Project

Collaborators:
Name: Jiang Shibiao
Student ID: 3170102587

Problem. Basic: A Walk Through Ensemble Models (35%)

In this problem, you will implement a whole bunch of ensemble models and compare their performance and properties.

(a) Implement decision tree algorithm (in decision tree.py), then answer the following questions.

Answer:

1. Add another condition to make the visualization clearer and reduce the overfitting.

```
if len(set(y)) == 1 or self._totally_same(X):                    # must stop
    return self.majority_vote(y, sample_weights)
if X.shape[0] < self.min_samples_leaf or depth > self.max_depth:   # cut
    return self.majority_vote(y, sample_weights)
```

Figure 1: visualization of shallow decision tree with *max_depth=2*

Actually, leaves with same label can be combined together to reduce the space. I don't do that, because different branches are more clear to visualize and debug.

2. I choose two pieces of train data with id 0 and 1.

```
train.head()
```

	Pclass	Survived	Sex	Age	Parch	Fare	Embarked	Has_Cabin	FamilySize	IsAlone	Title
0	1	1	0	1	0	3	1	1	1	1	4
1	3	0	1	2	0	0	0	0	1	1	1

Figure 2: The data point I choose with label 1 (survived) and 0 (not survived).

For id 0:
- "Title"=4
- "Pclass"=1
- Survived.(Correct)

For id 1:
- "Title"=1
- "Has_Cabin"=0
- Not Survived.(Correct)

3. If we add these two parameters, the overfitting situation will be reduced partly or even cause underfitting.
   - *max_depth* limits the height of the decision tree.
   - *min_data_leaf* limits branches when one node contains not enough data.

   max_depth can be regarded as $+\infty$ and *min_data_leaf* can be regarded as 1 in the normal decision trees.

4. I use entropy (information gain) method and build the decision tree without much controls ($\text{min\_data\_leaf} = 1, \text{max\_depth} = 10$).
   The train accuracy is 87.9%, while the test accuracy is 79.0%.
   Obviously it meets **Overfitting** problem.

```
dt = DecisionTree(criterion='entropy', max_depth=10, min_samples_leaf=1, sample_feature=False)
dt.fit(X, y)
y_train_pred = dt.predict(X)
print (calc_height(dt._tree))
print("Accuracy on train set: {}".format(accuracy(y, dt.predict(X))))
print("Accuracy on test set: {}".format(accuracy(y_test, dt.predict(X_test))))
```

```
10
Accuracy on train set: 0.8787010506208214
Accuracy on test set: 0.7900763358778626
```

Figure 3: The result for decision tree.

5. First I iterate *method* in ['entropy', 'infogain_ratio', 'gini'], *min_samples_leaf* in range $[1, 10)$ and *max_depth* in range $[3, 11)$ to choose the best hyper-parameters. **The only metric is: the average validation accuracy in k-fold cross-validation.**

   The best average validation accuracy over all is 79.8% with *method=infogain_ratio*, *min_samples_leaf*$= 5$, *max_depth*$= 5$.
   The train accuracy 80.6% and the test accuracy is 78.6%.

   I'm not satisfied with that. The train accuracy can reach $86\% \sim 88\%$ without much controls, so 80.6% **seems underfitting**.

   So I adjust the choosing algorithm. I use **composed accuracy** to evaluate the parameters, with *composed accuracy=0.7validation accuracy + 0.3train accuracy*.

```
best_acc, best_min_samples_leaf, best_max_depth = 0, 0, 0
best_method = ""
for now_method in ['entropy', 'infogain_ratio', 'gini']:
    for now_min_samples_leaf in range(1, 10):
        for now_max_depth in range(5, 11):
            dt = DecisionTree(criterion=now_method, max_depth=now_max_depth, min_samples_leaf=now_min_samples_leaf, sa
            ave_acc = 0
            for train_indice, valid_indice in kf.split(X, y):
                X_train_fold, y_train_fold = X.loc[train_indice], y.loc[train_indice]
                X_val_fold, y_val_fold = X.loc[valid_indice], y.loc[valid_indice]
                dt.fit(X_train_fold, y_train_fold)
                y_train_pred = dt.predict(X_train_fold)
                y_valid_pred = dt.predict(X_val_fold)
                ave_acc += accuracy(y_val_fold, y_valid_pred) * 0.7 + accuracy(y_train_fold, y_train_pred) * 0.3
            ave_acc /= 5
            print(now_method, now_min_samples_leaf, now_max_depth, ave_acc)
            if ave_acc > best_acc:
                best_acc = ave_acc
                best_method = now_method
                best_min_samples_leaf = now_min_samples_leaf
                best_max_depth = now_max_depth
```

Figure 4: The code for improved choosing algorithm.

