Universidade do Minho
Escola de Ciências

Mestrado em Matemática e Computação

Relatório do Trabalho Prático T3.1: Aplicação do PCA em reconhecimento de dígitos

Métricas em Machine Learning

Ano Letivo de 2023/2024

Grupo 4

Gabriella Lima, pg54401
Guilherme Martins, pg52214
Maria Laires, pg52220
Matheus Ribeiro, pg52254

Braga, Gualtar, Portugal - Janeiro de 2024

Conteúdo

1 Introdução

O surgimento dos números remonta aos primórdios da civilização, quando as necessidades de contagem e medida impulsionaram a criação de sistemas numéricos. Os primeiros vestígios do uso de números remontam às civilizações antigas, como os sumérios, egípcios e babilônios, que desenvolveram sistemas de numeração baseados em símbolos e representações gráficas para quantificar e registrar informações. A numeração indo-arábica é a que utilizamos amplamente hoje em dia (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Esse sistema posicional, onde o valor de um número depende da posição que ocupa, revolucionou o modo como lidamos com quantidades, cálculos e operações matemáticas.

Neste trabalho, desenvolvemos um sistema em *Python* que reconhece números manuscritos de um determinado dataset, usando o método *Principal Component Analysis* (PCA). O ficheiro com o código *Python* tem a denominação de '**PCA_MNIST.ipynb**'. O PCA é considerado um dos resultados mais valiosos da álgebra linear aplicada e é usado abundantemente em todas as formas de análise, da neurociência à computação gráfica - porque é um método simples e não paramétrico de extrair informações relevantes de datasets complexos, reduzindo sua dimensão e tornando sua interpretação facilitada.

2 Dataset

O dataset usado foi o *MNIST*, que acedemos através do "WayBack Machine"(registo Março de 2022) através do link, no dia 20 de dezembro de 2023. O dataset consiste em dígitos manuscritos que foram normalizados em tamanho e centrados em uma imagem de tamanho fixo. Neste dataset, dentro da pasta '*mnist_database*', temos um *training set* que possui 60.000 imagens (de tamanho 28x28) para treinar os modelos, e um *test set* que possui 10.000 exemplos para avaliar os resultados pelos modelos anteriores.

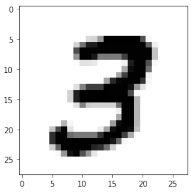


Figura 2.1: Exemplo imagem do dataset.

Figura 2.2: Escala de cores das imagens após conversão.

Para além das imagens, o dataset possui ficheiros que mostram o rótulo / label de cada imagem (quer de treino, quer de teste), que nos permite saber qual é o dígito correto da imagem correspondente. No caso da figura 2.1, o rótulo/label é 3.

Inicialmente, as imagens não se encontravam numa escala de cores a preto-e-branco, antes numa escala de cores, o que poderia dificultar a análise destas mesmas. Assim, utilizou-se a biblioteca "**idx2numpy**"para converter as imagens no formato *ubyte* num formato onde os pixeis da imagem passam para uma escala preto-e-branco (entre 0 e 255), (tal como na representado figura 2.2).

3 Leitura do dataset de treino

Para começarmos a implementação do modelo criado, devemos começar com a leitura do dataset de treino. Para isso, devemos carregar as imagens (*train-images.idx3-ubyte*) e rótulos / labels (*train-labels.idx1-ubyte*) do conjunto de dados MNIST, e manipular os dados das imagens, convertendo-as do formato IDX (é o formato standart do dataset) para o formato de uma matriz e depois para arrays unidimensionais. Dessa forma, cada elemento do array fica associado a um rótulo específico.

Veremos na secção 4.1 que para facilitar a futura aplicação das imagens e dos respetivos rótulos para treino de modelos, criou-se um array **X_digito**, onde cada valor de index entre 0 e 9 revela o conjunto de imagens com uma label igual ao index em X_digito. Ou seja, se quiseremos aceder à primeira imagem com uma label igual a 5, deve-se aceder X_digito[5][0], como podemos ver abaixo:

```
# exemplo de imagem dígito 5
plt.imshow(np.reshape(X_digito[5][0], (28,28)), cmap=plt.cm.binary)
```

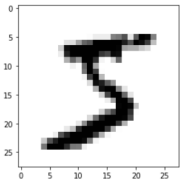


Figura 3.1: Exemplo de imagem associada ao label 5.

