01 - Supervised Learning - Classification - Bagging and Random Forests(Solution)_st122097_Thantham

September 15, 2021

1 Programming for Data Science and Artificial Intelligence

1.1 Supervised Learning - Classification - Bagging and Random Forests

1.2 Name: Thantham Khamyai

1.3 Student ID: 122097

1.3.1 Tasks

Modify the Bagging scratch code in our lecture such that: - 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) - Modify the code from above to randomize features. Set the number of **features** to be used in each tree to be **sqrt(n)**, and then select a subset of features for each tree. This can be easily done by setting our **DecisionTreeClassifier max_features to 'sqrt'**

[1]:
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier
import numpy as np
import matplotlib.pyplot as plt
```

Load Iris Dataset into training and testing data

[2]:
```
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=48)
```

This section is to construct the Random Forest model. lets see the rounghly algorithm

1.4 Random Forest Algorithm

1.4.1 fit:

1. define bootstap ratio, and n_tree
2. Init booststrap samples and oob samples
3. fill boostrap samples and oob samples
4. use boostrap samples to make tree and check acc for oob

1.4.2 predict:

1. fetch all test x to each tree
2. get all predict and find majority

1.4.3 Task - Bagging class

1.4.4 Task - Max features

[10]:
```
from sklearn.tree import DecisionTreeClassifier
from scipy import stats
import random

#class RandomForest:
class Bagging:

    def __init__(self, n_estimators=10, bootstrap_ratio=0.8,␣
↪no_replacement=True, # constructor===============
                 max_features='sqrt', max_depth=None):

        self.B = n_estimators # same as number of decision tree in side models
        self.bootstrap_ratio = bootstrap_ratio # fraction of sampling
        self.no_replacement = no_replacement # not allow duplicate?
        self.max_depth = max_depth # max depth of tree
        self.max_features = max_features # mas feature allow in tree
        self.tree_params = {'max_depth': self.max_depth, 'max_features': self.
↪max_features} # make params for tree
        self.trees = [DecisionTreeClassifier(**self.tree_params) for _ in␣
↪range(self.B)] # init d.trees by B


    def fit(self, X, y): # fit method␣
↪======================================================================

        m,n = X.shape # init m,n
        self.num_class = len(np.unique(y)) # init number of unique output for␣
↪make proba prediction

        # 1. define number of samples in boostrap out of all␣
↪samples----------------------------------
```

```
        n_sample_bootstrap = int(self.bootstrap_ratio * m)


        # 2. init boostrap samples and oob␣
↪smaples-----------------------------------------------------
        x_bootstrap = np.zeros((self.B, n_sample_bootstrap, n)) # shape as␣
↪(n_trees, n_samples, n_features)
        y_bootstrap = np.zeros((self.B, n_sample_bootstrap)) # shape as␣
↪(n_trees, n_samples)

        x_oob = [] # unknow shape
        y_oob = [] # unknow shape


        # 3. fill them bootstrap and␣
↪oob----------------------------------------------------------------

        for tree_i in range(len(self.trees)): # for each tree

            sample_i = 0 # sample in bootstrap counter

            used_idx = [] # init replacement used idx

            bootstrap_idx = [] # init idx which are for bootstrap

            while sample_i < n_sample_bootstrap: # while sample in bootstrap is␣
↪not more than defined n samples

                idx = random.randrange(0, m) # random bootstrap idx

                if self.no_replacement: # if not allow duplicate samples in␣
↪bootstrap

                    while idx in used_idx: # if randomed idx is duplicate with␣
↪used

                        idx = random.randrange(0, m) # newly random until found␣
↪not duplicate

                    used_idx.append(idx) # after found not duplicate idx save␣
↪that idx for fuether random

                x_bootstrap[tree_i, sample_i, :] = X[idx, :] # append bootstrap␣
↪X
                y_bootstrap[tree_i, sample_i] = y[idx] # append bootstrap y
```

```
                bootstrap_idx.append(idx) # save this bootstrap idx

                sample_i += 1 # then -> find next sample to be bootstrao

             #after fullfill bootstrap, fetch idx which is not in bootstrap
            oob_idx = list(set(np.arange(m)) - set(bootstrap_idx))

            # define oob samples
            x_oob.append(X[oob_idx])
            y_oob.append(y[oob_idx])

        # make it numpy ass because we will need it for indexing while␣
↪validation
        x_oob = np.asarray(x_oob, dtype='object')
        y_oob = np.asarray(y_oob, dtype='object')


        # 4. use boostrap samples making␣
↪trees-------------------------------------------------------------

        self.oob_score_ = np.zeros((self.B)) # init list of oob score in each␣
↪trees (n_trees,)

        for tree_i in range(self.B): # for each tree in model

            self.trees[tree_i].fit(x_bootstrap[tree_i], y_bootstrap[tree_i]) #␣
↪fit x, y at bootstrap i

            y_pred_oob = self.trees[tree_i].predict(x_oob[tree_i]) # predict␣
↪y_oob_pred at oob i

            self.oob_score_[tree_i] = sum(y_oob[tree_i]==y_pred_oob)/y_pred_oob.
↪shape[0] # acc for current tree

        self.oob_avg_score_ = np.mean(self.oob_score_) # find everage oob score


    def predict(self, X): # predict method␣
↪=================================================================

        y_pred = np.zeros((self.B, X.shape[0])) # init prediction by (n_trees,␣
↪n_output)

        for tree_i in range(self.B): # for each tree in model
            y_pred[tree_i] = self.trees[tree_i].predict(X) # predict and keep␣
↪in prediction of that tree
```

```
        return stats.mode(y_pred)[0][0] # return majority of predictions


    def predict_proba(self, X): # predict by probability␣
↪=====================================================

        y_pred = np.zeros((self.B, X.shape[0])) # init prediction by (n_trees,␣
↪n_output)

        for tree_i in range(self.B): # for each tree in model
            y_pred[tree_i] = self.trees[tree_i].predict(X) # predict and keep␣
↪in prediction of that tree

        y_pred = y_pred.T.astype('int') # tranpose prediction (n_output,␣
↪n_tree) -> (output i, result of tree i)

        y_prob = np.apply_along_axis(lambda x: np.bincount(x, minlength=self.
↪num_class),
                                     axis=1, arr=y_pred) # map bincount for␣
↪each test sample

        return y_prob/self.B # return counted bin each sample with n_trees
```

After constructing Bagging class (Random Forest), lets try model

1.4.5 Task - selective no_replacement option

[11]:
```
model = Bagging(n_estimators=10, bootstrap_ratio = 0.8, no_replacement=False,␣
↪max_features=None, max_depth=None)
model.fit(X_train, y_train)
```

While fitting model, the oob scores were calculated. we can see by this

1.4.6 Task - OOB score

[12]:
```
print(f'=========== {model.B} trees ================')
for i in range(model.B):
    print(f' Tree {i+1} has oob score: {model.oob_score_[i]}')
```

```
=========== 10 trees ================
 Tree 1 has oob score: 0.9215686274509803
 Tree 2 has oob score: 1.0
 Tree 3 has oob score: 0.9811320754716981
 Tree 4 has oob score: 0.9545454545454546
 Tree 5 has oob score: 0.9347826086956522
```

```
Tree 6 has oob score: 0.9148936170212766
Tree 7 has oob score: 1.0
Tree 8 has oob score: 0.9565217391304348
Tree 9 has oob score: 0.9302325581395349
Tree 10 has oob score: 0.9777777777777777
```

Moreover, we can print out averaged oob of the model

[13]:
```python
print(f'Overall average oob score: {model.oob_avg_score_}')
```

```
Overall average oob score: 0.9571454458232809
```

Next, we can predict the output from testing set, and the output of predict method will be

[14]:
```python
y_pred = model.predict(X_test)
print(y_pred)
```

```
[1. 1. 2. 0. 2. 2. 0. 2. 0. 1. 2. 0. 0. 2. 1. 1. 0. 1. 1. 2. 0. 2. 2. 1.
 1. 0. 0. 2. 2. 1. 2. 1. 2. 0. 1. 2. 2. 1. 0. 1. 1. 1. 2. 2. 1.]
```

This is additional method, we can fuether predict by probability

[15]:
```python
y_prob = model.predict_proba(X_test)
print(y_prob[:10])
```

```
[[0.  1.  0. ]
 [0.  1.  0. ]
 [0.  0.3 0.7]
 [1.  0.  0. ]
 [0.  0.4 0.6]
 [0.  0.  1. ]
 [1.  0.  0. ]
 [0.  0.4 0.6]
 [1.  0.  0. ]
 [0.  1.  0. ]]
```

Therefore, we try to examine classification report here

[16]:
```python
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       0.94      0.89      0.91        18
           2       0.88      0.94      0.91        16

    accuracy                           0.93        45
   macro avg       0.94      0.94      0.94        45
weighted avg       0.93      0.93      0.93        45
```

1.5 Completed Tasks

- Perform **OOB** score calculation
- add option of **No replacement** bootstrap
- make **Bagging class**
- add option**max_feature** and set 'sqrt' by default

```
[ ]:
```