Feature Importance Calculation with Oversampling Methods In case of Imbalanced data

Gábor Vitrai

Department of Data Science and Engineering
Faculty of Informatics, ELTE - Eötvös Loránd University in Budapest
abiowe@inf.elte.hu

Abstract: The paper describes a brute force method, to see how effective oversampling methods are, to calculate the importance of each feature using subsets of features. This way we can check whether a feature's importance will change dramatically if we remove an other one. During the process we will measure how the scores of each subset will change, and we will see a custom evaluation of importance changes. The final target of the paper is to see, how similar results can we get as an original feature importance set of a Random Forest Classifier...

1 Theory

With this chapter, after the aim of the paper, we will describe some necessary definitions we aim to use later in the paper.

1.1 The General Aim

In this subsection, we describe the general aim of the algorithm. We know, that a simple Random Forest Classifier, - for example - can rank features by its importance in terms of classification. However, in the case of imbalanced data these numbers can be misguiding. As an advanced tool, we decided to see how oversampling on different subsets of a data set can help to get a better plot about the importance of each feature in case of oversampling. The target is to set up an evaluation method which results in a similar importance list/plot as the original one.

1.2 Definitions

There are a few non trivial definitions we wanted to discuss in this subsection.

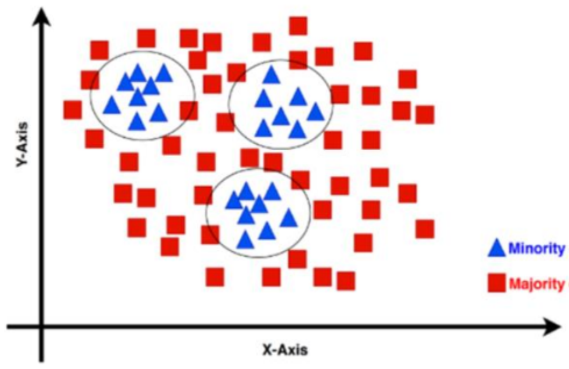
Majority Class: The class (or classes) in an imbalanced classification predictive modeling problem that has many examples.

Minority Class: The class in an imbalanced classification predictive modeling problem that has few examples

Copyright ©2021 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Disjoint subsets: In mathematics, two sets are said to be disjoint sets if they have no element in common.This is a well known weakness of the original SMOTE algorithm.

Figure 1: Small disjoint subsets of minority class



Class overlapping imbalance: Is caused due to ambiguous regions in the data where the prior probability of two or more classes are approximately equal

1.3 SMOTE Variants

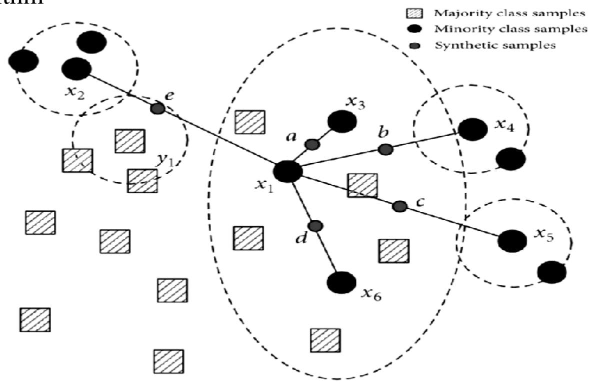
In this sub chapter we plan to describe the logic behind some variants of **Synthetic Minority Oversampling Technique** (SMOTE).
SMOTE is the most popular oversampling method that was proposed to improve random oversampling. There are multiple variations of SMOTE which aim to combat the original algorithm's weaknesses. Yet, many of these approaches are either very complex or alleviate only one of SMOTE's shortcomings.

SMOTE: The original algorithm works the following way:
1. Choose a minority class input vector.
2. Find its K-nearest neighbors
3. Choose one of these neighbors and place a synthetic point anywhere on the line joining the point under consideration and its chosen neighbor.

This has the advantage of being effective for between-class imbalance scenarios, but its weak for within-class imbalance and small disjoints. (See Figure 1 as an example for disjoints.)

Note: The following variants were included into this paper as they are suitable for the data set described later

Figure 2: Graphical Representation of the SMOTE algorithm



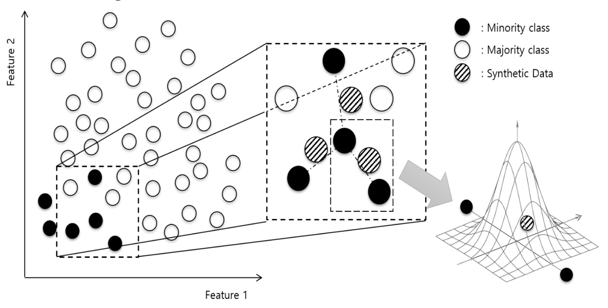
Borderline_SMOTE1: The technique replaces SMOTE's random selection with a targeted selection of instances close to the class border. It does not employ a clustering procedure.

Borderline_SMOTE2: Extends the Borderline-SMOTE1 method to allow interpolation of a minority instance and one of its majority class neighbours.

Cluster-SMOTE: Goal of this method is to boost class regions by creating samples within naturally occurring clusters.
• Emphasizes certain class regions.
• Uses k-means to cluster the minority class before applying SMOTE.
• It does not help to eliminate small disjoints.

Gaussian_SMOTE: The basic underlying principles of the algorithm are same as the SMOTE method:
1. Compute differences between the minority class data and its randomly selected nearest neighbor.
2. After that, the Gaussian-based SMOTE method draws a number between 0 and difference value for roughly estimating a location of a synthetic candidate.
3. As a next step, another number draws from Gaussian (or normal) distribution with heuristically selected parameter.
4. Finally, a synthetic data is generated by using derived parameters.

Figure 3: The operation principle of the Gaussian-based SMOTE algorithm.



2 Data set Selection

Upon starting the project, we wanted a data set which is well known, and has some noisy data. The data sets are from the KEEL repository and from Kaggle.

2.1 Experimenting with data sets

After some research, we selected 3 different data sets with proper number of attributes. The binary version of the "glass", "vehicle0" and "heart disease" data set. We imported and cross validated each of them using **Repeated Stratified K-Fold** with 5 splits and 5 repeats. While a Random Forest could score around 90% accuracy on the *glass* and *vehicle0* data sets, the *heart disease* proved to be the toughest (77,5%), so we decided to use it to test the algorithm proposed by this paper.

Table 1: Details about the datasets

Dataset	Features	Instances	Imbalance Ratio
heart disease	13	270	1.25
glass	9	215	2.06
vehicle0	18	846	3.25

Table 2: Results of Cross Validation with RF Classifier

Dataset	Accuracy	F1 score	Precision	Recall
heart dis.	**0.833**	**0.807**	**0.844**	**0.767**
glass	0.960	0.887	0.932	0.842
vehicle0	0.972	0.941	0.931	0.946

$$accuracy = \frac{true_n + true_p}{(true_p + false_p + true_n + false_n)}$$

$$precision = \frac{true_p}{(true_p + false_p)}$$

$$recall = \frac{true_p}{(true_p + false_n)}$$

$$f1 = 2 * \frac{precision * recall}{precision + recall}$$

2.2 Introduction to the Heart Disease data set

This database contains 76 attributes, but all published experiments refer to using a subset of 14 of them. (Just like in our case) The "goal"/"Heart Disease" field refers to the presence of heart disease in the patient. It is integer valued 0 or 1. The 14 features include, properties of patients like: Age, Sex, Chest pain, Maximum Heart Rate, or EKG results.

3 The Algorithm

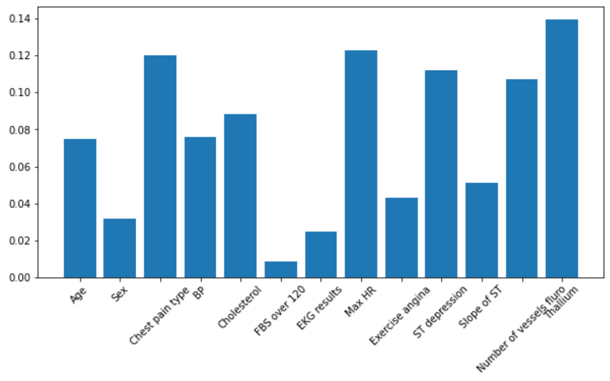
In this subsection, we describe the general process of the algorithm. First, we **normalize** the data. Afterwards we filter out features with less then 0.05 importance. This is needed as the algorithm requires a lot of processing power, so optimization is very important.
Formula of Normalization:

$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}.$

3.1 Original Feature Importances

The plot below, shows all the features in the data set. It can be seen well, that if we apply the threshold, we will loose 4 features. As we stated above, this is important as the next step is to create subsets of these features. Each of these subsets will run a sampler which is very exhaustive.

Figure 4: Original Feature Importances



3.2 Creating Subsets - Properties of Subsets

To apply the method, we created a function which returns all the possible combinations of attributes without repetition, ignoring the order of elements. We also decided to set the number minimum elements in a subset to 4 as subsets with less then 4 features are less likely to have any effect.

```
Data: Set of features
Result: structured collected data inside a dictionary
initialization;
combinations = [];
for every L feature do
    for subset in
      itertools.combinations(feature$_s$et, L) : do
        combinations.append(subset);
        output(subset);
    end
end
```

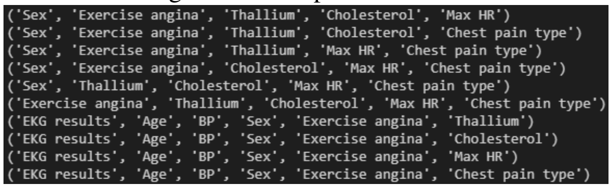
Algorithm 1: How to generate subsets with the give criteria

See examples of combination generation with numbers form 1 to 4:

```
(1, 2)          (1, 3)
(1, 4)          (2, 3)
(2, 4)          (3, 4)
(1, 2, 3)       (1, 2, 4)
(1, 3, 4)       (2, 3, 4)
(1, 2, 3, 4)
```

Given this example we can easily imagine how the real result would look like:

Figure 5: Example for Subsets



```
('Sex', 'Exercise angina', 'Thallium', 'Cholesterol', 'Max HR')
('Sex', 'Exercise angina', 'Thallium', 'Cholesterol', 'Chest pain type')
('Sex', 'Exercise angina', 'Thallium', 'Max HR', 'Chest pain type')
('Sex', 'Exercise angina', 'Cholesterol', 'Max HR', 'Chest pain type')
('Sex', 'Thallium', 'Cholesterol', 'Max HR', 'Chest pain type')
('Exercise angina', 'Thallium', 'Cholesterol', 'Max HR', 'Chest pain type')
('EKG results', 'Age', 'BP', 'Sex', 'Exercise angina', 'Thallium')
('EKG results', 'Age', 'BP', 'Sex', 'Exercise angina', 'Cholesterol')
('EKG results', 'Age', 'BP', 'Sex', 'Exercise angina', 'Max HR')
('EKG results', 'Age', 'BP', 'Sex', 'Exercise angina', 'Chest pain type')
```

3.3 Optimization - Getting the best Samplers

Finding the best sampler variant is very computationally heavy. That is why we had to make a sacrifice not to run all the possible SMOTE variants on every subset, but run the only the best 5 or 3 algorithm on every subset. And while this might result in using some variants on subsets where its not the best for the given features, the final results still turned out to be absolutely acceptable (See final plots). For the *Heart Disease* data set, the top 3 suitable SMOTE variants were calculated to be: **Borderline_SMOTE1, Gaussian_SMOTE, SPY**

3.4 Data gathering - Saving valuable results

At this point, we had the combinations to work with, it is time to gather the numeric data we needed. For every combination we run the following steps:

```
Data: Combinations
Result: Structured data about oversampling
initialization;
output = [];
for each combination do
    split data into X_train, X_test, y_train, y_test;
    declare a RandomForestClassifier instance;
    fit the classifier to the training set;
    save score of classifier;
    save feature importances from the give subset;
    find the most suitable over sampler for the
      given subset;
    use it to over sample;
    fit the classifier to the over sampled training set;
    save score of classifier;
    save feature importances from the give subset;
    if score improvement is better then delta (0.02)
      then
        save combination to "combination";
        save type of sampler used to "sampler";
        save the original feature importance to
          "original_feature_importance";
        save the new feature importance to
          "feature_importance";
        save the original score to "original_score";
        save the new score to "score";
    end
end
```

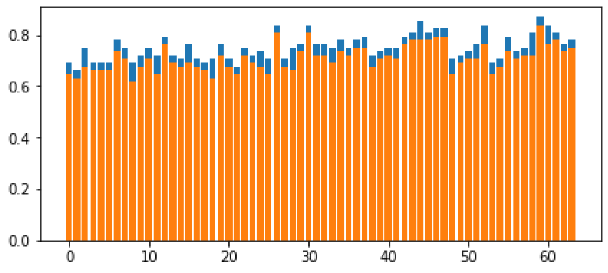
Algorithm 2: Collecting information about subsets

We split the data into Train and Test sets, and train a Random Forest Classifier to get a score about it's predicting ability. (test set size = 0.25) The next thing we save before oversampling is saving the feature importacnes.
Next, we run the built in method of the SMOTE package, to find the best variant for the given subset. The selected sampler then is used to sample data.
The final step in data collection is to save the same attributes before oversampling, thus the sampler which was used, as well as the combination, the scores and importances were generated with.
All of these information were saved into a dictionary. The process in our example took 23 minutes, running on 382 different combinations.

3.5 Examining improved subsets

Out of 382 only 64 combinations had more then 2% improvement, where the average improvement was 2% with 8% maximum.

Figure 6: Improvements on the subsets



3.6 Point distribution results

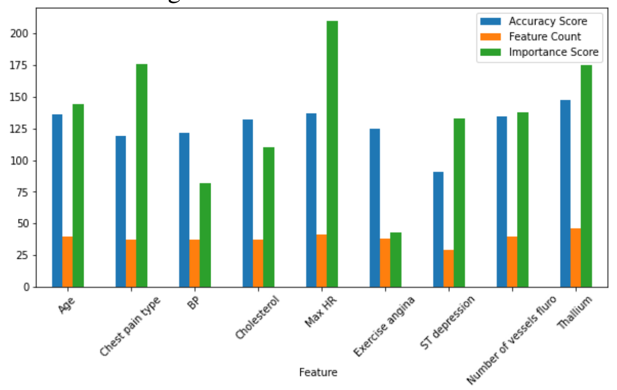
In this subsection, we will write about how we evaluated the collected data. We decided to give "points" for every aspect individually and weight them similarly. For accuracy, each combination can get points between 1-5 for every percentage of improvement. Negative changes were filtered out, but later the model can be improved with penalty points.
In the case of importances, for every combination, we ranked the importance of each feature inside every combination and as a next step, we assign points for every combination the following way: *If a given feature got better rank then before oversampling*, the algorithm rewards it with 2 extra points, aside the the 1-5 points for its rank/position. If a feature is turned out to be the most important, it would get 5 points. So if the feature was originally the 2nd most important feature, but now its the 1st, it will get 5+2 points. The number of usage of each sampler is also counted, more about this later.
Finally, when the evaluation method finished, we have a dictionary with all the possible features, graded with points. The structure looks the following way:

```
{
    'Age':
        {
        'accuracy_score': 136,
        'importance_score': 144,
        'feature_appearance': 40
        },
    'Chest pain type':
        {
        'accuracy_score': 119,
        'importance_score': 176,
        'feature_appearance': 37
        },
    'Cholesterol':
        {
        'accuracy_score': 132,
        'importance_score': 110,
        'feature_appearance': 37
        },...
}
```

3.7 Evaluating the scores

In this subsection we will introduce how the "points" by our simple weighting system correlates to each other and some plots will also be presented to prove the achieved results. First, we wanted to showcase that the results are meaningful, and for each feature, we have reasonable points collected compared to the number of occurrences pf the features.

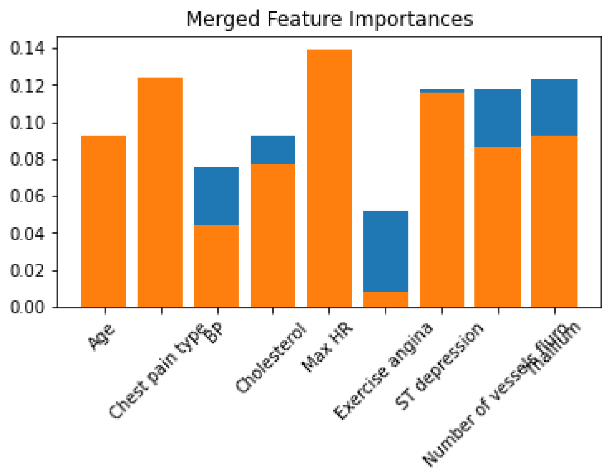
Figure 7: Feature Point Results



Here we can see, how the Accuracy Points (Blue) and the Feature Importance Points (Green) relates in the case of each feature. We can also see, that the Orange columns has almost the same heights in the case of every feature. This is what we would expect. This means that In case of the results, where improvement happened during oversampling, each feature appeared almost in the same number of cases. (Counter example can be the "ST depression" feature, where we see that if we do oversampling on a subset including this attribute, the results are less likely to change).

