ANN_classifier

July 26, 2019

The following is an ANN classifier for determining the probability that a specific customer leaves a financial institution

Geographic and financial customer attributes are included altogether, as these are believed to have an impacting factor on the decison of leaving.

[12]:
```
#ANN implementation for classification problem



# Importing libraries

import keras
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt



# Importing useful classes

from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import GridSearchCV, cross_val_score
from sklearn.preprocessing import LabelEncoder, OneHotEncoder, StandardScaler #␣
 ↪For cat. data encoding
from sklearn.metrics import confusion_matrix



# Importing dataset

ds = pd.read_csv('Churn_Modelling.csv') # Importing dataset from parent dir
X = ds.iloc[:, 3:13].values # Selecting input array from dataset (independent␣
 ↪variables: Customer indicators)
y = ds.iloc[:, 13].values # Selecting output vector from dataset (dependent␣
 ↪variable: Categorical (binary) variable)
```

```python
# Data pre-processing

labelencoder_X_1 = LabelEncoder() # Encoder object for encoding non-numerical␣
↪data into numerical data (Non-numerical data values are encoded in columns 1␣
↪and 2)
X[:, 1] = labelencoder_X_1.fit_transform(X[:, 1])
labelencoder_X_2 = LabelEncoder()
X[:, 2] = labelencoder_X_2.fit_transform(X[:, 2])
onehotencoder = OneHotEncoder(categorical_features = [1]) # There are more than␣
↪2 countries in ds, so one-hot enc. needed in column 1
X = onehotencoder.fit_transform(X).toarray()
X = X[:, 1:] # One variable removed after one-hot encoding because of data␣
↪multi-collinearity


# Splitting dataset into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,␣
↪random_state = 0) # Test size is 0.2 (20%) of total ds


# Feature scaling for easing numerical computatations

sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)


# Initialising ANN

# There are 11 nodes in input layer and 1 in output layer, hence (1+11)/2 nodes␣
↪in hidden layers

def build_classifier():

    classifier = Sequential() # Classifier object for ANN structure
    classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation␣
↪= 'relu', input_dim = 11)) # Input layer has 11 neurons, rect. lin. unit␣
↪function
    classifier.add(Dropout(p = 0.1)) # Avoid overfitting
    classifier.add(Dense(units = 6, kernel_initializer = 'uniform', activation␣
↪= 'relu')) # Second layer added
    classifier.add(Dropout(p = 0.1))
    classifier.add(Dense(units = 1, kernel_initializer = 'uniform', activation␣
↪= 'sigmoid')) # Output layer has 1 node, sigmoid funct. for 0-1 range values␣
↪(probabilities)
```

```
    classifier.compile(optimizer = 'adam', loss = 'binary_crossentropy',␣
↪metrics = ['accuracy']) # Using adam alg. for weight update                                                                                                                                                                                                                                ␣
↪                  # Binary value (YES OR NO) classification problem:␣
↪binary_crossentropy loss func.                                                                                                                                                                                                                                                              ␣
↪                  # Accuracy metric for monitoring model performance      ␣
↪
    return classifier


#Fitting training set to model

classifier = KerasClassifier(build_fn = build_classifier, batch_size = 10,␣
↪epochs = 100) #Weights are updated every 10 data samples
classifier.fit(X_train, y_train)

# Predicting the test set results

y_predicted = classifier.predict(X_test) # Extracting classifier predicitions
y_predicted = (y_predicted > 0.5) # Below 0.5 data value is transformed to 0,␣
↪above 0.5 to 1
```

```
/home/alexbocc/anaconda3/lib/python3.7/site-
packages/sklearn/preprocessing/_encoders.py:414: FutureWarning: The handling of
integer data will change in version 0.22. Currently, the categories are
