6. Miscellaneous samplers

6.1. Custom samplers

A fully customized sampler, `FunctionSampler`, is available in imbalanced-learn such that you can fast prototype your own sampler by defining a single function. Additional parameters can be added using the attribute `kw_args` which accepts a dictionary. The following example illustrates how to retain the 10 first elements of the array `X` and `y`:

```
>>> import numpy as np
>>> from imblearn import FunctionSampler
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=5000, n_features=2,
n_informative=2,
...                            n_redundant=0, n_repeated=0,
n_classes=3,
...                            n_clusters_per_class=1,
...                            weights=[0.01, 0.05, 0.94],
...                            class_sep=0.8,
random_state=0)
>>> def func(X, y):
...   return X[:10], y[:10]
>>> sampler = FunctionSampler(func=func)
>>> X_res, y_res = sampler.fit_resample(X, y)
>>> np.all(X_res == X[:10])
True
>>> np.all(y_res == y[:10])
True
```

In addition, the parameter `validate` control input checking. For instance, turning `validate=False` allows to pass any type of target `y` and do some sampling for regression targets.

```
>>> from sklearn.datasets import make_regression
>>> X_reg, y_reg = make_regression(n_samples=100,
random_state=42)
>>> rng = np.random.RandomState(42)
>>> def dummy_sampler(X, y):
...     indices = rng.choice(np.arange(X.shape[0]), size=10)
...     return X[indices], y[indices]
>>> sampler = FunctionSampler(func=dummy_sampler,
validate=False)
>>> X_res, y_res = sampler.fit_resample(X_reg, y_reg)
>>> y_res
array([  41.49112498, -142.78526195,   85.55095317,
141.43321419,
         75.46571114,  -67.49177372,  159.72700509,
-169.80498923,
        211.95889757,  211.95889757])
```


Edit this page

We illustrate the use of such sampler to implement an outlier rejection estimator which can be easily used within a **`Pipeline`**: Customized sampler to implement an outlier rejections estimator

6.2. Custom generators

Imbalanced-learn provides specific generators for TensorFlow and Keras which will generate balanced mini-batches.

6.2.1. TensorFlow generator

The **`balanced_batch_generator`** allow to generate balanced mini-batches using an imbalanced-learn sampler which returns indices:

```
>>> X = X.astype(np.float32)
>>> from imblearn.under_sampling import RandomUnderSampler
>>> from imblearn.tensorflow import balanced_batch_generator
>>> training_generator, steps_per_epoch =
balanced_batch_generator(
...     X, y, sample_weight=None,
sampler=RandomUnderSampler(),
...     batch_size=10, random_state=42)
```

The `generator` and `steps_per_epoch` is used during the training of the Tensorflow model. We will illustrate how to use this generator. First, we can define a logistic regression model which will be optimized by a gradient descent:

```
>>> learning_rate, epochs = 0.01, 10
>>> input_size, output_size = X.shape[1], 3
>>> import tensorflow as tf
>>> def init_weights(shape):
...     return tf.Variable(tf.random_normal(shape,
stddev=0.01))
>>> def accuracy(y_true, y_pred):
...     return np.mean(np.argmax(y_pred, axis=1) == y_true)
>>> # input and output
>>> data = tf.placeholder("float32", shape=[None,
input_size])
>>> targets = tf.placeholder("int32", shape=[None])
>>> # build the model and weights
>>> W = init_weights([input_size, output_size])
>>> b = init_weights([output_size])
>>> out_act = tf.nn.sigmoid(tf.matmul(data, W) + b)
>>> # build the loss, predict, and train operator
>>> cross_entropy =
tf.nn.sparse_softmax_cross_entropy_with_logits(
...     logits=out_act, labels=targets)
>>> loss = tf.reduce_sum(cross_entropy)
>>> optimizer =
tf.train.GradientDescentOptimizer(learning_rate)
>>> train_op = optimizer.minimize(loss)
>>> predict = tf.nn.softmax(out_act)
>>> # Initialization of all variables in the graph
>>> init = tf.global_variables_initializer()
```

Once initialized, the model is trained by iterating on balanced mini-batches of data and minimizing the loss previously defined:

```
>>> with tf.Session() as sess:
...     print('Starting training')
...     sess.run(init)
...     for e in range(epochs):
...         for i in range(steps_per_epoch):
...             X_batch, y_batch = next(training_generator)
...             sess.run([train_op, loss], feed_dict={data:
X_batch, targets: y_batch})
...         # For each epoch, run accuracy on train and test
...         feed_dict = dict()
...         predicts_train = sess.run(predict, feed_dict=
{data: X})
...         print(f"epoch: {e} train accuracy: {accuracy(y,
predicts_train):.3f}")
...
Starting training
[...
```

6.2.2. Keras generator

Keras provides an higher level API in which a model can be defined and train by calling `fit_generator` method to train the model. To illustrate, we will define a logistic regression model:

```
>>> import keras
>>> y = keras.utils.to_categorical(y, 3)
>>> model = keras.Sequential()
>>> model.add(keras.layers.Dense(y.shape[1],
input_dim=X.shape[1],
...                               activation='softmax'))
>>> model.compile(optimizer='sgd',
loss='categorical_crossentropy',
...               metrics=['accuracy'])
```

`balanced_batch_generator` creates a balanced mini-batches generator with the associated number of mini-batches which will be generated:

```
>>> from imblearn.keras import balanced_batch_generator
>>> training_generator, steps_per_epoch =
balanced_batch_generator(
...     X, y, sampler=RandomUnderSampler(), batch_size=10,
random_state=42)
```

Then, `fit_generator` can be called passing the generator and the step:

```
>>> callback_history =
model.fit_generator(generator=training_generator,
...
steps_per_epoch=steps_per_epoch,
...                                         epochs=10,
verbose=0)
```

The second possibility is to use **`BalancedBatchGenerator`**. Only an instance of this class will be passed to `fit_generator`:

```
>>> from imblearn.keras import BalancedBatchGenerator
>>> training_generator = BalancedBatchGenerator(
...     X, y, sampler=RandomUnderSampler(), batch_size=10,
random_state=42)
>>> callback_history =
model.fit_generator(generator=training_generator,
...                                         epochs=10,
verbose=0)
```

References

- Porto Seguro: balancing samples in mini-batches with Keras

<< 5. Ensemble of samplers

7. Metrics >>