Classification and Performance Evaluation Techniques on Imbalanced Datasets

William Young, Ben ElDarragi, and Eric Cheng Hsuan Chiang

1 Introduction and Motivation

A dataset is said to be "imbalanced" when there is a significant, or in some cases extreme, disproportion among the number of samples of each class. Although multi-class problems with skewed class distributions exist, in this project we will focus on addressing imbalance in binary classification problems, using a synthetic dataset of fraudulent and non-fraudulent transactions as a motivating example.

The underrepresented or "minority" class is often of particular interest from a user's perspective. This is almost certainly the case in our fraud application; potential losses that could result from false negatives are likely to far outweigh the minor inconvenience resulting from false positives. Applications in the medical field present an even more extreme demonstration of this phenomenon; one could imagine a scenario where the classification task involved identifying the presence of cancer or health defects, which may be in the minority class of a given dataset.

Standard classification models will typically exhibit bias toward the majority class due to its increased prior probability. As a result, observations in the minority class are more often misclassified than those in the majority one.

2 Review of Existing Literature

Bartosz Krawczyk, in *Learning from Imbalanced Data: Open Challenges and Future Directions*, suggests that we may distinguish three main approaches to learning from imbalanced data [1]:

1. *Data-Level Methods*, which aim to transform an imbalanced training dataset into a balanced one. Data-level methods include undersampling and oversampling. However, this can lead to the removal of important samples or to the introduction of meaningless new objects
2. *Algorithm-Level Methods*, which concentrate on modifying existing learners to alleviate their bias towards majority groups. However, specialized methods are required to use one-class learners for more complex problems.
3. *Hybrid Methods*, which includes making use of ensemble learning algorithms to obtain better performance than what could be obtained from any of the constituent learning algorithms alone. There are additionally some works that propose hybridization of sampling and cost-sensitive learning.

Nguyen, Zeno and Lars suggest that, along with sampling techniques and modifying the classifiers internally, *Cost-Sensitive Learning* ("CSL") can be used to take into account different misclassification costs for false negatives and false positives. The authors proposed two simple methods to deal with class imbalance. The first method combines sampling techniques with CSL to reduce the total misclassification costs of the model. The second method ("CSL-OCRL") optimizes the cost ratio locally and applies this ratio to train the full model.

Fernandez, Garcia, Galar, Krawczyk, and Herrera suggest that Hybrid cost-sensitive learning, which combines cost-sensitive approaches with sampling, or potentially other algorithm-level solutions, are worthwhile as an alternative to the above techniques. One idea lies in conducting a small-scale sampling of the training set before learning a cost-sensitive classifier. This will reduce misclassification costs for minority instances and thus make the search process less biased.

3 Algorithm-Level Methods Used

3.1 Support Vector Machines

Support Vector Machines ("SVMs") are among the most popular models used for classification tasks due to their flexibility and generalization capabilities. SVMs aim to find the optimal hyperplane which separates the observations into two classes; non-linear classification can be achieved via kernel methods, which we will briefly review below.

Let Φ be a non-linear mapping from the original feature space $\mathbb{R}^p \mapsto \mathbb{R}^d$. We can then separate two sets of linearly separable points along a non-linear hyperplane constructed in $\mathbb{R}^d$ defined as the set of points x satisfying

$$w^\top \Phi(x) + b = 0$$

where w is the normal vector to the hyperplane.

In practice, datasets are rarely linearly separable. Thus it is helpful to use the *hinge loss* function:

$$L(w) = \sum_{i=1}^{n} (1 - y_i \phi(x_i)^\top w)_+$$

where the goal of the optimization is to minimize

$$\lambda ||w||^2 + \left[\frac{1}{n} \sum_{i=1}^{n} (1 - y_i \phi(x_i)^\top w)_+ \right]$$

Where the W obtained after training the data is the argument w that minimizes this hinge loss. We can then develop α as

$$\arg\min_{\alpha} \sum_{j} (1 - y(\sum_{j,k} \alpha_j k(x_i, x_j))_+ + \lambda \sum_{j} \sum_{i} \alpha \alpha_j k(x_i, x_j) \tag{1}$$

Obtaining the labels α for this problem typically involves gradient descent or another numerical scheme as there are no closed form solutions.

