DATA MINING II

Modelo de Neural Network (KERAS)

FOREST FIRES DATASET
UCI Machine Learning Repository

Trabalho realizado por Marta Lobo e Marta Barros

Definição do problema e recolha de dados

O nosso dataset é constituido por treze variáveis, numéricas e categóricas, cujo objetivo é a previsão da área queimada em incêndios florestais na zona nordeste de Portugal, utilizando para isso dados metereológicos e índices de risco de incêndio (FWI), entre outros fatores.

Tabela 1. Tabela-resumo das variáveis do dataset, obtido por `data.info()`.

	VARIÁVEL	TIPO
INPUT	**X** - x-axis spatial coordinate within the Montesinho park map: 1 to 9	*int64*
	Y - y-axis spatial coordinate within the Montesinho park map: 2 to 9	*int64*
	Month - month of the year: 'jan' to 'dec'	*string*
	Day - day of the week: 'mon' to 'sun'	*string*
	FFMC - FFMC index from the FWI system: 18.7 to 96.20	*float64*
	DMC - DMC index from the FWI system: 1.1 to 291.3	*float64*
	DC - DC index from the FWI system: 7.9 to 860.6	*float64*
	ISI - ISI index from the FWI system: 0.0 to 56.10	*float64*
	Temp - temperature in Celsius degrees: 2.2 to 33.30	*float64*
	RH - relative humidity in %: 15.0 to 100	*int64*
	Wind - wind speed in km/h: 0.40 to 9.40	*float64*
	Rain - outside rain in mm/m² : 0.0 to 6.4	*float64*
OUTPUT	**Area** - the burned area of the forest (in ha): 0.00 to 1090.84	*float64*

Deste modo, destas treze variáveis, a área, constituí a nossa variável de *output*, ou seja, aquela que queremos prever através de neural network (modelo keras). O primeiro passo foi abrir o dataset cedido pela UCI Machine Learning Repository.

```
# bibliotecas a usar

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
#load data
data=pd.read_csv("C:/Users/marta/OneDrive/Documentos/ISAG/Data Mining II/forestfires.csv")

month_dummies=pd.get_dummies(data['month'],prefix='month_')
day_dummies=pd.get_dummies(data['day'],prefix='day_')

data=pd.concat([data,month_dummies,day_dummies], axis=1)
data=data.drop(['month','day'],axis=1)
#dataset = data.values
```

Preparação dos dados

As variáveis categóricas mês e dia foram transformadas para numérico, em versão *dummy*, nomeadamente 1 a 12 e 1 a 5. Ao efetuar esta alteração o nosso *input* shape inicial de 13 variáveis passa a 29, na definição do modelo *keras*.

Recorrendo à biblioteca *sklearn*, foi feito um pré-processamento dos dados para sistema binário utilizando *LabelEncoder* e *OneHotEncoder,* codificando os dados categóricos em variáveis independentes.

Foi codificada a área, variável de *output*, tendo em vista a conversão de um vetor de classes em matriz de classe binária para uso apropriado da função custo. Os dados foram sujeitos a estandardização para melhor previsão pelos estimadores (*input*) através de *MinMaxScaler*, parte integrante da mesma biblioteca acima mencionada.

```
#Data Cleaning and Preprocessing
#Classification
#Convert to Acres then Classify Size
'''
Class 1.A - one acre or less;
Class 2.B - more than one acre, but less than 10 acres;
Class 3.C - 10 acres or more, but less than 100 acres;
Class 4.D - 100 acres or more, but less than 300 acres;
Class 5.E - 300 acres or more, but less than 1,000 acres;
Class 6.F - 1,000 acres or more, but less than 5,000 acres;
'''
import numpy as np
import pandas as pd
for i in range(0, len(y)):
    y[i] = (y[i]*2.47)
    if y[i] < 1.0:
        y[i] = 1
    elif y[i] < 10.0:
        y[i] = 2
    elif y[i] < 100.0:
        y[i] = 3
    elif y[i] < 300.0:
        y[i] = 4
    elif y[i] < 1000.0:
        y[i] = 5
    elif y[i] < 5000.0:
        y[i] = 6
    else:
        y[i] = 7
```

```
'''Encoding For Classification'''
from keras.utils import np_utils
y = np_utils.to_categorical(y)
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state=42)

# Feature Scaling to optimize
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
```

Definição do objetivo

Pretendemos obter uma resposta de *output*, área queimada, com base nas variáveis de *input* atribuidas. A função custo definida foi *categorical_crossentropy*uma vez que iremos definir sete categorias como neurónios de *output* (área). Deste modo, estamos perante um problema multi-classse.

