09.5 - Supervised Learning - Classification - Bagging and Random Forests Scratch

October 12, 2020

1 Programming for Data Science and Artificial Intelligence

1.1 9.5 Supervised Learning - Classification - Bagging and Random Forests Scratch

1.1.1 Readings:

- [GERON] Ch7
- [VANDER] Ch5
- [HASTIE] Ch15
- https://scikit-learn.org/stable/modules/ensemble.html

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

1.2 Bagging

A single decision tree does not perform well as it tends to overfit. A possible solution is the construct multiple trees to reduce variances. To make sure each tree is not exactly learning the same thing since it will then be all same trees, we need to inject some differences to these trees (i.e., make them as diverse as possible but at the same time they also see some overlappinp samples). One simple idea is that each of the tree is trained on a subset of **bootstrapping sample** and then perform some sort of aggregation of the decision.

The process has the following steps:

1. Sample m times **with replacement** from the original training data
2. Repeat B times to generate B "boostrapped" training datasets $D_1, D_2, \cdots, D_B$
3. Train B trees using the training datasets $D_1, D_2, \cdots, D_B$

Boostrapping the data plus performing some sort of aggregation (averaging or majority votes) is called **boostrap aggregation** or **bagging**.

Example:

Assume that we have a training set where $m = 4$, and $n = 2$:

$$D = (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$$

We generate, say, $B = 3$ datasets by boostrapping:

$$D_1 = (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_3, y_3)$$
$$D_2 = (x_1, y_1), (x_4, y_4), (x_4, y_4), (x_3, y_3)$$
$$D_3 = (x_1, y_1), (x_1, y_1), (x_2, y_2), (x_2, y_2)$$

We can then train 3 trees.

Note: When sampling is performed **without** replacement, it is called **pasting**. In other words, both bagging and pasting allow training instacnes to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.

Let's try to code from scratch. To make our life easier, we shall use DecisionTree from the sklearn library (since we already code it from scratch in the previous class)

1.2.1 Scratch

[2]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y,
                test_size=0.3, shuffle=True, random_state=42)
```

[3]:
```
from sklearn.tree import DecisionTreeClassifier
import random
from scipy import stats
from sklearn.metrics import classification_report

B = 5
m, n = X_train.shape
boostrap_ratio = 1
tree_params = {'max_depth': 2, 'criterion':'gini', 'min_samples_split': 5}
models = [DecisionTreeClassifier(**tree_params) for _ in range(B)]

#sample size for each tree
sample_size = int(boostrap_ratio * len(X_train))

xsamples = np.zeros((B, sample_size, n))
ysamples = np.zeros((B, sample_size))

#subsamples for each model
for i in range(B):
    ##sampling with replacement; i.e., sample can occur more than once
```

```
    #for the same predictor
    for j in range(sample_size):
        idx = random.randrange(m)    #<----with replacement #change so no␣
↪repetition
        xsamples[i, j, :] = X_train[idx]
        ysamples[i, j] = y_train[idx]
        #keep track of idx that i did not use for ith tree

#fitting each estimator
for i, model in enumerate(models):
    _X = xsamples[i, :]
    _y = ysamples[i, :]
    model.fit(_X, _y)

#make prediction and return the probabilities
predictions = np.zeros((B, X_test.shape[0]))
for i, model in enumerate(models):
    yhat = model.predict(X_test)
    predictions[i, :] = yhat

yhat = stats.mode(predictions)[0][0]

print(classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

1.2.2 Sklearn

[4]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

tree = DecisionTreeClassifier()

'''
To perform in sklearn, we can use the BaggingClassifier API.
Pasting can be done using BaggingClassifier< setting boostrap=False
'''
```

```
bag = BaggingClassifier(tree, n_estimators=5, max_samples=0.99)

bag.fit(X_train, y_train)
yhat = bag.predict(X_test)
print(classification_report(y_test, yhat))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

1.2.3 ===Classwork===

Out of Bag Evaluation Well, it seems like our bagging technique is quite good. Anyhow, one interesting observation is that each tree only see a subset of the dataset. Any data that a particular tree did not see is called **out of bag** (oob). Note that oob is not the same for all predictors.

