Machine Learning

Practice 2

Quan Minh Phan & Ngoc Hoang Luong

University of Information Technology (UIT)

April 7, 2021

Table of contents

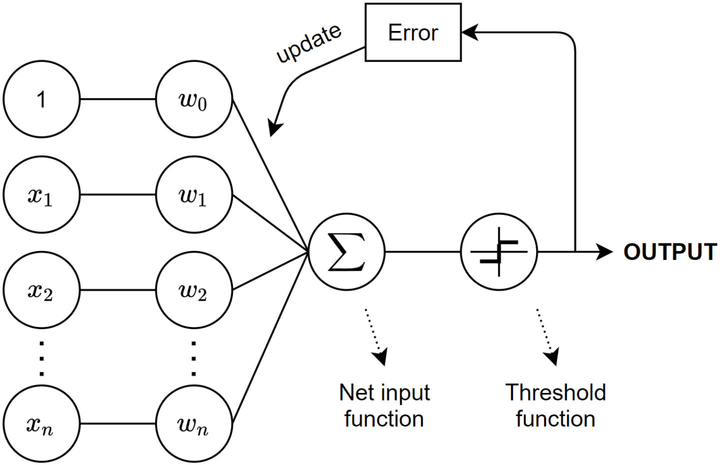


Figure: The general concept of the perceptron

Recall

Problem

Using the perceptron model to classify the species of flowers ('setosa' or 'versicolor') based on the sepal and petal width.

- 2 features: the sepal width x_1; the petal width x_2
  $x = [x_1, x_2]$
- 2 labels: 'setosa' $\leftarrow$ -1; 'versicolor' $\leftarrow$ 1
- $w = [w_1, w_2]$
- Net input function (z)
  $z = w_1 * x_1 + w_2 * x_2$
- Unit step function $(\phi(\cdot))$

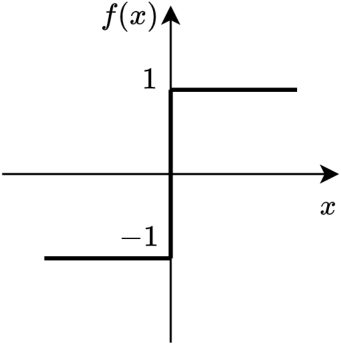
$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta, \\ -1 & \text{otherwise.} \end{cases}$$

Recall (next)

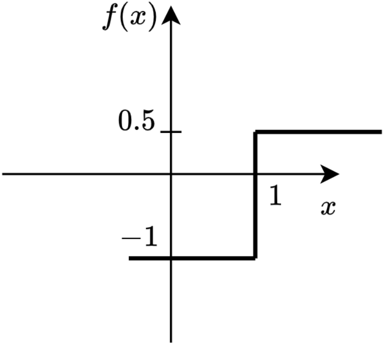
- $w = [w_0, w_1, w_2]$
- Net input function (z)
  $z = w_0 + w_1 * x_1 + w_2 * x_2 = w^T x$
- Unit step function $(\phi(\cdot))$

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0, \\ -1 & \text{otherwise.} \end{cases}$$

Unit Step Function



$$f(x) = \begin{cases} 1 & x \geq 0, \\ -1 & otherwise. \end{cases}$$



$$f(x) = \begin{cases} 0.5 & x \geq 1, \\ -1 & otherwise. \end{cases}$$

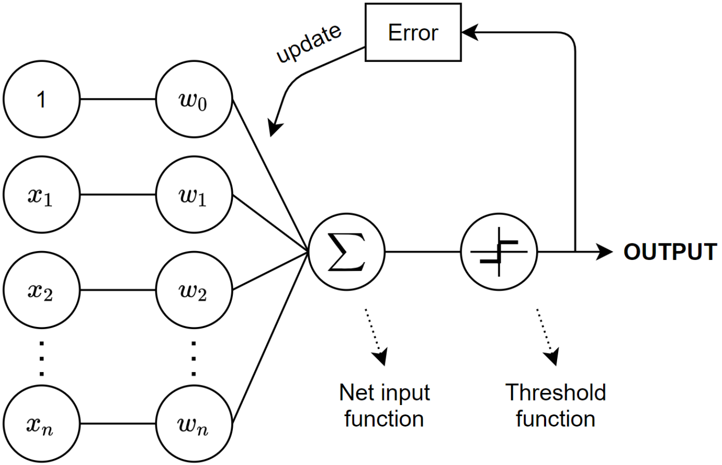


Figure: The general concept of the perceptron

Training process

Algorithm 1 Pseudocode for the training process

```
Initialize the weights, w
while Stopping Criteria is not satisfied do
    for x ∈ X do
        Compute the output value, ŷ
        Updates the weights
    end for
end while
```

Updating the weights

- $w = w + \Delta w$
- $\Delta w_i = \eta * (y - \hat{y}) * x_i$
  where:
  - η: learning rate
  - y: the true class label
  - $\hat{y}$: the predicted class label

Examples

$\Delta w_0 = \eta * (y - \hat{y})$
$\Delta w_1 = \eta * (y - \hat{y}) * x_1$
$\Delta w_2 = \eta * (y - \hat{y}) * x_2$

Components

Hyperparameters

- eta → the learning rate
- max_iter → the maximum number of epochs
- random_state → to make the reproducible results

Parameters

- w → the weights of model
- errors → to store the error in each epoch

Methods

- fit(X, y) → to train the model
- predict(X) → to predict the output value
- net_input(X) → to combine the features with the weights

Implement (code from scratch)

```
class Peceptron_:
    def __init__(self, eta = 0.01, max_iter = 20, random_state = 1):
        self.eta = eta
        self.max_iter = max_iter
        self.random_state = random_state
        self.w = None
        self.errors = [ ]
```

```
    def net_input(self, X):
        return np.dot(X, self.w[1:]) + self.w[0]
```

```
    def predict(self, X):
        return np.where(self.net_input(X) ≥ 0.0, 1, -1)
```

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w = rgen.normal(loc = 0.0, scale = 0.01, size = 1 + X.shape[1])
    self.errors = [ ]
    for n_iter in range (self.max_iter):
        n_wronglabels = 0
        idx = rgen.permutation(len(y))
        X, y = X[idx], y[idx]
        for xi, yi in zip(X, y):
            error = yi - self.predict(xi)
            self.w[1:] += self.eta * error * xi
            self.w[0] += self. eta * error
            n_wronglabels += int(error != 0.0)
        self.errors.append(n_wronglabels)
```

Implement (library)

```
from sklearn.linear_model import Perceptron
```

Hyperparameters

- eta
- max_iter
- random_state

Parameters

- coef_
- intercept_

Methods

- fit(X, y)
- predict(X)

Practice

- Using 'iris.csv' dataset
- How can we use the 'sepal length' and 'sepal width' to classify the speices of flower?

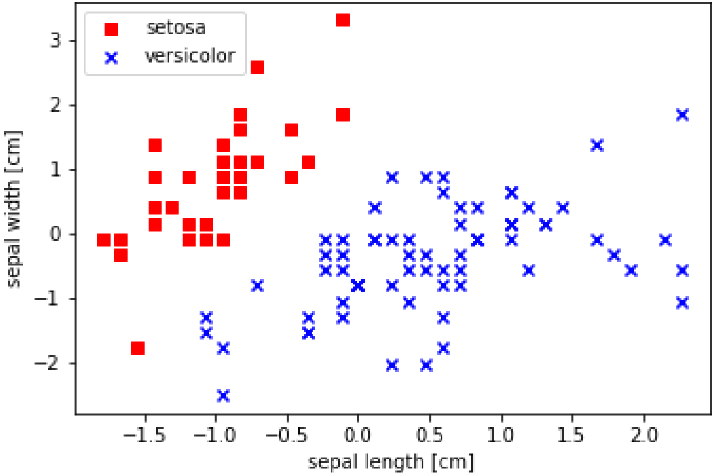
Data visualization

```
>> import matplotlib.pyplot as plt
```

```
>> idx_setosa = y_train == -1
   idx_versicolor = y_train != -1
```

```
>> plt.scatter(X_train[idx_setosa, 0], X_train[idx_setosa, 1], color='red',
   marker='s', label='setosa')
   plt.scatter(X_train[idx_versicolor, 0], X_train[idx_versicolor, 1],
   color='blue', marker='x', label='versicolor')
   plt.xlabel('sepal length [cm]')
   plt.ylabel('sepal width [cm]')
   plt.legend(loc='upper left')
   plt.show()
```

Data visualization



Practice

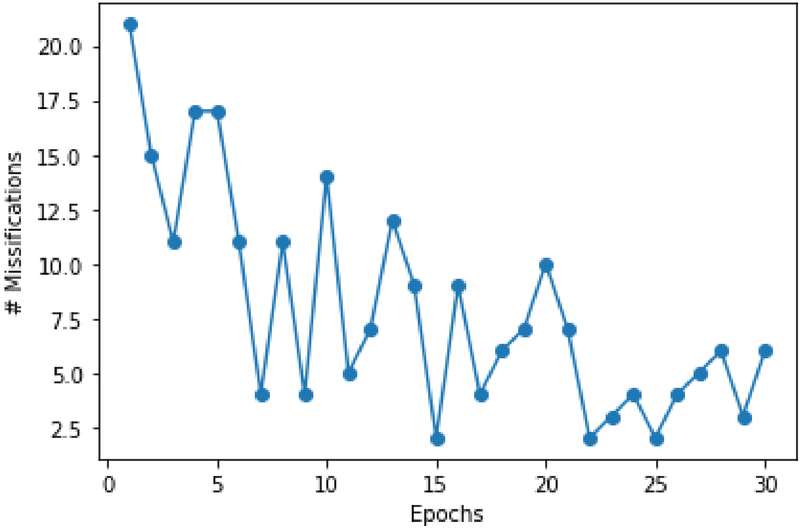
```
>> ppn = Perceptron_(eta = 0.001, max_iter = 30, random_state = 1)
   ppn.fit(X_train, y_train)
```

```
>> from sklearn.linear_model import Perceptron
>> ppn_ = Perceptron(eta = 0.001, max_iter = 30, random_state = 1)
   ppn_.fit(X_train, y_train)
```

Plotting the errors

```
>> plt.plot(range(1, len(ppn.errors) + 1), ppn.errors, marker='o')
   plt.xlabel('Epochs')
   plt.ylabel('# Missifications')
   plt.show()
```

Plotting the errors



Practice

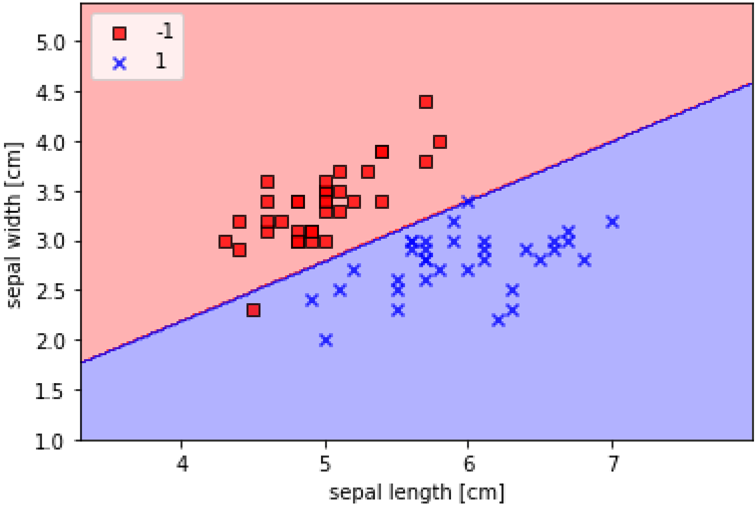
```
>> w_ppn = ppn.w
   w_ppn
>> [-0.01575655   0.06508244   -0.11108172]
```

```
>> w_ppn_ = np.append(ppn_.intercept_, ppn_.coef_)
   w_ppn_
>> [-0.006   0.0196   -0.0325]
```

Visualization

```
>> plot_decision_regions(X_train, y_train, classifier=ppn)
   plt.xlabel('sepal length [cm]')
   plt.ylabel('sepal width [cm]')
   plt.legend(loc='upper left')
   plt.show()
```

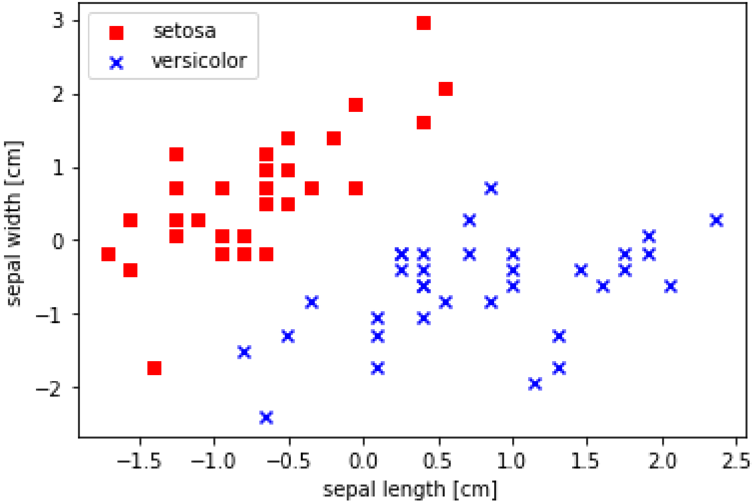
Visualization



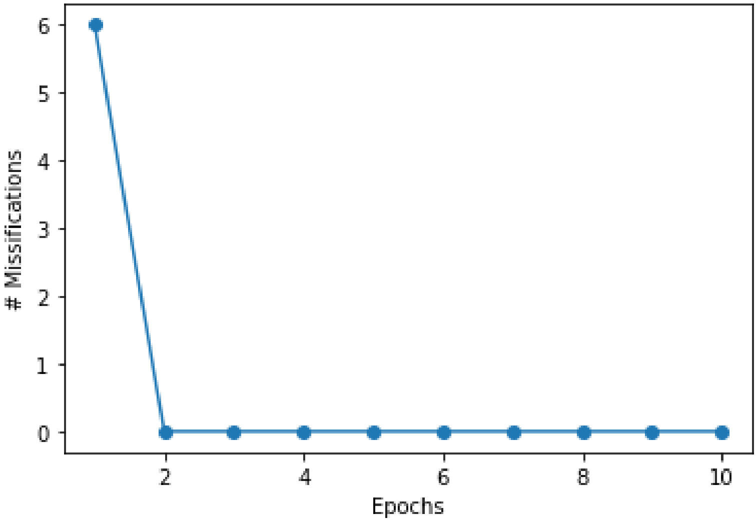
Practice (next)

- Create a new model and train it on data after standardizing

Data visualization



Plotting the costs



Plotting the results

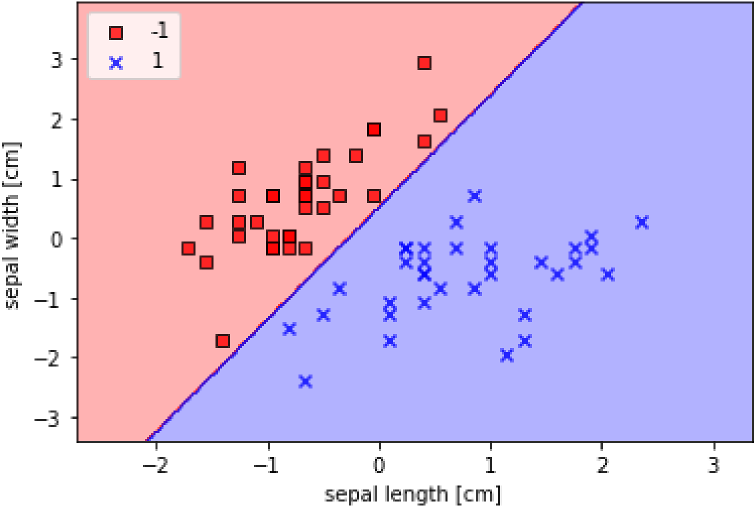


Table of contents

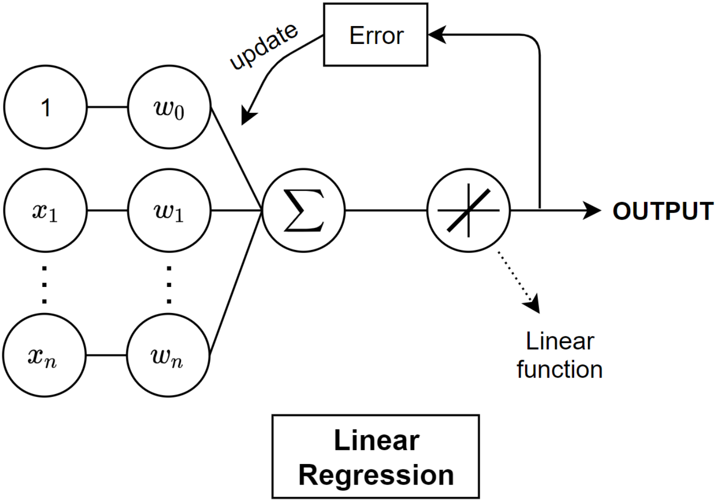


Figure: The general concept of Linear Regression

Minimizing cost functions with gradient descent

Cost function:

$$J(w) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

Update the weights:

$$w := w + \Delta w$$

$$\Delta w = -\eta \nabla J(w)$$

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

Minimizing cost functions with gradient descent

$$w_j = \begin{cases} w_j + \eta * X^T.dot((y - \phi(z)) & j \in [1, \dots, n] \\ w_j + \eta * sum(y - \phi(z)) & j = 0 \end{cases}$$

Pseudocode of Training process

Algorithm 2 Gradient Descent

```
Initialize the weights, w
while Stopping Criteria is not satisfied do
    Compute the output value, ŷ
    Updates the weights
end while
```

Components

Hyperparameters

- eta
- max_iter
- random_state

Parameters

- w
- costs

Methods

- fit(X, y)
- predict(X)
- net_input(X)

Implement (code from scratch)

```
class LinearRegression_GD:
    def __init__(self, eta = 0.001, max_iter = 20, random_state = 1):
        self.eta = eta
        self.max_iter = max_iter
        self.random_state = random_state
        self.w = None
        self.costs = [ ]
```

```
    def net_input(self, X):
        return np.dot(X, self.w[1:]) + self.w[0]
```

```
    def predict(self, X):
        return self.net_input(X)
```

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w = rgen.normal(loc = 0.0, scale = 0.01, size = 1 + X.shape[1])
    self.costs = [ ]
    for n_iters in range (self.max_iter):
        errors = y - self.predict(X)
        self.w[1:] += self.eta * X.T.dot(error)
        self.w[0] += self.eta * error.sum()
        cost = (error**2).sum() / 2
        self.costs.append(cost)
```

Implement (library)

Stochastic Gradient Descent

```
from sklearn.linear_model import SGDRegressor
```

Hyperparameters

- eta0
- max_iter
- random_state

Parameters

- intercept_
- coef_

Methods

- fit(X, y)
- predict(X)

Implement (library)

Normal Equation

```
from sklearn.linear_model import LinearRegression
```

Parameters

- intercept_
- coef_

Methods

- fit(X, y)
- predict(X)

Differences

Gradient Descent

- $w := w + \Delta w$
- $\Delta w = \eta \sum_i (y^{(i)} - \phi(z^{(i)})x^i$

Stochastic Gradient Descent

- $w := w + \Delta w$
- $\Delta w = \eta(y^{(i)} - \phi(z^{(i)})x^i$

Normal Equation

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

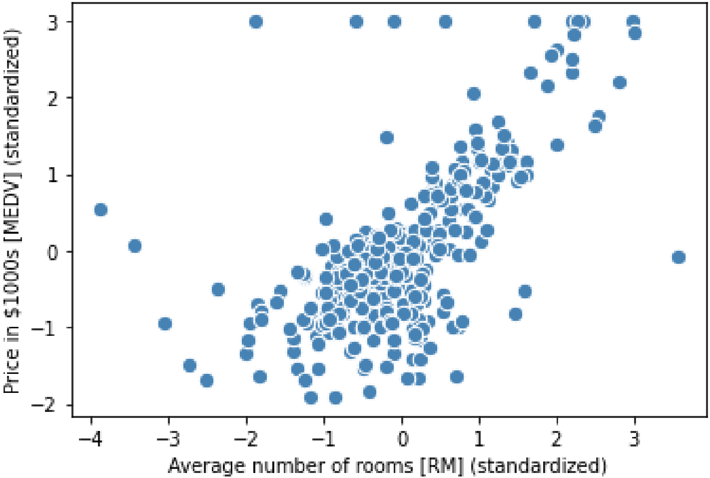
Practice

- Using 'housing.csv' dataset
- How can we use the 'average number of rooms' (RM) to estimate the 'price' of houses (MEDV)?

