Trabalho Final - Inteligência Computacional Aplicada (TIP7077)

Aluno: Carlos Eduardo Sousa Lima

Prof. Guilherme de Alencar Barreto

Questão 01 - Classificação de Padrões - MNIST database of handwritten digits

As classes e funções apresentadas no bloco de código abaixo foram comuns a todos os classificadores implementados. Nelas, estão implementadas a leitura dos dados, adequação da codificação do vetor de saída (alvo) e normalização dos dados. A seguir, cada uma delas é melhor descrita:

- MNIST_data:

  Essa classe foi criada para conter as funções de aquisição dos dados de treino **get_train_data()** e dados de teste **get_test_data()**. Essas funções utilizam a biblioteca mnist (#https://pypi.org/project/python-mnist/#description), a qual, a partir dos dados obtidos em http://yann.lecun.org/exdb/mnist/index.html, descomprime e transforma-os em um objeto Numpy Array (np.array). Cada uma dessas funções retornam dois objetos np.array, um com os dados de entrada e o outro com seus respectivos labels. Os dados de entrada são retornados de forma vetorizada, ou seja, a matriz 28×28 é empilhada dando origem a um vetor 784×1. Os labels são o valor inteiro entre 0 e 9 que esse vetor representa. Cabe destacar que a base de dados de treino possuem 60.000 elementos e a de teste 10.000 elementos. Dessa forma, os dados de entrada formam uma matriz 60.000×738, para a base de treino, e 10.000×738, para a base de teste. Os labels, por sua vez, forma um vetor de 60.000×1, para a base de treino, e 10.000×1, para a base de teste

- one_hot_enconding():

  Essa função altera o formato do vetor labels. Para cada elemento da base de dados de entrada do MNIST, seja a de treino ou de teste, existirá um label que representará o valor associado a esse elemento da base de dados. Tomando como exemplo os dados de treino, sua base de dados terá dimensão 60.000×738, visto que o procedimento de obtenção dos dados retorna esses dados de forma vetorizada. No caso dos seu respectivo vetor de label ($\vec{y}$), ele terá dimensão 60.000×1. Os possíveis valores de cada um dos elementos contidos no vetor de labels são representados por $x \in [0, 9]$, tal que $x \in Z$. A função aqui descrita, portanto, cria uma nova codificação para os labels baseada na codificação one-hot. Como x pode assumir 10 valores (classes), cada elemento do vetor de labels dará origem a um vetor de cardinalidade igual 10. Portanto, após a aplicação dessa função, o vetor de label $\vec{y}$ passará a ter dimensão 60.000×10, para a base de treino, e 10.000×10, para a base de teste. Esse vetor terá valor igual a 1 no índice que coincide com o valor representando no respectivo elemento do vetor de labels original, nos demais índices receberá o valor 0. Exemplificando, após a utilização da função one_hot_enconding():

- $0 \rightarrow [1,0,0,0,0,0,0,0,0,0]$
- $1 \rightarrow [0,1,0,0,0,0,0,0,0,0]$
- $\vdots \rightarrow \qquad \vdots$
- $9 \rightarrow [0,0,0,0,0,0,0,0,0,1]$

A utilização dessa codificação é interessante, pois assume que os vetores que representam cada classe são ortogonais entre si, ou seja, mutuamente exclusivos.

- norm_data():

A MNIST *database of handwritten digits* consiste na representação matricial de imagens de caracteres cursivos. Nessa representação, utiliza-se matrizes quadradas de dimensão 28×28, ou seja, cada imagem contém 784 pixels. Em cada um desses pixels, representasse uma tonalidade de cinza, considerando uma escala em que zero é totalmente branco e 255 totalmente preto. Como supracitado, cada uma dessas matrizes de dados são vetorizadas nas presente analises, dando origem a um vetor de dimensão $1 \times p$. Agrupando esses vetores em linhas, obtém-se a matriz de dados de dimensão $n \times p$ utilizada nesse trabalho, sendo n o número de imagens disponibilizadas para treino ou teste e $p = 784$. A função nomr_data() atua normalizando os valores da escala de tons de cinza associados aos pixels das imagens de caracteres cursivos. Em outras palavras, para cada n elemento da matriz de dados, seja de treino ou de teste, essa função varia intervalo de variação dos valores de $[0, 255]$ para $[0, 1]$. O processo de normalização é amplamente recomendado para algoritmos de classificação baseado em aprendizado, pois dados de entrada com elevados valores, ou que suas variáveis apresentem grandes diferença na magnitude dos seus valores, podem prejudicar o processo de aprendizado. Para normalização desses dados, adotou-se a seguinte equação:

$$x_j^{norm} = \frac{x_j - x_j^{max}}{x_j^{max} - x_j^{min}}$$

In [ ]:
```
import time
import numpy as np
from mnist import MNIST  # https://pypi.org/project/python-mnist/#description
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
from collections import namedtuple
import warnings
warnings.filterwarnings("ignore")


class MNIST_data():

    def __init__(self):
        pass

    def get_train_data():

        mndata = MNIST("./")
        mndata.gz = True
        x, y = mndata.load_training()

        return (np.array(x), np.array(y))

    def get_test_data():

        mndata = MNIST("./")
        mndata.gz = True
        x, y = mndata.load_testing()

        return (np.array(x), np.array(y))


def one_hot_enconding(y, n):

    y_enc = np.zeros((y.shape[0], n))
    for i in range(y.shape[0]):
        y_enc[i, y[i]] = 1

    return y_enc


def norm_data(df):
    df_std = np.zeros(df.shape)
```

```python
    for i in range(df.shape[0]):
        # df_std[i,:] = (df[i,:] - df[i,:].mean())/(df[i,:].std(ddof = std_ddof))
        df_std[i, :] = (df[i, :] - df[i, :].min()) / \
            (df[i, :].max() - df[i, :].min())
    return df_std
```

Avaliando o posto das matrizes dos dados de treino (X) e dados de teste (X_test)

In [ ]:
```python
X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

if np.linalg.matrix_rank(X) == min(X.shape):
    print("Dados de entrada (X) - Matriz de Posto Completo\n")
else:
    print("Dados de entrada (X) - Matriz de Posto Incompleto\n")
if np.linalg.matrix_rank(X_test) == min(X_test.shape):
    print("Dados de entrada (X_test) - Matriz de Posto Completo\n")
else:
    print("Dados de entrada (X_test) - Matriz de Posto Incompleto\n")
```

```
Dados de entrada (X) - Matriz de Posto Incompleto

Dados de entrada (X_test) - Matriz de Posto Incompleto
```

Classificador Linear de Mínimos Quadrados

In [ ]:
```python
# O código está utilizando a orientação nxp
# n é o número de amostras, p o número de característica
# As operações matriciais apresentam ordem contrária ao apresentando nas notas de aula
# Y = W*X (Nota de Aula) - Y = X*W (Presente Código)
def LSM_Class(X, Y, X_test, Y_test, Nr):
    Nr = 5
    tx_ok = np.zeros(Nr)
    tic = time.perf_counter()
    for r in range(Nr):
        rand_index = np.random.permutation(X.shape[0])
        X = X[rand_index,:]
        Y = Y[rand_index,:]

        if X.shape[0] != X.shape[1]:
            W = np.linalg.lstsq(X,Y)[0]
```

```
        else:
            W = np.linalg.solve(X,Y)[0]


        Y_mod = np.dot(X_test, W)
        count_ok = 0

        for j in range(Y_mod.shape[0]):
            if Y_mod[j,:].argmax() == Y_test[j,:].argmax():
                count_ok += 1

        tx_ok[r] = count_ok/Y_mod.shape[0]
        print("Rodada {} - Taxa de Acerto = {:.2%}".format(r+1, tx_ok[r]))

    toc = time.perf_counter()

    print("\nTaxa de acerto Média = {:.2%}".format(tx_ok.mean()))
    print("Taxa de erro Média = {:.2%}".format(1-tx_ok.mean()))
    print("Melhor Taxa de Acerto = {:.2%}".format(tx_ok.max()))
    print("Pior Taxa de Acerto = {:.2%}".format(tx_ok.min()))
    print("Desv. Pad. Taxa de Acerto = {:.2%}".format(tx_ok.std()))

    return("Tempo de Calibração e Validação para {} rodadas = {} segundos".format(Nr, round(toc-tic, 2)))
```

In [ ]:

```
X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

LSM_Class(X, Y, X_test, Y_test, 5)
```

```
Rodada 1 - Taxa de Acerto = 85.30%
Rodada 2 - Taxa de Acerto = 85.34%
Rodada 3 - Taxa de Acerto = 85.31%
Rodada 4 - Taxa de Acerto = 85.28%
Rodada 5 - Taxa de Acerto = 85.33%

Taxa de acerto Média = 85.31%
Taxa de erro Média = 14.69%
Melhor Taxa de Acerto = 85.34%
Pior Taxa de Acerto = 85.28%
Desv. Pad. Taxa de Acerto = 0.02%
```

Out[ ]:

```
'Tempo de Calibração e Validação para 5 rodadas = 25.28 segundos'
```

De acordo com LeCun *et al.* (1998), o classificador linear baseado em uma rede neural de uma camada (*1-layer NN*) obteve uma taxa de erro de 12.00% com a base de dados de teste, considerando nenhum pré-processamento desses dados.

O Classificador Linear de Mínimos Quadrados (CLMQ), por sua vez, apresenta um desempenho similar, com uma taxa de erro média de 14.68%. O desempenho desse classificador foi considerado satisfatório e evidencia que esse problema de classificação pode ser resolvido satisfatoriamente por uma superfície de decisão linear, ou seja, um problema linearmente separável.

Destaca-se que a operação para determinação da matriz $\tilde{W}$ utilizou o método de eliminação de Gauss, não sendo necessário a inversão explícita de $\tilde{X}\tilde{X}^T$. Como apresentado, as matrizes de dados de entrada para treino e teste possuem posto incompleto, logo haveria problema nessa inversão explícita supracitada.

Destaca-se, também, que o método de eliminação de Gauss possui um menor custo de processamento, sendo eficiente para dados de alta dimensão, como os utilizados no presente trabalho.

Perceptron Logístico

In [ ]:

```python
def act_fun(u, fun):
    action_fun = namedtuple("act_fun", ["f", "df"])
    if fun == "step":
        u[np.where(u >= 0)] = 1
        u[np.where(u < 0)] = 0
        du = np.nan
    elif fun == "tanh":
        u = np.array(list(map(lambda x: (1-np.exp(-x))/(1+np.exp(-x)), u)))
        du = 0.5*(1-np.power(u,2)) + 0.05
```

```python
    elif fun == "log":
        u = np.array(list(map(lambda x: 1/(1+np.exp(-x)), u)))
        du = u*(1-u) + 0.05
    return (action_fun(f = u, df = du))

def Hardamad_Prod(a, b):

    prod = np.array([np.multiply(x,y) for x, y in zip(a,b)])

    return prod

def PS (X, Y, X_test, Y_test, eta, Nr, Ne, fun_type, mom):
#Função que treina e valida a Perceptron Simples
    tx_ok = np.empty((Nr))
    best_run = {"Acc": 0, "EQM": 0, "W": 0}
    worst_run = {"Acc": 1, "EQM": 0, "W": 0}
    EQM_ep = np.empty(0)

    if fun_type == "tanh":
        #codificação da saída fica -1 e 1 para tangente hiperbólica
        Y[Y == 0] = -1
        Y_test[Y == 0] = -1

    tic = time.perf_counter()

    for r in range(Nr):
        #Não embaralhei a cada rodada, pois os dados de treino sempre serão X e Y
        #Assim, como não haverá split do conjunto em dados de treino e teste
        #Decidi embaralhar só dentro de cada época
        #Inicialização aleatória dos pesos
        ###Treino###
        W = np.random.rand(Y.shape[1], X.shape[1]+1)
        W_old = W.copy()
        for ep in range(Ne):
            #Embaralhamento da matriz de dados saída
            rand_index = np.random.permutation(X.shape[0])
            X = X[rand_index, :]
            Y = Y[rand_index, :]
            EQM = 0

            for i in range(X.shape[0]):
                x = np.append(-1, X[i,:]) #add bias
                U = np.dot(W, x)
                y, dy = act_fun(U, fun_type)
```

```python
                err = Y[i,:] - y

                EQM = EQM + 0.5*np.power(err, 2).sum()

                x = np.expand_dims(x, 1)

                Di = np.expand_dims(Hardamad_Prod(err, dy), 1)

                W_aux = W.copy()

                W = W + eta*np.dot(Di, x.T) + mom*(W - W_old)

                W_old = W_aux.copy()

            EQM_ep = np.append(EQM_ep, EQM/X.shape[0])

        ###Validação###
        count = 0
        for j in range(X_test.shape[0]):
            x = np.append(-1, X_test[j,:])
            U = np.dot(W, x)
            y = act_fun(U, fun_type).f

            if Y_test[j,:].argmax() == y.argmax():
                count += 1

        tx_ok[r] = count/X_test.shape[0]

        if tx_ok[r] > best_run["Acc"]:
            best_run["Acc"], best_run["EQM"], best_run["W"] = (tx_ok[r], EQM_ep, W)


        if tx_ok[r] < worst_run["Acc"]:
            worst_run["Acc"], worst_run["EQM"], worst_run["W"] = (tx_ok[r], EQM_ep, W)

        print("Rodada {} - Taxa de acerto = {:.2%}".format(r+1, tx_ok[r]))
    toc = time.perf_counter()

    EQM_ep = EQM_ep.reshape((Nr,Ne))
    fig, ax = plt.subplots()
    for i in range(EQM_ep.shape[0]):
        ax.plot(EQM_ep[i,:], label = "Rodada {}".format(i+1))
    ax.legend()
    ax.set_ylabel("EQM")
```

```
    ax.set_xlabel("epochs")

    print("\nTaxa de acerto Média = {:.2%}".format(tx_ok.mean()))
    print("Taxa de erro Média = {:.2%}".format(1-tx_ok.mean()))
    print("Melhor Taxa de Acerto = {:.2%}".format(tx_ok.max()))
    print("Pior Taxa de Acerto = {:.2%}".format(tx_ok.min()))
    print("Desv. Pad. Taxa de Acerto = {:.2%}".format(tx_ok.std()))

    return("Tempo de Calibração e Validação para {} rodadas = {} segundos".format(Nr, round(toc-tic, 2)))
```

