am1-iron-ore-dataset

February 15, 2022

1 Iron Ore Dataset

In this notebook we are going to use the dataset **Iron Ore** explored in week **05-simple-predictions** to illustrate a simple Deep Artificial Neural Network and discuss some of its disatvantages.

As this dataset is not for classification, we borrow from week **05-simple-predictions** the code below to transform the problem into a classification problem:

```
# Splits from oscar Fe>60%, SiO2<9, Al2O3<2, P<0.08
split_points = [
    ('FE', 60, [False, True]),
    ('SIO2', 9, [True, False]),
    ('AL2O3', 2, [True, False]),
    ('P', 0.08, [True, False]),
]

# It's ore if everything is True
df['is_ore'] = np.vstack([
    pandas.cut(df[elem], bins=[0, split, 100], labels=is_ore)
    for elem, split,
```

In order to have a clean notebook, some functions are implemented in the file *utils.py* (e.g., plot_decision_boundary). We are not going to discuss the implementation aspects of these functions as it is not the scope, but you can to explore and read the content of the functions later on.

Summary: - Data Pre-processing - Building the DNN - Training the Model - Prediction and Performance Analysis

All the libraries used in this notebook are Open Source.

1.1 Data Pre-processing

[1]:
```
# Standard libraries
import numpy as np   # written in C, is faster and robust library for numerical␣
↪and matrix operations
import pandas as pd # data manipulation library, it is widely used for data␣
↪analysis and relies on numpy library.
import matplotlib.pyplot as plt # for plotting
import seaborn as sns # Plot nicely =) . Importing seaborn modifies the default␣
↪matplotlib color schemes and plot
                      # styles to improve readability and aesthetics.

# Auxiliar functions
from utils import *

from sklearn.model_selection import train_test_split #split arrays or matrices␣
↪into random train and test subsets
from sklearn.preprocessing import StandardScaler #Standardize features by␣
↪removing the mean and scaling to unit variance

# the following to lines will tell to the python kernel to always update the␣
↪kernel for every utils.py
# modification, without the need of restarting the kernel.
%load_ext autoreload
%autoreload 2

# using the 'inline' backend, your matplotlib graphs will be included in your␣
↪notebook, next to the code
%matplotlib inline
```

We are going to use Pandas for the data processing. The function read_csv is going to be used to read the csv file.

[2]:
```
# reading dataset
df = pd.read_csv('../data/iron_ore_study.csv')

# Splits from oscar Fe>60%, SiO2<9, Al2O3<2, P<0.08
split_points = [
    ('FE', 60, [False, True]),
    ('SIO2', 9, [True, False]),
    ('AL2O3', 2, [True, False]),
    ('P', 0.08, [True, False]),
]

# It's ore if everything is True
df['is_ore'] = np.vstack([
    pd.cut(df[elem], bins=[0, split, 100], labels=is_ore)
    for elem, split, is_ore in split_points
]).sum(axis=0) == 4

y = df.is_ore # set the variable 'y' to store the labels
# removing is_ore from the dataframe
list = ['is_ore']
df = df.drop(list,axis = 1 )
```

```
# split data train 70% and test 30%. You can try other splits here.
x_train, x_test, y_train, y_test = train_test_split(df, y, test_size=0.3,
                                                    random_state=42)

y_train = y_train.values # converting to numpy array
y_test = y_test.values # converting to numpy array

# normalising the data
scaler = StandardScaler()
scaler.fit(x_train) # not considering the label is_ore
x_train_scaled = scaler.transform(x_train)

x_test_scaled = scaler.transform(x_test)
```

Here we transform this dataset into a classification task using the strategy adopted in week **05-simple-predictions**.

[12]:
```
print(type(x_train_scaled), type(x_test_scaled))
print(x_train_scaled.shape)
```

```
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
(2450, 4)
```

1.2 Building the Deep Neural Network

We are going to use Keras to build our first Deep Neural Network. It is a high-level API for deep learning, which is a wrapper of TensorFlow, CNTK, or Theano. Keras was developed with a focus on enabling fast experimentation, which makes this framework easy to use and learn deep learning.

To build the model, we will use the following components from Keras:

- Sequencial: allows us to create models layer-by-layer.
- Dense: provides a regular fully-connected layer
- Dropout: provides dropout regularisation

Basically, we are going to define the sequence of our model by using *Sequential()*, which include the layers:

```
model = Sequential()
model.add(Dense(...))
...
```

once created the model we can configure the model for training using the method compile. Here we need to define the loss function (mean squared error, mean absolute error, cosine proximity, categorical cross entropy, among others.), the optimizer (Stochastic gradient descent, RMSprop, adam, among others) and the metric to define the evaluation metric to be used to evaluate the performance of the model in the training step, as follows:

```
model.compile(loss = "...",
              optimizer = "...")
```

Also, we have the option to see a summary representation of the model by using the function summary. This function summarises the model and tell us the number of parameters that we need to tune.

[4]:
```
from keras.models import Sequential # implements sequential function
from keras.layers import Dense # implements the fully connected layer
from keras.layers import Dropout # implements Dropout regularisation
```

[5]:
```
# Define the model.
model = Sequential()

# Dense layer (fully connected)
# First layer:
#  we need to define the input shape, which is the same of the shape of the␣
 ↪training set
#  which is 4
#  128 neurons
#  relu for activation function
model.add(Dense(128, input_shape=(x_train_scaled.shape[1],), activation='relu',␣
 ↪name='first_layer'))

# Regularisation:
#  Dropout consists in randomly setting a fraction rate of input units to
#  0 at each update during training time, which helps prevent overfitting.
#  for this model, we are going to define a rate equal 60% (0.6)
model.add(Dropout(0.6))

# second layer: output,
#  one neuron (binary output)
#  sigmoid for activation function
model.add(Dense(1, activation='sigmoid', name='output_layer'))

# Compile:
#  Optimiser: adam
#  Loss: binary_crossentropy, as our problem is binary classification (is_ore␣
 ↪or not is_ore)
#  Metric: binary_accuracy
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['binary_accuracy'])
```

```
2022-02-03 06:20:52.575829: I tensorflow/compiler/jit/xla_cpu_device.cc:41] Not
creating XLA devices, tf_xla_enable_xla_devices not set
2022-02-03 06:20:52.576010: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
```

```
compiler flags.
2022-02-03 06:20:52.578202: I
tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool
with default inter op setting: 2. Tune using inter_op_parallelism_threads for
best performance.
```

[6]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
first_layer (Dense)          (None, 128)               640
_________________________________________________________________
dropout (Dropout)            (None, 128)               0
_________________________________________________________________
output_layer (Dense)         (None, 1)                 129
=================================================================
Total params: 769
Trainable params: 769
Non-trainable params: 0
_________________________________________________________________
```

We have 769 parameters to be tuned. The first layer (first_layer) has 640 parameters, given by:

$$\#first\_layer = input\_shape * W_1 + b_1$$

where input_shape is given by the features (FE, SIO2, AL203 and P), W_1 are the weights in the first layer and b_1 is the bias shape, so we have:

$$\#first\_layer = 4 * 128 + 128 = 640$$

The second layer (output_layer) has 129 paramers, given by:

$$\#output\_layer = \#W_1 * W_2 + b_2$$

where $\#W_1$ is the shape of the *first_layer*, W_2 are the weights in the second layer and b_2 is the bias shape, so we have:

$$\#output\_layer = 128 * 1 + 1 = 129$$

1.2.1 Training the Model

Once defined the model, we need to train it by using the function fit. This function performs the optmisation step. Hence, we can define the following parameters such as:

- batch size: defines the number of samples that will be propagated through the network.

- epochs: defines the number of times in which all the training set (x_train_scaled) are used once to update the weights.
- validation split: defines the percentage of training data to be used for validation
- among others (click here for more information)

This function return the *history* of the training, that can be used for further performance analysis.

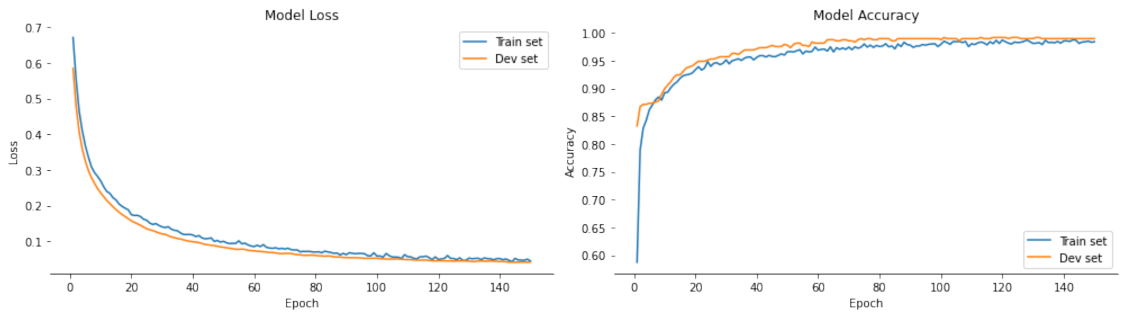
[7]:
```
# training the model
batch_size = 128
n_epochs = 150
history = model.fit(x_train_scaled,  #training set
                    y_train, # labels
                    batch_size = batch_size,
                    epochs = n_epochs,
                    validation_split = 0.2, # 20% percent for validation and␣
 ↪80% for training
                    verbose = 0 # muting the output, you can set this to 1␣
 ↪(default) to get insights
                                # about the loss and accuracy for every epoch
                   )
```

