Capítulo 4

Redes Neuronais Artificiais

A ideia de redes neuronais artificiais foi desenvolvida à semelhança das redes neuronais do cérebro humano. Estudando cuidadosamente a complexidade do cérebro, os cientistas e engenheiros criaram uma arquitetura idêntica mas que se pode encaixar na linguagem binária dos computadores. Uma dessas arquiteturas típicas é mostrada no diagrama em baixo,

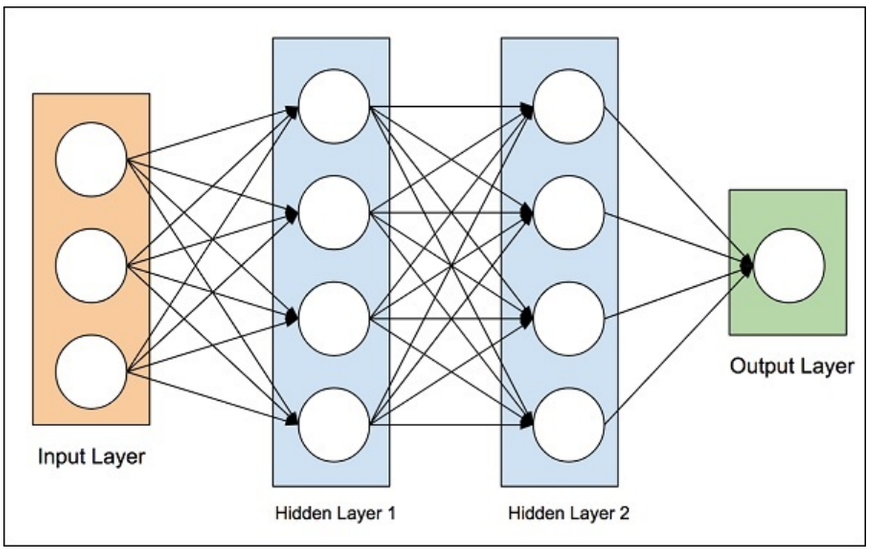

Figura 4.1: Exemplo de arquitetura de rede neuronal

Existe uma camada de entrada que possui muitos sensores para recolher dados do mundo exterior. No lado direito, existe uma camada de saída que dá o resultado previsto pela rede. Entre estas, outras camadas ocultas adicionam mais complexidade no treino da rede, fornecendo geralmente melhores resultados.

Existem vários tipos de arquiteturas projetadas que se apresentam a seguir.

4.1 Arquiteturas de RNA

Os diagramas a seguir mostram várias arquiteturas de RNA desenvolvidas ao longo do tempo e que são prática hoje em dia.

Cada arquitetura foi desenvolvida para um tipo específico de aplicação. Assim, ao pretender usar uma rede neural numa aplicação de aprendizagem de máquina, terá que se recorrer a uma das arquiteturas já existentes ou, em última instância, desenvolver a própria arquitetura.

