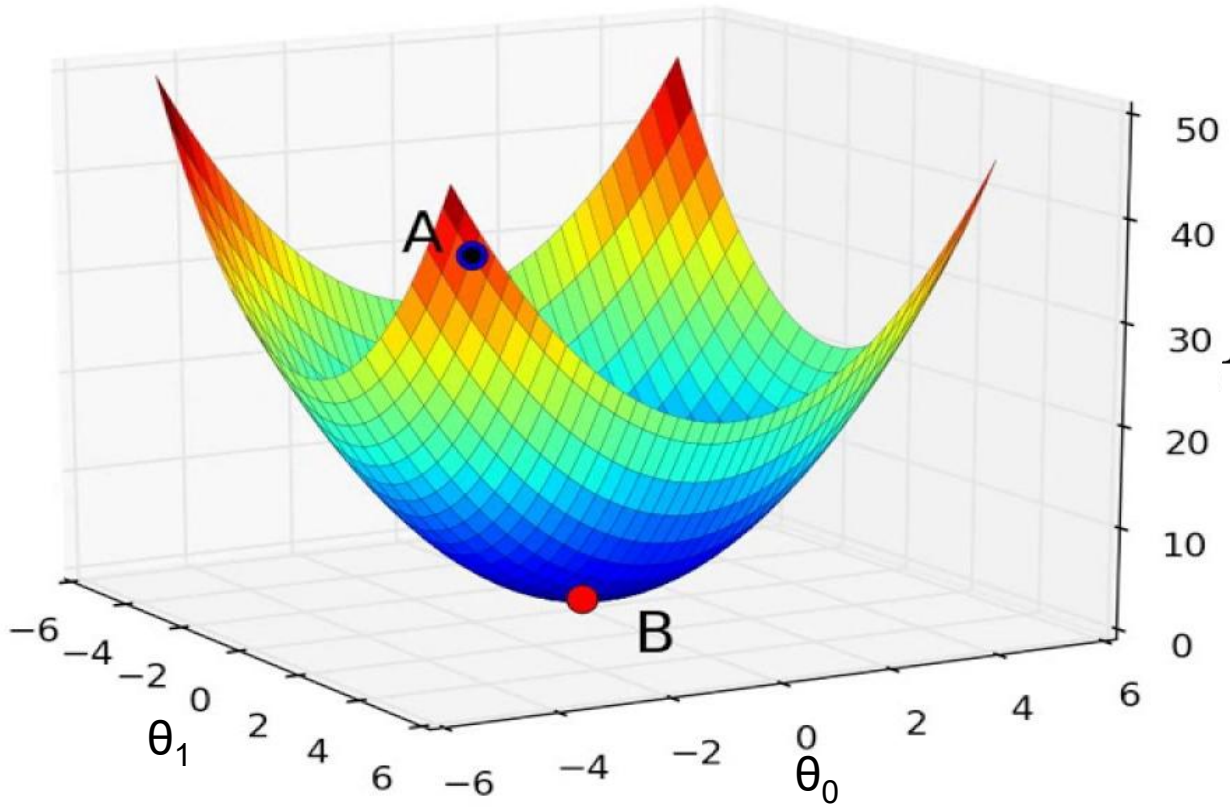
$$\hat{y} = \theta_1 x_1 + \theta_0$$

$$MSE(\Theta) = \frac{1}{m} \sum_{j=1}^{m} \left(\hat{y}^{(j)} - y^{(j)} \right)^2$$

Predicted value

Observed value

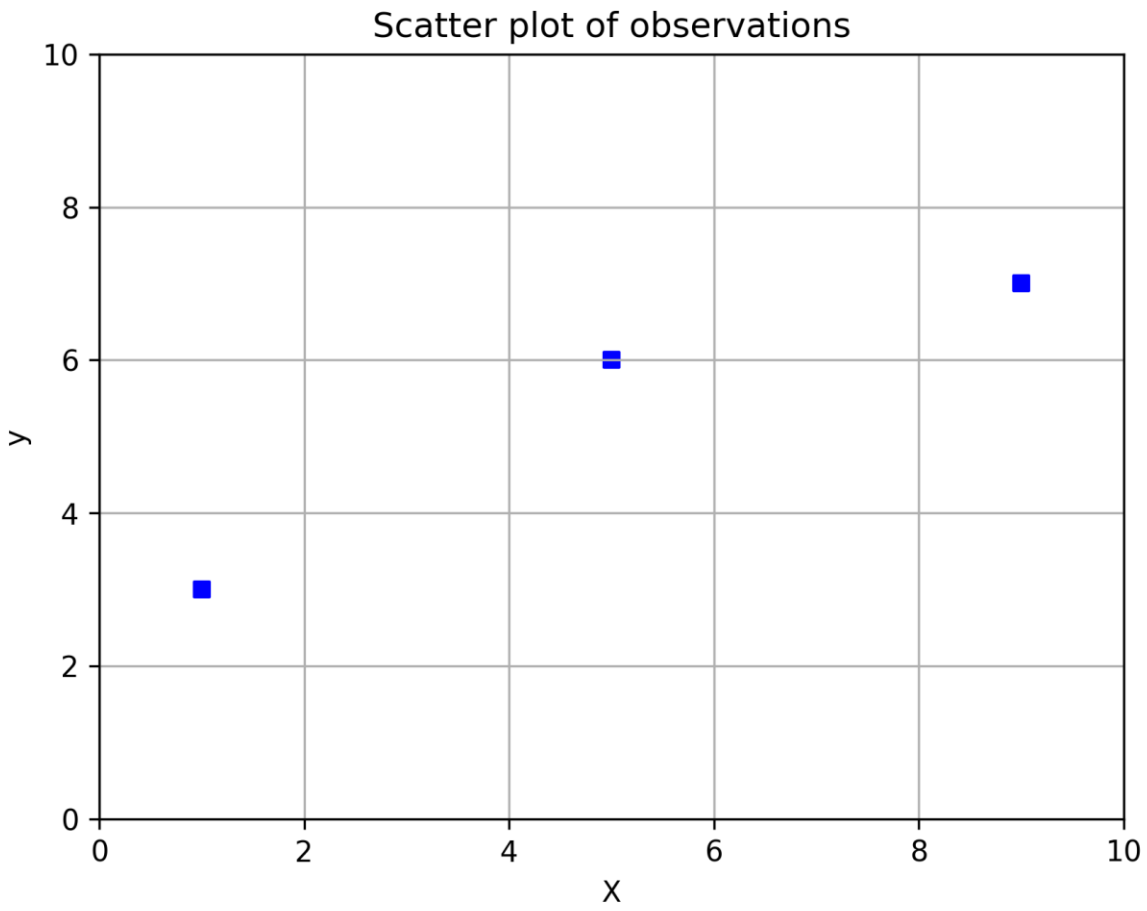
LR: Error function



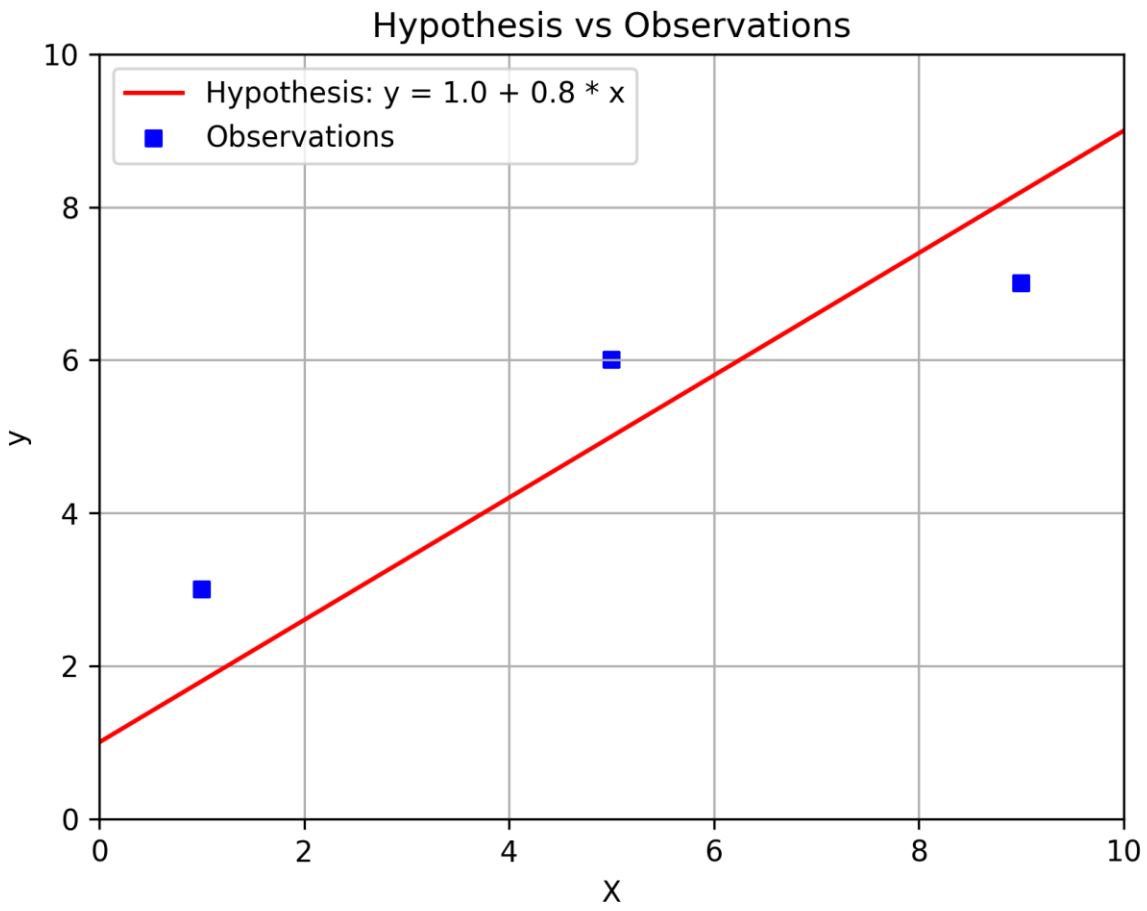
$$\hat{y} = \theta_1 x_1 + \theta_0$$

$$MSE(\Theta) = \frac{1}{m} \sum_{j=1}^{m} \left(\Theta^T x^{(j)} - y^{(j)} \right)^2$$

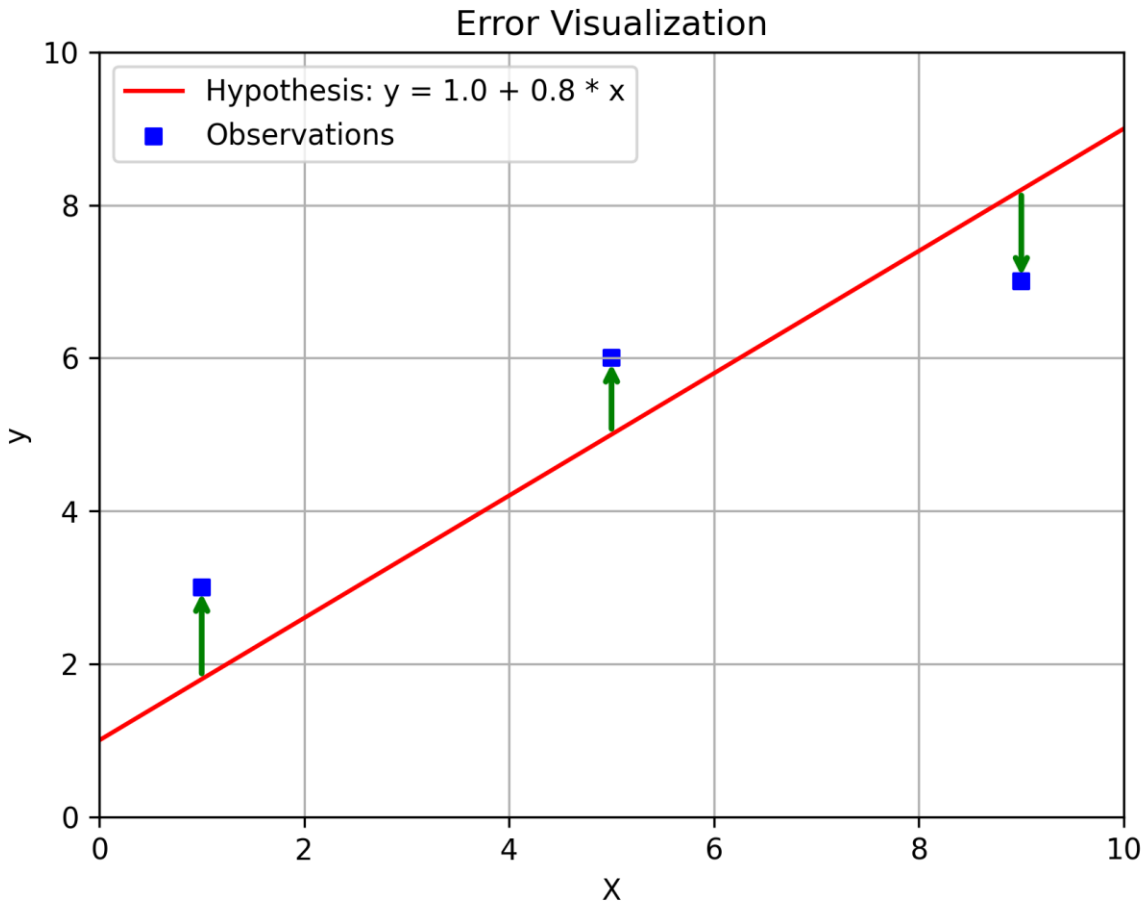
LR: Error function example



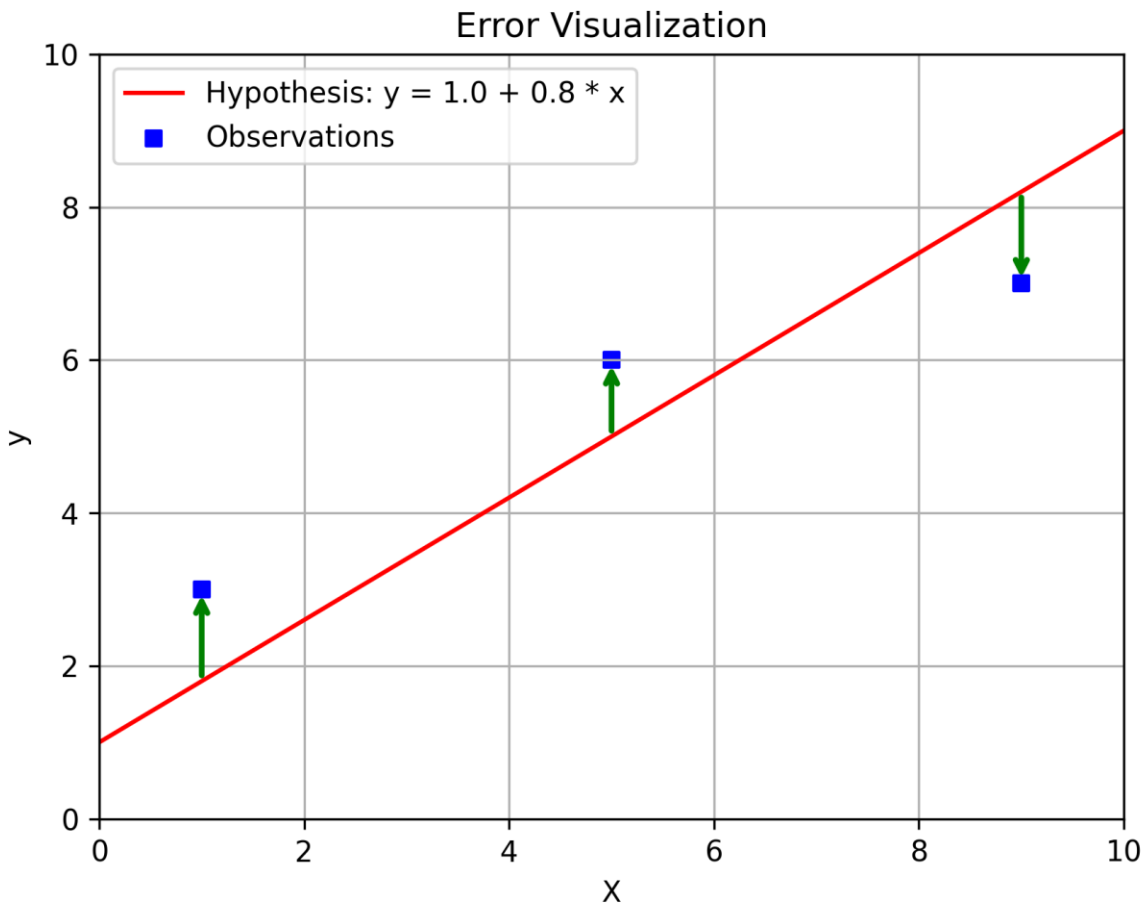
LR: Error function example



LR: Error function example



LR: Error function example

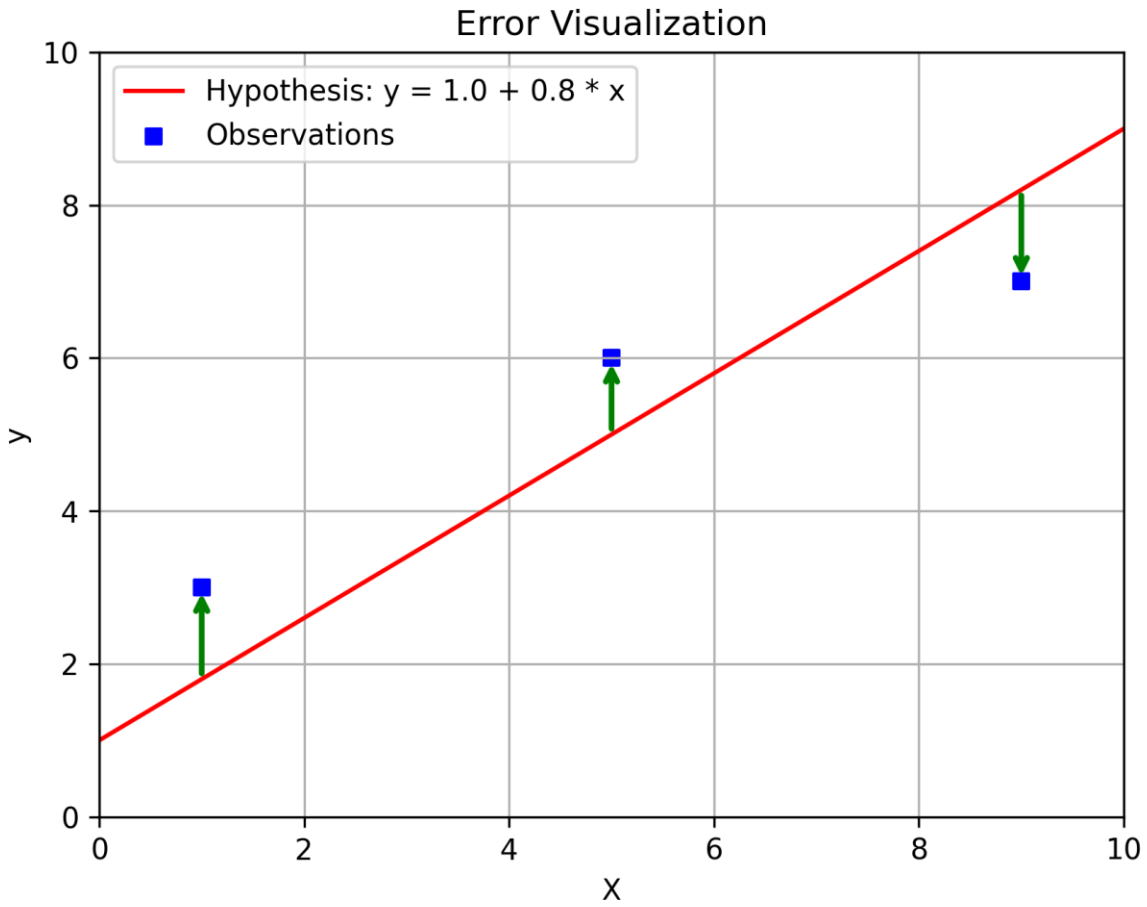


$$X = \begin{pmatrix} 1 & 5 & 9 \end{pmatrix} \qquad y = \begin{pmatrix} 3 \\ 6 \\ 7 \end{pmatrix}$$

$$\hat{y} = 1.0 + 0.8x$$

$$\hat{y} = \begin{pmatrix} 1.0 + 0.8 * 1 \\ 1.0 + 0.8 * 5 \\ 1.0 + 0.8 * 9 \end{pmatrix} = \begin{pmatrix} 1.8 \\ 5.0 \\ 8.2 \end{pmatrix}$$

LR: Error function example



$$
\begin{aligned}
\text{MSE} &= \frac{1}{3}\sum_{j=1}^{3}\left(\hat{y}^{(j)} - y^{(j)}\right)^2 \\
&= \frac{1}{3}\left((1.8-3)^2 + (5.0-6)^2 + (8.2-7)^2\right) \\
&= \frac{1}{3}\left(1.44 + 1.0 + 1.44\right) \\
&= 1.2933
\end{aligned}
$$

LR: fitting the model

Linear regression problem can be solved algebraically:

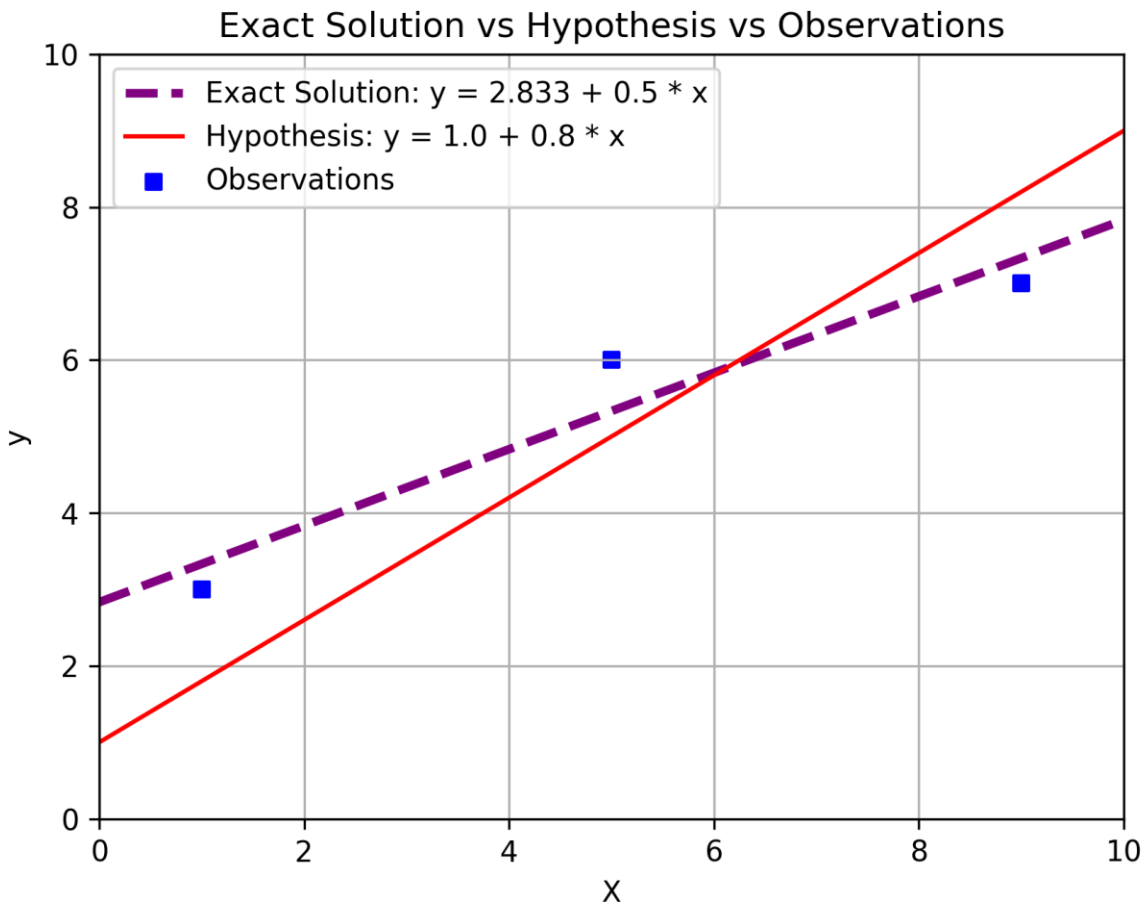
$$\Theta = (X^T X)^{-1} X^T y$$

For our toy problem this yields:

$$\Theta = \begin{pmatrix} 2.833 \\ 0.5 \end{pmatrix}$$

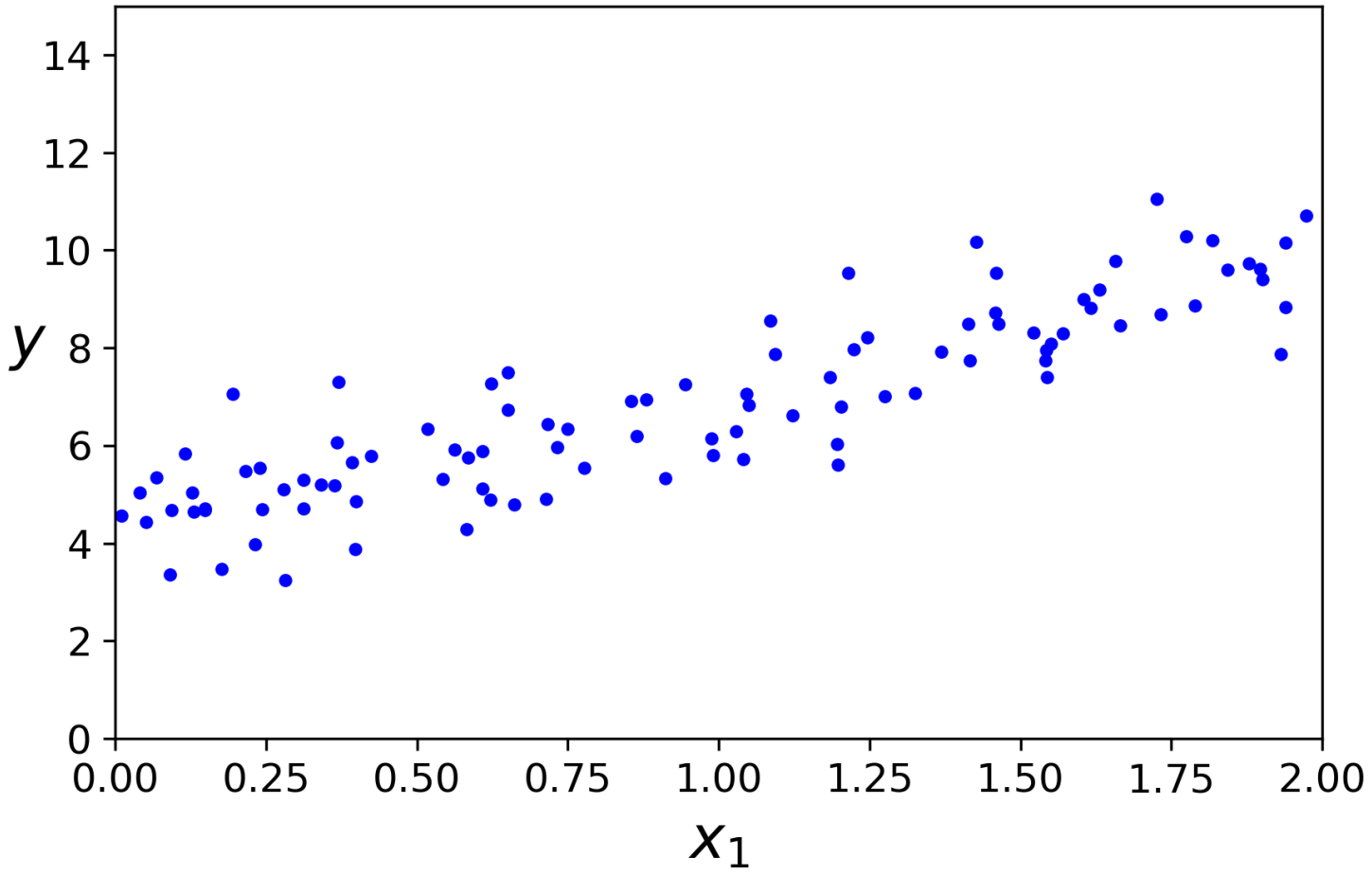
$$\hat{y} = 2.833 + 0.5x$$

LR: fitting the model



$$\text{MSE}_{exact} = 0.2222$$

Linear Regression



- Observation

x_1 Feature

$$\hat{y} = \theta_1 x_1 + \theta_0$$

LR: fitting the model

Linear regression problem can be solved algebraically:

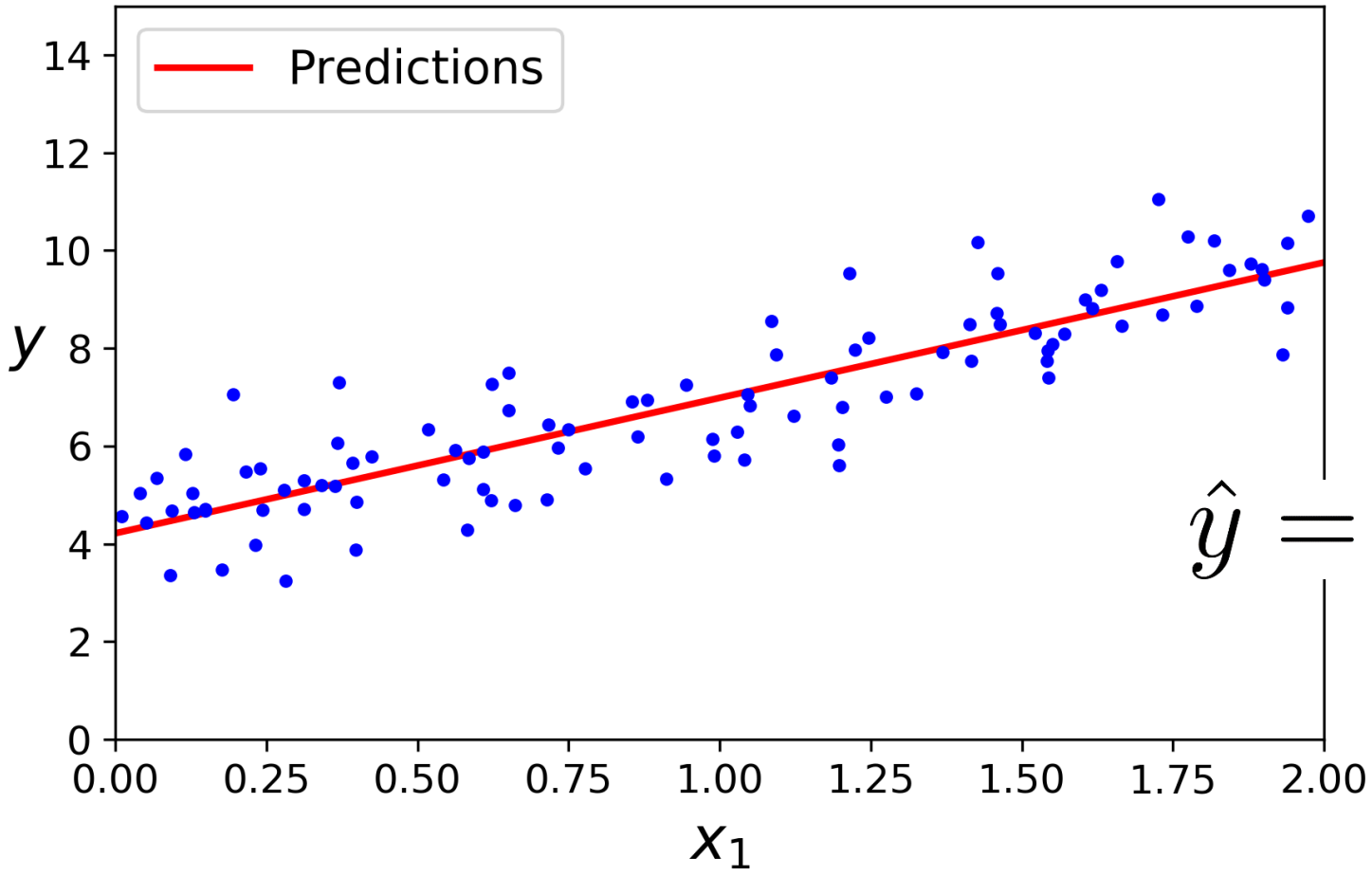
```
# assume X and y variables are numpy arrays in the scope

import numpy as np

X_b = np.c_[np.ones((100, 1)), X] # add x0 = 1 (bias) to each obs.

theta = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)

print(theta)
> [4.09324151] [[2.94615702]]
```

LR: fitting the model

Or better yet using **scikit-learn** directly:

```
# assume X and y variables are numpy arrays in the scope

from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()

lin_reg.fit(X, y)

print(lin_reg.intercept_, lin_reg.coef_)
> [4.09324151] [[2.94615702]]
```