4 Construção de modelo para cada dígito

Agora, vamos organizar as imagens do conjunto MNIST de acordo com seus rótulos (dígitos de 0 a 9) e realizar algumas análises sobre a distribuição dos dígitos no conjunto de dados.

4.1 Distribuição dos rótulos no dataset de treino

Organização das imagens por dígito: Iteramos sobre cada imagem no conjunto X e atribui a cada array em ***X_digito*** a imagem correspondente ao seu rótulo. Por exemplo, se a imagem ***X[i]*** tem o rótulo ***labelarray[i]*** igual a 3, essa imagem será adicionada à lista ***X_digito[3]***.

Por exemplo, se quisermos exibir quantas imagens cada dígito possui, teremos o seguinte:

```
Distribuição das Labels: 0: 5923 ; 1: 6742 ; 2: 5958 ; 3: 6131 ; 4: 5842
 ; 5: 5421 ; 6: 5918 ; 7: 6265 ; 8: 5851 ; 9: 5949
```

Figura 4.1: Número de imagens por dígito.

E podemos representar isso em forma de gráfico de barras:

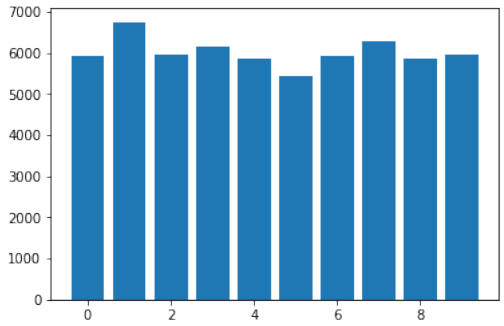


Figura 4.2: Distribuição das imagens de treino por cada dígito.

O gráfico de barras é uma maneira visual de entender a distribuição do número de imagens para cada dígito no conjunto de dados MNIST. Idealmente, em um conjunto de dados bem distribuído, cada dígito teria aproximadamente o mesmo número de representações (aproximadamente 6000 imagens de treino por cada dígito), resultando em barras de altura semelhante no gráfico.

4.2 Singular Value Decomposition ou SVD

Antes de prosseguirmos para a aplicação do PCA, vale relembrarmos o que é o SVD e como é feito. O algoritmo SVD é fundamental para redução de dimensionalidade, compressão de dados, e é usado em várias áreas, incluindo análise de dados, processamento de imagens e reconhecimento de padrões.

Este algoritmo consiste em decompor uma matriz em três outras matrizes significativas. Por exemplo, se tomarmos uma matriz A de dimensão $m \times n$, teremos:

1. Iremos decompor A da forma:

$$A = U.\Sigma.V^T$$

onde U é uma matriz ortogonal de tamanho $m \times m$, Σ é matriz diagonal de tamanho $m \times n$ (com zeros fora da diagonal principal), e V^T a transposta da matriz ortogonal V de tamanho $n \times n$ (devemos lembrar que uma matriz ortogonal é aquela que possui a transposta igual a sua matriz inversa).

2. Calculamos $A^T A$, seus valores e vetores próprios. Esses valores próprios são os quadrados dos ***valores singulares*** e os vetores próprios correspondentes são os vetores próprios associados aos ***valores singulares***.

3. Para construir U e V, temos que tomar em conta que:

   (a) Os vetores próprios normalizados de $A^T A$ formam as colunas das matrizes U e V.

   (b) U consiste nos vetores próprios de $A^T A$ (e tem dimensão $m \times m$).

   (c) V consiste nos valores próprios de $A^T\ A$ (e tem dimensão $n \times n$).

4. E para construir a matriz Σ, devemos lembrar do passo 2, que os valores singulares são as raízes quadradas dos valores próprios de $A^T A$. Dessa forma, usaremos estes valores singulares para construir Σ, onde estes serão a sua diagonal principal, dispostos em ordem decrescente. Note que Σ será uma matriz diagonal.

