BInary classification in case of imbalanced data

I have data about 2100 employees: 16 variables and whether they are fired or not. I need to create a model for data classification.

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.calibration import CalibratedClassifierCV
from sklearn.model_selection import StratifiedKFold, train_test_split, GridS
from sklearn.metrics import classification_report
```

In [2]:
```
data = pd.read_csv('/file.csv')
```

In [3]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2100 entries, 0 to 2099
Data columns (total 18 columns):
employee_id    2100 non-null object
factor_1       2100 non-null int64
factor_2       2100 non-null int64
factor_3       2100 non-null int64
factor_4       2100 non-null int64
factor_5       2100 non-null int64
factor_6       2100 non-null int64
factor_7       2100 non-null int64
factor_8       2100 non-null int64
factor_9       2100 non-null int64
factor_10      2100 non-null int64
factor_11      2100 non-null int64
factor_12      2100 non-null int64
factor_13      2100 non-null int64
factor_14      2100 non-null int64
factor_15      2100 non-null int64
factor_16      2100 non-null int64
fired          2100 non-null int64
dtypes: int64(17), object(1)
memory usage: 295.4+ KB
```

No missing data, all variables are numeric except employee id.

In [4]:
```
data.head()
```

Out[4]:

	employee_id	factor_1	factor_2	factor_3	factor_4	factor_5	fa
0	957099050466813076	3529	500934542	3012	378	1557	
1	13164358679111999796	2017	490702594	1958	453	1238	
2	5442250097157630866	169	43802030	162	28	0	
3	9345017131298737624	844	201061365	781	256	172	
4	6389462342858680146	213	95858986	197	46	0	

In [5]:
```
data.fired.unique()
```

Out[5]:
```
array([0, 1], dtype=int64)
```

In [6]:
```
print('There are {:.2f}% zero values in "fired" column.'.format((1 - sum(dat
```

```
There are 95.52% zero values in "fired" column.
```

The dataset has imbalanced classes: 95.52% of all rows have "0" value in the column "fired". This means that predicting "1" is more difficult that "0". And model accuracy of 95.52% could mean that model dumbly predicts "0" in all cases. To increase predictive capacity of the model the following methods are commonly used:

- collecting more data to decrease imbalance;
- using other metrics (not accuracy) to measure model performance, for example F1-score;
- using less samples of the common class and/or more samples of the rare class for training;
- changing threshold for predicting the class;
- trying various algorithms;
- using weights to increase the importance of the rare class;
- generating artificial data;
- trying to solve a model of anomaly detection instead of classification;

Random Forest is a great model to use in case of imbalanced data.

In [7]:
```
X_train = data.drop(['fired', 'employee_id'], axis=1)
Y_train = data.fired
```

In [8]:
```
#Evaluating feature importance.
clf = RandomForestClassifier(n_estimators=200)
clf = clf.fit(X_train, Y_train)
indices = np.argsort(clf.feature_importances_)[::-1]

print('Feature ranking:')

for f in range(X_train.shape[1]):
```

```
    print('%d. feature %d %s (%f)' % (f + 1, indices[f], X_train.columns[inc
                                      clf.feature_importances_[indices[f]]))
```

```
Feature ranking:
1. feature 12 factor_13 (0.294455)
2. feature 14 factor_15 (0.224225)
3. feature 10 factor_11 (0.103989)
4. feature 11 factor_12 (0.067701)
5. feature 7 factor_8 (0.054108)
6. feature 15 factor_16 (0.043029)
7. feature 8 factor_9 (0.042708)
8. feature 13 factor_14 (0.038124)
9. feature 6 factor_7 (0.025042)
10. feature 9 factor_10 (0.019295)
11. feature 2 factor_3 (0.018302)
12. feature 0 factor_1 (0.017599)
13. feature 1 factor_2 (0.014488)
14. feature 3 factor_4 (0.012968)
15. feature 5 factor_6 (0.012587)
16. feature 4 factor_5 (0.011381)
```

Only several features have high importance. But I can't just throw away other features due to class imbalance, as less important factors could be importand for predicting "1".

In [9]:
```
Xtrain, Xtest, ytrain, ytest = train_test_split(X_train, Y_train, test_size=
```

I used GridSearchCV to tune model's parameters. Also I use CalibratedClassifierCV to improve probability prediction.

In [10]:
```
clf = RandomForestClassifier(n_estimators=150, n_jobs=-1, criterion = 'gini'
                             min_samples_split=7, min_weight_fraction_leaf=0
                             max_leaf_nodes=40, max_depth=10)

calibrated_clf = CalibratedClassifierCV(clf, method='sigmoid', cv=5)
calibrated_clf.fit(Xtrain, ytrain)
y_val = calibrated_clf.predict_proba(Xtest)
```

Let's see several metrics to measure model's performance.

At first simple accuracy.

In [11]:
```
print("Accuracy {0}%".format(round(sum(pd.DataFrame(y_val).idxmax(axis=1).va
```

```
Accuracy 99.5238%
```

The accuracy is quite good, but this metric doesn't show how accurate are predictions for each class.

In [12]:
```
print(classification_report(ytest, pd.DataFrame(y_val).idxmax(axis=1).values
```

```
             precision    recall  f1-score   support

          0     0.9950    1.0000    0.9975       401
          1     1.0000    0.8947    0.9444        19

avg / total     0.9953    0.9952    0.9951       420
```

Now we can cee that the models give good predictions for both classes.

But the result may be improves if the threshold of choosing the class is changed.

In [13]:
```
y_threshold = np.zeros(ytest.shape).astype(int)
for i in range(len(y_val)):
    if y_val[i][1] > 0.1:
        y_threshold[i] = 1
print(classification_report(ytest, y_threshold, target_names=['0', '1'], dig
```

```
             precision    recall  f1-score   support

          0     1.0000    0.9975    0.9988       401
          1     0.9500    1.0000    0.9744        19

avg / total     0.9977    0.9976    0.9976       420
```

Model's quality has improved. But it is better to automate the search of the optimal threshold.

In [14]:
```
def optimal_threshold(do):
    threshold = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5]
    f1 = []
    for j in range(len(threshold)):
        y_threshold = np.zeros(ytest.shape).astype(int)
        for i in range(len(y_val)):
            if y_val[i][1] > threshold[j]:
                y_threshold[i] = 1
        f1.append(classification_report(ytest, y_threshold, target_names=['0

    if do == 'print':
        print('Maximum value of F1-score is {0} with threshold {1}.'.format(
    elif do == 'calc':
        return max(f1)
```

In [15]:
```
optimal_threshold('print')
```

```
Maximum value of F1-score is 0.9976 with threshold 0.1.
```

So, the model is quite accurate. But there is one more challenge: the amount of observations isn't very hish, so data splitting has a serious influence on the predictions. And in some cases optimal threshold may be 0.5. So it is better to use closs-validation.

In the code below I split data into train and test 10 times, each time model is fitted on train data, predicts values for test data and the best f1 score is calculate. After ten iterations mean value of F1-score and standard deviation is shown.

In [16]:
```
j = 0
score = []
while j < 10:
    Xtrain, Xtest, ytrain, ytest = train_test_split(X_train, Y_train, test_s
    calibrated_clf.fit(Xtrain, ytrain)
    y_val = calibrated_clf.predict_proba(Xtest)
    y_ = np.zeros(ytest.shape).astype(int)
    score_max = optimal_threshold('calc')
    score.append(float(score_max))
    j = j + 1
print('Average value of F1-score is {0} with standard deviation of {1}'.form
```

```
Average value of F1-score is 0.9949 with standard deviation of 0.003
```