Plotting data

```
>> plt.scatter(X_train, y_train, c='steelblue', edgecolor='white', s=70)
   plt.xlabel('Average number of rooms [RM] (standardized)')
   plt.ylabel('Price in $1000s [MEDV] (standardized)')
   plt.show()
```

Practice



Price in $1000s [MEDV] (standardized)
3
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2
3
Average number of rooms [RM] (standardized)

Practice

Gradient Descent

```
>> reg_GD = LinearRegression_GD(eta=0.001, max_iter=20,
   random_state=1)
   reg_GD.fit(X_train, y_train)
```

Stochastic Gradient Descent

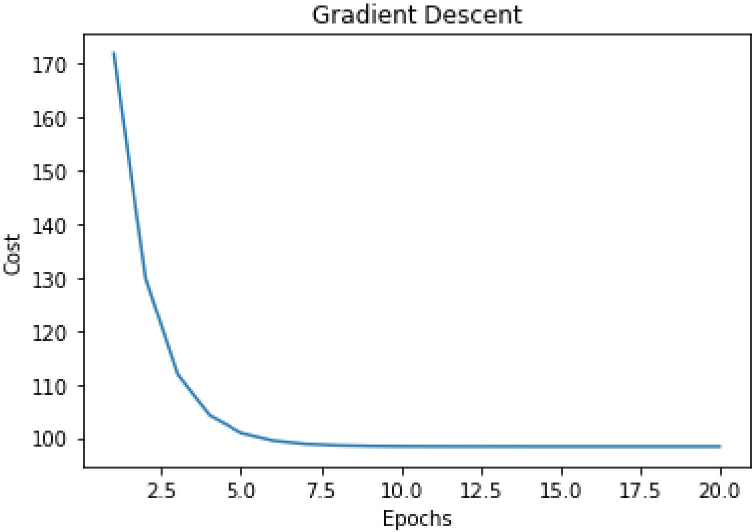
```
>> reg_SGD = SGDRegressor(eta0=0.001, max_iter=20,
   random_state=1, l1_ratio=0, tol=None, learning_rate='constant')
   reg_SGD.fit(X_train, y_train)
```

Normal Equation

```
>> reg_NE = LinearRegression()
   reg_NE.fit(X_train, y_train)
```

Plotting the cost

```
>> plt.plot(range(1, len(reg_GD.costs) + 1), reg_GD.costs)
   plt.xlabel('Epochs')
   plt.ylabel('Cost')
   plt.title('Gradient Descent')
   plt.show()
```


Gradient Descent
Cost
170
160
150
140
130
120
110
100
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

Practice

```
>> w_GD = reg_GD.w
   w_GD
>> [0.00767139   0.64623542]
```

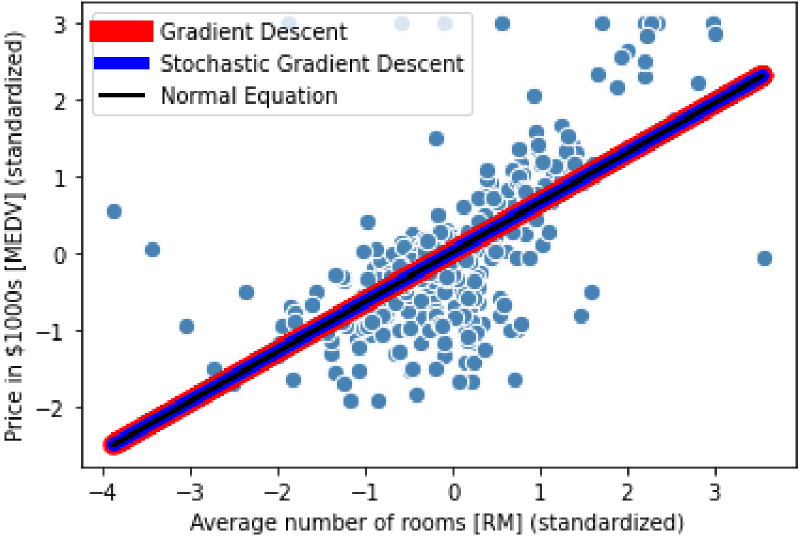
```
>> w_SGD = np.append(reg_SGD.intercept_, reg_SGD.coef_)
   w_SGD
>> [0.00783841   0.64551218]
```

```
>> w_NE = np.append(reg_NE.intercept_, reg_NE.coef_)
   w_NE
>> [0.00773059   0.64638912]
```

Plotting the results

```
>> plt.scatter(X_train, y_train, c='steelblue', edgecolor='white', s=70)
   plt.plot(X_train, reg_GD.predict(X_train), color='red', lw=10,
   label='Gradient Descent')
   plt.plot(X_train, reg_SGD.predict(X_train), color='blue', lw=6,
   label='Stochastic Gradient Descent')
   plt.plot(X_train, reg_NE.predict(X_train), color='black', lw=2,
   label='Normal Equation')
   plt.xlabel('Average number of rooms [RM] (standardized)')
   plt.ylabel('Price in $1000s [MEDV] (standardized)')
   plt.legend()
   plt.show()
```

Plotting the results



Practice

```
>> y_pred_1 = reg_GD.predict(X_test)
```

```
>> y_pred_2 = reg_SGD.predict(X_test)
```

```
>> y_pred_3 = reg_NE.predict(X_test)
```

Performance Evaluation

```
>> from sklearn.metrics import mean_absolute_error as MAE
   from sklearn.metrics import mean_squared_error as MSE
   from sklearn.metrics import r2_score as R2
```

Mean Absolute Error

```
>> print('MAE of GD:', round(MAE(y_test, y_pred_1), 6))
   print('MAE of SGD:', round(MAE(y_test, y_pred_2), 6))
   print('MAE of NE:', round(MAE(y_test, y_pred_3), 6))
```

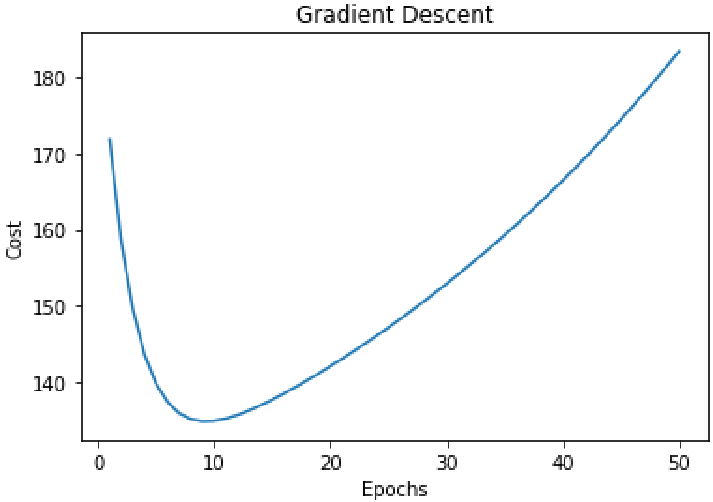
Mean Squared Error

```
>> print('MSE of GD:', round(MSE(y_test, y_pred_1), 6))
   print('MSE of SGD:', round(MSE(y_test, y_pred_2), 6))
   print('MSE of NE:', round(MSE(y_test, y_pred_3), 6))
```

R^2 score

```
>> print('R2 of GD:', round(R2(y_test, y_pred_1), 6))
   print('R2 of SGD:', round(R2(y_test, y_pred_2), 6))
   print('R2 of NE:', round(R2(y_test, y_pred_3), 6))
```

Learning rate too large



Polynominal Regression

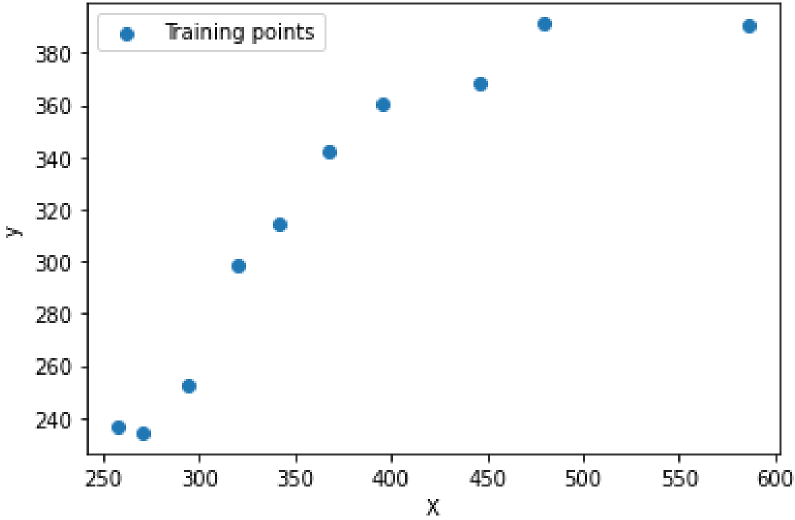
Example

$X = [258.0, 270.0, 294.0, 320.0, 342.0, 368.0, 396.0, 446.0, 480.0, 586.0]$
$y = [236.4, 234.4, 252.8, 298.6, 314.2, 342.2, 360.8, 368.0, 391.2, 390.8]$

```
>> X = np.array([258.0, 270.0, 294.0, 320.0, 342.0, 368.0, 396.0, 446.0,
   480.0, 586.0])[:, np.newaxis]
   y = np.array([236.4, 234.4, 252.8, 298.6, 314.2, 342.2, 360.8, 368.0,
   391.2, 390.8])
```

```
>> plt.scatter(X, y, label='Training points')
   plt.xlabel('X')
   plt.ylabel('y')
   plt.legend()
   plt.show()
```

Plotting data



Polynominal Regression

```
>> from sklearn.linear_model import LinearRegression
   lr = LinearRegression()
   lr.fit(X, y)
```

Polynominal Regression

Syntax

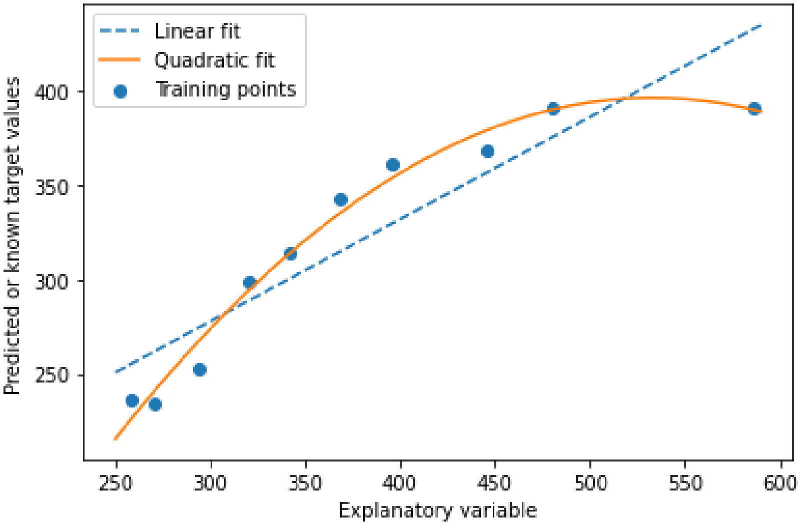
```
from sklearn.preprocessing import PolynomialFeatures
```

```
>> from sklearn.preprocessing import PolynomialFeatures
   pr = LinearRegression()
   quadratic = PolynomialFeatures(degree=2)
   X_quad = quadratic.fit_transform(X)
   pr.fit(X_quad, y)
```

```
>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
```

```
>> y_fit_linear = lr.predict(X_fit)
   y_fit_quad = pr.predict(quadratic.fit_transform(X_fit))
```

```
>> plt.scatter(X, y, label='Training points')
   plt.xlabel('X')
   plt.ylabel('y')
   plt.plot(X_fit, y_fit_linear, label='Linear fit', linestyle='--')
   plt.plot(X_fit, y_fit_quad, label='Quadratic fit')
   plt.legend()
   plt.tight_layout()
   plt.show()
```


Linear fit
Quadratic fit
Training points
400
350
300
250
Predicted or known target values
250
300
350
400
450
500
550
600
Explanatory variable

Practice

Linear regression

```
>> lr = LR()
   lr.fit(X_train, y_train)
```

Polynominal regression (quadratic)

```
>> quadratic = PolynomialFeatures(degree=2)
   X_quad = quadratic.fit_transform(X_train)
   pr_quad = LR() pr_quad = pr_quad.fit(X_quad, y_train)
```

Polynominal regression (cubic)

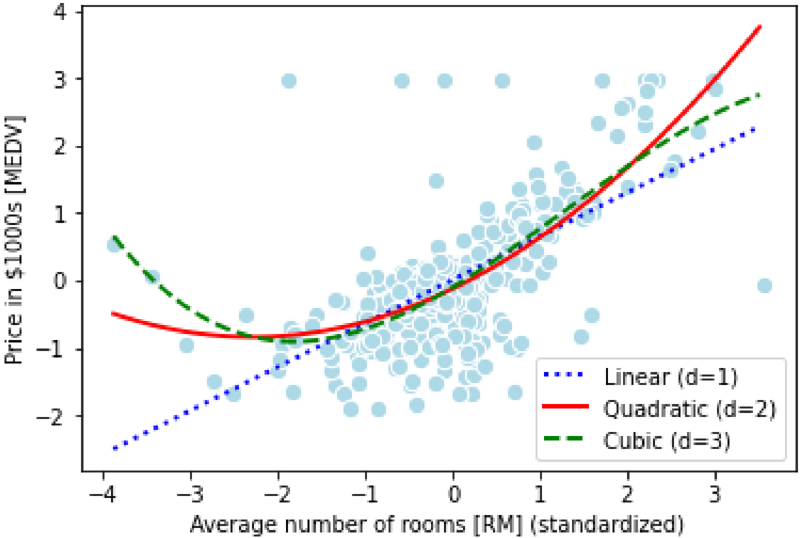
```
>> cubic = PolynomialFeatures(degree=3)
   X_cubic = cubic.fit_transform(X_train)
   pr_cubic = LR() pr_cubic = pr_cubic.fit(X_cubic, y_train)
```

```
>> X_fit = np.arange(X_train.min(), X_train.max(), 0.1)[:, np.newaxis]
```

```
>> y_linear_fit = lr.predict(X_fit)
   y_quad_fit = pr_quad.predict(quadratic.fit_transform(X_fit))
   y_cubic_fit = pr_cubic.predict(cubic.fit_transform(X_fit))
```

Plotting the results

```
>> plt.scatter(X_train, y_train, c='steelblue', edgecolor='white', s=70)
   plt.plot(X_fit, y_lin_fit, label='Linear (d=1)', color='blue', lw=2,
   linestyle=':')
   plt.plot(X_fit, y_quad_fit, label='Quadratic (d=2)', color='red', lw=2,
   linestyle='-')
   plt.plot(X_fit, y_cubic_fit, label='Cubic (d=3)', color='green',
   lw=2,linestyle='--')
   plt.xlabel('Average number of rooms [RM] (standardized)')
   plt.ylabel('Price in $1000s [MEDV] (standardized)')
   plt.legend()
   plt.show()
```



Price in $1000s [MEDV]
Average number of rooms [RM] (standardized)
Linear (d=1)
Quadratic (d=2)
Cubic (d=3)
4
3
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2
3

