004DS01 BootCamp

Jonas Jagers & Luis Reina

August 2018

1 Introduction

We demonstrate how a data science projects can be structured from end-to-end. The final model will be dockerized and put into "production", using the python package **Flask** as a web service. Read about **Docker** here www.docker.com, and Flask here http://flask.pocoo.org/ .

We want our readers to google all the *emphasized* or **bolded** terms marked throughout the tutorial. Our intention here is to show a way to tackle a problem we get as a Data Scientist. Take into account that the algorithms discussed and presented during the tutorial have not been thoroughly analyzed. The analysis is by no means complete.

2 Poker data set

You can read about (and find other data sets to play with) the data set here:

https://archive.ics.uci.edu/ml/datasets/Poker+Hand

2.1 Setting up a virtual Python environment

It is a good practise to use a virtual environment to avoid problem with package dependencies. This can be done with

```
virtualenv env -p python3 #(or 2 if you want a python 2 environment)
source env/bin/activate
```

Then install necessary packages in you environment.

2.2 Read and evaluate the data

Consider first to read the training and testing data sets. To read the files it is not needed to write your own read functions, we use the **Pandas** (https://pandas.pydata.org/) library for this.

```
from pandas import read_csv

# Import the training data with pandas read_csv function
data_file = read_csv('/home/user/dev/Poker/poker-hand-training-true.data')
```

Your should try to see the output of the following commands:

```
data_file.values
data_file.columns
data_file.describe()
data_file.head(10)
```

The Pandas library has some nice features included, as you can notice with the last commands. You can see that the column names are not related to our data set. If we want to define the column names we can do as follows: (here we just give column 1 name 1 etc)

```
# Define the names of the columns for the data
names = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'class']
# Import the training data with pandas read_csv function
data_file = read_csv('/home/user/dev/Poker/poker-hand-training-true.data',names=names)
```

You can now test again the data_file.columns command. Be sure to check some properties of the data_file with these commands:

```
test_data.shape
test.data.head(10) # first 10 rows
print data_file.describe() # some statistical properties
print(test_data.groupby( 'class' ).size()) # elements in each class
```

While runnig the last command you should get the following result, which tells you how many appearances has each class in the data set. You can see that the higher the class the less the representation. This will make classification harder on low frequency appearance classes. You should take this into account as the data is *skewed*.

$$\begin{bmatrix} class & \\ 0 & 12493 \\ 1 & 10599 \\ 2 & 1206 \\ 3 & 513 \\ 4 & 93 \\ 5 & 54 \\ 6 & 36 \\ 7 & 6 \\ 8 & 5 \\ 9 & 5 \\ dtype: & int64 \end{bmatrix}$$

So summarizing reading the csv data we have:

```
from pandas import read_csv

# Import the training data with pandas read_csv function
# Define the names of the columns for the data
names = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'class']
data_file = read_csv('/home/user/dev/Poker/poker-hand-training-true.data',names=names)
train_data = data_file.values
# Import the testing data with pandas read_csv function
data_file = read_csv('/home/user/dev/Poker/poker-hand-testing.data',names=names)
test_data = data_file.values
```

While looking at the train and test data, notice that they have both shape (N,11), where N is the size of the sample. Notice that we also lost the structure at data_file while copying just the ́values ́to the variable train_data and test_data. We want to separate though, both the train and the test data set into a shape (N, 10) & (N, 1). We can then define our variables as:

```
X_test = test_data[:,0:-1] # Variables - input
X_train = train_data[:,0:-1] # Variables - input
y_test = test_data[:,-1] # Poker Class LABEL
y_train = train_data[:,-1] # Poker Class LABEL
```

Verify that the shapes are correct, that is (N, 10) & (N, 1). If everything looks good, we can finally start to tackle the problem!

It is important to note that this data has a very good quality (since it is data for demonstration purpose). Normally, one has to spend a lot of time (approximately 80 %) cleaning and preparing the data. Google data preparation or data cleaning.

2.3 Training a model

After reading our data sets we can now start to train a model. Which model? You are the expert, so you choose.