3.2 Cost-Sensitive Modifications to SVMs

The algorithm-level method of focus in this project is *cost-sensitive learning* via class weighting. Cost-sensitive modifications introduce new misclassification costs to a standard loss functions such as hinge loss. We can thus take our previous loss function and include C_i, a penalty/weight associated with given training instance x_i:

$$\lambda ||w||^2 + \left[\frac{1}{n} \sum_{i=1}^{n} C_i (1 - y_i \phi(x_i)^\top w)_+ \right]$$

This can be configured to result in larger penalties and thus fewer false classifications of the minority set in the training data. The parameter C_i can intuitively be thought as a degree of importance assigned to misclassifications, so it follows that this is a useful quantity to perturb.

Values for class weightings can be solved by appealing to hyper-parameter tuning via an exhaustive method like cross validation grid search or random search.

4 Data-Level Methods Used

As mentioned previously, one of the first mechanisms proposed to address the problem of imbalanced learning was the use of data sampling methods. Broadly, sampling methods fall into one of two categories: *Undersampling Methods* and *Oversampling Methods*. Undersampling methods aim to create a subset of the original dataset by eliminating observations (usually in the majority class), while oversampling methods aim to create a superset of the original dataset by replicating observations or creating new observations from existing ones. Approaches that combine both sampling methods have been proposed, but are beyond the scope of this paper.

Due to the massive size of the transactions dataset used in this project and the breadth of different data-level techniques that can be explored, we chose to focus on undersampling rather than oversampling methods in this project.

4.1 Undersampling Methods

A classical undersampling method is *random undersampling*, which simply eliminates observations of the majority class at random until a final ratio of balancing is reached. This is a relatively efficient approach, but a drawback is that it does not distinguish between observations that are more or less "useful" in training a dataset. Other methods are more discerning regarding the observations from the majority class that are removed. In particular, there are many techniques that aim to identify redundant examples for deletion or useful examples for non-deletion. This paper will examine two such methods, the use of *Tomek Links* and the *Condensed Nearest Neighbor Rule*.

Suppose we have a training dataset X_{train} with n samples, which consists of pairs $E_i = (x_i, y_i), i = 1, ..., n$, where x_i defines a sample in the feature space and y_i defines the associated class label. Let $\hat{E} \subseteq E$ be the subset of selected pairs resulting from executing an undersampling method. Then we can define *Tomek Links* as follows:

Let two examples $E_i = (x_i, y_i)$ and $E_j = (x_j, y_j)$, such that $y_i \neq y_j$ and $d(E_i, E_j)$ is the distance between E_i and E_j. A pair (E_i, E_j) is called a *Tomek Link* if an example E_l does not exist such that $d(E_i, E_l) < d(E_i, E_j)$ or $d(E_j, E_l) < d(E_i, E_j)$ [3].

Tomek Links are a useful tool to clean up overlap between classes and define clearer decision boundaries in the feature space; however, cleaning up overlapping data is often not sufficient enough to rebalance a dataset. For this reason, Tomek Links are often used in combination with the *Condensed Nearest Neighbor Rule* ("CNN"). This method makes use of the 1 Nearest-Neighbor ("1-NN") classifier and is defined as follows:

A subset $\hat{E} \subset E$ is consistent with E if using a 1-NN, $\hat{E}$ correctly classifies the examples in E. Hart's Thinning Algorithm, or the "US-CNN" algorithm, creates such a subset. It works as follows. First, it randomly draws an example from the majority class, then puts this example in $\hat{E}$. Then, it takes *all* the examples from the minority class, and puts them in $\hat{E}$ as well. Afterwards, use a 1-NN over the examples in $\hat{E}$ to classify the examples in E. Every misclassified example from E is moved to $\hat{E}$.[4]

The CNN rule aims to eliminate examples from the majority class that are farther from the decision border. In this project, we aim to test whether this leads to improved classification performance.

