Tony_Submission

March 15, 2023

1 CHALLENGE #1: P>N

Required Depndencies:

```
[ ]: # pip install numpy
# pip install pandas
# pip install matplotlib
# pip install seaborn
# pip install scikit-learn
# pip install Jinja2
# pip install keras
# pip install tensorflow
# pip install statsmodels
```

1.1 Import libraries

```
[ ]: # Import your libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

1.2 Loading in the dataset

```
[ ]: df = pd.read_csv('data.csv')
df.head(10)
```

```
[ ]:    id  train  target_eval  var_1  var_2  var_3  var_4  var_5  var_6  var_7  \
0    1      1            1  0.422  0.521  0.493  0.206  0.144  0.203  0.709
1    2      1            0  0.345  0.974  0.330  0.643  0.931  0.664  0.146
2    3      1            1  0.590  0.135  0.046  0.852  0.655  0.765  0.261
3    4      1            1  0.226  0.952  0.773  0.070  0.800  0.320  0.081
4    5      1            0  0.250  0.698  0.781  0.060  0.427  0.096  0.176
5    6      1            1  0.446  0.065  0.008  0.224  0.448  0.976  0.629
6    7      1            0  0.729  0.904  0.545  0.137  0.516  0.862  0.386
7    8      1            0  0.169  0.427  0.296  0.765  0.131  0.002  0.643
8    9      1            0  0.197  0.979  0.061  0.505  0.665  0.279  0.595
9   10      1            0  0.465  0.981  0.983  0.685  0.678  0.553  0.552
```

```
   …  var_291  var_292  var_293  var_294  var_295  var_296  var_297  \
0  …    0.188    0.143    0.432    0.872    0.282    0.152    0.878
1  …    0.164    0.676    0.647    0.437    0.853    0.908    0.141
2  …    0.147    0.822    0.769    0.743    0.293    0.806    0.610
3  …    0.155    0.240    0.553    0.102    0.092    0.016    0.785
4  …    0.699    0.765    0.946    0.112    0.744    0.181    0.861
5  …    0.432    0.818    0.120    0.994    0.421    0.298    0.857
6  …    0.322    0.586    0.366    0.673    0.819    0.188    0.341
7  …    0.747    0.854    0.695    0.149    0.744    0.704    0.151
8  …    0.613    0.750    0.932    0.013    0.220    0.728    0.644
9  …    0.317    0.863    0.225    0.147    0.221    0.901    0.105

   var_298  var_299  var_300
0    0.750    0.670    0.358
1    0.705    0.974    0.240
2    0.172    0.825    0.330
3    0.320    0.548    0.888
4    0.383    0.570    0.777
5    0.133    0.093    0.342
6    0.625    0.862    0.246
7    0.089    0.074    0.717
8    0.455    0.238    0.603
9    0.536    0.300    0.674

[10 rows x 303 columns]
```

[ ]:
```
# Check for any missing values. Since it's False, that means we don't have to␣
↪dropna or impute any missing values.
df.isnull().values.any()
```

[ ]: False

[ ]:
```
# df.info
# df.describe
```

[ ]:
```
# Split the dataset into train dataset and test dataset
train_set = df[df['train']==1]
test_set = df[df['train']==0]
```

[ ]:
```
# Verify the number of training dataset and testing dataset
len_train = len(train_set)
len_test = len(test_set)

print(f'The number of training dataset is: {len_train}')
print(f'The number of testing dataset is: {len_test}')
```

```
The number of training dataset is: 250
The number of testing dataset is: 19750
```

For our train/test split, we can drop the train feature because all it tells us is which part of the dataset are train data vs testing dataset. Note that in the challenge stated that there are 250 training datas and 19750 testing data. Meaning, this feature won't help us in the prediction analysis part of our target variable.

[ ]:
```
train_set = train_set.drop('train',axis=1)
test_set = test_set.drop('train',axis=1)
```

It seems that no transformation like StandardScaler or MinMaxScaler seems to be needed in this assessment, since all the target attributes seems to be bound between 0 and 1. If there were overfitting that are noticed in our Machine Learning model, then we can try incorporating it.

[ ]:
```
# Split our dataset into X and y
X = train_set.drop('target_eval',axis=1)
y = train_set['target_eval']
```

[ ]:
```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_classif

X_new = SelectKBest(f_classif, k=100).fit_transform(X,y)
X_new.shape
```

[ ]:
```
(250, 100)
```

1.3 Feature Selection

We'll use SelectKBest features with f_classif for our Feature Selection method.

[ ]:
```
bestfeatures = SelectKBest(k=100, score_func=f_classif)
fit = bestfeatures.fit(X,y)
df_columns = pd.DataFrame(X.columns)
df_scores = pd.DataFrame(fit.scores_)

feature_scores_df = pd.concat([df_columns, df_scores], axis=1)
feature_scores_df = feature_scores_df.dropna()
feature_scores_df.columns = ['Specs','Score']

top100 = feature_scores_df.nlargest(100, 'Score').set_index('Specs')
top100
```

[ ]:
```
             Score
Specs
var_180  13.870855
var_172  12.964053
var_219  11.137178
var_77   10.687868
```

```
var_252   9.649807
…             …
var_208   1.330205
var_233   1.312750
var_166   1.274114
var_255   1.252865
var_287   1.239691

[100 rows x 1 columns]
```

The higher the value, the more significant the feature is to the model. If we look at the bottom features (shown below), we can see that these functions that have a score of 0, which means they don't play a major part in the model, so we can drop those features.

[ ]:
```
feature_scores_df.nlargest(300, 'Score').set_index('Specs')
```

[ ]:
```
               Score
Specs
var_180  13.870855
var_172  12.964053
var_219  11.137178
var_77   10.687868
var_252   9.649807
…             …
var_85    0.000896
var_69    0.000235
var_238   0.000197
var_105   0.000130
var_130   0.000089

