Machine Learning Labs

Carayon Chloé – Taillieu Victor

Regression Class

```python
def normalizeFeatures(self, X, fit=False):
    if fit:
        self.scaling = list(zip(X.min(), X.max()))

    minmax = list(zip(*self.scaling))
    min, max = np.array(minmax[0]), np.array(minmax[1])
    X = np.matrix((X - min) / (max - min))

    return np.insert(X, 0, 1, axis=1)
```

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

```python
def gradientDescent(self, alpha=0.03, threshold=1e-3, iter=1000, autoAlpha=True):
    i = 0
    self.J = []
    self.w = np.matrix([[1] for i in range(self.n + 1)])

    while True:
        i += 1
        self.J.append(self.costFunction())

        self.w = self.w - alpha * self.gradient()

        if len(self.J) > 1:
            if autoAlpha and self.J[-1] > self.J[-2]:
                alpha /= 1.1

            if abs(self.J[-1] - self.J[-2]) < threshold or i == iter:
                break

    self.updateScores()
```

$$w_k = w_k - \frac{\alpha}{m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) x_k^{(i)}$$

$$w = w - \alpha \Delta$$

Linear Regression Class

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

$$J = \frac{1}{2m}(Xw - y)^T(Xw - y)$$

$$\Delta = \frac{1}{m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})x_k^{(i)}$$

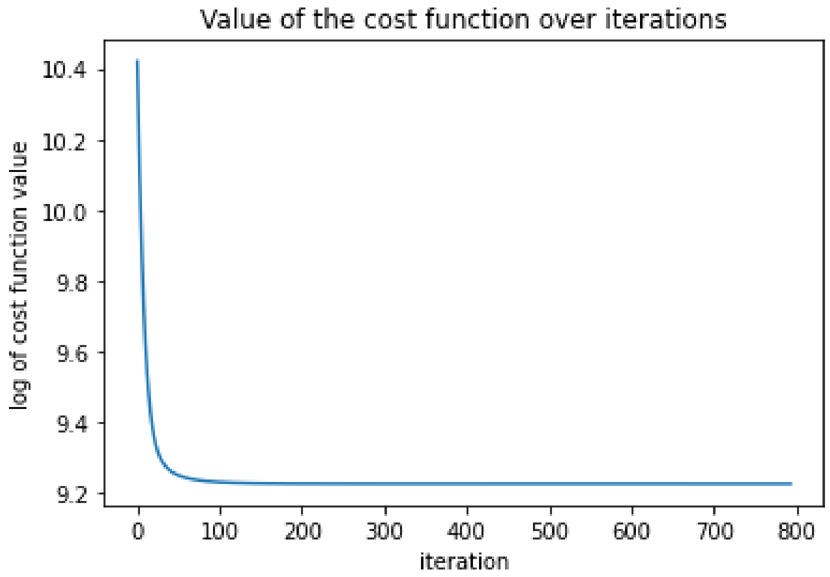
$$\Delta = \frac{1}{m}(X^T(Xw - y))$$

```
class LinearRegression(Regression):
    def costFunction(self):
        X, y, w, m = self.X_train, self.y_train, self.w, self.m
        return float(1 / (2 * m) * (X * w - y).T * (X * w - y))

    def gradient(self):
        X, y, w, m = self.X_train, self.y_train, self.w, self.m
        return 1 / m * X.T * (X * w - y)

    def normalEquation(self):
        X, y = self.X_train, self.y_train
        self.w = (X.T * X)**-1 * X.T * y

        self.updateScores()
```

$$w = (X^TX)^{-1}.(X^Ty)$$

Linear regression Results

Gradient descent




```
Gradient descent coefficients:
[[  15.25614432]
 [  62.16363578]
 [  70.1350214 ]
 [ -10.19451843]
 [ 176.85420204]
 [ -27.4104932 ]
 [   4.30315651]
 [ -11.33107788]
 [ 273.63532282]
 [-198.095751  ]
 [  20.9882641 ]
 [  -3.36688933]
 [  24.98548201]]
MAE: 106.66332152370752
R2:
Train: 0.38761758431685356
Test: 0.3855197492585182
```

Normal Equation

```
Normal equation coefficients:
[[  16.82540449]
 [  62.01144869]
 [ 116.58965098]
 [  32.48776091]
 [ 176.96206809]
 [ -27.50480882]
 [   4.30596978]
 [ -11.2490898 ]
 [ 273.51027645]
 [-198.28934502]
 [  20.67036957]
 [  -3.39974329]
 [ -67.35622144]]
MAE: 106.66142249364468
R2:
Train: 0.3876553087897192
Test: 0.38567137931379636
```

Scikit Learn

```
Scikit-learn
[[  16.82540449]
 [  62.01144869]
 [ 116.58965098]
 [  32.48776091]
 [ 176.96206809]
 [ -27.50480882]
 [   4.30596978]
 [ -11.2490898 ]
 [ 273.51027645]
 [-198.28934502]
 [  20.67036957]
 [  -3.39974329]
 [ -67.35622144]]
MAE: 106.661422493657l
Train: 0.38765530878971943
Test: 0.38567137931378104
```

Logistic Regression Class

$$S(z) = h_w(z) = \frac{1}{1+e^{-z}}, z = Xw$$

$$h_w(Xw) = \frac{1}{1+e^{-Xw}}$$

```
class LogisticRegression(Regression):
    def sigmoid(self, X):
        return 1 / (1 + np.exp(-X * self.w))

    def costFunction(self):
        y, m, sigmoid = self.y_train, self.m, self.sigmoid(self.X_train)
        return float(-1 / m * (y.T * np.log(sigmoid) + (1 - y.T) * np.log(1 - sigmoid)))

    def gradient(self):
        X, y = self.X_train, self.y_train
        return X.T * (self.sigmoid(X) - y)
```

$$J = \frac{-1}{m}(y^T log(h_w(x^{(i)})) + (1 - y^T) log(1 - h_w(x^{(i)})))$$

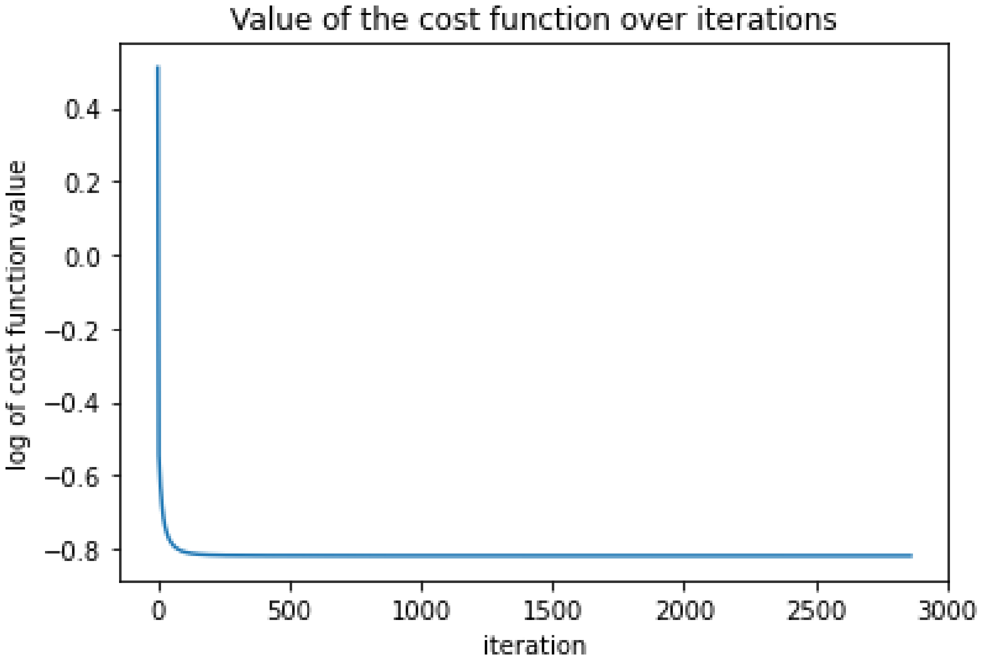
$$\Delta = \sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})x_k^{(i)}$$

$$\Delta = (X^T.(sigmoid(X) - y))$$

Logistic Regression Results

Gradient descent

```
Logistic model
[[-2.82598959]
 [-0.07721819]
 [ 0.45676453]
 [ 0.22917433]
 [ 0.63103143]
 [ 0.06463871]
 [ 0.7269864 ]
 [ 0.75345777]
 [-0.30125548]
 [-0.59108348]
 [ 3.85793586]
 [ 0.16918771]]
```




```
Confusion matrix:
 [[ 71  76]
 [  7 334]]
Accuracy: 0.8299180327868853
Precision: 0.8146341463414634
Recall: 0.9794721407624634
F1-score: 0.8894806924101198
```

Scikit Learn

```
Scikit-Learn
[[-2.44816876]
 [-0.06222258]
 [ 0.4205428 ]
 [ 0.17793724]
 [ 0.57452944]
 [ 0.08444219]
 [ 0.11094913]
 [ 0.09099877]
 [-0.06057049]
 [-0.33916367]
 [ 3.32797682]
 [ 0.15472534]]

Confusion matrix:
 [[ 71  76]
 [  7 334]]
Accuracy: 0.8299180327868853
Precision: 0.8146341463414634
Recall: 0.9794721407624634
F1-score: 0.8894806924101198
```