5 Results

Accuracies for the different algorithmic schemes are reported in terms of area under the ROC curve (AUC):

Method	Kernel	Accuracy (ROC AUC)
SVM	Gaussian	0.686
Cost-sensitive SVM	Gaussian	0.931

The results demonstrate a clear advantage for using Cost-sensitive SVMs for fitting severely imbalanced data.

We can conclude that the cost-sensitive SVM classified fewer "false positives", as was desired. In contrast, applying Condensed-Nearest Neighbor did *not* yield the same improvement in accuracy. Two error measurements were applied for the rebalanced dataset: the average accuracy measured for each class (the "balanced accuracy"), and the ROC Area Under the Curve. It is also worth mentioning that we were not able to find any "Tomek Links" in the data. The performance results are summarized below:

Method	Dataset	ROC AUC	Balanced Accuracy
SVM (Gaussian Kernel)	Unbalanced	0.686	0.500
SVM (Gaussian Kernel)	Balanced	0.623	0.504

Needless to say, we were quite surprised by these results. Balancing the dataset using CNN yielded negligible increases in balanced accuracy and actually *worsened* ROC AUC. It is likely that, in removing majority class instances that were farther from the decision boundary, our model had a more difficult time distinguishing true positives from false ones. We can also see via the raw balanced accuracy results that there was not a significant increase in instances from the minority class being correctly classified compared to before. The structure of the data, which exhibited some overlap between the two classes, may have made it difficult to yield the desired results.

References

[1] Krawczyk, B. *Learning from Imbalanced Data: Open Challenges and Future Directions*

[2] Nguyen, Zeno and Lars *Cost-Sensitive Learning Methods for Imbalanced Data*

[3] Fernández, Alberto; García, Salvador; Galar, Mikel; Prati, Ronaldo C.; Krawczyk, Bartosz; Herrera, Francisco. *Learning from Imbalanced Data Sets.* Springer International Publishing.

[4] Cover, T.M., Hart, P.E.: *Nearest Neighbor Pattern classification.* IEEE Trans. Inf. Theory, 13 (1967)

CMSC 35300 Project Proposal William Young, Ben ElDarragi, and Eric Chiang

Classification and Performance Evaluation Techniques on Imbalanced Datasets

Description and Initial Literature Review:

Many machine learning problems require training models against data in which the number of samples with one label greatly outnumbers the number of samples with another label. When training models to identify fraud detection, for example, historical observations will typically record a vast majority of transactions as being legitimate, with a tiny portion being labeled as fraudulent. This "imbalance" in label proportions can cause standard classification techniques to perform poorly as they assume a relatively balanced label distribution and equal misclassification costs.

Existing literature on learning from imbalanced datasets is extensive, partially due to the breadth of the topic itself. As such, we will focus on only a select few different evaluation measures and machine learning techniques when confronted with a severely imbalanced dataset of transactions labeled as fraudulent and non-fraudulent. Highly-cited publications include Learning from Unbalanced Datasets by Fernandez, Garcia, Galar, Krawczyk, and Herrera and Imbalanced Learning: Foundations, Algorithms, and Applications by He and Ma. Techniques that we aim to cover include cost-sensitive modifications of Support Vector Machines, data oversampling and undersampling, and "one-class" Support Vector Machines. Evaluation metrics we aim to treat include Area Under the Receiver Operating Characteristic Curve ("ROC AUC"), Precision-Recall Area Under the Curve ("PR AUC"), and Geometric Mean ("G-Mean").

In [ ]:
```
'''
Project Name: Classification and Performance Evaluation Techniques on Imbalanced Datase
File name: Project Code.ipynb
Authors: Ben ElDarragi, William Young, Eric Cheng Hsuan Chiang
Date last modified: 11/28/2021
'''
__authors__ = "Ben ElDarragi, William Young, Eric Cheng Hsuan Chiang"
__version__ = "0.0.1"
__emails__ = ""
```

In [ ]:
```
## imports
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import time
from imblearn.under_sampling import TomekLinks
from imblearn.under_sampling import CondensedNearestNeighbour
from sklearn.decomposition import PCA
from collections import Counter
from numpy import mean
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from matplotlib import pyplot
from mlxtend.plotting import plot_decision_regions
```

