Universidade da Beira Interior

Departamento de Informática

Nº Trabalho Prático 2: *Regressão Logistica*

Elaborado por:

João Miguel Baltazar Martins

Orientador:

Professor Doutor Hugo Pedro Proença

27 de outubro de 2023

UBI Universidade da Beira Interior

Conteúdo

Capítulo 1

Introdução

1.1 Enquadramento

Este trabalho aborda o desenvolvimento de um modelo de regressão logística para a classificação de dígitos de 0 a 9 no conjunto de dados MNIST em formato CSV disponível no Kaggle. O script `logistic_regression.py` incorpora diferentes estratégias de normalização de recursos, como min-max e Z-score, explora diversos valores de regularização, critérios de paragem e taxas de aprendizagem para otimização do modelo. Essas considerações visam melhorar o desempenho do modelo na tarefa de classificação. Este trabalho foi realizado no contexto do primeiro ano do mestrado em Engenharia Informática da Universidade da Beira Interior (UBI) na unidade curricular de *Machine Learning*.

1.2 Organização do Documento

1. O primeiro capítulo – **Introdução** – apresenta o projeto, o enquadramento para o mesmo e a respetiva organização do documento.

2. O segundo capítulo – **Desenvolvimento** – é apresentado o *script* desenvolvido. São apresentadas as estratégias de normalização utilizadas, é apresentado o valor de regularização (conceito, consequências e escolha), apresentado o método elaborado para o critério de paragem e a taxa de aprendizagem utilizada. Por fim, é feita a exposição e análise ao *output* final, abordando assim a matriz confusão e a precisão do teste.

3. O terceiro capítulo – **Conclusão** – refere as conclusões principais a tirar deste trabalho, bem como uma reflexão sobre um trabalho futuro.

Capítulo 2

Desenvolvimento

2.1 Introdução

Neste capítulo, irá ser apresentado o *script* que contém um modelo capaz de distinguir entre as classes "0"..."9" e as devidas conclusões provenientes das alterações propostas ao mesmo no enunciado.

1. Exibição do *Script* `logistic_regression.py`

    a) Diferentes Estratégias de Normalização das *Features*.

    a) Valores de Regularização do Modelo.

    a) Critério de Paragem Desenvolvido e Taxa de Aprendizagem.

    a) Exposição e Análise do *Output* Final do `Output` (Matriz Confusão e Precisão do Teste)

2.2 *Script* `logistic_regression.py`

Função de Leitura dos Dados

Parâmetros:

- `file_name` (str): O nome do ficheiro CSV onde estão os dados, tanto o de treino como o de teste.

Retornos:

- `X` (`numpy.ndarray`): Um *array* contendo os dados de entrada (valores dos pixeis) após o pré-processamento.

- y (`numpy.ndarray`): Um *array* contendo rótulos no formato *one-hot encoding* após o pré-processamento.

Esta função lê dados de imagens de um ficheiro CSV, normaliza os valores dos pixeis e prepara-os para tarefas de aprendizagem automática.

O ficheiro CSV deve ter o seguinte formato:

- A primeira coluna deve conter rótulos (0, 1, 2, ...) indicando a classe de cada imagem.
- As colunas subsequentes devem conter os valores dos pixeis para cada imagem.

A função realiza os seguintes passos:

1. Lê os dados do ficheiro CSV, ignorando a linha de cabeçalho.
2. Normaliza os valores dos pixeis, dividindo-os por 255.0 para os colocar no intervalo [0, 1].
3. Padroniza (normalização Z-score) os valores dos pixeis.
4. Adiciona um termo de polarização (1) aos dados de entrada, o que é um passo comum na aprendizagem automática.
5. Converte os rótulos (y) para one-hot encoding, em que cada classe é representada como um vetor binário.

```
#
    ##############################################################################

#   Read, Normalize and Split Data
#
    ##############################################################################


def load_and_process_data(file_name):
    # Read data from CSV file and preprocess it
    with open(file_name) as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        next(csv_reader)  # Skip the header row
        X = []  # Initialize a list to store input data (pixel values)
        y = []  # Initialize a list to store labels
        for row in csv_reader:
            y.append(int(row[0]))
            temp = [float(i) / 255.0 for i in row[1:]]  # Normalize
                pixel values
```

```
            X.append(temp)

    # Convert data to NumPy arrays for further processing
    X = np.asarray(X)
    y = np.asarray(y)

    # Normalize the pixel values using Min-Max scaling
    '''
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
    '''


    # Normalize the pixel values using Z-score (standardization)
    scaler = StandardScaler()
    X = scaler.fit_transform(X)


    # Add a bias term (1) to the input data
    X = np.append(X, np.ones((X.shape[0], 1), np.float64), axis=1)

    # Convert labels (y) to one-hot encoding
    num_classes = len(np.unique(y))
    y = np.eye(num_classes)[y]

    return X, y
```

Excerto de Código 2.1: Função de leitura dos dados do *Script* logistic_regression.py

Função Custo

Parâmetros:

- y_true (numpy.ndarray): Um array contendo os valores reais (verdadeiros) dos rótulos.
- y_pred (numpy.ndarray): Um array contendo os valores previstos (preditos) dos rótulos.
- weights (numpy.ndarray): Um array de pesos para regularização L2.
- lambda_reg (float): Parâmetro de regularização L2.

Retorno:

- Um valor numérico representando a perda de entropia cruzada binária com regularização L2.

Esta função calcula a perda de entropia cruzada binária entre os valores reais (`y_true`) e os valores previstos (`y_pred`), incluindo uma regularização L2. A entropia cruzada binária é uma métrica comum para avaliar o desempenho de modelos de classificação binária, e a regularização L2 (também chamada regularização Ridge) que ajuda a evitar o **overfitting** do modelo.

A penalização é aplicada aos coeficientes do modelo com base na norma L2, o que tende a manter os coeficientes relativamente pequenos, desencorajando o modelo a se ajustar em excesso aos dados de treino.

A função utiliza a seguinte fórmula para calcular a perda com regularização L2:

$$L(y_{\text{true}}, y_{\text{pred}}, \text{weights}, \lambda) = -\frac{1}{N}\sum_{i=1}^{N}\left(y_{\text{true}}^{(i)} \cdot \log(y_{\text{pred}}^{(i)} + \epsilon) + (1 - y_{\text{true}}^{(i)}) \cdot \log(1 - y_{\text{pred}}^{(i)} + \epsilon)\right) + \frac{\lambda}{2N}\sum_{j=1}^{M} w_j^2$$

onde: - N é o número de exemplos.
- $y_{\text{true}}^{(i)}$ é o valor verdadeiro do rótulo do exemplo i.
- $y_{\text{pred}}^{(i)}$ é o valor previsto do rótulo do exemplo i.
- ϵ é um valor pequeno (geralmente 1×10^{-9}) para evitar cálculos instáveis.
- λ é o parâmetro de regularização L2.
- M é o número de pesos no vetor `weights`.
- w_j é o j-ésimo peso do vetor `weights`.

Nota:

- A entropia cruzada binária é frequentemente usada como função de perda em tarefas de classificação binária.
- A regularização L2 ajuda a evitar o **overfitting** ao penalizar pesos maiores no modelo.
- ϵ é um valor pequeno adicionado aos cálculos para evitar problemas numéricos quando y_{pred} é igual a 0 ou 1.

```
def compute_loss(y_true, y_pred):
    # Compute the binary cross-entropy loss
    epsilon = 1e-9
    y1 = y_true * np.log(y_pred + epsilon)
    y2 = (1 - y_true) * np.log(1 - y_pred + epsilon)
    return -np.mean(y1 + y2)
```

Excerto de Código 2.2: Função Custo

Função Sigmoide

Parâmetros:

- `x` (`numpy.ndarray` ou `float`): Um valor ou array numérico para o qual se aplica a função sigmoid.

Retorno:

- Um valor numérico no intervalo [0, 1] após a aplicação da função sigmoid.

A função `sigmoid` implementa a função de ativação sigmoid, que é comumente usada em redes neurais e outros modelos de aprendizagem de máquina. A função sigmoid transforma a entrada x em um valor no intervalo [0, 1] de acordo com a fórmula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

onde: - x é a entrada. - e é o número de Euler, uma constante matemática.

A função sigmoid é usada para introduzir não-linearidade em modelos e é frequentemente utilizada para problemas de classificação binária.

Nota:

- A função sigmoid é útil para converter valores de entrada em probabilidades no intervalo [0, 1].

```
def sigmoid(x):
    # Sigmoid activation function
    return 1 / (1 + np.exp(-x))
```

Excerto de Código 2.3: Função Sigmoide

Função *Feed Foward*

Parâmetros:

- `X` (`numpy.ndarray`): Um array contendo os dados de entrada.
- `weights` (`numpy.ndarray`): Um array de pesos para a operação de feedforward.
- `bias` (`float` ou `numpy.ndarray`): Um valor de polarização (bias) ou array de polarizações para a operação de feedforward.

