kaggle Search Competitions Datasets Notebooks Discussion Courses •••

Bank Customer Classification

Given a dataset consisiting of Bank Customer information, we are asked to build a classifier which will tell us if a customer will exit the bank or not.

In [1]:

```
import theano
import tensorflow
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import os
print(os.listdir("../input"))
```

```
/opt/conda/lib/python3.6/site-packages/h5
py/__init__.py:36: FutureWarning: Convers
ion of the second argument of issubdtype
from `float` to `np.floating` is deprecat
ed. In future, it will be treated as `np.
float64 == np.dtype(float).type`.
  from ._conv import register_converters
as _register_converters
```

```
['Churn_Modelling.csv']
```

Data Preprocessing

In this dataset, we have to consider which of the factors may play a role in someone exiting a bank. To do that we must look at all the column and infer whether it will matter in classifying a new customer or not. The information about a customer is entailed in columns 0 through 12 (RowNumber-EstimatedSalary), while the output (whether the customer exited or not) is stored in the 13th row (Exited).

For as much as we care, neither the customer ID, nor the surname should matter in classification. Therefore, we will use columns 3 (CreditScore) inclusive through the 13th column (exclusive).

In [2]:

```
#importing the dataset
dataset = pd.read_csv('../input/Churn_Modelling.csv')
X = dataset.iloc[:,3:13].values # Credit Score through
Estimated Salary
y = dataset.iloc[:, 13].values # Exited
```

In [3]:

```
# Encoding categorical (string based) data. Country: th
ere are 3 options: France, Spain and Germany
# This will convert those strings into scalar values fo
r analysis
print(X[:8,1], '... will now become: ')
from sklearn.preprocessing import LabelEncoder, OneHot
Encoder
label_X_country_encoder = LabelEncoder()
X[:,1] = label_X_country_encoder.fit_transform(X[:,1])
print(X[:8,1])
```

```
['France' 'Spain' 'France' 'France' 'Spai
n' 'Spain' 'France' 'Germany'] ... will n
ow become:
[0 2 0 0 2 2 0 1]
```

In [4]:

```
# We will do the same thing for gender. this will be bi
nary in this dataset
print(X[:6,2], '... will now become: ')
from sklearn.preprocessing import LabelEncoder, OneHot
Encoder
label_X_gender_encoder = LabelEncoder()
X[:,2] = label_X_gender_encoder.fit_transform(X[:,2])
print(X[:6,2])
```

```
['Female' 'Female' 'Female' 'Female' 'Fem
ale' 'Male'] ... will now become:
[0 0 0 0 0 1]
```

The Problem here is that we are treating the countries as one variable with ordinal values (0 < 1 < 2). Therefore, one way to get rid of that problem is to split the countries into respective dimensions. that is,

Country	->	Country	->	Spain	France	Germany
Spain	->	0	->	1	0	0
France	->	1	->	0	1	0
France	->	1	->	0	1	0
Germany	->	2	->	0	0	1

Gender doesn't need to go through a similar process becasue it is binary

In [5]:

```
# Converting the string features into their own dimensi
ons. Gender doesn't matter here because its binary
countryhotencoder = OneHotEncoder(categorical_features
= [1]) # 1 is the country column
X = countryhotencoder.fit_transform(X).toarray()
```

In [6]:

```
X
```

Out[6]:

```
array([[1.0000000e+00, 0.0000000e+00, 0.0
000000e+00, ..., 1.0000000e+00,
        1.0000000e+00, 1.0134888e+05],
       [0.0000000e+00, 0.0000000e+00, 1.0
000000e+00, ..., 0.0000000e+00,
        1.0000000e+00, 1.1254258e+05],
       [1.0000000e+00, 0.0000000e+00, 0.0
000000e+00, ..., 1.0000000e+00,
        0.0000000e+00, 1.1393157e+05],
       ...,
       [1.0000000e+00, 0.0000000e+00, 0.0
000000e+00, ..., 0.0000000e+00,
        1.0000000e+00, 4.2085580e+04],
       [0.0000000e+00, 1.0000000e+00, 0.0
000000e+00, ..., 1.0000000e+00,
        0.0000000e+00, 9.2888520e+04],
       [1.0000000e+00, 0.0000000e+00, 0.0
000000e+00, ..., 1.0000000e+00,
        0.0000000e+00, 3.8190780e+04]])
```

You can now see that the first three columns represent the three countries that constituted the "country" category. We can now observe that we essentially only need two columns: a 0 on two countries means that the country has to be the one variable which wasn't included. This will save us from the problem of using too many dimensions

Spain	France	Germany	->	France	Germany

1	0	0	->	0	0
0	1	0	->	1	0
0	1	0	->	1	0
0	0	1	->	0	1

In [7]:

```
X = X[:,1:] # Got rid of Spain as a dimension. It is st
ill there through out inferences
```

In [8]:

```
# Splitting the dataset into the Training and Testing s
et.
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 = 0)
```

Feature scaling is a method used to standardize the range of independent variables or features of data. It is basically scaling all the dimensions to be even so that one independent variable does not dominate another. For example, bank account balance ranges from millions to 0, whereas gender is either 0 or 1. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance.

In [9]:

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

In [10]:

```
X_train
```

Out[10]:

```
array([[-0.5698444 ,  1.74309049,  0.1695
8176, ...,  0.64259497,
        -1.03227043,  1.10643166],
       [ 1.75486502, -0.57369368, -2.3045
5945, ...,  0.64259497,
         0.9687384 , -0.74866447],
       [-0.5698444 , -0.57369368, -1.1911
9591, ...,  0.64259497,
        -1.03227043,  1.48533467],
       ...,
       [-0.5698444 , -0.57369368,  0.9015
152 , ...,  0.64259497,
        -1.03227043,  1.41231994],
       [-0.5698444 ,  1.74309049, -0.6242
0521, ...,  0.64259497,
         0.9687384 ,  0.84432121],
       [ 1.75486502, -0.57369368, -0.2840
1079, ...,  0.64259497,
        -1.03227043,  0.32472465]])
```

END OF PREPROCESSING

Making the ANN

In [11]:

```
import keras # Test out Theano when time permits as wel
l
```

```
Using TensorFlow backend.
```

In [12]:

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

In [13]:

```
# Initializing the ANN
classifier = Sequential()
```

A hurestic tip is that the amount of nodes (dimensions) in your hidden layer should be the average of your input and output layers, which means that since we have 11 dimensions (representing **Independent variables** Note: Countries still compose only **one** dimension) and we are looking for a binary output, we calculate this to be $(11 + 1) \div 2 = 6$.

The breakdown of the inputs for the first layer is as follows:

activiation: **relu** becasue we are in an input layer. uses the ReLu activation function for ϕ

input_dim: **11** because we span 11 dimensions in our input layer. This is needed for the first added layer. The subsequent layers's input dimensions can be inferred using the previously added layer's output dimension. The next hidden layer will know what to expect.

units: **6** nodes (number of nodes in hidden layer). Can think of this as number of nodes are in the next layer.

kernel_initializer: **uniform** the distribution with which we randomly initialize weights for the nodes in this layer.

In [14]:

```
# This adds the input layer (by specifying input dimens
ion) AND the first hidden layer (units)
classifier.add(Dense(activation = 'relu', input_dim =
11, units=6, kernel_initializer='uniform'))
```

We are going to add another layer to this model because we want to implement Deep Learning, which is an artificial Neural network with many layers.

We will make our second hidden layer also have 6 nodes, just playing with the same arithmetic we used to determine the dimensions of the first hidden layer (average of your input and output layers) $(11 + 1) \div 2 = 6$.

