The Implementation of Decision Tree

Cheng Tan

1. The implementation of the decision tree algorithm.

(1) Node

```
# My DecisionTree
class Node():
    """
        A Node in the decision tree.

        Parameters
        ----------
        value : double, default=None
            The possible value.

        best_feat : int, default=None
            The best feature to split in the current node.

        best_thr : double, default=None
            The best threshold of the best feature.

        leaf_branch : Node, default=None
            The left branch of the current node.

        right_branch : Node, default=None
            The right branch of the current node.
    """
    def __init__(self, value, best_feat, best_thr, left_branch, right_branch):
        self.value = value
        self.best_feat = best_feat
        self.best_thr = best_thr
        self.left_branch = left_branch
        self.right_branch = right_branch
```

(2) Decision Tree

```
class DecisionTree():
    """
        A classifier based on decision tree.

        Parameters
        ----------
```

```python
        criterion : {'id3', 'c45'}, default='id3'
            The criterion function to measure the quality of a split.

        max_depth : int, default=None
            The maximum depth of the tree.

        min_sample_leaf : int, default=None
            The minimum numbers of samples required to be at a leaf node.

        min_impurity_split : double, default=None
            Threshold for early stopping in tree growth.

    """
    def __init__(self, criterion='id3', max_depth=None, \
                 min_sample_leaf=4, min_impurity_split=1e-7):
        self.criterion = criterion
        self.max_depth = (np.iinfo(np.int32).max if max_depth is None else max_depth)
        self.min_sample_leaf = min_sample_leaf
        self.min_impurity_split = min_impurity_split
        self.root = None

    def __entropy__(self, x):
        """
            Calculate the entropy.

            Returns
            -------
            H : double
                The entropy of x.
        """
        D = x.shape[0]
        H = 0
        for k in np.unique(x):
            Ck = len(x[x == k])
            H += - (Ck / D) * np.log2(Ck / D)
        return H

    def __gini__(self, x):
        """
            Calculate the gini coefficient.

            Returns
            -------
            G : double
```

```
                The gini coeeficient of x.
        """
        D = x.shape[0]
        G = 0
        for k in np.unique(x):
            Ck = len(x[x == k])
            G += (Ck / D) ** 2
        G = 1 - G
        return G

    def __id3__(self, y, yl, yr):
        """
            Information gain.

            Returns
            -------
              : double

        """
        D, Dl, Dr = len(y), len(yl), len(yr)
        H_D = self.__entropy__(y)
        H_Dl = Dl / D * self.__entropy__(yl)
        H_Dr = Dr / D * self.__entropy__(yr)
        return H_D - (H_Dl + H_Dr)

    def __c45__(self, y, yl, yr):
        """
            Information gain ratio.

            Returns
            -------
              : double
        """
        D, Dl, Dr = len(y), len(yl), len(yr)
        H_D = self.__entropy__(y)
        H_Dl = Dl / D * self.__entropy__(yl)
        H_Dr = Dr / D * self.__entropy__(yr)
        return (H_D - (H_Dl + H_Dr)) / H_D

    def __cart__(self, y, yl, yr):
        """
            Gini coefficient.

            Returns
```

```python
            -------
              : double
        """
        D, Dl, Dr = len(y), len(yl), len(yr)
        G_Dl = Dl / D * self.__gini__(yl)
        G_Dr = Dr / D * self.__gini__(yr)
        # To get minimum gini, add negative symbol
        return - (G_Dl + G_Dr)

    def __criterion__(self, y, yl, yr):
        """
            Choose one of the criterion function to measure the split.

            Returns
            -------
              : double
        """
        if self.criterion == 'id3':
            return self.__id3__(y, yl, yr)
        elif self.criterion == 'c45':
            return self.__c45__(y, yl, yr)
        else:
            raise ValueError('The criterion should be one of [\'id3\', \'c45\'].')

    def __getThr__(self, feature):
        """
            Get the initialized thresholds of a feature.

            Returns
            -------
              : list
        """
        t = sorted(feature)
        return [(t[i] + t[i-1]) / 2 for i in range(1, len(t))]

    def __getSplit__(self, X, y, feat_ind, thr):
        """
            Get the split of X and y.

            Returns
            -------
              : tuple of list
        """
        t = X[:, feat_ind] < thr
```

```python
        return X[t], y[t], X[~t], y[~t]

    def __build__(self, X, y, depth):
        """
            Build the decision tree.

            Returns
            -------
              node : Node
        """
        node = Node(None, None, None, None, None)
        row, col = X.shape

        if depth <= self.max_depth and row > self.min_sample_leaf:
            # Find the best feature to split
            bfeat_ind = None
            bfeat_gain = np.iinfo(np.int32).min
            bfeat_thr = None
            for feat_ind in range(col):
                thresholds = self.__getThr__(X[:, feat_ind])
                for thr in thresholds:
                    X1, y1, X2, y2 = self.__getSplit__(X, y, feat_ind, thr)
                    gain = self.__criterion__(y, y1, y2)
                    if gain >= bfeat_gain:
                        bfeat_gain = gain
                        bfeat_thr = thr
                        bfeat_ind = feat_ind

            if bfeat_gain > self.min_impurity_split:
                # Continue splitting
                X1, y1, X2, y2 = self.__getSplit__(X, y, bfeat_ind, bfeat_thr)
                node.best_thr = bfeat_thr
                node.best_feat = bfeat_ind
                node.value = None
                del X
                node.left_branch = self.__build__(X1, y1, depth + 1)
                node.right_branch = self.__build__(X2, y2, depth + 1)
            else:
                # Stop splitting
                del X
                node.value = np.argmax(np.bincount(y.flatten()))
            return node
        else:
            del X
```

```python
            node.value = np.argmax(np.bincount(y.flatten()))
            return node

    def __find__(self, x, node):
        """
            Find the potential predicted label in the decision tree.

            Returns
            -------
              : int
        """
        if node.value is None:
            if x[node.best_feat] < node.best_thr:
                return self.__find__(x, node.left_branch)
            else:
                return self.__find__(x, node.right_branch)
        else:
            return node.value

    def fit(self, X, y):
        assert isinstance(X, np.ndarray) and isinstance(y, np.ndarray)
        if y.ndim == 1:
            y = np.reshape(y, (-1, 1))
        # Build tree
        self.root = self.__build__(X, y, 0)

    def predict(self, X):
        assert isinstance(X, np.ndarray)
        y = []
        for x in X:
            y.append(self.__find__(x, self.root))
        return np.array(y)

    def score(self, X, y):
        assert isinstance(X, np.ndarray) and isinstance(y, np.ndarray)
        pred = self.predict(X)
        return (pred == y).sum() / pred.shape[0]
```

2. Comparison

(1) Accuracy

Accuracy (%)	Fold1	Fold2	Fold3	Fold4	Fold5	Avg.
Built-in	92.98	95.61	91.22	96.49	92.92	93.85
Mine	93.86	95.61	92.98	94.73	95.57	94.55

(2) Training Time

Time (s)	Fold1	Fold2	Fold3	Fold4	Fold5	Avg.
Built-in	0.0081	0.0069	0.0064	0.0062	0.0080	0.0071
Mine	4.9146	5.5333	5.1658	5.1595	5.2357	5.2018

(3) Test Time

Time (ms)	Fold1	Fold2	Fold3	Fold4	Fold5	Avg.
Built-in	0.4890	0.6101	0.3319	0.2789	0.3757	0.4171
Mine	0.2398	0.2649	0.2441	0.2868	0.2698	0.2611