Retorno:

- Um array contendo os valores de ativação após a operação de feedforward.

A função `feed_forward` realiza a operação de feedforward numa rede neural ou modelo de aprendizagem automática. Ela calcula os valores de ativação (saída) da camada atual com base nos dados de entrada (`X`), nos pesos (`weights`) e no valor de polarização (`bias`).

A operação é realizada da seguinte forma: 1. Calcula a combinação linear dos dados de entrada (`X`) e pesos (`weights`) adicionando o valor de polarização (`bias`):

$$z = X \cdot weights + bias$$

onde: - z é a combinação linear. - X é o array de dados de entrada. - `weights` é o array de pesos. - `bias` é o valor de polarização.

2. Aplica a função sigmoid (`sigmoid`) aos valores da combinação linear para obter os valores de ativação (A):

$$A = \sigma(z)$$

onde: - A é o array de valores de ativação. - $\sigma(z)$ é a função sigmoid.

```
def feed_forward(X, weights, bias):
    # Perform feedforward operation
    z = np.dot(X, weights) + bias
    A = sigmoid(z)
    return A
```

Excerto de Código 2.4: *Feed Foward*

Função *Fit*

Parâmetros:

- `X` (`numpy.ndarray`): Um array contendo os dados de entrada.
- `weights` (`numpy.ndarray`): Um array de pesos para a operação de feedforward.
- `bias` (`float` ou `numpy.ndarray`): Um valor de polarização (bias) ou array de polarizações para a operação de feedforward.

Retorno:

- Um array contendo os valores de ativação após a operação de feedforward.

A função `feed_forward` realiza a operação de feedforward em uma rede neural ou modelo de aprendizagem automática. Ela calcula os valores de ativação (saída) da camada atual com base nos dados de entrada (`X`), nos pesos (`weights`) e no valor de polarização (`bias`).

A operação é realizada da seguinte forma: 1. Calcula a combinação linear dos dados de entrada (`X`) e pesos (`weights`) adicionando o valor de polarização (`bias`):

$$z = X \cdot weights + bias$$

onde: - z é a combinação linear. - X é o array de dados de entrada. - `weights` é o array de pesos. - `bias` é o valor de polarização.

2. Aplica a função sigmoid (`sigmoid`) aos valores da combinação linear para obter os valores de ativação (A):

$$A = \sigma(z)$$

onde:
- A é o array de valores de ativação.
- $\sigma(z)$ é a função sigmoid.

```
def fit(X, y, lr, lambda_reg, stopping_threshold):
    n_samples, n_features = X.shape
    n_classes = y.shape[1]

    weights = np.zeros((n_features, n_classes))
    bias = np.zeros(n_classes)
    losses = []  # To store loss at each iteration

    previous_loss = float('inf')  # Set to a large value initially
    iteration = 0

    while True:
        iteration += 1
        A = feed_forward(X, weights, bias)
        loss = compute_loss(y, A, weights, lambda_reg)
        losses.append(loss)

        if abs(loss - previous_loss) < stopping_threshold:
            print("Stopping training as loss change is smaller than the
                stopping threshold.")
            break

        dz = A - y
        dw = (1 / n_samples) * (np.dot(X.T, dz) + lambda_reg * weights)
        db = (1 / n_samples) * np.sum(dz, axis=0)
        weights -= lr * dw
        bias -= lr * db
```

```
        previous_loss = loss

        if iteration % 10 == 0:
            print(f"Iteration {iteration} - Loss: {loss}")

    return weights, bias, losses, iteration
```

Excerto de Código 2.5: Função *Fit*

2.3 Estratégias de Normalização das Features

Min-Max

```
# Normalize the pixel values using Min-Max scaling
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
```

Excerto de Código 2.6: Min-Max

Z-Zero

```
# Normalize the pixel values using Min-Max scaling
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
```

Excerto de Código 2.7: Z-Zero

Min-Max vs Z-Score Aplicado ao Problema

1. Min-Max Scaling:

    a) O conjunto de dados MNIST é composto por valores de pixel que variam de 0 (preto) a 255 (branco). Se é desejado que todos os valores de pixel estejam dentro de um intervalo específico, como [0, 1], para serem compatíveis com algoritmos que esperam dados nesse intervalo, deve-se usar Min-Max Scaling.

2. Z-Score (Padronização):

a) A padronização (Z-Score) é útil quando se assume que os dados têm uma distribuição normal ou quando se deseja remover a escala dos dados para torná-los comparáveis em termos de unidades de desvio padrão.

b) Ajuda a reduzir o impacto de `outliers` nos dados.

```
# Normalize the pixel values using Min–Max scaling
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
```

Excerto de Código 2.8: Resultado do Min-Max ao ser aplicado ao nosso problema

```
# Normalize the pixel values using Min–Max scaling
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
```

Excerto de Código 2.9: Resultado do Z-Score ao ser aplicado ao nosso problema

2.4 Diferentes Valores de Regularização do Modelo

```
lambda_reg = 0.01
```

Excerto de Código 2.10: Linha do código que permite a variação da variável de regularização

1. Impacto no Modelo:

a) Se o valor de `lambda_reg` for muito baixo (próximo de zero), o modelo terá menos regularização e pode estar mais propenso ao overfitting.

b) Se o valor de `lambda_reg` for muito alto, o modelo terá mais regularização e será menos flexível, mas geralmente será melhor em termos de generalização.

2.5 Critério de Paragem e Taxa de Aprendizagem

Critério de Paragem

O critério de paragem neste código é baseado na variação da função de perda ao longo das iterações. A ideia é interromper o treino quando a mudança na perda entre iterações consecutivas for menor do que um valor predefinido chamado `stopping_threshold`. O `stopping_threshold` é definido como 1×10^{-6}, o que significa 0.000001.

O critério de paragem foi implementado da seguinte forma:

1. No início do treino, a variável `previous_loss` é inicializada com um valor infinito positivo, representado por `float('inf')`. Isso é feito para garantir que na primeira iteração, a diferença entre a perda atual e a perda anterior seja maior do que o `stopping_threshold`.

2. Após cada iteração, a função de perda é calculada e armazenada na lista `losses`. A diferença entre a perda atual e a perda anterior é calculada usando `abs(loss  previous_loss)`.

3. Se a diferença entre a perda atual e a perda anterior for menor do que o `stopping_threshold`, isso indica que a perda está a convergir para um valor mínimo ou que a variação na perda é muito pequena. Nesse caso, o treino é interrompido.

Em resumo, o treino será interrompido quando a mudança na perda entre iterações consecutivas for menor do que o valor predefinido `stopping_threshold`, indicando que a otimização convergiu ou que a perda está a mudar muito lentamente. Isso ajuda a economizar tempo de computação, uma vez que continuar o treino nesse ponto provavelmente não levará às melhorias significativas nos parâmetros do modelo.

Taxa de Aprendizagem

```
# Normalize the pixel values using Min-Max scaling
    scaler = MinMaxScaler()
    scaler.fit(X)
    X = scaler.transform(X)
```

Excerto de Código 2.11: Learing rate escolhido

Sabendo que:

1. Taxa de Aprendizagem Muito Alta:

a) Taxas de aprendizagem muito altas podem causar oscilações no processo de otimização e até mesmo fazer com que o treino divirja. Isso significa que os parâmetros do modelo podem não convergir para um mínimo da função de perda, e o treino pode se tornar instável.

2. Taxa de Aprendizagem Muito Baixa:

a) Taxas de aprendizagem muito baixas podem levar a um treino extremamente lento. O modelo pode levar um tempo excessivo para convergir, ou talvez nunca atinja uma solução aceitável, especialmente se o conjunto de dados ou a arquitetura do modelo forem complexos.

Exposição e Análise do *Output* Final do *Script* (Matriz Confusão e Precisão do Teste)

1. **Hiperparâmetros**:

   - Taxa de Aprendizagem = 0,001. Este é o tamanho do passo que o modelo usa para atualizar os pesos durante o treino.
   - Variável de Regularização = 0,1: A regularização é uma técnica para evitar o *overfitting*. Um termo de regularização é adicionado à função de custo para controlar a complexidade do modelo.

2. **Após 25150 iterações**:

   - Função de Custo: 0,09057835366948284: A função de custo representa o quão bem o modelo se ajusta aos dados de treino. Um valor menor indica um melhor ajuste.
   - Precisão no Teste: 0,896: O modelo obteve uma precisão de aproximadamente 89,6% no conjunto de dados de teste. Isso significa que ele classificou corretamente 89,6% dos exemplos de teste.

3. **Matriz de Confusão**: Uma matriz de confusão é uma tabela frequentemente usada para descrever o desempenho de um modelo de classificação. Neste caso, a matriz de confusão mostra quantas instâncias de cada classe foram classificadas corretamente (diagonal) e quantas foram classificadas de forma incorreta.

	0	1	2	3	4	5	6	7	8	9
0	958	0	1	1	0	8	8	1	2	1
1	0	1104	3	3	1	1	5	0	18	0
2	13	21	887	20	11	0	17	21	37	5
3	5	7	25	894	3	29	5	17	17	8
4	0	10	7	1	911	2	7	5	5	34
5	16	8	2	36	19	741	21	13	25	11
6	11	8	9	0	17	18	895	0	0	0
7	4	26	19	7	11	1	1	923	0	36
8	15	28	9	22	21	31	14	19	800	15
9	16	9	1	16	57	9	0	43	6	852

- A diagonal principal da matriz representa as classificações corretas. Por exemplo, na posição (0,0), temos 958, o que significa que 958 imagens da classe 0 foram corretamente classificadas como classe 0. Isso é consistente com as outras classes também.
- Os valores fora da diagonal principal representam as classificações incorretas. Por exemplo, na posição (0,5), temos o valor 8, o que significa que 8 imagens da classe 0 foram incorretamente classificadas como classe 5.
- Cada linha da matriz representa a classe real, enquanto cada coluna representa a classe prevista pelo modelo.

Conclusões a tirar desta matriz:

a) O modelo está acertando a maioria das classificações, pois a diagonal principal contém valores significativamente maiores do que os valores fora dela.

b) Algumas confusões são mais comuns do que outras. Por exemplo, a classe 5 parece ser confundida com outras classes com mais frequência, como indicado pelos valores fora da diagonal principal.

c) A classe 7 tem alguns erros notáveis, como indicado pela última coluna da linha 7, onde vários dígitos 7 foram classificados como outras classes, incluindo a classe 4 e 9.

Capítulo 3

Conclusão

3.1 Conclusões Principais

Neste trabalho, foi desenvolvido um modelo de regressão logística para a tarefa de classificação das classes de dígitos de 0 a 9 no conjunto de dados MNIST, que consiste em 60.000 exemplos de treino e 10.000 exemplos de teste. Após treinar o modelo com 25.150 iterações e usando hiperparâmetros específicos, obtivémos os seguintes resultados:

- A função de custo alcançou o valor de 0,09057835366948284, indicando um bom ajuste do modelo aos dados de treino.
- A precisão no conjunto de teste foi de aproximadamente 89,6%, o que mostra que o modelo é capaz de classificar corretamente a maioria das imagens de dígitos.
- A matriz de confusão revelou o desempenho do modelo para cada classe, destacando que algumas classes são mais frequentemente confundidas com outras.

As principais conclusões deste trabalho são as seguintes:

- O modelo de regressão logística é eficaz na tarefa de classificação de dígitos escritos à mão.
- A taxa de aprendizagem e o termo de regularização escolhidos parecem ser apropriados para este problema, resultando num modelo bem ajustado.
- A precisão de 89,6% é um resultado promissor, mas ainda há espaço para melhorias, especialmente na identificação de dígitos frequentemente confundidos com outros.

3.2 Limitações e Próximos Passos

No entanto, este trabalho também apresentou algumas limitações e deixa espaço para futuras melhorias:

- Embora o modelo tenha alcançado uma boa precisão, ele ainda comete erros notáveis, principalmente na classificação de dígitos que se assemelham a outros. Melhorar a precisão dessas classes específicas é um desafio futuro.
- Este modelo é relativamente simples. Para obter um desempenho ainda melhor, pode-se considerar a utilização de abordagens mais avançadas, como redes neurais convulsionais (CNNs) que são especialmente projetadas para tarefas de visão computacional.
- Além disso, o aumento do tamanho do conjunto de treino e a realização de validação cruzada para otimização de hiperparâmetros podem contribuir para melhorias adicionais na precisão do modelo.

Em resumo, este trabalho demonstrou a eficácia de um modelo de regressão logística na classificação de dígitos do MNIST, mas também apontou desafios e oportunidades para futuras pesquisas na melhoria da precisão e da robustez do modelo.