Definição do protocolo de avaliação

O dataset será dividido pelo método de *hold-out* onde os dados de treino são sujeitos a nova divisão, sendo que uma serve o propósito de validação de dados e a outra o ajuste dos hiperparâmetros para melhoria contínua da performance do modelo.

A compilação do modelo é uma etapa fundamental que nos permite avançar para a fase de treino, correndo o nosso código e validando através de métricas o seu desempenho.Assim sendo, foi definida a função *loss*, usada para determinar o erro no processo de aprendizagem, um otimizador, cuja tarefa passa pela otimização dos pesos de *input* ao comparar a previsão do modelo com a função *loss* e, por fim, foi estabelecida uma métrica de avaliação, neste caso, a precisão para comparação do modelo base com os restantes.

Por ultimo, a avaliação do modelo ocorre através da função *fit()*. Os nossos dados de treino e os nossos dados de validação foram então treinados por 50 iterações (*epochs*) em conjuntos de 32 (*batch_size*).

Desenvolvimento de um modelo que seja melhor que uma baseline

Modelo base

Foi desenvolvido um modelo sequencial inicial com quatro camadas neuronais, em que a primeira dispõe de 12 neurónios com função de ativação *relu*, a segunda de novo com 12 neurónios e, por fim, 7 neurónios com a função de ativação *softmax*para que o *output* da rede neuronal seja normalizado a uma distribuição de probabilidades em classes.

```
# define the keras model
import tensorflow
from tensorflow import keras
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense
from keras import optimizers


model = Sequential()
model.add(layers.Dense(12, activation='relu', input_shape=(29,)))
model.add(layers.Dense(12, activation='relu'))
model.add(layers.Dense(7, activation = 'softmax'))
```

```
model.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
history=model.fit(X_train, y_train,epochs=50, batch_size=32, verbose=1, validation_data=(X_test, y_test))
```

```
Epoch 39/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1237 - accuracy: 0.4757 - val_loss: 1.2842 - val_accuracy: 0.5000
Epoch 40/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1005 - accuracy: 0.5058 - val_loss: 1.2857 - val_accuracy: 0.5000
Epoch 41/50
13/13 [==============================] - 0s 5ms/step - loss: 1.1066 - accuracy: 0.5191 - val_loss: 1.2853 - val_accuracy: 0.5096
Epoch 42/50
13/13 [==============================] - 0s 6ms/step - loss: 1.0650 - accuracy: 0.5269 - val_loss: 1.2858 - val_accuracy: 0.5096
Epoch 43/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1318 - accuracy: 0.4846 - val_loss: 1.2850 - val_accuracy: 0.5000
Epoch 44/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1035 - accuracy: 0.5216 - val_loss: 1.2859 - val_accuracy: 0.5096
Epoch 45/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1331 - accuracy: 0.5010 - val_loss: 1.2875 - val_accuracy: 0.5000
Epoch 46/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0629 - accuracy: 0.5435 - val_loss: 1.2900 - val_accuracy: 0.5000
Epoch 47/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0823 - accuracy: 0.4954 - val_loss: 1.2883 - val_accuracy: 0.4904
Epoch 48/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0714 - accuracy: 0.5089 - val_loss: 1.2894 - val_accuracy: 0.4904
Epoch 49/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0510 - accuracy: 0.5262 - val_loss: 1.2897 - val_accuracy: 0.4904
Epoch 50/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0427 - accuracy: 0.5596 - val_loss: 1.2922 - val_accuracy: 0.4904
```

Regularização e tuning dos híper-parâmetros

Posteriormente, este modelo *baseline* foi sujeito a variações ao nível do número de *layers* e do número de neurónios,assim como lhe foram acrescentadas camadas de *Dropout* e *Regularizadores L2*.

Os valores de precisão obtidos foram sujeitos a comparação para afinamento dos parâmetros. O protocolo de avaliação, acima mencionado, foi utilizado sem alterações em todas as versões do modelo testadas.

Com a aplicação de camadas de *Dropout* permitimos que o nosso modelo experimente diferentes arquiteturas de rede neuronal, em que aleatoriamente se excluem 20% dos neurónios durante o treino. Trata-se de um método de regularização para reduzir o *overfi*t e melhorar o erro.

```
#Modelo 2 - Dropout
```

```
#Modelo 2 - Dropout

from keras import optimizers
from keras.layers import Dropout

modeldrop = Sequential()
modeldrop.add(layers.Dense(24, activation='relu', input_shape=(29,)))
modeldrop.add(Dropout(0.2))
modeldrop.add(layers.Dense(64, activation='relu'))
modeldrop.add(Dropout(0.2))
modeldrop.add(layers.Dense(32, activation='relu'))
modeldrop.add(Dropout(0.2))
modeldrop.add(layers.Dense(7, activation = 'softmax'))
modeldrop.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
historydrop=modeldrop.fit(X_train, y_train,epochs=50, batch_size=32, verbose=1, validation_data=(X_test, y_test))
```

```
Epoch 39/50
13/13 [==============================] - 0s 5ms/step - loss: 1.1695 - accuracy: 0.5020 - val_loss: 1.2895 - val_accuracy: 0.5096
Epoch 40/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1983 - accuracy: 0.4808 - val_loss: 1.2867 - val_accuracy: 0.5096
Epoch 41/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1663 - accuracy: 0.4797 - val_loss: 1.2883 - val_accuracy: 0.5096
Epoch 42/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1423 - accuracy: 0.5020 - val_loss: 1.2891 - val_accuracy: 0.5096
Epoch 43/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1014 - accuracy: 0.5506 - val_loss: 1.2931 - val_accuracy: 0.5096
Epoch 44/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1180 - accuracy: 0.5155 - val_loss: 1.2982 - val_accuracy: 0.5096
Epoch 45/50
13/13 [==============================] - 0s 5ms/step - loss: 1.1498 - accuracy: 0.5313 - val_loss: 1.2954 - val_accuracy: 0.5000
Epoch 46/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0747 - accuracy: 0.5374 - val_loss: 1.3013 - val_accuracy: 0.5096
Epoch 47/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0613 - accuracy: 0.5158 - val_loss: 1.3116 - val_accuracy: 0.4904
Epoch 48/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0686 - accuracy: 0.5794 - val_loss: 1.3086 - val_accuracy: 0.5000
Epoch 49/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1339 - accuracy: 0.4915 - val_loss: 1.3077 - val_accuracy: 0.5096
Epoch 50/50
13/13 [==============================] - 0s 4ms/step - loss: 1.1053 - accuracy: 0.5367 - val_loss: 1.2961 - val_accuracy: 0.5096
```

Por sua vez, a utilização do regularizador L2 acrescenta um fator de penalização à função *loss* e permite analisar dados de regressão múltipla, que possam sofrer de colinearidade, medindo a complexidade do modelo segundo a soma dos quadrados dos pesos dos *inputs* que, de facto, contribuem para o modelo (feature selection).

```
#Modelo 3 - Kernel Regularizador
```

```
#Modelo 3 - Kernel Regularizador

from keras.layers import Dense
from keras.regularizers import l2

modelreg = Sequential()
modelreg.add(Dense(64, kernel_regularizer=l2(0.01), bias_regularizer=l2(0.01)))
modelreg.add(layers.Dense(24, activation='relu'))
modelreg.add(layers.Dense(24, activation='relu'))
modelreg.add(layers.Dense(7, activation = 'softmax'))
modelreg.compile(loss='categorical_crossentropy', optimizer='adam',metrics=['accuracy'])
historyreg=modelreg.fit(X_train, y_train,epochs=50, batch_size=32, verbose=1, validation_data=(X_test, y_test))
```

```
13/13 [==============================] - 0s 4ms/step - loss: 1.0291 - accuracy: 0.6368 - val_loss: 1.5393 - val_accuracy: 0.4423
Epoch 40/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0061 - accuracy: 0.6599 - val_loss: 1.5323 - val_accuracy: 0.4423
Epoch 41/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9830 - accuracy: 0.6619 - val_loss: 1.5121 - val_accuracy: 0.4135
Epoch 42/50
13/13 [==============================] - 0s 5ms/step - loss: 0.9801 - accuracy: 0.6691 - val_loss: 1.5332 - val_accuracy: 0.4231
Epoch 43/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9707 - accuracy: 0.6591 - val_loss: 1.5284 - val_accuracy: 0.4135
Epoch 44/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0130 - accuracy: 0.6495 - val_loss: 1.5247 - val_accuracy: 0.4327
Epoch 45/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9743 - accuracy: 0.6391 - val_loss: 1.5387 - val_accuracy: 0.4135
Epoch 46/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9861 - accuracy: 0.6499 - val_loss: 1.5468 - val_accuracy: 0.4135
Epoch 47/50
13/13 [==============================] - 0s 4ms/step - loss: 1.0037 - accuracy: 0.6265 - val_loss: 1.5493 - val_accuracy: 0.4135
Epoch 48/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9719 - accuracy: 0.6321 - val_loss: 1.5598 - val_accuracy: 0.4135
Epoch 49/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9536 - accuracy: 0.6844 - val_loss: 1.5671 - val_accuracy: 0.4135
Epoch 50/50
13/13 [==============================] - 0s 4ms/step - loss: 0.9400 - accuracy: 0.6762 - val_loss: 1.5675 - val_accuracy: 0.4135
```

Discussão

A precisão dos diferentes modelos, assim como, o peso da função custo encontram-se abaixo definidos em gráfico, como aplicação da biblioteca *matplotlib.pyplot*na **Figura 1**, para facilitar leitura e interpretação da performance entre eles.

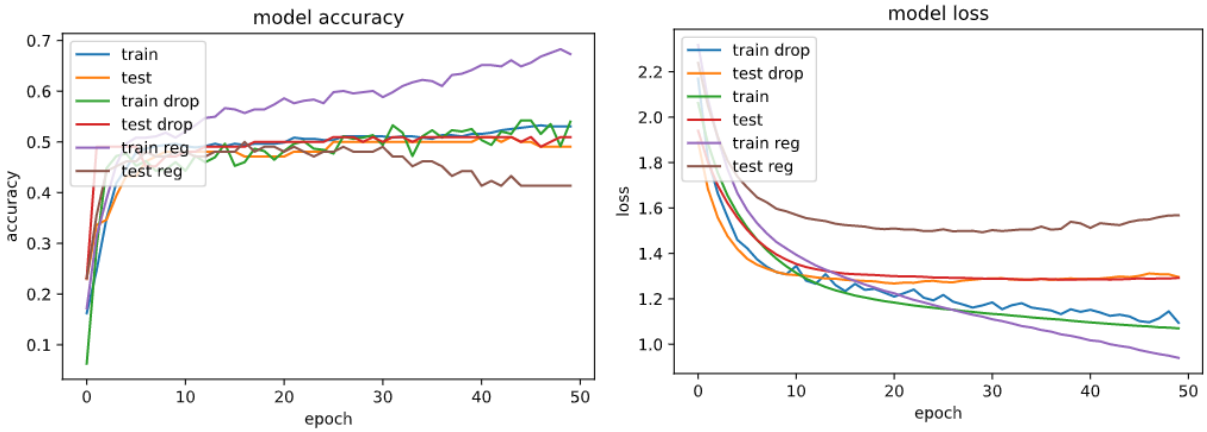

Figura 1. Comparação da precisão dos modelos ao longo do número de iterações (*epochs*):

1- Modelo Base (**Accuracy:--- Loss: ---**)
2- Modelo *Dropout* (**Accuracy:--- Loss: ---**)
3- Modelo Regularizador (**Accuracy:--- Loss: ---**)

A função*loss* é utilizada para a otimização do algoritmo de machine learning. Esta é calculada no set de treino e de validação e a interpretação da mesma deve ser feita com base na resposta obtida nos dois sets referidos. É nada mais do que a soma dos erros em cada exemplo e implica como o modelo se comporta em cada iteração de otimização.

A métrica precisão é usada para medir a performance do algoritmo e dita quão precisa é a precisão do modelo comparada aos dados verdadeiros.

Em todos os casos, a precisão do set de treino aumentou ao passo que a precisão do set de teste se manteve relativamente constante. No que se refere à função *loss*, temos a lógica contrária em que a perda diminuí no set de treino e aumenta no set de teste (**Tabela 2**).

Ao contrário do set de treino, que é parte do todo, o set de teste (ou validação) dispõe da totalidade dos dados, pelo que explica a constância da sua precisão.Sumariamente, verificamos que os modelos sofrem de *overfitting*, uma situação em que o modelo irá prever os dados de treino com elevada precisão, mas não os de validação.

Tabela 2. Variação da precisão e da função *loss* no dataset de treino e no de teste.

	TREINO	**TESTE**
Cálculo de Precisão	↑	↔
Função *Loss*	↓	↑

Conclusão

A partir da análise dos três modelos observamos que aplicando Drop Out, ou seja, assumindo uma perda de informação de 20% em cada iteração, este é o que se adequa melhor à previsão de área ardida, pois obtemos valores de *"accuracy"*de 54% e de *"validation accuracy"*de 51%. Aplicando os outros modelos como podemos observar no gráfico a diferença entre estes valores é maior.

No processo de treino destes modelos, aplicamos vários testes, acrescentamos layers e neurónios para perceber se aumentassemos estes valores se obteríamos modelos mais capazes, no entanto reparamos que apenas o valor de *"accuracy"* é que aumentava e aí sim observavamos valores próximos de 100%, contudo a *"validation accuracy"* manteve-se sempre nos 48%. Este processo levou-nos a concluir que o melhor modelo que poderíamos obter seria então o que nos desse valores de 50%.

Tendo em conta que o problema em questão é a previsão de área ardida, estes valores parecem-nos bastante satisfatórios e realistas uma vez que existem variáveis externas ao modelo que não são possíveis de controlar e o facto de ainda assim termos uma validação de 50% é um ótimo modelo.