MSDS 621 - Introduction to Machine Learning

Homework 4

Robert Clements

1 Implementing Random Forests

1.1 Turning it in

We will use Github classroom to submit assignments. For this assignment you should submit two files:

- rf.py file
- dtree.py file

1.2 Goal

In this homework you will be implementing a basic version of random forests, using code from the decision tree implementation from the previous homework. Note that we are not all that concerned with efficiency for this implementation, but we do want accuracy comparable to sklearn. You will be implementing both regression and classification random forests, as well as the ability to estimate the out-of-bag (OOB) error.

1.3 Getting Started

As always, there is template code in the repository. To make your life easier, I would use the template code as a guide, but if you are particularly ambitious, you can ignore it and create your own implementation, so long as the tests still pass. You should also use some of the code you created in `dtree.py`, which you will have to modify in order to train the decision trees correctly.

1.4 Discussion

We know all about random forests from our lectures, so you should already understand that they solve some of the problems that you face with decision trees in terms of their high variance by fitting an ensemble of decorrelated trees. We reduce correlation of our trees in two primary ways: by training each tree on a different bootstrapped sample of the training data; and by sampling a subset of our features at every potential decision node. This results in a set of individual decision trees that, alone, do not predict very well, but when averaged out form a powerful ensemble that can have high predictive performance.

1.4.1 Bootstrapping

Recall that the reason we use bootstrapping (sample w/replacement) is to create new training sets that come from the same distribution, so they are *i.d.* but not *i.i.d.*. In practice, about 2/3 of the data will be sampled with each bootstrap sample, which leaves 1/3 of the data to be used as part of our out-of-bag (OOB) data. See sklearn's resample function for an easy way to get a list of indexes to help create a bootstrap sample training set.

1.4.2 Changes to decision tree training

There are two important changes we need to make to the decision tree code so that it is suitable for random forests. First, we need to update the code for our decision nodes so that decision trees know how to return the leaf of the tree that should make the prediction, rather than the prediction itself as `predict()` does. Remember that with random forests we can use a weighted average of the individual tree predictions to make a final random forest prediction. To produce this weighted average, we need to know not just the prediction in the leaf node, but also the number of observations within that leaf. For classification we also might need to retain the actual y values, or the counts of each class.

```
class DecisionNode:
    def __init__(self, col, split, lchild, rchild):
        self.col = col
        self.split = split
        self.lchild = lchild
        self.rchild = rchild
    def predict(self, x_test):
        ...
    def leaf(self, x_test):
        """
        Given a single test record, x_test, return the leaf node reached by running
        it down the tree starting at this node.  This is just like prediction,
```

```
except we return the decision tree leaf rather than the prediction from that leaf.
"""
...
```

A `LeafNode` obviously just returns itself (`self`) rather than the prediction.

The second change is in the node splitting functionality. For fitting decision trees, we normally check all available features and all available feature values for the optimal variable/split combination. For random forests, each split should pick from a random subset of the features; the subset size is the hyperparameter `max_features`. Function `np.random.choice()` is useful here to get a list of feature indexes and then `X[:, i]` gives us the ith column.

Note that you can also retain the sampling of the feature values (size = 11) that you implemented in the previous homework to keep the model from taking too long to fit. Meaning, for each feature that you sample, you only need to check 11 of the values for splitting.