Linear Regression



Model parameters

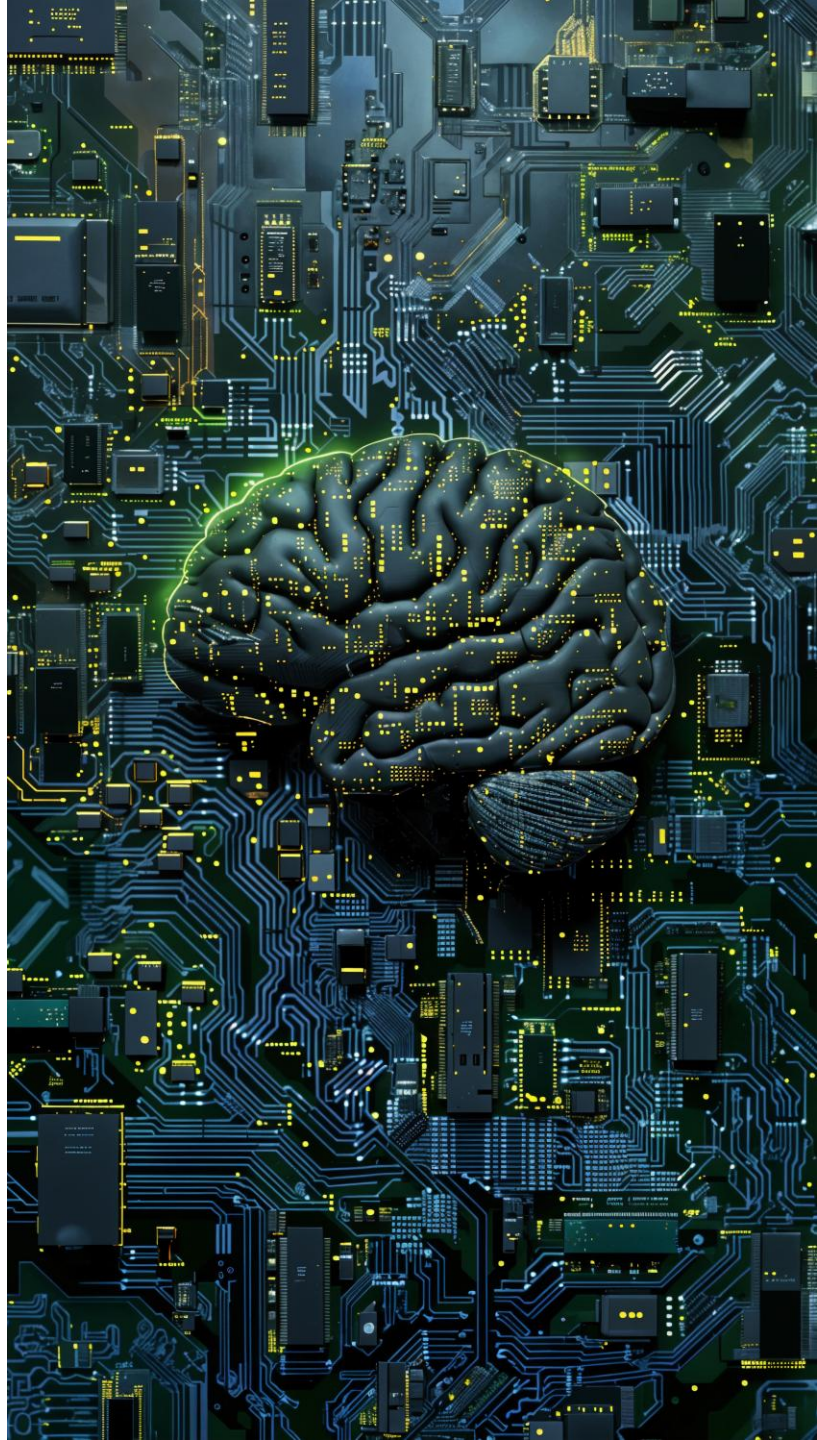
$$\hat{y} = 2.9462x_1 + 4.0932$$

Coefficient Intercept

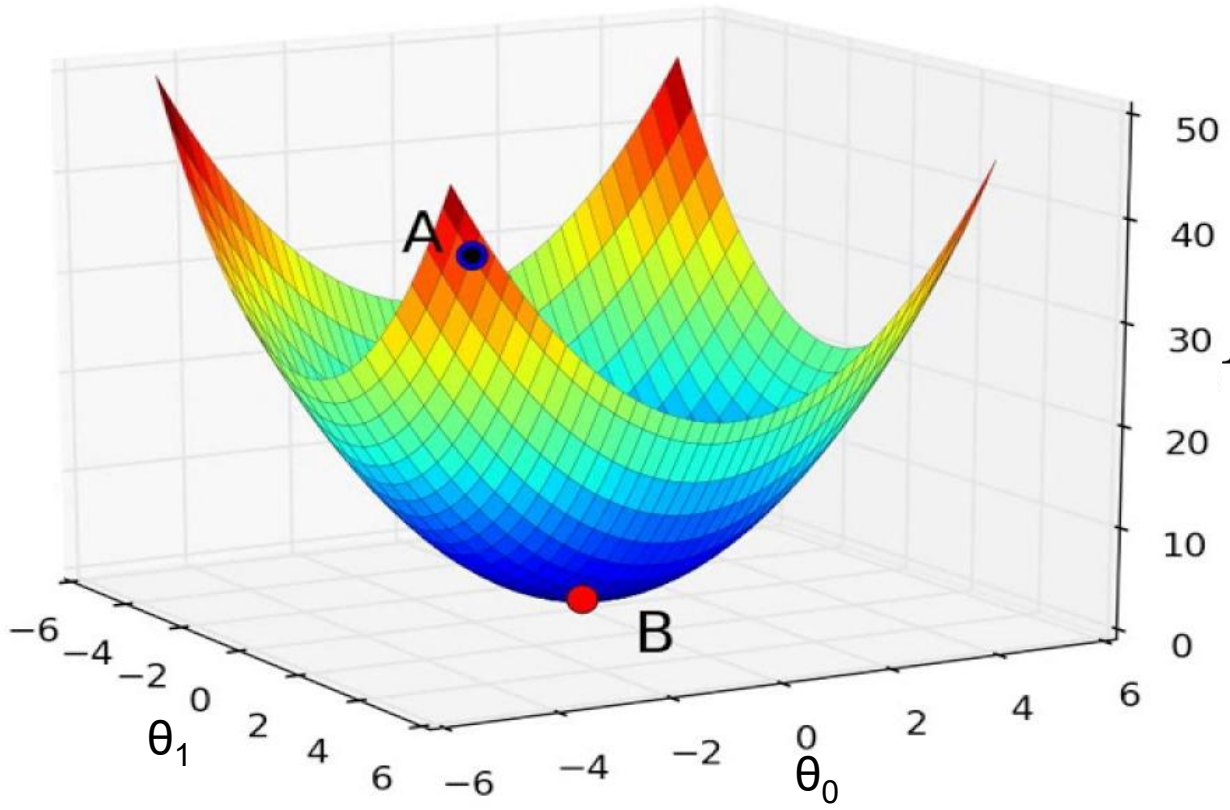
Linear Regression

Gradient Descent

SAXION
UNIVERSITY OF
APPLIED SCIENCES



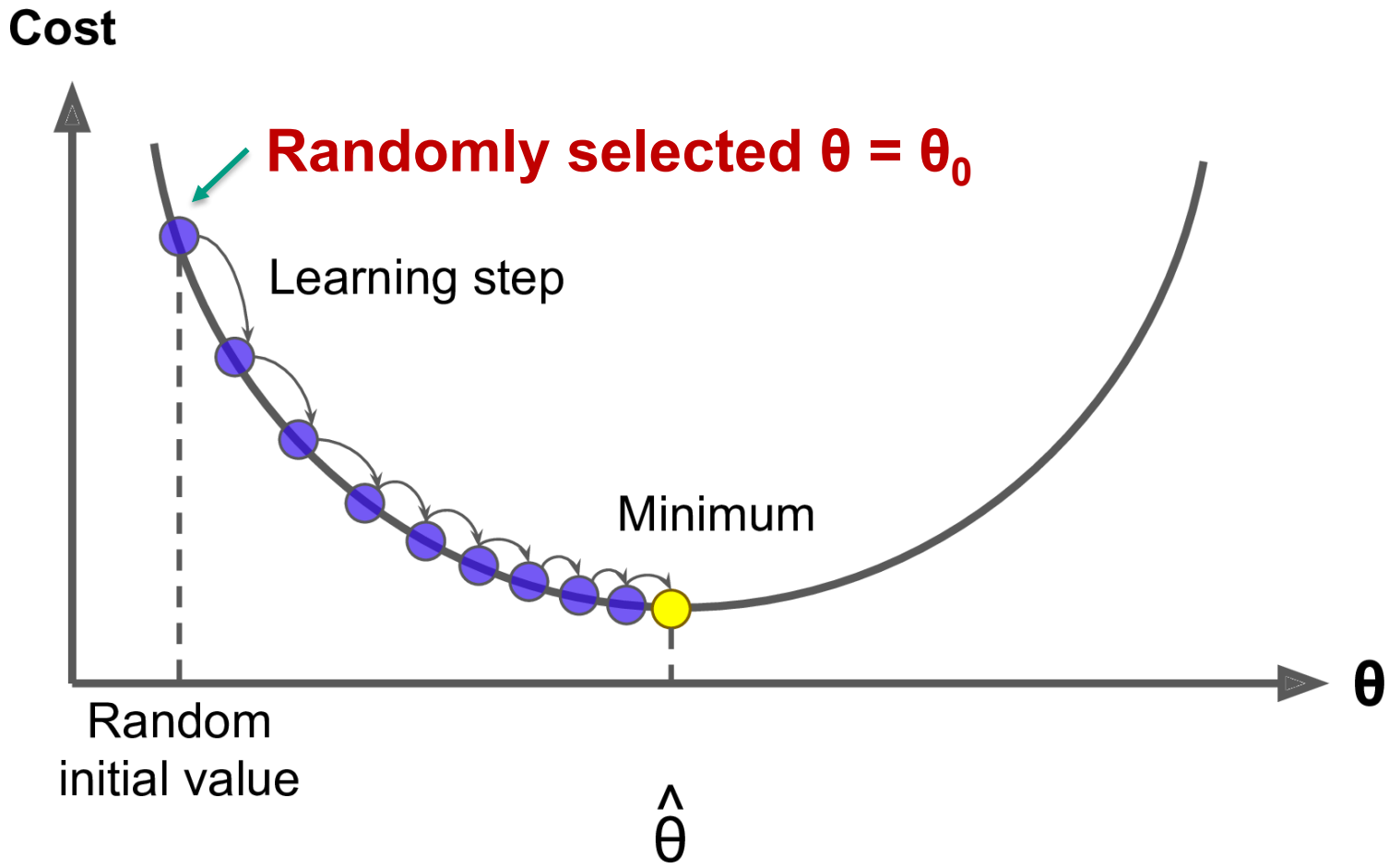
LR: Error function



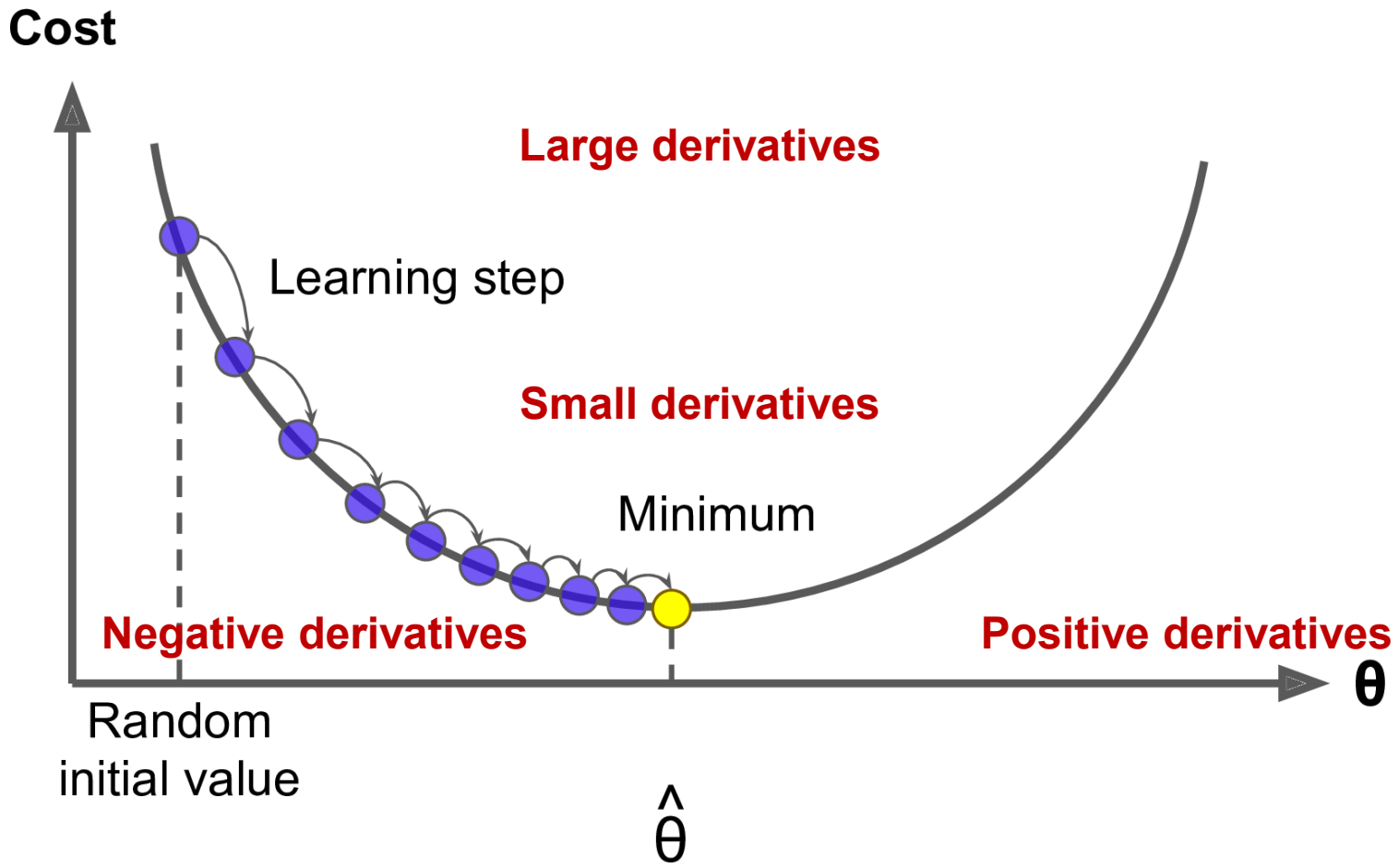
$$\hat{y} = \theta_1 x_1 + \theta_0$$

$$MSE(\Theta) = \frac{1}{m} \sum_{j=1}^{m} \left(\Theta^T x^{(j)} - y^{(j)} \right)^2$$

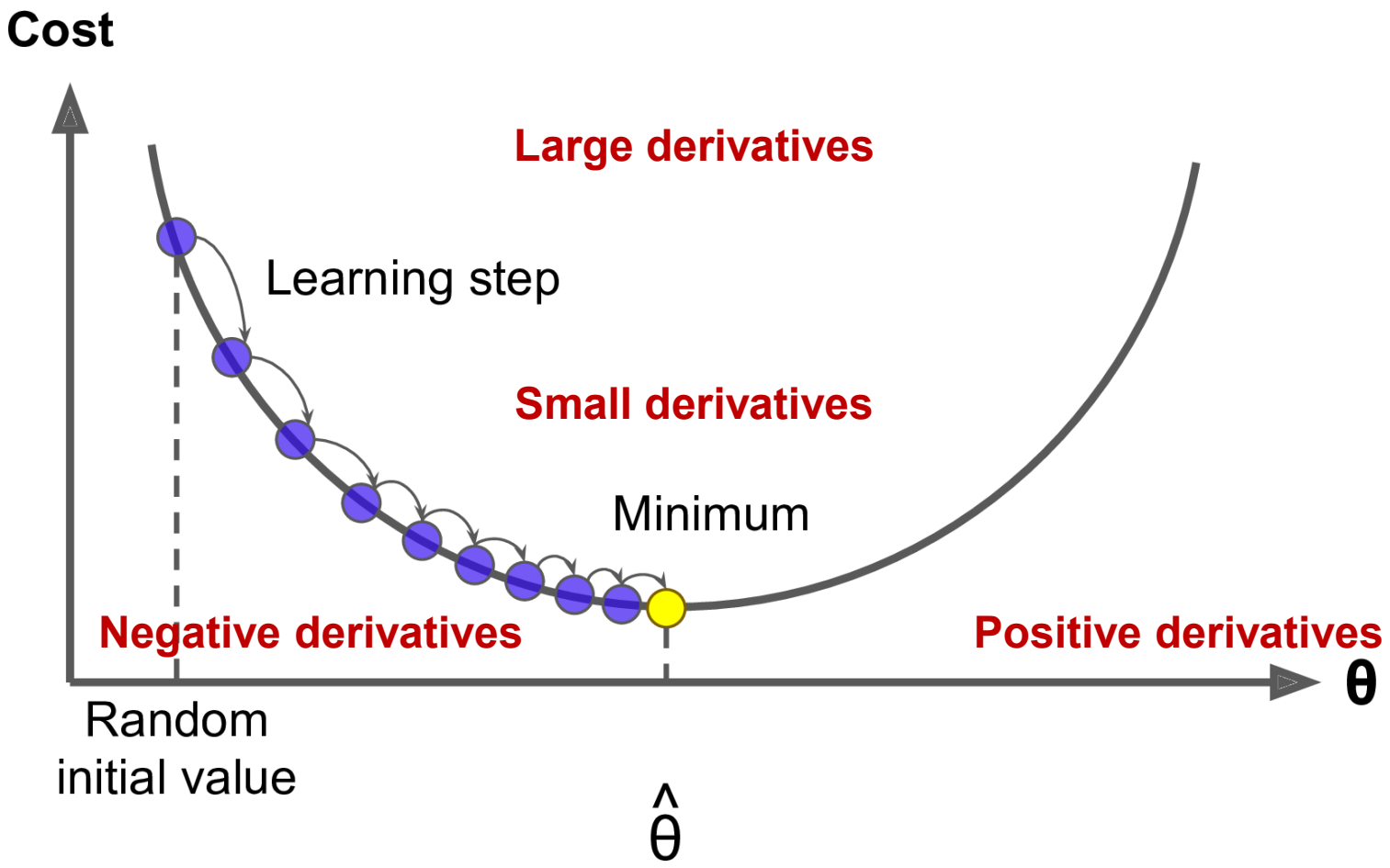
LR: Gradient Descent



LR: Gradient Descent



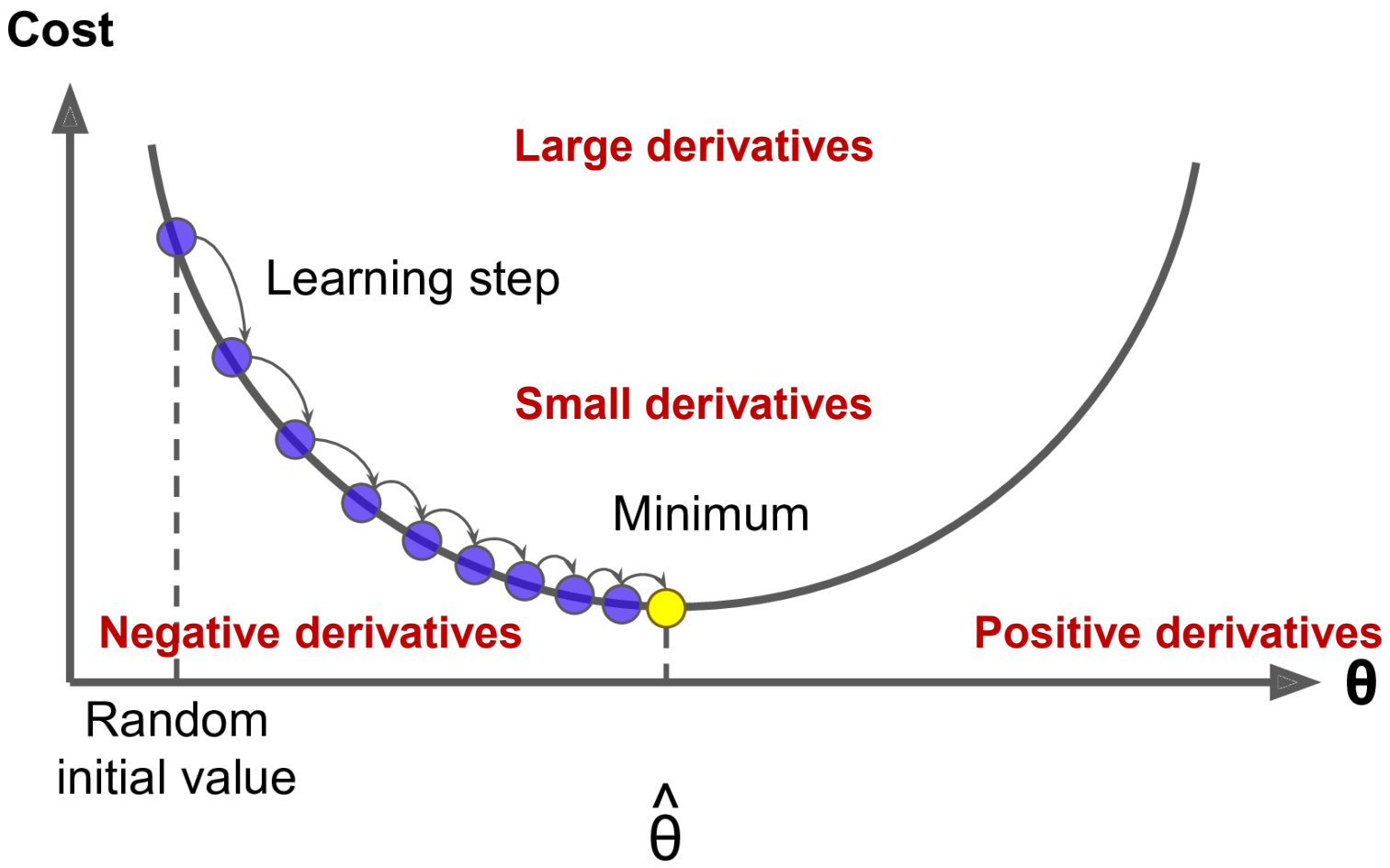
LR: Gradient Descent



At each iteration:

$$\Theta = \Theta - (\text{Derivative of Error})$$

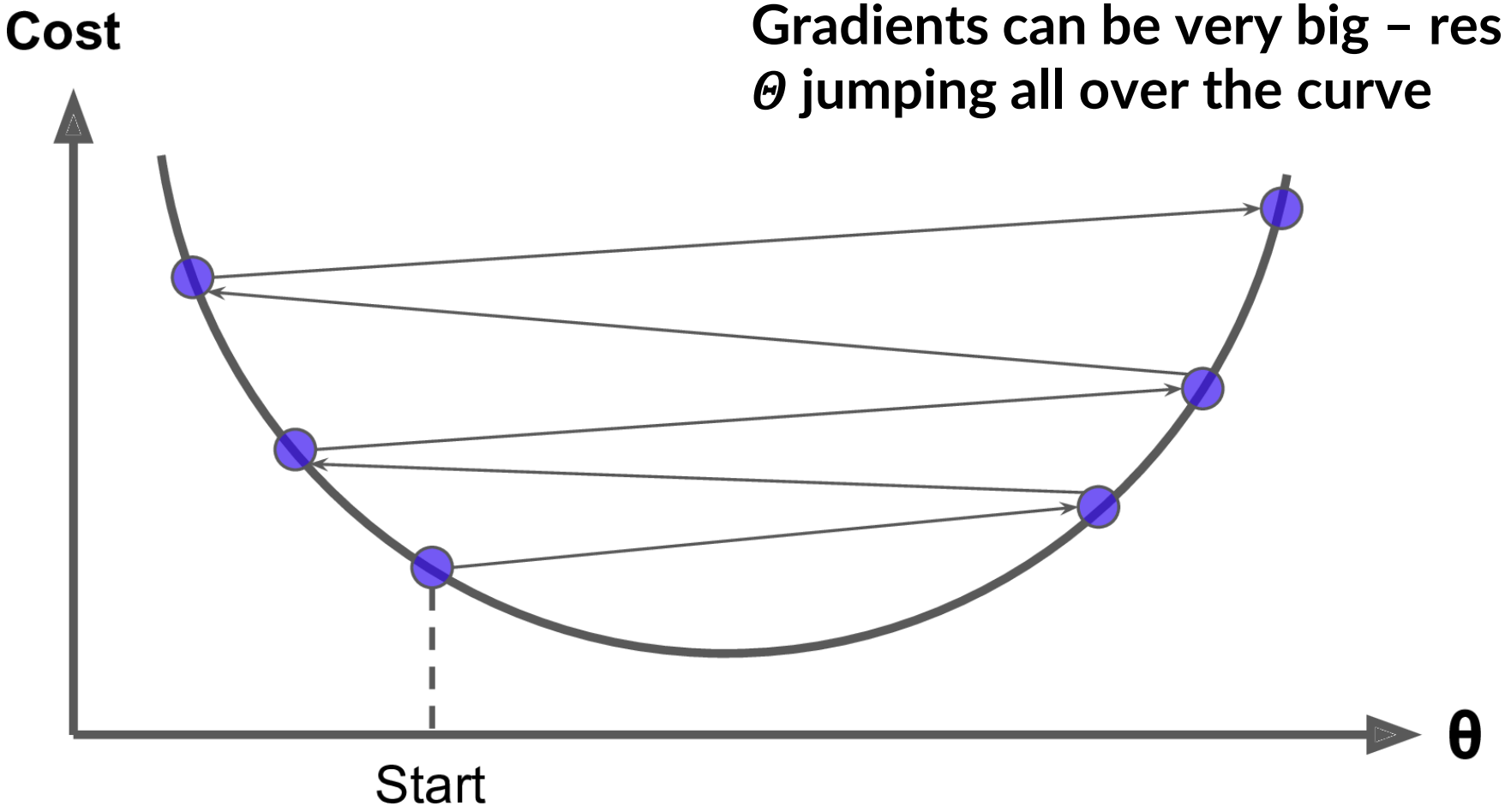
LR: Gradient Descent



At each iteration:

$$\Theta = \Theta - \eta \nabla_\theta MSE$$

Gradient Descent



Gradients can be very big – resulting in θ jumping all over the curve

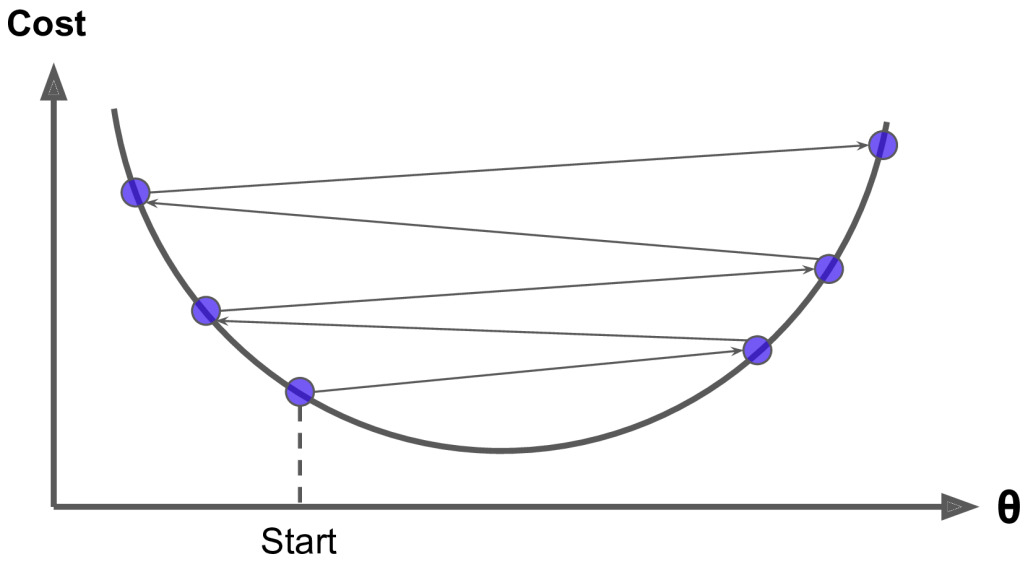
LinReg: learning rate

To avoid jumping around the error curve:

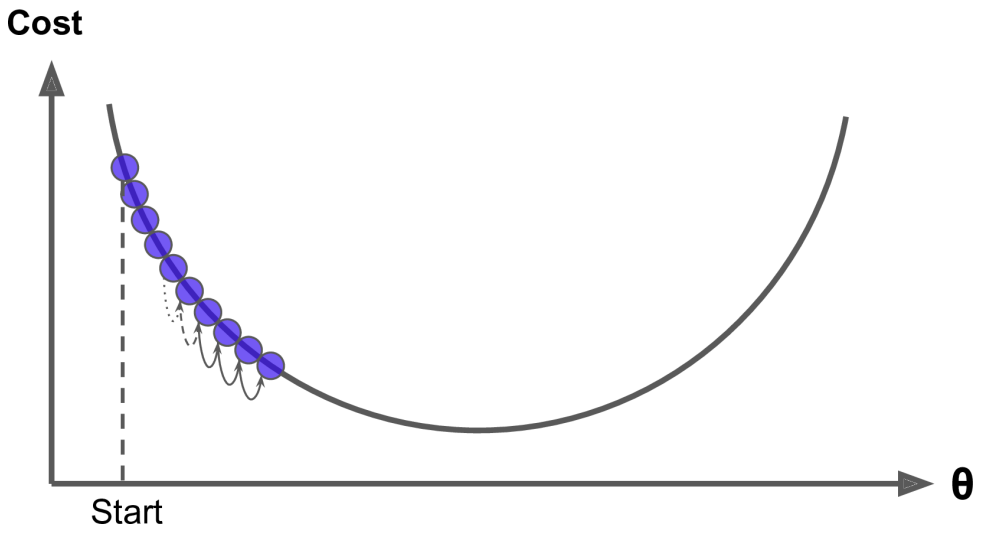
$$\Theta = \Theta - \eta \nabla_\theta MSE$$

η is called the **Learning Rate**

LinReg: learning rate

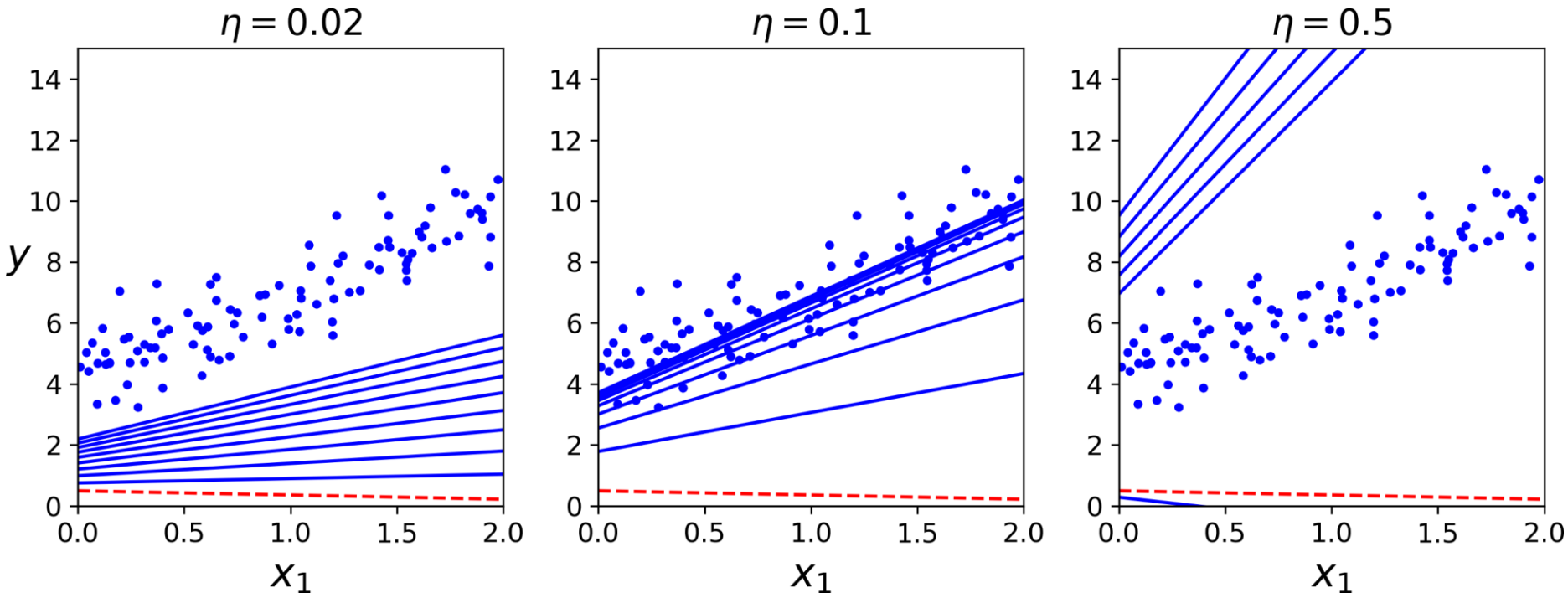


η too big



η too small

LinReg: learning rate

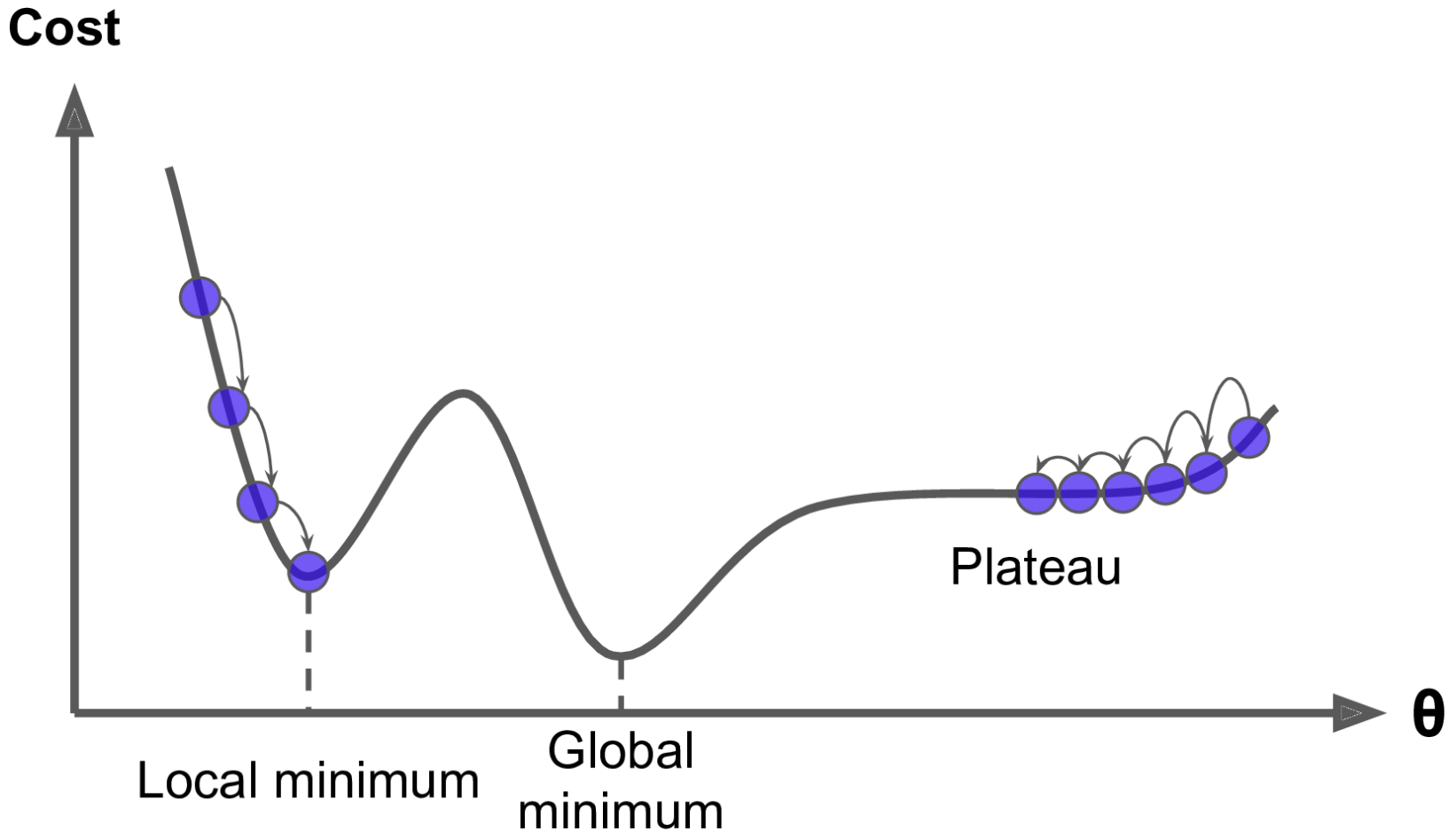


Choosing the right learning:

η too big can result in overshooting the optimum

η too small results in very small training progress

Gradient Descent

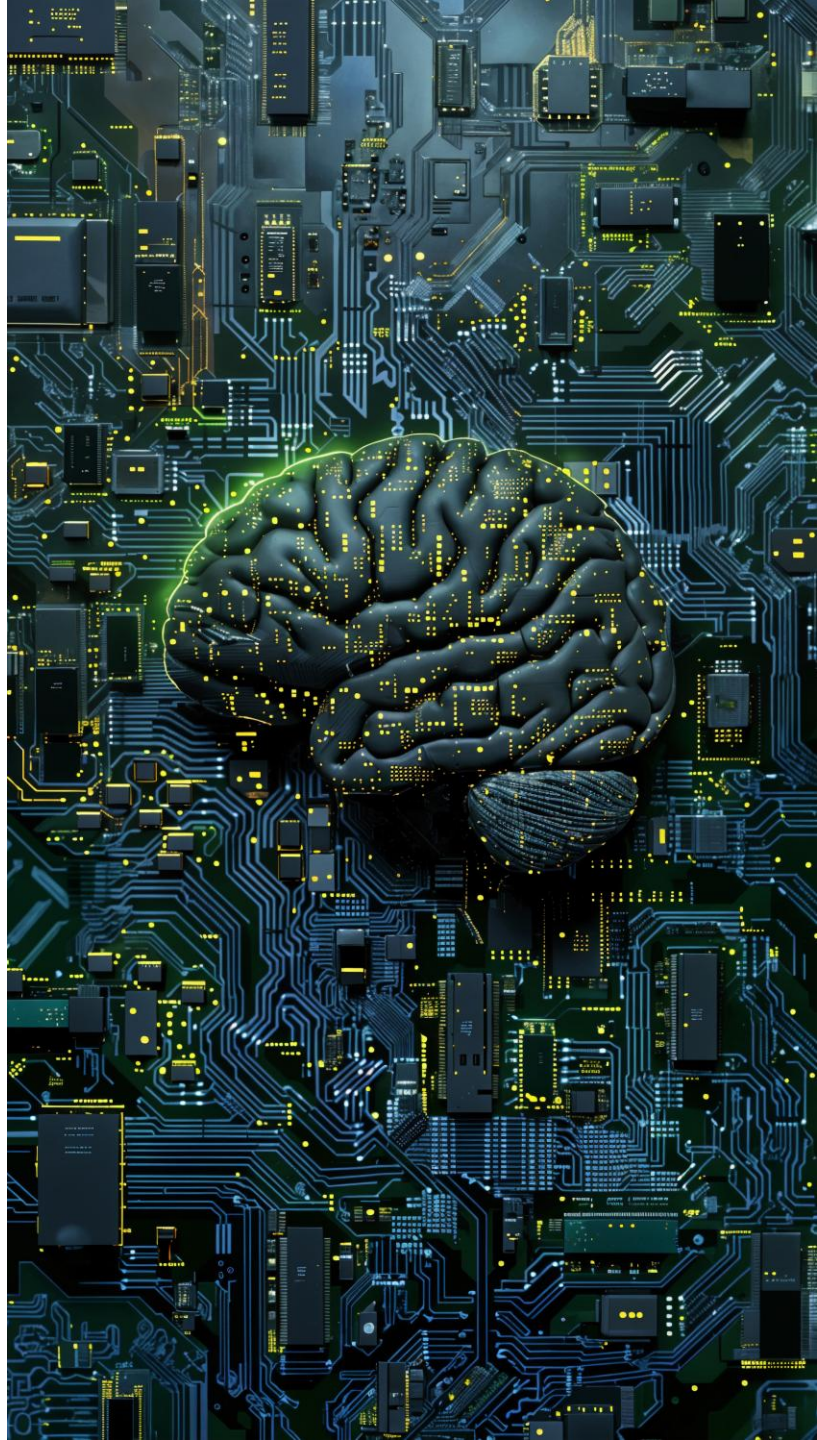


Gradient Descent is not so good with problems with many minima

Gradient Descent variations

Linear Regression

SAXION
UNIVERSITY OF
APPLIED SCIENCES



Gradient Descent

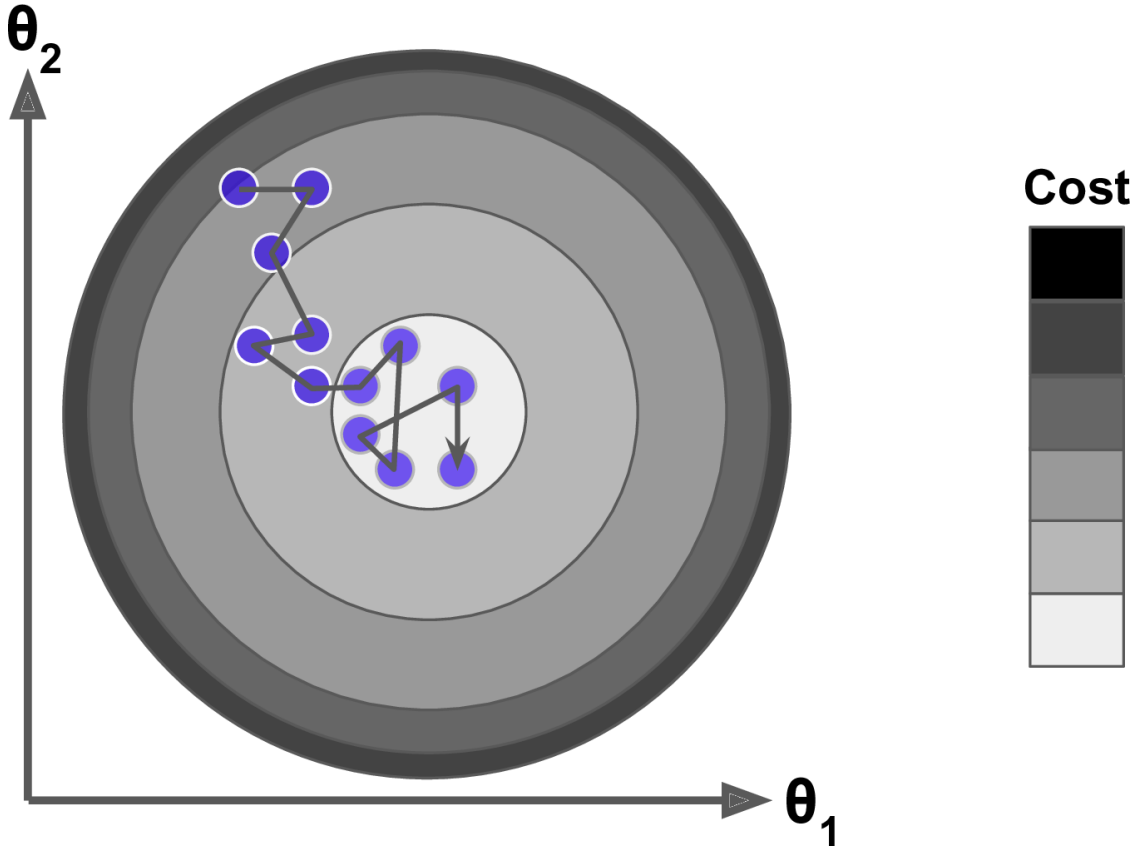
Batch Gradient Descent uses all the observations in each iteration.

This costs a lot of resources (time).

There are others:

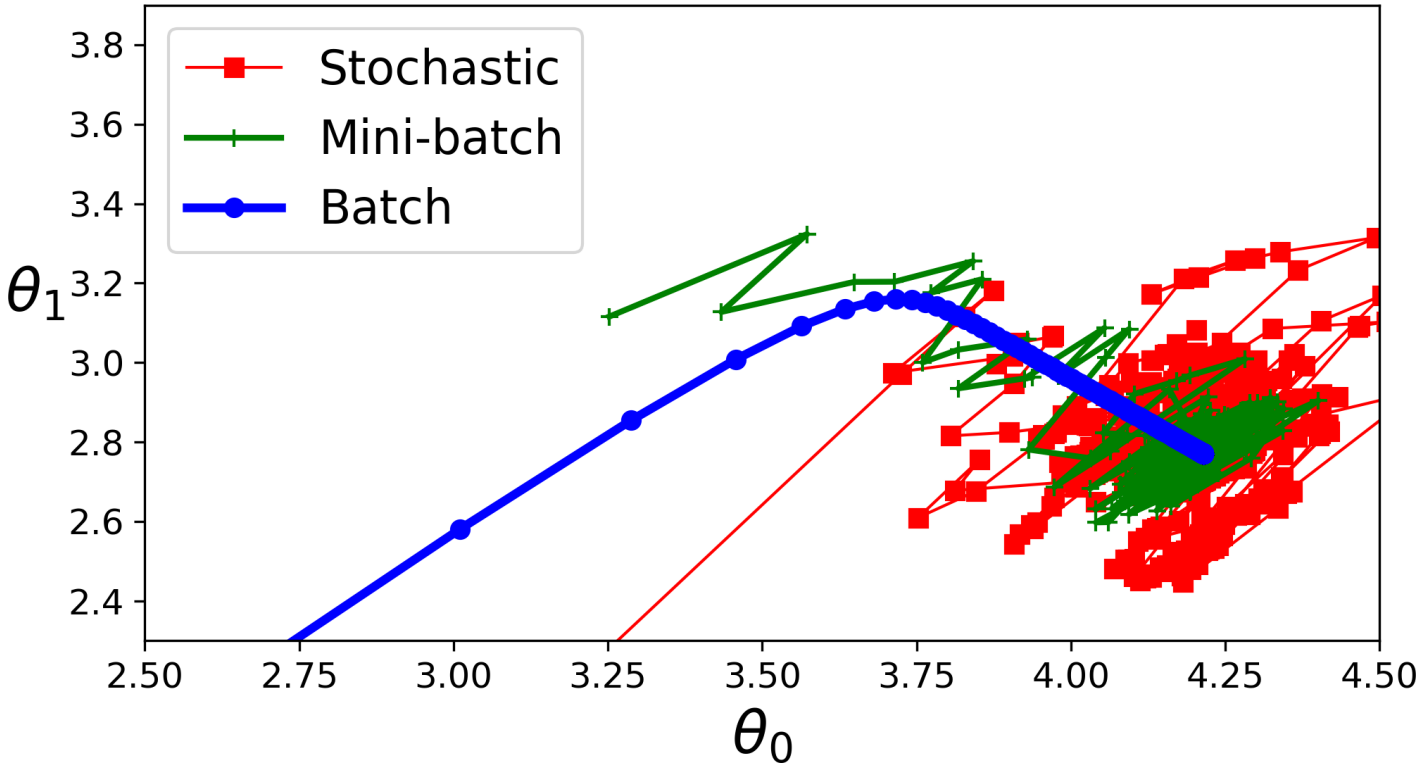
- **Mini-batch Gradient Descent**
- **Stochastic Gradient Descent**

Stochastic GD (SGDRegressor in scikit-learn)



Stochastic Gradient Descent uses one randomly chosen observation at each iteration.

Mini-batch GD (not in scikit-learn)

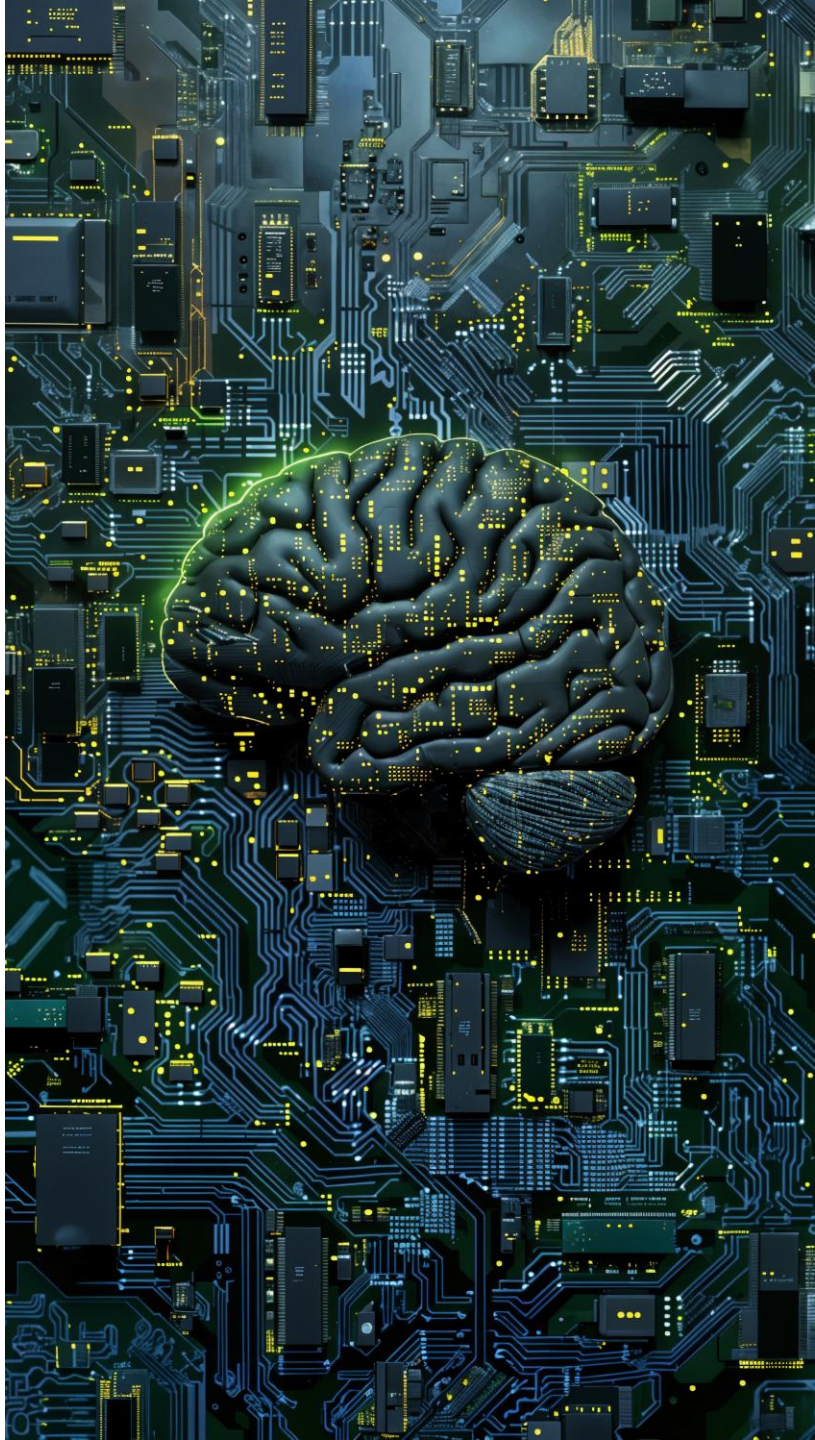


Mini-Batch Gradient Descent uses a number of randomly selected observations at each iteration.

Training & predicting

Linear Regression

SAXION
UNIVERSITY OF
APPLIED SCIENCES



Simple Linear Regression

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()

lin_reg.fit(X, y)

print(lin_reg.intercept_, lin_reg.coef_)

> [4.09324151] [[2.94615702]]
```

Gradient Descent algo

Using **Stochastic Gradient Descent** with **scikit-learn**:

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=100, tol=1e-3,
                       learning_rate='constant', eta0=0.1)
sgd_reg.fit(X, y)

print(sgd_reg.intercept_, sgd_reg.coef_)

> [4.52409112], [3.15327724]
```

SGDRegressor can be also used for other Gradient Descent types

Gradient Descent & errors

```
from sklearn.linear_model import SGDRegressor


sgd_reg = SGDRegressor(max_iter=100, tol=1e-3,
                       learning_rate='constant', eta0=0.1)
sgd_reg.fit(X, y)
```

$$RMSE(\Theta) = \sqrt{MSE(\Theta)}$$

Gradient Descent & errors

```
from sklearn.linear_model import SGDRegressor
from sklearn.metrics import mean_squared_error

sgd_reg = SGDRegressor(max_iter=100, tol=1e-3,
                       learning_rate='constant', eta0=0.1)
sgd_reg.fit(X, y)
```

```
y_train_predict = sgd_reg.predict(X)
print( np.sqrt(mean_squared_error(y, y_train_predict)) )

> 1.6331262379637725
```

Cross-validation

💡 Split the data set into the training and validation sets

For instance:

- 80% of points is training
- 20% of points is validation

Training data is used to train the model.
Validation data is used to see how well it performs.

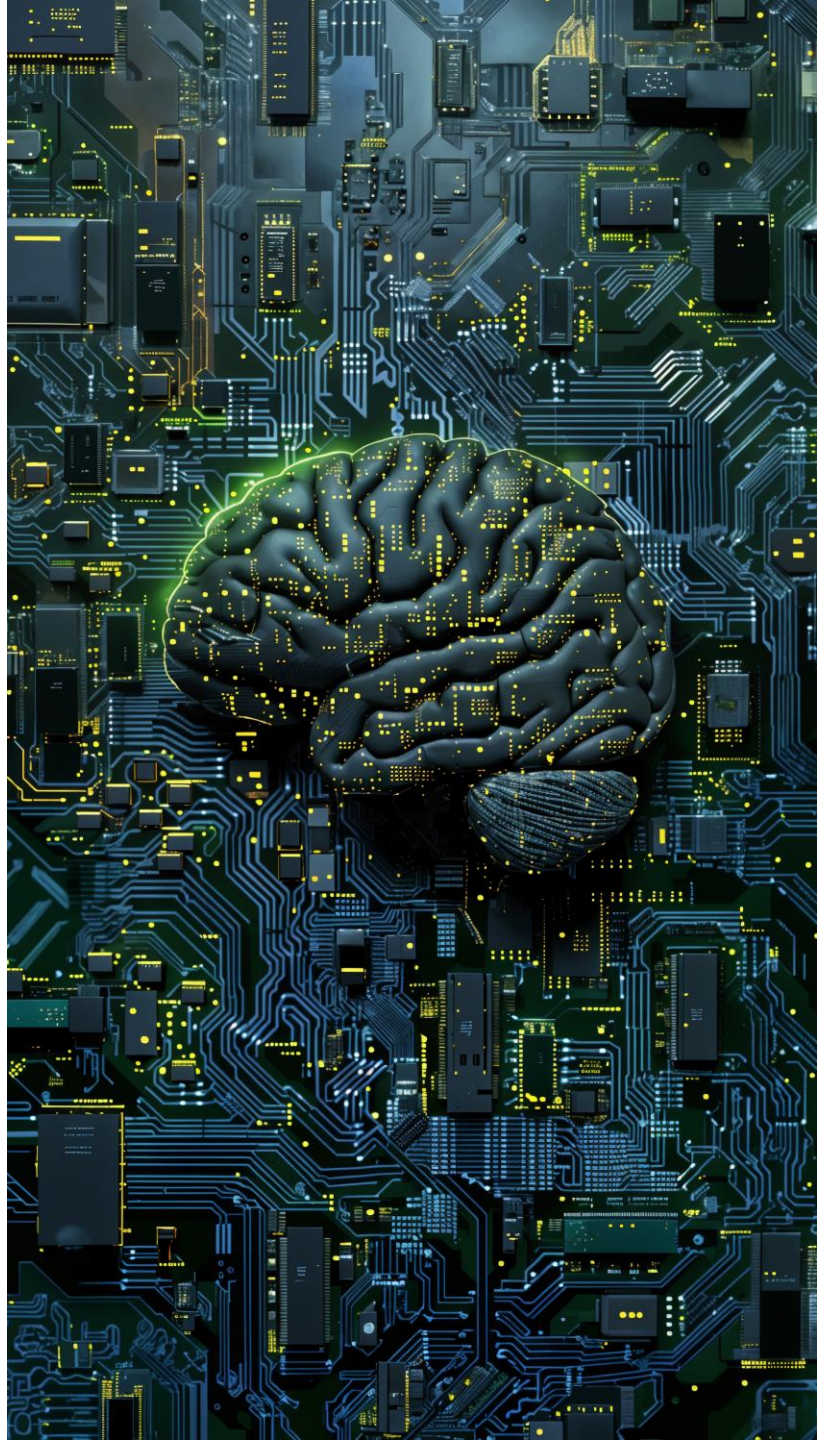
Cross-validation

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)

y_train_predict = lin_reg.predict(X_train)
y_val_predict = lin_reg.predict(X_val)

print(np.sqrt(mean_squared_error(y_train, y_train_predict)))
print(np.sqrt(mean_squared_error(y_val, y_val_predict)))

> 1.6331262379637725
> 1.773473724018952
```

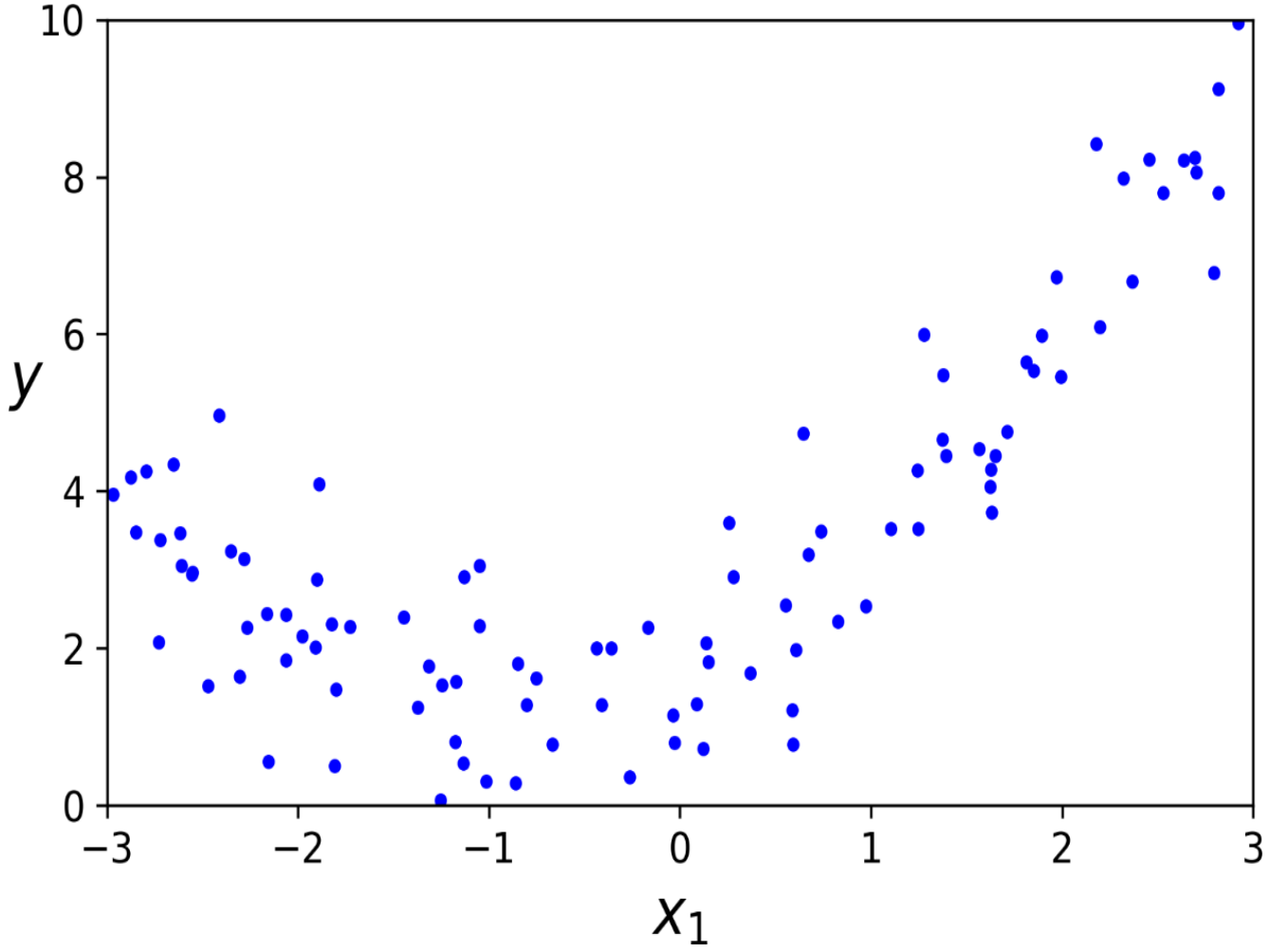
Non-linear problems

Linear Regression

SAXION
UNIVERSITY OF
APPLIED SCIENCES



Non-linear function



$$\hat{y} = \theta_2 x_1^2 + \theta_1 x_1 + \theta_0$$

$$x_2 = x_1^2$$

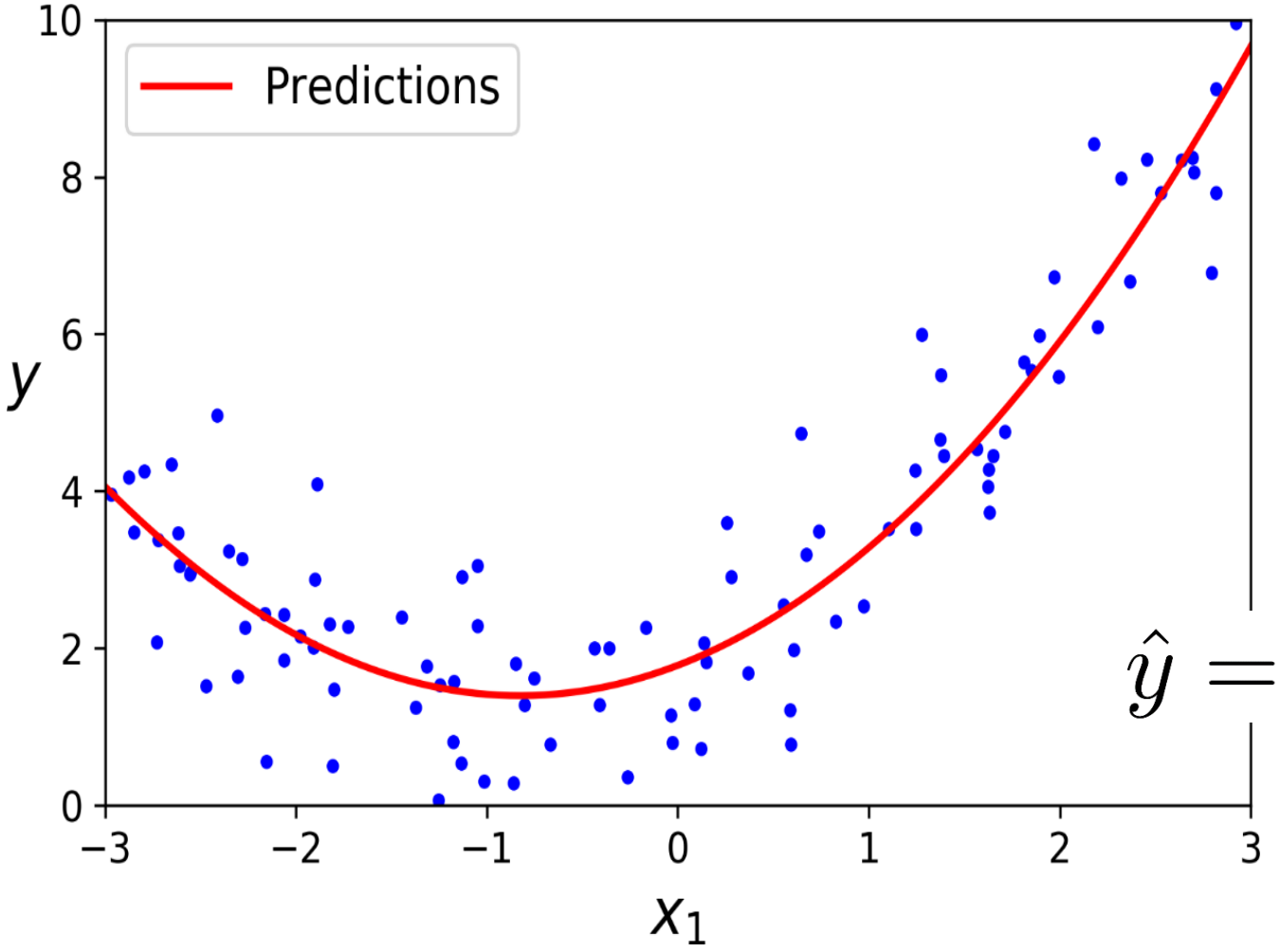
$$\hat{y} = \theta_2 x_2 + \theta_1 x_1 + \theta_0$$

Polynomial Regression

Adding a non-linear (polynomial) feature:

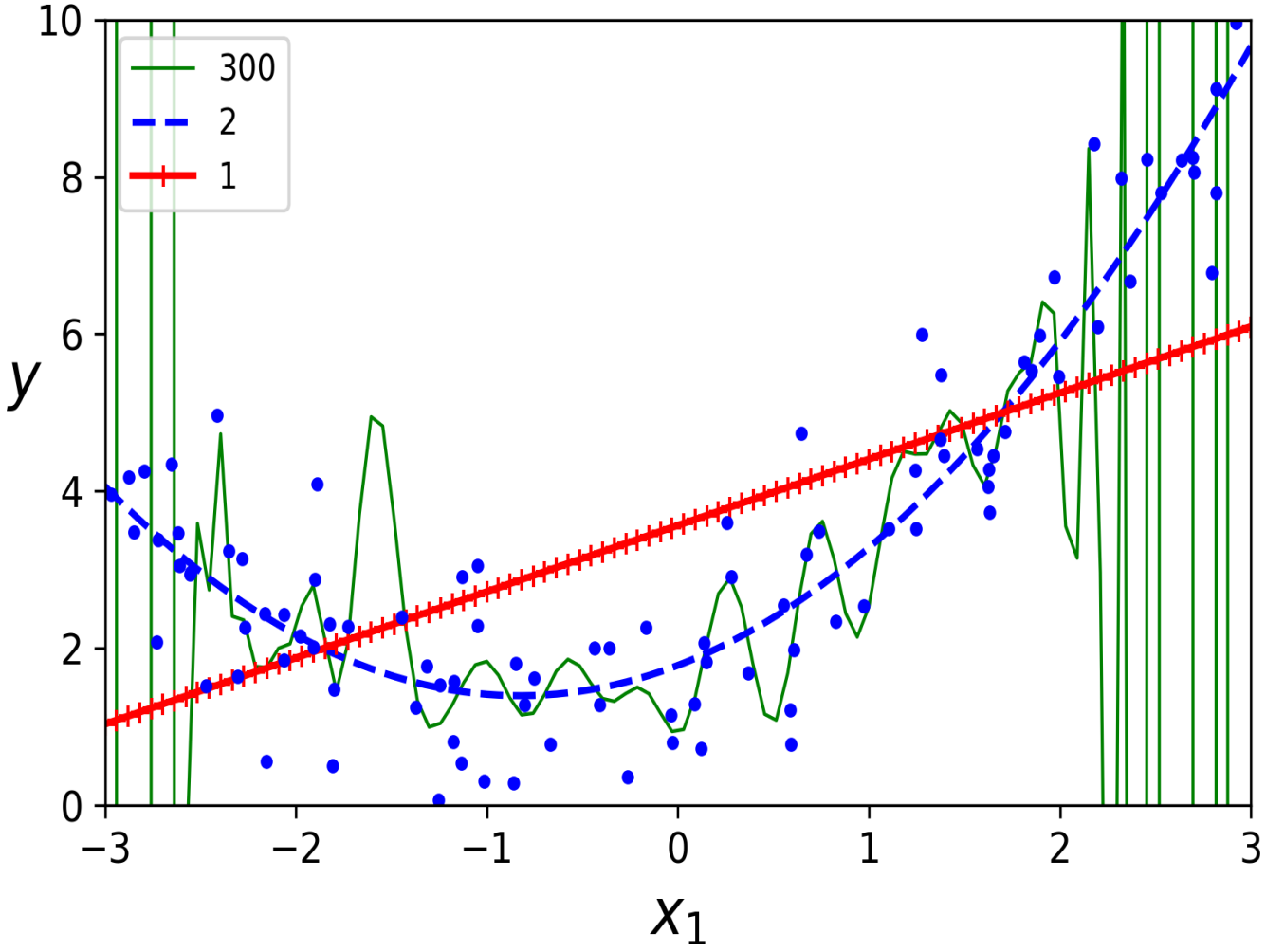
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

poly_features = PolynomialFeatures(degree=2, include_bias=False)

X_poly = poly_features.fit_transform(X)

lin_reg = LinearRegression()
lin_reg.fit(X_poly, y)
print(lin_reg.intercept_, lin_reg.coef_)

> [1.78134581]), [[0.93366893, 0.56456263]]
```

Polynomial Regression



$$\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$$

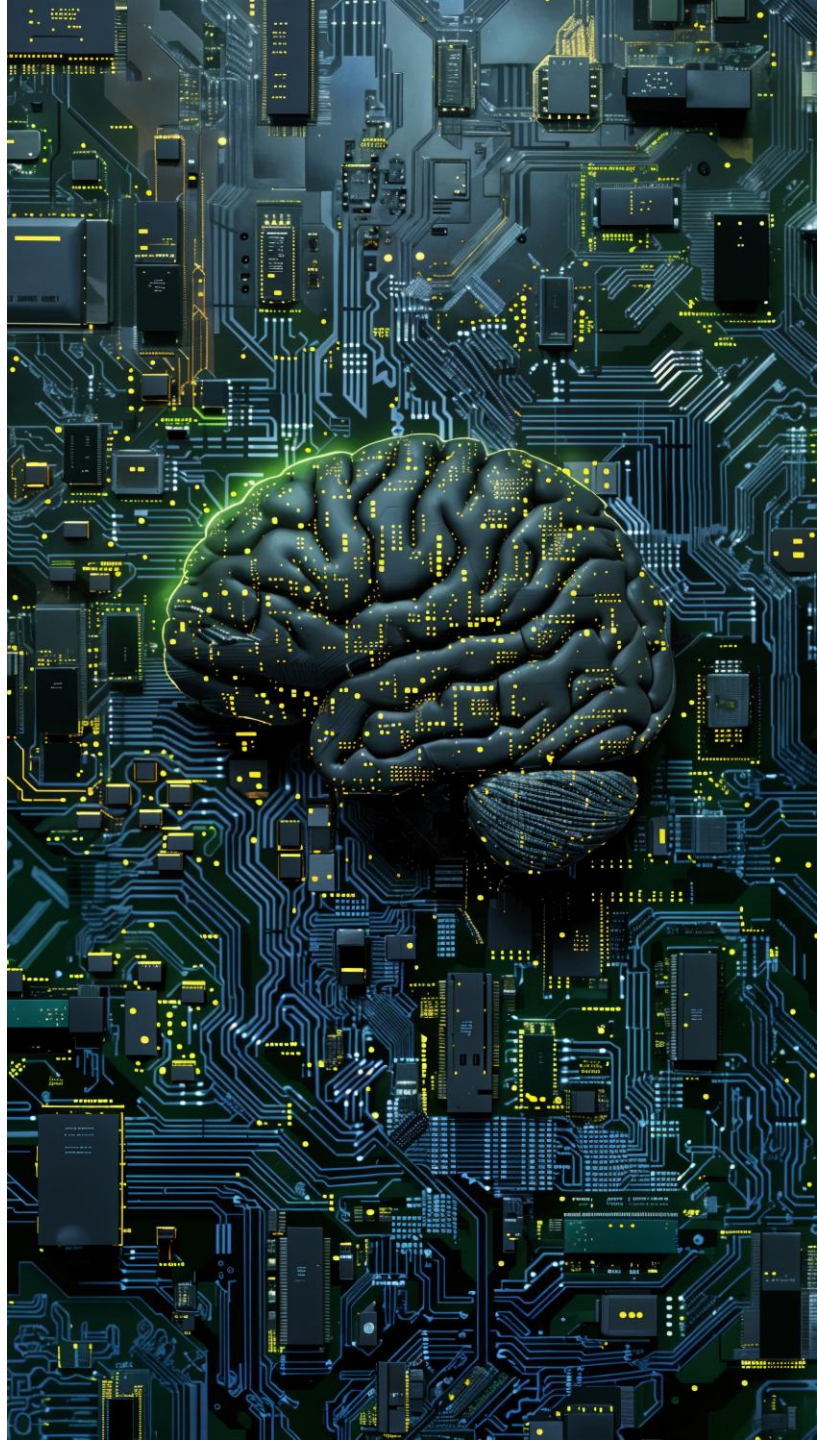
Selecting the right model



Selecting the right model is difficult:

- Degree 2 is fine
- Degree 300 is **overfitted**
- Degree 1 is **underfitted**

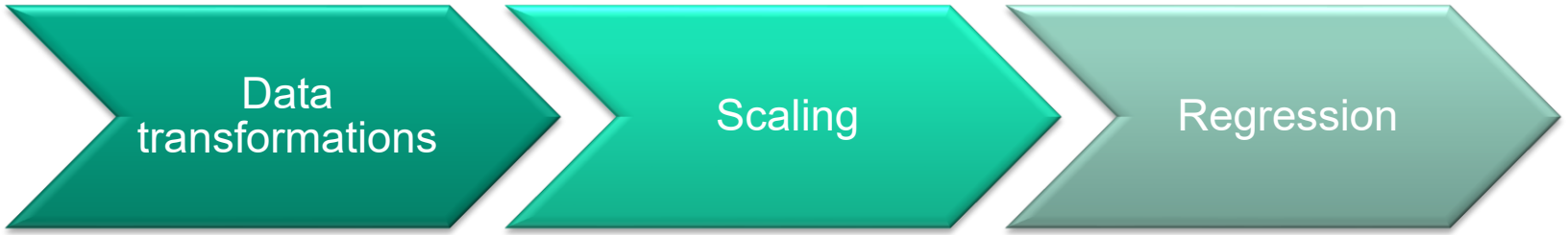
Pipelines



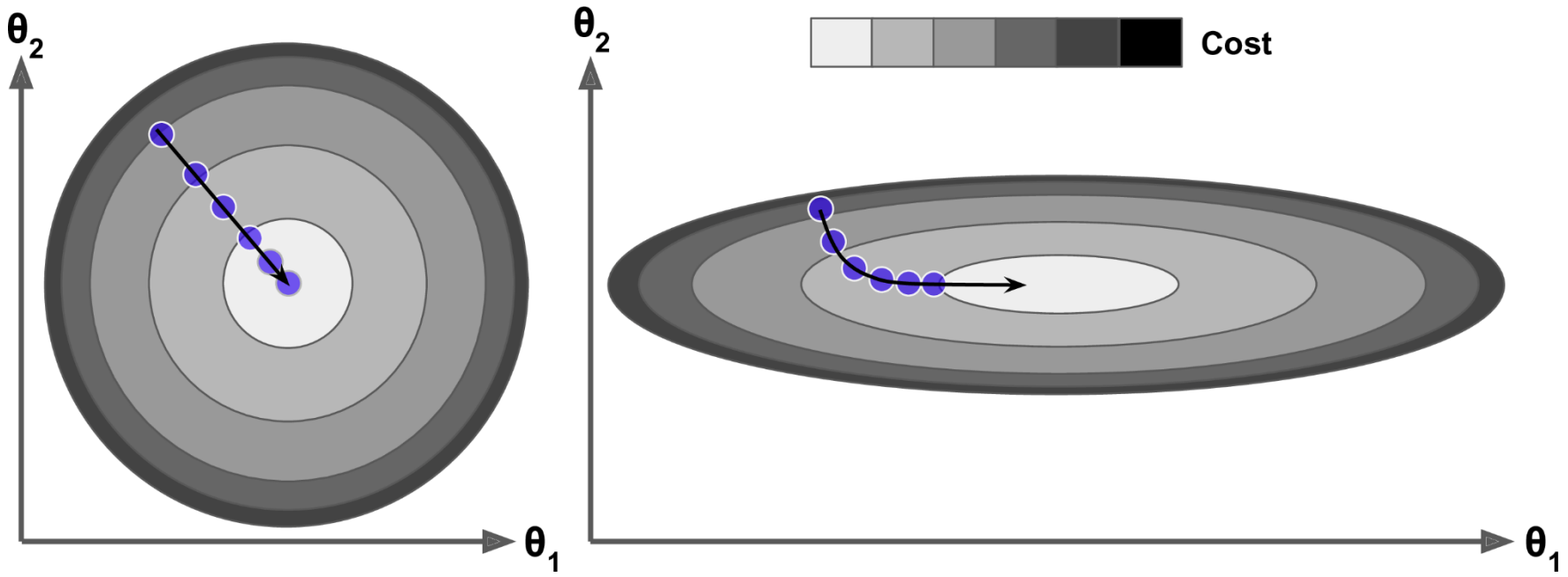
SAXION
UNIVERSITY OF
APPLIED SCIENCES

Pipelines

Typical machine learning (regression) process



Scaling features



Piepelines

```
from sklearn.pipeline import Pipeline

from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

std_scaler = StandardScaler()
lin_reg = LinearRegression()


regressor = Pipeline([

  ("std_scaler", std_scaler),
  ("lin_reg", lin_reg),
])
```

Piepelines

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

std_scaler = StandardScaler()
lin_reg = LinearRegression()
poly_features = PolynomialFeatures(degree=2, include_bias=False)

regressor = Pipeline([
  ("poly_2", poly_features),
  ("std_scaler", std_scaler),
  ("lin_reg", lin_reg),
])
```

Pipelines: scaling

StandardScaler

$$X_{norm} = \frac{X - \mu}{\sigma}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

Mean

$$\sigma = \sqrt{\sum_{i=1}^{N} (x_i - \mu)^2}$$

Standard Deviation

MinMaxScaler

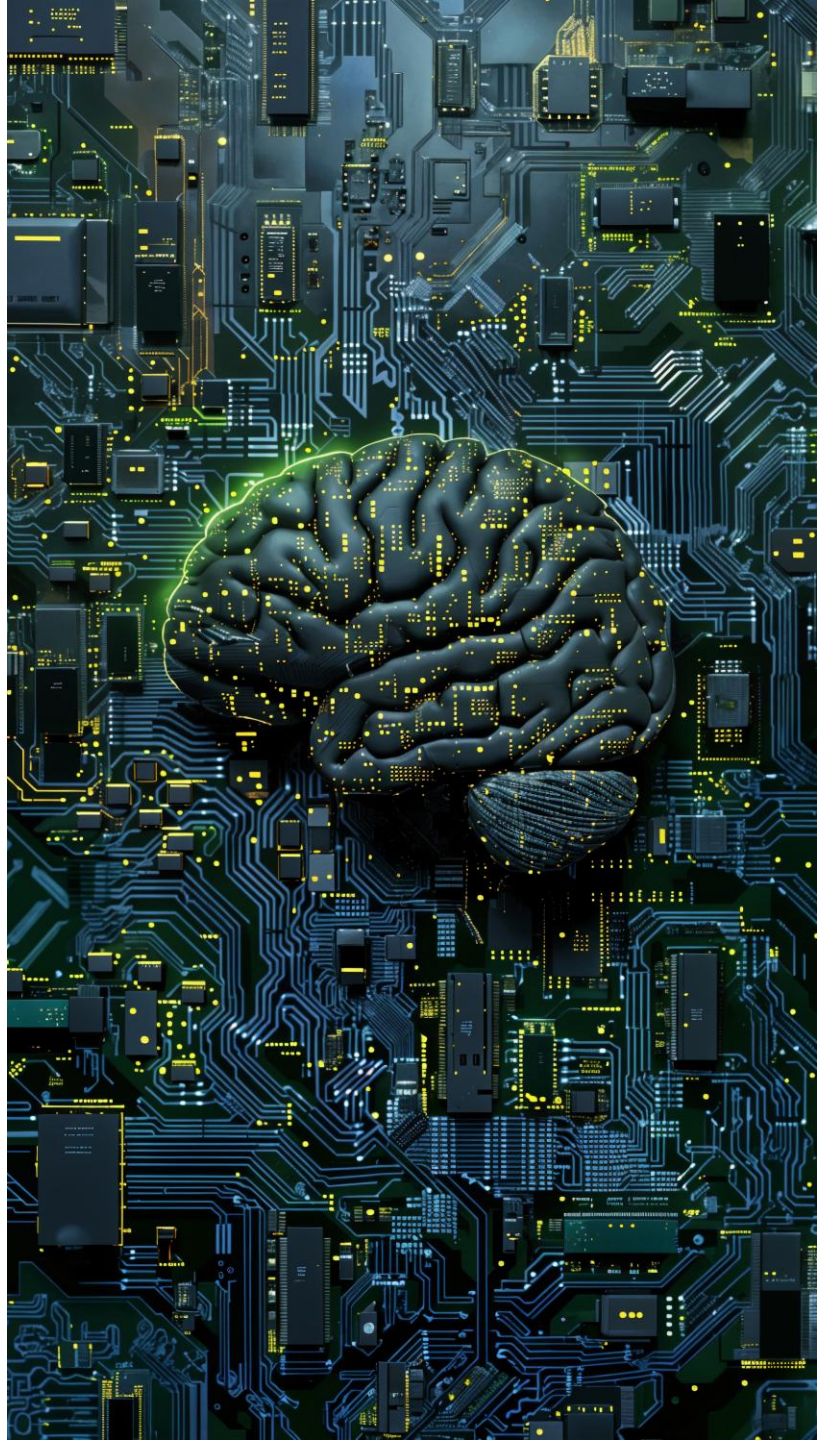
$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Work, work, work

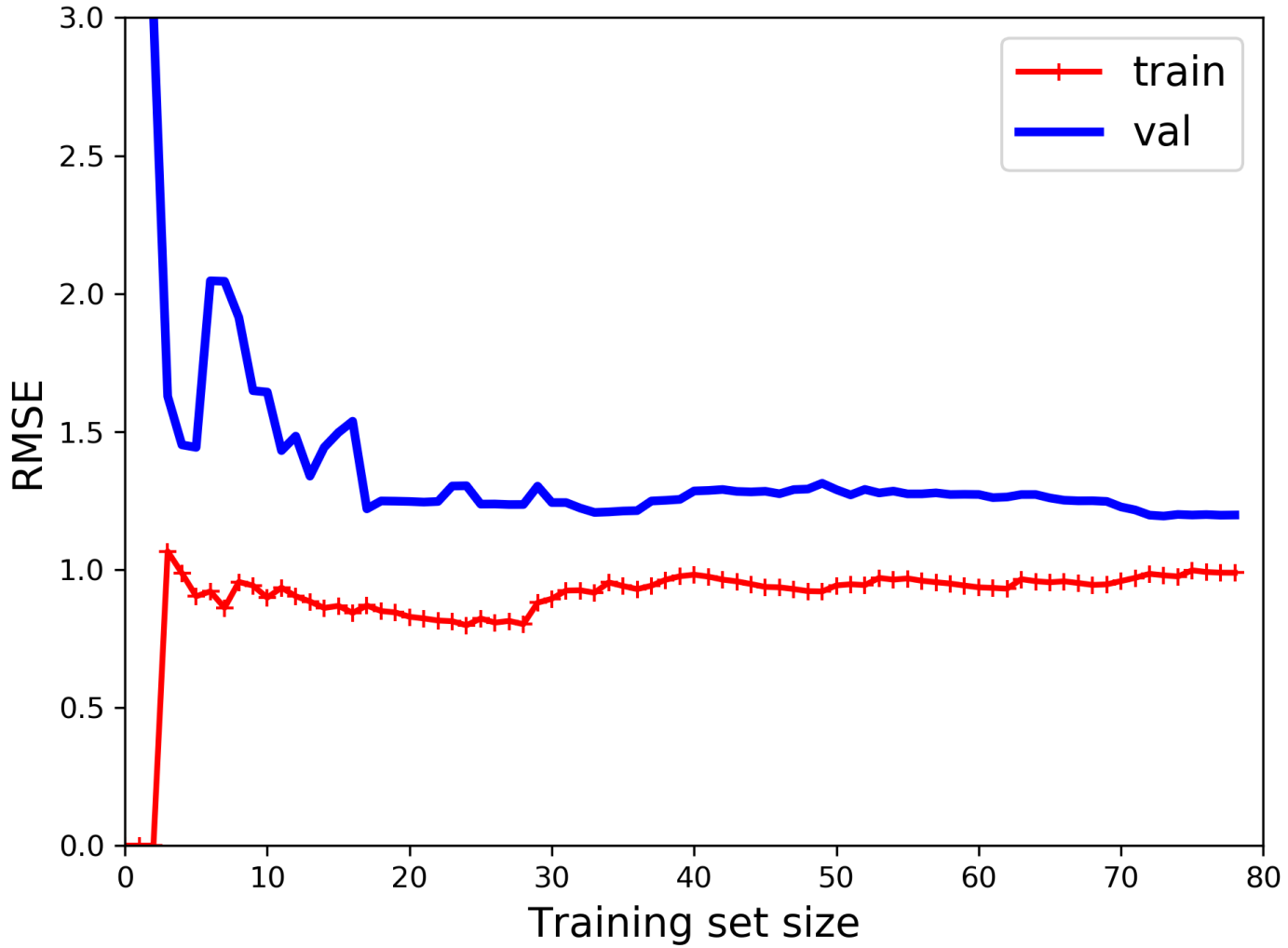
Learning Curves

Linear Regression

SAXION
UNIVERSITY OF
APPLIED SCIENCES



This is a learning curve



Learning curves

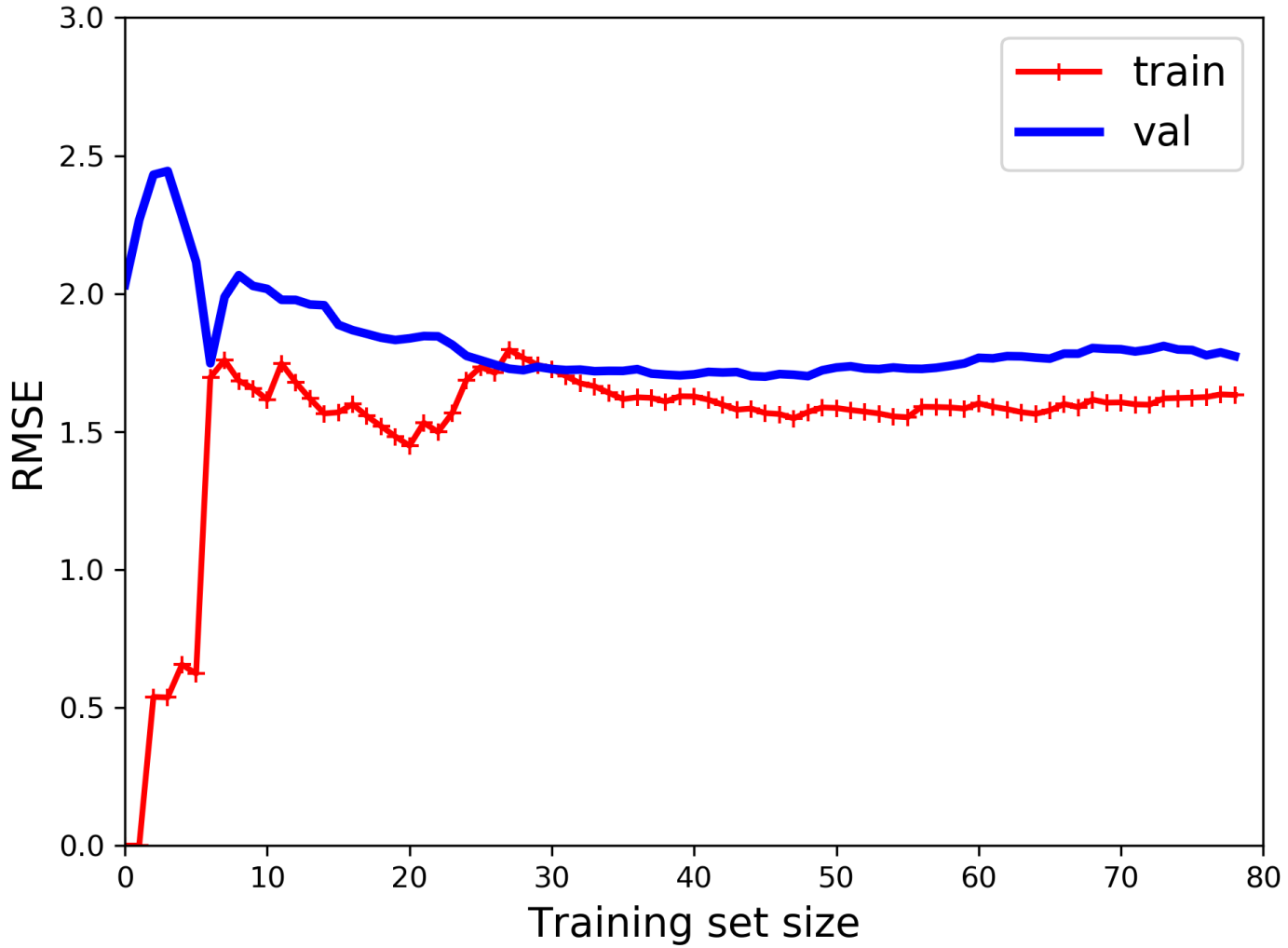
```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

train_sizes = range(1, len(X_train))

train_errors, val_errors = [], []
for m in train_sizes:

  regressor.fit(X_train[:m], y_train[:m])

  y_train_predict = regressor.predict(X_train[:m])
  y_val_predict = regressor.predict(X_val)

  train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
  val_errors.append(mean_squared_error(y_val, y_val_predict))
```

Learning curves

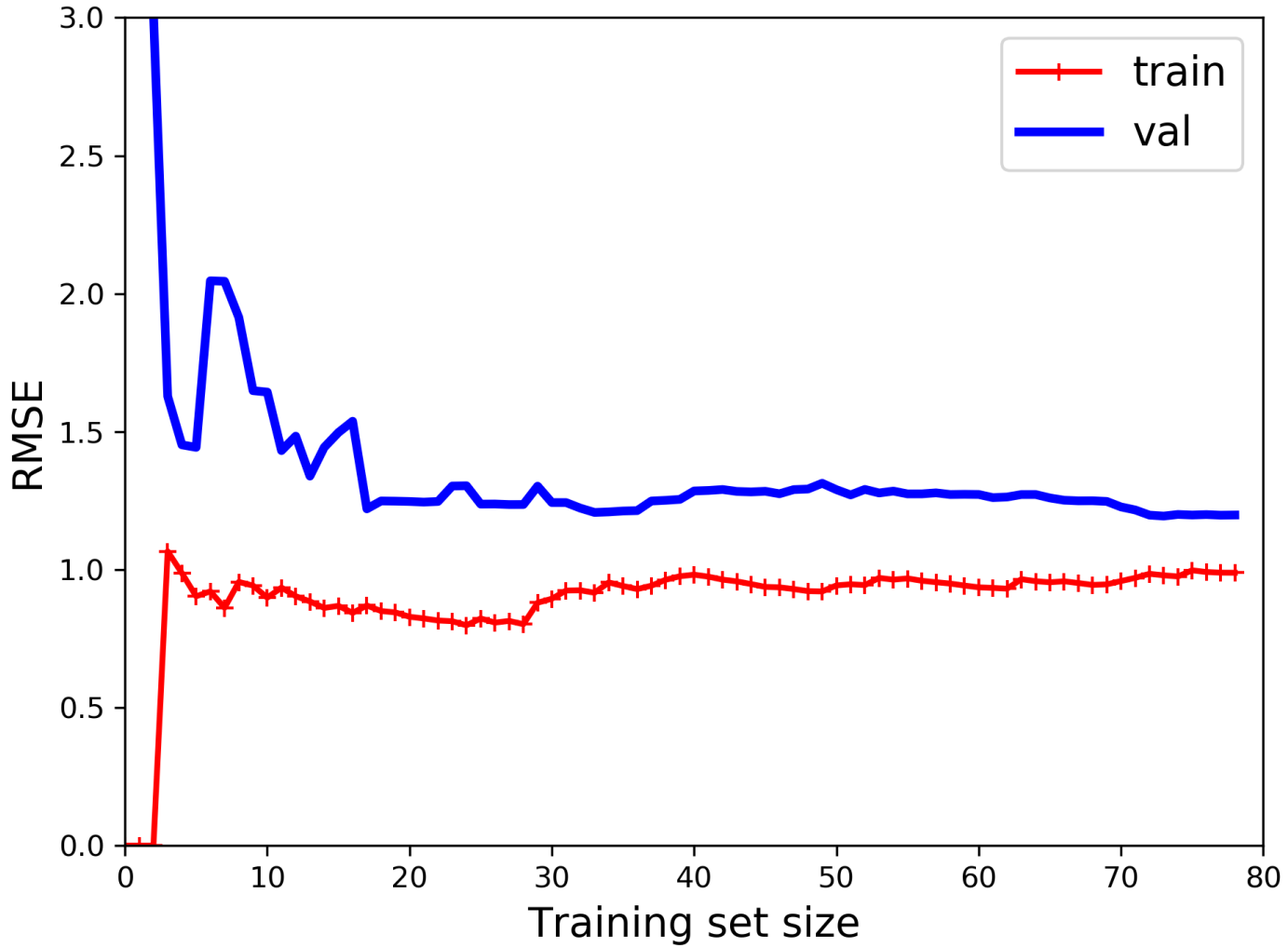
```
# ...
for m in train_sizes:
  regressor.fit(X_train[:m], y_train[:m])
  y_train_predict = regressor.predict(X_train[:m])
  y_val_predict = regressor.predict(X_val)
  train_errors.append(mean_squared_error(y_train[:m], y_train_predict))
  val_errors.append(mean_squared_error(y_val, y_val_predict))

plt.plot(train_sizes, np.sqrt(train_errors), "r-", linewidth=2, label="train")

plt.plot(train_sizes, np.sqrt(val_errors),   "b-", linewidth=2, label="val")

plt.show()
```

There's also **learning_curve** in **sklearn.model_selection**

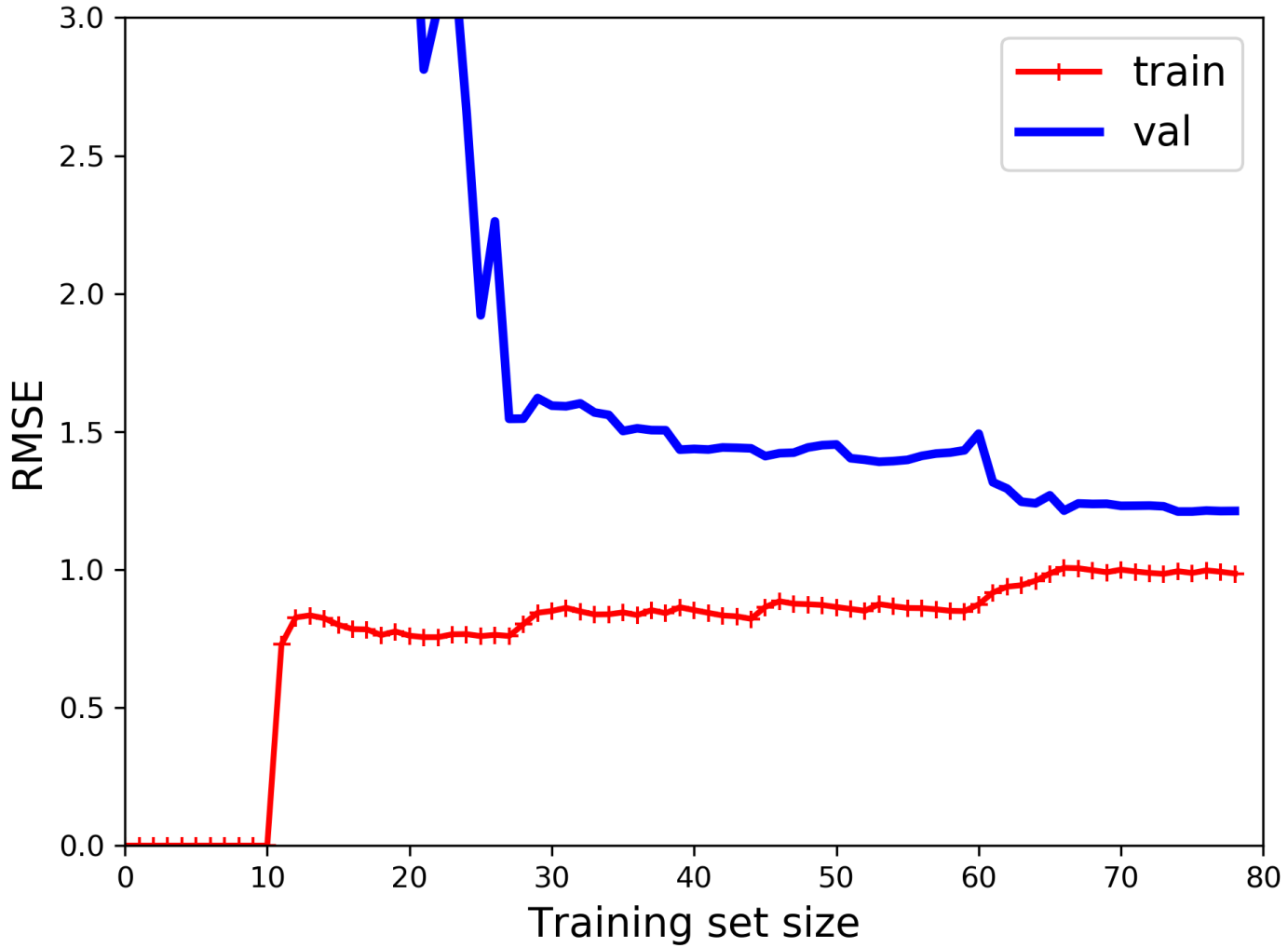
Underfitting



Good fit

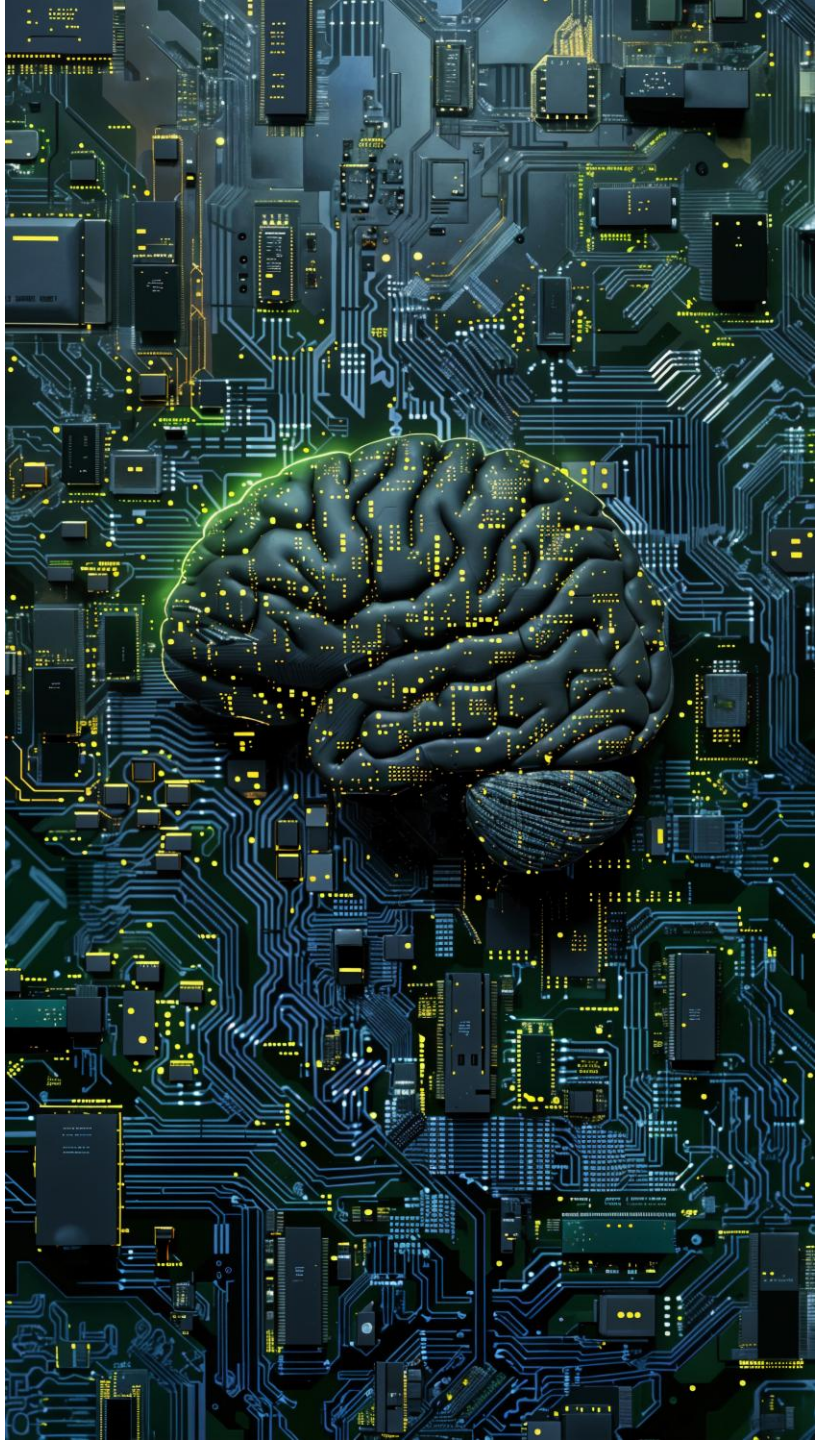

3.0
2.5
2.0
1.5
1.0
0.5
0.0
RMSE
0
10
20
30
40
50
60
70
80
Training set size
train
val

Overfitting



Regularization

Linear Regression



SAXION
UNIVERSITY OF
APPLIED SCIENCES

Overfitting: Regularization

Normal cost function

$$J(\Theta) = MSE(\Theta)$$

L_2 regularization (Ridge Regression)

$$J(\Theta) = MSE(\Theta) + \frac{\alpha}{2}\sum_{i=1}^{n}\theta_i^2$$

Overfitting: Regularization

Normal cost function

$$J(\Theta) = MSE(\Theta)$$

L_1 regularization (Lasso Regression)

$$J(\Theta) = MSE(\Theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Ridge & Lasso regressors

To use Ridge or Lasso:

Mixed L1 & L2

- Either SGDRegressor with **`penalty='l2', 'l1', or 'elasticnet'`**
- Or:

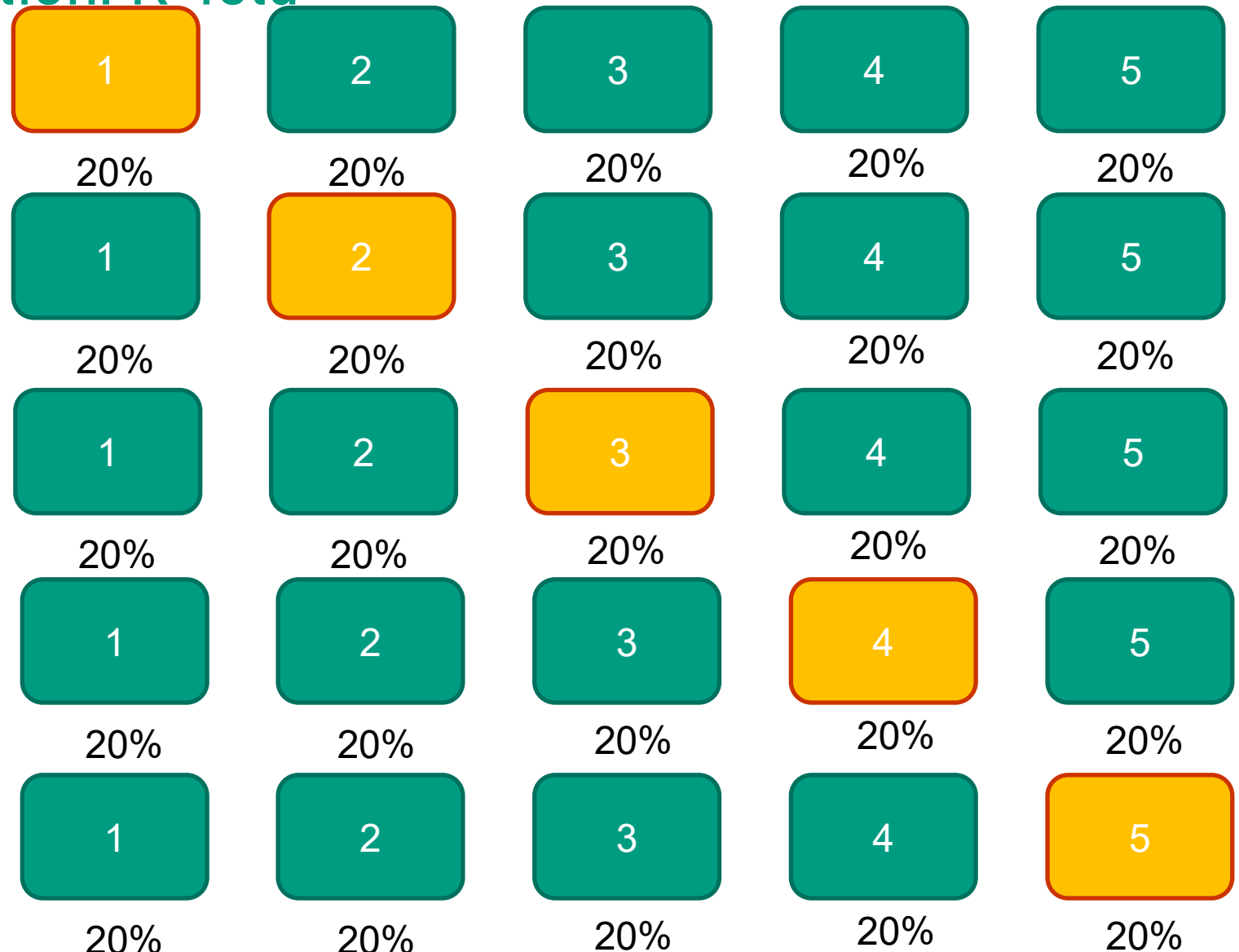
```
from sklearn.linear_model import Ridge, Lasso

regressor = Ridge()
regressor.fit(X, y)
```

Always scale and use regularization with polynomial features

```
regressor = Pipeline([
  ("poly_2", PolynomialFeatures(degree=2),
  ("std_scaler", StandardScaler()),
  ("reg", Ridge()),
])

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

regressor.fit(X_train, y_train)

y_train_pred, y_val_pred = regressor.predict(X_train), regressor.predict(X_val)

print(np.sqrt(mean_squared_error(y_train, y_train_pred)))
print(np.sqrt(mean_squared_error(y_val, y_val_pred)))
```

Cross-validation (auto)

Cross-validation: K-fold



Cross-validation: K-fold

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold

regressor = Ridge()

kf = KFold(n_splits=5)

scores = cross_val_score(regressor, X, y, cv=kf,
                                    scoring='neg_mean_squared_error')

print("MSE: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()))
```

Work, work, work