[300 rows x 1 columns]
```

There's a lot of features for our target variable. We can use PCA to compress our dataset while retaining 95% of variance to speed up our ML models.

[ ]:
```
top100.index
```

[ ]:
```
Index(['var_180', 'var_172', 'var_219', 'var_77', 'var_252', 'var_203',
       'var_170', 'var_239', 'var_271', 'var_117', 'var_93', 'var_198',
       'var_116', 'var_286', 'var_276', 'var_181', 'var_138', 'var_40',
       'var_247', 'var_53', 'var_294', 'var_253', 'var_249', 'var_270',
       'var_298', 'var_195', 'var_119', 'var_153', 'var_71', 'var_29',
       'var_218', 'var_127', 'var_1', 'var_50', 'var_4', 'var_57', 'var_258',
       'var_19', 'var_34', 'var_157', 'var_223', 'var_272', 'var_282',
       'var_78', 'var_204', 'var_296', 'var_13', 'var_199', 'var_186',
       'var_10', 'var_281', 'var_112', 'var_235', 'var_28', 'var_168',
       'var_58', 'var_63', 'var_45', 'var_237', 'var_288', 'var_60', 'var_70',
       'var_42', 'var_131', 'var_178', 'var_236', 'var_275', 'var_179',
```

```
       'var_158', 'var_206', 'var_151', 'var_31', 'var_139', 'var_15',
       'var_222', 'var_224', 'var_257', 'var_84', 'var_279', 'var_26',
       'var_62', 'var_291', 'var_143', 'var_64', 'var_43', 'var_269', 'var_76',
       'var_164', 'var_200', 'var_182', 'var_201', 'var_126', 'var_66',
       'var_184', 'var_12', 'var_208', 'var_233', 'var_166', 'var_255',
       'var_287'],
      dtype='object', name='Specs')
```

[ ]:
```
X1 = train_set[['var_180', 'var_172', 'var_219', 'var_77', 'var_252', 'var_203',
       'var_170', 'var_239', 'var_271', 'var_117', 'var_93', 'var_198',
       'var_116', 'var_286', 'var_276', 'var_181', 'var_138', 'var_40',
       'var_247', 'var_53', 'var_294', 'var_253', 'var_249', 'var_270',
       'var_298', 'var_195', 'var_119', 'var_153', 'var_71', 'var_29',
       'var_218', 'var_127', 'var_1', 'var_50', 'var_4', 'var_57', 'var_258',
       'var_19', 'var_34', 'var_157', 'var_223', 'var_272', 'var_282',
       'var_78', 'var_204', 'var_296', 'var_13', 'var_199', 'var_186',
       'var_10', 'var_281', 'var_112', 'var_235', 'var_28', 'var_168',
       'var_58', 'var_63', 'var_45', 'var_237', 'var_288', 'var_60', 'var_70',
       'var_42', 'var_131', 'var_178', 'var_236', 'var_275', 'var_179',
       'var_158', 'var_206', 'var_151', 'var_31', 'var_139', 'var_15',
       'var_222', 'var_224', 'var_257', 'var_84', 'var_279', 'var_26',
       'var_62', 'var_291', 'var_143', 'var_64', 'var_43', 'var_269', 'var_76',
       'var_164', 'var_200', 'var_182', 'var_201', 'var_126', 'var_66',
       'var_184', 'var_12', 'var_208', 'var_233', 'var_166', 'var_255',
       'var_287']]
X1
```

[ ]:
```
     var_180  var_172  var_219  var_77  var_252  var_203  var_170  var_239  \
0      0.101    0.475    0.000   0.205    0.485    0.603    0.428    0.095
1      0.801    0.727    0.301   0.260    0.866    0.886    0.302    0.355
2      0.407    0.625    0.392   0.518    0.010    0.891    0.765    0.261
3      0.271    0.770    0.253   0.483    0.887    0.919    0.770    0.031
4      0.515    0.298    0.339   0.535    0.772    0.292    0.799    0.929
..       …        …       …       …        …       …       …       …
245    0.777    0.604    0.759   0.847    0.416    0.435    0.211    0.248
246    0.541    0.938    0.333   0.929    0.872    0.424    0.823    0.143
247    0.371    0.238    0.849   0.690    0.194    0.169    0.318    0.646
248    0.764    0.633    0.404   0.865    0.496    0.579    0.991    0.652
249    0.333    0.210    0.864   0.290    0.802    0.268    0.041    0.023

     var_271  var_117  …  var_201  var_126  var_66  var_184  var_12  \
0      0.629    0.131  …    0.354    0.568   0.551    0.317   0.001
1      0.881    0.669  …    0.320    0.733   0.597    0.607   0.817
2      0.265    0.195  …    0.382    0.152   0.006    0.698   0.731
3      0.265    0.507  …    0.928    0.223   0.219    0.571   0.974
4      0.555    0.552  …    0.226    0.283   0.424    0.465   0.731
..       …        …  …      …        …       …       …
```

```
245    0.978    0.248  …    0.565    0.717    0.196    0.976    0.435
246    0.939    0.336  …    0.725    0.816    0.902    0.726    0.405
247    0.297    0.316  …    0.995    0.015    0.554    0.808    0.359
248    0.458    0.307  …    0.665    0.119    0.229    0.022    0.191
249    0.191    0.777  …    0.820    0.398    0.491    0.328    0.564

     var_208  var_233  var_166  var_255  var_287
0      0.324    0.835    0.674    0.008    0.281
1      0.324    0.847    0.984    0.902    0.275
2      0.406    0.716    0.573    0.856    0.385
3      0.829    0.299    0.262    0.223    0.391
4      0.508    0.047    0.637    0.490    0.921
..       …      …      …      …      …
245    0.335    0.193    0.074    0.428    0.225
246    0.756    0.121    0.154    0.117    0.615
247    0.257    0.221    0.703    0.320    0.699
248    0.806    0.495    0.388    0.277    0.715
249    0.937    0.444    0.404    0.640    0.402

[250 rows x 100 columns]
```

[ ]:
```
# Here, we can split both our training set using train_test_split on our feature selected dataset.

