The Pipeline

1. Question & required data
2. Acquire the data
3. Data Analysis
4. Prepare the data for the Deep Learning Model
5. Building & training the model
6. Testing the model
7. Interpreting the model
8. Improving the model

Problem introduction: predicting whether a bank customer will leave the bank

Question & required data

The dataset consists of a bunch of features describing the customers of a bank. The question that is posed is that, given these features could we detect some patterns that allow us to predict if a customer is about to leave a bank?

It would then be possible to direct marketing efforts to the customers if it would be beneficial to retain them.

ML problem characteristics

1. Task: **classification**
2. Attribute Types: **Categorical, continuous and binary**
3. **14** attributes
4. **10k** instances
5. ML model: **Deep neural net**
6. Deep learning library: **Keras (Tensorflow)**
7. Hyperparameter tuning: **Talos**

The feature vector

1. *RowNumber*
2. *CustomerId:* unique id to identify a customer
3. *Surname*
4. *CreditScore:* a creditscore compiled by the bank for each customer
5. *Geography*
6. *Gender*
7. *Age*
8. *Tenure:* how long has the customer been with the bank
9. *Balance*
10. *NumOfProducts:* how many products of the bank (credit/debit cards etc.) does the customer have
11. *HasCrCard:* does the customer have a credit card
12. *IsActiveMember:* is the customer an active member of the bank
13. *EstimatedSalary:* the customer's estimated salary

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from collections import Counter

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dropout, Dense
import talos

sns.set()
tf.__version__
```

```
Using TensorFlow backend.
```

Out[1]: '2.0.0'

In [2]:
```
data = pd.read_csv('dataset/exit_prediction.csv')
print('Our dataset holds {} records and {} attributes'.format(*data.shape))
data.head()
```

```
Our dataset holds 10000 records and 14 attributes
```

Out[2]:

	RowNumber	CustomerId	Surname	CreditScore	Geography	Gender	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMe
0	1	15634602	Hargrave	619	France	Female	42	2	0.00	1	1	
1	2	15647311	Hill	608	Spain	Female	41	1	83807.86	1	0	
2	3	15619304	Onio	502	France	Female	42	8	159660.80	3	1	
3	4	15701354	Boni	699	France	Female	39	1	0.00	2	0	
4	5	15737888	Mitchell	850	Spain	Female	43	2	125510.82	1	1	

RowNumber, *CustomerId* & *Surname* won't actually be useful for the deep learning model that we will build in the end so we can just go ahead and drop them.

In [3]:
```
data.drop(labels=['RowNumber', 'CustomerId', 'Surname'], axis=1, inplace=True)
```

Data Analysis

1. Tidy Data: by default
2. **Acquaintance with the data**
3. Outliers
4. Missing Data
5. Class-Imbalance problem
6. Feature correlation

Acquaintance with the data

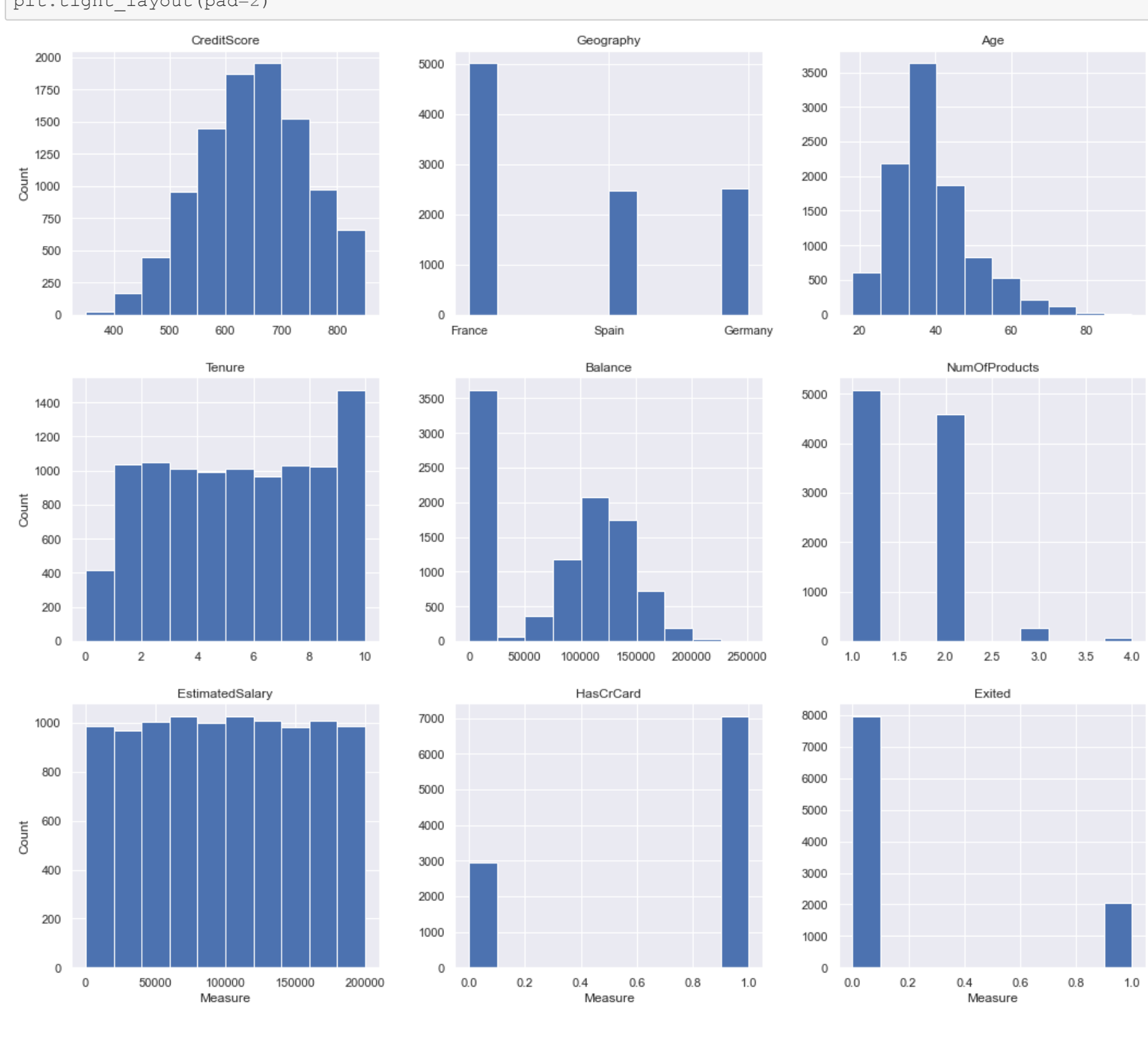
Lets visualize a subset of the feature vector for this purpose

In [4]:
```
plt.figure(figsize=(15,13))
selected_features = data[['CreditScore', 'Geography', 'Age', 'Tenure', 'Balance', 'NumOfProducts',
                                                            'EstimatedSalary', 'HasCrCard',
'Exited']]

for i in range(1, selected_features.shape[1]+1):
    plt.subplot(3,3,i)
    figure=plt.gca()
    figure.set_title(selected_features.columns[i-1])

    plt.hist(selected_features.iloc[:, i-1])
    if selected_features.columns[i-1] in ['EstimatedSalary', 'HasCrCard', 'Exited']:
        plt.xlabel('Measure')
    if selected_features.columns[i-1] in ['CreditScore', 'Tenure', 'EstimatedSalary']:
        plt.ylabel('Count')

plt.tight_layout(pad=2)
```



Outliers

It appears that we have no outliers. This observation is also confirmed by the above plot

In [5]:
```
data.describe()
```

Out[5]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	E
count	10000.000000	10000.000000	10000.000000	10000.000000	10000.000000	10000.00000	10000.000000	10000.000000	10000.0
mean	650.528800	38.921800	5.012800	76485.889288	1.530200	0.70550	0.515100	100090.239881	0.2
std	96.653299	10.487806	2.892174	62397.405202	0.581654	0.45584	0.499797	57510.492818	0.4
min	350.000000	18.000000	0.000000	0.000000	1.000000	0.00000	0.000000	11.580000	0.0
25%	584.000000	32.000000	3.000000	0.000000	1.000000	0.00000	0.000000	51002.110000	0.0
50%	652.000000	37.000000	5.000000	97198.540000	1.000000	1.00000	1.000000	100193.915000	0.0
75%	718.000000	44.000000	7.000000	127644.240000	2.000000	1.00000	1.000000	149388.247500	0.0
max	850.000000	92.000000	10.000000	250898.090000	4.000000	1.00000	1.000000	199992.480000	1.0

Missing Data

Our dataset seems to have no missing values. We cannot call the values of the *Balance = 0* as missing values since this can be a value that the balance can indeed take.

For an approach of taking care of missing values see the other project on Autism Predictor

In [6]:
```
print('Does our dataset contain missing values?', data.isnull().any().any())
```

```
Does our dataset contain missing values? False
```

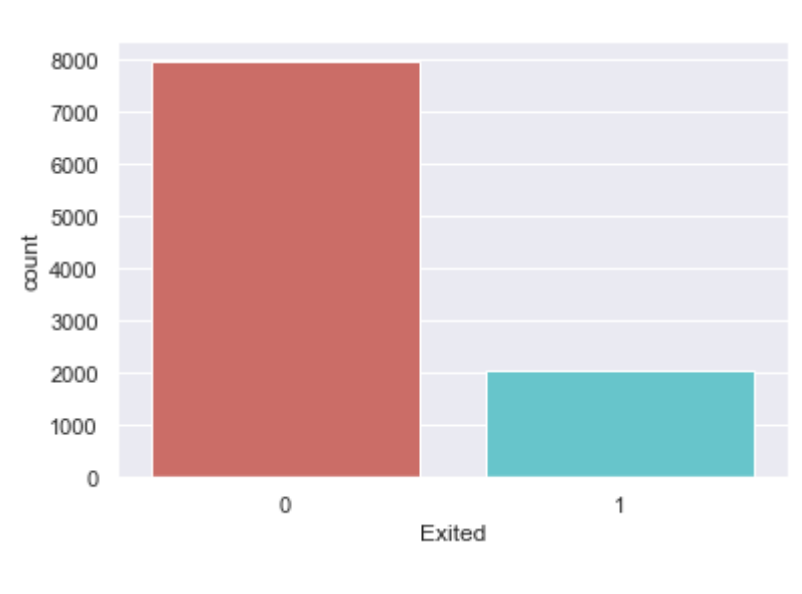
Class-Imbalance problem

Our data seems indeed to have imbalanced classes. Obviously, most of the people did not leave the bank.

In [7]:
```
print(data['Exited'].value_counts()/data.shape[0])

sns.countplot(x='Exited', data=data, palette='hls'); plt.show()
```

```
0    0.7963
1    0.2037
Name: Exited, dtype: float64
```



Are the classes well separated?

If our classes are well separated, then the class-imbalance problem is not really a problem. We can simply visualize a couple of the attributes for this purpose.

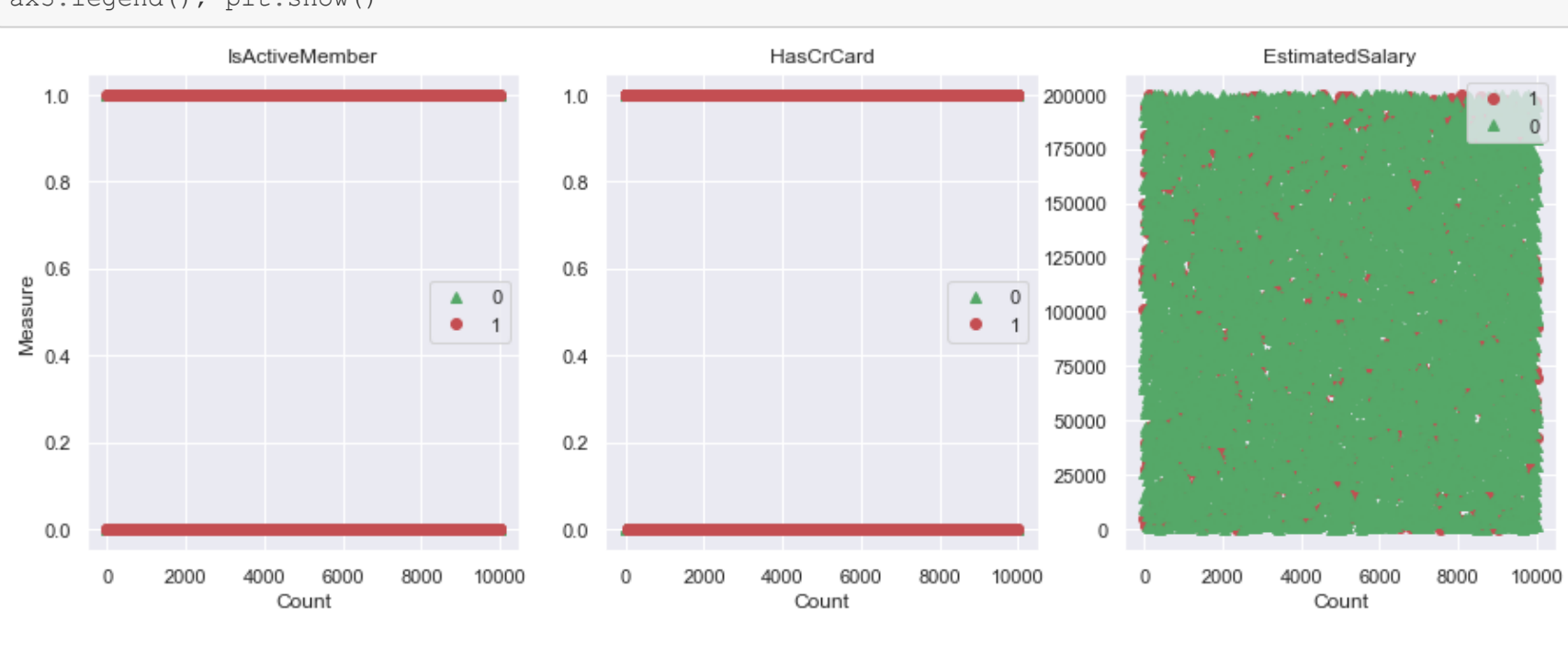
In [8]:
```
fig, ((ax1, ax2, ax3)) = plt.subplots(nrows=1, ncols=3, figsize=(15,5))

att1, att2, att3 = np.random.choice(data.columns.delete([13]), 3, replace=False)

att1_data = list(data[att1].groupby(data['Exited']))
ax1.plot(att1_data[0][1], 'g^', label=att1_data[0][0])
ax1.plot(att1_data[1][1], 'ro', label=att1_data[1][0])
ax1.set_xlabel('Count'); ax1.set_ylabel('Measure'); ax1.set_title(att1_data[0][1].name)
ax1.legend()

att2_data = list(data[att2].groupby(data['Exited']))
ax2.plot(att2_data[0][1], 'g^', label=att2_data[0][0])
ax2.plot(att2_data[1][1], 'ro', label=att2_data[1][0])
ax2.set_xlabel('Count'); ax2.set_ylabel(''); ax2.set_title(att2_data[0][1].name)
ax2.legend()

att3_data = list(data[att3].groupby(data['Exited']))
ax3.plot(att3_data[1][1], 'ro', label=att3_data[1][0])
ax3.plot(att3_data[0][1], 'g^', label=att3_data[0][0])
ax3.set_xlabel('Count'); ax3.set_ylabel(''); ax3.set_title(att3_data[0][1].name)
ax3.legend(); plt.show()
```



It seems that our classes are not well separated. There are techniques to fight the class-imbalance problem but in this case we are just going to continue like this.

One-Hot Encoding

Convert categorical data to numerical. We have two categorical features in our case: *geography* & *gender*

In [9]:
```
data = pd.get_dummies(data)
data.head(2)
```

Out[9]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Exited	Geography_France	Geogra
0	619	42	2	0.00	1	1	1	101348.88	1	1	
1	608	41	1	83807.86	1	0	1	112542.58	0	0	

Feature correlation

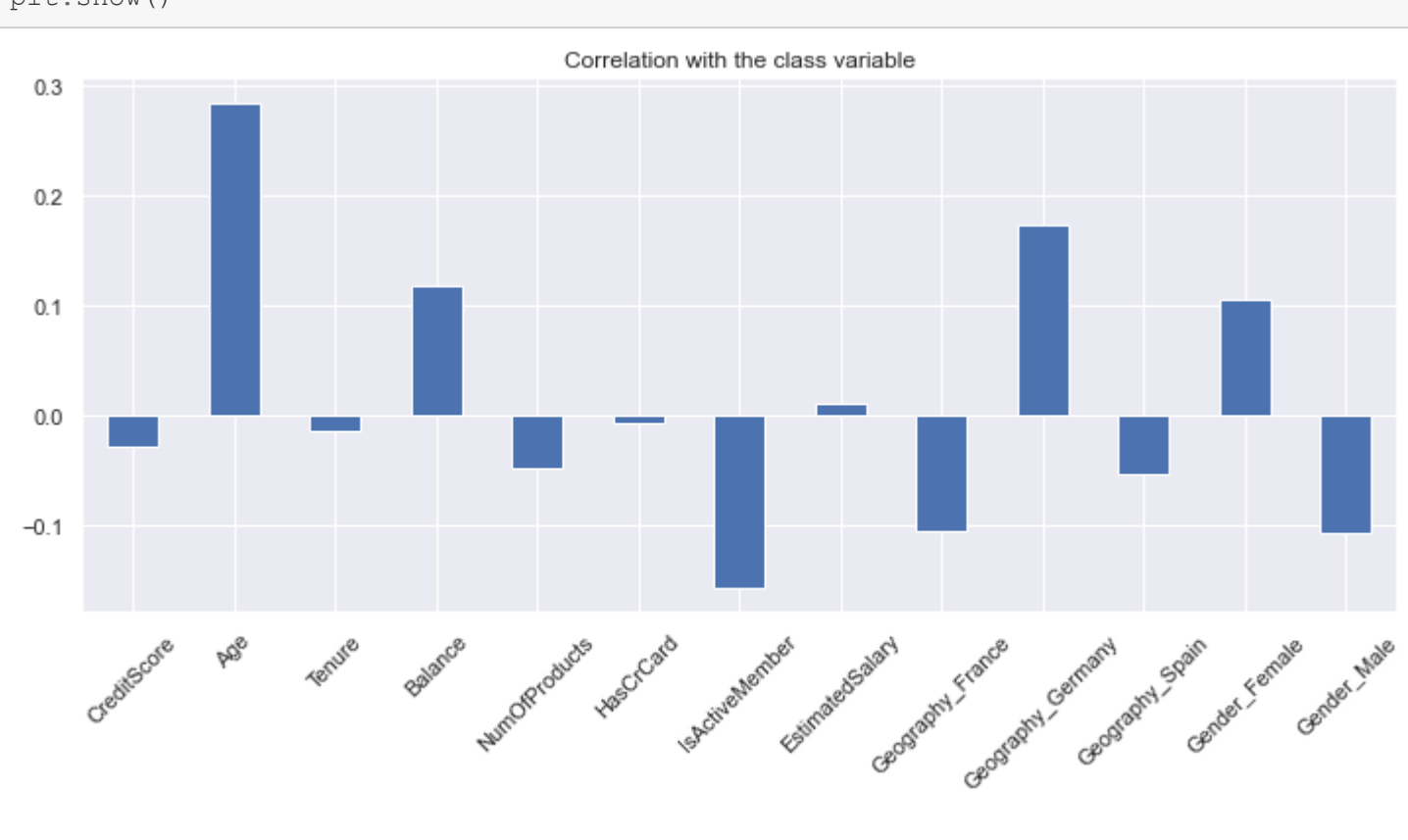
We want the features to be independent of each-other. Most ML algorithms work on this assumption.

Correlation of features with the class label

This will also give us an impression into which variable is the more important for predicting the class variable.

None of the attributes seems to be significantly correlated with the class.

In [10]:
```
data.drop('Exited', axis=1).corrwith(data['Exited']).plot.bar(figsize=(12,5),
                                              title='Correlation with the class variable', rot=45, fontsi
ze=11)
plt.show()
```



We can also check the influence of the attributes on the class by aggregating the mean values after grouping by the class label. The below result confirms the correlation plot above.

In [11]:
```
class_group = data.groupby('Exited')
print(class_group.agg(np.mean))
```

```
        CreditScore        Age    Tenure       Balance  NumOfProducts  \
Exited
0        651.853196  37.408389  5.033279  72745.296779       1.544267
1        645.351497  44.837997  4.932744  91108.539337       1.475209

        HasCrCard  IsActiveMember  EstimatedSalary  Geography_France  \
Exited
0        0.707146        0.554565     99738.391772          0.527942
1        0.699067        0.360825    101465.677531          0.397644

        Geography_Germany  Geography_Spain  Gender_Female  Gender_Male
Exited
0                0.212859         0.259199       0.427477     0.572523
1                0.399607         0.202749       0.559156     0.440844
```

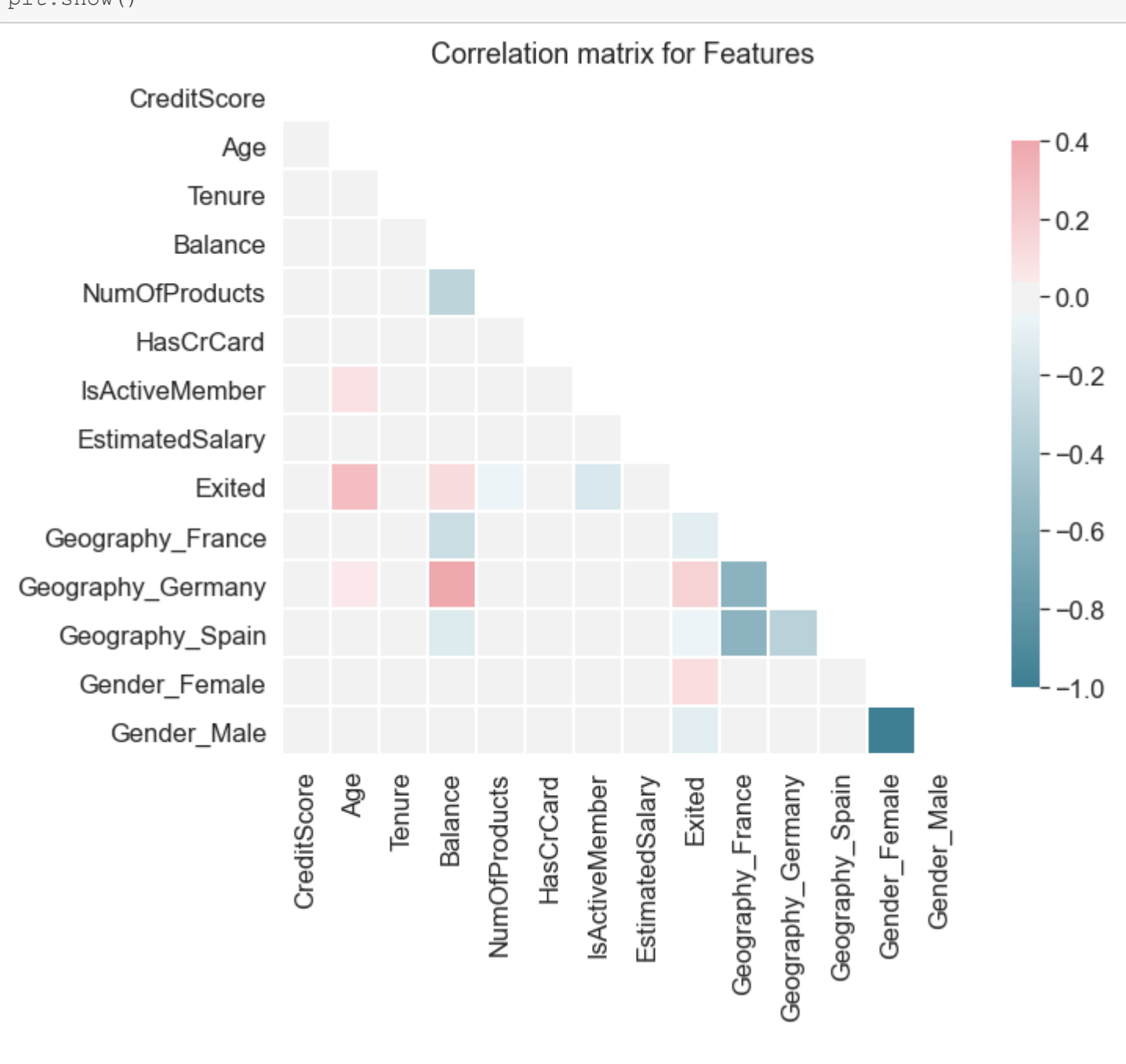
Correlation of features with each-other

In [12]:
```
sns.set(style='white', font_scale=1.5)
corr_matrix = data.corr()

# create a mask for the upper triangle so that we can ignore it when later building the heatmap
mask = np.zeros_like(corr_matrix, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# color map for the different values of the correlation matrix
cmap = sns.diverging_palette(220, 10, as_cmap=True)

plt.figure(figsize=(11,8))
plt.title('Correlation matrix for Features')
sns.heatmap(corr_matrix, square=True, mask=mask, cmap=cmap, center=0, linewidths=1.0, cbar_kws={'shrin
k':0.8})
plt.show()
```



We can see some correlations, like for instance the negative correlation between the number of products and the balance but they are less than 0.3 and so do not pose any problems for the algorithm. We may further proceed like this.

Prepare the data for the Neural Net Model

1. Class & Features separation
2. Train/Test split
3. Scaling/Standardization if needed

Class & Feature separation

In [13]:
```
labels = data.pop('Exited')
features = data
print('Shape of features', features.shape)
print('Shape of labels', labels.shape)
features.head(2)
```

```
Shape of features (10000, 13)
Shape of labels (10000,)
```

Out[13]:

	CreditScore	Age	Tenure	Balance	NumOfProducts	HasCrCard	IsActiveMember	EstimatedSalary	Geography_France	Geography_Ge
0	619	42	2	0.00	1	1	1	101348.88	1	
1	608	41	1	83807.86	1	0	1	112542.58	0	

Train/Test split

In [14]:
```
train_features, test_features, train_labels, test_labels = train_test_split(features, labels, test_size
=0.3,
                                                                            random_state=1, stratify=la
bels)
print('Shape of training data', train_features.shape)
print('Shape of testing data', test_features.shape)
```

```
Shape of training data (7000, 13)
Shape of testing data (3000, 13)
```

Feature Scaling

In [15]:
```
scaler = StandardScaler()
to_scale = ['CreditScore', 'Age', 'Balance', 'EstimatedSalary']

train_features_sc = pd.DataFrame(scaler.fit_transform(train_features[to_scale]))
train_features_sc.columns = train_features[to_scale].columns
train_features_sc.index = train_features[to_scale].index.values
train_features[to_scale] = train_features_sc

test_features_sc = pd.DataFrame(scaler.fit_transform(test_features[to_scale]))
test_features_sc.columns = test_features[to_scale].columns
test_features_sc.index = test_features[to_scale].index.values
test_features[to_scale] = test_features_sc
```

Build & train the Neural Net model

We shall first build a basic Neural Net model using Keras. On the next chapter we shall see how to tune its parameters using talos in a bid to improve the model's performance.

Build the Deep Neural Net

We can instantiate different aspects of the neural net either by passing a string identifier (and in this case the default parameters for the optimizer can be used) or by building an object from the class. The latter should be the used way as it gives us the possibility to tweak stuff.

Docs on the dense layers

Docs for the compile method

Docs on optimizers

Docs on the loss function

- *from_logits=True*: will significantly boost testing accuracy at a cost of training
- *label_smoothing=1*: can be played with in the range of [0,1]
- *reduction*: different options include, 'sum', keras.losses.Reduction.NONE

Docs on the fit method

1. *class_weight*: can be useful to tell the model to "pay more attention" to samples from an under-represented class

In [16]:
```
ann = Sequential()
ann.add(Dense(units=100, activation='relu', use_bias=True))
ann.add(Dense(units=64, activation='relu', use_bias=True))
ann.add(Dense(units=32, activation='relu', use_bias=True))
ann.add(Dense(units=32, activation='relu', use_bias=True))

# Use sigmoid in the final layer since we have a binary classification problem
ann.add(Dense(units=1, activation='sigmoid', use_bias=False))

optimizer = keras.optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)

# binary cross entropy as loss since we have a binary classification problem
loss = keras.losses.BinaryCrossentropy(name='binary_crossentropy', from_logits=False,
                                                                            label_smoothing=0.05, reduc
tion='auto')

ann.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
```

In [17]:
```
def get_class_weights(labels):
    counter = Counter(labels)
    majority = max(counter.values())
    return {cls: float(majority/count) for cls, count in counter.items()}
```

In [18]:
```
c_weight = {0:1, 1:4}
ann.fit(x=train_features.values, y=train_labels.values, epochs=100, verbose=2, shuffle=True, batch_size
=70,
                 use_multiprocessing=True, )
```

```
Epoch 1/100
 - 0s - loss: 0.4943 - accuracy: 0.7957
Epoch 2/100
 - 0s - loss: 0.4509 - accuracy: 0.8199
Epoch 3/100
 - 0s - loss: 0.4327 - accuracy: 0.8319
Epoch 4/100
 - 0s - loss: 0.4137 - accuracy: 0.8451
Epoch 5/100
 - 0s - loss: 0.3913 - accuracy: 0.8564
Epoch 6/100
 - 0s - loss: 0.3835 - accuracy: 0.8594
Epoch 7/100
 - 0s - loss: 0.3798 - accuracy: 0.8631
Epoch 8/100
 - 0s - loss: 0.3745 - accuracy: 0.8644
Epoch 9/100
 - 0s - loss: 0.3718 - accuracy: 0.8680
Epoch 10/100
 - 0s - loss: 0.3694 - accuracy: 0.8684
Epoch 11/100
 - 0s - loss: 0.3673 - accuracy: 0.8706
Epoch 12/100
 - 0s - loss: 0.3657 - accuracy: 0.8706
Epoch 13/100
 - 0s - loss: 0.3638 - accuracy: 0.8707
Epoch 14/100
 - 0s - loss: 0.3618 - accuracy: 0.8726
Epoch 15/100
 - 0s - loss: 0.3603 - accuracy: 0.8743
Epoch 16/100
 - 0s - loss: 0.3574 - accuracy: 0.8750
Epoch 17/100
 - 0s - loss: 0.3544 - accuracy: 0.8750
Epoch 18/100
 - 0s - loss: 0.3550 - accuracy: 0.8744
Epoch 19/100
 - 0s - loss: 0.3510 - accuracy: 0.8784
Epoch 20/100
 - 0s - loss: 0.3511 - accuracy: 0.8784
Epoch 21/100
 - 0s - loss: 0.3501 - accuracy: 0.8774
Epoch 22/100
 - 0s - loss: 0.3465 - accuracy: 0.8816
Epoch 23/100
 - 0s - loss: 0.3445 - accuracy: 0.8807
Epoch 24/100
 - 0s - loss: 0.3441 - accuracy: 0.8836
Epoch 25/100
 - 0s - loss: 0.3423 - accuracy: 0.8823
Epoch 26/100
 - 0s - loss: 0.3430 - accuracy: 0.8816
Epoch 27/100
 - 0s - loss: 0.3383 - accuracy: 0.8833
Epoch 28/100
 - 0s - loss: 0.3402 - accuracy: 0.8833
Epoch 29/100
 - 0s - loss: 0.3367 - accuracy: 0.8824
Epoch 30/100
 - 0s - loss: 0.3374 - accuracy: 0.8859
Epoch 31/100
 - 0s - loss: 0.3314 - accuracy: 0.8883
Epoch 32/100
 - 0s - loss: 0.3310 - accuracy: 0.8896
Epoch 33/100
 - 0s - loss: 0.3275 - accuracy: 0.8897
Epoch 34/100
 - 0s - loss: 0.3249 - accuracy: 0.8930
Epoch 35/100
 - 0s - loss: 0.3272 - accuracy: 0.8890
Epoch 36/100
 - 0s - loss: 0.3235 - accuracy: 0.8904
Epoch 37/100
 - 0s - loss: 0.3256 - accuracy: 0.8923
Epoch 38/100
 - 0s - loss: 0.3226 - accuracy: 0.8941
Epoch 39/100
 - 0s - loss: 0.3184 - accuracy: 0.8949
Epoch 40/100
 - 0s - loss: 0.3163 - accuracy: 0.8949
Epoch 41/100
 - 0s - loss: 0.3154 - accuracy: 0.8967
Epoch 42/100
 - 0s - loss: 0.3135 - accuracy: 0.8971
Epoch 43/100
 - 0s - loss: 0.3110 - accuracy: 0.8977
Epoch 44/100
 - 0s - loss: 0.3088 - accuracy: 0.9009
Epoch 45/100
 - 0s - loss: 0.3061 - accuracy: 0.9004
Epoch 46/100
 - 0s - loss: 0.3070 - accuracy: 0.9007
Epoch 47/100
 - 0s - loss: 0.3050 - accuracy: 0.9029
Epoch 48/100
 - 0s - loss: 0.3013 - accuracy: 0.9049
Epoch 49/100
 - 0s - loss: 0.2992 - accuracy: 0.9017
Epoch 50/100
 - 0s - loss: 0.2969 - accuracy: 0.9049
Epoch 51/100
 - 0s - loss: 0.2942 - accuracy: 0.9063
Epoch 52/100
 - 0s - loss: 0.2949 - accuracy: 0.9066
Epoch 53/100
 - 0s - loss: 0.2924 - accuracy: 0.9113
Epoch 54/100
 - 0s - loss: 0.2913 - accuracy: 0.9077
Epoch 55/100
 - 0s - loss: 0.2884 - accuracy: 0.9121
Epoch 56/100
 - 0s - loss: 0.2855 - accuracy: 0.9134
Epoch 57/100
 - 0s - loss: 0.2848 - accuracy: 0.9147