5. Realizamos a multiplicação das três matrizes e assim temos a matriz A decomposta.

Temos que o SDV **representa uma expansão dos dados originais em um sistema de coordenadas onde a matriz de covariância é diagonal**.

4.3 Aplicação do PCA e Cálculo das Componentes Principais de cada dígito

Na implementação do PCA sob a forma da função "**PCA(X, confianca_alvo)**", onde X é o conjunto de dados onde será aplicado o PCA e confianca_alvo um valor entre 0 e 1 que indica a percentagem de informação que desejamos que se conserve após a aplicação do PCA. Nesta função, encontram-se os vetores próprios e os valores próprios das componentes principais de um conjunto de dados X (usando o *Singular Value Decomposition*, ou SVD), juntamente com a *média* dos dados.

Dessa forma, o código da função consiste em:

1. Calcular a *média* de todas as imagens no conjunto X.

2. Centralização dos dados: Subtrai-se a *média* de cada imagem em X, calculada anteriormente, e centraliza os dados em torno desta média.

3. Cálculo dos Vetores e Valores Próprios usando SVD

   (a) Realiza-se a decomposição SVD da matriz de dados centralizada ***phi***. Como vimos na Secção 4.2, nossa intenção é obter as matrizes U e V e uma matriz com os valores singulares em sua diagonal, que é a matriz Σ.

   (b) Obtém-se os vetores próprios ***vet_prop*** e os valores singulares ***sigma***, que estão relacionados aos valores próprios através de ***val_prop = sigma * sigma***.

4. Ordenam-se os Valores Próprios por ordem decrescente, reorganizando-se os vetores próprios de acordo com essa mesma ordem.

5. Determinação do número de Componentes Principais.

   (a) Para isso, devemos calcular o número mínimo de vetores próprios a serem usados para atingir uma certa confiança ***confianca_alvo*** especificada.

   (b) Iteramos sobre os valores próprios, aumentando progressivamente k (o número de componentes) até alcançar a confiança desejada.

6. Selecionamos os k primeiros vetores próprios associados às componentes principais.

E por fim, teremos que a função **pca** retornará os resultados: o número de componentes principais k, os valores próprios ***val_prop***, os vetores próprios ***val_prop***, a matriz de dados centralizada ***phi***, a média dos dados (***media***), v sendo a variância, e a confiança alcançada (***confianca***).

Para o estudo efetuado neste trabalho, aplicou-se o PCA para cada conjunto de imagens que representam o mesmo dígito, todos com o mesmo valor de confiança-valor de 90%.

Assim, suponhamos que queremos reduzir a dimensionalidade de todas as imagens com dígito zero, com uma confiança-alvo de 0.9. Além disso, também queremos exibir a *imagem média* do dígito zero.

```
# Exemplo para o dígito 0
k_0, val_prop_0, vet_prop_0, phi_0, media_0, v_0, confianca_0 = pca(X_digito[0], 0.9)

plt.imshow(np.reshape(media_0, (28,28)), cmap=plt.cm.binary) # visualização da imagem média do dígito
```

Figura 4.3: Código usado para exibir resultados da função PCA e a imagem média do dígito zero.

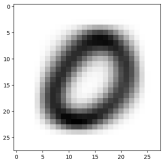


Figura 4.4: Output código 4.3.

Esta imagem é a média das imagens individuais do dígito 0 no conjunto de dados MNIST. Mas, também podemos gerar uma imagem a partir de uma das componentes principais do vetor próprio, que representará uma direção específica de variação nos dados associados ao dígito 0. Dessa forma, essa imagem mostrará um padrão ou direção de variação que é capturado pelo primeiro vetor próprio, indicando como uma das principais direções de variação se parece nos dados do dígito 0 após a aplicação do PCA.

```
# Exemplo resultante de um vetores próprios / uma das componentes principais da imagem
plt.imshow(np.reshape(vet_prop_0.T[0], (28,28)), cmap=plt.cm.binary)
```

Figura 4.5: Código usado para exibir a imagem em função de um vetor próprio do dígito 0.

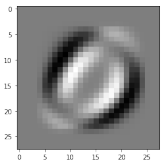


Figura 4.6: Output código 4.5.