In [15]:

```
# Adding the second hidden layer
# Notice that we do not need to specify input dim.
classifier.add(Dense(activation = 'relu', units=6, ker
nel_initializer='uniform'))
```

Adding the output layer

The breakdown of the inputs for the output layer is as follows:

activiation: **sigmoid** becasue we are in an output layer. uses the Sigmoid activation function for ϕ. This is used instead of the ReLu function becasue it generates probabilities for the outcome. We want the probability that each customer leaves the bank.

input_dim: **11** because we span 11 dimensions in our input layer. This is needed for the first added layer. The subsequent layers's input dimensions can be inferred using the previously added layer's output dimension. The next hidden layer will know what to expect.

units: **6** nodes (number of nodes in hidden layer). Can think of this as number of nodes are in the next layer.

kernel_initializer: **uniform** the distribution with which we randomly initialize weights for the nodes in this layer.

In [16]:

```
# Adding the output layer
# Notice that we do not need to specify input dim.
# Notice that we do not need to specify input dim.
# we have an output of 1 node, which is the the desired
dimensions of our output (stay with the bank or not)
# We use the sigmoid because we want probability outcom
es
classifier.add(Dense(activation = 'sigmoid', units=1,
kernel_initializer='uniform'))
```

If we want more than two categories, then we will need to change

1) the *units* parameter to match the desired category count

2) the *activation* field to **softmax**. Basically a sigmoid function but applied to a dependent variable that has more than 2 categories.

Compiling the Neural Network

Basically applying Stochastic Gradient descent on the whole Neural Network. We are Tuning the individual weights on each neuron.

The breakdown of the inputs for compiling is as follows:

optimizer: **adam** The algorithm we want to use to find the optimal set of weights in the neural networks. Adam is a very efficeint variation of Stochastic Gradient Descent.

loss: **binary_crossentropy** This is the loss function used within adam. This should be the logarthmic loss. If our dependent (output variable) is *Binary*, it is **binary_crossentropy**. If *Categorical*, then it is called **categorical_crossentropy**

metrics: **[accuracy]** The accuracy metrics which will be evaluated(minimized) by the model. Used as accuracy criteria to imporve model performance.

kernel_initializer: **uniform** the distribution with which we randomly initialize weights for the nodes in this layer.

In [17]:

```
classifier.compile(optimizer='adam', loss = 'binary_cr
ossentropy', metrics=['accuracy'])
```

Fitting the Neural Network

This is where we will be fitting the ANN to our training set.

The breakdown of the inputs for compiling is as follows:

X_train The independent variable portion of the data which needs to be fitted with the model.

Y_train The output portion of the data which the model needs to produce after fitting.

batch_size: How often we want to back-propogate the error values so that individual node weights can be adjusted.

nb_epochs: The number of times we want to run the entire test data over again to tune the weights. This is like the fuel of the algorithm.