As a good practice you can try to have a control model. We use a dummy model to choose the most frequent class for example. This is:

```
# Import a dummy classifier from the sklearn library
from sklearn.dummy import DummyClassifier
# Define the classifier with a "most frequent" strategy.
model = DummyClassifier(strategy='most_frequent')
# Traing the model
model.fit(X_train, y_train)
```

As easy as it looks, we just need three lines to train our model. This is pretty much the same for every classifier but neural networks of course. Now the next step is predicting. We predict with:

```
y_pred = model.predict(X_test)
```

2.4 Evaluation

But if this is so easy, what is the big fuss, right? Well, it is all about performance from this point forward. And we need some metrics to evaluate the performance of our classifier to compare it to other classifiers. How? Yes, you guessed correctly there is also ready to use code to evaluate performance within the **Sklearn** library (http://scikit-learn.org/stable/).

```
# Import Confusion matrix and accuracy_score for evaluation
from sklearn.metrics import confusion_matrix, accuracy_score
y_pred = model.predict(X_test)
print confusion_matrix(y_test, y_pred)
print accuracy_score(y_test, y_pred)
```

How good is this result? How do we know? One parameter is the accuracy score we printed. You should get a 0.5012. This is not so bad for 10 classes but it is not great either. What about the confusion matrix? You can see that just as we said, the classifier will always take the most frequent class, i.e. class 0. So all the observations are classified as class 0.

$$\begin{bmatrix}
501208 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
422498 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
47622 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
21121 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3885 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1996 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1424 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
230 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}$$

If it is a simple classifier, why do we get a high rate? Remember we said the data was skewed? Well this is what we mean with that. If we get the first class completely correct we will already be having more than 50% of the data set correct. You should always look out for this type of data sets.

We can now try to compare with another classifier. We can for example take kNN. For this, you do not need to implement the kNN algorithm yourself but to use the sklearn library:

```
# Create and fit a nearest-neighbor classifier
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier()
model.fit(X_train, y_train)
```

Notice that we are using a similar code structure for training the model. The accuray score should be around 0.5524 and the confusion matrix as follows:

$$\begin{bmatrix}
353495 & 146848 & 723 & 129 & 13 & 0 & 0 & 0 & 0 & 0 \\
220963 & 197340 & 3177 & 907 & 107 & 1 & 1 & 1 & 1 & 0 \\
19504 & 26370 & 1323 & 375 & 47 & 0 & 2 & 0 & 0 & 1 \\
4903 & 15271 & 637 & 292 & 13 & 0 & 5 & 0 & 0 & 0 \\
735 & 2921 & 183 & 29 & 16 & 0 & 0 & 0 & 1 & 0 \\
1496 & 499 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
237 & 1008 & 130 & 43 & 5 & 0 & 1 & 0 & 0 & 0 \\
8 & 173 & 32 & 16 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 8 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

This shows that kNN has a better performance than our dummy classifier, which is actually a good thing to know. But if we had nothing to compare to, how could we have known?

2.5 Data transformations

It is common, and may result in better performance, to transform the data. Read more here: http://scikit-learn.org/stable/modules/preprocessing.html

If any transformations are used, make sure that the transformation on the training data do not influence the test data. This is known as *data leakage*, and can be handled manually or can be avoided using a so called *pipeline*, see http://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

2.6 Model choice

As you have already noticed, as long as we have an algorithm implementation we can test different algorithms as classifiers for our poker data set. Here you find a piece of code we wrote to test four algorithms. Notice we still use the same structure as before.

```python
from sklearn.model_selection import KFold, cross_val_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier

# check some models
models = []
models.append(( 'LDA' , LinearDiscriminantAnalysis()))
models.append(( 'KNN' , KNeighborsClassifier()))
models.append(( 'CART' , DecisionTreeClassifier()))
models.append(( 'XGB' , XGBClassifier()))
# evaluate models
results = []
```

```
names = []
seed = 5
for name, model in models:
    kfold = KFold(n_splits=5, random_state=seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring= 'accuracy' )
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

If your run this code snippet you should get an output similar to:

```
 LDA:   0.499500  (0.006843)
 KNN:   0.549242  (0.006944)
CART:   0.491903  (0.005911)
 XGB:   0.595786  (0.003962)
```

You can immediately notice (by simple accuracy comparison) that LDA and CART are not suitable for this data set, at least not running with its default parameters. We already knew that kNN would perform better from previous sections and now we have XGB which performs better than kNN. Could we do something to improve the results?
Yes! We could perform a transformation to the data to try to improve the performance. Just to illustrate how much this can affect our results we paste the results after performing a *binary encoding* transformation:

```
 LDA:   0.492823  (0.001901)
 KNN:   0.525651  (0.004231)
CART:   0.540486  (0.018158)
 XGB:   0.678675  (0.015118)
```

As you can see we get a better performance from CART and XGB but not from KNN and LDA. Is there still something else we can do?
Yes! As we said before there is (often) a lot of different parameter settings for each model. You should try changing them and see if your results get better. See http://scikit-learn.org for information. Anyway, with the default parameter setting XGBoost is best with almost mean accuracy of almost 0.6 and standard deviation of 0.0039. Not great yet!
We will now use **Keras** and try some neural networks to evaluate its performance and try yet another solution. This gives a more acceptable accuracy of 0.93. Again, test different parameters and network settings. Actually, we got 0.998 after using a binary encoding transformation and changing the default parameters. See https://keras.io/ for more information.