4.4 Resultados da aplicação do PCA para cada conjunto de imagens de cada dígito

Aplicando o PCA para cada dígito entre 0 e 9, usando a função ***pca(X_digito[i], 0.9)***, obtemos diferentes resultados, como o número de componentes principais ***k_i***, os valores próprios ***val_prop_i***, os vetores próprios ***vet_prop_i***, a matriz de dados centralizados ***phi_i***, a média dos dados ***media_i***, a variância ***v_i***, e a confiança alcançada ***confianca_i***.

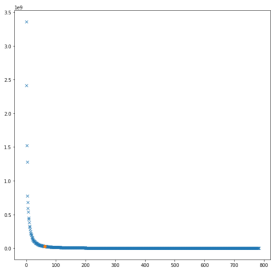
Após agregar estes resultados em arrays separados, como ***k_digito, val_prop_digito, vet_prop_digito, phi_digito, media_digito, v_digito, confianca_digito***, obteve-se a seguinte tabela para o número de componentes principais de cada modelo de dígito, para uma confiança-alvo de 90%:

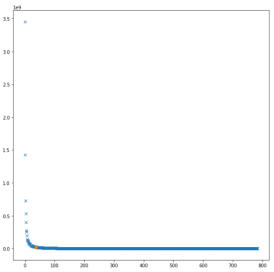
Dígito	0	1	2	3	4	5	6	7	8	9
#CP	63	37	82	81	77	76	63	67	83	63

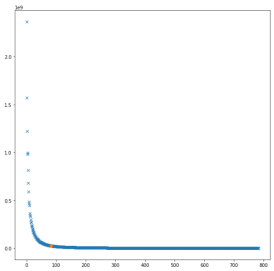
Tabela 4.1: Número de componentes principais (#CP) obtidas para cada dígito

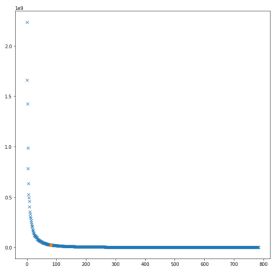
4.5 Elbow Method (Método do Cotovelo)

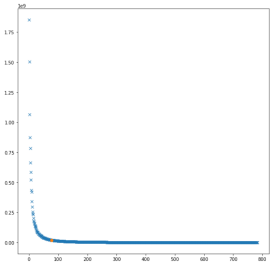
O construção dos gráficos segundo o método do cotovelo foi utilizado para verificar se o número de componentes principais obtido para cada de imagens do mesmo dígito.

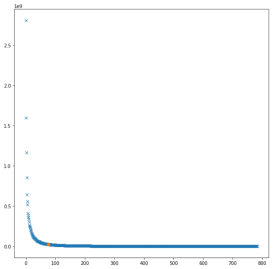

Elbow - Dígito 0

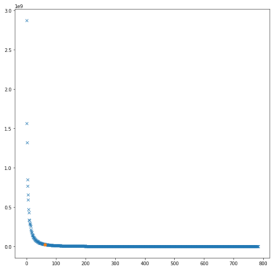

Elbow - Dígito 1

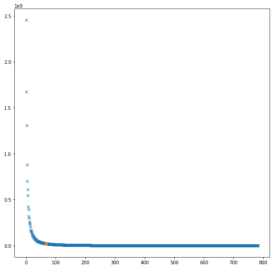

Elbow - Dígito 2

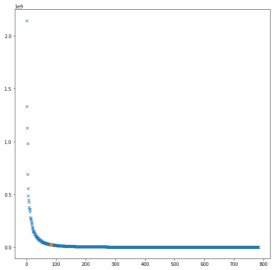

Elbow - Dígito 3


Elbow - Dígito 4


Elbow - Dígito 5


Elbow - Dígito 6


Elbow - Dígito 7


Elbow - Dígito 8

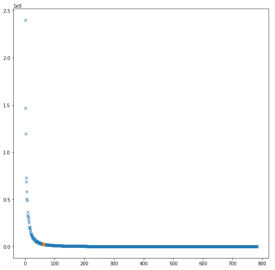

Elbow - Dígito 9

Figura 4.7: Gráficos do Método do Cotovelo obtidos para os 10 modelos de reconhecimento

Em todos os gráficos observa-se que o ponto amarelo (cuja abcissa indica o número de valores próprios e a ordenada a respetiva quantidade de informação que estes valores próprios preservam) está na zona de maior curvatura (ou seja, na zona do '*cotovelo*'). Deste modo, para os 10 modelos de cada dígito, o número de componentes principais é adequado para a preservação da informação das imagens.

4.6 Cálculo dos coeficientes das projeções para cada um dos modelos

No contexto específico do reconhecimento de dígitos, as projeções podem representar características importantes das imagens (como bordas, formas, inclinação, entre outras) de maneira mais compacta, permitindo a identificação e a comparação eficiente dos dígitos para tarefas de classificação.

Os coeficientes das projeções para cada um dos 10 são calculados a partir do produto interno entre ***phi*** e as componentes principais do modelo do dígito em causa. Todos os resultados são armazenados num único array ***coef_proj***, onde o índice ***i*** do array mostra-nos os resultados das projeções para o dígito ***i***.

Estes coeficientes serão usados mais para a frente na implementação de distâncias.

```python
# Calculo dos coeficientes da projecao
def coefProj(phi, vet_prop, size):
    coef_proj = [np.dot(phi[i], vet_prop) for i in range(size)]
    return coef_proj
```

Figura 4.8: **A função coefProj** é responsável por calcular as projeções das imagens sobre as componentes principais.

5 Leitura do dataset de teste MNIST

De forma parecida como fizemos para o dataset de treino, vamos carregar as imagens *(t10k-images.idx3-ubyte)* e seus respectivos rótulos de teste *(t10k-labels.idx1-ubyte)*. Após isso, cada imagem de teste é transformada em um array unidimensional usando o método *flatten* e estes arrays unidimensionais resultantes são armazenados na lista ***gamma***. Além disso, também é realizada uma verificação se todas as etiquetas estão dentro do intervalo $[0, 1, ..., 9]$.

Após a construção dos modelos, foi necessário testar se estes são capazes de reconhecer com precisão dígitos em imagens que lhes fornecemos.

5.1 Distribuição dos rótulos no dataset de teste

Da mesma forma que fizemos para o dataset de treino, verificamos a distribuição da frequência de cada dígito (de 0 a 9) no conjunto de dados de teste do MNIST (aproximadamente 1000 imagens de teste para cada um dos 10 dígitos possíveis).

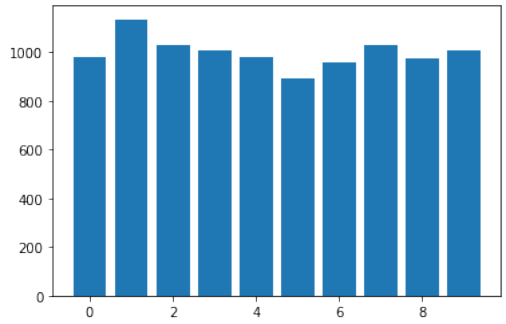


Figura 5.1: Distribuição da frequência no dataset de teste.

6 Distâncias: Euclidiana e Mahalanobis

Neste projeto, implementaremos duas métricas para verificar a distância dentro do nosso conjunto de dados, são elas: A **distância Euclidiana** e a **distância Mahalanobis**.

A distância euclidiana é uma métrica de distância padrão que mede a distância ”direta”entre dois pontos em um espaço euclidiano. Isto é, para um par de pontos

$$x = (x_1, x_2, x_3, ..., x_n) \; e \; y = (y_1, y_2, y_3, ..., y_n)$$

a distância euclidiana é dada por:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{1}$$

Já a distância Mahalanobis é uma medida que leva em conta a correlação entre variáveis em um conjunto de dados. Ela é uma generalização da distância euclidiana que considera a matriz de covariância dos dados.

Consideremos dois vetores x e y com matriz de covariância C. Como vimos na Secção 4.2, o SVD expande os dados originais para sistema de coordenadas onde a matriz de covariância é diagonal. Dessa forma, a fórmula de Mahalanobis que usaremos é dada, para os valores próprios $(\lambda_1, \lambda_2, ...\lambda_p)$, por:

$$d(x, y) = \sum_{i=1}^{p} \frac{1}{\lambda_p}(x_i - y_i)^2 \tag{2}$$

6.1 Implementação das distâncias Euclidiana e Mahalanobis

Para a implementação da distância Eucludiana no nosso projeto, primeiro a definimos na função ***euclidiana***, que utiliza o método ***np.linalg.norm***.

Para implementar a distância Mahalanobis, definimos a função ***mahalanobis***, que usa os métodos ***np.sum*** e ***np.divide***.

Além disso, também foram criadas funções para calcular a distância mínima entre as projeções do dataset. Foi feito o seguinte:

1. Criamos a função ***test_coef_proj***, que recebe como entrada o conjunto de teste ***gamma*** (Relembramos que gamma é um array unimensional criado na Secção 5), a média dos dígitos conhecidos ***media***, e os vetores próprios ***vet_prop*** associados aos dígitos. Onde o que esta função faz é:
   (a) Centralizar os dados de ***gamma*** subtraindo a média.
   (b) Calcular os coeficientes de projeção para o conjunto de teste utilizando os vetores próprios.
   (c) Retornar os coeficientes de projeção calculados para o conjunto de teste.
2. Criamos a função ***dist_min_euclidiana***, que:
   (a) Recebe como entrada os coeficientes de projeção calculados pela função anterior ***test_coef_proj***, o tamanho do conjunto de projeções dos dígitos conhecidos ***size***, e as projeções dos dígitos ***coef_proj*** (como foi definida na Secção 4.6).
   (b) Calcula a distância euclidiana entre os coeficientes de projeção do dataset e cada conjunto de projeções dos dígitos conhecidos usando a função ***euclidiana***, criada anteriormente.
   (c) Retorna a menor distância euclidiana encontrada.
3. Criamos a função ***dist_min_mahalanobis***, que:
   (a) Recebe como entrada os coeficientes de projeção calculados pela função anterior ***test_coef_proj***, os valores próprios ***val_prop***, o tamanho do conjunto de projeções dos dígitos conhecidos ***size***, e as projeções dos dígitos ***coef_proj*** (como foi definida na Secção 4.6), e k os componentes principais.
   (b) Calcula a distância de Mahalanobis normalizada entre os coeficientes de projeção do conjunto de teste e cada conjunto de projeções dos dígitos conhecidos usando a função ***mahalanobis***.
   (c) Retorna a menor distância de Mahalanobis encontrada.

6.2 Identificação do dígito numa imagem de teste usando a distância Euclidiana

Agora, vamos realizar testes usando as distâncias implementadas. Para a distância euclidiana, foi feito o seguinte:

1. Foi criada a função ***testar_imagem_euclidiana(imagem)***, que recebe uma imagem como entrada para tentar reconhecer o dígito contido nela, com os passos especificados a seguir:
   (a) Cria-se uma lista vazia de 10 elementos (um para cada dígito de 0 a 9), que será usada para armazenar as distâncias mínimas calculadas.

(b) Itera sobre os possíveis dígitos de 0 a 9.

(c) Centraliza os dados da imagem de teste subtraindo a média do conjunto de treinamento associada ao dígito atual (***media_digito[dig]***).

(d) Calcula os coeficientes de projeção para a imagem de teste utilizando os vetores próprios associados ao dígito atual (***vet_prop_digito[dig]***).

(e) Calcula a distância mínima euclidiana entre os coeficientes de projeção da imagem de teste e os coeficientes de projeção do dígito atual usando a função ***dist_min_euclidiana*** definida anteriormente.

(f) Depois, usando a função ***np.min***, calculamos o valor mínimo de distância na lista ***dist_min_digito***.

(g) O dígito reconhecido corresponderá ao valor mínimo no array ***dist_min_digito***.

(h) Por fim, a função retornará o dígito reconhecido e a distância mínima calculada.

2. Criamos dois arrays vazios, o ***results_euclidiana_imagens*** (que armazenará as previsões dos dígitos reconhecidos para cada imagem), e ***dists_euclidiana_test*** (que armazenará as distâncias calculadas para cada imagem em relação ao dígito reconhecido), e um array com zeros, o ***positives_euclidiana*** (que armazenará a contagem de casos em que o dígito previsto é igual ao verdadeiro dígito da imagem para cada dígito de 0 a 9).

3. Agora, criamos um loop que passa por cada uma das 10.000 imagens de teste, que para cada imagem:

   (a) Obtém a imagem e sua respectiva label (***label***) do conjunto de teste (***gamma*** e ***label_test_array***).

   (b) Usa a função ***testar_imagem_euclidiana(imagem)*** para reconhecer o dígito na imagem.

   (c) Armazena o dígito reconhecido em ***results_euclidiana_imagens***.

   (d) Verifica se o dígito reconhecido é igual ao dígito real da imagem (label). Se for, incrementa o contador correspondente em ***positives_euclidiana***.

   (e) Armazena a distância mínima calculada para a imagem em ***dists_euclidiana_test***.

6.3 Accuracy Total e dos 10 Modelos - Distância Euclidiana

Após aplicação da função ***testar_imagem_euclidiana*** nas imagens de teste, para se avaliar a sua eficácia no reconhecimento dos dígitos, utilizou-se a fórmula da accuracy para dados de classificação não binários:

$$accuracy = \frac{\#\text{ classificações corretas}}{\#\text{ número total classificações}} \text{ OU accuracy\_digito} = \frac{\#\text{ classificações corretas dígito}}{\#\text{ total labels dígito}} \quad (3)$$

tendo-se obtido a seguinte tabela:

Accuracy	Total	Dígito 0	Díg. 1	Díg. 2	Díg. 3	Díg. 4	Díg. 5	Díg. 6	Díg. 7	Díg. 8	Díg. 9
Valor (%)	83.96	92.04	99.91	65.69	77.82	78.51	79.37	90.91	86.08	79.36	88.00

Tabela 6.1: Resultados da Accuracy Total e dos 10 modelos de dígito, usando a distância euclidiana

A accucury total de 83.96 % prova que o reconhecimento de dígitos com a distância euclidiana foi concluída com sucesso na maior parte das imagens, excetuando-se no caso do dígito 2, onde a sua accuracy foi consideravelmente mais reduzida. Mas estes resultados devem ser vistos com cuidado, tal como se vai explicar na secção seguinte.

6.4 Limitação identificada na Distância Euclidiana e Solução

Um dos problemas com a utilização da distância euclidiana como métrica para ver qual distâncias entre os 10 modelos de dígitos é a menor é que as distâncias geradas podem provir de vetores com dimensões de espaço muito diferentes entre si. Isso pode trazer problemas na comparação de resultados, principalmente quando se faz a comparação das distâncias entre os 10 modelos para uma mesma imagem pela distância de menor valor, em vez de se utilizar um limite de similaridade (que não foi implementado devido à complexidade, tempo de execução elevados e testes que este exigiria).

Para ultrapassarmos esta limitação, existe a distância de Mahalanobis (também conhecida por distância euclidiana Ponderada), onde o somatório do quadrado da diferença entre as coordenadas dos dois vetores é dividido pelos valores próprios que pertencem às componenetes principais. Deste modo, a distância euclidiana é equilibrada pelos valores próprios, tirando o efeito das dimensões dos vetores no cálculo da distância.

6.5 Identificação do dígito numa imagem de teste usando a distância Mahalanobis

Para a distância de Mahalanobis, o processo foi análogo ao da secção 6.2, adaptando os nomes das funções e variáveis de ***... _euclidiana*** para ***... _mahalanobis***.

6.6 Accuracy Total e dos 10 Modelos - Distância de Mahalanobis

Agora, de forma análoga à secção 6.3, calculamos a eficácia da função ***testar_imagem_mahalanobis*** utilizando a fórmula de accuracy novamente. Tendo-se obtido a seguinte tabela:

Accuracy	Total	Dígito 0	Díg. 1	Díg. 2	Díg. 3	Díg. 4	Díg. 5	Díg. 6	Díg. 7	Díg. 8	Díg. 9
Valor (%)	90.64	98.26	99.82	87.69	87.92	82.99	83.29	96.86	87.93	85.31	94.54

Tabela 6.2: Resultados da Accuracy Total e dos 10 modelos de dígito, usando a distância de Mahalanobis

As accuracys em todos os casos (total e dos dígitos) foram bastante elevadas, pelo que se pode concluir que o uso da distância de Mahalanobis como métrica permitiu o reconhecimento correto dos dígitos em quase da totalidade do dataset de teste.