In [18]:

```
classifier.fit(X_train, y_train, batch_size=10, epochs
=100)
```

```
Epoch 1/100
8000/8000 [=============================
=] - 2s 246us/step - loss: 0.4831 - acc:
0.7960
Epoch 2/100
8000/8000 [=============================
=] - 2s 217us/step - loss: 0.4299 - acc:
0.7960
Epoch 3/100
8000/8000 [=============================
=] - 2s 202us/step - loss: 0.4257 - acc:
0.7960
Epoch 4/100
8000/8000 [=============================
=] - 2s 191us/step - loss: 0.4215 - acc:
0.8107
Epoch 5/100
8000/8000 [=============================
=] - 2s 219us/step - loss: 0.4190 - acc:
0.8214
Epoch 6/100
8000/8000 [=============================
=] - 2s 219us/step - loss: 0.4171 - acc:
0.8240
Epoch 7/100
8000/8000 [=============================
=] - 2s 224us/step - loss: 0.4152 - acc:
0.8277
Epoch 8/100
8000/8000 [=============================
=] - 2s 229us/step - loss: 0.4135 - acc:
0.8296
Epoch 9/100
8000/8000 [=============================
=] - 2s 238us/step - loss: 0.4121 - acc:
0.8301
Epoch 10/100
8000/8000 [=============================
8000/8000 [
=] - 2s 223us/step - loss: 0.4111 - acc:
0.8316
Epoch 11/100
8000/8000 [=============================
=] - 2s 229us/step - loss: 0.4105 - acc:
0.8321
Epoch 12/100
8000/8000 [=============================
=] - 2s 206us/step - loss: 0.4091 - acc:
0.8334
Epoch 13/100
8000/8000 [=============================
=] - 2s 210us/step - loss: 0.4089 - acc:
0.8337
Epoch 14/100
8000/8000 [=============================
=] - 2s 229us/step - loss: 0.4085 - acc:
0.8339
Epoch 15/100
8000/8000 [=============================
=] - 2s 227us/step - loss: 0.4076 - acc:
0.8345
Epoch 16/100
8000/8000 [=============================
=] - 2s 219us/step - loss: 0.4076 - acc:
0.8345
Epoch 17/100
8000/8000 [=============================
=] - 2s 218us/step - loss: 0.4065 - acc:
0.8351
Epoch 18/100
8000/8000 [=============================
=] - 2s 212us/step - loss: 0.4061 - acc:
0.8327
Epoch 19/100
8000/8000 [=============================
=] - 2s 199us/step - loss: 0.4049 - acc:
0.8335
Epoch 20/100
8000/8000 [=============================
=] - 1s 187us/step - loss: 0.4032 - acc:
0.8345
Epoch 21/100
8000/8000 [=============================
=] - 2s 192us/step - loss: 0.4021 - acc:
0.8341
Epoch 22/100
8000/8000 [=============================
=] - 2s 207us/step - loss: 0.4015 - acc:
0.8339
Epoch 23/100
8000/8000 [=============================
=] - 2s 210us/step - loss: 0.4004 - acc:
0.8360
Epoch 24/100
8000/8000 [=============================
=] - 2s 221us/step - loss: 0.4000 - acc:
0.8357
```

```
Epoch 25/100
8000/8000 [==============================
=] - 2s 228us/step - loss: 0.3992 - acc:
0.8355
Epoch 26/100
8000/8000 [==============================
=] - 2s 234us/step - loss: 0.3983 - acc:
0.8362
Epoch 27/100
8000/8000 [==============================
=] - 2s 243us/step - loss: 0.3980 - acc:
0.8355
Epoch 28/100
8000/8000 [==============================
=] - 2s 239us/step - loss: 0.3974 - acc:
0.8354
Epoch 29/100
8000/8000 [==============================
=] - 2s 220us/step - loss: 0.3972 - acc:
0.8361
Epoch 30/100
8000/8000 [==============================
=] - 2s 228us/step - loss: 0.3964 - acc:
0.8356
Epoch 31/100
8000/8000 [==============================
=] - 2s 230us/step - loss: 0.3968 - acc:
0.8364
Epoch 32/100
8000/8000 [==============================
=] - 2s 216us/step - loss: 0.3958 - acc:
0.8345
Epoch 33/100
8000/8000 [==============================
=] - 2s 218us/step - loss: 0.3965 - acc:
0.8364
Epoch 34/100
8000/8000 [==============================
=] - 2s 214us/step - loss: 0.3956 - acc:
0.8360
Epoch 35/100
8000/8000 [==============================
=] - 2s 242us/step - loss: 0.3954 - acc:
0.8344
Epoch 36/100
8000/8000 [==============================
=] - 2s 235us/step - loss: 0.3953 - acc:
0.8371
Epoch 37/100
8000/8000 [==============================
=] - 2s 242us/step - loss: 0.3950 - acc:
0.8366
Epoch 38/100
8000/8000 [==============================
=] - 2s 236us/step - loss: 0.3953 - acc:
0.8379
Epoch 39/100
8000/8000 [==============================
=] - 2s 243us/step - loss: 0.3947 - acc:
0.8365
Epoch 40/100
8000/8000 [=============================
=] - 2s 226us/step - loss: 0.3945 - acc:
0.8374
Epoch 41/100
8000/8000 [=============================
=] - 2s 234us/step - loss: 0.3947 - acc:
0.8360
Epoch 42/100
8000/8000 [=============================
=] - 2s 241us/step - loss: 0.3942 - acc:
0.8376
Epoch 43/100
8000/8000 [=============================
=] - 2s 253us/step - loss: 0.3946 - acc:
0.8364
Epoch 44/100
8000/8000 [=============================
=] - 2s 251us/step - loss: 0.3944 - acc:
0.8380
Epoch 45/100
8000/8000 [=============================
=] - 2s 251us/step - loss: 0.3938 - acc:
0.8357
Epoch 46/100
8000/8000 [=============================
=] - 2s 250us/step - loss: 0.3932 - acc:
0.8356
Epoch 47/100
8000/8000 [=============================
=] - 2s 243us/step - loss: 0.3932 - acc:
0.8389
Epoch 48/100
8000/8000 [=============================
=] - 2s 216us/step - loss: 0.3924 - acc:
0.8371
Epoch 49/100
8000/8000 [=============================
=] - 2s 206us/step - loss: 0.3910 - acc:
0.8370
Epoch 50/100
8000/8000 [=============================
=] - 2s 210us/step - loss: 0.3895 - acc:
0.8387
Epoch 51/100
8000/8000 [=============================
=] - 2s 201us/step - loss: 0.3864 - acc:
0.8370
Epoch 52/100
8000/8000 [=============================
=] - 2s 202us/step - loss: 0.3839 - acc:
0.8382
Epoch 53/100
8000/8000 [=============================
=] - 2s 212us/step - loss: 0.3818 - acc:
0.8387
```

```
Epoch 54/100
8000/8000 [==============================
=] - 2s 214us/step - loss: 0.3798 - acc:
0.8379
Epoch 55/100
8000/8000 [==============================
=] - 2s 220us/step - loss: 0.3780 - acc:
0.8381
Epoch 56/100
8000/8000 [==============================
=] - 2s 222us/step - loss: 0.3765 - acc:
0.8384
Epoch 57/100
8000/8000 [==============================
=] - 2s 219us/step - loss: 0.3767 - acc:
0.8370
Epoch 58/100
8000/8000 [==============================
=] - 2s 227us/step - loss: 0.3755 - acc:
0.8396
Epoch 59/100
8000/8000 [==============================
=] - 2s 205us/step - loss: 0.3752 - acc:
0.8379
Epoch 60/100
8000/8000 [==============================
=] - 2s 213us/step - loss: 0.3745 - acc:
0.8365
Epoch 61/100
8000/8000 [==============================
=] - 2s 221us/step - loss: 0.3742 - acc:
0.8377
Epoch 62/100
8000/8000 [==============================
=] - 2s 233us/step - loss: 0.3734 - acc:
0.8379
Epoch 63/100
8000/8000 [==============================
=] - 2s 260us/step - loss: 0.3722 - acc:
0.8387
Epoch 64/100
8000/8000 [==============================
=] - 2s 265us/step - loss: 0.3721 - acc:
0.8387
Epoch 65/100
8000/8000 [==============================
=] - 2s 272us/step - loss: 0.3702 - acc:
0.8407
Epoch 66/100
8000/8000 [==============================
=] - 2s 260us/step - loss: 0.3695 - acc:
0.8422
Epoch 67/100
8000/8000 [==============================
=] - 2s 233us/step - loss: 0.3679 - acc:
0.8455
Epoch 68/100
8000/8000 [==============================
=] - 2s 212us/step - loss: 0.3662 - acc:
0.8441
```

ANN Bank Customer Classification

Python notebook using data from Bank Customer Churn Modeling · 3,878 views · 2y ago

17 Copy and Edit 44

Version 4
4 commits

Did you find this Kernel useful?
Show your appreciation with an upvote

17

Data

Data Sources

Bank Customer Churn Modeling
Churn_Modelling.csv 14 columns

Bank Customer Churn Modeling

Can you predict if bank customers will turnover next cycle?

Last Updated: 2 years ago (Version 1)

About this Dataset

No description yet

Comments (2)

Sort by
All Comments Hotness

Click here to comment...

malaka123 • Posted on Latest Version • 3 months ago • Options • Reply 1

the data is strongly unbalanced, this could lead to problems when predicting data
you could use methods like SMOTE oversample the dataset

WalterMehra • Posted on Latest Version • a year ago • Options • Reply 1

Great job describing each step, this is a very good tutorial.

Notebook Data Comments