```
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
```

```
y_test = to_categorical(Y_test)
y_train = to_categorical(Y_train)
# create a neural network
model = Sequential()
model.add(Dense(85, input_dim=10, activation= 'relu' ))
model.add(Dense(120,activation ='tanh'))
model.add(Dense(50, activation= 'relu' ))
model.add(Dense(10, activation= 'softmax' ))
# Compile the network
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ]
# Fit the network
model.fit(X_train, y_train, epochs=50, batch_size=100)
# evaluate
scores = model.evaluate(X_test, y_test)
print(scores[1])
```

2.7 Model to production

When we are satisfied with a model, we are ready to put it in production. First save the trained model so we do not need to retrain it (this can take a lot of time if the data set is large). To be able to interact with the model from outside we use the python package Flask.

Flask uses the *Jinja* template library to render templates. This enables python to interact directly with the html code. When everything runs properly, we put it in a ”closed container”, that runs in an OS-independent manner.

You can find a simple application of the flask api to be used with the model files exported from this tutorial at: https://github.com/lfreina/Poker_Tutorial

2.8 Code

You can copy and paste the whole code here:

```
# Load libraries
from pandas import read_csv
from pandas.plotting import scatter_matrix
#from matplotlib import pyplot
#from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from xgboost import XGBClassifier
from keras.models import Sequential
from keras.layers import Dense
from keras.utils import to_categorical
from keras.models import model_from_json
# Load dataset
names = ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'class']
testData = read_csv('poker-hand-testing.data',names=names)
trainData = read_csv('poker-hand-training-true.data')
print(testData.shape)

X_train = trainData.values[:,0:-1]
Y_train = trainData.values[:,-1]
X_test = testData.values[:,0:-1]
Y_test = testData.values[:,-1]

# shape of dataset
print(testData.shape)

# see first rows of the data
print(testData.describe())
#names = [ sepangth , sepal-width , petal-length ,
#dataset = read_csv(filename, names=names)

# class distribution
print(testData.groupby( 'class' ).size())

# check models
models = []
models.append(( 'LDA' , LinearDiscriminantAnalysis()))
models.append(( 'KNN' , KNeighborsClassifier()))
models.append(( 'CART' , DecisionTreeClassifier()))
models.append(( 'XGB' , XGBClassifier(n_estimators=100,max_depth=3,learning_rate=0.8)))
# evaluate models
results = []
names = []
seed = 5

for name, model in models:
    kfold = KFold(n_splits=5, random_state=seed)
    cv_results = cross_val_score(model, X_train, Y_train, cv=kfold, scoring= 'accuracy' )
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
# transform labels to categorical
y_test = to_categorical(Y_test)
y_train = to_categorical(Y_train)
# create a neural network
model = Sequential()
model.add(Dense(85, input_dim=10, activation= 'relu' ))
model.add(Dense(120,activation ='tanh'))
model.add(Dense(80, activation= 'relu' ))
model.add(Dense(40, activation= 'relu' ))
model.add(Dense(10, activation= 'softmax' ))
# Compile the network
model.compile(loss= 'categorical_crossentropy' , optimizer= 'adam' , metrics=[ 'accuracy' ])
# Fit the network
model.fit(X_train, y_train, epochs=50, batch_size=50)#,verbose=1,validation_data=(X_test,y_
# evaluate
scores = model.evaluate(X_test, y_test)
print(scores[1])
```

```
# save model as JSON
model_json = model.to_json()
with open("model.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("model.h5")
print("Saved model to disk")

# load json file and create model
json_file = open( 'model.json' , 'r' )
loaded_model_json = json_file.read()

json_file.close()
loaded_model = model_from_json(loaded_model_json)
# load weights into new model
loaded_model.load_weights("model.h5")
print("Loaded model from disk")
prediction = loaded_model.predict(X_test)
```