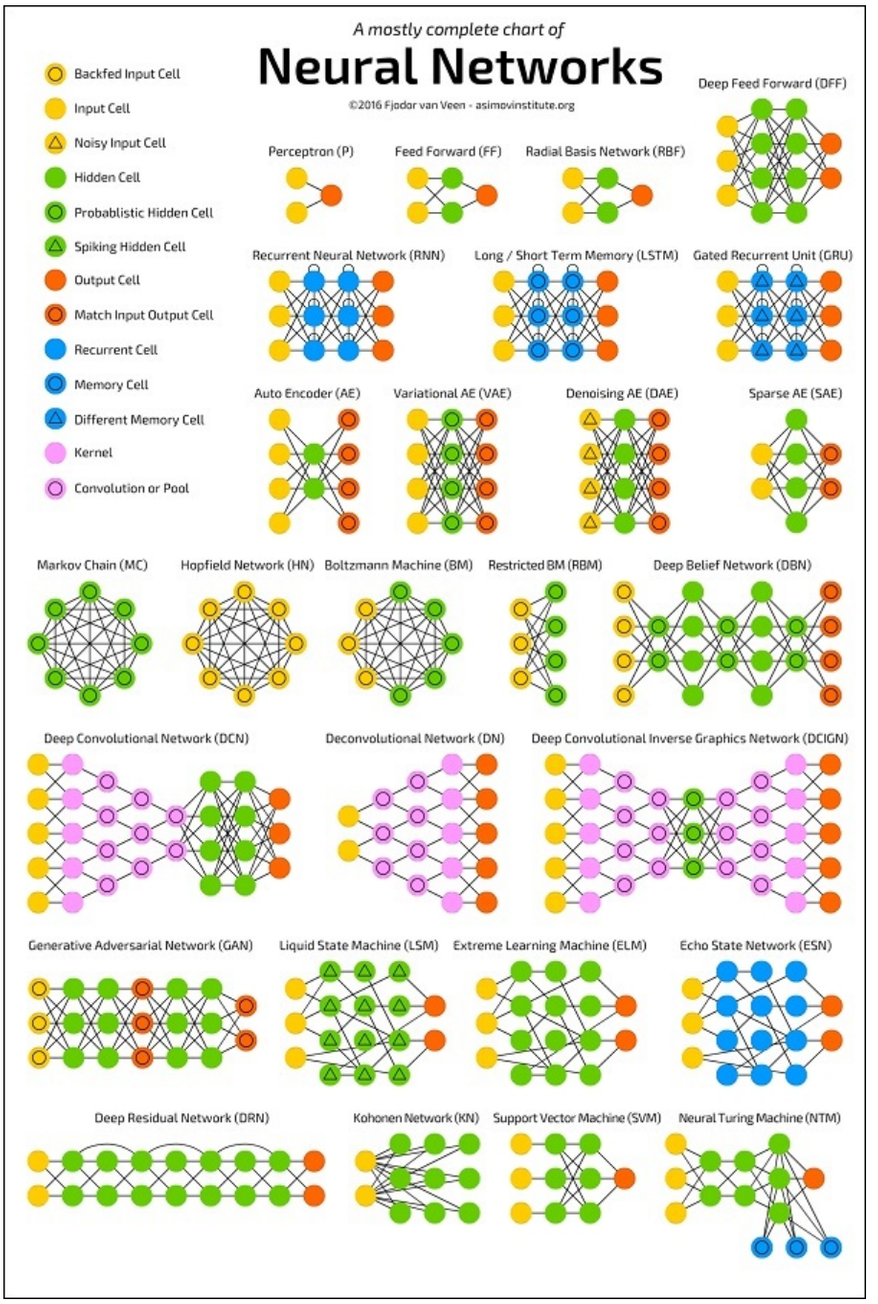

Figura 4.2: Diagrama de arquiteturas de RNA. Fonte

O tipo de arquitetura a usar depende das necessidades da aplicação. Não há uma diretriz que defina que arquitetura de rede específica se deve utilizar.

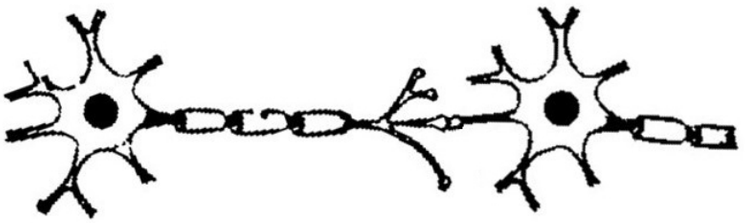

Figura 4.3: Esquema de transmissão de um neurónio biológico

Mesmo que o esquema da Figura 4.3 acima já seja uma imagem estilizada é possível transpor esse conceito para outro nível de abstração, um *perceptron*, isto é, uma simulação de um neurónio biológico. Da mesma forma que um neurónio biológico comunica com outro através da "sinapse", o *perceptron* multiplica os sinais de entrada recebidos pelos respetivos valores de peso, e transmite o resultado obtido à camada seguinte.

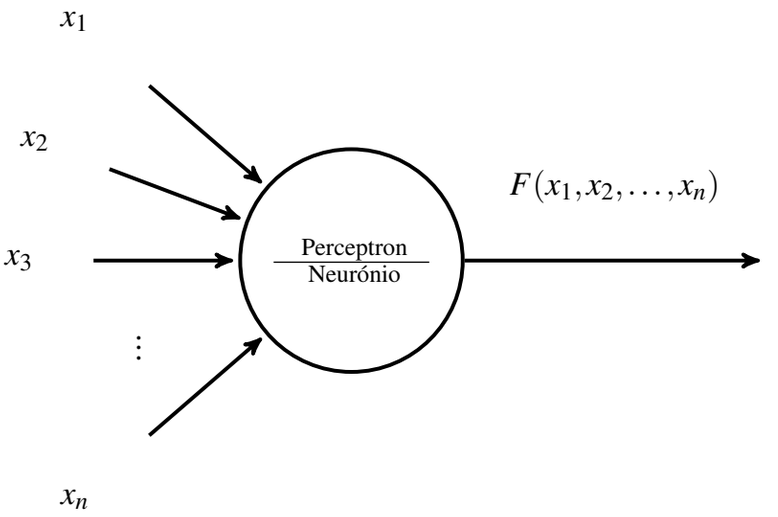

Figura 4.4: Esquema de um *perceptron*

Cada coordenada do input $X = (x_1, x_2, \ldots, x_n)$ tem um peso correspondente e, desta forma, a entrada pode ser ajustada individualmente para cada x_i.

Se W for o vetor dos pesos e $W \cdot X$ o produto interno dos dois vetores, o *perceptron* transforma o input $X = (x_1, x_2, ..., x_n)$ num valor de saída $F(X) = \Phi(W.X)$ (F chamada função de ativação). Por exemplo, no caso mais simples (um valor binário simples) a função de ativação pode ser definida através da fórmula,

$$F(X) = \begin{cases} 1 & \text{se} \quad W \cdot X + b \geq 0 \\ 0 & \text{caso contrário} \end{cases},$$

onde b é um termo constante independente do valor de entrada e que pode ser ajustado na fase de treino.

É o exemplo de uma rede capaz de separar duas classes no plano através de uma linha reta.

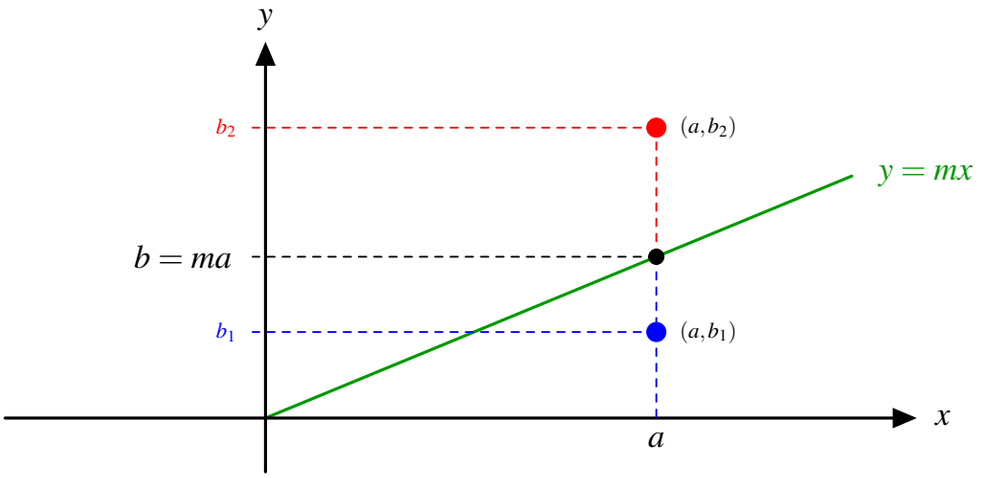

Figura 4.5: Reta que separa duas classes no plano.

Numa primeira fase, procuram-se retas (Figura 4.5) que passam pela origem $y = mx$ e que separam dois pontos ou duas classes no plano.
O ponto (a,b_1) pertence à classe azul pois $b_1 < b = ma$ ou $b_1 - ma < 0$. O ponto (a,b_2) pertence à classe vermelha pois $b_2 > b = ma$ ou $b_2 - ma > 0$.
Isto é, um ponto de coordenadas (x,y) pertencerá à classe vermelha se $y - mx > 0$ e à classe azul se $y - mx < 0$,

$$F(x,y) = \begin{cases} vermelho & \text{se} \quad (-m,1).(x,y) > 0 \\ azul & \text{se} \quad (-m,1).(x,y) < 0 \end{cases},$$

onde $W = (-m,1)$ é o vetor dos pesos de cada uma das coordenadas (x,y).

De uma forma geral, havendo um número considerável de pontos, o vetor $W = (-m,1)$ (ou neste caso o valor do declive m) é otimizado durante a fase de treino de modo que a separação em classes seja o mais acertada possível.

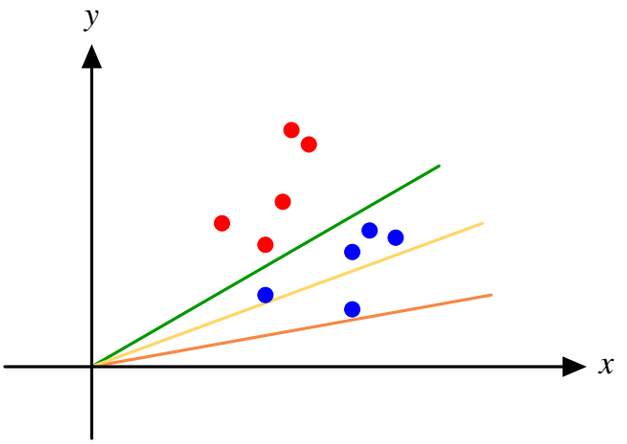

Figura 4.6: Otimização da reta que separa duas classes no plano.

Começa-se com pesos arbitrários, e compara-se o resultado com o valor esperado `y_train`, corrigindo esses parâmetros.

Adicionam-se pequenos valores aos pesos (positivos ou negativos). Para cada amostra, se o erro for negativo, considera-se um incremento negativo, e um valor positivo caso contrário. Havendo muitas amostras altera-se ligeiramente os pesos, usando uma taxa de aprendizagem (`self.learning_rate`) que controla a rapidez com que esses valores são atualizados.

4.2 MLPClassifier

O `MLPClassifier` assenta num método de treino iterativo, pois em cada passo, os parâmetros são atualizados com base no cálculo das derivadas parciais da função de perda em relação a esses mesmos parâmetros.

O Multi-layer Perceptron (MLP) é um algoritmo de aprendizagem supervisionada que, regra geral, cria uma função $f : \mathbf{R}^n \longrightarrow \mathbf{R}^p$, onde n é a dimensão do input e p é a dimensão do output. Simultaneamente, cada *perceptron* da camada oculta transforma os valores da camada anterior numa soma linear ponderada $W.X$, seguida por uma função de ativação não linear Φ (por ex. a função $\Phi(x) = \tanh(x)$).

Assim, em cada camada, os *perceptrons* atuam no output da camada anterior, de acordo com uma transformação $F(X) = \Phi(f(X))$.

A camada de saída recebe os valores da última camada oculta e transforma-os nos valores de saída.

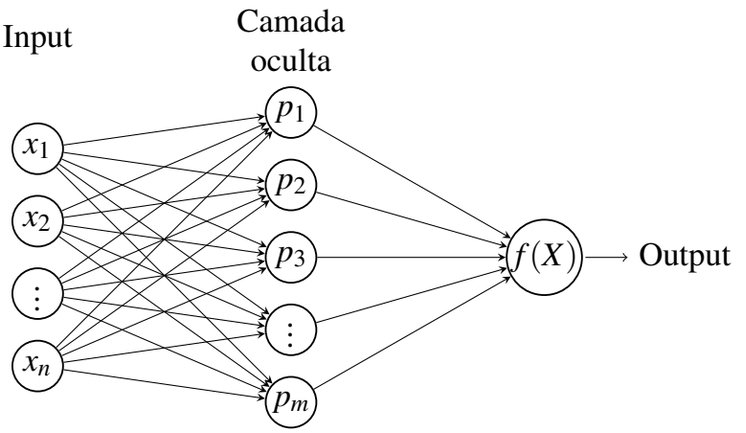

Figura 4.7: MLP de uma camada oculta

Deste ponto de vista, o algoritmo `MLP` é diferente da regressão logística, pois entre a entrada e saída pode haver uma ou mais camadas não lineares, chamadas camadas ocultas. A Figura 4.7 mostra um MLP de uma camada oculta com saída escalar ($p = 1$).

O algoritmo `MLP` apresenta ainda diversas funcionalidades e extensões, algumas descritas na tabela seguinte.

MLP_Classifier

from sklearn.neural_network import MLPClassifier	
MLPClassifier( ) . métodos	
fit(X, y[, sample_weight])	Ajusta o classificador MLP
get_params([deep])	Indica os parâmetros deste estimador.
partial_fit(X, y[, classes])	Atualiza o modelo com uma única iteração sobre os dados.
predict(X)	Prevê os resultados calculados pelo modelo.
predict_log_proba(X)	Devolve o log das estimativas de probabilidade.
predict_proba(X)	Devolve estimativas de probabilidade.
score(X, y[, sample_weight])	Devolve a precisão média nos dados de teste fornecidos.
set_params(**params)	Define os parâmetros deste modelo (“predictor").

Apresenta-se a seguir um possível código Python para a solução de um problema de classificação, usando o algoritmo `MLPClassifier`.

Ficheiro “MLPC_predictor_4.py"

```
from sklearn.neural_network import MLPClassifier
import pandas as pd
import pickle
data_train = pd.read_csv("./files-MLPC/train_set_3.csv")
data_validation = pd.read_csv("./files-MLPC/validation_set_3.csv")
X_train = data_train.iloc[:,3:-1].values
Y_train = data_train.iloc[:,-1].values
X_val = data_validation.iloc[:,3:-1].values
y_val = data_validation.iloc[:,-1].values
#Fitting the training data to the network
#Initializing the MLPClassifier
classifier = MLPClassifier(hidden_layer_sizes=(150,100,50), max_iter=100000,
        activation = 'relu',solver='adam',random_state=1)
# ativations é o tipo de ativação de cada neurónio
#Solver é o tipo de treino critério de paragem para as épocas (iterações)
classifier.fit(X_train, Y_train)
print("x=",X_train[1])
print("y=",Y_train[1])
#Predicting y for X_val
filename = './files-MLPC/predictorMLPC_4.sav'
pickle.dump(classifier, open(filename, 'wb'))
```

O dataset em questão `Final_Train_Dataset.csv` simula uma base de dados sobre "*The Data Scientists Salary*". Nesta fase de treino do modelo `MLPClassifier` já não se considera o dataset original mas uma versão trabalhada e estandardizada `scale_data_2.csv` e dividida em dados de treino (`train_set_3.csv`) e de teste (`validation_set_3.csv`).

O algoritmo `MLPClassifier` depende de uma série de parâmetros que se descrevem a seguir (os mais comuns).

MLPClassifier(`hidden_layer_sizes, max_iter, activation, solver, random_state`**):**

hidden_layer_sizes: tuple, length = n_layers - 2, default=(100,)
Se for do tipo array (por ex. (150,100,50)) cada elemento da sequência indica o número de neurónios em cada camada oculta.

max_iter: int, default=200
Número máximo de iterações. O algoritmo para quando se atinge este número de iterações ou a "convergência"(determinada por 'tol').

tol: float, default=1e-4
Tolerância para a otimização. Quando a perda ou pontuação não está melhorando em pelo menos "tol"para n_iter_no_change iterações consecutivas, a menos que learning_rate esteja definido como 'adaptativo', a convergência é considerada alcançada e o treino é interrompido.

activation: 'identity', 'logistic', 'tanh', 'relu', default='relu'
Função de ativação para a camada oculta.

- – 'identidade', ativação sem operação $f(x) = x$, útil para implementar regressão linear
- – 'logística', a função "sigmóide"logística $f(x) = \frac{1}{1+e^{-x}}$.
- – 'tanh', a função tangente hiperbólica $f(x) = \tanh(x)$.
- – 'relu', a função de unidade linear retificada $f(x) = \max(0,x)$.

solver: 'lbfgs', 'sgd', 'adam', default='adam'
O método para otimização dos pesos.
– 'lbfgs' é um otimizador da família de métodos quase-Newton.
– 'sgd' refere-se à descida de gradiente estocástica.
– 'adam' refere-se a um otimizador estocástico baseado no gradiente proposto por Kingma, Diederik e Jimmy Ba.

random_state: int, RandomState instance, default=None

O hiperparâmetro de estado aleatório é denotado por `random_state`. Geralmente usa-se um dos seguintes valores.

`None`: Este é o valor padrão. Isso permite que a função use o estado aleatório global de `np.random`. Se se considera `random_state=None`, cada vez que se corre o algoritmo este irá produzir resultados diferentes.

`int`: Pode-se usar qualquer número inteiro positivos ou 0. Os inteiros mais populares são 0 e 42. Quando usamos um inteiro para `random_state`, a função produz os mesmos resultados em diferentes execuções. Os resultados só são alterados alterando o valor atribuído a `random_state`.

Voltando ao dataset original (extrato na tabela a seguir), observam-se uma série de questões que poderão condicionar o seu tratamento e análise:

- os dados estão apresentados em formatos pouco adequados para o seu tratamento digital (por ex. `5-7 yrs`);
- há linhas com colunas sem dados;
- há colunas muito descritivas que poderão não ter qualquer relevância;
- ...

experience	job_description	job_desig	job_type	salary	...
5-7 yrs	Exp: Minimum 5 years				
10-17 yrs	He should have handled ...	Deputy Manager ...			
5-9 yrs	Must be an ...			15to25	...
6-10 yrs				6to10	...
1-3 yrs	Proven experience ...	Finance operatior ...		15to20	...

Portanto, é evidente que o dataset terá que ser refinado previamente antes que possa ser analisado por qualquer algoritmo, nomeadamente pelo `MLPClassifier`.

```
data = pd.read_csv("./files-MLPC/Final_Train_Dataset.csv")
data = data[['company_name_encoded','experience', 'location', 'salary']]
print(data)
```

O ficheiro a seguir faz uma limpeza dos dados, codifica strings e colunas sem dados em valores numéricos (imputação de *missing values*), adequa valores, elimina colunas pouco relevantes,... e grava o resultado num novo ficheiro (`clean_data_1.csv`).

Ficheiro “MLPC_clean_data_1.py"

```
#Cleaning the experience
exp = list(data.experience)
min_ex = []
max_ex = []
for i in range(len(exp)): # alteração do formato "experience" 5-7 yrs
   exp[i] = exp[i].replace("yrs","").strip()
   print(exp[i])
   min_ex.append(int(exp[i].split("-")[0].strip()))
   print(exp[i].split("-")[0])
   max_ex.append(int(exp[i].split("-")[1].strip()))
data["minimum_exp"] = min_ex
data["maximum_exp"] = max_ex
#Encoding the textual data in ‘location’ and ‘salary’ columns
#Label encoding location and salary
from sklearn.preprocessing import LabelEncoder
#Resolve problemas do dataset - codifica strings e missing values
        em valores numéricos
le = LabelEncoder()
data[’location’] = le.fit_transform(data[’location’])
data[’salary’] = le.fit_transform(data[’salary’])
#Delete the original experience column and reorder the data-frame.
#Deleting the original experience column and reordering
data.drop([’experience’], inplace = True, axis = 1)
data = data[[’company_name_encoded’, ’location’,’minimum_exp’,
        ’maximum_exp’, ’salary’]]
print(type(data))
data.to_csv("./files-MLPC/clean_data_1.csv")
```

De seguida, o código seguinte `MLPC_scale_data_2.py` procede à normalização/standardização destes dados, através da ferramenta `StandardScaler` (importação através de `from sklearn.preprocessing import StandardScaler`), e grava o resultado num novo ficheiro (`scale_data_2.csv`).

Ficheiro “MLPC_scale_data_2.py"

```
data = pd.read_csv("./files-MLPC/clean_data_1.csv")
sc = StandardScaler()
data[[’company_name_encoded’, ’location’, ’minimum_exp’, ’maximum_exp’]]
        =\sc.fit_transform(data[[’company_name_encoded’, ’location’,
        ’minimum_exp’, ’maximum_exp’]])
data.to_csv("./files-MLPC/scale_data_2.csv")
```

No ficheiro `MLPC_split_dataset_3.py` faz-se ainda a partição dos dados de `scale_data_2.csv` em dados de treino e de teste e gravam-se esses resultados em dois novos ficheiros (`train_set_3.csv` e `validation_set_3.csv`).

Ficheiro “MLPC_split_dataset_3.py"

```
#Splitting the dataset into  training and validation sets
from sklearn.model_selection import train_test_split
data = pd.read_csv("./files-MLPC/scale_data_2.csv")
training_set, validation_set = train_test_split(data, test_size = 0.2,
        random_state = 21)
print(type(validation_set))
training_set.to_csv("./files-MLPC/train_set_3.csv")
validation_set.to_csv("./files-MLPC/validation_set_3.csv")
```

Por fim, é ajustado o modelo `MLPClassifier`, através do código já descrito no ficheiro `MLPC_predictor_4.py` e criado o “executável"`predictorMLPC_4.sav`. Este último, depois testado no ficheiro `MLPC_evaluator_5.py`, é aplicado em `MLPC_predictorANN_6.py`.

Ficheiro “MLPC_evaluator_5.py"

```
from sklearn.metrics import confusion_matrix # load Confusion Matrix
filename="./files-MLPC/predictorMLPC_4.sav"
classifier = pickle.load(open(filename, 'rb'))
data_train = pd.read_csv("./files-MLPC/train_set_3.csv")
# As 3 primeiras colunas foram acrescentadas durante a normalização
data_validation = pd.read_csv("./files-MLPC/validation_set_3.csv")
X_val = data_validation.iloc[:,3:-1].values
# colunas a partir da coluna 4 excepto a última
y_val = data_validation.iloc[:,-1].values # a última coluna



print(type(X_val))   #print(X_val)
y_pred = classifier.predict(X_val)   # observations in y_val
cm = confusion_matrix(y_pred, y_val)
print("traço =",cm.trace()," Matriz confusão =",cm.sum()," score =",
classifier.score(X_val,y_val))
print("Accuracy of MLPClassifier : ", accuracy(cm))
```

Ficheiro “MLPC_predictorANN_6.py”

```
import pickle
import pandas as pd

filename="./files-MLPC/predictorMLPC_4.sav"
classifier = pickle.load(open(filename, 'rb'))
val_input=input("Introduction a value: ")
val=val_input.split(",")
val0=['company_name_encoded','location','minimum_exp','maximum_exp']
#val0=['company_name_encoded','location',
    'minimum_exp','maximum_exp']
list1=[]
for u in range(0,len(val0)):
    list1.append(val[u])
#list.append([list1])
df = pd.DataFrame([list1],columns=['company_name_encoded','location',
    'minimum_exp','maximum_exp'])

val_pred = classifier.predict(df.values)
print(val_pred)
#print(X_val[0])
# observations in y_val

"""
 testar com: !!!!!Dados já normalizados!!!!!!
 input para o predictopr ---> output correto
0.3668909827077753,-0.0108685439554915,-1.294266348246657,-1.9055666256418995  --> 0
-1.4957745915983542,1.7715743120468603,-1.294266348246657,-1.9055666256418995 --> 4
1.703563658301548,0.4085297751038854,-0.3991364025694826,-0.2559297165509968 --> 5
0.2896188300618879,0.7906482435802066,1.689500137343925,1.6293696081243203 --> 2
1.6405258495641135,0.7906482435802066,1.3911234887848665,1.393707192539906 --> 2
"""
```

Laboratório 4