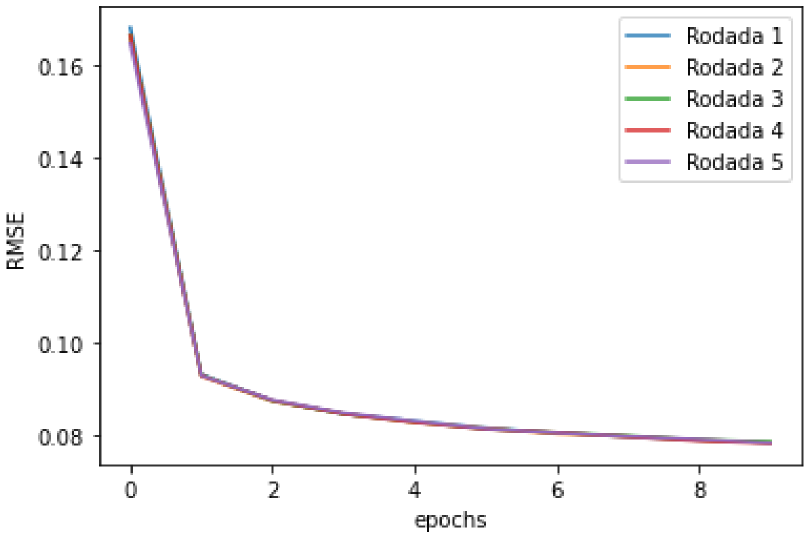
In [ ]:

```
# O código está utilizando a orientação nxp
# n é o número de amostras, p o número de característica
# As operações matriciais apresentam ordem contrária ao apresentando nas notas de aula
# Y = W*X (Nota de Aula) - Y = X*W (Presente Código)

X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

#PS (X, Y, X_test, Y_test, eta, Nr, Ne, fun_type)
PS(X, Y, X_test, Y_test, eta=0.01, Nr=5, Ne=10, fun_type="log", mom=0.85)
```

```
Rodada 1 - Taxa de acerto = 91.88%
Rodada 2 - Taxa de acerto = 91.46%
Rodada 3 - Taxa de acerto = 91.73%
Rodada 4 - Taxa de acerto = 91.09%
Rodada 5 - Taxa de acerto = 91.47%

Taxa de acerto Média = 91.53%
Taxa de erro Média = 8.47%
Melhor Taxa de Acerto = 91.88%
Pior Taxa de Acerto = 91.09%
Desv. Pad. Taxa de Acerto = 0.27%
```

Out[ ]:

```
'Tempo de Calibração e Validação para 5 rodadas = 969.36 segundos'
```



De acordo com LeCun *et al.* (1998), o classificador linear baseado em uma rede neural de uma camada (*1-layer NN*) obteve uma taxa de erro de 12.00% com a base de dados de teste, considerando nenhum pré-processamento desses dados. A taxa de erro desse classificador será usada como referência comparativa do classificador baseado na rede Perceptron Logístico.

A rede perceptron logístico implementada para classificação dessa mesma base de dados do MNIST apresentou uma taxa de erro média para 5 rodadas igual a 8.47%, taxa de erro abaixo da referência comparativa utilizada. A inserção de uma não-linearidade suave à função de ativação da rede perceptron resulta numa melhoria desse desempenho em relação a o classificador linear.

Em cada uma dessas rodadas, o treinamento dessa rede neural contabilizou 10 épocas. Para cada uma dessas épocas, embaralhou-se os dados de treinamento, evitando overfitting a esses dados. A base de treino contém 60.000 elementos, assim, atualizou-se a matriz de peso 600.000 (10 épocas). A taxa de aprendizado utilizada fio igual a $\eta = 0.01$ e a taxa de momento igual a $m = 0.85$. A taxa de momento, ou termo de momento, é utilizado para tornar mais estável o processo de utilização dos pesos.

A atualização desses pesos utilizou a regra delta generalizada. Essa regra, utiliza o gradiente local do erro do i-ésimo neurônio (δ_i), obtido pelo produtor de Hardamad entre o erro do i-ésimo neurônio (e_i) e a derivada da função de ativação da saída do i-ésimo neurônio (ϕ'_i) $(\delta_i = e_i \circ \phi'_i)$. Entende-se que essa derivada atua como moduladora do erro, com valores altos de erro gerando valores pequenos de derivada, conferindo uma estabilidade adicional no processo de aprendizado ao longo das épocas. Estabilidade, essa, maior que a da regra LMS convencional.

No tocanto a essa estabilidade, é digno de nota a adição de $0,05$ à derivada da função de ativação. Como discutido ao longo das aulas, a modulação de (e_i) feita por (ϕ'_i) pode ocasionar a paralisia da rede neural, não modificando a matriz de pesos quando erros grandes surgirem. Além da inicialização da matriz de pesos com valores pequenos, foi nos apresentado a solução *Ad-Hoc* de adicionar esse pequeno elemento à derivada da função de ativação.

PCA

In [ ]:
```
def PCA(x_train, x_test, compress, spt_point):
    # x dimensão pxn
    # p número de variáveis de cada amostra e n número de amostras

    # Centralizando os dados
    x_train = x_train - x_train.mean(axis=1, keepdims=True)
    x_test = x_test - x_test.mean(axis=1, keepdims=True)

    cov_x = np.cov(x_train)

    u, l, v = np.linalg.svd(cov_x)

    ve = np.cumsum(l)/l.sum()

    if compress == True:

        sel_index = np.where(ve <= spt_point)
        v = v[:, sel_index[0]]

        fig, ax = plt.subplots()
        ax.plot(ve)
        ax.plot(ve[sel_index], c="red", label="Dados Selecionados")
        ax.set_ylabel("Variância Explicada Acumulada")
        ax.yaxis.set_major_formatter(mtick.PercentFormatter(1))
        ax.legend()

        z = np.dot(v.T, x_train)
        z_test = np.dot(v.T, x_test)
    else:
        z = np.dot(v.T, x_train)
        z_test = np.dot(v.T, x_test)
    # Transpor para que Z saia no formato utilizado nos códigos dos classificados
    # formato nxp, n número de amostras e p número de características
    return (z.T, z_test.T)
```

PCA + Classificador Linear de Mínimos Quadrados

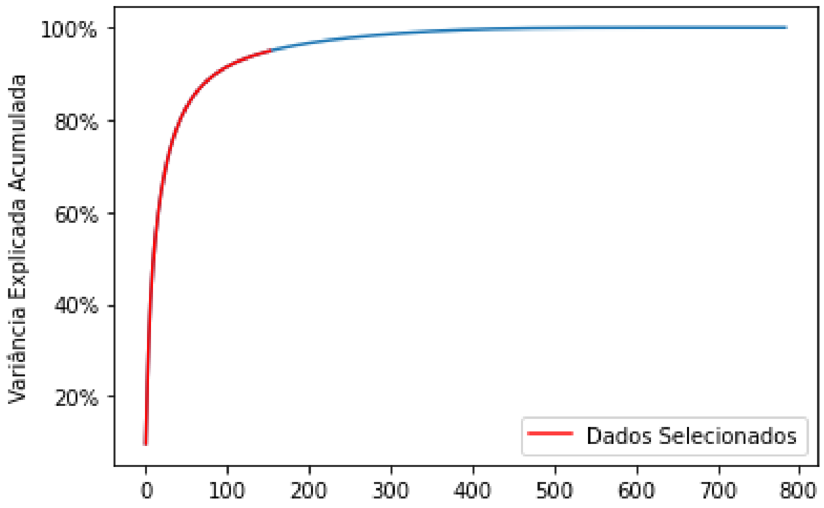
In [ ]:

```
X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

print("Dimensão da matriz de dados de treino antes do PCA: {}".format(X.shape))
print("Dimensão da matriz de dados de teste antes do PCA: {}".format(X_test.shape))
# A base de dados do MNIST é disponibilizada no seguinte padrão nxp
# n é o número de amostras, p o número de característica
# Para aplicação do PCA, deve-se transpor essa matriz para que se torne pxn

X, X_test = PCA(X.T, X_test.T, compress=True, spt_point=0.95)
print("Dimensão da matriz de dados de treino após PCA (95% da variância): {}".format(X.shape))
print("Dimensão da matriz de dados de teste após PCA (95% da variância): {}\n".format(X_test.shape))

LSM_Class(X, Y, X_test, Y_test, 5)
```

```
Dimensão da matriz de dados de treino antes do PCA: (60000, 784)
Dimensão da matriz de dados de teste antes do PCA: (10000, 784)
Dimensão da matriz de dados de treino após PCA (95% da variância): (60000, 153)
Dimensão da matriz de dados de teste após PCA (95% da variância): (10000, 153)

Rodada 1 - Taxa de Acerto = 82.09%
Rodada 2 - Taxa de Acerto = 82.09%
Rodada 3 - Taxa de Acerto = 82.09%
Rodada 4 - Taxa de Acerto = 82.12%
Rodada 5 - Taxa de Acerto = 82.09%

Taxa de acerto Média = 82.10%
Taxa de erro Média = 17.90%
Melhor Taxa de Acerto = 82.12%
Pior Taxa de Acerto = 82.09%
Desv. Pad. Taxa de Acerto = 0.01%
```

Out[ ]:

```
'Tempo de Calibração e Validação para 5 rodadas = 5.12 segundos'
```