One interesting thing is that since oob is something that each tree never see, thus oob is somewhat a validation set. Thus what we can do is after we fit each tree. We can ask each tree to test their accuracy with their own oob, and then we can average the accuracy from all trees.

Your work: Let's modify the above scratch code to

Calculate for oob evaluation for each bootstrapped dataset, and also the average score

Change the code to "without replacement"

Put everything into a class Bagging. It should have at least two methods, fit(X_train, y_train), and predict(X_test)

No score, no pressure, only intrinic motivation

1.3 Random Forests

So far, it seems bagging works well. However, the B bootstrapped dataset are correlated, thus the power of variance reduction is diminished. How do we further de-correlate these B trees?

A **random forest** is constructed by bagging, but for each split in each tree, only a random subset of $q \leq n$ features are considered as splitting variables.

Rule of thumb: $q = \sqrt{n}$ for classification trees and $q = \frac{n}{3}$ for regression trees

1.3.1 Scratch

[5]:
```
# Your code here
```

1.3.2 ===Task===

Your work: Let's modify the above scratch code to

To introduce random features during each split. You may want to use scratch version of Decision-Tree you made in previous class so you can implement how the split is done

Change the class into RandomForest. It should have at least two methods, fit(X_train, y_train), and predict(X_test)

1.3.3 Sklearn

[6]:
```
#this is the same as RandomForest
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

param_grid = {"n_estimators": [10, 50, 100],
              "criterion": ["gini", "entropy"],
              "max_depth": np.arange(1, 10)}
model = RandomForestClassifier()

grid = GridSearchCV(model, param_grid)
grid.fit(X, y)

print(grid.best_params_)

model = grid.best_estimator_
model.fit(X_train, y_train)

yhat = model.predict(X_test)

print(classification_report(y_test, yhat))
```

```
{'criterion': 'gini', 'max_depth': 4, 'n_estimators': 100}
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

1.4 When to use Random Forests

Advantages of Random Forest:

- Voting helps overcome overfitting
- Because bagging and pasting support parallel computing (e.g., using n_jobs), they are very popular methods.
- Random forest can solve both type of problems that is classification and regression and does a decent estimation at both fronts.
- The power to handle large data sets with higher dimensionality. It can handle thousands of input variables and identity most significant variables so it is considered as one of the dimensionality reduction method. Further, the model outputs importance of variable, which can be a very handy feature. Sklearn implements feature_importances_ in RandomForestClassifier which helps you understand which feature is useful for classification in Random Forest
- It has an effective method for estimating missing data and maintains accuracy when large proportion of the data are missing (I did not really touch this, but I recommend you to check it out)
- It has methods for balancing errors in data sets where classes are imbalanced.
- The capability of the above can be extended to unlabeled data, leading to unsupervised clustering,data views and outlier detection.
- Just like other ensemble, it works well with structured/tabular data. Indeed, XGBoost (another ensemble method) is among the best classifier for structured/tabular data and often used for Kaggle competition. But if we are working with image, sound, brain signal, deep learning remains the way to go.
- Unlike Decision Trees, multiple trees can give out probability
- Out of bag evaluation is handy

Disadvantages of Random Forest:

- It surely does a good job at classification but not as for regression problem as it does not gives precise continuous nature prediction. In case of regression, it doesn't predict beyond the range in the training data, and that they may over fit data sets that are particularly noisy.
- Random forest can feel like a black box approach for a statistical modelers we have very little control on what the model does. You can at best try different parameters and random seeds.
- At one point, more samples will not improve the accuracy, unlike deep neural network
- It fails when there are rare outcomes or rare predictors, as the algorithm is based on bootstrap sampling. This makes it non-ideal if you're working with rare personality traits, high segmented customer behavior, or rare variants in genomics research.

In conclusion, if you are working with structured/tabular data, and would like high accuracy but does not care much about interpretability (just like most Kaggle competition does), you may want to use ensemble methods (including Random Forests and the like)

[ ]: