Version 0.8.0

8. Common pitfalls and recommended practices

This section is a complement to the documentation given [here] in scikit-learn. Indeed, we will highlight the issue of misusing resampling, leading to a **data leakage**. Due to this leakage, the performance of a model reported will be over-optimistic.

8.1. Data leakage

As mentioned in the scikit-learn documentation, data leakage occurs when information that would not be available at prediction time is used when building the model.

In the resampling setting, there is a common pitfall that corresponds to resample the **entire** dataset before splitting it into a train and a test partitions. Note that it would be equivalent to resample the train and test partitions as well.

Such of a processing leads to two issues:

- the model will not be tested on a dataset with class distribution similar to the real use-case. Indeed, by resampling the entire dataset, both the training and testing set will be potentially balanced while the model should be tested on the natural imbalanced dataset to evaluate the potential bias of the model;
- the resampling procedure might use information about samples in the dataset to either generate or select some of the samples. Therefore, we might use information of samples which will be later used as testing samples which is the typical data leakage issue.

We will demonstrate the wrong and right ways to do some sampling and emphasize the tools that one should use, avoiding to fall in the trap.

We will use the adult census dataset. For the sake of simplicity, we will only use the numerical features. Also, we will make the dataset more imbalanced to increase the effect of the wrongdoings:

Edit this page

```
>>> from sklearn.datasets import fetch_openml
>>> from imblearn.datasets import make_imbalance
>>> X, y = fetch_openml(
...     data_id=1119, as_frame=True, return_X_y=True
... )
>>> X = X.select_dtypes(include="number")
>>> X, y = make_imbalance(
...     X, y, sampling_strategy={">50K": 300},
random_state=1
... )
```

Let's first check the balancing ratio on this dataset:

```
>>> y.value_counts(normalize=True)
<=50K    0.98801
>50K     0.01199
Name: class, dtype: float64
```

To later highlight some of the issue, we will keep aside a left-out set that we will not use for the evaluation of the model:

```
>>> from sklearn.model_selection import train_test_split
>>> X, X_left_out, y, y_left_out = train_test_split(
...     X, y, stratify=y, random_state=0
... )
```

We will use a **`sklearn.ensemble.HistGradientBoostingClassifier`** as a baseline classifier. First, we will train and check the performance of this classifier, without any preprocessing to alleviate the bias toward the majority class. We evaluate the generalization performance of the classifier via cross-validation:

```
>>> from sklearn.experimental import
enable_hist_gradient_boosting
>>> from sklearn.ensemble import
HistGradientBoostingClassifier
>>> from sklearn.model_selection import cross_validate
>>> model = HistGradientBoostingClassifier(random_state=0)
>>> cv_results = cross_validate(
...     model, X, y, scoring="balanced_accuracy",
...     return_train_score=True, return_estimator=True,
...     n_jobs=-1
... )
>>> print(
...     f"Balanced accuracy mean +/- std. dev.: "
...     f"{cv_results['test_score'].mean():.3f} +/- "
...     f"{cv_results['test_score'].std():.3f}"
... )
Balanced accuracy mean +/- std. dev.: 0.609 +/- 0.024
```

We see that the classifier does not give good performance in terms of balanced accuracy mainly due to the class imbalance issue.

In the cross-validation, we stored the different classifiers of all folds. We will show that evaluating these classifiers on the left-out data will give close statistical performance:

```
>>> import numpy as np
>>> from sklearn.metrics import balanced_accuracy_score
>>> scores = []
>>> for fold_id, cv_model in
enumerate(cv_results["estimator"]):
...     scores.append(
...         balanced_accuracy_score(
...             y_left_out, cv_model.predict(X_left_out)
...         )
...     )
>>> print(
...     f"Balanced accuracy mean +/- std. dev.: "
...     f"{np.mean(scores):.3f} +/- {np.std(scores):.3f}"
... )
Balanced accuracy mean +/- std. dev.: 0.628 +/- 0.009
```

Let's now show the **wrong** pattern to apply when it comes to resampling to alleviate the class imbalance issue. We will use a sampler to balance the **entire** dataset and check the statistical performance of our classifier via cross-validation:

```
>>> from imblearn.under_sampling import RandomUnderSampler
>>> sampler = RandomUnderSampler(random_state=0)
>>> X_resampled, y_resampled = sampler.fit_resample(X, y)
>>> model = HistGradientBoostingClassifier(random_state=0)
>>> cv_results = cross_validate(
...     model, X_resampled, y_resampled,
scoring="balanced_accuracy",
...     return_train_score=True, return_estimator=True,
...     n_jobs=-1
... )
>>> print(
...     f"Balanced accuracy mean +/- std. dev.: "
...     f"{cv_results['test_score'].mean():.3f} +/- "
...     f"{cv_results['test_score'].std():.3f}"
... )
Balanced accuracy mean +/- std. dev.: 0.724 +/- 0.042
```

We see that the statistical performance are worse than in the previous case. Indeed, the data leakage gave us too optimistic results due to the reason stated earlier in this section.

We will now illustrate the correct pattern to use. Indeed, as in scikit-learn, using a **`Pipeline`** avoids to make any data leakage because the resampling will be delegated to imbalanced-learn and does not require any manual steps:

```
>>> from imblearn.pipeline import make_pipeline
>>> model = make_pipeline(
...     RandomUnderSampler(random_state=0),
...     HistGradientBoostingClassifier(random_state=0)
... )
>>> cv_results = cross_validate(
...     model, X, y, scoring="balanced_accuracy",
...     return_train_score=True, return_estimator=True,
...     n_jobs=-1
... )
>>> print(
...     f"Balanced accuracy mean +/- std. dev.: "
...     f"{cv_results['test_score'].mean():.3f} +/- "
...     f"{cv_results['test_score'].std():.3f}"
... )
Balanced accuracy mean +/- std. dev.: 0.732 +/- 0.019
```

We observe that we get good statistical performance as well. However, now we can check the performance of the model from each cross-validation fold to ensure that we have similar performance:

```
>>> scores = []
>>> for fold_id, cv_model in
enumerate(cv_results["estimator"]):
...     scores.append(
...         balanced_accuracy_score(
...             y_left_out, cv_model.predict(X_left_out)
...         )
...     )
>>> print(
...     f"Balanced accuracy mean +/- std. dev.: "
...     f"{np.mean(scores):.3f} +/- {np.std(scores):.3f}"
... )
Balanced accuracy mean +/- std. dev.: 0.727 +/- 0.008
```

We see that the statistical performance are very close to the cross-validation study that we perform, without any sign of over-optimistic results.

<< 7. Metrics

9. Dataset loading utilities >>