Epoch 58/100
 - 0s - loss: 0.2827 - accuracy: 0.9157
Epoch 59/100
 - 0s - loss: 0.2803 - accuracy: 0.9183
Epoch 60/100
 - 0s - loss: 0.2792 - accuracy: 0.9191
Epoch 61/100
 - 0s - loss: 0.2780 - accuracy: 0.9170
Epoch 62/100
 - 0s - loss: 0.2740 - accuracy: 0.9214
Epoch 63/100
 - 0s - loss: 0.2751 - accuracy: 0.9209
Epoch 64/100
 - 0s - loss: 0.2682 - accuracy: 0.9221
Epoch 65/100
 - 0s - loss: 0.2704 - accuracy: 0.9239
Epoch 66/100
 - 0s - loss: 0.2667 - accuracy: 0.9229
Epoch 67/100
 - 0s - loss: 0.2687 - accuracy: 0.9207
Epoch 68/100
 - 0s - loss: 0.2662 - accuracy: 0.9260
Epoch 69/100
 - 0s - loss: 0.2646 - accuracy: 0.9237
Epoch 70/100
 - 0s - loss: 0.2574 - accuracy: 0.9281
Epoch 71/100
 - 0s - loss: 0.2578 - accuracy: 0.9287
Epoch 72/100
 - 0s - loss: 0.2611 - accuracy: 0.9267
Epoch 73/100
 - 0s - loss: 0.2573 - accuracy: 0.9300
Epoch 74/100
 - 0s - loss: 0.2528 - accuracy: 0.9337
Epoch 75/100
 - 0s - loss: 0.2533 - accuracy: 0.9330
Epoch 76/100
 - 0s - loss: 0.2536 - accuracy: 0.9347
Epoch 77/100
 - 0s - loss: 0.2524 - accuracy: 0.9310
Epoch 78/100
 - 0s - loss: 0.2519 - accuracy: 0.9314
Epoch 79/100
 - 0s - loss: 0.2510 - accuracy: 0.9326
Epoch 80/100
 - 0s - loss: 0.2432 - accuracy: 0.9357
Epoch 81/100
 - 0s - loss: 0.2439 - accuracy: 0.9367
Epoch 82/100
 - 0s - loss: 0.2402 - accuracy: 0.9416
Epoch 83/100
 - 0s - loss: 0.2407 - accuracy: 0.9406
Epoch 84/100
 - 0s - loss: 0.2464 - accuracy: 0.9361
Epoch 85/100
 - 0s - loss: 0.2376 - accuracy: 0.9370
Epoch 86/100
 - 0s - loss: 0.2354 - accuracy: 0.9436
Epoch 87/100
 - 0s - loss: 0.2359 - accuracy: 0.9407
Epoch 88/100
 - 0s - loss: 0.2318 - accuracy: 0.9421
Epoch 89/100
 - 0s - loss: 0.2304 - accuracy: 0.9453
Epoch 90/100
 - 0s - loss: 0.2299 - accuracy: 0.9451
Epoch 91/100
 - 0s - loss: 0.2253 - accuracy: 0.9490
Epoch 92/100
 - 0s - loss: 0.2307 - accuracy: 0.9447
Epoch 93/100
 - 0s - loss: 0.2227 - accuracy: 0.9481
Epoch 94/100
 - 0s - loss: 0.2210 - accuracy: 0.9510
Epoch 95/100
 - 0s - loss: 0.2249 - accuracy: 0.9474
Epoch 96/100
 - 0s - loss: 0.2247 - accuracy: 0.9480
Epoch 97/100
 - 0s - loss: 0.2196 - accuracy: 0.9513
Epoch 98/100
 - 0s - loss: 0.2196 - accuracy: 0.9511
Epoch 99/100
 - 0s - loss: 0.2178 - accuracy: 0.9514
Epoch 100/100
 - 0s - loss: 0.2176 - accuracy: 0.9514
```

Out[18]: <keras.callbacks.callbacks.History at 0x199ae503a08>

Test the Model

In [19]:
```
print('Shape of testing dataset', test_features.shape)
predictions = ((ann.predict(test_features, batch_size=70)) > 0.5)
print('Our models accuracy on test data: ', round(100*accuracy_score(test_labels, predictions), 2))
```

```
Shape of testing dataset (3000, 13)
Our models accuracy on test data:  83.87
```

Interpret the Model

Our Deep Neural Network performed good. It achieved an ~83% accuracy on the testing data. This was achieved only by utilizing a few hyperparameters of the neural network.

We shall next see if we can improve this number by tuning the various hyperparameters that a neural network can take. We shall do this using talos.