Part 1: Preprocessing and Data Exploration

In [ ]:
```
paysim_data=pd.read_csv('paysim_data.csv', nrows=100000)
# first 7 hours of txns
paysim_data = paysim_data[paysim_data['step']<=7]
```

In [ ]:
```
paysim_data
```

Out[ ]:

	step	type	amount	nameOrig	oldbalanceOrg	newbalanceOrig	nameDest	oldbala
0	1	PAYMENT	9839.64	C1231006815	170136.0	160296.36	M1979787155	
1	1	PAYMENT	1864.28	C1666544295	21249.0	19384.72	M2044282225	
2	1	TRANSFER	181.00	C1305486145	181.0	0.00	C553264065	
3	1	CASH_OUT	181.00	C840083671	181.0	0.00	C38997010	
4	1	PAYMENT	11668.14	C2048537720	41554.0	29885.86	M1230701703	
...	...	...	...	...	...	...	...	...
13996	7	PAYMENT	707.83	C853730833	216082.0	215374.17	M2118675919	
13997	7	PAYMENT	7348.55	C200335811	39686.0	32337.45	M1693375200	
13998	7	PAYMENT	8594.91	C1100577282	80059.0	71464.09	M2088911339	
13999	7	PAYMENT	5750.32	C1894795585	691.0	0.00	M1650177125	
14000	7	PAYMENT	12856.87	C1754831686	0.0	0.00	M108728999	

14001 rows × 11 columns

In [ ]:
```
# percent transactions fraud vs. non-fraud
paysim_data['isFraud'].value_counts(normalize=True)
```

Out[ ]:
```
0    0.994429
1    0.005571
Name: isFraud, dtype: float64
```

In [ ]:
```
paysim_data_numerical = paysim_data.copy()
```

In [ ]:
```
################################################################################
# numerically encode all categorical data: account ids and transaction types
################################################################################

labels = paysim_data['type'].astype('category').cat.categories.tolist()
replace_map_comp = {'type' : {k: v for k,v in zip(labels,list(range(1,len(labels)+1)))}
paysim_data_numerical.replace(replace_map_comp, inplace=True)

paysim_data_numerical['nameOrig'] = paysim_data_numerical['nameOrig'].astype('category'
paysim_data_numerical['nameOrig'] = paysim_data_numerical['nameOrig'].cat.codes

paysim_data_numerical['nameDest'] = paysim_data_numerical['nameDest'].astype('category'
paysim_data_numerical['nameDest'] = paysim_data_numerical['nameDest'].cat.codes
```

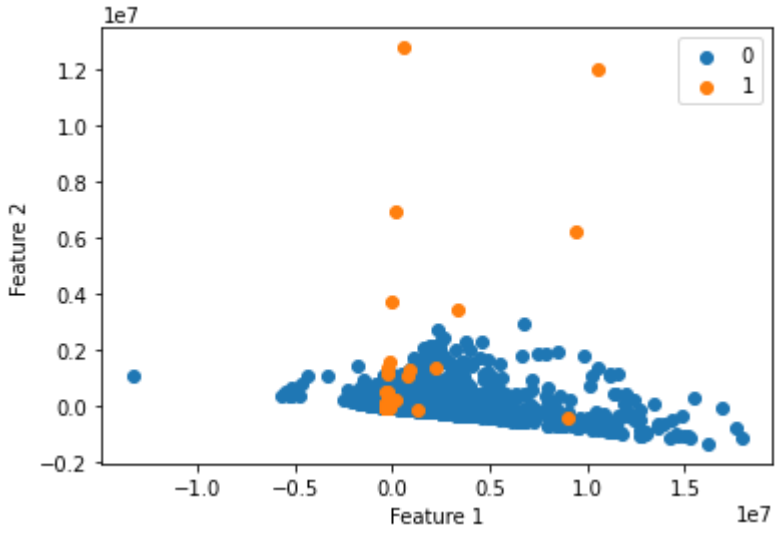
In [ ]:
```
################################################################################
# the change between old and new balances is much more likely to be useful to a model r
```

```
##############################################################################################
paysim_data_numerical['destBalanceChg'] = paysim_data_numerical['newbalanceDest'] - pay
paysim_data_numerical['origBalanceChg'] = paysim_data_numerical['newbalanceOrig'] - pay
```

In [ ]:
```
paysim_data_numerical.drop(labels=['oldbalanceOrg', 'newbalanceOrig', 'oldbalanceDest',
```

In [ ]:
```
y = paysim_data_numerical['isFraud'].to_numpy()
yhat = paysim_data_numerical['isFlaggedFraud'].to_numpy()
X = paysim_data_numerical[['step', 'type', 'amount', 'nameOrig', 'nameDest', 'destBalan
```

In [ ]:
```
##############################################################################################
# plot the majority and minority classes in a reduced feature space
##############################################################################################
X_transformed = np.copy(X)
pca = PCA()
X_transformed = pca.fit_transform(X_transformed)
counter = Counter(y)
# scatter plot of examples by class label
for label, _ in counter.items():
        row_ix = np.where(y == label)[0]
        pyplot.scatter(X_transformed[row_ix, 0], X_transformed[row_ix, 1], label=str(la
pyplot.xlabel('Feature 1')
pyplot.ylabel('Feature 2')
pyplot.legend()
pyplot.show()
```



Part 2: Undersampling Methods

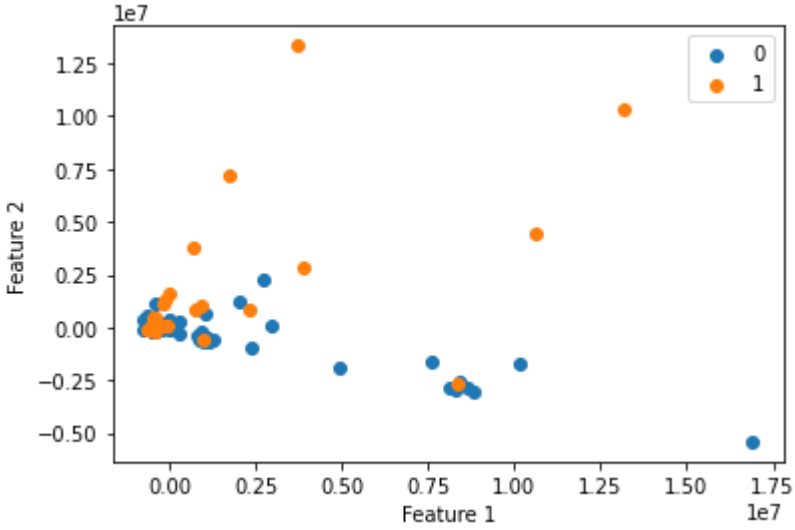
In [ ]:
```
##############################################################################################
# identify any tomek links for removal
##############################################################################################
X_resampled, y_resampled = np.copy(X), np.copy(y)
tomek_links = TomekLinks()
start = time.time()
undersample = tomek_links.fit_resample(X_resampled, y_resampled)
end = time.time()
counter_before = Counter(y)
counter_after = Counter(y_resampled)
print(f"{counter_after[0] - counter_before[0]} Tomek Links were found in {(end-start):.
```

```
0 Tomek Links were found in 0.19 seconds
```

In [ ]:
```
##############################################################################################
# use Hart's CNN algorithm to remove majority class instances
##############################################################################################
cnn = CondensedNearestNeighbour(n_neighbors=1)
start = time.time()
X_resampled, y_resampled = cnn.fit_resample(X_resampled, y_resampled)
end = time.time()
counter_after = Counter(y_resampled)
print(f"Majority class observations reduced from {counter_before[0]} instances to {coun
```

```
Majority class observations reduced from 13923 instances to 251 instances in 100.66 seco
nds.
```

In [ ]:
```
##############################################################################################
# plot the majority and minority classes in a reduced feature space in the rebalanced d
##############################################################################################
X_transformed = np.copy(X_resampled)
pca = PCA()
X_transformed = pca.fit_transform(X_transformed)
counter = Counter(y_resampled)
# scatter plot of examples by class label
for label, _ in counter.items():
        row_ix = np.where(y_resampled == label)[0]
        pyplot.scatter(X_transformed[row_ix, 0], X_transformed[row_ix, 1], label=str(la
pyplot.xlabel('Feature 1')
pyplot.ylabel('Feature 2')
pyplot.legend()
pyplot.show()
```