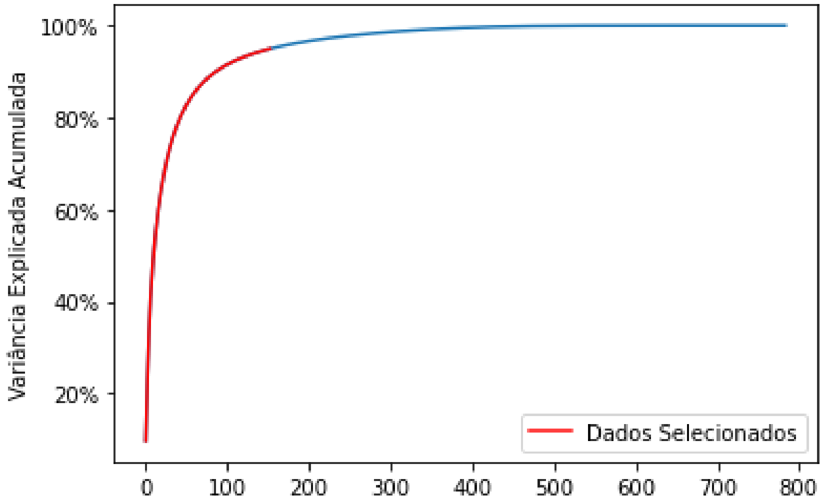
PCA + Perceptron Logístico

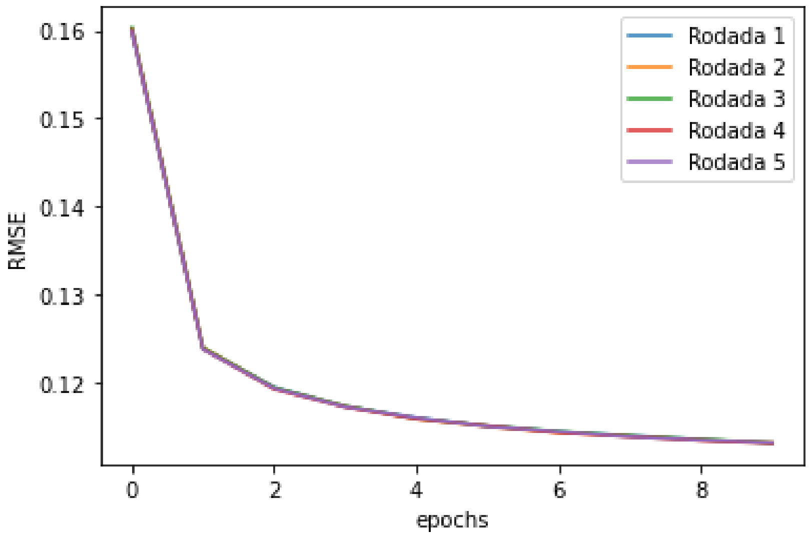
In [ ]:
```
# O código está utilizando a orientação nxp
# n é o número de amostras, p o número de característica
# As operações matriciais apresentam ordem contrária ao apresentando nas notas de aula
# Y = W*X (Nota de Aula) - Y = X*W (Presente Código)

X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

print("Dimensão da matriz de dados de treino antes do PCA: {}".format(X.shape))
print("Dimensão da matriz de dados de teste antes do PCA: {}".format(X_test.shape))
# A base de dados do MNIST é disponibilizada no seguinte padrão nxp
# n é o número de amostras, p o número de característica
# Para aplicação do PCA, deve-se transpor essa matriz para que se torne pxn

X, X_test = PCA(X.T, X_test.T, compress=True, spt_point=0.95)
print("Dimensão da matriz de dados de treino após PCA (95% da variância): {}".format(X.shape))
print("Dimensão da matriz de dados de teste após PCA (95% da variância): {}\n".format(X_test.shape))
```

```
#PS (X, Y, X_test, Y_test, eta, Nr, Ne, fun_type)
PS(X, Y, X_test, Y_test, eta=0.01, Nr=5, Ne=10, fun_type="log", mom=0.85)
```

```
Dimensão da matriz de dados de treino antes do PCA: (60000, 784)
Dimensão da matriz de dados de teste antes do PCA: (10000, 784)
Dimensão da matriz de dados de treino após PCA (95% da variância): (60000, 153)
Dimensão da matriz de dados de teste após PCA (95% da variância): (10000, 153)

Rodada 1 - Taxa de acerto = 88.82%
Rodada 2 - Taxa de acerto = 88.72%
Rodada 3 - Taxa de acerto = 88.68%
Rodada 4 - Taxa de acerto = 88.86%
Rodada 5 - Taxa de acerto = 88.87%

Taxa de acerto Média = 88.79%
Taxa de erro Média = 11.21%
Melhor Taxa de Acerto = 88.87%
Pior Taxa de Acerto = 88.68%
Desv. Pad. Taxa de Acerto = 0.08%
```

Out[ ]: 'Tempo de Calibração e Validação para 5 rodadas = 866.46 segundos'





A *Principal Component Analysis* (PCA) foi utilizada para remoção de redudância dos dados de treino e dados de teste. Entende-se por redundância, a covariância entre as variáveis que descrevem os elementos das bases de dados (treino e teste). Dessa forma, a matriz de covariância dos dados transformados é uma matriz diagonal.

Além da remoção das redundâncias, a PCA permite redução de dimensionalidade dessas bases de dados, mantendo apenas uma determinada variância explicada acumulada. Na presente aplicação, a redução de dimensionalidade aplicada manteve 95% da variância explicada dessas bases de dados. Nessa redução, o número de variáveis de cada elemento dos dados de treino e teste passou de 784 para 153, assim as matrizes de dados de treino e teste tiveram, respectivamente as seguintes alterações em suas dimensões: $(60.000 \times 784) \rightarrow (60.000 \times 153)$ e $(10.000 \times 784) \rightarrow (10.000 \times 153)$.

Após a aplicação do PCA nos dados de treino e teste, o desempenho dos classificadores Linear de Mínimos Quadrados e Percepton Logístico tiveram, respectivamente, uma taxa de erro médio (5 rodadas) de 17,90% e 11,21%. Observa-se, na devida ordem, um aumento de +3,22% e +2,74% na taxa de erro médio desses classificadores.

Apesar dessa piora no desempenho dos classificadores, é digno de nota a melhoria nos seus tempos de execução. O classificador de mínimo quadrados apresentou, para 5 rodadas, um tempo de execução de 25,28 segundos utilizando a base de dados original e 5,12 segundos utlizando a base de dados transformada pelo PCA; uma redução de 20,16 segundos, ou seja, uma redução de quase 80% no tempo de execução. Já para o perceptron logístico, também para 5 rodadas, teve seu tempo de execução indo de 969,36 segundos para 866,46 segundos, uma redução de 102,9 segundos (-10,62%).

Vale ressaltar, também, a redução do desvio padrão da taxa de acerto com a aplicação do PCA para o Perceptron logístico, reduzindo de 0.27% para 0.08%.

MLP

In [ ]:
```
def act_fun(u, fun):
    action_fun = namedtuple("act_fun", ["f", "df"])
    if fun == "step":
        u[np.where(u >= 0)] = 1
        u[np.where(u < 0)] = 0
        du = np.nan
    elif fun == "tanh":
        u = np.array(list(map(lambda x: (1-np.exp(-x))/(1+np.exp(-x)), u)))
        du = 0.5*(1-np.power(u, 2)) + 0.05
    elif fun == "log":
        u = np.array(list(map(lambda x: 1/(1+np.exp(-x)), u)))
        du = u*(1-u) + 0.05
    return (action_fun(f=u, df=du))


def Hardamad_Prod(a, b):

    prod = np.array([np.multiply(x, y) for x, y in zip(a, b)])

    return prod


def weight_matrix(dict_q):
    # Cria as matrizes de peso sinapticos
    # Mantendo a compatibilidade entre as dimensões de forma que viabiliza as operações matriciais

    W = {}
    for i in range(len(dict_q)):
        if i == 0:
            # camada oculta
            W[i] = np.random.rand(dict_q[i], X.shape[1] + 1)*0.01
        else:
            # Demais camadas, se houver
            W[i] = np.random.rand(dict_q[i], dict_q[i-1] + 1)*0.01
    return (W)
```

```python
def MLP(X, Y, X_test, Y_test, eta, Nr, Ne, mon, fun_type):

    tx_ok = np.empty((Nr))
    EQM_ep = np.empty(0)
    fun_type = "log"

    if fun_type == "tanh":
        # codificação da saída fica -1 e 1 para tangente hiperbólica
        Y[Y == 0] = -1
        Y_test[Y == 0] = -1

    # Arquitetura da MLP (p, q, c) - 1 Camada Ocula
    # p = entradas, q = neuronicos oculos, c = neuronios de saída
    # dict_q Número de neuronios em cada camada

    dict_q = {
        0: 89,
        1: Y.shape[1]
    }

    tic = time.perf_counter()
    for r in range(Nr):
        W = weight_matrix(dict_q)
        W_old = W.copy()

        for ep in range(Ne):
            rand_index = np.random.permutation(X.shape[0])
            X = X[rand_index, :]
            Y = Y[rand_index, :]
            EQM = 0

            for i in range(X.shape[0]):
                x = np.append(-1, X[i, :])
                U1 = np.dot(W[0], x)
                z, dz = act_fun(U1, fun_type)

                z = np.append(-1, z)
                U2 = np.dot(W[1], z)
                y, dy = act_fun(U2, fun_type)

                err = Y[i, :] - y

                EQM = EQM + 0.5*np.power(err, 2).sum()
```

```
                err = np.expand_dims(Hardamad_Prod(err, dy),  1)

                x = np.expand_dims(x, 1)
                z = np.expand_dims(z, 1)

                W_aux = W.copy()

                W[0] = W[0] + eta*np.dot(Hardamad_Prod(dz, np.dot(
                    W[1][:, 1:].T, err)), x.T) + mon*(W[0] - W_old[0])

                W[1] = W[1] + eta*np.dot(err, z.T) + mon*(W[1] - W_old[1])

                W_old = W_aux.copy()

            EQM_ep = np.append(EQM_ep, EQM/X.shape[0])

        count = 0
        for j in range(X_test.shape[0]):
            x = np.append(-1, X_test[j, :])
            U1 = np.dot(W[0], x)
            z = act_fun(U1, fun_type).f
            z = np.append(-1, z)
            U2 = np.dot(W[1], z)
            y = act_fun(U2, fun_type).f

            if Y_test[j, :].argmax() == y.argmax():
                count += 1

        tx_ok[r] = count/X_test.shape[0]
        print("Rodada {} - Taxa de acerto = {:.2%}".format(r+1, tx_ok[r]))
    toc = time.perf_counter()

    EQM_ep = EQM_ep.reshape((Nr, Ne))
    fig, ax = plt.subplots()
    for i in range(EQM_ep.shape[0]):
        ax.plot(EQM_ep[i, :], label="Rodada {}".format(i+1))
    ax.legend()
    ax.set_ylabel("EQM")
    ax.set_xlabel("epochs")

    print("\nTaxa de acerto Média = {:.2%}".format(tx_ok.mean()))
    print("Taxa de erro Média = {:.2%}".format(1-tx_ok.mean()))
    print("Melhor Taxa de Acerto = {:.2%}".format(tx_ok.max()))
```

```
    print("Pior Taxa de Acerto = {:.2%}".format(tx_ok.min()))
    print("Desv. Pad. Taxa de Acerto = {:.2%}".format(tx_ok.std()))

    return("Tempo de Calibração e Validação para {} rodadas = {} segundos".format(Nr, round(toc-tic, 2)))
```

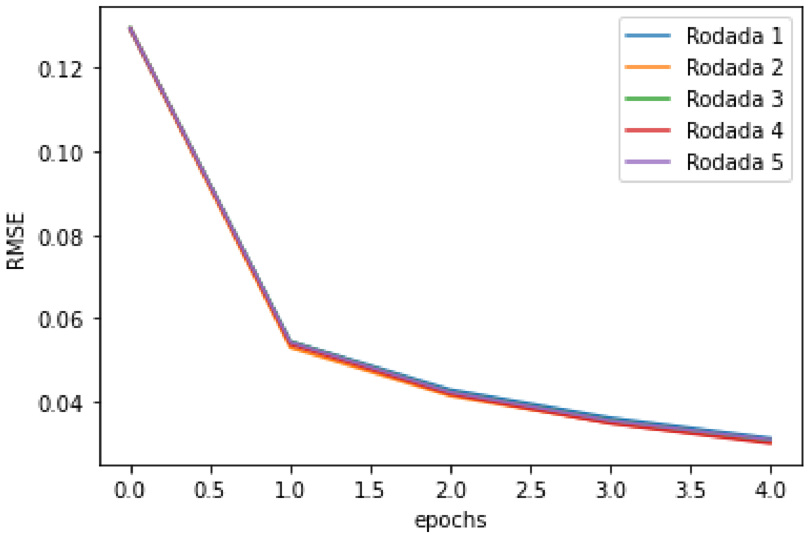
In [ ]:

```
X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

#LP(X, Y, X_test, Y_test, eta, Nr, Ne, mon, fun_type)
MLP(X, Y, X_test, Y_test, eta=0.05, Nr=5, Ne=5, mom=0.75, fun_type="log")
```

```
Rodada 1 - Taxa de acerto = 96.31%
Rodada 2 - Taxa de acerto = 96.62%
Rodada 3 - Taxa de acerto = 96.34%
Rodada 4 - Taxa de acerto = 96.74%
Rodada 5 - Taxa de acerto = 96.51%

Taxa de acerto Média = 96.50%
Taxa de erro Média = 3.50%
Melhor Taxa de Acerto = 96.74%
Pior Taxa de Acerto = 96.31%
Desv. Pad. Taxa de Acerto = 0.16%
```

Out[ ]:

```
'Tempo de Calibração e Validação para 5 rodadas = 2007.28 segundos'
```



De acordo com LeCun *et al.* (1998), o classificador *2-layer NN, 300 hidden units, mean square error* obteve uma taxa de erro de 4.7% com a base de dados de teste, considerando nenhum pré-processamento desses dados. A taxa de erro desse classificador será usada como referência comparativa do classificador baseado na rede MLP .

A rede MLP implementada conta com apenas uma camada oculta, possuindo uma arquitetura do tipo (p, q, c), sendo $p = 784$, $q = 89$ e $c = 10$. O número de neurônio da camada ocula (q) foi determinado com a Regra da Raiz Quadrada $q = \sqrt{p \times c}$. A função de ativação utilizada foi a logística.

O número de rodadas foi igual a 5. Para a etapa de treinamento da rede, considerou-se 5 épocas, na qual os dados eram embaralhados no inicio da execução de cada uma delas evitando overfitting aos dados de treino. A taxa de aprendizado utilizada foi igual a $\eta = 0.05$ e a taxa de momento igual a $m = 0.75$. A taxa de momento, ou termo de momento, é utilizado para tornar mais estável o processo de utilização dos pesos. A taxa de aprendizado e o fator de momento foram determinados empiricamente, variando seu valor até se observar um comportamento estável e de rápida redução do Erro Quadrático Médio (EQM) ao longo das épocas.

A taxa de erro médio da rede MLP implementada foi de 3,50%, indicando um ótimo desempenho desse classificador, inclusive melhor que a referência comparativa utilizada. A melhoria do desempenho da MLP de uma camada oculta em relação ao Perceptron Logístico pode ser atribuído a capacidade dessa primeira delimitar superfícies de decisão complexas e não lineares.

Uma das principais vantagens das redes MLP em relação as redes Perceptron é a capacidade resolver problemas de classificação não lineares. Todavia, há um custo computacional mais elevado. A rede MLP implementada no presente trabalho, para 5 rodadas de 5 épocas teve um tempo de treinamento e validação igual a 2007,28 segundos, enquanto a Perceptron Logístico, para 5 rodadas e 10 épocas, apresentou um tempo de treinamento e validação igual a 969,36 segundos. Levanta-se esse ponto, pois deve-se avaliar se a melhoria no desempenho justifica o custo computacional empregado.

No caso, temos cerca de um erro médio de 8,47% para a Perceptron Logística e um erro médio de 3,50% para a MLP com uma camada oculta. Nesse caso, julga-se que o custo computacional mais elevado da MLP é justificado pela melhoria no desempenho da classificação.

No tocante à referência comparativa utilizada, LeCun *et al.* (1998) destacam que obtiveram uma taxa de erro médio de 3,6% quando utilizaram distorções artificiais para gerar mais dados de treino para a *2-layer NN, 300 hidden units*, valor próximo ao obtido com a MLP de 1 camada oculta implementada. Portanto supõe-se que o melhor desempenho da MLP apresentada possa estar relacionado ao número de épocas considerado, uma vez que os dados de treino são apresentados cinco vezes de forma embaralhada. Além disso, destaca-se também os diferentes números de neurônios da camada oculta, uma vez que a quantidade de neurônios dessa(s) camada(s) pode(m) prejudicar a capacidade de generalização da rede neural quando em excesso.

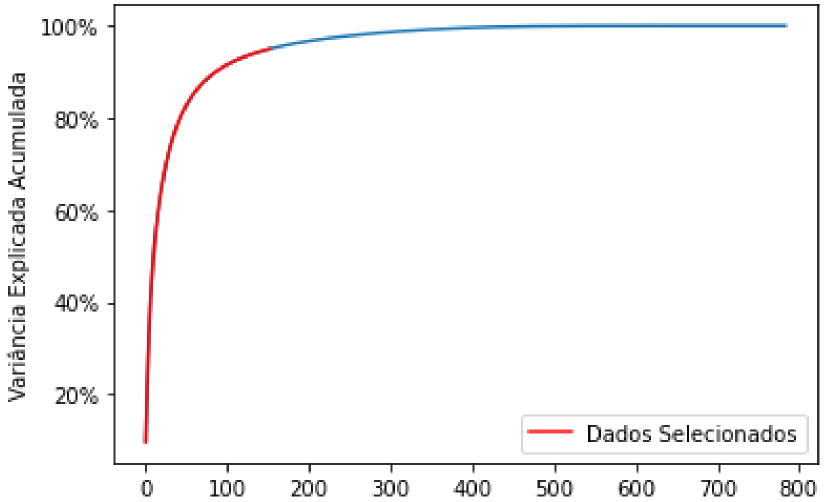
PCA + MLP

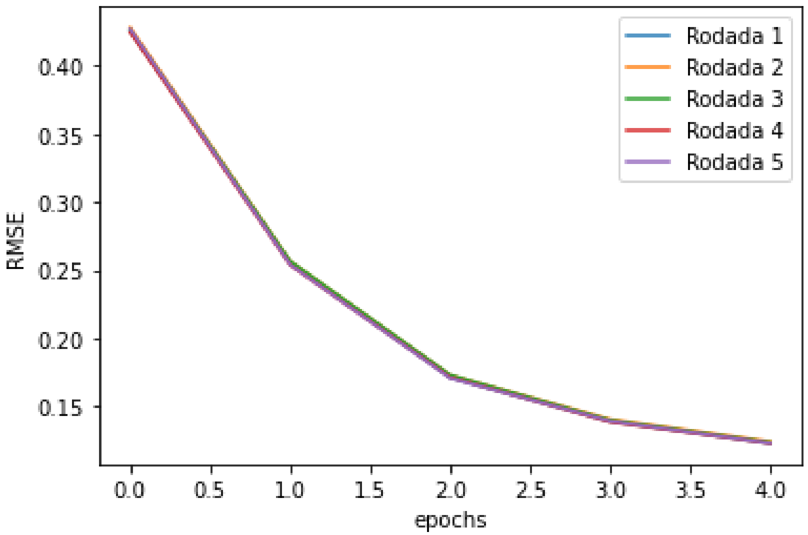
In [ ]:
```
X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

print("Dimensão da matriz de dados de treino antes do PCA: {}".format(X.shape))
print("Dimensão da matriz de dados de teste antes do PCA: {}".format(X_test.shape))
# A base de dados do MNIST é disponibilizada no seguinte padrão nxp
# n é o número de amostras, p o número de característica
# Para aplicação do PCA, deve-se transpor essa matriz para que se torne pxn

X, X_test = PCA(X.T, X_test.T, compress=True, spt_point=0.95)
print("Dimensão da matriz de dados de treino após PCA (95% da variância): {}".format(X.shape))
print("Dimensão da matriz de dados de teste após PCA (95% da variância): {}\n".format(X_test.shape))

#PS (X, Y, X_test, Y_test, eta, Nr, Ne, fun_type)
MLP(X, Y, X_test, Y_test, 0.01, 5, 5, 0.75, "log")
```

```
Dimensão da matriz de dados de treino antes do PCA: (60000, 784)
Dimensão da matriz de dados de teste antes do PCA: (10000, 784)
Dimensão da matriz de dados de treino após PCA (95% da variância): (60000, 153)
Dimensão da matriz de dados de teste após PCA (95% da variância): (10000, 153)
Rodada 1 - Taxa de acerto = 87.08%
Rodada 2 - Taxa de acerto = 87.05%
Rodada 3 - Taxa de acerto = 87.13%
Rodada 4 - Taxa de acerto = 87.12%
Rodada 5 - Taxa de acerto = 87.27%

Taxa de acerto Média = 87.13%
Taxa de erro Média = 12.87%
Melhor Taxa de Acerto = 87.27%
Pior Taxa de Acerto = 87.05%
Desv. Pad. Taxa de Acerto = 0.08%
```

Out[ ]:

```
'Tempo de Calibração e Validação para 5 rodadas = 1522.55 segundos'
```





A aplicação do PCA+MLP seguiu as mesmas espeficiações já citadas. A redução de dimensionalidade nas bases de treino e teste também mantiveram 95% da Variância Explicada. As matrizes de dados de treino e teste tiveram, respectivamente as seguintes alterações em suas dimensões: $(60.000 \times 784) \rightarrow (60.000 \times 153)$ e $(10.000 \times 784) \rightarrow (10.000 \times 153)$.

O número de rodadas, número de épocas, taxa de aprendizagem, fator de momento, número de neurônios da camada oculta e toda as outras especificidades da MLP de 1 camada implementada foram mantidas, alterando apenas os dados de entrada que passaram a ser os dados transformados e com dimensionalidade reduzida oriundos da aplicação do PCA na base de dados original.

Observa-se que o desempenho da MLP de 1 camada oculta foi bastante prejudicado, com sua taxa de erro médio indo de 3,50% (sem PCA) para 12,87% (com PCA), inclusive apresentando desemepnho inferior ao obtido com PCA + Perceptron Logístico, que apresentou uma taxa de erro médio de 11,21%. Esse comportamento sugere que os dados transformados pela aplicação do PCA são melhor classificados com apenas um hiperplano.

É interessante notar que, da mesma forma que para a aplicação da PCA + Perceptron Logístico, o uso da PCA antes da aplicação da MLP de 1 camada oculta reduziu o desvio padrão da taxa de acerto de 0,16% (Sem PCA) para 0,08% (Com PCA).

Em relação ao tempo de calibração e validação para 5 rodadas reduziu de 2007,28 segundos (Sem PCA) para 1522,55 segundos (Com PCA), uma redução de 484,73 segundos. Essa redução, de cerca de 8 minutos, ocorre em troca de uma penalização considerável do desempenho da MLP de 1 camada oculta na classificação da base de dados de teste.

Ensemble (M = 25 Perceptrons Logísticos)

In [ ]:
```
def act_fun(u, fun):
    action_fun = namedtuple("act_fun", ["f", "df"])
    if fun == "step":
        u[np.where(u >= 0)] = 1
        u[np.where(u < 0)] = 0
        du = np.nan
    elif fun == "tanh":
        u = np.array(list(map(lambda x: (1-np.exp(-x))/(1+np.exp(-x)), u)))
        du = 0.5*(1-np.power(u, 2)) + 0.05
    elif fun == "log":
        u = np.array(list(map(lambda x: 1/(1+np.exp(-x)), u)))
        du = u*(1-u) + 0.05
    return (action_fun(f=u, df=du))


def Hardamad_Prod(a, b):

    prod = np.array([np.multiply(x, y) for x, y in zip(a, b)])

    return prod


def bagging_sampler(x, y):
    baggin_index = np.random.randint(0, x.shape[0], size=x.shape[0])

    x = x[baggin_index, :]
    y = y[baggin_index, :]

    return (x, y)


def initialize_weights(shape, Number_machine):
    W = {}
    for i in range(Number_machine):
        W[i] = np.random.rand(shape[0], shape[1])
    return W
```

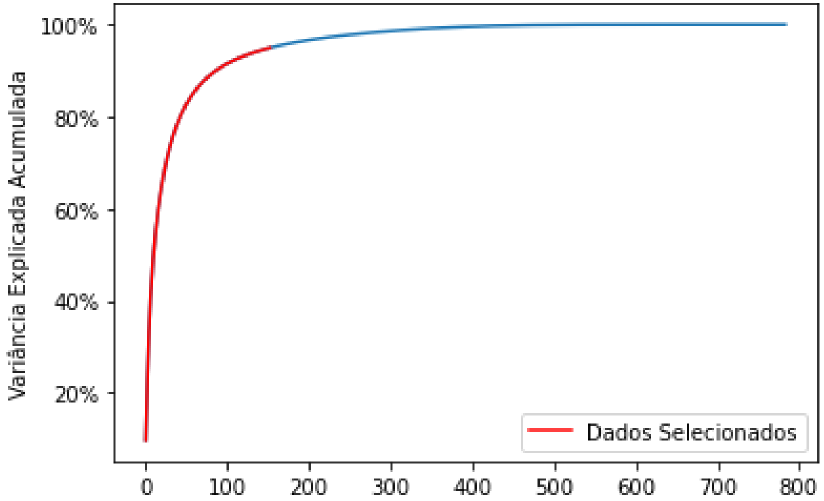
In [ ]:
```
X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)
```

```python
X = norm_data(X)
X_test = norm_data(X_test)

print("Dimensão da matriz de dados de treino antes do PCA: {}".format(X.shape))
print("Dimensão da matriz de dados de teste antes do PCA: {}".format(X_test.shape))
# A base de dados do MNIST é disponibilizada no seguinte padrão nxp
# n é o número de amostras, p o número de característica
# Para aplicação do PCA, deve-se transpor essa matriz para que se torne pxn

X, X_test = PCA(X.T, X_test.T, compress=True, spt_point=0.95)
print("Dimensão da matriz de dados de treino após PCA (95% da variância): {}".format(X.shape))
print("Dimensão da matriz de dados de teste após PCA (95% da variância): {}\n".format(X_test.shape))

eta = 0.01
mom = 0.85
Ne = 10
Nr = 5
Nm = 25  # Número de Máquinas do Ensemble
fun_type = "log"
tx_ok = np.empty((Nr))
EQM_ep = np.empty(0)

tic = time.perf_counter()
for r in range(Nr):
    # contado para taxa de acerto de cada máquina
    count_m = np.full(Nm, np.nan)
    W = initialize_weights(
        shape=(Y.shape[1], X.shape[1] + 1), Number_machine=Nm)
    W_old = W.copy()
    for m in range(Nm):
        # X_m e Y_m são as entradas do m-ésimo integrante do ensemble
        # A diversificação é feita através do Bagging, amostrador com reposição
        X_m, Y_m = bagging_sampler(X, Y)

        for e in range(Ne):
            # Embaralhamento para cada época
            rand_index = np.random.permutation(X_m.shape[0])
            X_m = X_m[rand_index, :]
            Y_m = Y_m[rand_index, :]
            EQM = 0
            for i in range(X_m.shape[0]):
                x = np.append(-1, X_m[i, :])
                U = np.dot(W[m], x)
                y, dy = act_fun(U, fun_type)
```

```
                err = Y_m[i, :] - y

                EQM = EQM + 0.5*np.power(err, 2).sum()

                Di = np.expand_dims(Hardamad_Prod(err, dy), 1)
                x = np.expand_dims(x, 1)

                # Atualiza os pesos da m-ésimo integrante do ensemble
                W_aux = W[m].copy()

                W[m] = W[m] + eta*np.dot(Di, x.T) + mom*(W[m] - W_old[m])

                W_old[m] = W_aux.copy()
        # Cálculo dos pesos para cada integrante do ensemble
        # Os integrantes do ensemble serão agregados através de uma média ponderada
        # A ponderação será dada pela taxa de acerto de cada integrante, calculada abaixo
        count = 0
        for j in range(X_test.shape[0]):
            x = np.append(-1, X_test[j, :])
            U = np.dot(W[m], x)
            y = act_fun(U, fun_type).f
            if Y_test[j, :].argmax() == y.argmax():
                count += 1

        count_m[m] = count/X_test.shape[0]

    count_ensemble = 0
    for j in range(X_test.shape[0]):
        num = 0
        for m in range(Nm):
            x = np.append(-1, X_test[j, :])
            U = np.dot(W[m], x)
            y = act_fun(U, fun_type).f
            num = num + y*count_m[m]
            # num - acumula o produto da saída por sua taxa de acerto
        y_m = num/count_m.sum()
        # y_m é a saída ponderada de cada integrante do ensemble
        if Y_test[j, :].argmax() == y_m.argmax():
            count_ensemble += 1
    print("Rodada {}, Acerto do Ensemble = {:.2%}".format(
        r+1, count_ensemble/X_test.shape[0]))

toc = time.perf_counter()
```

```
print("Tempo de Calibração e Validação para {} rodadas = {} segundos".format(
    Nr, round(toc-tic, 2)))
```

```
Dimensão da matriz de dados de treino antes do PCA: (60000, 784)
Dimensão da matriz de dados de teste antes do PCA: (10000, 784)
Dimensão da matriz de dados de treino após PCA (95% da variância): (60000, 153)
Dimensão da matriz de dados de teste após PCA (95% da variância): (10000, 153)

Rodada 1, Acerto do Ensemble = 89.10%
Rodada 2, Acerto do Ensemble = 89.02%
Rodada 3, Acerto do Ensemble = 89.07%
Rodada 4, Acerto do Ensemble = 88.95%
Rodada 5, Acerto do Ensemble = 89.06%
Tempo de Calibração e Validação para 5 rodadas = 17571.56 segundos
```



De acordo com LeCun *et al.* (1998), o classificador *committee of 25 NN 784-800-10 [elastic distortions]* obteve uma taxa de erro de 0,39% com a base de dados de teste, considerando width normalization e deslanting no pré-processamento desses dados. A taxa de erro desse classificador será usada como referência comparativa do classificador baseado no Ensemble de Perceptron Logístico.

Como pode ser observado, mesmo com aplicação da PCA para redução de dimensionalidade das bases de dados, o tempo de treinamento e validação foi de 17571,56 segundos, cerca de 293 minutos, quase 5 horas. Portanto, mesmo sem a instrução explícita de se utilizar PCA, adotou-se esse procedimento visando reduzir o tempo de treinamento e validação do ensemble. A aplicação do PCA foi identica as outras aplicações já realizadas no presente trabalho.

A etapa de treinamento de cada um dos 25 membros do ensemble utilizou o método *Bagging* (Amostrador com reposição) para diversificar os dados de treinamento de cada membro, permitindo a existência de pelo menos um padrão diferente em cada conjunto reamostrado. Após essa diversificação, os membros do ensemble foram treinados considerando o número de épocas igual a 10, uma taxa de aprendizado igual a 0,01 e um fator de momento igual a 0,85, semelhante às aplicações de Perceptron Logístico já implementadas. Ao início de cada época, os dados de treino foram embaralhados randomicamente. O número de rodadas do ensemble foi igual a 5.

Após o treinamento de cada um dos membros do ensemble, aferiu-se sua taxa de acerto para os dados de teste, guardando essa informação em um vetor. No tocante à validação do ensemble com os dados de teste, as informações oriundas de cada um dos seus membros foram agregadas utilizando uma média ponderada que adotou a taxa de acerto do membro como fator de ponderação. Dessa forma, o vetor de saída de cada um dos membros foi modulado pela sua taxa de acerto, gerando uma saída modulada para cada um dos 25 membros, a qual foi acumulada e dividida pelo somatório das taxas de acerto.

No tocante ao desempenho do ensemble de 25 Perceptrons Logísticos, para as 5 rodadas, obteve-se uma taxa de acerto média de 89,40%, uma taxa de erro média de 10,60% e um desvio padrão da taxa de acerto de 0,52%. Em relação à referência comparativa considerada, o desempenho desse ensemble está bem abaixo. Essa referência comparativa pré-processa os dados de treinamento e teste, normalizando-os e removendo a inclinação dos caracteres manuscritos (Deslanting). Apesar desse pré-processamento, supõe-se que apenas ele sozinha não justifica a tamanha diferença nos desempenhos.

O baixo desempenho do ensemble em relação à essa referência comparativa pode ser oriundo de vários fatores: má implementação do ensemble, hiperparâmetros que dificultaram o processo de aprendizado e generalização, o tipo de rede neural adotado para cada membro que não considerada uma camada oculta de neurônios, utilização do PCA, dentre outros. Destaca-se que a referência comparativa, *committee of 25 NN 784-800-10*, possui uma camada oculta com 800 neurônios, enquanto as Perceptrons Logísticas utilizadas no ensemble possuem uma única camada com 10 neurônios. Como já citado, a utilização de uma camada oculta permite que problemas não linearmente separáveis sejam solucionados através da delimitação de superfície decisória não linear.

O desempenho do ensemble em relação à uma única Perceptron Logística + PCA (PS + PCA) mostrou uma leve melhoria, reduzindo em cerca de 0,61% a taxa de erro média. Julga-se que essa melhoria não justifica o considerável aumento no tempo de treinamento e validação do ensemble, de 866 segundos (PS + PCA) para 17571,56 segundos (ensemble de 25 PS + PCA). Evidentemente, deve-se considerar uma implementação não otimizada do ensemble.

Seria interessante avaliar o desempenho do ensemble considerando outros método de agregação das saídas dos seus membros, por exemplo, considerando voto majoritário simples ao invés de média ponderada. Provavelmente não haveria melhoria substanciais no desempenho, uma vez que a agregação por média ponderada é apenas uma extensão do voto majoritário: a primeira pondera a informação de cada membro fazendo com que a contribuição do membro de melhor desempenho prevaleça sobre as demais enquanto essa última pondera igualmente a saída de cada membro, assumindo que eles possuem o mesmo desempenho.

Optou-se por não realizar o teste com agregação por voto majoritário tendo em vista o tempo necessário para treinamento e validação supracitado.

Ensemble (M = 25 MLPs)

In [ ]:
```
from sklearn.neural_network import MLPClassifier as MLP
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from glob import glob
import pickle
import os


def bagging_sampler(x, y):
    baggin_index = np.random.randint(0, x.shape[0], size=x.shape[0])

    x = x[baggin_index, :]
    y = y[baggin_index, :]

    return (x, y)


def make_ensemble(n_log, n_tanh, n_relu, q_list, eta, mom_rate):
    models = list()

    for i in range(n_log):
        n_q = np.random.choice(q_list)
        models.append((
            "{}_log_784_{}_10".format(i, n_q),
            MLP(random_state=1,
                hidden_layer_sizes=n_q,
                activation="logistic",
                learning_rate="constant",
                learning_rate_init=eta,
                momentum=mom_rate)))

    for i in range(n_tanh):
        n_q = np.random.choice(q_list)
        models.append((
            "{}_tanh_784_{}_10".format(i, n_q),
            MLP(random_state=1,
                hidden_layer_sizes=n_q,
                activation="tanh",
```

```
                learning_rate="constant",
                learning_rate_init=eta,
                momentum=mom_rate)))

    for i in range(n_relu):
        n_q = np.random.choice(q_list)
        models.append((
            "{}_relu_784_{}_10".format(i, n_q),
            MLP(random_state=1,
                hidden_layer_sizes=n_q,
                activation="relu",
                learning_rate="constant",
                learning_rate_init=eta,
                momentum=mom_rate)))

    return models


def train_models(x, y, models):

    for name, model in models:
        x_bag, y_bag = bagging_sampler(x, y)
        tic = time.perf_counter()
        model.fit(x_bag, y_bag)
        toc = time.perf_counter()
        pickle.dump(model, open("Ens_Trained_MLP/{}.pkl".format(name), "wb"))
        print("{} Treinado - Tempo = {:.2f} segundos".format(name, toc - tic))

    return "Todos os Membros Treinados"


X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)
```

Treinamento dos Membros do Ensemble

Após os modelos treinados, eles são salvos no formato "pkl" utilizando a biblioteca Pickle

In [ ]:
```
tic = time.perf_counter()
models = make_ensemble(
    n_log=10, n_tanh=10, n_relu=5,
    q_list=[70, 90, 400, 800],
    eta=0.001, mom_rate=0.9)

train_models(X, Y, models)
toc = time.perf_counter()
print("\n Tempo para treino dos {} membros do Ensemble = {:.2f} segundos".format(25, toc-tic))
```

```
0_log_784_800_10 Treinado - Tempo = 1175.26 segundos
1_log_784_90_10 Treinado - Tempo = 198.40 segundos
2_log_784_70_10 Treinado - Tempo = 206.23 segundos
3_log_784_70_10 Treinado - Tempo = 193.72 segundos
4_log_784_90_10 Treinado - Tempo = 209.91 segundos
5_log_784_400_10 Treinado - Tempo = 673.73 segundos
6_log_784_800_10 Treinado - Tempo = 1094.35 segundos
7_log_784_90_10 Treinado - Tempo = 188.04 segundos
8_log_784_800_10 Treinado - Tempo = 1046.28 segundos
9_log_784_400_10 Treinado - Tempo = 602.08 segundos
0_tanh_784_70_10 Treinado - Tempo = 130.31 segundos
1_tanh_784_400_10 Treinado - Tempo = 383.89 segundos
2_tanh_784_90_10 Treinado - Tempo = 128.25 segundos
3_tanh_784_800_10 Treinado - Tempo = 676.14 segundos
4_tanh_784_800_10 Treinado - Tempo = 648.26 segundos
5_tanh_784_400_10 Treinado - Tempo = 391.83 segundos
6_tanh_784_400_10 Treinado - Tempo = 386.07 segundos
7_tanh_784_400_10 Treinado - Tempo = 410.78 segundos
8_tanh_784_400_10 Treinado - Tempo = 375.06 segundos
9_tanh_784_400_10 Treinado - Tempo = 383.47 segundos
0_relu_784_800_10 Treinado - Tempo = 546.82 segundos
1_relu_784_800_10 Treinado - Tempo = 503.97 segundos
2_relu_784_70_10 Treinado - Tempo = 102.51 segundos
3_relu_784_800_10 Treinado - Tempo = 558.44 segundos
4_relu_784_400_10 Treinado - Tempo = 357.04 segundos

 Tempo para treino dos 25 membros do Ensemble = 11575.93 segundos
```

Determinação das predições e taxas de acerto de cada um dos membros do Ensemble

In [ ]:
```
models_path = glob("Ens_Trained_MLP/*")
models = []
predictions = []
for path in models_path:
    model = pickle.load(open(path, "rb"))
    models.append(model)
    predictions.append(model.predict(X_test))

predictions = np.array(predictions)
v_maj = np.empty(Y_test.shape)

for i in range(predictions.shape[0]):
    if i == 0:
        v_maj = predictions[i, :, :]
    else:
        v_maj = v_maj + predictions[i, :, :]

count = 0
for j in range(Y_test.shape[0]):
    if Y_test[j, :].argmax() == v_maj[j, :].argmax():
        count += 1

print(
    "Voto Majoritário Simples - Acerto do Ensemble = {:.2%}".format(count/Y_test.shape[0]))
```

```
Voto Majoritário Simples - Acerto do Ensemble = 98.27%
```

De acordo com LeCun *et al.* (1998), o classificador *committee of 25 NN 784-800-10 [elastic distortions]* obteve uma taxa de erro de 0,39% com a base de dados de teste, considerando width normalization e deslanting no pré-processamento desses dados. A taxa de erro desse classificador será usada como referência comparativa do classificador baseado no Ensemble de MLPs (25 membros).

Tendo em vista o elevado tempo de treinamento e validação do Ensemble de Perceptrons Logísticos, do item anterior, optou-se por não utlizar a função MLP implementada no presente trabalho. Ao invés dela, adotou-se a biblioteca Scikit-learn por ela ter um processamento mais otimizado.

A diversificação entre os membros do ensemble foi feita, além do *Bagging* com os dados de treinamento, com a função de ativação e o número de neurônio da camada oculta. No tocanto as funções de ativação, os 25 membros do ensemble foram divididos em: 10 logísticas, 10 tangentes hiperbólicas e 5 ReLU *(rectified linear unit)*. O número de neuronios da camada oculta de cada um desses 25 membros foi escolhido de forma aleatória dentre os seguintes valores: [70, 90, 400, 800]. Essas opções são baseados nas heurísticas de escolha do número de neurônios, considerando uma entrada com 784 elementos $p = 784$ e uma camada de saída com 10 neurônios $c = 10$. A regra de Fletcher-Gloss estipula que o número de neurônios ocultos poderia ser até 1568, optou-se por adotar 800 para igualar o número de neurônios ocultos da referência comparativa adotada.

Inicialmente, criou-se cada um dos modelos com a função MLPClassifirer do submódulo sklearn.neural_network. Para isso, informações referentes a estrutura do modelo e demais hiperparâmetros foram setados. Adotou-se uma taxa de aprendizado constante de 0.001 e um fator de momento de 0.9. O número de neurônios foi escolhido aleatoriamente entre [70, 90, 400, 800], como já citado. Como default da função, ela realiza o embaralhamento dos dados de treinamento ao longo das épocas, o que foi mantido na presente aplicação.

No treinamento de cada um dos membros do ensemble, os dados de treinamento foram reamostrados com o método *Bagging* (reamostragem com reposição). Ao final do treinamento de cada um desses modelos, eles foram salvos no formato ".pkl" através da biblioteca Pickle do Python. Esse procedimento foi realizado para não ser necessário treina-las novamente, caso fosse necessário realizar alguma alteração na etapa de generalização do modelo.

Na etapa de generalização do ensemble, as saídas (predições) de cada um dos seus membros foram agregadas com o voto majoritário simples. Nesse método de agregação, a predição do ensemble é igual ao somatório das predições de cada um dos seus membros. Ao final, a classe predita pelo ensemble é a que possuir o maior valor acumulado.

A taxa de acerto obtida para o ensemble de MLPs implementado foi de 98,27%, ou seja, uma taxa de erro igual a 1,73%. O desempenho obtido foi abaixo do desempenho da referência comparativa adotada. De forma diferente do Ensemble de Perceptrons Logísticos, assume-se que os pré-processamentos dos dados de treinamento e teste *(width normalization e deslanting)* podem ter sido fator decisivo para o melhor desempenho da referência comparativa considerada.

Em relação à MLP implementada no presente trabalho, o desempenho do ensemble teve uma melhora saindo de uma taxa de erro média de 3,50% (5 rodadas) para 1,73%, ou seja, reduziu pela metade a taxa de erro. Destaca-se que a MLP implementada adotou 89 neurônios em sua camada oculta. Além das diversificação na estrutura da rede, o ensemble realiza a reamostragem dos dados de treinamento para cada um dos membros do ensemble, permitindo que cada conjunto reamostrado tenha pelo menos um padrão que não exista em outro conjunto reamostrado.

Em relação ao tempo de treinamento e validação, a MLP de uma camada oculta com 86 neurônios levou 1522.55 segundos para 5 rodadas de treinamento e validação, enquanto o Ensemble realizou o treinamento das 25 MLPs em 11575.93 segundos. Tomando como efeito comparativo, as MLPs do ensemble com 90 neurônios na camada oculta levaram entre 120 segundos até 209 segundos, enquanto a MLP implementada levou em média 304 segundos para cada rodada de treinamento e validação, mostrando que esta última não está com seus processos tão bem otimizados, validando a opção por utilizar a biblioteca scikitlearn.

Classificador Adicinal (Extreme Learning Machine - ELM)

In [ ]:
```
def act_fun(u, fun):

    if fun == "step":
        u[np.where(u >= 0)] = 1
        u[np.where(u < 0)] = 0

    elif fun == "tanh":
        u = np.array(list(map(lambda x: (1-np.exp(-x))/(1+np.exp(-x)), u)))

    elif fun == "log":
        u = np.array(list(map(lambda x: 1/(1+np.exp(-x)), u)))

    return u


X, Y = MNIST_data.get_train_data()
X_test, Y_test = MNIST_data.get_test_data()

Y = one_hot_enconding(Y, 10)
Y_test = one_hot_enconding(Y_test, 10)

X = norm_data(X)
X_test = norm_data(X_test)

print("Dimensão da matriz de dados de treino antes do PCA: {}".format(X.shape))
print("Dimensão da matriz de dados de teste antes do PCA: {}".format(X_test.shape))

# Transpondo para aproveitar uma implementação desenvolvida durante as aulas da disciplina
X, Y = (X.T, Y.T)
X_test, Y_test = (X_test.T, Y_test.T)

fun_type = "log"

Nr = 5
q = 86
tx_ok = np.empty(Nr)
W = np.random.normal(loc=0, scale=0.1, size=(q, X.shape[0]+1))

tic = time.perf_counter()
for r in range(Nr):
```

```python
    rand_index = np.random.permutation(X.shape[1])

    X = X[:, rand_index]
    Y = Y[:, rand_index]

    Z = []

    for i in range(X.shape[1]):
        x = np.append(-1, X[:, i])
        U1 = np.dot(W, x)
        z = act_fun(U1, fun_type)
        z = np.append(-1, z)
        Z.append(z)

    Z = np.array(Z)

    tic2 = time.perf_counter()

    M = np.dot(Y, np.linalg.pinv(Z.T))

    count = 0

    for j in range(X_test.shape[1]):
        x = np.append(-1, X_test[:, j])
        U1 = np.dot(W, x)
        z = act_fun(U1, fun_type)
        z = np.append(-1, z)
        y = np.dot(M, z)

        if Y_test[:, j].argmax() == y.argmax():
            count += 1
    tx_ok[r] = count/X_test.shape[1]
    print("Rodada {}, Taxa de acerto {:.2%}".format(
        r+1, count/X_test.shape[1]))
toc = time.perf_counter()

print("\nTaxa de acerto Média = {:.2%}".format(tx_ok.mean()))
print("Taxa de erro Média = {:.2%}".format(1-tx_ok.mean()))
print("Melhor Taxa de Acerto = {:.2%}".format(tx_ok.max()))
print("Pior Taxa de Acerto = {:.2%}".format(tx_ok.min()))
print("Desv. Pad. Taxa de Acerto = {:.2%}".format(tx_ok.std()))

print("\nTempo de Calibração e Validação para {} rodadas = {} segundos".format(
    Nr, round(toc-tic, 2)))
```

```
Dimensão da matriz de dados de treino antes do PCA: (60000, 784)
Dimensão da matriz de dados de teste antes do PCA: (10000, 784)
Rodada 1, Taxa de acerto 83.67%
Rodada 2, Taxa de acerto 83.67%
Rodada 3, Taxa de acerto 83.67%
Rodada 4, Taxa de acerto 83.67%
Rodada 5, Taxa de acerto 83.67%

Taxa de acerto Média = 83.67%
Taxa de erro Média = 16.33%
Melhor Taxa de Acerto = 83.67%
Pior Taxa de Acerto = 83.67%
Desv. Pad. Taxa de Acerto = 0.00%

Tempo de Calibração e Validação para 5 rodadas = 122.74 segundos
```

De maneira adicional ao pedido na Questão 01 do Tabalho 01, implementou-se um classificador baseado em Extreme Learning Machine - ELM. Discutindo rapidamente sobre os resultados obtidos, o ELM teve uma taxa de acerto de 83,67% e foi executado em 122,74 segundos (treinamento + validação para 5 rodadas).

A ELM apresentou um pior desempenho em relação aos classificadores baseados em Perceptrons Logísticos e MLPs de uma camada oculta, todavia, é digno de nota o tempo de treinamento e validação considervalmente inferior da ELM. Em relação ao classificador de mínimos quadrados, pode-se considerar que os desempenhos foram semelhantes.

Uma das principais vantagens da ELM, além do seu rápido treinamento, é a possibilidade de solução de problemas não linearmente separáveis, uma vez que ela tem a mesma estrutura de uma MLP com uma camada oculta, possibilitando a delimitação de fronteiras de decisão não lineares. Todavia, para esse seu rápido treinamento, sacrifica-se o seu desempenho.

Externando uma opinião pessoal, julgo que a ELM está para MLP assim como o classificador de mínimos quadrados está para a Perceptron Simples.

Referências

LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. "Gradient-based learning applied to document recognition." Proceedings of the IEEE, 86(11):2278-2324, November 1998.

Trabalho Final - Inteligência Computacional Aplicada (TIP7077)

Aluno: Carlos Eduardo Sousa Lima

Prof. Guilherme de Alencar Barreto

Questão 02 - Regressão e Ajuste de Curvas - *Real estate valuation dataset*

The data set was randomly split into the training data set (2/3 samples) and the testing data set (1/3 samples). (https://archive.ics.uci.edu/dataset/477/real+estate+valuation+data+set)

In [ ]:
```
#Bibliotecas Utilizadas
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

Regressão Linear Múltipla de Mínimos Quadrados

In [ ]:
```
def R2(y_true, y_pred):

    coef = 1 - (np.power(y_true-y_pred,2).sum()/np.power(y_true-y_true.mean(),2).sum())

    return coef

def RMSE(y_true, y_pred):

    coef = np.power((np.power(y_true - y_pred, 2).sum())/y_true.shape[0], 0.5)

    return coef

data = pd.read_excel("Real estate valuation data set.xlsx", index_col = 0)

X = data.iloc[:, 0:6].to_numpy()

Y = data.iloc[:, -1].to_numpy()
```

```python
RMSE_r, tx_ok_20, tx_ok_10, R2_r = [], [], [], []

best_dict = {
    "tx_ok_10":0,
    "tx_ok_20":0,
    "y_pred": 0,
    "y_true": 0,
    "RMSE": 0,
    "R2": 0,
    "R2_aj": 0
}

Nr = 20
for i in range(Nr):
    rand_index = np.random.permutation(X.shape[0])
    X = X[rand_index,:]
    Y = Y[rand_index]

    spt_point = int(2/3*X.shape[0])

    X_train = X[:spt_point,:]
    Y_train = Y[:spt_point]

    X_test = X[spt_point:,:]
    Y_test = Y[spt_point:]

    X_train = np.c_[np.ones(X_train.shape[0]), X_train]
    X_test = np.c_[np.ones(X_test.shape[0]), X_test]
    M = M = np.linalg.lstsq(X_train, Y_train, rcond = -1)[0]

    Y_pred = np.dot(X_test, M)

    count_10 = 0
    count_20 = 0
    for i in range(Y_test.shape[0]):

        if np.abs(Y_test[i] - Y_pred[i]) <= 0.2*np.abs(Y_test[i]):
            count_20 += 1
        if np.abs(Y_test[i] - Y_pred[i]) <= 0.1*np.abs(Y_test[i]):
            count_10 += 1
```

```python
    RMSE_r.append(RMSE(Y_test, Y_pred))
    R2_r.append(R2(Y_test, Y_pred))
    tx_ok_10.append(count_10/Y_test.shape[0])
    tx_ok_20.append(count_20/Y_test.shape[0])

    #Para ser a melhor rodada, deve ser melhor nas duas taxas de acerto
    if (best_dict["tx_ok_10"] <= count_10/X_test.shape[0]) & (best_dict["tx_ok_20"] <= count_20/X_test.shape[0]):
        best_dict["y_pred"] = Y_pred
        best_dict["y_true"] = Y_test
        best_dict["R2"] = R2(Y_test, Y_pred)
        best_dict["RMSE"] = RMSE(Y_test, Y_pred)
        best_dict["tx_ok_10"] = count_10/X_test.shape[0]
        best_dict["tx_ok_20"] = count_20/X_test.shape[0]

print("###### Resumo das {} Rodadas ######".format(Nr))
print('''Taxa de acerto para uma taxa de erro admssível de 20%:
Média = {:.2%},
Desv. Pad. = {:.2%},
Máximo = {:.2%},
Mínimo = {:.2%}\n'''.format(np.mean(tx_ok_20),
    np.std(tx_ok_20), np.max(tx_ok_20), np.min(tx_ok_20)))

print('''Taxa de acerto para uma taxa de erro admssível de 10%:
Média = {:.2%},
Desv. Pad. = {:.2%},
Máximo = {:.2%},
Mínimo = {:.2%}\n'''.format(np.mean(tx_ok_10),
    np.std(tx_ok_10), np.max(tx_ok_10), np.min(tx_ok_10)))

print('''R2:
Média = {:.3f},
Desv. Pad. = {:.3f},
Máximo = {:.3f},
Mínimo = {:.3f}\n'''.format(np.mean(R2_r),
    np.std(R2_r), np.max(R2_r), np.min(R2_r)))

print('''RMSE:
Média = {:.3f},
Desv. Pad. = {:.3f},
Máximo = {:.3f},
Mínimo = {:.3f}'''.format(np.mean(RMSE_r),
    np.std(RMSE_r), np.max(RMSE_r), np.min(RMSE_r)))
```

```python
print("\n###### Melhor Rodada em relação as taxas de acerto para erros admissíveis de 10% e 20% ######")
print("Taxa de acerto para uma taxa de erro admssível de 20% = {:.2%}".format(best_dict["tx_ok_20"]))
print("Taxa de acerto para uma taxa de erro admssível de 10% = {:.2%}".format(best_dict["tx_ok_10"]))
print("R2 = {:.3f}".format(best_dict["R2"]))
print("RMSE = {:.3f}".format(best_dict["RMSE"]))
```

```
###### Resumo das 20 Rodadas ######
Taxa de acerto para uma taxa de erro admssível de 20%:
Média = 68.62%,
Desv. Pad. = 4.44%,
Máximo = 78.99%,
Mínimo = 60.87%

Taxa de acerto para uma taxa de erro admssível de 10%:
Média = 39.02%,
Desv. Pad. = 3.31%,
Máximo = 46.38%,
Mínimo = 35.51%

R2:
Média = 0.561,
Desv. Pad. = 0.067,
Máximo = 0.663,
Mínimo = 0.428

RMSE:
Média = 9.044,
Desv. Pad. = 1.110,
Máximo = 11.068,
Mínimo = 7.447

###### Melhor Rodada em relação as taxas de acerto para erros admissíveis de 10% e 20% ######
Taxa de acerto para uma taxa de erro admssível de 20% = 78.99%
Taxa de acerto para uma taxa de erro admssível de 10% = 42.75%
R2 = 0.630
RMSE = 8.032
```

Yeh e Hsu (2018) apresentam o resultado da análise de regressão multivariada na Tabela 6 de seu artigo. Os resultados são apresentados de acordo com os círculos de abastecimento e demanda apresentados no Trabalho, são eles: Sindia, Danshuei, Wunshan e Beitou. Os resultados também são apresentados em termos da média obtida para cada um desses círculos de abastecimento e demanda.

A base de dados disponibilizada refere-se as informações da círculo de abastecimento e demanda do Sindia, portanto os resultados obtidos nesse trabalho serão comparados aos resultados obtidos para essa região. Os autores destacam que a base de dados total foi divida aleatoriamente em 2/3 para treinamento e 1/3 para teste. As métricas apresentadas pelo autores correspondem a avaliação das predições para os dados de teste.

Tendo em vista essa divisão aleatória realizada pelos autores, decidiu-se rodar o modelo de regressão linear múltipla 20 vezes, gerando uma estatística para essas 4 métricas. Para cada uma dessas rodadas, embaralhou-se a base de dados e realizou-se a separação entre dados de treino e dados de teste, segundo as proporções especificadas pelos autores.

Foram comparadas 4 métricas: i) Taxa de acerto para um erro admissível de 20%, ii) Taxa de acerto para um erro admissível de 10%, iii) R^2 e iv) RMSE. A tabela abaixo apresenta as métricas para a rodada da regressão linear múltipla com melhor taxa de acerto para um erro tolerável de 10% e 20% e os valores obtidos por Yeh e Hsu (2018).

Métrica	Regressão Linear Múltipla	Yeh e Hsu (2018)
i)	78,99%	78,00%
ii)	42,75%	48,80%
iii)	0,630	0,627
iv)	8,032	8,06

Observa-se que o modelo implementado se aproximou dos resultados obtidos por Yeh e Hsu (2018). Exceto pela métrica ii), a a diferença no desempenho do modelo de Regressão Linear Múltipla e os resultados apresentado pelos referidos autores foram insignificantes. O desempenho do modelo implemetando para a métrica ii), por sua vez, ficou abaixo dos obtidos por Yeh e Hsu (2018) em cerca de 6%.

Destaca-se que o desempenho superior ou inferior do modelo implementado em relação aos resultados de Yeh e Hsu (2018) é bastante influenciado pela divisão aleatória da base de dados, uma vez que haverá uma possível combinação de dados de treino e de teste que otimize ou reduza essas métricas.

Extreme Learning Machine

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```python
import pandas as pd
import time

def act_fun(u, fun):

    if fun == "step":
        u[np.where(u >= 0)] = 1
        u[np.where(u < 0)] = 0

    elif fun == "tanh":
        u = np.array(list(map(lambda x: (1-np.exp(-x))/(1+np.exp(-x)), u)))

    elif fun == "log":
        u = np.array(list(map(lambda x: 1/(1+np.exp(-x)), u)))

    return u

def norm_data(x):

    for i in range(x.shape[1]):
        x[:,i] = x[:,i]/x[:,i].max()

    return x

def R2(y_true, y_pred):

    coef = 1 - (np.power(y_true-y_pred,2).sum()/np.power(y_true-y_true.mean(),2).sum())

    return coef

def RMSE_calc(y_true, y_pred):

    coef = np.power((np.power(y_true - y_pred, 2).sum())/y_true.shape[0], 0.5)

    return coef

data = pd.read_excel("Real estate valuation data set.xlsx", index_col = 0)

X = data.iloc[:, 0:6].to_numpy()

Y = data.iloc[:, -1].to_numpy()
```

```python
X = norm_data(X)
Y = norm_data(np.expand_dims(Y,1))

fun_type = "log"
Nr = 10
q = 5
RMSE_r = []
R2_r = []
R2aj_r = []
tx_ok_20 = []
tx_ok_10 = []

best_dict = {
    "tx_ok_10":0,
    "tx_ok_20":0,
    "y_pred": 0,
    "y_true": 0,
    "RMSE": 0,
    "R2": 0,
    "R2_aj": 0
}
tic = time.perf_counter()
for r in range(Nr):
    rand_index = np.random.permutation(X.shape[0])

    X = X[rand_index,:]
    Y = Y[rand_index]

    spt_point = int(2/3*X.shape[0])
    X_train = X[:spt_point,:]
    Y_train = Y[:spt_point]

    X_test = X[spt_point:,:]
    Y_test = Y[spt_point:]

    W = np.random.normal(loc = 0, scale = 0.1, size = (q, X.shape[1]+1))

    Z = []

    for i in range(X_train.shape[0]):
        x = np.append(-1, X_train[i,:])
        U = np.dot(W, x)
```

```
        z = act_fun(U, fun_type)
        z = np.append(-1, z)
        Z.append(z)
    Z = np.array(Z)
    M = np.dot(np.linalg.pinv(Z), Y_train)

    count_20 = 0
    count_10 = 0
    y_pred = []
    RMSE = 0
    for j in range(X_test.shape[0]):
        x = np.append(-1, X_test[j,:])
        U1 = np.dot(W, x)
        z = act_fun(U1, fun_type)
        z = np.append(-1, z)
        y = np.dot(z, M)
        y_pred.append(y)
        RMSE = RMSE + RMSE_calc(Y_test[j], y)
        if np.abs(Y_test[j] - y) <= 0.2*np.abs(Y_test[j]):
            count_20 += 1
        if np.abs(Y_test[j] - y) <= 0.1*np.abs(Y_test[j]):
            count_10 += 1
    y_pred = np.array(y_pred)

    RMSE_r.append(RMSE)
    R2_r.append(R2(Y_test, y_pred))
    tx_ok_20.append(count_20/X_test.shape[0])
    tx_ok_10.append(count_10/X_test.shape[0])

    #Para ser a melhor rodada, deve ser melhor nas duas taxas de acerto
    if (best_dict["tx_ok_10"] <= count_10/X_test.shape[0]) & (best_dict["tx_ok_20"] <= count_20/X_test.shape[0]):
        best_dict["y_pred"] = y_pred
        best_dict["y_true"] = Y_test
        best_dict["R2"] = R2(Y_test, y_pred)
        best_dict["RMSE"] = RMSE
        best_dict["tx_ok_10"] = count_10/X_test.shape[0]
        best_dict["tx_ok_20"] = count_20/X_test.shape[0]

toc = time.perf_counter()

print("\n###### Resumo das {} Rodadas ######".format(Nr))
print('''Taxa de acerto para uma taxa de erro admssível de 20%:
```

```
Média = {:.2%},
Desv. Pad. = {:.2%},
Máximo = {:.2%},
Mínimo = {:.2%}\n'''.format(np.mean(tx_ok_20),np.std(tx_ok_20), np.max(tx_ok_20), np.min(tx_ok_20)))
print('''Taxa de acerto para uma taxa de erro admssível de 10%:
Média = {:.2%},
Desv. Pad. = {:.2%},
Máximo = {:.2%},
Mínimo = {:.2%}\n'''.format(np.mean(tx_ok_10), np.std(tx_ok_10), np.max(tx_ok_10), np.min(tx_ok_10)))
print('''R2:
Média = {:.3f},
Desv. Pad. = {:.3f},
Máximo = {:.3f},
Mínimo = {:.3f}\n'''.format(np.mean(R2_r), np.std(R2_r), np.max(R2_r), np.min(R2_r)))
print('''RMSE:
Média = {:.3f},
Desv. Pad. = {:.3f},
Máximo = {:.3f},
Mínimo = {:.3f}'''.format(np.mean(RMSE_r), np.std(RMSE_r), np.max(RMSE_r), np.min(RMSE_r)))

print("\n###### Melhor Rodada em relação as taxas de acerto para erros de 10% e 20% ######")
print("Taxa de acerto para uma taxa de erro admssível de 20% = {:.2%}".format(best_dict["tx_ok_20"]))
print("Taxa de acerto para uma taxa de erro admssível de 10% = {:.2%}".format(best_dict["tx_ok_10"]))
print("R2 = {:.3f}".format(best_dict["R2"]))
print("RMSE = {:.3f}".format(best_dict["RMSE"]))

print("\nTempo de Calibração e Validação = {:.3f} segundos, Rodadas = {}".format(toc - tic, Nr))
```

```
###### Resumo das 10 Rodadas ######
Taxa de acerto para uma taxa de erro admssível de 20%:
Média = 71.30%,
Desv. Pad. = 5.88%,
Máximo = 82.61%,
Mínimo = 62.32%

Taxa de acerto para uma taxa de erro admssível de 10%:
Média = 42.90%,
Desv. Pad. = 6.48%,
Máximo = 52.90%,
Mínimo = 32.61%

R2:
Média = 0.612,
Desv. Pad. = 0.080,
Máximo = 0.763,
Mínimo = 0.434

RMSE:
Média = 6.944,
Desv. Pad. = 0.743,
Máximo = 7.943,
Mínimo = 5.426

###### Melhor Rodada em relação as taxas de acerto para erros de 10% e 20% ######
Taxa de acerto para uma taxa de erro admssível de 20% = 82.61%
Taxa de acerto para uma taxa de erro admssível de 10% = 52.90%
R2 = 0.763
RMSE = 5.426

Tempo de Calibração e Validação = 0.271 segundos, Rodadas = 10
```

Yeh e Hsu (2018) apresentam o resultado da predição dos modelos de redes neurais na Tabela 7 de seu artigo. Os resultados são apresentados de acordo com os círculos de abastecimento e demanda apresentados no Trabalho, são eles: Sindia, Danshuei, Wunshan e Beitou. Os resultados também são apresentados em termos da média obtida para cada um desses círculos de abastecimento e demanda.

A base de dados disponibilizada refere-se as informações da círculo de abastecimento e demanda do Sindia, portanto os resultados obtidos nesse trabalho serão comparados aos resultados obtidos para essa região. Yeh e Hsu (2018) destacam que a base de dados total foi divida aleatoriamente em 2/3 para treinamento e 1/3 para teste. As métricas apresentadas pelo autores correspondem a avaliação das predições para os dados de teste.

A ELM implementada possui a seguinte arquitetura (6, 5, 1). Adotou o número de neurônio ocultos $(q = 5)$ igual ao especificado por Yeh e Hsu (2018). A ELM foi rodada 10 vezes e, para cada uma dessas rodadas, embaralhou-se a base de dados e realizou-se a separação entre dados de treino e dados de teste, segundo as proporções especificadas pelos autores. Cabe destacar que esses dados de treino e teste foram previamente normalizados, trazendo todos os valores para uma escala comum sem distorcer as diferenças nos intervalos de valores. Como foi adotado a função de ativação logística, essa nova escala comum adotada foi $x_{norm} \in [0, 1]$

Foram comparadas 4 métricas: i) Taxa de acerto para um erro admissível de 20%, ii) Taxa de acerto para um erro admissível de 10%, iii) R^2 e iv) RMSE. A tabela abaixo apresenta as métricas para a rodada da ELM com melhor taxa de acerto para um erro tolerável de 10% e 20% e os valores obtidos por Yeh e Hsu (2018).

Métrica	Extreme Learning Machine	Yeh e Hsu (2018)
i)	82,61%	78,00%
ii)	52,90%	48,80%
iii)	0,763	0,627
iv)	5,426	8,06

Observa-se que a rede ELM implementada apresentou um desempenho superior à MLP implementada pelos autores para todas as métricas avaliadas. Esse melhor desempenho pode ser oriundo da aleatoriedade implementada na separação da base de treino e teste, existindo uma combinação de dados de treino e de teste que otimize essas métricas. Além do melhor desempenho, destaca-se a velocidade de treinamento e validação do algoritmo implementado, cerca de 0,271 segundos.

MLP

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from collections import namedtuple
import time
```

```python
def act_fun(u, fun):
    action_fun = namedtuple("act_fun", ["f", "df"])
    if fun == "step":
        u[np.where(u >= 0)] = 1
        u[np.where(u < 0)] = 0
        du = np.nan
    elif fun == "tanh":
        u = np.array(list(map(lambda x: (1-np.exp(-x))/(1+np.exp(-x)), u)))
        du = 0.5*(1-np.power(u,2)) + 0.05
    elif fun == "log":
        u = np.array(list(map(lambda x: 1/(1+np.exp(-x)), u)), dtype = "float")
        du = u*(1-u)
    return (action_fun(f = u, df = du))

def norm_data(x):

    for i in range(x.shape[1]):
        x[:,i] = x[:,i]/x[:,i].max()
        # x[:,i] = 2*((x[:,i] - x[:,i].min())/(x[:,i].max() - x[:,i].min())) - 1
        #x_norm = 2*((x - x.min())/(x.max() - x.min()) - 1
    return x

def R2(y_true, y_pred):

    coef = 1 - (np.power(y_true-y_pred,2).sum()/np.power(y_true-y_true.mean(),2).sum())

    return (coef)

def RMSE_calc(y_true, y_pred):

    coef = np.power((np.power(y_true - y_pred, 2).sum())/y_true.shape[0], 0.5)

    return coef

def Hardamad_Prod(a, b):

    prod = np.array([np.multiply(x,y) for x, y in zip(a,b)])

    return prod

data = pd.read_excel("Real estate valuation data set.xlsx", index_col = 0)
```

```
X_real = data.iloc[:, 0:6].to_numpy()

Y_real = data.iloc[:, -1].to_numpy()

X = norm_data(X_real)
Y = norm_data(np.expand_dims(Y_real,1))

fun_type = "log"
Nr = 10
q = 5
eta = 0.6
Ne = 1500
mon = 0.9

RMSE_r = []
R2_r = []
tx_ok_20 = []
tx_ok_10 = []

best_dict = {
    "tx_ok_10":0,
    "tx_ok_20":0,
    "y_pred": 0,
    "y_true": 0,
    "RMSE": 0,
    "R2": 0,
    "R2_aj": 0
}

tic = time.perf_counter()

for r in range(Nr):
    rand_index = np.random.permutation(X.shape[0])

    X = X[rand_index,:]
    Y = Y[rand_index]

    spt_point = int(2/3*X.shape[0])
    X_train = X[:spt_point,:]
    Y_train = Y[:spt_point,:]

    X_test = X[spt_point:,:]
```

```
    Y_test = Y[spt_point:,:]

    W = {
        0: np.random.rand(q, X_train.shape[1]+1)*0.01,
        1: np.random.rand(Y_train.shape[1], q+1)*0.01
    }
    W_old = W.copy()

    for e in range(Ne):
        rand_index = np.random.permutation(X_train.shape[0])
        X_train = X_train[rand_index,:]
        Y_train = Y_train[rand_index,:]


        for i in range(X_train.shape[0]):
            x = np.append(-1, X_train[i,:])
            U1 = np.dot(W[0], x)
            z, dz = act_fun(U1, fun_type)
            z = np.append(-1, z)
            U2 = np.dot(W[1], z)
            y, dy = act_fun(U2, fun_type)

            err = Y_train[i,:] - y

            err = np.expand_dims(Hardamad_Prod(err, dy), 1)
            x = np.expand_dims(x,1)
            z = np.expand_dims(z,1)

            W_aux = W.copy()

            DDi = Hardamad_Prod(dz, np.dot(W[1][:,1:].T, err))

            W[0] = W[0] + eta*np.dot(DDi, x.T) + mon*(W[0] - W_old[0])
            W[1] = W[1] + eta*np.dot(err, z.T) + mon*(W[1] - W_old[1])

            W_old = W_aux.copy()

    RMSE_test = 0
    count_20 = 0
    count_10 = 0
    y_pred = []
    for j in range(X_test.shape[0]):
```

```python
        x = np.append(-1, X_test[j,:])
        U1 = np.dot(W[0],x)
        z = act_fun(U1, fun_type).f
        z = np.append(-1, z)
        U2 = np.dot(W[1], z)
        y = act_fun(U2, fun_type).f
        y_pred.append(y)
        err = Y_test[j,:] - y

        RMSE_test = RMSE_test + RMSE_calc(Y_test[j,:], y)

        if np.abs(err) <= 0.2*np.abs(Y_test[j,:]):
            count_20 += 1
        if np.abs(err) <= 0.1*np.abs(Y_test[j,:]):
            count_10 += 1

    RMSE_r.append(RMSE_test)
    R2_r.append(R2(Y_test, y_pred))
    tx_ok_20.append(count_20/X_test.shape[0])
    tx_ok_10.append(count_10/X_test.shape[0])

    #Para ser a melhor rodada, deve ser melhor nas duas taxas de acerto
    if (best_dict["tx_ok_10"] <= count_10/X_test.shape[0]) & (best_dict["tx_ok_20"] <= count_20/X_test.shape[0]):
        best_dict["y_pred"] = y_pred
        best_dict["y_true"] = Y_test
        best_dict["R2"] = R2(Y_test, y_pred)
        best_dict["RMSE"] = RMSE_test
        best_dict["tx_ok_10"] = count_10/X_test.shape[0]
        best_dict["tx_ok_20"] = count_20/X_test.shape[0]

toc = time.perf_counter()

print("\n###### Resumo das {} Rodadas ######".format(Nr))
print('''Taxa de acerto para uma taxa de erro admssível de 20%:
Média = {:.2%},
Desv. Pad. = {:.2%},
Máximo = {:.2%},
Mínimo = {:.2%}\n'''.format(np.mean(tx_ok_20),np.std(tx_ok_20), np.max(tx_ok_20), np.min(tx_ok_20)))
print('''Taxa de acerto para uma taxa de erro admssível de 10%:
Média = {:.2%},
Desv. Pad. = {:.2%},
Máximo = {:.2%},
```

```
Mínimo = {:.2%}\n'''.format(np.mean(tx_ok_10), np.std(tx_ok_10), np.max(tx_ok_10), np.min(tx_ok_10)))
print('''R2:
Média = {:.3f},
Desv. Pad. = {:.3f},
Máximo = {:.3f},
Mínimo = {:.3f}\n'''.format(np.mean(R2_r), np.std(R2_r), np.max(R2_r), np.min(R2_r)))
print('''RMSE:
Média = {:.3f},
Desv. Pad. = {:.3f},
Máximo = {:.3f},
Mínimo = {:.3f}'''.format(np.mean(RMSE_r), np.std(RMSE_r), np.max(RMSE_r), np.min(RMSE_r)))

print("\n###### Melhor Rodada em relação as taxas de acerto para erros admissíveis de 10% e 20% ######")
print("Taxa de acerto para uma taxa de erro admssível de 20% = {:.2%}".format(best_dict["tx_ok_20"]))
print("Taxa de acerto para uma taxa de erro admssível de 10% = {:.2%}".format(best_dict["tx_ok_10"]))
print("R2 = {:.3f}".format(best_dict["R2"]))
print("RMSE = {:.3f}".format(best_dict["RMSE"]))

print("\nTempo de Calibração e Validação = {:.3f} segundos, Rodadas = {}, Épocas = {}".format(toc - tic, Nr, Ne))
```

```
###### Resumo das 10 Rodadas ######
Taxa de acerto para uma taxa de erro admssível de 20%:
Média = 67.68%,
Desv. Pad. = 11.39%,
Máximo = 80.43%,
Mínimo = 38.41%

Taxa de acerto para uma taxa de erro admssível de 10%:
Média = 38.70%,
Desv. Pad. = 7.96%,
Máximo = 48.55%,
Mínimo = 18.12%

R2:
Média = 0.492,
Desv. Pad. = 0.204,
Máximo = 0.686,
Mínimo = -0.034

RMSE:
Média = 7.771,
Desv. Pad. = 1.452,
Máximo = 11.351,
Mínimo = 6.277

###### Melhor Rodada em relação as taxas de acerto para erros admissíveis de 10% e 20% ######
Taxa de acerto para uma taxa de erro admssível de 20% = 80.43%
Taxa de acerto para uma taxa de erro admssível de 10% = 44.20%
R2 = 0.614
RMSE = 6.281

Tempo de Calibração e Validação = 1675.135 segundos, Rodadas = 10, Épocas = 1500
```

Yeh e Hsu (2018) apresentam o resultado da predição dos modelos de redes neurais na Tabela 7 de seu artigo. Os resultados são apresentados de acordo com os círculos de abastecimento e demanda apresentados no Trabalho, são eles: Sindia, Danshuei, Wunshan e Beitou. Os resultados também são apresentados em termos da média obtida para cada um desses círculos de abastecimento e demanda.

A base de dados disponibilizadas refere-se as informações da círculo de abastecimento e demanda do Sindia, portanto os resultados obtidos nesse trabalho serão comparados aos resultados obtidos para essa região. Yeh e Hsu (2018) destacam que a base de dados total foi divida aleatoriamente em 2/3 para treinamento e 1/3 para teste. As métricas apresentadas pelo autores correspondem a avaliação das predições para os dados de teste.

A MLP implementada possui uma a seguinte arquitetura (6, 5, 1). Adotou-se o número de neurônio ocultos $(q = 5)$, taxa de aprendizado $(\eta = 0,6)$ e o número de épocas $(Ne = 1500)$ iguais ao especificado por Yeh e Hsu (2018). Yeh e Hsu (2018) não destacam a utilização do fator de momento, todavia optou-se pela sua utilização na MLP implementada no presente projeto, visando uma modificação mais estável dos pesos sinápticos ao longo do treinamento da rede. O fator de momento considerado foi igual a $0,9$

Tendo em vista a divisão aleatória da base de dados total em dados de treino e teste realizada por Yeh e Hsu (2018), a MLP implementada foi rodada 10 vezes e, para cada uma dessas rodadas, embaralhou-se a base de dados total e realizou-se a separação entre dados de treino e dados de teste, segundo as proporções especificadas pelos autores.

Da mesma forma que para a ELM, a base de dados total foi previamente normalizada antes do treinamento e validação da MLP, trazendo todos os valores para uma escala comum sem distorcer as diferenças nos intervalos de valores. Como foi adotado a função de ativação logística, essa nova escala comum adotada foi $x_{norm} \in [0,1]$. Busca-se, com esse processo, melhorar a estabilidade e convergência do processo de aprendizado.

Foram comparadas 4 métricas: i) Taxa de acerto para um erro admissível de 20%, ii) Taxa de acerto para um erro admissível de 10%, iii) R^2 e iv) RMSE. A tabela abaixo apresenta as métricas para a rodada da MLP com melhor taxa de acerto para um erro de 10% e 20% e os valores obtidos por Yeh e Hsu (2018).

Métrica	MLP (6,5,1)	Yeh e Hsu (2018)
i)	80,43%	78,00%
ii)	44,20%	48,80%
iii)	0,614	0,627
iv)	6,281	8,06

Observa-se que a MLP implementada teve resultados próximos aos obtidos por Yeh e Hsu (2018). A MLP implementada teve um desempenho um pouco melhor para as métricas i) e iv), ficando atrás nas métricas ii) e iii). Apesar da arquitetura semelhante da MLP implementada no presente trabalho e a implementada por Yeh e Hsu (2018), destaca-se que a divisão aleatória da base de dados pode gerar combinações de dados de treino e de teste capazes de dificultar ou melhorar o treinamento e generalização da rede neural. Como pode ser observado na estatística das métricas adotadas para as 10 rodadas, o R^2 teve um valor mínimo de -0,034 e um valor máximo de 0,614, evidenciando a influência dessa divisão aleatória da base de dados no desempenho da rede neural.

Comparando a MLP implementada com a ELM implementada no item anterior, pode-se constatar que a ELM teve um melhor desempenho em todas as métricas avaliadas, todavia, a influência da divisão aleatória da base de dados também é válida para essa comparação. Além dessas métricas, observa-se um tempo de treinamento e validação consideravelmente menor para ELM (0,271 segundos) em relação a MLP (1675,135 segundos), o que torna a ELM bem mais atrativa para solução do problema proposto.

Referências

Yeh, I-C.; Hsu, T-K. Building real estate valuation models with comparative approachthrough case-based reasoning, **Applied Soft Computing**, 65, 2018, p. 260-271, doi: 10.1016/j.asoc.2018.01.029