determined based on the range [0, max(values)], while in the future they will be
determined based on the unique values.
If you want the future behaviour and silence this warning, you can specify
"categories='auto'".
In case you used a LabelEncoder before this OneHotEncoder to convert the
categories to integers, then you can now use the OneHotEncoder directly.
  warnings.warn(msg, FutureWarning)
/home/alexbocc/anaconda3/lib/python3.7/site-
packages/sklearn/preprocessing/_encoders.py:450: DeprecationWarning: The
'categorical_features' keyword is deprecated in version 0.20 and will be removed
in 0.22. You can use the ColumnTransformer instead.
  "use the ColumnTransformer instead.", DeprecationWarning)
/home/alexbocc/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:59:
UserWarning: Update your `Dropout` call to the Keras 2 API: `Dropout(rate=0.1)`
/home/alexbocc/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:61:
UserWarning: Update your `Dropout` call to the Keras 2 API: `Dropout(rate=0.1)`

Epoch 1/100
8000/8000 [==============================] - 7s 876us/step - loss: 0.5013 - acc:
0.7951
Epoch 2/100
```

```
8000/8000 [==============================] - 4s 507us/step - loss: 0.4330 - acc:
0.8015
Epoch 3/100
8000/8000 [==============================] - 4s 528us/step - loss: 0.4310 - acc:
0.8112
Epoch 4/100
8000/8000 [==============================] - 4s 518us/step - loss: 0.4266 - acc:
0.8155
Epoch 5/100
8000/8000 [==============================] - 4s 482us/step - loss: 0.4235 - acc:
0.8202
Epoch 6/100
8000/8000 [==============================] - 4s 501us/step - loss: 0.4182 - acc:
0.8237
Epoch 7/100
8000/8000 [==============================] - 5s 564us/step - loss: 0.4212 - acc:
0.8205
Epoch 8/100
8000/8000 [==============================] - 4s 504us/step - loss: 0.4203 - acc:
0.8216
Epoch 9/100
8000/8000 [==============================] - 4s 500us/step - loss: 0.4167 - acc:
0.8249
Epoch 10/100
8000/8000 [==============================] - 4s 507us/step - loss: 0.4156 - acc:
0.8246
Epoch 11/100
8000/8000 [==============================] - 4s 509us/step - loss: 0.4199 - acc:
0.8231
Epoch 12/100
8000/8000 [==============================] - 4s 503us/step - loss: 0.4155 - acc:
0.8261
Epoch 13/100
8000/8000 [==============================] - 4s 483us/step - loss: 0.4154 - acc:
0.8244
Epoch 14/100
8000/8000 [==============================] - 4s 484us/step - loss: 0.4173 - acc:
0.8251
Epoch 15/100
8000/8000 [==============================] - 4s 507us/step - loss: 0.4138 - acc:
0.8256
Epoch 16/100
8000/8000 [==============================] - 4s 525us/step - loss: 0.4119 - acc:
0.8265
Epoch 17/100
8000/8000 [==============================] - 4s 492us/step - loss: 0.4152 - acc:
0.8261
Epoch 18/100
```

```
8000/8000 [==============================] - 4s 513us/step - loss: 0.4104 - acc:
0.8290
Epoch 19/100
8000/8000 [==============================] - 4s 535us/step - loss: 0.4151 - acc:
0.8245
Epoch 20/100
8000/8000 [==============================] - 4s 498us/step - loss: 0.4120 - acc:
0.8262
Epoch 21/100
8000/8000 [==============================] - 4s 521us/step - loss: 0.4123 - acc:
0.8280
Epoch 22/100
8000/8000 [==============================] - 4s 503us/step - loss: 0.4084 - acc:
0.8291
Epoch 23/100
8000/8000 [==============================] - 4s 509us/step - loss: 0.4124 - acc:
0.8251
Epoch 24/100
8000/8000 [==============================] - 4s 514us/step - loss: 0.4097 - acc:
0.8282
Epoch 25/100
8000/8000 [==============================] - 4s 515us/step - loss: 0.4129 - acc:
0.8254
Epoch 26/100
8000/8000 [==============================] - 4s 499us/step - loss: 0.4088 - acc:
0.8286
Epoch 27/100
8000/8000 [==============================] - 4s 499us/step - loss: 0.4106 - acc:
0.8262
Epoch 28/100
8000/8000 [==============================] - 4s 477us/step - loss: 0.4116 - acc:
0.8246
Epoch 29/100
8000/8000 [==============================] - 4s 515us/step - loss: 0.4103 - acc:
0.8271
Epoch 30/100
8000/8000 [==============================] - 4s 526us/step - loss: 0.4095 - acc:
0.8285
Epoch 31/100
8000/8000 [==============================] - 4s 493us/step - loss: 0.4099 - acc:
0.8300
Epoch 32/100
