Over_Sampling_for_Time_Series_Classification

Lan Wei

2017-08-23

Introduction

The package provides functions to deal with binary classification problems in time series imbalanced classes. Synthetic balanced samples are generated by approaches ESPO and ADASYN.

ESPO is short for enhanced structure preserving oversampling, which is synergistically combined with interpolation-based oversampling ADASYN to form OSTSC. "ESPO is used to generate a large percentage of the synthetic minority samples based on multivariate Gaussian distribution, by estimating the covariance structure of the minority-class samples and by regularizing the unreliable eigen spectrum."[1]

Given the positive data $P = \left\{x_{11}, x_{12}, ..., x_{1|P|}\right\}$ and the negative data $N = \left\{x_{01}, x_{02}, ..., x_{0|N|}\right\}$, where $|N| \gg |P|$, $x_{ij} \in \mathbb{R}^{n\times 1}$, the new samples will be generated in the following steps.

1. Removal of Common Null Space

Using $q_{ij} = L_s^T x_{ij}$ to represent x_{ij} in a lower-dimensional signal space, where L_s consists of eigenvectors in the signal space.

2. ESPO

Given $\hat{D}$ is the diagonal matrix of regularized eigen values $\left\{\hat{d_1}, ..., \hat{d_n}\right\}$, V is the eigenvector matrix from the positive-class covariance matrix, $\hat{F} = V\hat{D}^{-1/2}$, $\bar{q_1}$ is the corresponding positive-class mean vector, $z = \hat{F}(b - \bar{q_1})$, the sample in the signal space is computed by $b = \hat{D}^{1/2}V^T z + \bar{q_1}$

3. ADASYN

ADASYN is executed in the transformed signal for the efficiency. Given the transformed positive data $P_t = \{q_{1i}\}$ and negative data $N_t = \{q_{0j}\}$, each sample q_{1i} is duplicated in ratio $\Gamma_i = |S_{i:k-NN} \bigcap N_t| / Z$, where $S_{i:k-NN}$ is this sample's kNN in the entire dataset, Z is a normalization factor to make $\sum_{i=1}^{|P_t|} \Gamma_i = 1$.

More details about the theories can be read in the referenced article.

References


```
H. Cao, X.-L. Li, Y.-K. Woon and S.-K. Ng,
"Integrated Oversampling for Imbalanced Time Series Classification"
IEEE Trans. on Knowledge and Data Engineering (TKDE),
vol. 25(12), pp. 2809-2822, 2013
```


Examples

First of all, we need to load some data for oversampling. The dataset_synthetic_control is included in the OSTSC package, which is generated by the process in Alcock and Manolopoulos (1999) (via). The dataset has already split training and testing data, feature and label data.

[1] H. Cao, X.-L. Li, Y.-K. Woon and S.-K. Ng, Integrated Oversampling for Imbalanced Time Series Classification. 2013

```
library(OSTSC)
data(dataset_synthetic_control)

train_label <- dataset_synthetic_control$train_y
train_sample <- dataset_synthetic_control$train_x
test_label <- dataset_synthetic_control$test_y
test_sample <- dataset_synthetic_control$test_x
```

The imbalance of synthetic_control data is 1:10. The time series sequences recorded body moving sensor data. Class 1 aims to Normal status, while class 0 aims to Cyclic, Increasing trend, Decreasing trend, Upward shift and Downward shift. The imbalance of the train dataset is shown below.

```
table(train_label)
```

```
## train_label
##   0   1
## 250  25
```

This is a simple example to show how to oversample the class 1 to the same amount of class 0, and export the sample and label from oversampled data. There are ten parameters in the OSTSC function, the details of them can be read in the help documents. Users only need to input at least first three variables to be able to call the function.

```
MyData <- OSTSC(train_sample, train_label, target_class = 1)
over_sample <- MyData$sample
over_label <- MyData$label
```

Now the positive data and negative data are balanced. Let's check the (im)balance of new dataset.

```
table(over_label)
```

```
## over_label
##   0   1
## 250 250
```

Here an Long short-term memory (LSTM) classifier is used to analysis the performance of the OSTSC approach. Using R package keras, to build a LSTM classifier to do time series data classification is effectively and fast.

For comparison, first to determine how does the classifier perform on the original data before oversampling.