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X1,y,test_size=0.2, random_state=42)
```

[ ]:
```
# Check if our dimension for train and test set is correct
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(200, 100) (50, 100) (200,) (50,)
```

1.4 Principal Component Space (PCA)

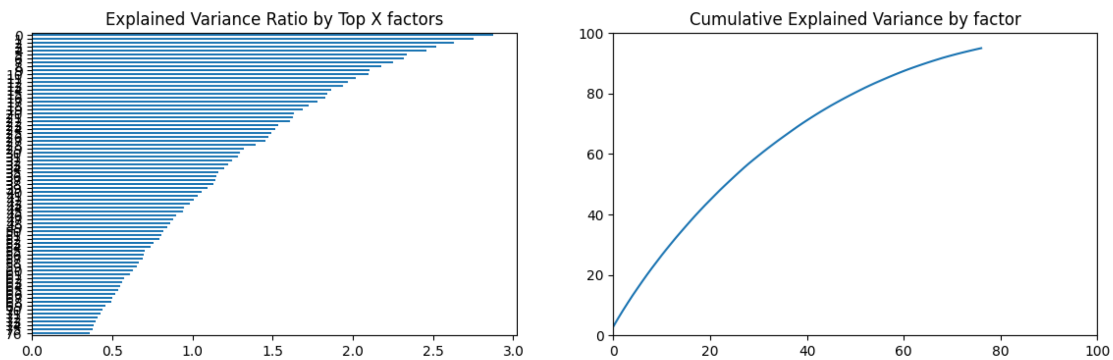
Here, since our dataset has a lot of features, it would be helpful to reduce our model complexity by introducing PCA. We retain 95% explained variance.

[ ]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=0.95)
pcomp = pca.fit(X_train)
```

[ ]:
```
# number of principal components projected in the Eigen space with 95% explained variance
len(pca.components_)
```

[ ]:
```
77
```

[ ]:
```
num_eigenvalues = 100
fig, axes = plt.subplots(ncols=2, figsize=(14,4))
Series1= pd.Series(pca.explained_variance_ratio_[:num_eigenvalues]).
  ↪sort_values()*100
Series2 = pd.Series(pca.explained_variance_ratio_[:num_eigenvalues]).
  ↪cumsum()*100

#Series1 will be hard to see, but if there were less target attributes to work␣
  ↪with, you would be able to visually see it properly
Series1.plot.barh(ylim=(0,9), label='woohoo', title='Explained Variance Ratio␣
  ↪by Top X factors', ax=axes[0]);
Series2.plot(ylim=(0,100),xlim=(0,100),ax=axes[1], title='Cumulative Explained␣
  ↪Variance by factor');
```



If we uncomment this, we can see that at 77 Eigen values, we can expect to see 95% explained variance retained when using PCA, which is better than having 100 features (or 300 had we not performed feature selection) to go through, which time is always a constraint to the many machine learning models to loop through.

[ ]:
```
pd.Series(np.cumsum(pca.explained_variance_ratio_)).to_frame('Explained␣
  ↪Variance').head(num_eigenvalues).style.format('{:,.2%}'.format)
```

[ ]:
```
<pandas.io.formats.style.Styler at 0x181bd3c3c40>
```

[ ]:
```
X_train_PCA = pca.transform(X_train)
X_test_PCA = pca.transform(X_test)
```

Now we get to the fun part with various machine learning models and later explore the neural networks or Deep Learning models.

[ ]:
```
from sklearn.model_selection import KFold, cross_val_score
from sklearn.metrics import roc_auc_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier,␣
↪ExtraTreesClassifier, GradientBoostingClassifier
```

[ ]:
```
models = []
models.append(('LR', LogisticRegression(n_jobs=-1)))
models.append(('KNN',KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('NN', MLPClassifier()))
models.append(('AB', AdaBoostClassifier()))
models.append(('RF', RandomForestClassifier()))
models.append(('ETC', ExtraTreesClassifier()))
models.append(('GBC', GradientBoostingClassifier()))
```

[ ]:
```
# Evaluation metrics:
num_folds = 10
scoring = 'roc_auc'
```

[ ]:
```
import warnings
warnings.filterwarnings('ignore')
```

1.5 Model Selection

[ ]:
```
# K Fold Cross Validation WITH PCA
names = []
results = []
train_results = []
test_results = []

for name, model in models:
    kfold = KFold(n_splits=num_folds)
    cv_results = cross_val_score(model, X_train, y_train, cv = kfold,␣
↪scoring=scoring)

    res = model.fit(X_train, y_train)
    train_result = roc_auc_score(res.predict(X_train), y_train)
    train_results.append(train_results)

    test_result = roc_auc_score(res.predict(X_test),y_test)
    test_results.append(test_result)

    names.append(name)
    results.append(cv_results)
```

```
    msg = "%s, %f (%f) %f %f" % (name, cv_results.mean(), cv_results.std(),␣
↪train_result, test_result)
    print(msg)
```

```
LR, 0.923316 (0.042986) 0.989999 0.844551
KNN, 0.777087 (0.108644) 0.876054 0.669082
CART, 0.507413 (0.120956) 1.000000 0.557471
NB, 0.889004 (0.071419) 0.949995 0.836538
NN, 0.847869 (0.095834) 1.000000 0.791667
AB, 0.751262 (0.111099) 1.000000 0.606732
RF, 0.780786 (0.107457) 1.000000 0.711397
ETC, 0.782378 (0.070804) 1.000000 0.676282
GBC, 0.662250 (0.120158) 1.000000 0.564103
```

[ ]:
```
# K Fold Cross Validation WITH PCA
names = []
results = []
train_results = []
test_results = []

for name, model in models:
    kfold = KFold(n_splits=num_folds)
    cv_results = cross_val_score(model, X_train_PCA, y_train, cv = kfold,␣
↪scoring=scoring)

    res = model.fit(X_train_PCA, y_train)
    train_result = roc_auc_score(res.predict(X_train_PCA), y_train)
    train_results.append(train_results)

    test_result = roc_auc_score(res.predict(X_test_PCA),y_test)
    test_results.append(test_result)

    names.append(name)
    results.append(cv_results)

    msg = "%s, %f (%f) %f %f" % (name, cv_results.mean(), cv_results.std(),␣
↪train_result, test_result)
    print(msg)
```

```
LR, 0.925316 (0.042324) 0.989999 0.844551
KNN, 0.773796 (0.128578) 0.846820 0.644231
CART, 0.573508 (0.075165) 1.000000 0.613636
NB, 0.790352 (0.118268) 0.950255 0.756410
NN, 0.911417 (0.057789) 1.000000 0.820000
AB, 0.873605 (0.089109) 1.000000 0.620000
RF, 0.815684 (0.107201) 1.000000 0.764423
ETC, 0.800874 (0.113546) 1.000000 0.773752
GBC, 0.815762 (0.099273) 1.000000 0.695246
```

From comparing our KFold Cross Validation, we can see that the KFold applied on our dataset from PCA had similar results.

We can see that: * Logistic Regression, Naive Bayes, and MLP had good ROC_AUC score in both training/testing set WITHOUT PCA. * Only Logistic Regression had a good ROC_AUC score in both training/testing set with PCA. * Most of the other algorithms fit our training set perfectly, but predicted poorly on our test set WITH and WITHOUT PCA, overfitted..

tldr: Our top candidates are Neural Network and Logistic Regression. We explore these ML algorithms further.

1.6 Neural Network

[ ]:
```
from keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_loss', mode='min', verbose=1,␣
 ↪patience=10)

from keras.models import Sequential
from keras.layers import Dense
from keras.metrics import AUC

model0 = Sequential()

# input layer
model0.add(Dense(100, activation='relu'))
# model0.add(Dropout(0.2))

# hidden layer
model0.add(Dense(50, activation='relu'))
# model0.add(Dropout(0.2))
model0.add(Dense(25, activation='relu'))

# output layyer
model0.add(Dense(1, activation='sigmoid'))

# compile model
model0.compile(loss='binary_crossentropy', optimizer='adam',metrics=[AUC()])
```

[ ]:
```
model0.fit(X_train, y_train,
           epochs=100, batch_size=256,
           verbose=1, callbacks=[early_stop],
           validation_data=(X_test, y_test))
```

```
Epoch 1/100
1/1 [==============================] - 1s 1s/step - loss: 0.7064 - auc: 0.5062 -
val_loss: 0.7036 - val_auc: 0.5267
Epoch 2/100
1/1 [==============================] - 0s 45ms/step - loss: 0.6900 - auc: 0.5579
- val_loss: 0.6876 - val_auc: 0.5600
```

```
Epoch 3/100
1/1 [==============================] - 0s 44ms/step - loss: 0.6813 - auc: 0.6174
- val_loss: 0.6798 - val_auc: 0.5758
Epoch 4/100
1/1 [==============================] - 0s 40ms/step - loss: 0.6753 - auc: 0.6679
- val_loss: 0.6771 - val_auc: 0.5867
Epoch 5/100
1/1 [==============================] - 0s 45ms/step - loss: 0.6694 - auc: 0.7188
- val_loss: 0.6775 - val_auc: 0.6017
Epoch 6/100
1/1 [==============================] - 0s 46ms/step - loss: 0.6626 - auc: 0.7542
- val_loss: 0.6806 - val_auc: 0.6008
Epoch 7/100
1/1 [==============================] - 0s 42ms/step - loss: 0.6552 - auc: 0.7842
- val_loss: 0.6837 - val_auc: 0.6175
Epoch 8/100
1/1 [==============================] - 0s 39ms/step - loss: 0.6482 - auc: 0.8111
- val_loss: 0.6860 - val_auc: 0.6283
Epoch 9/100
1/1 [==============================] - 0s 35ms/step - loss: 0.6415 - auc: 0.8268
- val_loss: 0.6864 - val_auc: 0.6317
Epoch 10/100
1/1 [==============================] - 0s 41ms/step - loss: 0.6345 - auc: 0.8451
- val_loss: 0.6829 - val_auc: 0.6433
Epoch 11/100
1/1 [==============================] - 0s 46ms/step - loss: 0.6265 - auc: 0.8611
- val_loss: 0.6759 - val_auc: 0.6575
Epoch 12/100
1/1 [==============================] - 0s 43ms/step - loss: 0.6176 - auc: 0.8780
- val_loss: 0.6670 - val_auc: 0.6750
Epoch 13/100
1/1 [==============================] - 0s 42ms/step - loss: 0.6087 - auc: 0.8926
- val_loss: 0.6587 - val_auc: 0.6892
Epoch 14/100
1/1 [==============================] - 0s 40ms/step - loss: 0.5998 - auc: 0.9062
- val_loss: 0.6524 - val_auc: 0.7008
Epoch 15/100
1/1 [==============================] - 0s 38ms/step - loss: 0.5905 - auc: 0.9141
- val_loss: 0.6482 - val_auc: 0.7167
Epoch 16/100
1/1 [==============================] - 0s 46ms/step - loss: 0.5802 - auc: 0.9190
- val_loss: 0.6457 - val_auc: 0.7167
Epoch 17/100
1/1 [==============================] - 0s 42ms/step - loss: 0.5693 - auc: 0.9247
- val_loss: 0.6440 - val_auc: 0.7225
Epoch 18/100
1/1 [==============================] - 0s 38ms/step - loss: 0.5574 - auc: 0.9294
- val_loss: 0.6419 - val_auc: 0.7233
```

```
Epoch 19/100
1/1 [==============================] - 0s 39ms/step - loss: 0.5449 - auc: 0.9342
- val_loss: 0.6372 - val_auc: 0.7292
Epoch 20/100
1/1 [==============================] - 0s 38ms/step - loss: 0.5313 - auc: 0.9394
- val_loss: 0.6301 - val_auc: 0.7292
Epoch 21/100
1/1 [==============================] - 0s 41ms/step - loss: 0.5173 - auc: 0.9449
- val_loss: 0.6225 - val_auc: 0.7325
Epoch 22/100
1/1 [==============================] - 0s 41ms/step - loss: 0.5030 - auc: 0.9484
- val_loss: 0.6162 - val_auc: 0.7442
Epoch 23/100
1/1 [==============================] - 0s 42ms/step - loss: 0.4880 - auc: 0.9505
- val_loss: 0.6119 - val_auc: 0.7483
Epoch 24/100
1/1 [==============================] - 0s 40ms/step - loss: 0.4725 - auc: 0.9534
- val_loss: 0.6084 - val_auc: 0.7533
Epoch 25/100
1/1 [==============================] - 0s 51ms/step - loss: 0.4564 - auc: 0.9557
- val_loss: 0.6037 - val_auc: 0.7583
Epoch 26/100
1/1 [==============================] - 0s 44ms/step - loss: 0.4397 - auc: 0.9573
- val_loss: 0.5956 - val_auc: 0.7642
Epoch 27/100
1/1 [==============================] - 0s 46ms/step - loss: 0.4228 - auc: 0.9595
- val_loss: 0.5855 - val_auc: 0.7667
Epoch 28/100
1/1 [==============================] - 0s 44ms/step - loss: 0.4060 - auc: 0.9626
- val_loss: 0.5770 - val_auc: 0.7733
Epoch 29/100
1/1 [==============================] - 0s 40ms/step - loss: 0.3890 - auc: 0.9642
- val_loss: 0.5722 - val_auc: 0.7767
Epoch 30/100
1/1 [==============================] - 0s 44ms/step - loss: 0.3718 - auc: 0.9660
- val_loss: 0.5686 - val_auc: 0.7783
Epoch 31/100
1/1 [==============================] - 0s 49ms/step - loss: 0.3547 - auc: 0.9680
- val_loss: 0.5616 - val_auc: 0.7783
Epoch 32/100
1/1 [==============================] - 0s 52ms/step - loss: 0.3375 - auc: 0.9701
- val_loss: 0.5531 - val_auc: 0.7817
Epoch 33/100
1/1 [==============================] - 0s 50ms/step - loss: 0.3206 - auc: 0.9720
- val_loss: 0.5484 - val_auc: 0.7858
Epoch 34/100
1/1 [==============================] - 0s 55ms/step - loss: 0.3042 - auc: 0.9733
- val_loss: 0.5482 - val_auc: 0.7900
```

```
Epoch 35/100
1/1 [==============================] - 0s 46ms/step - loss: 0.2886 - auc: 0.9753
- val_loss: 0.5461 - val_auc: 0.7967
Epoch 36/100
1/1 [==============================] - 0s 39ms/step - loss: 0.2735 - auc: 0.9771
- val_loss: 0.5411 - val_auc: 0.8033
Epoch 37/100
1/1 [==============================] - 0s 56ms/step - loss: 0.2588 - auc: 0.9794
- val_loss: 0.5391 - val_auc: 0.8092
Epoch 38/100
1/1 [==============================] - 0s 44ms/step - loss: 0.2446 - auc: 0.9816
- val_loss: 0.5393 - val_auc: 0.8133
Epoch 39/100
1/1 [==============================] - 0s 38ms/step - loss: 0.2309 - auc: 0.9836
- val_loss: 0.5376 - val_auc: 0.8225
Epoch 40/100
1/1 [==============================] - 0s 41ms/step - loss: 0.2177 - auc: 0.9855
- val_loss: 0.5366 - val_auc: 0.8258
Epoch 41/100
1/1 [==============================] - 0s 37ms/step - loss: 0.2051 - auc: 0.9874
- val_loss: 0.5398 - val_auc: 0.8275
Epoch 42/100
1/1 [==============================] - 0s 42ms/step - loss: 0.1928 - auc: 0.9891
- val_loss: 0.5412 - val_auc: 0.8350
Epoch 43/100
1/1 [==============================] - 0s 41ms/step - loss: 0.1809 - auc: 0.9906
- val_loss: 0.5431 - val_auc: 0.8408
Epoch 44/100
1/1 [==============================] - 0s 42ms/step - loss: 0.1697 - auc: 0.9921
- val_loss: 0.5503 - val_auc: 0.8425
Epoch 45/100
1/1 [==============================] - 0s 46ms/step - loss: 0.1591 - auc: 0.9930
- val_loss: 0.5528 - val_auc: 0.8467
Epoch 46/100
1/1 [==============================] - 0s 45ms/step - loss: 0.1489 - auc: 0.9944
- val_loss: 0.5569 - val_auc: 0.8492
Epoch 47/100
1/1 [==============================] - 0s 41ms/step - loss: 0.1392 - auc: 0.9953
- val_loss: 0.5651 - val_auc: 0.8517
Epoch 48/100
1/1 [==============================] - 0s 41ms/step - loss: 0.1298 - auc: 0.9958
- val_loss: 0.5704 - val_auc: 0.8500
Epoch 49/100
1/1 [==============================] - 0s 44ms/step - loss: 0.1209 - auc: 0.9966
- val_loss: 0.5731 - val_auc: 0.8533
Epoch 50/100
1/1 [==============================] - 0s 40ms/step - loss: 0.1123 - auc: 0.9973
- val_loss: 0.5794 - val_auc: 0.8525
```

```
Epoch 50: early stopping
```

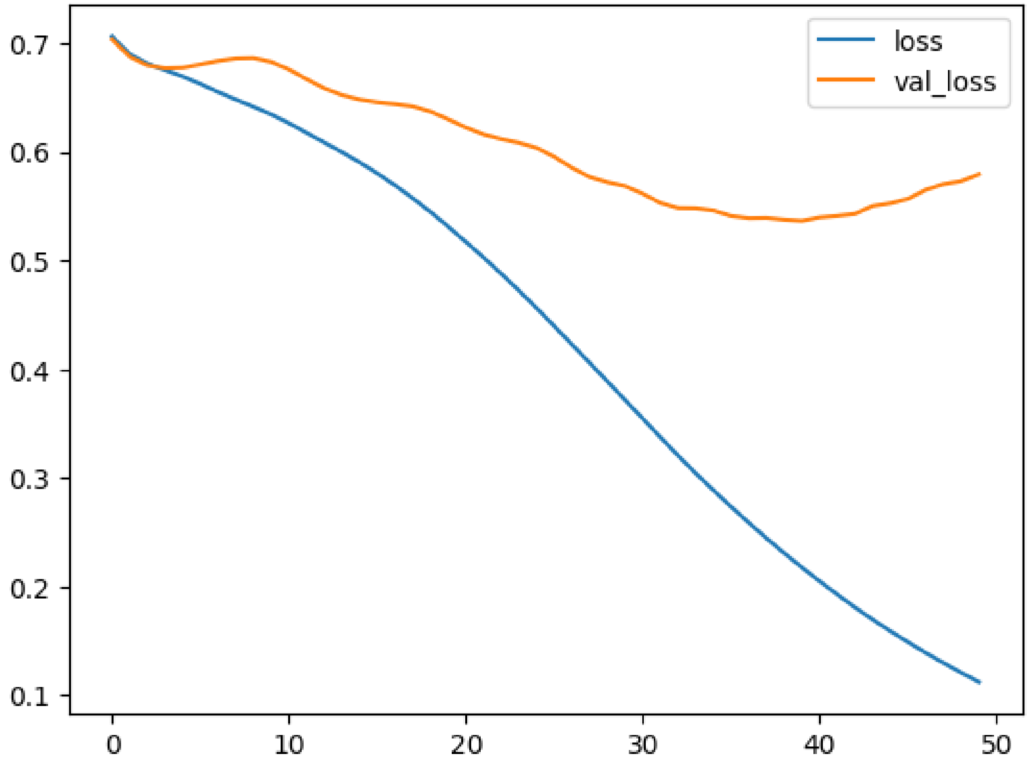
[ ]:
```
<keras.callbacks.History at 0x181ce538f70>
```

[ ]:
```
model0.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 100)               10100

 dense_1 (Dense)             (None, 50)                5050

 dense_2 (Dense)             (None, 25)                1275

 dense_3 (Dense)             (None, 1)                 26

=================================================================
Total params: 16,451
Trainable params: 16,451
Non-trainable params: 0
_________________________________________________________________
```

[ ]:
```
losses = pd.DataFrame(model0.history.history)
losses[['loss','val_loss']].plot()
```

[ ]:
```
<Axes: >
```



[ ]:
```
predictions = model0.predict(X_test)
predictions = np.where(predictions < 0.5,0,1)
predictions[0:5]
```

```
2/2 [==============================] - 0s 2ms/step
```

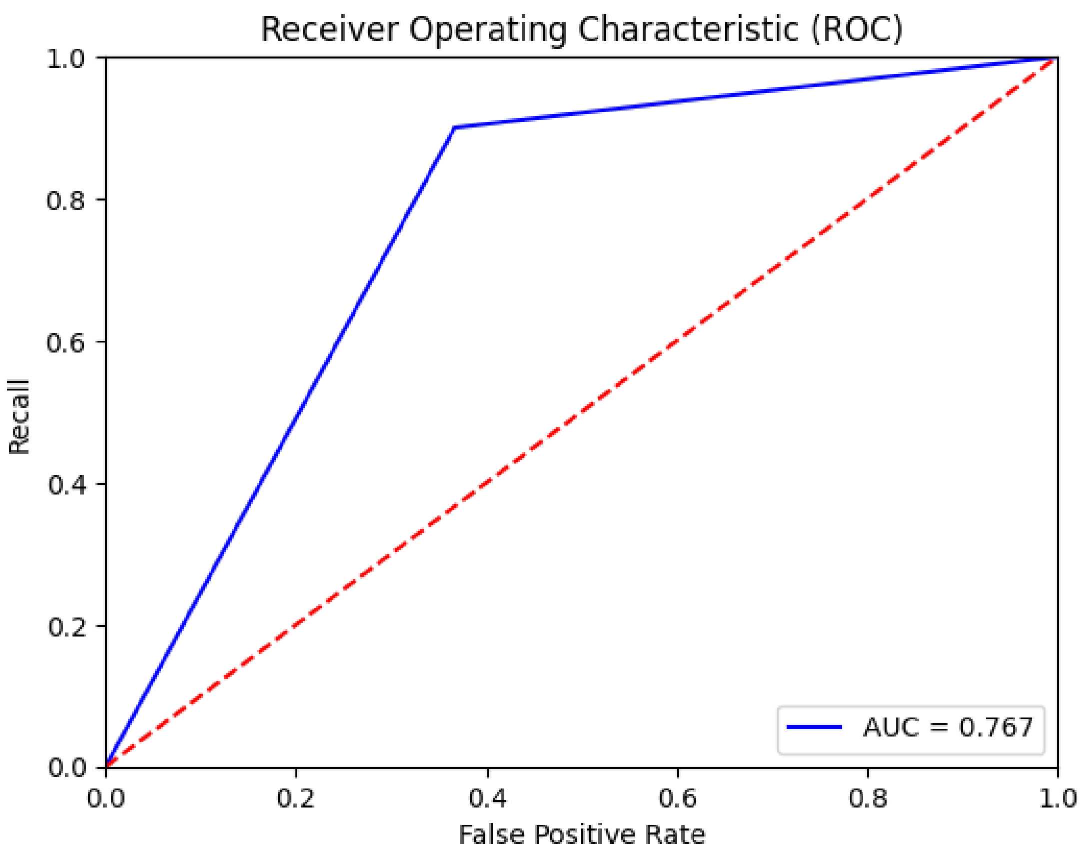
[ ]:
```
array([[0],
       [1],
       [0],
       [1],
       [1]])
```

[ ]:
```
y_test.head(5)
```

[ ]:
```
142    0
6      0
97     0
60     0
112    0
Name: target_eval, dtype: int64
```

[ ]:
```
from sklearn.metrics import confusion_matrix, accuracy_score, roc_curve, auc

false_positive_rate, recall, threshold = roc_curve(y_test, predictions)
roc_auc = auc(false_positive_rate, recall)
```

[ ]:
```
plt.figure()
plt.title('Receiver Operating Characteristic (ROC)')
plt.plot(false_positive_rate, recall, 'b', label='AUC = %0.3f' %roc_auc)

plt.legend(loc='lower right')
plt.plot([0,1], [0,1], 'r--')
plt.xlim([0,1])
plt.ylim([0,1])
plt.ylabel('Recall')
plt.xlabel('False Positive Rate')
plt.show()
```



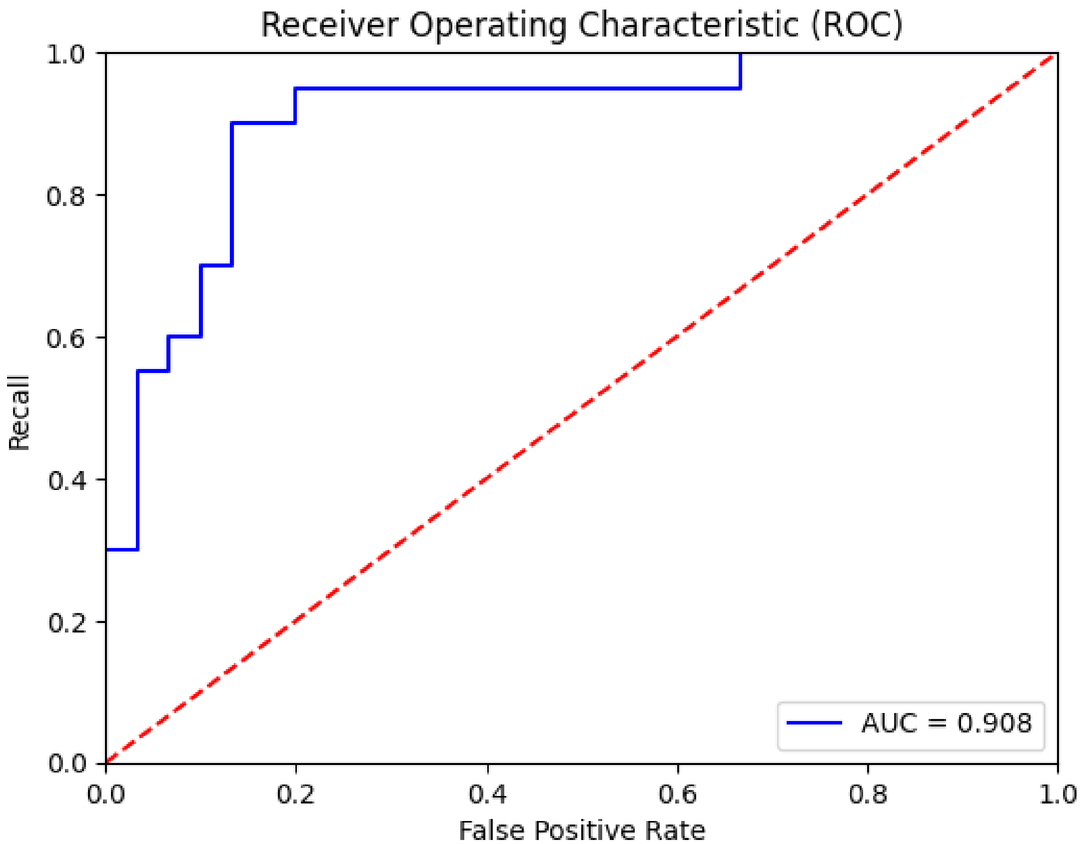
A score of 0.817 isn't terrible for a ROC_AUC score, however it would be considered barely good. TWe will explore