8000/8000 [==============================] - 4s 501us/step - loss: 0.4064 - acc:
0.8284
Epoch 33/100
8000/8000 [==============================] - 4s 535us/step - loss: 0.4120 - acc:
0.8264
Epoch 34/100
```

```
8000/8000 [==============================] - 4s 515us/step - loss: 0.4085 - acc:
0.8274
Epoch 35/100
8000/8000 [==============================] - 4s 508us/step - loss: 0.4108 - acc:
0.8251
Epoch 36/100
8000/8000 [==============================] - 4s 525us/step - loss: 0.4085 - acc:
0.8236
Epoch 37/100
8000/8000 [==============================] - 4s 507us/step - loss: 0.4023 - acc:
0.8309
Epoch 38/100
8000/8000 [==============================] - 4s 520us/step - loss: 0.4033 - acc:
0.8322
Epoch 39/100
8000/8000 [==============================] - 5s 573us/step - loss: 0.4026 - acc:
0.8321
Epoch 40/100
8000/8000 [==============================] - 4s 539us/step - loss: 0.4048 - acc:
0.8317
Epoch 41/100
8000/8000 [==============================] - 4s 508us/step - loss: 0.4059 - acc:
0.8282
Epoch 42/100
8000/8000 [==============================] - 4s 476us/step - loss: 0.4045 - acc:
0.8312
Epoch 43/100
8000/8000 [==============================] - 4s 545us/step - loss: 0.4059 - acc:
0.8295
Epoch 44/100
8000/8000 [==============================] - 6s 718us/step - loss: 0.4054 - acc:
0.8295
Epoch 45/100
8000/8000 [==============================] - 6s 739us/step - loss: 0.4027 - acc:
0.8324
Epoch 46/100
8000/8000 [==============================] - 5s 663us/step - loss: 0.4030 - acc:
0.8316
Epoch 47/100
8000/8000 [==============================] - 4s 467us/step - loss: 0.4025 - acc:
0.8341
Epoch 48/100
8000/8000 [==============================] - 4s 530us/step - loss: 0.4042 - acc:
0.8295
Epoch 49/100
8000/8000 [==============================] - 4s 468us/step - loss: 0.4031 - acc:
0.8307
Epoch 50/100
```

```
8000/8000 [==============================] - 4s 471us/step - loss: 0.4030 - acc:
0.8326
Epoch 51/100
8000/8000 [==============================] - 4s 470us/step - loss: 0.4028 - acc:
0.8335
Epoch 52/100
8000/8000 [==============================] - 4s 533us/step - loss: 0.4052 - acc:
0.8284
Epoch 53/100
8000/8000 [==============================] - 4s 496us/step - loss: 0.4030 - acc:
0.8290
Epoch 54/100
8000/8000 [==============================] - 4s 500us/step - loss: 0.4020 - acc:
0.8299
Epoch 55/100
8000/8000 [==============================] - 4s 483us/step - loss: 0.4018 - acc:
0.8332
Epoch 56/100
8000/8000 [==============================] - 4s 480us/step - loss: 0.4070 - acc:
0.8281
Epoch 57/100
8000/8000 [==============================] - 4s 545us/step - loss: 0.4044 - acc:
0.8311
Epoch 58/100
8000/8000 [==============================] - 4s 526us/step - loss: 0.4043 - acc:
0.8307
Epoch 59/100
8000/8000 [==============================] - 4s 518us/step - loss: 0.4016 - acc:
0.8311
Epoch 60/100
8000/8000 [==============================] - 5s 574us/step - loss: 0.4034 - acc:
0.8272
Epoch 61/100
8000/8000 [==============================] - 6s 698us/step - loss: 0.4028 - acc:
0.8291
Epoch 62/100
8000/8000 [==============================] - 5s 615us/step - loss: 0.4022 - acc:
0.8310
Epoch 63/100
8000/8000 [==============================] - 4s 546us/step - loss: 0.4028 - acc:
0.8317
Epoch 64/100
8000/8000 [==============================] - 4s 492us/step - loss: 0.4009 - acc:
0.8294
Epoch 65/100
8000/8000 [==============================] - 4s 512us/step - loss: 0.4053 - acc:
0.8301
Epoch 66/100
```

```
8000/8000 [==============================] - 4s 533us/step - loss: 0.4033 - acc:
0.8307
Epoch 67/100
8000/8000 [==============================] - 4s 511us/step - loss: 0.4049 - acc:
0.8302
Epoch 68/100
8000/8000 [==============================] - 4s 537us/step - loss: 0.4047 - acc:
0.8300
Epoch 69/100
8000/8000 [==============================] - 4s 535us/step - loss: 0.4022 - acc:
0.8319
Epoch 70/100
8000/8000 [==============================] - 4s 493us/step - loss: 0.3981 - acc:
0.8325
Epoch 71/100
8000/8000 [==============================] - 4s 493us/step - loss: 0.4028 - acc:
0.8320
Epoch 72/100
8000/8000 [==============================] - 4s 494us/step - loss: 0.4006 - acc:
0.8324
Epoch 73/100
8000/8000 [==============================] - 4s 506us/step - loss: 0.4041 - acc:
0.8317
Epoch 74/100
8000/8000 [==============================] - 4s 500us/step - loss: 0.3989 - acc:
0.8342
Epoch 75/100
8000/8000 [==============================] - 4s 503us/step - loss: 0.3973 - acc:
0.8375
Epoch 76/100
8000/8000 [==============================] - 4s 516us/step - loss: 0.4046 - acc:
0.8291
Epoch 77/100
8000/8000 [==============================] - 4s 517us/step - loss: 0.4057 - acc:
0.8302
Epoch 78/100
8000/8000 [==============================] - 4s 509us/step - loss: 0.4044 - acc:
0.8300
Epoch 79/100
8000/8000 [==============================] - 4s 498us/step - loss: 0.4001 - acc:
0.8324
Epoch 80/100
8000/8000 [==============================] - 4s 508us/step - loss: 0.4016 - acc:
0.8332
Epoch 81/100
8000/8000 [==============================] - 4s 512us/step - loss: 0.4032 - acc:
0.8311
Epoch 82/100
```

```
8000/8000 [==============================] - 4s 488us/step - loss: 0.4030 - acc:
0.8334
Epoch 83/100
8000/8000 [==============================] - 4s 536us/step - loss: 0.3983 - acc:
0.8356
Epoch 84/100
8000/8000 [==============================] - 4s 482us/step - loss: 0.4018 - acc:
0.8306
Epoch 85/100
8000/8000 [==============================] - 4s 515us/step - loss: 0.3974 - acc:
0.8336
Epoch 86/100
8000/8000 [==============================] - 4s 526us/step - loss: 0.3972 - acc:
0.8295
Epoch 87/100
8000/8000 [==============================] - 4s 502us/step - loss: 0.3963 - acc:
0.8344
Epoch 88/100
8000/8000 [==============================] - 4s 509us/step - loss: 0.3938 - acc:
0.8339
Epoch 89/100
8000/8000 [==============================] - 4s 510us/step - loss: 0.3951 - acc:
0.8334
Epoch 90/100
8000/8000 [==============================] - 4s 513us/step - loss: 0.3928 - acc:
0.8341
Epoch 91/100
8000/8000 [==============================] - 4s 510us/step - loss: 0.3927 - acc:
0.8345
Epoch 92/100
8000/8000 [==============================] - 4s 499us/step - loss: 0.3906 - acc:
0.8347
Epoch 93/100
8000/8000 [==============================] - 4s 526us/step - loss: 0.3893 - acc:
0.8342
Epoch 94/100
8000/8000 [==============================] - 4s 516us/step - loss: 0.3909 - acc:
0.8349
Epoch 95/100
8000/8000 [==============================] - 4s 493us/step - loss: 0.3872 - acc:
0.8376
Epoch 96/100
8000/8000 [==============================] - 4s 520us/step - loss: 0.3885 - acc:
0.8365
Epoch 97/100
8000/8000 [==============================] - 4s 537us/step - loss: 0.3883 - acc:
0.8331
Epoch 98/100
```

```
8000/8000 [==============================] - 5s 570us/step - loss: 0.3853 - acc:
0.8379
Epoch 99/100
8000/8000 [==============================] - 4s 551us/step - loss: 0.3849 - acc:
0.8371
Epoch 100/100
8000/8000 [==============================] - 4s 502us/step - loss: 0.3868 - acc:
0.8337
```

0.0.1 After learning the correlations patterns in training dataset, the model predicts a set of customers likely to leave the institution.

[14]:
```
# Analysing ANN classifier accuracy

cm = confusion_matrix(y_test, y_predicted) # Summarises model prediction␣
↪accuracy
cm
```

[14]:
```
array([[1544,   51],
       [ 238,  167]])
```

0.0.2 According to the convolutional matrix, the model made 1711 correct predictions out of 2000

[ ]:
```
# Further analysis and improvements (Computationally heavy section)

accuracy = cross_val_score(estimator = classifier, X = X_train, y = y_train, cv␣
↪= 10, n_jobs = -1) # 10 fold cross validation
mean = accuracy.mean()
variance = accuracy.std() # Mean and variance of accuracies



# Improving ANN classifier by tuning of parameters

parameters = {'batch_size': [25, 32],
              'epochs': [100, 500],
              'optimizer': ['adam', 'rmsprop']}

grid_search = GridSearchCV(estimator = classifier,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 10) # Test ANN over grid of parameters (batch␣
↪sizes, training alg.) defined above

grid_search = grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_ # Best parameters placed in array
```

```
best_accuracy = grid_search.best_score_ # Best accuracy score obtained with␣
↪parameters from best_parameters
```