The next step is to create a normalized view from these results. To do this for every feature, we add together the points of the accuracy and feature importance and divide it with then number of occurrences. The result is a bar plot with feature importances taking into consideration the accuracy changes and the importacnes (and their changes) rank's. This plot is not on the same scale as an importance plot of a Random Forest Classifier, so after using a Min Max Scaler, set to scale to the same interval, the result looks like the same.

Figure 8: Feature Importances by the proposed algorithm



The plot shows the original Feature Importances by the RF Classifier with blue, and the new ones with Orange. We can see, how the numbers are extremely close in some cases.

3.8 Conclusion

As we can see on the plot above, the method works well, as it produces a very similar result as the built in function of the RF Classifier. The differences might be caused by the feature selection in the very beginning. This selection could be removed to see the missing importances getting better, however this would be computationally expensive. As we have less combinations where a feature appears, the importance of that given feature might decrease.
The algorithm proved to produce a similar importance list with the technique of oversampling subsets.

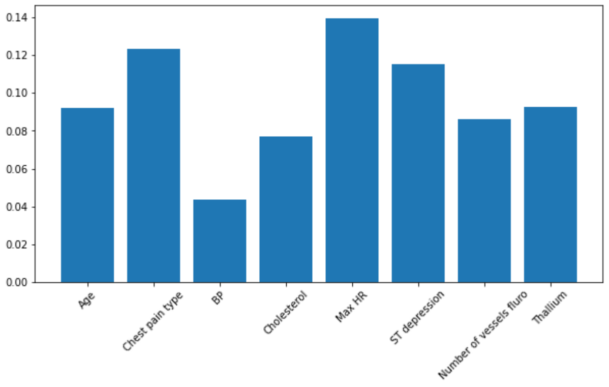
3.9 Testing the Result

The results are similar in terms of feature importance, but lets see how to method works when the results are used for classification. The cross validated results are the following:

Table 3: Results of Classification

Description	Results
The original dataset	82.4%
Manual Feature Selection	81.2%
Feature Selection with the introduced method	81.0%

Figure 9: The final importances of the used classifier



As we can see on the plot above, the algorithm filtered out an additional column as it did not prove to have enough importance. This seemed to be a less important attribute however as the table shows, the cross validated results proved to be less accurate due to this extra feature removal. The result are still very close to the original one, however these numbers might vary.
The biggest drawback of the algorithm is the computational requirement, as running oversampling on all the subsets takes a lot of time. On the other hand, the algorithm showed a pretty accurate prediction with feature importances on imbalanced datasets.

3.10 Possible Improvement of the method

There are many possible ways of improving the method, in case of computational optimizations. The algorithm can be used to plot more information about the oversampling of imbalanced data. A useful measure could be added, where we would drop different features and we would measure how the given combination modifies, gets batter or worse.

References

[1] SMOTE-variants by György Kovács https://github.com/analyticalmindsltd/smote_variants.

[2] Gaussian-SMOTE: Hansoo Lee and Jonggeun Kim and Sungshin Kim, "Gaussian-Based SMOTE Algorithm for Solving Skewed Class Distributions" , Int. J. Fuzzy Logic and Intelligent Systems, 2017, pp. 229-234

[3] Dang, X. T. and Tran, D. H. and Hirose, O. and Satou, K., "SPY: A Novel Resampling Method for Improving Classification Performance in Imbalanced Data" , 2015 Seventh International Conference on Knowledge and Systems Engineering (KSE), 2015, pp. 280-285

[4] Majority and Minority class definitions: https://machinelearningmastery.com/what-is-imbalanced-classification/

[5] KEEL Repository: https://sci2s.ugr.es/keel/

[6] Heart Disease Dataset from KEEL: https://sci2s.ugr.es/keel/dataset.php?cod=99