1.4.3 Random Forest Prediction

Once we've trained a forest of decision trees, we can make predictions for one or more feature vectors using `predict()`. For regression, the prediction for the forest is the weighted average of the predictions from the individual decision trees. If `X_test` passed to `predict()` is a two-dimensional matrix of n rows, then n predictions should be returned in an array from `predict()`. To make a prediction for an observation, call `leaf()` on each tree to get the leaf node that contains the prediction information for the feature vector. Each leaf has `n`, the number of observations in that leaf that can serve as our weight. The leaf also has a `prediction` that is the predicted y value for regression or class for classification. For regression, compute the total weight of all `leaf.n` and then compute the sum of `leaf.prediction * leaf.n`. The prediction is then the weighted sum divided by the total weight. For example, if we have two trees, and the leaf node in tree one contained `[2, 5, 9]` and the leaf node in tree two contained `[3, 5, 6, 9]`, then the final prediction would be

$$\frac{1}{7}(2+5+9+3+5+6+9)$$

which is equivalent to

$$\frac{1}{7}(3 \times \text{tree one prediction} + 4 \times \text{tree two prediction})$$

For classification, it's a little more complicated because we need a weighted majority vote across all trees. As with regression, go through all of the trees, and get the leaves associated with the prediction of a single observation. Create a numpy array, e.g. `class_counts` to hold the counts for each class. Then, increment `class_counts[y]` for each `y` in each leaf associated

with the `x` test vector. This means that if we had two trees, and the leaf node in tree one contained `[0,0,0,1]` and tree two contained `[0,0,1]`, then our `class_counts` array should be `[5,2]` because there is a count of five 0s and two 1s. Then our final prediction can be the argmax of this array. This assumes, of course, that index 0 corresponds to class 0, and index 1 corresponds to class 1, etc..

Technically, in `sklearn's` implementation of `RandomForestRegressor`, the `predict` method computes the *unweighted* average of the individual tree predictions, meaning the above example would be

$$\frac{1}{2}\left(\frac{2+5+9}{3}+\frac{3+5+6+9}{4}\right).$$

But you should implement the weighted average. The unit tests should still pass even though the prediction methods are different.

For `RandomForestClassifier` in `sklearn` it computes what it calls a weighted majority vote, which is the average of the individual tree predicted probabilities (proportions), meaning for the above example it would be

$$\frac{1}{2}\left(\frac{1}{4}+\frac{1}{3}\right)$$

for class 1. Rather than have to deal with individual tree probabilities you can use the approach mentioned above which only deals with counts. This should work fine for the unit tests.

1.5 Out-of-bag (OOB) error

Recall from lecture that we can estimate the validation error using the out-of-bag error. We can calculate the R^2 and accuracy scores using the OOB observations. For each decision tree that is fit you should keep track of the OOB observation indexes for each tree. Remember that these are the observations that are not included in your bootstrap sample (just store these indexes as an attribute in each tree of the ensemble). After training all of the trees in `fit()`, you can loop through the trees and compute the OOB score (if hyperparameter `self.oob_score` is set to `True`). You do this by getting the predictions for each out-of-bag observation, i.e. for each OOB observation calculate the predicted value by using only the trees that do not contain that observation. When you have predictions for each OOB observation, you can then calculate the R^2 or the accuracy score. Save this score in `self.oob_score_`.

1.5.1 Regression and classification class definitions

To mimic sklearn machine learning models, we need to create some class definitions. You are free to implement the regression and classification tree and forest objects as you like, but the unit tests should be able to run and pass as-is.

The `RandomForest621` class has a generic `fit()` method that is inherited by subclasses `RandomForestRegressor621` and `RandomForestClassifier621`. Field `n_estimators` is the number of trees in the forest. I also store the number of unique `y` values in `nunique` because, for classification, we need to know how many classes there are.

Method `compute_oob_score()` is just a helper method that I used to encapsulate that functionality, but you can do whatever you want. `RandomForest621.fit()` calls `self.compute_oob_score()` and that calls the implementation either in regressor or classifier, depending on which object I created.