Table of contents

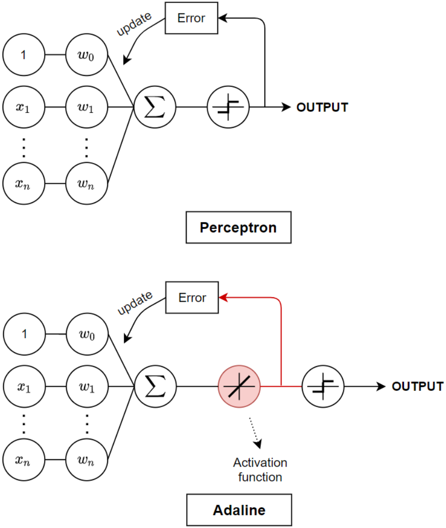


Figure: Differences between Perceptron and Adaline

Training process

Algorithm 3 Pseudocode for the training process

```
Initialize the weights, w
while stopping criteria is not satisfied do
    for x ∈ X do
        Compute the output value, ŷ
        Updates the weights
    end for
end while
```

Updating the weights

- $w = w + \Delta w$
- $\Delta w_i = \eta * (y - \hat{y}) * x_i$
  where:
  - η: learning rate
  - y: the true class label
  - $\hat{y}$: the predicted class label

Examples

$\Delta w_0 = \eta * (y - \hat{y})$
$\Delta w_1 = \eta * (y - \hat{y}) * x_1$
$\Delta w_2 = \eta * (y - \hat{y}) * x_2$

Components

Hyperparameters

- eta
- max_iter
- random_state

Parameters

- w
- costs

Methods

- fit(X, y)
- predict(X)
- net_input(X)
- activation(X)

Implement (code from scratch)

```
class Adaline:
    def __init__(self, eta = 0.01, max_iter = 50, random_state = 1):
        self.eta = eta
        self.max_iter = max_iter
        self.random_state = random_state
        self.w = None
        self.costs = [ ]
```

```
    def activation(self, X):
        return self.net_input(X)
```

```
    def predict(self, X):
        return np.where(self.activation(X) ≥ 0.0, 1, -1)
```

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w = rgen.normal(loc = 0.0, scale = 0.01, size = 1 + X.shape[1])
    self.costs = [ ]
    for n_iter in range (self.max_iter):
        idx = rgen.permutation(len(y))
        X, y = X[idx], y[idx]
        cost = 0
        for xi, yi in zip(X, y):
            error = yi - self.predict(xi)
            self.w[1:] += self.eta * error * xi
            self.w[0] += self.eta * error
            cost += error**2
        cost /= 2
        self.costs.append(cost)
```

Table of contents

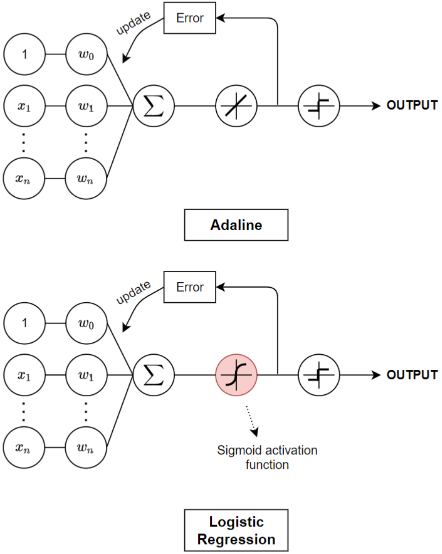


Figure: Differences between Adaline and Logistic regression

Components

Hyperparameters

- eta
- max_iter
- random_state

Parameters

- w
- costs

Methods

- fit(X, y)
- predict(X)
- net_input(X)
- activation(X)

Implement (code from scratch)

```
class LogisticRegression:
    def __init__(self, eta = 0.01, max_iter = 50, random_state = 1):
        self.eta = eta
        self.max_iter = max_iter
        self.random_state = random_state
        self.w = None
        self.costs = [ ]
```

```
    def activation(self, X):
        return 1. / (1. + np.exp(-np.clip(self.net_input(X), -250, 250)))
```

```
    def predict(self, X):
        return np.where(self.activation(X) ≥ 0.5, 1, 0)
```

```
def fit(self, X, y):
    rgen = np.random.RandomState(self.random_state)
    self.w = rgen.normal(loc = 0.0, scale = 0.01, size = 1 + X.shape[1])
    self.costs = [ ]
    for n_iter in range (self.max_iter):
        output = self.activation(X)
        errors = y - output
        self.w[1:] += self.eta * X.T.dot(errors)
        self.w[0] += self.eta * errors.sum()
        cost = (-y.dot(np.log(output)) - ((1 - y).dot(np.log(1 - output))))
        self.costs.append(cost)
```

Practice

```
>> clf_LR = LogisticRegression(eta=0.01, max_iter=20,
   random_state=1)
   clf_LR.fit_SGD(X_train, y_train)
>> y_pred = clf_LR.predict(X_test)
```

Implement (library)

Syntax (import)

```
from sklearn.linear_model import LogisticRegression
```

Examples

```
>> from sklearn.linear_model import LogisticRegression as LogisticRegression_
>> clf_LR_lib = LogisticRegression_(random_state=1)
   clf_LR_lib.fit(X_train, y_train)
>> y_pred_lib1 = clf_LR_lib.predict(X_test)
```