6.7 Comparação dos Resultados da distância de Mahalanobis com os da Euclidiana

Tal como se esperava, o uso da distância de Mahalanobis mostrou-se muito mais eficaz no reconhecimento de dígitos do que através da distância Euclidiana (basta comparar os valores das accuracys das tabelas 6.1 e 6.2). Este resultado é um exemplo de que a escolha de métricas adequadas para verificar a similaridade de resultados é muito importante para a construção de bons modelos de Machine Learning.

7 Possíveis causas de erro nas imagens

Apesar das elevadas accuracys obtidas para as duas distâncias, o programa não foi capaz de identificar corretamente todas as imagens de teste fornecidas. Em alguns casos, deve-se à semelhança de uma imagem de um dígito com a de um número diferente do esperado; por outro lado, existem imagens cujos dígitos estão mal desenhados, dificultando uma análise correta por parte do programa. Mostram-se de seguida alguns exemplos:

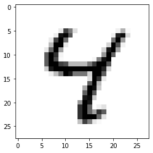


Figura 7.1: Não Reconhecimento distância Euclidiana

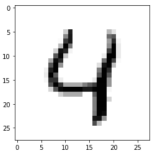


Figura 7.2: Não Reconhecimento distância Mahalanobis

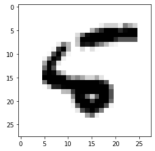


Figura 7.3: Não Reconhecimento ambas as distâncias

A figura 8.1 mostrou um caso em que o uso da distância euclidiana não fez com que reconhecesse que se tratava do dígito 4; em vez disso, identificou-o como o número 1, provavelmente devido à semelhança da parte da direita da imagem com o desenho do dígito 1. Já para a figura 8.2, a distância de Mahalanobis não foi eficaz para reconhecer o dígito 4, reconhecendo-o como 9 (faltaria a parte de cima do 4 para parecer 9). Por fim, na figura 8.3, observamos um caso em que o uso quer da distância euclidiana, quer a de mahalanobis não foram capazes de reconhecer o dígito represnetado na imagem, muito provavelmente do desenho mal feito do dígito 5 (curiosamente, ambas as distâncias reconheceram-no como dígito 1).

8 Conclusão

O trabalho prático apresentado neste relatório mostrou como o PCA é uma técnica capaz de reconhecer dígitos em imagens com grande sucesso (desde que os dígitos estejam bem representados nas respetivas imagens). Para além disso, a utilização de uma métrica / distância correta para quando os dados têm de sofrer redução de dimensionalidade facilita a comparação das projeções da imagem com as dos modelos.

Mas, os resultados obtidos devem ser vistos com cuidado, já que as imagens fornecidas para a construção dos modelos e para os testes possuem uma qualidade relatiavemente elevada (já centralizadas, sem sombras, nem desfoques e fácil leitura). Caso as imagens fornecidas tivessem menor qualidade, como por exemplo: com sombras, desfocado, descentralizado... os modelos construídos através da técnica PCA poderiam não ter capacidade para reconhecer os dígitos representados.

Outros algoritmos de Machine Learning como de Deep Learning ou até de combinações do PCA com um algoritmo de clustering, como o K-means, poderiam trazer resultados melhores no reconhecimento, mas que estão fora do âmbito da unidade curricular.

Referências

[1] Apontamentos teóricos (PCA) - Métricas em Machine Learning

[2] StackOverflow - Extract images from .idx3-ubyte file or GZIP via Python

[3] Singular Value Decomposition (SVD) tutorial - MIT, https://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm

[4] Principal Component Analysis Tutorial, *https://www.dezyre.com/data-science-in-python-tutorial/principal-component-analysis-tutorial*

[5] Principal Component Analysis in 3 Simple Steps, *https://sebastianraschka.com/Articles/2015_pca_in_3_steps.html*

[6] PCA and SVD explained with numpy, *https://towardsdatascience.com/pca-and-svd-explained-with-numpy-5d13b0d2a4d8*