The best composed validation accuracy over all is 80.8% with *method=infogain_ratio*, *min_samples_leaf*$= 1$, *max_depth*$= 8$. **The train accuracy is** 83.4% **while the test accuracy is** 80.9%**, which improves** $2 + \%$ **than before.**

```
0.8083764484150884 infogain_ratio 1 8
```

```
# report the accuracy on test set
dt = DecisionTree(criterion=best_method, max_depth=best_max_depth, min_samples_leaf=best_min_samples_leaf, sample_feature=False)
dt.fit(X, y)
print("Accuracy on train set: {}".format(accuracy(y, dt.predict(X))))
print("Accuracy on test set: {}".format(accuracy(y_test, dt.predict(X_test))))
```

```
Accuracy on train set: 0.833810888252149
Accuracy on test set: 0.8091603053435115
```

Figure 5: The result for the chosen hyper-parameters.

(b) Implement Random Forest (in random forest.py), then answer the following questions.

Answer:

With the help of numpy function *np.random.choice*, I could implement random forest within a few lines of codes.

Bootstrap is used to select different samples for differest trees.

```
def _get_bootstrap_dataset(self, X, y):
    """Create a bootstrap dataset for X.

    Args:
        X: training features, of shape (N, D). Each X[i] is a training sample.
        y: vector of training labels, of shape (N,).

    Returns:
        X_bootstrap: a sampled dataset, of shape (N, D).
        y_bootstrap: the labels for sampled dataset.
    """
    # YOUR CODE HERE
    # TODO: re-sample N examples from X with replacement
    # begin answer
    chosen_data = np.random.choice(np.arange(0, X.shape[0]), X.shape[0], replace=True)
    return X[chosen_data], y[chosen_data]
    # end answer
```

Figure 6: The code for randomly select samples.

And I should also modify *decision_tree.py* to support *sample_feature=True*. Typically, I set $m = \sqrt{M}$ and select m **different** features from all features.

```
if self.sample_feature:
    m = max(int(D**0.5), 1)
    ids = np.random.choice(range(D), m, replace = False)
    best_feature_idx = max([(idx, self.criterion(X, y, idx, sample_weights)) for idx in ids], key = lambda x: x[1])[0]
else:
    best_feature_idx = max([(idx, self.criterion(X, y, idx, sample_weights)) for idx in range(D)], key = lambda x: x[1])[0]
# end answer
return best_feature_idx
```

Figure 7: The code for randomly select features.

1. One of good parameters for decision trees in **random forest** are *method=gini*, *min_samples_leaf*$= 7$, *max_depth*$= 4$.
   While one of good parameters for **single decision tree** are *method=infogain_ratio*, *min_samples_leaf*$= 1$, *max_depth*$= 8$.

For decision trees in random forest, *min_samples_leaf* becomes bigger and *max_depth* becomes smaller. It is because multiple decision trees tend to choose the parameters with small bias and large variance.

2. I use the previous *composed accuracy* to find the best parameters.
When $k = 10$, I enumerate *method* in ['entropy', 'infogain_ratio', 'gini'], *max_depth* in range $[3, 10)$ and *min_samples_leaf* in range $[1, 10)$ to choose the best hyperparameters. It takes about 30 minutes to work out $3 \times 9 \times 7 = 189$ possibilities. Among these results, *mathod=gini, min_samples_leaf=8, max_depth=4* has the best *composed accuracy*.
The train accuracy is 83.0% and the test accuracy is 82.1%.

```
# report the accuracy on test set
# k=10
# begin answer
base_learner = DecisionTree(criterion=best_method, max_depth=best_max_depth, min_samples_leaf=best_min_samples_leaf, sample_feature=True)
rf = RandomForest(base_learner=base_learner, n_estimator=10, seed=2020)
# end answer
rf.fit(X, y)
print("Accuracy on train set: {}".format(accuracy(y, rf.predict(X))))
print("Accuracy on test set: {}".format(accuracy(y_test, rf.predict(X_test))))
```

```
Accuracy on train set: 0.8299904489016237
Accuracy on test set: 0.8206106870229007
```

Figure 8: The results for $k = 10$.

$k = 100$ **runs very slow**, so I decide to reduce the combination of enumerations. First I use *method=entroy*, *min_leaf_size*$= [1, 3]$, $k = 100$ and collect results.

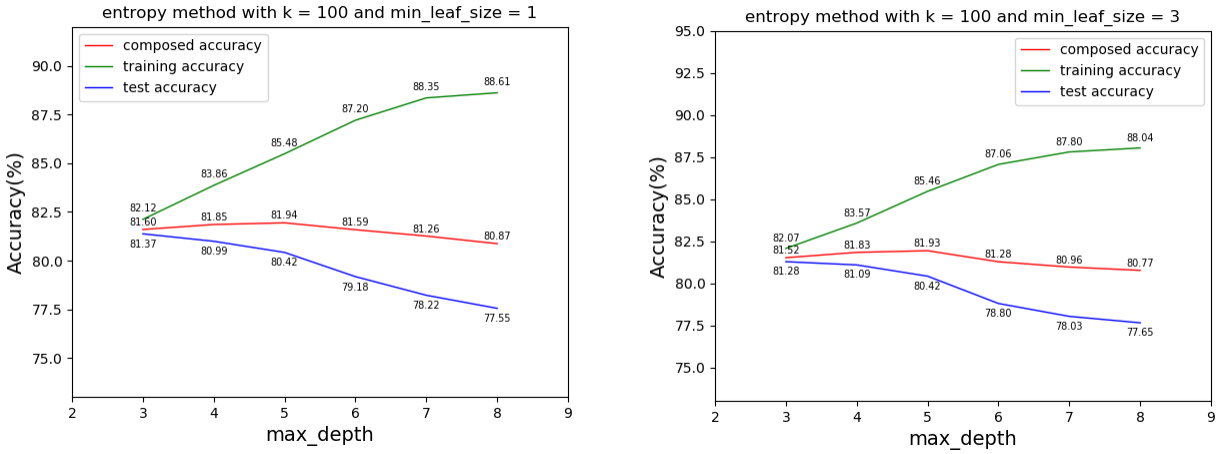

Figure 9: The results for $k = 100$ when *min_leaf_size*$= 1, 3$.

From the above figure, decision trees in random forest prefer smaller *max_depth* than single decision tree, so I adjust the range of *max_depth* to range $[3, 4, 5]$. And I set the range of *min_leaf_size* to $[1, 3, 5, 7, 9]$ to reduce possible combinations.

It takes about an hour to work out $3 \times 3 \times 5 = 45$ possibilities. Among these results, *mathod=gini, min_samples_leaf=7, max_depth=4* has the best *composed accuracy*. (This result is similar with $k = 10$).

```
# report the accuracy on test set
# k=100
# begin answer
base_learner = DecisionTree(criterion='gini', max_depth=best_max_depth, min_samples_leaf=best_min_samples_leaf, sample_feature=True)
rf = RandomForest(base_learner=base_learner, n_estimator=100, seed=2020)
# end answer
rf.fit(X, y)
print("Accuracy on train set: {}".format(accuracy(y, rf.predict(X))))
print("Accuracy on test set: {}".format(accuracy(y_test, rf.predict(X_test))))
```

```
Accuracy on train set: 0.8347659980897804
Accuracy on test set: 0.8206106870229007
```

Figure 10: The results for $k = 100$.

The train accuracy is 83.5% and the test accuracy is 82.1%.

3. From my expriment results, the random forest prefer smaller max_depth and bigger $\min\_ leaf\_size$. That means random forest tends to choose more steadier data(i.e. small bias and large variance). And it will shorten the variance.

4. The result has been reported before.

(c) Implement Adaboost (in adaboost.py), then answer the following questions.

