ensemble_hist_gradient_boosting

May 29, 2021

1 Speeding-up gradient-boosting

In this notebook, we present a modified version of gradient boosting which uses a reduced number of splits when building the different trees. This algorithm is called "histogram gradient boosting" in scikit-learn.

We previously mentioned that random-forest is an efficient algorithm since each tree of the ensemble can be fitted at the same time independently. Therefore, the algorithm scales efficiently with both the number of cores and the number of samples.

In gradient-boosting, the algorithm is a sequential algorithm. It requires the `N-1` trees to have been fit to be able to fit the tree at stage `N`. Therefore, the algorithm is quite computationally expensive. The most expensive part in this algorithm is the search for the best split in the tree which is a brute-force approach: all possible split are evaluated and the best one is picked. We explained this process in the notebook "tree in depth", which you can refer to.

To accelerate the gradient-boosting algorithm, one could reduce the number of splits to be evaluated. As a consequence, the statistical performance of such a tree would be reduced. However, since we are combining several trees in a gradient-boosting, we can add more estimators to overcome this issue.

We will make a naive implementation of such algorithm using building blocks from scikit-learn. First, we will load the California housing dataset.

[ ]:
```
from sklearn.datasets import fetch_california_housing

data, target = fetch_california_housing(return_X_y=True, as_frame=True)
target *= 100  # rescale the target in k$
```

Note

If you want a deeper overview regarding this dataset, you can refer to the Appendix - Datasets description section at the end of this MOOC.

We will make a quick benchmark of the original gradient boosting.

[ ]:
```
from sklearn.model_selection import cross_validate
from sklearn.ensemble import GradientBoostingRegressor

gradient_boosting = GradientBoostingRegressor(n_estimators=200)
cv_results_gbdt = cross_validate(
```

```
    gradient_boosting, data, target, scoring="neg_mean_absolute_error",
    n_jobs=-1
)
```

[ ]:
```
print("Gradient Boosting Decision Tree")
print(f"Mean absolute error via cross-validation: "
      f"{-cv_results_gbdt['test_score'].mean():.3f} +/- "
      f"{cv_results_gbdt['test_score'].std():.3f} k$")
print(f"Average fit time: "
      f"{cv_results_gbdt['fit_time'].mean():.3f} seconds")
print(f"Average score time: "
      f"{cv_results_gbdt['score_time'].mean():.3f} seconds")
```

We recall that a way of accelerating the gradient boosting is to reduce the number of split considered within the tree building. One way is to bin the data before to give them into the gradient boosting. A transformer called `KBinsDiscretizer` is doing such transformation. Thus, we can pipeline this preprocessing with the gradient boosting.

We can first demonstrate the transformation done by the `KBinsDiscretizer`.

[ ]:
```
import numpy as np
from sklearn.preprocessing import KBinsDiscretizer

discretizer = KBinsDiscretizer(
    n_bins=256, encode="ordinal", strategy="quantile")
data_trans = discretizer.fit_transform(data)
data_trans
```

Note

The code cell above will generate a couple of warnings. Indeed, for some of the features, we requested too much bins in regard of the data dispersion for those features. The smallest bins will be removed.

We see that the discretizer transforms the original data into an integer. This integer represents the bin index when the distribution by quantile is performed. We can check the number of bins per feature.

[ ]:
```
[len(np.unique(col)) for col in data_trans.T]
```

After this transformation, we see that we have at most 256 unique values per features. Now, we will use this transformer to discretize data before training the gradient boosting regressor.

[ ]:
```
from sklearn.pipeline import make_pipeline

gradient_boosting = make_pipeline(
    discretizer, GradientBoostingRegressor(n_estimators=200))
cv_results_gbdt = cross_validate(
    gradient_boosting, data, target, scoring="neg_mean_absolute_error",
```

```
    n_jobs=-1,
)
```

[ ]:
```
print("Gradient Boosting Decision Tree with KBinsDiscretizer")
print(f"Mean absolute error via cross-validation: "
      f"{-cv_results_gbdt['test_score'].mean():.3f} +/- "
      f"{cv_results_gbdt['test_score'].std():.3f} k$")
print(f"Average fit time: "
      f"{cv_results_gbdt['fit_time'].mean():.3f} seconds")
print(f"Average score time: "
      f"{cv_results_gbdt['score_time'].mean():.3f} seconds")
```

Here, we see that the fit time has been drastically reduced but that the statistical performance of the model is identical. Scikit-learn provides a specific classes which are even more optimized for large dataset, called `HistGradientBoostingClassifier` and `HistGradientBoostingRegressor`. Each feature in the dataset `data` is first binned by computing histograms, which are later used to evaluate the potential splits. The number of splits to evaluate is then much smaller. This algorithm becomes much more efficient than gradient boosting when the dataset has over 10,000 samples.

Below we will give an example for a large dataset and we will compare computation times with the experiment of the previous section.

[ ]:
```
from sklearn.experimental import enable_hist_gradient_boosting
from sklearn.ensemble import HistGradientBoostingRegressor

histogram_gradient_boosting = HistGradientBoostingRegressor(
    max_iter=200, random_state=0)
cv_results_hgbdt = cross_validate(
    gradient_boosting, data, target, scoring="neg_mean_absolute_error",
    n_jobs=-1,
)
```

[ ]:
```
print("Histogram Gradient Boosting Decision Tree")
print(f"Mean absolute error via cross-validation: "
      f"{-cv_results_hgbdt['test_score'].mean():.3f} +/- "
      f"{cv_results_hgbdt['test_score'].std():.3f} k$")
print(f"Average fit time: "
      f"{cv_results_hgbdt['fit_time'].mean():.3f} seconds")
print(f"Average score time: "
      f"{cv_results_hgbdt['score_time'].mean():.3f} seconds")
```

The histogram gradient-boosting is the best algorithm in terms of score. It will also scale when the number of samples increases, while the normal gradient-boosting will not.