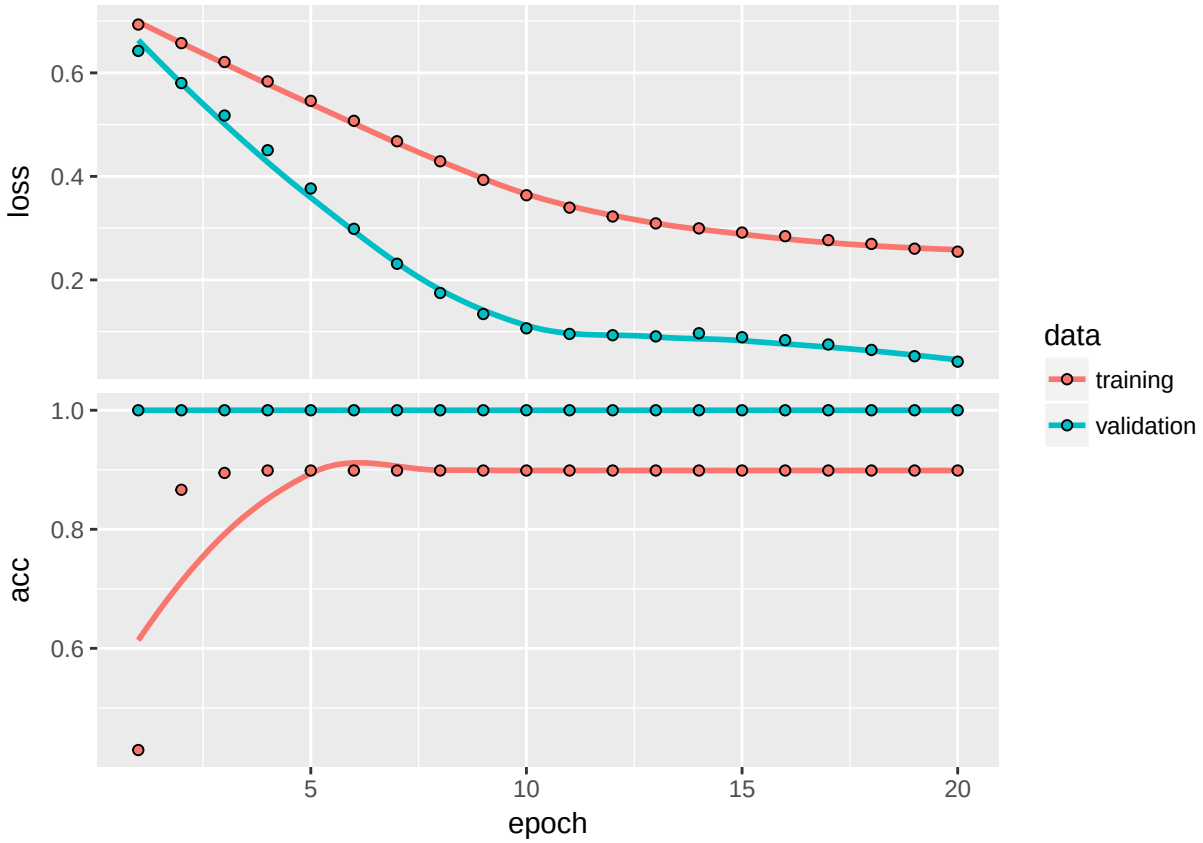
1. One-hot encode the label vectors into binary class matrices using the Keras to_categorical() function. And transform the sample array to 3-dimension for LSTM.

```
library(keras)
train_y <- to_categorical(train_label)
test_y <- to_categorical(test_label)
train_x <- array(train_sample, dim = c(dim(train_sample),1))
test_x <- array(test_sample, dim = c(dim(test_sample),1))
```

2. Initialize a sequential model. Add layers to the model. Compile the model. Store the fitting history and show the plot.

```
model = keras_model_sequential()
model %>%
  layer_lstm(10, input_shape = c(dim(train_x)[2], dim(train_x)[3])) %>%
  layer_dense(dim(train_y)[2]) %>%
  layer_activation("softmax")
model %>% compile(
```

```
  loss = "categorical_crossentropy",
  optimizer = "adam",
  metrics = "accuracy"
)
lstm_before <- model %>% fit(
  x = train_x,
  y = train_y,
  validation_split = 0.1,
  epochs = 20
)
plot(lstm_before)
```



3. Evaluate the model.

```
score <- model %>% evaluate(test_x, test_y)
```

```
## The loss value is 0.2317213 .
```

```
## The metric value (in this case 'accuracy') is 0.9090909 .
```

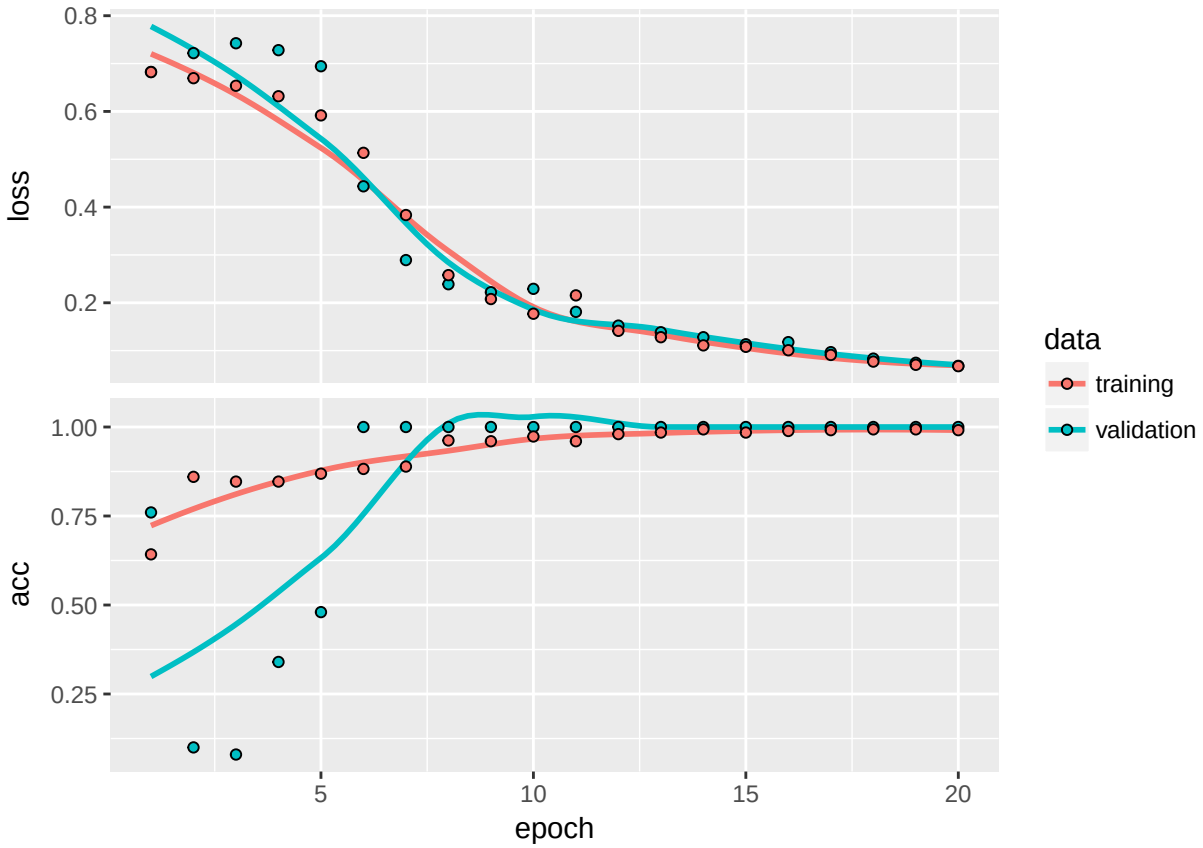
Then to determine how does the classifier perform on the new data after oversampling.

1. One-hot encode the label vectors into binary class matrices using the Keras to_categorical() function. And transform the sample array to 3-dimension for LSTM.

```
over_y <- to_categorical(over_label)
over_x <- array(over_sample, dim = c(dim(over_sample),1))
```

2. Initialize a sequential model. Add layers to the model. Compile the model. Store the fitting history and show the plot.

```
model_over = keras_model_sequential()
model_over %>%
  layer_lstm(10, input_shape = c(dim(over_x)[2], dim(over_x)[3])) %>%
  layer_dense(dim(over_y)[2]) %>%
  layer_activation("softmax")
model_over %>% compile(
  loss = "categorical_crossentropy",
  optimizer = "adam",
  metrics = "accuracy"
)
lstm_after <- model_over %>% fit(
  x = over_x,
  y = over_y,
  validation_split = 0.1,
  epochs = 20
)
plot(lstm_after)
```



3. Evaluate the model.

```
score_over <- model_over %>% evaluate(test_x, test_y)
```

```
## The loss value is 0.0840981 .
```

```
## The metric value (in this case 'accuracy') is 1 .
```

Besides the loss and accuracy, let's compare the confusion matrices. The original data trained model has severe mis-classification on positive data, while after oversampling this flaw has been corrected.

```
pred_label <- model %>% predict_classes(test_x)
pred_label_over <- model_over %>% predict_classes(test_x)
cm_before <- table(test_label, pred_label)
cm_after <- table(test_label, pred_label_over)
```

```
## The confusion matrix before oversampling:
```

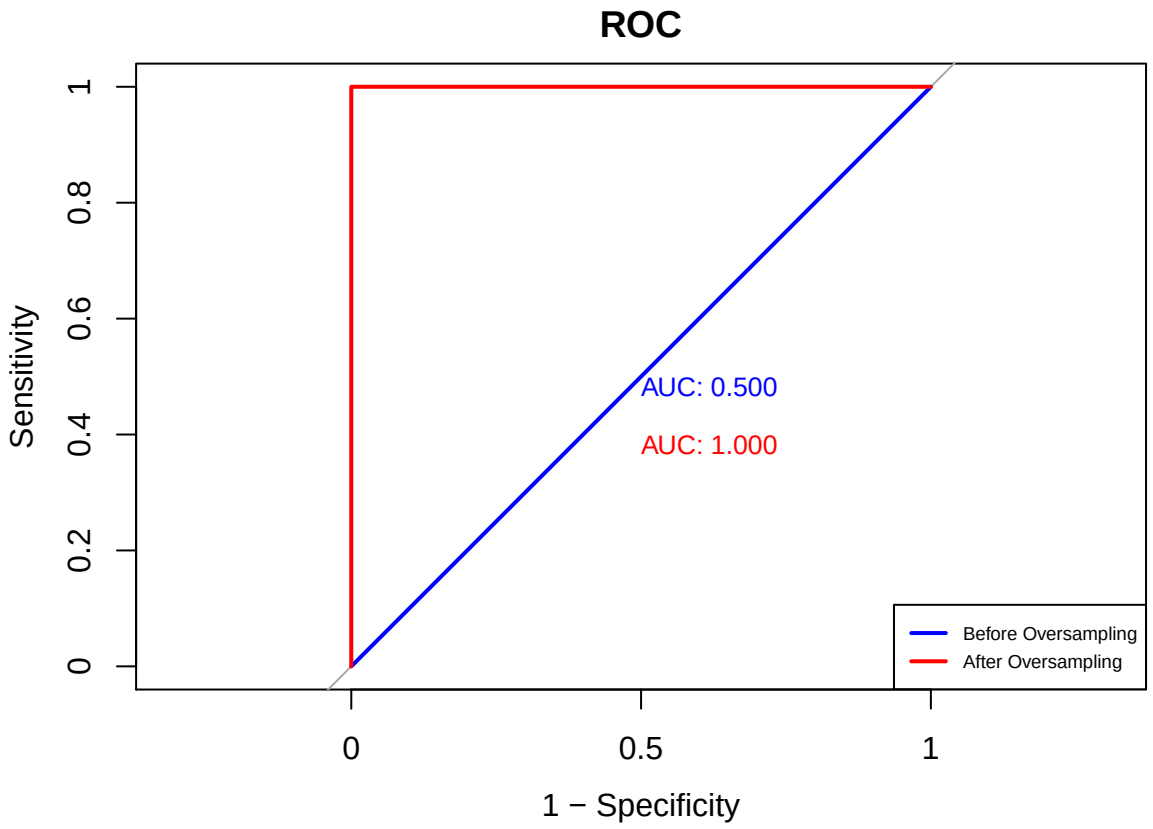
```
##           pred_label
## test_label   0
##          0 250
##          1  25
```

```
## The confusion matrix after oversampling:
```

```
##           pred_label_over
## test_label   0   1
##          0 250   0
##          1   0  25
```

The ROC plot tells the same.

```
plot.roc(test_label, pred_label, legacy.axes = TRUE, col = "blue", print.auc = TRUE,
         main="ROC", print.auc.cex= .8)
plot.roc(test_label, pred_label_over, legacy.axes = TRUE, col = "red", print.auc = TRUE,
         print.auc.y = .4, print.auc.cex= .8, add = TRUE)
legend("bottomright", legend=c("Before Oversampling", "After Oversampling"),
       col=c("blue", "red"), lwd=2, cex= .6)
```



The power of OSTSC will come out more with larger size or extremely imbalanced datasets. The examples on larger datasets are provided in demo().