1.7 Logistic Regression

[ ]:
```
LR = LogisticRegression(n_jobs=-1)
LR.fit(X_train_PCA, y_train)
```

[ ]:
```
LogisticRegression(n_jobs=-1)
```

[ ]:
```
probs = LR.predict_proba(X_test_PCA)
preds = probs[:,1]

false_positive_rate, recall, threshold = roc_curve(y_test, preds)
roc_auc = auc(false_positive_rate, recall)

plt.figure()
plt.title('Receiver Operating Characteristic (ROC)')
plt.plot(false_positive_rate, recall, 'b', label='AUC = %0.3f' %roc_auc)

plt.legend(loc='lower right')
plt.plot([0,1], [0,1], 'r--')
plt.xlim([0,1])
plt.ylim([0,1])
plt.ylabel('Recall')
plt.xlabel('False Positive Rate')
plt.show()
```



With default Logistic Regression parameter, our test data with PCA had a good AUC score of 0.908. We can most likely improve this with GridSearchCV.

[ ]:
```
# LogisticRegression?
```

[ ]:
```
from sklearn.model_selection import GridSearchCV

penalty = ['l2', 'l1']
solver = ['liblinear']
C = [0.001, 0.01, 0.05, 0.1, 0.5, 1, 10, 100, 1000]
param_grid = dict(penalty=penalty, solver=solver, C=C)
model_0 = LogisticRegression()
kfold = KFold(n_splits=num_folds)
grid = GridSearchCV(estimator=model_0, param_grid=param_grid,␣
 ↪scoring='roc_auc', return_train_score=True, n_jobs=-1)
grid_result = grid.fit(X_train_PCA, y_train)
```

[ ]:
```
grid_result.best_score_
```

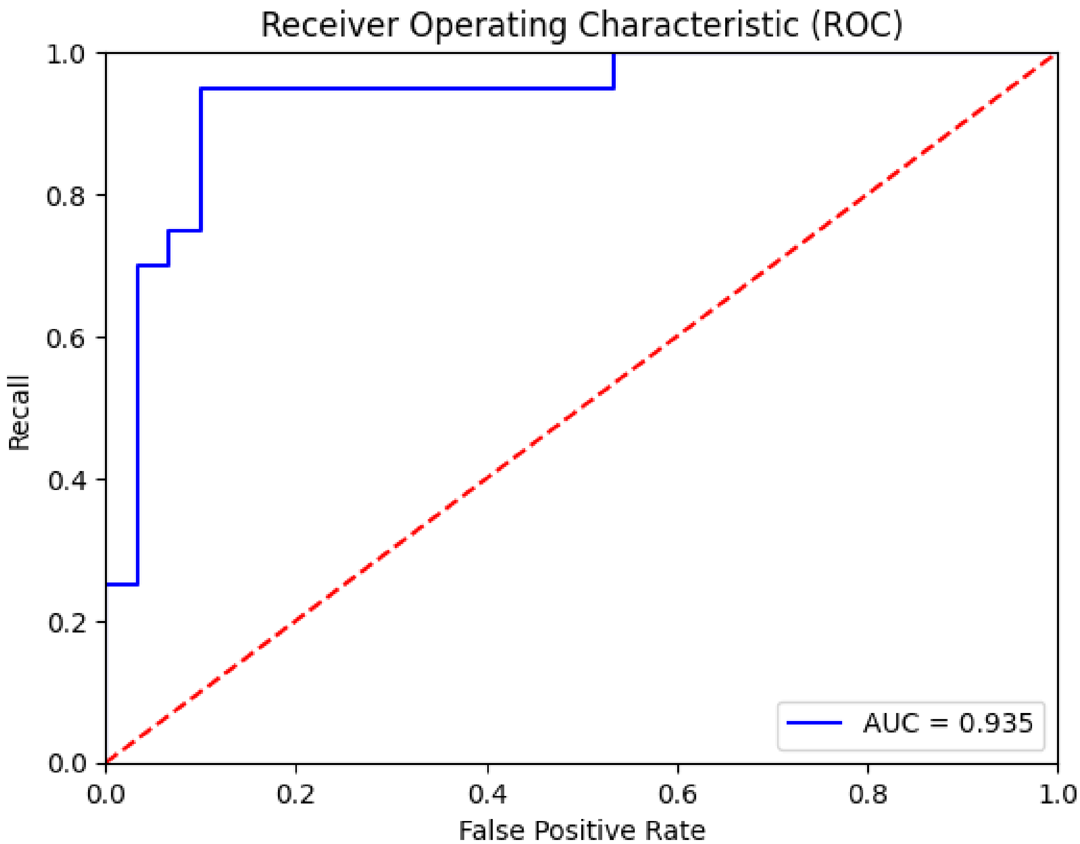
[ ]: 0.9339473684210526

[ ]:
```
grid_result.best_params_
```

[ ]:
```
{'C': 0.1, 'penalty': 'l2', 'solver': 'liblinear'}
```

[ ]:
```
LR = LogisticRegression(penalty='l2',solver='liblinear', C=0.1)
LR.fit(X_train_PCA, y_train)
new_test_pred = LR.predict(X_test_PCA)
```

[ ]:
```
probs = LR.predict_proba(X_test_PCA)
preds = probs[:,1]

false_positive_rate, recall, threshold = roc_curve(y_test, preds)
roc_auc = auc(false_positive_rate, recall)

plt.figure()
plt.title('Receiver Operating Characteristic (ROC)')
plt.plot(false_positive_rate, recall, 'b', label='AUC = %0.3f' %roc_auc)

plt.legend(loc='lower right')
plt.plot([0,1], [0,1], 'r--')
plt.xlim([0,1])
plt.ylim([0,1])
plt.ylabel('Recall')
plt.xlabel('False Positive Rate')
plt.show()
```



On default Logistic Regression parameter, our AUC score was 0.908. On GridSearchCV Logistic Regression parameter, our AUC score is 0.935 (an increase of 0.027).

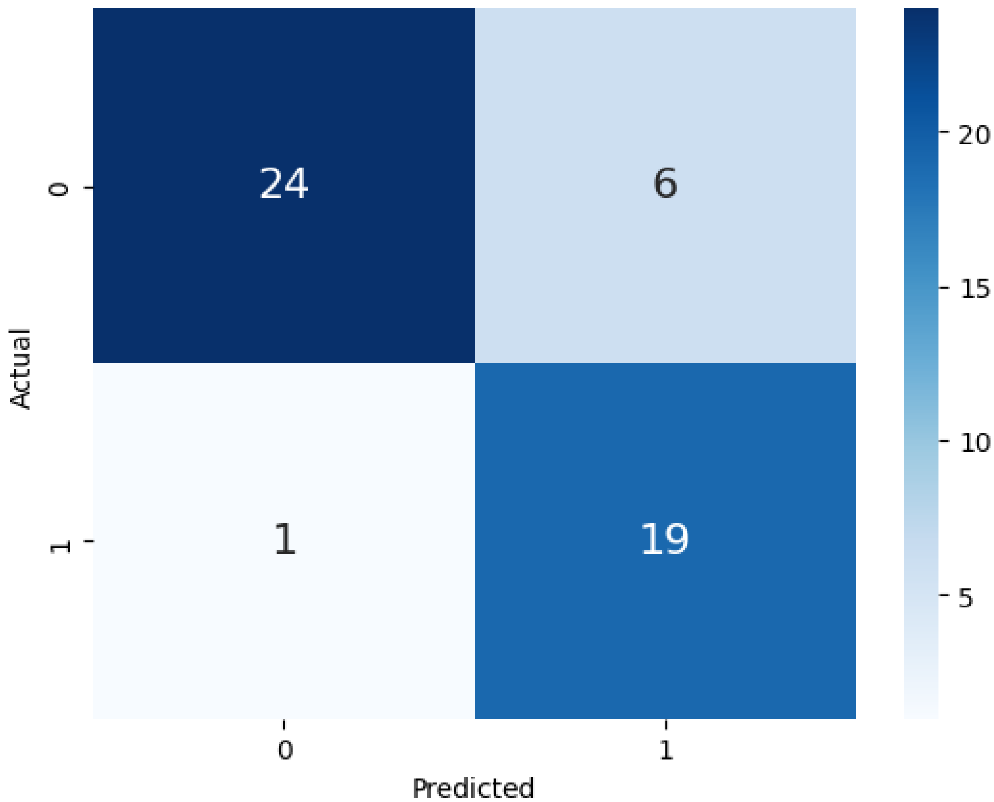
We can check out other metrics like accuracy and look at the confusion matrix.

[ ]:
```
print(accuracy_score(y_test, new_test_pred))
```

```
0.86
```

[ ]:
```
df_cm = pd.DataFrame(confusion_matrix(y_test, new_test_pred), columns=np.
 ↪unique(y_test), index=np.unique(y_test))
df_cm.index.name = 'Actual'
df_cm.columns.name = 'Predicted'
sns.heatmap(df_cm, cmap='Blues', annot=True, annot_kws={"size":16})
```

[ ]:
```
<Axes: xlabel='Predicted', ylabel='Actual'>
```



1.8 Finalizing our actual "TEST" dataset (19750 rows)

```
[ ]: X1_final = test_set[['var_180', 'var_172', 'var_219', 'var_77', 'var_252',␣
      ↪'var_203',
             'var_170', 'var_239', 'var_271', 'var_117', 'var_93', 'var_198',
             'var_116', 'var_286', 'var_276', 'var_181', 'var_138', 'var_40',
             'var_247', 'var_53', 'var_294', 'var_253', 'var_249', 'var_270',
             'var_298', 'var_195', 'var_119', 'var_153', 'var_71', 'var_29',
             'var_218', 'var_127', 'var_1', 'var_50', 'var_4', 'var_57', 'var_258',
             'var_19', 'var_34', 'var_157', 'var_223', 'var_272', 'var_282',
             'var_78', 'var_204', 'var_296', 'var_13', 'var_199', 'var_186',
             'var_10', 'var_281', 'var_112', 'var_235', 'var_28', 'var_168',
             'var_58', 'var_63', 'var_45', 'var_237', 'var_288', 'var_60', 'var_70',
             'var_42', 'var_131', 'var_178', 'var_236', 'var_275', 'var_179',
             'var_158', 'var_206', 'var_151', 'var_31', 'var_139', 'var_15',
             'var_222', 'var_224', 'var_257', 'var_84', 'var_279', 'var_26',
             'var_62', 'var_291', 'var_143', 'var_64', 'var_43', 'var_269', 'var_76',
             'var_164', 'var_200', 'var_182', 'var_201', 'var_126', 'var_66',
             'var_184', 'var_12', 'var_208', 'var_233', 'var_166', 'var_255',
             'var_287']]
     X_PCA = pca.transform(X1_final)
```

```
len(pca.components_)
final_pred = LR.predict(X_PCA)
```

[ ]:
```
final_probs = LR.predict_proba(X_PCA)
```

[ ]:
```
test_set['target_eval'] = final_pred
test_set['proba_class_0'] = final_probs[:,0]
test_set['proba_class_1'] = final_probs[:,1]
```

1.9 Save our final csv file :)

[ ]:
```
test_set[['id','target_eval','proba_class_0','proba_class_1']].
↪to_csv('final_pred.csv',index=False)
```