Part 3: Support Vector Classification and Cost-Sensitive Modifications

To train a support vector machine, we can use $\nabla_w f = \lambda w - \sum_{i|y_i(wx_i)<1}^{m} y_i x_i$:

In [ ]:
```
import random

def svm(X, y, lam, n, p):
  w = np.zeros((p,))
  tau = 0.2
  runs = 100
  for t in range(1, runs+1):
```

```
    for i in range(n):
      if (y[i]*np.dot(X[i,:], w)) < 1:
        w = w + tau * ((X[i, :] * y[i]) + (-2 * (1/(t*lam)) * w))
      else:
        w = w + tau * (-2 * (1/(t*lam) * w))
  return w

n, p = X.shape
w = svm(X, y, 1, n, p)
print(w)
```

```
[ 3.65549298e-007  2.61106641e-007  1.13471724e-003  2.60166657e-004
  7.40498435e-005  5.97819431e-322 -1.13471724e-003]
```

In [ ]:
```
y_hat = np.sign(X@w)
```

In [ ]:
```
##########################################################################################
# an unmodified SVM predicts all elements to be 1! Shows the inadequacy of a linear SVM
##########################################################################################

unique_elements, counts_elements = np.unique(y_hat, return_counts=True)
unique_elements, counts_elements
```

Out[ ]:
```
(array([1.]), array([14001]))
```

In [ ]:
```
##########################################################################################
# baseline performance of SVC on imbalanced dataset using k-fold cross validation
##########################################################################################

# define model and train
model = SVC(kernel='rbf', gamma='scale')
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
score = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)

# performance
print('Average ROC AUC: %.3f' % mean(score))
```

```
Average ROC AUC: 0.686
```

In [ ]:
```
##########################################################################################
# performance of SVC on dataset rebalanced using Condensed Nearest Neighbor
##########################################################################################

# get accuracy per class on unbalanced dataset
model = SVC(kernel='rbf', gamma='scale')
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
score = cross_val_score(model, X, y, scoring='balanced_accuracy', cv=cv, n_jobs=-1)

# balanced dataset accuracy per class and ROC AUC
model_balanced = SVC(kernel='rbf', gamma='scale')
cv_balanced = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
balanced_score = cross_val_score(model_balanced, X_resampled, y_resampled, scoring='bal

model_roc = SVC(kernel='rbf', gamma='scale')
cv_roc = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
roc_score = cross_val_score(model_roc, X_resampled, y_resampled, scoring='roc_auc', cv=

# performance
print('Average Accuracy per Class, Unbalanced Dataset: %.3f' % mean(score))
```

```
print('Average Accuracy per Class, Balanced Dataset: %.3f' % mean(balanced_score))
print('Area under ROC Curve, Balanced Dataset: %.3f' % mean(roc_score))
```

```
Average Accuracy per Class, Unbalanced Dataset: 0.500
Average Accuracy per Class, Balanced Dataset: 0.507
Area under ROC Curve, Balanced Dataset: 0.616
```

In [ ]:

```
# enabling cost sensitive SVC via class weighting settings within the model
# define model and train
model = SVC(kernel='rbf', gamma='scale', class_weight={0: 1, 1: 10})
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
score = cross_val_score(model, X, y, scoring='roc_auc', cv=cv, n_jobs=-1)

# performance
print('Average ROC AUC: %.3f' % mean(score))
```

```
Average ROC AUC: 0.931
```

In [ ]:

```
# enabling cost sensitive SVC via class weighting through Grid Search
# define model
model = SVC(kernel='rbf', gamma='scale')

# define grid
balance = [{0:100,1:1}, {0:10,1:1}, {0:1,1:1}, {0:1,1:10}, {0:1,1:100}]
param_grid = dict(class_weight=balance)
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# grid search
grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=-1, cv=cv, scoring='
grid_result = grid.fit(X, y)

# print best class grid
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
```

```
Best: 0.930918 using {'class_weight': {0: 1, 1: 10}}
```