You can use the following class definitions as templates:

```
class RandomForest621:
    def __init__(self, n_estimators=10, oob_score=False):
        self.n_estimators = n_estimators
        self.oob_score = oob_score
        self.oob_score_ = np.nan

    def fit(self, X, y):
        """
        Given an (X, y) training set, fit all n_estimators trees to different,
        bootstrapped versions of the training data.  Keep track of the indexes of
        the OOB records for each tree.  After fitting all of the trees in the forest,
        compute the OOB validation score estimate and store as self.oob_score_, to
        mimic sklearn.
        """
        ...
        if self.oob_score:
            self.oob_score_ = ... compute OOB score ...

class RandomForestRegressor621(RandomForest621):
    def __init__(self, n_estimators=10, min_samples_leaf=3,
    max_features=0.3, oob_score=False):
        super().__init__(n_estimators, oob_score=oob_score)
        self.trees = ...

    def predict(self, X_test) -> np.ndarray:
```

```
        """
        Given a 2D nxp array with one or more records, compute the weighted average
        prediction from all trees in this forest. Weight each trees prediction by
        the number of observations in the leaf making that prediction.  Return a 1D vector
        with the predictions for each input record of X_test.
        """
        ...

    def score(self, X_test, y_test) -> float:
        """
        Given a 2D nxp X_test array and 1D nx1 y_test array with one or more records,
        collect the prediction for each record and then compute R^2 on that and y_test.
        """
        ...


class RandomForestClassifier621(RandomForest621):
    def __init__(self, n_estimators=10, min_samples_leaf=3,
    max_features=0.3, oob_score=False):
        super().__init__(n_estimators, oob_score=oob_score)
        n_estimators = n_estimators
        self.min_samples_leaf = min_samples_leaf
        self.trees = ...

    def predict(self, X_test) -> np.ndarray:
        ...

    def score(self, X_test, y_test) -> float:
        """
        Given a 2D nxp X_test array and 1D nx1 y_test array with one or more records,
        collect the predicted class for each record and then compute accuracy between
        that and y_test.
        """
        ...
```

1.5.2 Tying it all together

Ok, first you should modify your decision tree code so that you: (a) randomly sample a set of features for each node split (using a `max_features` argument); (b) create a new method that will return the actual leaf node for a given observation, rather than just the predicted value in the leaf node.

Once your new decision tree classes are working, you can create the random forest classes that you will need. I recommend having a generic `RandomForest621` class, with a `fit` method that is inherited by the two required classes `RandomForestRegressor621` and `RandomForestClassifier621` so you only have to write the `fit` method once. Within `fit` you should keep track of all out-of-bag indexes for each tree, and calculate the out-of-bag error and save it. Within the two required classes you should have `predict` methods which return the weighted predictions deescribed above, `score` methods which will calculate R^2 or accuracy for a given set of data, and optionally you can have a method that computes the out-of-bag error (which might be called from `RandomForest621`).

1.6 Deliverables

To submit your homework you must provide the following files at the root of the repository directory:

- `dtree.py` This is the code from your previous project but with the updates specified above to randomly select from a subset of the features during each split.
- `rf.py` This is the file containing your `RandomForestRegressor621` and `RandomForestClassifier621` implementations, and any other functions or classes you need.

You **must** implement the following classes:

- `RandomForestRegressor621`: should contain `score` and `predict` methods, and should contain `oob_score_` attribute if you choose to implement out-of-bag error estimation
- `RandomForestClassifier621`: should contain `score` and `predict` methods, and should contain `oob_score_` attribute if you choose to implement out-of-bag error estimation

1.7 Evaluation

To evaluate your projects I will download your repository and run `test_rf.py` from your root directory. As always, I encourage you to test your code both locally and using Github Actions by putting the `test.yml` file in a directory called `.github/workflows`. I will be using the unit tests **as a guide**, meaning that if you pass all unit tests, and the timing test, you will get 100%. If you do not pass all unit tests, then your code will be inspected and evaluated for errors. Note that if you *almost always* pass the unit tests (say at least 90% of the time) then your code is probably ok and you've run into a bit of bad luck on the training and testing splits. If you are failing multiple tests in a single run, there is likely something wrong with your code.

Note that there are 8 OOB tests, but the OOB functionality of your implementation will be weighted less than the fitting and predicting functionality. If you cannot get the OOB methods to work, or if you choose not to do them, you will lose a total of 8 points, meaning you can get a maximum of 92% if you decide to skip this part.

Because these tests take so long and they are completely independent, we can test a number of them in parallel to speed things up.

```
$ pip install pytest-xdist
```

For me, it's 4x faster when I use `-n 8` option (on my Apple M1 Max CPU): `pytest -v -n 8 test_rf.py`. Your code should run on your local machine in less than two minutes using this option (or there will be a 10 point penalty).

```
============================= test session starts ==============================
test_rf.py::test_boston_min_samples_leaf
test_rf.py::test_boston
test_rf.py::test_iris
test_rf.py::test_boston_most_features
test_rf.py::test_diabetes_oob
test_rf.py::test_diabetes_all_features
test_rf.py::test_california_housing_oob
test_rf.py::test_diabetes
[gw7] [  3%] PASSED test_rf.py::test_iris
test_rf.py::test_iris_all_features
[gw7] [  7%] PASSED test_rf.py::test_iris_all_features
test_rf.py::test_iris_most_features
[gw6] [ 11%] PASSED test_rf.py::test_california_housing_oob
test_rf.py::test_iris_ntrees
[gw3] [ 15%] PASSED test_rf.py::test_diabetes
test_rf.py::test_diabetes_ntrees
[gw5] [ 19%] PASSED test_rf.py::test_diabetes_oob
test_rf.py::test_california_housing
[gw7] [ 23%] PASSED test_rf.py::test_iris_most_features
test_rf.py::test_iris_oob
[gw6] [ 26%] PASSED test_rf.py::test_iris_ntrees
test_rf.py::test_wine
[gw0] [ 30%] PASSED test_rf.py::test_boston
test_rf.py::test_boston_oob
[gw7] [ 34%] PASSED test_rf.py::test_iris_oob
test_rf.py::test_wine_oob
[gw5] [ 38%] PASSED test_rf.py::test_california_housing
test_rf.py::test_wine_most_features
[gw6] [ 42%] PASSED test_rf.py::test_wine
test_rf.py::test_wine_min_samples_leaf
[gw1] [ 46%] PASSED test_rf.py::test_boston_min_samples_leaf
test_rf.py::test_boston_all_features
```

```
[gw7] [ 50%] PASSED test_rf.py::test_wine_oob
test_rf.py::test_breast_cancer
[gw3] [ 53%] PASSED test_rf.py::test_diabetes_ntrees
test_rf.py::test_wine_all_features
[gw0] [ 57%] PASSED test_rf.py::test_boston_oob
test_rf.py::test_wine_min_samples_leaf_oob
[gw6] [ 61%] PASSED test_rf.py::test_wine_min_samples_leaf
[gw5] [ 65%] PASSED test_rf.py::test_wine_most_features
test_rf.py::test_breast_cancer_oob
[gw4] [ 69%] PASSED test_rf.py::test_diabetes_all_features
test_rf.py::test_diabetes_most_features
[gw0] [ 73%] PASSED test_rf.py::test_wine_min_samples_leaf_oob
[gw3] [ 76%] PASSED test_rf.py::test_wine_all_features
[gw7] [ 80%] PASSED test_rf.py::test_breast_cancer
[gw4] [ 84%] PASSED test_rf.py::test_diabetes_most_features
[gw5] [ 88%] PASSED test_rf.py::test_breast_cancer_oob
[gw2] [ 92%] PASSED test_rf.py::test_boston_most_features
test_rf.py::test_boston_min_samples_leaf_oob
[gw2] [ 96%] PASSED test_rf.py::test_boston_min_samples_leaf_oob
[gw1] [100%] PASSED test_rf.py::test_boston_all_features
============================== warnings summary ==============================
...
============================= 26 passed, 9 warnings in 25.54s ==================
```