Aprendizagem de máquina com Python

Resumo do *notebook* "Fundamentos de Python para aprendizagem de máquina"

Uso básico da linguagem de programação Python para implementar modelos de aprendizagem de máquina.

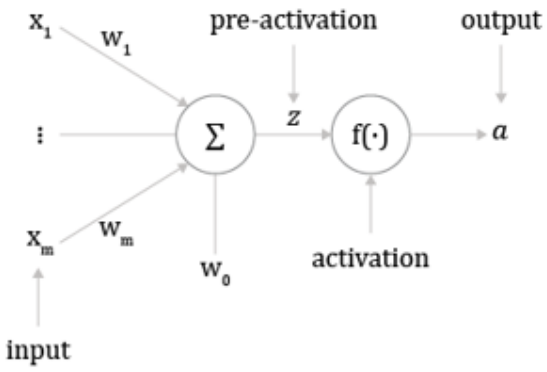
Escopo: Este documento é um resumo do *notebook* "Fundamentos de Python para aprendizagem de máquina", do módulo 2 do curso "Deep Learning: domínio das redes neurais", do MIT xPRO. Este resumo traz uma visão geral dos aspectos principais desse *notebook*.

Índice

1. Modelos de neurônio único

Modelos de neurônio único são compostos por um só neurônio, ou nó, que soma as multiplicações ponderadas dos atributos de uma amostra de entrada, adiciona um termo de viés e introduz essa soma em uma função de ativação.

A fórmula e o diagrama a seguir fornecem a notação matemática formal para um modelo de neurônio único:



$$z = \sum_{j=1}^{m} x_j w_j + w_0$$

$$z = xw^t + w_0$$

$$a = f(z)$$

Modelos de neurônio único podem ser automatizados com o uso da linguagem de programação Python. Os *scripts* em Python podem executar métodos de aprendizagem de máquina como **regressão** e **classificação**.

Lembre-se:

- Modelos de regressão são usados para prever um valor real $\hat{y}^{(i)} = f\left(x^{(i)}\right)$ para um ponto de dados *i* (*i* indica qual ponto de dados está sendo considerado).
- Os modelos de classificação apresentados aqui implementam um classificador binário – o que também é conhecido como regressão logística – para prever uma variável-alvo de natureza binária. Regressões logísticas contêm pares de dados do tipo entrada-saída $(x^{(i)}, y^{(i)})$ que são usados para gerar rótulos de saída para uma

entre apenas duas classes (valores entre 0 e 1). Portanto, em modelos de classificação, a função de ativação é sigmoide.

2. Modelo de regressão de neurônio único em Python

O seguinte *script* de código implementa a configuração necessária para um modelo de regressão de neurônio único em Python. O código usa a função linear *f(z) = z* como função de ativação.

```
# Primeiro, precisamos importar pacotes que serão usados para visualização.
# Isso pode ser ignorado por enquanto, pois será explicado em um futuro notebook
import numpy as np
import math, random
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Definir a função de ativação
def single_neuron_regression_model(w, w_0, x):
    # Executar o produto escalar na entrada x e nos pesos w
    z = 0
    for feature, weight in zip(x, w):
        z += feature * weight
    z += w_0  # Adicionar o termo de viés

    # Aplicar a função de ativação e devolver
    a = linear(z)
    return a

# Função de ativação linear simples: só devolve identidade
def linear(z):
    return z
```

Importante: A função `single_neuron_regression_model` está escrita no *script* para dados de entrada `x` que possam ter qualquer número de atributos. Portanto, o modelo pode usar um conjunto de dados com um vetor `x` de quaisquer dimensões, desde que os pesos `w` tenham extensão igual.

Aqui, zip(a,b) irá gerar tuplas de elementos correspondentes em a e b. Por exemplo, list(zip([1,2,3],[10,20,30])) devolverá [(1,10),(2,20),(3,30)].

Em seguida, atribuímos valores às variáveis de `single_neuron_regression_model(w, w_0, x)` para testar o modelo. O exemplo a seguir usa um ponto de dados bidimensional `x`.

```
# Testar a saída do modelo para um único ponto de dados em 2D
x = [1, 2]
w = [5, 3]
w_0 = -8

y = single_neuron_regression_model(w, w_0, x)
print("input", x, "=> output", y)

# Resultado de print
input [1, 2] => output 3
```

2.a. Treinamento de um modelo de regressão – descida do gradiente

Qualquer modelo de regressão deve ser treinado para fornecer previsões precisas.

O método de descida do gradiente ajuda a determinar os pesos w e w_0 que minimizam a função de custo do modelo.

Uma redução na função de custo significa um desvio menor em relação aos valores previstos $\hat{y}^{(i)}$ e os valores corretos $y^{(i)}$.

A fórmula a seguir traz a definição matemática formal da função de custo J, na qual L é a perda por ponto de dados:

$$J = \sum_{i}^{n} L(\hat{y}^{(i)}, y^{(i)})$$

Procedimento de treinamento por descida do gradiente:

O modelo executa *loops* de treinamento, também conhecidos como épocas, para um número especificado de iterações ao longo do conjunto de dados.

O modelo examina os pares de entradas e saídas em cada época:

1. Estimativa da perda L entre os valores previstos $\hat{y}^{(i)}$ e os valores corretos $y^{(i)}$
2. Cálculo do gradiente da perda L em relação a cada peso
3. Atualização dos pesos com base no gradiente e na taxa de aprendizagem

Em outras palavras, o algoritmo de descida do gradiente faz com que o modelo seja capaz de adaptar seus pesos para encontrar o ponto mais baixo na função de custo. Ou seja, o erro menor.

O *script* a seguir executa o procedimento de treinamento por descida do gradiente com uma função de perda de erro quadrático.

```
# Definir a função de treinamento com perda de erro quadrático
def train_model_SE_loss(model_function, w, w_0,

                        input_data, output_data,
                        learning_rate, num_epochs):
    for epoch in range(num_epochs):
        total_loss = 0 # Acompanhar a perda total ao longo do conjunto de dados
        for x, y in zip(input_data, output_data):
            y_predicted = model_function(w, w_0, x)
            error = y_predicted - y
            total_loss += (error**2)/2

            # Atualizar o coeficiente de viés usando gradiente em relação a w_0
            w_0 -= learning_rate * error * 1

            # Atualizar outros coeficientes do modelo
            # usando gradiente em relação a cada coeficiente
            for j, x_j in enumerate(x):
                w[j] -= learning_rate * error * x_j

        # Informar o progresso de poucas em poucas épocas
        report_every = max(1, num_epochs // 10)
        if epoch % report_every == 0:
            print("epoch", epoch, "has total loss", total_loss)

    return w, w_0
```

Além do treinamento de pesos, a função a seguir avalia o desempenho do modelo de regressão treinado calculando sua precisão (accuracy).

```
# Definir função para estimar a precisão do modelo treinado
def evaluate_regression_accuracy(model_function, w, w_0, input_data, output_data):
    total_loss = 0
    n = len(input_data)
    for x, y in zip(input_data, output_data):
        y_predicted = model_function(w, w_0, x)
        error = y_predicted - y
        total_loss += (error**2)/2
    accuracy = total_loss / n
    print("Our model has mean square error of", accuracy)
    return accuracy
```

2.b. Exemplo de regressão com neurônio único

O exemplo de neurônio único a seguir usa funções criadas anteriormente para treinar o modelo de regressão com um conjunto simples de dados de entrada.

```
# Temos um conjunto simples de dados de entrada em 1D: uma lista de pontos "x".
# Cada ponto x é uma lista de extensão 1 com uma resposta "y" correspondente.
X_1D = [[1], [-2], [3], [4,5], [0], [-4], [-1], [4], [-1]]
Y_1D = [4,   3,    6,   8,     2,   -3,   -2,   7,   2,5]

# Neste exemplo, definimos os pesos iniciais como zero.
# A taxa de aprendizagem é relativamente baixa.
w_0 = 0
w = [0]
learning_rate = 0.01
epochs = 11

# Chamar a função train_model_SE_loss para os dados de treinamento
w, w_0 = train_model_SE_loss(single_neuron_regression_model, w, w_0,
                             X_1D, Y_1D,
                             learning_rate, epochs)

# Imprimir resultados com pesos treinados
print("\nFinal weights:")
print(w, w_0)

# Avaliar a precisão do modelo final
evaluate_regression_accuracy(single_neuron_regression_model, w, w_0, X_1D, Y_1D)
```

O resultado da execução do treinamento no exemplo de regressão com neurônio único é mostrado abaixo:

```
# Resultado de print:
epoch 0 has total loss 75.15118164194563
epoch 1 has total loss 40.109157956509094
epoch 2 has total loss 29.801523004849642
epoch 3 has total loss 25.47970121810249
epoch 4 has total loss 22.84563043777799
epoch 5 has total loss 20.86922066083035
epoch 6 has total loss 19.269845776759738
epoch 7 has total loss 17.945486824052068
epoch 8 has total loss 16.841512143299816
epoch 9 has total loss 15.91945696552539
epoch 10 has total loss 15.148892782711584

Final weights:
[1.2616634248232683] 1.589875809042491
```

3. Modelo de classificação de neurônio único em Python

Em um modelo de classificação, a função de ativação deve ser mudada para uma função sigmoide para limitar a saída de ativação a valores entre 0 e 1.

O *script* a seguir aplica mudanças à função anterior `single_neuron_regression_model` do modelo de regressão para definir uma nova função `single_neuron_classification_model` que use uma função de ativação `sigmoide`.

```
# Definir a função sigmoide para um modelo de classificação
def single_neuron_classification_model(w, w_0, x):
    # Executar o produto escalar na entrada x e nos pesos w
    z = 0
    for feature, weight in zip(x, w):
        z += feature * weight
    z += w_0  # Adicionar o termo de viés

    # Aplicar a função de ativação e devolver
    a = sigmoid(z) # Nova função de ativação
    return a

# Função de ativação sigmoide; reduz um valor real z a valores entre 0 e 1
def sigmoid(z):
    non_zero_tolerance = 1e-8 # Adicionar às divisões para que não se divida por 0
    return 1 / (1 + math.exp(-z) + non_zero_tolerance)
```

3.a. Treinamento de um modelo de classificação – descida do gradiente

Modelos de classificação usam uma função de perda diferente para treinar pesos com otimização por descida do gradiente. A função de perda em problemas de classificação é a função de perda por logaritmo negativo da verossimilhança (*negative log-likelihood*, NLL).

A fórmula a seguir é a notação matemática formal para a função de perda L_{NLL}.

$$L_{NLL} = y\ log(\hat{y}) + (1 - y)\ log(1 - \hat{y})$$

O *script* a seguir aplica mudanças à função anterior `train_model_SE_loss` do modelo de regressão para definir uma nova função `train_model_NLL_loss` para o modelo de classificação.

```
# Processo de treinamento usando a perda por logaritmo
# negativo da verossimilhança (NLL) para classificação
def train_model_NLL_loss(model_function, w, w_0,
                         input_data, output_data,
                         learning_rate, num_epochs):
    non_zero_tolerance = 1e-8 # Adicionar para garantir que não teremos log de 0
    for epoch in range(num_epochs):
        total_loss = 0 # Acompanhar a perda total ao longo do conjunto de dados

        for x, y in zip(input_data, output_data):
            y_predicted = model_function(w, w_0, x)
            nll_loss = -(y * math.log(y_predicted + non_zero_tolerance) +
                         (1-y) * math.log(1-y_predicted + non_zero_tolerance))
            total_loss += nll_loss

         # Atualizar o coeficiente de viés usando gradiente em relação a w_0
            w_0 -= learning_rate * (y_predicted - y)

         # Atualizar outros coeficientes
         # usando gradiente em relação a cada coeficiente
            for j, x_j in enumerate(x):
                w[j] -= learning_rate * (y_predicted - y) * x_j

        report_every = max(1, num_epochs // 10)
        if epoch % report_every == 0: # Informar progresso
            print("epoch", epoch, "has total loss", total_loss)
```

Além do treinamento de pesos, a função a seguir avalia o desempenho do modelo de classificação treinado, calculando sua precisão (`accuracy`), mas não de forma customizada para a tarefa de classificação.

```
# Usaremos esta função para avaliar o desempenho de nosso classificador treinado
def evaluate_classification_accuracy(model_function, w, w_0, input_data, labels):

    # Contar número de amostras bem classificadas, dado um conjunto de pesos
    correct = 0
    n = len(input_data)
    for x, y in zip(input_data, labels):
        y_predicted = model_function(w, w_0, x)
        label_predicted = 1 if y_predicted > 0.5 else 0
        if label_predicted == y:
            correct += 1
        else:
            print("Misclassify", x, y, "with activation", y_predicted)
    accuracy = correct / n
    print("Our model predicted", correct, "out of", n,
          "correctly for", accuracy*100, "% accuracy")
    return accuracy
```

3.b. Exemplo de classificação com neurônio único

O exemplo de neurônio único a seguir usa funções criadas anteriormente para treinar o modelo de classificação com um novo conjunto de dados de entrada.

```
# Temos aqui um conjunto de pontos de dados separáveis linearmente em 2D
input_data = [[1, 1], [1, 5], [-2, 3], [3, -4], [4.5, 2], [0, 1], [-4, -4],
              [-1, 2], [4, -7], [-1, 8]]
# e seus rótulos correspondentes
labels = [1, 1, 0, 1, 1, 0, 0, 0, 1, 0]

# Inicializar pesos em valores baixos que não sejam zero
# e treinar por mais épocas
w_0 = 0.1
w = [0.1, 0.1]
learning_rate = 0.01
epochs = 101

w, w_0 = train_model_NLL_loss(single_neuron_classification_model, w, w_0,
                              input_data, labels,
                              learning_rate, epochs)
print("\nFinal weights:")
print(w, w_0)

# Avaliar a precisão do modelo final
evaluate_classification_accuracy(single_neuron_classification_model, w, w_0,
                                 input_data, labels)
```

O resultado da execução do treinamento no exemplo de classificação com neurônio único é mostrado abaixo:

```
# Resultados de print:
epoch 0 has total loss 6.223923692215003
epoch 10 has total loss 2.9287154378367712
epoch 20 has total loss 2.3584449039881488
epoch 30 has total loss 2.0636076120006903
epoch 40 has total loss 1.8696299804962355
epoch 50 has total loss 1.7267167165811537
epoch 60 has total loss 1.6144108218404705
epoch 70 has total loss 1.5224295144979199
epoch 80 has total loss 1.4448950765018669
epoch 90 has total loss 1.3781364940480267
epoch 100 has total loss 1.319708591557062

Final weights:
[1.7028723012501872, -0.09064040426016721] -0.1352352402813103
Our model predicted 10 out of 10 correctly for 100.0 % accuracy
```