Answer:

1. I tried some parameters and make the following table. It's clear that *min_leaf_size* influences the result **slightly**, but bigger *max_depth* makes both accuracy and time **worse**. So I can understand why people often use decision stump. It is reasonable, because Adaboost prefer base learners with low bais and high variance. Besides, decision stump also **relieves** the overfitting problem.

Table 1: The result for Adaboost with $k = 50$.

Test ID	min_leaf_size	max_depth	time(s)	train accuracy	test accuracy
1	1	1	20.0	81.4%	80.9%
2	1	3	28.7	87.1%	80.9%
3	5	1	20.3	81.4%	80.9%
4	5	3	28.3	87.1%	79.4%
5	1	2	24.3	84.5%	79.8%

2. According to the previous result, I use *min_leaf_size=max_depth=1* to check the influence of k (the number of decision trees).

   When $k = 10$, the train accuracy is 79.7% and the test accuracy is 79.0%.
   When $k = 100$, the train accuracy is 80.9% and the test accuracy is 81.3%.

3. From the table in the previous page, Adaboost improves (reduces) variance over a single decision tree. Adaboost (with proper parameters) will relieve the overfitting problem and become steadier.

4. I use $k = 50$, and enumerate *method* in ['entropy', 'infogain_ratio', 'gini'], *max_depth* in range $[1, 2, 3]$ and *min_samples_leaf* in range $[1, 3, 5]$ to choose the best hyper-parameters. It takes about an hour to work out $3 \times 3 \times 3 = 27$ possibilities. Among these results, *mathod=infogain_ratio, min_samples_leaf=3, max_depth=1* has the best *composed accuracy*.

   The train accuracy is 81.2%, and the test accuracy is 80.9%.

```
# report the accuracy on test set
# begin answer
# end answer
base_learner = DecisionTree(criterion='infogain_ratio', max_depth=1, min_samples_leaf=3, sample_feature=False)
ada = Adaboost(base_learner=base_learner, n_estimator=50, seed=2020)

ada.fit(X, y)
print("Accuracy on train set: {}".format(accuracy(y, ada.predict(X))))
print("Accuracy on test set: {}".format(accuracy(y_test, ada.predict(X_test))))
```

```
Accuracy on train set: 0.8118433619866284
Accuracy on test set: 0.8091603053435115
```

Figure 11: The result for *min_samples_leaf=3, max_depth=1, k=50*.

(d) Report the result of all learned classifiers and state their advantages and disadvantages (as many as you can).

Answer: I have tested all the methods mentioned in the problem. For fairness, all algorithms are tested with sklearn[1] instead of my own implementation (to avoid some mistakes). The parameters of all methods are the default except in special cases.

I have to mention that my approaches to decision tree, random forest and adaboost are **a little bit better** than sklearn, which gives me a great sense of achievement.
Besides, the training accuracy of almost all methods is higher than the testing accuracy, but not too much. It shows that many algorithms will occur the overfitting problem.

[1]The website introducing sklearn API: https://scikit-learn.org/stable/modules/classes.html

Table 2: The results for different sklearn methods on Titanic.

Algorithm	Training Accuracy	Testing Accuracy
Naive Bayes	76.7%	75.8%
Perception	62.3%	61.8%
Logistic Regression	79.8%	78.6%
SVM	81.9%	81.3%
Neural Network	81.8%	**82.8%**
KNN	82.8%	77.9%
Decision Tree	**87.7%**	80.2%
Random Forest	**87.7%**	81.3%
Adaboost	81.0%	81.3%

Some conclusions:

- **Naive Bayes** is a simple (just doing some counts) approach yet perform well and robust to noise in practice. The idea of posterior is really inspiring.
- **Perception** is a mathematical-driven algorithm. Usually not performs well.
- **Logistic Regression** is great and also explainable. The output can be interpreted as a probability, which supports convenient in many situations. It is widely used and along with some other techniques like Lasso or Ridge.
- **Support Vector Machines** is a really excellent linear model (or maybe the best). It is useful in a high-dimensional space with high accuracy. There also exists some kernel methods to extends its function.
- **K-Nearest-Neighbors** is a simple and widely known algorithm. Direct use of KNN usually not performs well, but it can be combined with other methods.
- **Neural network** performs best among all methods in this dataset. Deep Learning is really hot nowdays, and also widely used in the industry. However, it is really unexplainable compared with other methods.
- **Decision Tree** is a fast algorithm with easy and intuitive logic, but it's easy to meet the overfitting problem. It is widely used in ensemble algorithms because of its simple but robust structure.
- **Random Forest** is less likely to overfit compared with Decision Tree. Because each tree is independent, it's good for parallel or distributed computing.
- **Adaboost** is a good ensemble algorithm. It will *collect* the result from many basic(weak) learners and iterate a more powerful one.

Problem. Advanced1: Implement GBDT (Gradient Boosting Decision Tree)

(a) Introduction to GBDT

Answer: Gradient Boosting Decision Tree is a typical implementation of Gradient boosting[2]. Like other boosting methods, gradient boosting combines weak "learners" into a single strong learner in an iterative fashion. GBDT wants to predict values of the form $\hat{y} = F(x)$ by minimizing the loss function $Loss(F(x), y)$.

Denote F_m is the m-th function we predict. We hope F_m can correct the fault from F_{m-1} and fit the data well, such that $F_{m-1} + h_m = F_m$, or equivalently, $h_m(x) = y - F_{m-1}(x)$. Because we have already worked out F_{m-1}, it's natural to differentiate h_m with respect to F_{m-1} to optimize h_m.

So we are supposed to compute so-called **pseudo-residuals**:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n$$

.

And we fit a base learner $h_m(x)$ to pseudo-residuals. i.e. train it using the training set $(x_i, r_i m)$. Also we need to set up a *learning rate* to partly learn the new gradient. i.e.

$$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$$

.

(b) Implementation details

Answer:

Gradient Boosting Decision Tree use CART as the base learner. Notice that this tree is always **binary**, which is quite different from the decision tree I have implemented. CART will split data into two exactly parts, one with value $\leq threshold$ and the other with $> threshold$.

To avoid possible mistakes, I wrote a brand new class called **BinaryDecisionTree** to support the regression tree. Each internal node will have the following messages: *best feature id, the threshold of feature value, left son, right son.*

When fitting dataset (x_i, y_i), Each node not only splits data and passes it to sons, but also collects the return values from sons and reunions them as a list. This step is essential because it will help GBDT update F_m.

$$c_{m,j} \simeq \frac{\sum_{x_i \in R_{m,j}} r_{m,i}}{\sum_{x_i \in R_{m,j}} (y_i - r_{m,i})(1 - y_i + r_{m,i})}$$

$$F_m(x) = F_{m-1}(x) + \sum_{j=1}^{J_m} c_{m,j} I(x \in R_{m,j})$$

[2]The following introductions partly refer to Wikipedia and A detailed tutorial

```
mytree = dict()
best_index, best_value = self._choose_best_feature(X, res)
mytree[feature_names[best_index]] = {}
new_feature_names = feature_names.copy()
# delete this feature name
del new_feature_names[best_index]

# split data
l_data, r_data = X.T[best_index] <= best_value, X.T[best_index] > best_value
l_X, l_y, l_res = np.delete(X[l_data], best_index, axis = 1), y[l_data], res[l_data]
r_X, r_y, r_res = np.delete(X[r_data], best_index, axis = 1), y[r_data], res[r_data]

# new_f will be reunioned
new_f = np.zeros(X.shape[0])
mytree[feature_names[best_index]][0], new_f[l_data] = self._build_tree(l_X, l_y, l_res, new_feature_names, depth + 1)
mytree[feature_names[best_index]][1], new_f[r_data] = self._build_tree(r_X, r_y, r_res, new_feature_names, depth + 1)
mytree[feature_names[best_index]]['value'] = best_value
```

Figure 12: The core code to split and reunion data.

```
def _calc_leaf(self, y, res):
    numerator = np.sum(res)
    denominator = np.sum((y - res) * (1 - y + res))
    #print (y, res, numerator / denominator)
    if abs(denominator) <= 1e-10:
        denominator = 1e-10 if denominator >= 0 else -1e-10
    return numerator / denominator
```

Figure 13: The core code to calculate $c_{m,j}$.

Then I build another class called **GDBT**. In this class, I assemble many base learners and make m iterations to fit the data calculated by the previous step.

```
def fit(self, X, y):
    """Build the Adaboost according to the training data.

    Args:
        X: training features, of shape (N, D). Each X[i] is a training sample.
        y: vector of training labels, of shape (N,).
    """
    N = X.shape[0]
    y = np.array(y)
    positive = np.sum(y) / N
    self.f[0] = np.full(N, math.log(positive / (1 - positive)))
    for idx in range(self.n_estimator):
        res = y - 1 / (1 + np.exp(-self.f[idx]))
        dt = self._estimators[idx]
        new_f = dt.fit(X, y, res)
        self.f[idx+1] = self.f[idx] + self.learning_rate * new_f

    return self
```

Figure 14: The core code to fit data in GDBT.

(c) Experimental results and analysis

Answer: I print a last decision tree when $k = 10$ and *max_depth=3*.

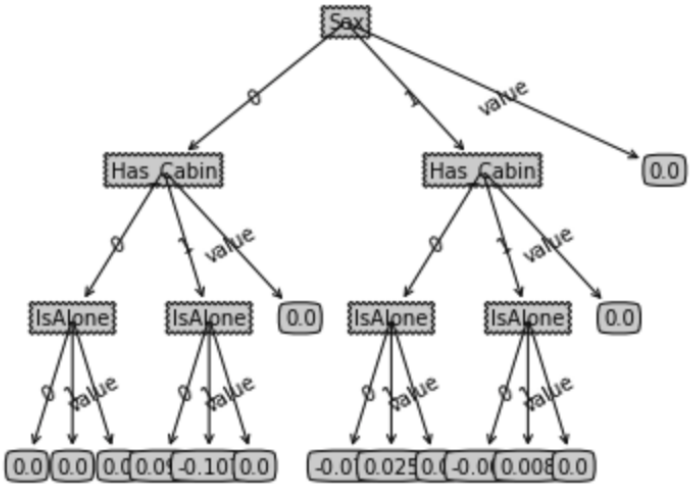

Figure 15: The tree structure for $k = 10$, *max_depth=3*.

Then I adjust some hyper-parameters to find better results. Generally speaking, the result of GBDT is quite **similar with** single decision tree, random forest and Adaboost.

If k (i.e. the number of base learners) is enough (≥ 50), the result for GBDT is quite **steady**. **Steady** means that the train and test accuracy doesn't change too much. Besides, the latter approaches the former, indicating that there is no overfitting problem.

Here is a result when $k = 100$, $lr = 0.6$, *max_depth=5* and *min_leaf_size=1*. The train accuracy is 82.1% and the test accuracy is 80.9%.

```
from GBDT import GBDT
from binary_decision_tree import BinaryDecisionTree

base_learner = BinaryDecisionTree(max_depth = 5, min_samples_leaf = 1)
gbdt = GBDT(base_learner = base_learner, n_estimator = 100, learning_rate = 0.6)

gbdt.fit(X, y)
print (accuracy(gbdt.predict(X),y))
print (accuracy(gbdt.predict(X_test),y_test))
```

```
0.8213944603629417
0.8091603053435115
```

Figure 16: The train accuracy and test accuracy for $k = 100$, $lr = 0.6$.

(d) Difficulties and your solutions

Answer:

It takes me **really a long time** to implement GBDT completely.

The main problem is, after I read a little time on Wikipedia, I thought I have known GBDT very well and I started to write BinaryDecisionTree at once.

Soon I was stucked by some details, and I checked the website again and fixed it. After that, I met another problem... Finally I decided to reviewe the algorithm thoroughly. I **didn't even expect** that the task of each iteration is to fit the data **provided by the previous one**, which means I didn't understand GBDT at all!

Another problem is, I spent a long time to **design the structure** of *build_tree*. The predicted value in each leaf should be passed to GBDT class so that we can update F_i. Finally I reconstruct the total structure, use the return value to pass them.

Problem. Advanced2: Pruning decision tree

(a) Implementation details

Answer:

There two major pruning strategies: **Prepruning** and **Postpruning**[3].

First is Prepruning, in which while building the decision tree keep on checking whether tree is overfitting. If so, directly cut the branches and leave as a leaf. Second is Postpruning, in which the tree is build first and then do some reduction.

I implement both Prepruning and Postpruning. Both of two methods need another pair of (X, y), means the validation data.

For the former, we should take a judgement **after** we find the best feature to divide but **before** we build branches. If the sum of divisions we split in validation data is worse than the whole data(i.e. Calculate the majority-vote in the whole data, compared with the sum of majority-vote for each part.), we should do pruning.

For the latter, we collect all the validation for subtrees and sum them. If this number is smaller than the majority-vote of whole data, we give up these branches.

Notice: Postpruning is a strategy we assure, but Prepruning is just an evaluation.

(b) Experimental results and analysis

Answer:

Besides accuracy, I also calculate **the average nodes and the time**, too.

[3]This part I refer to the book *Machine Learning* written *by Prof. Zhihua Zhou*

```
for repeat in range(10):
    dt = DecisionTree(criterion='entropy', max_depth=10, min_samples_leaf=repeat, sample_feature=False)
    dt.fit(X, y)
    #dt.fit_with_cut(X_train, y_train, X_valid, y_valid)
    ave_train += accuracy(y, dt.predict(X))
    ave_test += accuracy(y_test, dt.predict(X_test))
    ave_node += dt.calc_nodes(dt._tree)
```

```
time:  3.8547439575195312
Average accuracy on train set: 0.8611270296084049
Average accuracy on test set: 0.7942748091603054
Average nodes: 217.7
```

Figure 17: No cut method is used.

```
for repeat in range(10):
    dt = DecisionTree(criterion='entropy', max_depth=10, min_samples_leaf=repeat, sample_feature=False, first_cut = True)
    #dt.fit(X, y)
    dt.fit_with_cut(X_train, y_train, X_valid, y_valid)
    ave_train += accuracy(y, dt.predict(X))
    ave_test += accuracy(y_test, dt.predict(X_test))
    ave_node += dt.calc_nodes(dt._tree)
```

```
time:  1.8420741558074951
Average accuracy on train set: 0.7987583572110794
Average accuracy on test set: 0.784732824427481
Average nodes: 14.6
```

Figure 18: Use Prepruning.

```
for repeat in range(10):
    dt = DecisionTree(criterion='entropy', max_depth=10, min_samples_leaf=repeat, sample_feature=False, last_cut = True)
    #dt.fit(X, y)
    dt.fit_with_cut(X_train, y_train, X_valid, y_valid)
    ave_train += accuracy(y, dt.predict(X))
    ave_test += accuracy(y_test, dt.predict(X_test))
    ave_node += dt.calc_nodes(dt._tree)
```

```
time:  2.9122109413146973
Average accuracy on train set: 0.8187201528175742
Average accuracy on test set: 0.8076335877862595
Average nodes: 44.4
```

Figure 19: Use Postpruning.

It's obvious that:

1. Two methods will both decrease nodes. Prepruning remains **very small nodes**.
2. Prepruning runs very fast, and then does Postpruning. No cut runs very slow.
3. The accuracy is similar. The accuracy of Prepruning reduces the a little.
4. To my surprise, Postpruning improves the accuracy a little, and **relieves reduce the overfitting problem**. The *Machine Learning* book mentions that Postpruning is a steady solution, and now I **have a deep understanding** of this!

(c) Difficulties and your solutions

Answer:

At first, I had trouble searching the pruning strategy. Then I opened the *Machine Learning book* and found Prof.Zhou illustrated it very well.

It takes some time to implement the whole code.

Problem. Advanced3: Continuous Feature in the decision tree

(a) Implementation details

Answer:

In advanced 1(Gradient Boosting Decision Tree), I implement a new class called **BinaryDecisionTree**, which is just a continuous tree(also called regression trees).

The change from classification tree to regression tree is quite easy. If we have a decisive feature *age* and the values are $[5, 15, 30, 60]$, we can set up a threshold K so that the data will be splited into two parts: $age \leq K$ and $age > K$. Tipically we try all the midpoints (i.e. $\frac{a_i+a_{i+1}}{2}$) and find the best one with minimum loss function.

```
def _choose_best_feature(self, X, y):
    """Choose the best feature to split according to criterion.

    Args:
        X: training features, of shape (N, D).
        y: vector of training labels, of shape (N,).

    Returns:
        (int): the index for the best feature
        (int): the value for the best feature
    """
    best_feature_idx, best_feature_value = 0, 1e100
    D = X.shape[1]
    for idx in range(D):
        valuelist = np.unique(X.T[idx])
        valuelist = [valuelist[0]] + valuelist
        best_value = min([(value, self.criterion(X, y, idx, value)) for value in valuelist], key = lambda x: x[1])[0]
        if best_value < best_feature_value:
            best_feature_value = best_value
            best_feature_idx = idx
    return best_feature_idx, best_feature_value
```

Figure 20: The core code for the regression tree.

(b) Experimental results and analysis

Answer:

I directly use **BinaryDecisionTree** to do some experiments. Since the accuracy is similar to the normal decision tree, I will focus on other things.

```
base_learner = BinaryDecisionTree(max_depth = 5, min_samples_leaf = 3)
gbdt = GBDT(base_learner = base_learner, n_estimator = 100, learning_rate = 0.6)

gbdt.fit(X, y)
pred_test = gbdt.predict(X_test)
print (accuracy(gbdt.predict(X),y))
print (accuracy(pred_test,y_test))
```

```
0.8213944603629417
0.8091603053435115
```

Figure 21: The accuracy of regression tree is similar with normal decision tree.

If a decision tree supports the regression(real) value when it is predicting(Typically through the logistic function so that $[-\infty, +\infty] \rightarrow [0, 1]$), will the value with high confidence be more accurate? It is something like **confidence intervals**: If the decision tree reports 0.55 and 0.9 towards two samples, can we say that 0.9 is more likely to be survived. And I find that the answer is yes.

I write a function to calculate the accuracy for *different confidence intervals*.

```
def choose(th):
    chosen_data = np.logical_or(real_test <= th, real_test >= 1 - th)
    print (np.sum(chosen_data), accuracy(pred_test[chosen_data], y_test[chosen_data]))

choose(0.5)
choose(0.3)
choose(0.1)
```

```
262 0.8091603053435115
200 0.865
72 0.9027777777777778
```

Figure 22: The accuracy for *different confidence intervals*.

0.5 denotes the total accuracy which is around 80%. And if we select the prediction with value ≤ 0.1 or ≥ 0.9, the result will reach 90%.

(c) Difficulties and your solutions

Answer: After I checked the Machine Learning book, I found that the same feature is allowed to appear on the tree more than once. I didn't expect that, and directly delete a feature if it appears on a node.

Problem. Advanced4: Parallel random forest

(a) Implementation details

Answer: Besides **Thread**, python supports another parallel package called **Multiprocessing**[4]. We need to add a corresponding function for each code section expected to be paralleled. Then just create a new Multiprocessing.Process. The usage and operations to process is same as thread (*start, join, . . .*).

```
if not self.is_parallel:
    for idx in range(self.n_estimator):
        y_pred_origin[idx] = self._estimators[idx].predict(X)
else:
    parts = 4
    number = (self.n_estimator + parts - 1) // parts
    for idx in range(0, N, number):
        idy = min(idx + number, N)
        p = multiprocessing.Process(target = self.parallel_predict, args = (idx, idy, y_pred_origin, X))
        p.start()

print (self.is_parallel)
#if not self.is_parallel:
for idx in range(N):
    number, times = Counter(y_pred_origin.T[idx]).most_common(1)[0]
    y_pred[idx] = number
else:
    for idx in range(0, N, number):
        idy = min(idx + number, N)
        p = multiprocessing.Process(target = self.parallel_vote, args = (idx, idy, y_pred_origin, y_pred))
        p.start()
```

Figure 23: The core code for multiprocessing.

(b) Experimental results and analysis (you should report speedup here)

Answer: I don't get any positive result.

(c) Difficulties and your solutions

Answer: The terrible thing is, the program **seems slower** if I apply multiprocessing. I have searched on the Internet, and there is a mainstream view that Python's multi-threading is not good enough. I also suspect the jupyter notebook, but I can not work out directly using Python, too.

Now I think it's mainly about the platform. Windows uses **spawn** instead of **fork**, so creating a process will be really slow. I also try on **WSL** but other problems occur.

I also talked with my classmates Guanglin Li and Qihang Zhang. The former met the same problem with me, and the latter told me a even more weird thing: His code sometimes works but sometimes not.

[4]The website: https://docs.python.org/3.7/library/multiprocessing.html#module-multiprocessing.sharedctypes