```
2022-02-03 06:20:52.724709: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR
optimization passes are enabled (registered 2)
2022-02-03 06:20:52.742376: I
tensorflow/core/platform/profile_utils/cpu_utils.cc:112] CPU Frequency:
2599990000 Hz
```

1.2.2 Prediction and Performance Analysis

Here we can see if the model overfits or underfits. First, we are going to plot the 'loss' and the 'Accuracy' from the training step.

[13]:
```
plot_loss_and_accuracy(history=history)
```



Question: Does the model overfits?

Once the model was trained, we can use function predict_classes to do some predictions and evaluate the performance of the model. As we saw in the last week, the performance assessment of the classifiers is **extremely important** in practice, as this provide insights of how the classifier performs on new data, which measures the **generalisation error**.

[9]:
```
predictions = model.predict_classes(x_test_scaled, batch_size=batch_size) #␣
 ↪predict the classes
pred = predictions.ravel() # Return a contiguous flattened array, in this case␣
 ↪a 1-D array, containing the elements of the
                           # predictions.
pred = pred.astype(dtype=np.bool) # if nedeed, we can define the type of the␣
 ↪array as boolean.
```

[10]:
```
# we are going to use F-measure to evaluate the performance
from sklearn.metrics import f1_score # computes the f1 score

# we are using prediction computed previously
f1_score_knn = f1_score(y_test, pred)
print('F1-score: ', f1_score_knn)
```

```
F1-score:  0.997005988023952
```

Another way to evaluate the model is by using the function evaluate provided by keras. This return the loss value & metrics values for the model in test mode. The metrics used is the same define in the **fit** function, in which this case is "binary_accuracy".

[11]:
```
loss_value, acc_value = model.evaluate(x_test_scaled, y_test, verbose=0)
print('Loss value: ', loss_value)
print('Binary acurracy value: ', acc_value)
```

```
Loss value:  0.03502056375145912
Binary acurracy value:  0.9961904883384705
```

Your turn

1) Evaluate the performance of the model by changing the "activation function". Here you can try "sigmoid" or "tanh".

For example:

```
model.add(Dense(128, input_shape=(x_train_scaled.shape[1],), activation='sigmoid', name='first_
```

You will need to create a new model, compile it and fit it.

[ ]:
```

```

2) Evaluate the performance of the model by changing the number of neurons in the first layer.

For example:

```
model.add(Dense(64, input_shape=(x_train_scaled.shape[1],), activation='...', name='first_laye
```

You will need to create a new model, compile it and fit it.

[ ]:

3) Create a new DNN model, a deepest one (more layers) and evaluate its performance.

   3.1) Create the model

For example:

```
# Define the model.
dnn = Sequential()

# layer 1
dnn.add(Dense(128, input_shape=(x_train_scaled.shape[1],), activation='relu', name='first_laye
dnn.add(Dropout(0.6))

# layer 2
dnn.add(Dense(256, activation='relu', name='second_layer'))
dnn.add(Dropout(0.6))

# layer 3
dnn.add(Dense(128, activation='relu', name='third_layer'))
dnn.add(Dropout(0.6))

# output layer
dnn.add(Dense(1, activation='sigmoid', name='output_layer'))
```

[ ]:

3.2) Try other optimisers (‘rmsprop’, ‘sgd’, ‘adagrad’ or ‘adadelta’ if you wich), loss and metrics

For example:

```
dnn.compile(optimizer='adam',
            loss='binary_crossentropy',
            metrics=['binary_accuracy'])
```

[ ]:

3.3) Summarize the model by using.

```
dnn.summary()
```

[ ]:

3.4) Train the model, here you can define the number of epochs and batch_size that best fit for you model

For example:

```
batch_size = 32
n_epochs = 20
history = dnn.fit(x_train_scaled,
                              y_train,
                              batch_size = batch_size,
                              epochs = n_epochs,
                              validation_split = 0.1, # 10% percent for validation and 80% for training
                              verbose = 0
                             )
```

[ ]:

3.5) Evaluate the model performance by using the metric that you think is the best.

For example:

```
loss_value, acc_value = dnn.evaluate(x_test_scaled, y_test, verbose=0)
print('Loss value: ', loss_value)
print('Binary acurracy value: ', loss_value)
```

Plot the loss and accuracy if you which.

[ ]:

[ ]: