Semana 03 – Exercício em Aula

PRO6006 - Métodos de Otimização Não Linear com aplicações em aprendizado de máquina

Nathan Sampaio Santos – 8988661

IMPLEMENTAÇÃO REGRESSÃO LOGÍSTICA

O método principal desta classe é chamado *fit*. Nele, os parâmetros w e b da função *f(x)* são calculados até ser atingido a convergência.

$$f(x) = \frac{1}{1+e^{-(wx+b)}}$$

Primeiramente, é feita a previsão de probabilidade binária pela função sigmoide f(x). Após isso, é definida uma variável custo, de acordo com a seguinte fórmula:

$$\mathcal{L}(y_i, f(x_i)) = -\left[y_i \log f(x_i) + (1-y_i)\log(1-f(x_i))\right]$$

Também são calculados os fatores do vetor gradiente desta função de custo, de acordo com as seguintes fórmulas:

$$\nabla_w \mathcal{L} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)\cdot x^{(i)} \qquad \frac{\partial \mathcal{L}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)$$

De posse deste vetor e do passo de aprendizado definido como um parâmetro, a função calcula novos valores de *w* e *b* até serem atingidos os critérios de convergência, que são: o número de interações ou a norma do vetor gradiente ser menor do que o parâmetro de tolerância.

```
    def fit(self, X, y):
        n_samples, n_features = X.shape

        grad_norm = np.inf
        self.w    = np.zeros(n_features)
        self.b    = 0
        self.cost_history = []

        while self.n_iterations <= self.n_iterations_max and \
              grad_norm > self.tol:

            self.n_iterations += 1
            y_predicted = self.predict_proba(X)

            epsilon = 1e-9
            cost = (-1 / n_samples) * np.sum(y * np.log(y_predicted +
epsilon) + (1 - y) * np.log(1 - y_predicted + epsilon))
```

```
            self.cost_history.append(cost)

            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            grad_norm = np.linalg.norm(np.concatenate((dw, 
np.array([db]))))

            self.w -= self.learning_rate * dw
            self.b -= self.learning_rate * db
```

USO REGRESSÃO LOGÍSTICA

Para testar o uso dessa função, utilizamos uma base de diagnóstico de câncer, a qual possui 30 parâmetros listados a seguir que caracterizam quantitativamente um tumor. Além desses parâmetros, também é obtido a coluna *target*, binária, que indica a confirmação ou não de um câncer.

- *mean radius*
- *mean texture*
- *mean perimeter*
- *mean area*
- *mean smoothness*
- *mean compactness*
- *mean concavity*
- *mean concave points*
- *mean symmetry*
- *mean fractal dimension*
- *radius error*
- *texture error*
- *perimeter error*
- *area error*
- *smoothness error*
- *compactness error*
- *concavity error*
- *concave points error*
- *symmetry error*
- *fractal dimension error*
- *worst radius*
- *worst texture*
- *worst perimeter*
- *worst area*
- *worst smoothness*
- *worst compactness*
- *worst concavity*
- *worst concave points*
- *worst symmetry*
- *worst fractal dimension*

Todos esses dados podem ser encontrados publicamente na base de dados UCI ou dentro da biblioteca sklearn do Python.

```
from LogisticRegression       import LogisticRegression
from sklearn.datasets         import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing    import StandardScaler
from sklearn.metrics          import accuracy_score
import matplotlib.pyplot as plt

cancer_data = load_breast_cancer()
X = cancer_data.data
```

```python
y = cancer_data.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=7)

scaler         = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled  = scaler.transform(X_test)

model = LogisticRegression(learning_rate=0.01, n_iterations_max=1000,
tol=0.001)
model.fit(X_train_scaled, y_train)
y_predictions = model.predict(X_test_scaled)


accuracy = accuracy_score(y_test, y_predictions)
```

Após obter a base de dados, separamos ela em dois conjuntos: teste e treinamento, com a proporção 20/80, respectivamente. Isso é feito para que a análise da acurácia não seja contaminada por teste em dados que foram utilizados no treinamento do modelo.

Além disso, normalizamos os dados, isto é, subtraímos a média e dividimos pelo desvio padrão, para que todos os coeficientes dos parâmetros tenham pesos equivalentes na busca de uma função que minimize a função custo previamente determinada. O impacto deste passo será visto na seção a seguir.

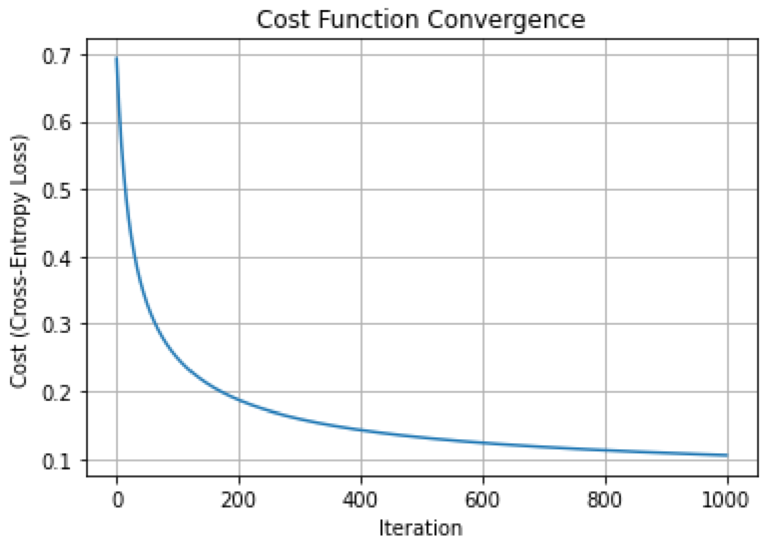
ANÁLISE DE RESULTADOS E CONVERGÊNCIA

Com a execução do código descrito acima, a função convergiu para os parâmetros *w* e *b* listados a seguir, com uma acurácia de 97.37%. Estes valores de acurácia podem variar conforme a divisão aleatória dos conjuntos de teste e treinos feita, porém o alto valor de acurácia neste caso já dá ótimos indicativos de que a função está bem calibrada.

```
------------------------------
Model parameters:
------------------------------
Weights (w): [-0.36552016 -0.31708783 -0.3605467  -0.36231673 -0.14111157 -0.11289294
 -0.28285338 -0.38545157 -0.1136004   0.1636088  -0.32539795  0.0288431
 -0.28193005 -0.29146192  0.04206538  0.10552436  0.09374887 -0.04060476
  0.0556362   0.18270845 -0.4508086  -0.40210415 -0.43271398 -0.42396803
 -0.32790248 -0.20234755 -0.29469698 -0.42512958 -0.31622898 -0.11213335]
Bias (b): 0.312670212257911

------------------------------
Model Accuracy: 97.37%
------------------------------
```

Podemos ainda analisar a curva da função custo ao longo das consecutivas interações. Vemos que, para uma tolerância de 0.001 do módulo do gradiente, o processo foi interrompido no segundo critério de parada que era 1000 interações. Mesmo assim, o valor final da função de custo é baixo, indicando que a solução está próxima do seu ótimo global, convergindo de modo exponencial e contínuo.



Por fim, vemos o efeito que a normalização dos dados trouxe para curva de custo. Sem normalizarmos os parâmetros, a função de custo torna-se instável sem convergir para um valor baixo desejável.