Neural Network Layers

$$a^i = sigmoid(z^i)$$
$$z^i = W^{i-1}a^{i-1}$$

```
class Layer:
    def __init__(self, n_inputs, n_neurons, random_state=42):
        np.random.seed(random_state)
        self.w = np.matrix(np.random.randn(n_neurons, n_inputs + 1))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_deriv(self, x):
        return np.multiply(x, (1 - x))

    def forward(self, inputs):
        z = self.w * inputs
        a = self.sigmoid(z)
        self.output = np.insert(a, 0, 1, axis=0)

    def backward(self, next_w, next_error):
        deriv = self.sigmoid_deriv(self.output)
        error = np.multiply(next_w.T * next_error, deriv)
        self.error = np.delete(error, 0, axis=0)
```

```
class OutputLayer(Layer):
    def forward(self, inputs):
        z = self.w * inputs
        a = self.sigmoid(z)
        self.output = a

    def backward(self, y):
        self.error = self.output - y
```

$$\delta^n = a^n - y$$

```
class InputLayer(Layer):
    def __init__(self):
        pass

    def forward(self, inputs):
        self.output = np.insert(inputs, 0, 1, axis=0)
```

$$i < n : \delta^i = (W^i)^T . delta^{i+1} * sigmoid'(W^i a^i)$$

Deep Neural Network Class

$$J = \frac{-1}{m}[\sum_{i=1}^{m}\sum_{k=1}^{k} y_k^i log(h_w(x^i))_k + (1-y_k^i)log(1-h_w(x^i))_k]$$

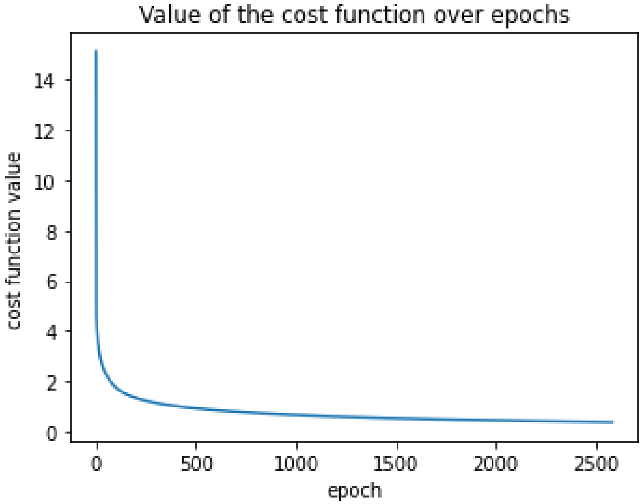
```python
def costFunction(self, y_pred):
    y, m = self.y_train, self.m

    return -1 / m * np.sum(np.multiply(y, np.log(y_pred))
                           + np.multiply((1 - y), np.log(1 - y_pred)))

def forward(self, x):
    inputs = x
    for l in range(len(self.layers)):
        self.layers[l].forward(inputs)
        inputs = self.layers[l].output

def backward(self):
    self.layers[-1].backward(self.y_train)
    next_w, next_error = self.layers[-1].w, self.layers[-1].error
    for l in range(len(self.layers) - 2, 0, -1):
        self.layers[l].backward(next_w, next_error)
        next_w, next_error = self.layers[l].w, self.layers[l].error

def gradient(self):
    grads = []
    for l in range(len(self.layers) - 1):
        grads.append(self.layers[l + 1].error * self.layers[l].output.T)

    return grads
```

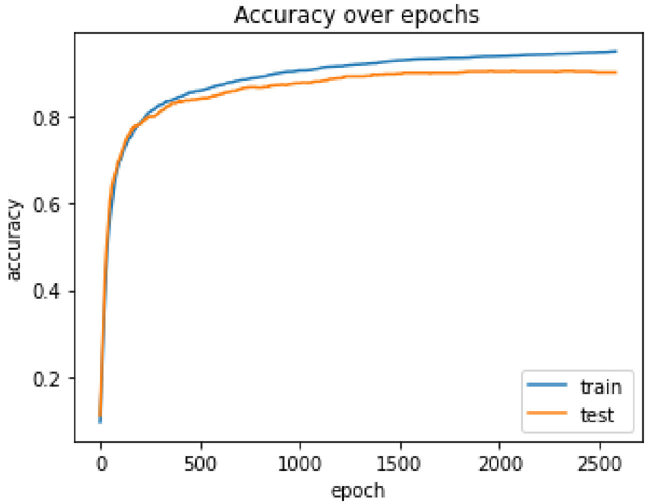
$$\Delta_{ij}^l = \delta_i^{l+1}(a_j^l)^T$$

```python
def gradientDescent(self, alpha=1e-3, threshold=1e-5, epochs=1000):
    i = 0
    self.J = []
    self.train_accuracy = []
    self.test_accuracy = []

    while True:
        i += 1
        self.forward(self.X_train)
        self.backward()

        self.J.append(self.costFunction(self.layers[-1].output))

        grads = self.gradient()

        for l in range(len(self.layers) - 1):
            self.layers[l + 1].w -= alpha * grads[l]

        self.train_accuracy.append(self.accuracy(self.X_train, self.labels_train))
        self.test_accuracy.append(self.accuracy(self.X_test, self.labels_test))

        if len(self.J) > 1:
            if abs(self.J[-1] - self.J[-2]) < threshold or i == epochs:
                break
```

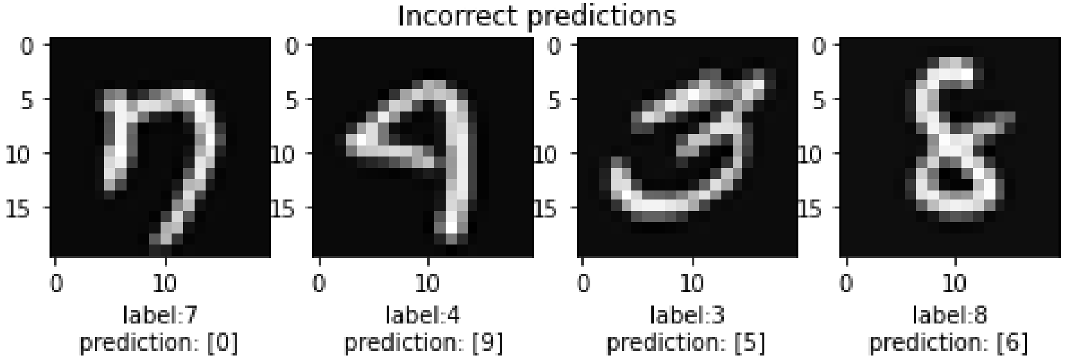
$$w^l = w^l - \alpha\Delta^l$$

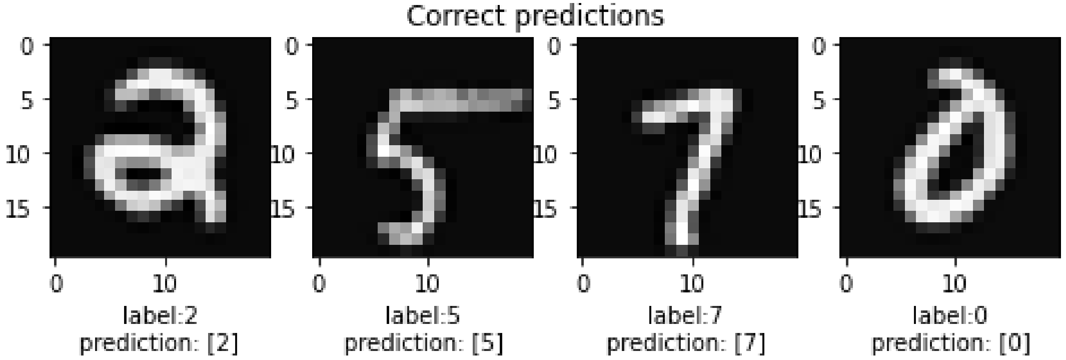
Neural Network Results






```
Train accuracy: 0.95175
Test accuracy: 0.903
```






```
DNN:
Test accuracy:  0.903
[[106   0   0   0   0   1   2   0   1   1]
 [  0 100   1   0   0   4   0   0   0   0]
 [  1   0  94   4   3   0   1   0   1   1]
 [  0   0   3  74   1   2   1   1   2   1]
 [  0   0   2   0  83   0   3   0   1   5]
 [  2   0   0   1   1  90   1   0   3   0]
 [  2   0   4   0   2   1  88   0   1   0]
 [  1   2   3   1   2   0   0 100   1   2]
 [  1   1   0   3   0   1   2   0  73   0]
 [  0   0   1   1   5   1   0   5   3  95]]
```

```
Keras:
Test accuracy:  0.9229999780654907
[[109   0   0   0   0   1   0   0   0   1]
 [  0 101   1   0   0   2   0   0   0   1]
 [  1   1  91   3   5   1   1   1   1   0]
 [  0   0   3  76   0   1   1   1   2   1]
 [  0   0   2   0  84   0   2   0   1   5]
 [  0   2   0   0   1  93   0   0   1   1]
 [  2   0   0   0   0   2  93   0   1   0]
 [  1   3   2   0   2   0   0 100   0   4]
 [  1   2   1   0   0   4   1   0  72   0]
 [  0   0   0   2   0   1   0   3   1 104]]
```

Conclusion