Artificial Intelligence HW2 Report

0716214 江岳勳

實作簡介

使用環境為 Python 3.8.7，使用的第三方套件請參照 Appendix. B.

- **Decision Tree**

  訓練時，從根節點出發，將 data 和對應的 label 傳入建出一棵樹。

  每個節點在一開始的時候會先檢查此節點是否應為葉節點。檢查的依據為：傳入的 label 中只剩下一個 unique 的值、節點深度抵達限制或傳入的 sample 數量低於限制。

  若此節點應為葉節點，則從 label 中選出出現次數最多的最為此節點的回答；若非葉節點則挑出幾個 feature，在納入考慮的 feature 中挑出 Gini's impurity 最小的 threshold 值作為此節點在之後預測時的分割依據，並將 data 和 label 以此 threshold 分為兩組作為兩個 child node 的 data 和 label 進行遞迴建樹。

  - ✓ **Feature bagging (attribute bagging)**

    在分割節點時只考慮部分的 feature，而不計算所有 feature，選擇方式為隨機挑選$\min(\mathrm{F}(m), m_{splitable})$個 feature。其中 m 為 feature 數量，F(m)為$\sqrt{m}$ (在 extremely random 的情況為 1)。

  預測時則根據傳入的 data 尋訪建立好的樹，抵達葉節點時給出葉節點的答案。

- **Random Forest**

  訓練時，使用傳入的 data 和 label 分別訓練 n 棵 Decision Tree。

  預測時，根據每棵樹對同一筆 data 的回答，選出最多棵樹給出的答案 (majority vote)。

  - ✓ Tree bagging

    在訓練時將 training set 分為跟樹數量同樣的 n 組，訓練第 i 棵樹的時後不使用第 i 組 data，來做到每棵樹使用不一樣的 subset。

結果統計

以下為使用 5 棵樹、深度限制 8、sample 數無限制的 Random Forest 執行 10 個 episode 的 K-fold 驗證(k=8)的平均結果：

Dataset	Iris	Bezdek Iris	Breast Cancer	Glass	Ionosphere	Wine
Training Accuracy	100.00%	100.00%	91.91%	97.38%	98.61%	100.00%
Testing Accuracy	94.35%	94.08%	76.43%	72.08%	92.28%	96.30%
Time Elapsed	< 3s	< 3s	9s	22s	82s	13s

實驗探討

- **執行時間**

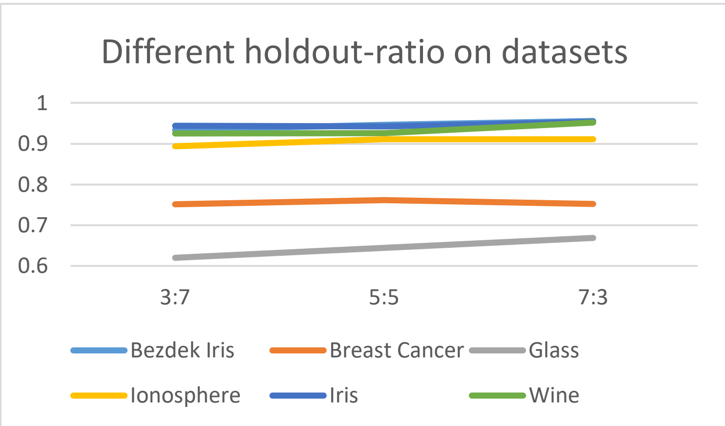
  執行時間我認為應該跟 dataset 的 feature 數量有直接的關係。因為 feature 的多寡會影響到深度時有沒有剩餘的 feature 能夠再繼續分下去。由上**結果統計**部分可以發現，Iris (4 個 feature)的執行時間較 Ionosphere (34 個 feature)來的快非常多，應該是因為使用 Iris 執行時數並沒有長的過深的必要。

- **Categorical Feature 的處理**

  雖然 spec 只要求我們做 Real attributes 的 dataset，但我還是有處理 categorical 的部分。處理方式是將 categorical 的 feature 進行 one-hot，將該 feature 中的每個類別分離出來變為 0 或 1，如此一來在選取 threshold 時只會選到 0.5，這個節點的意義就能轉為「feature X 的值是否為 Y？」。

- **Relative sizes of the training and validation subsets**

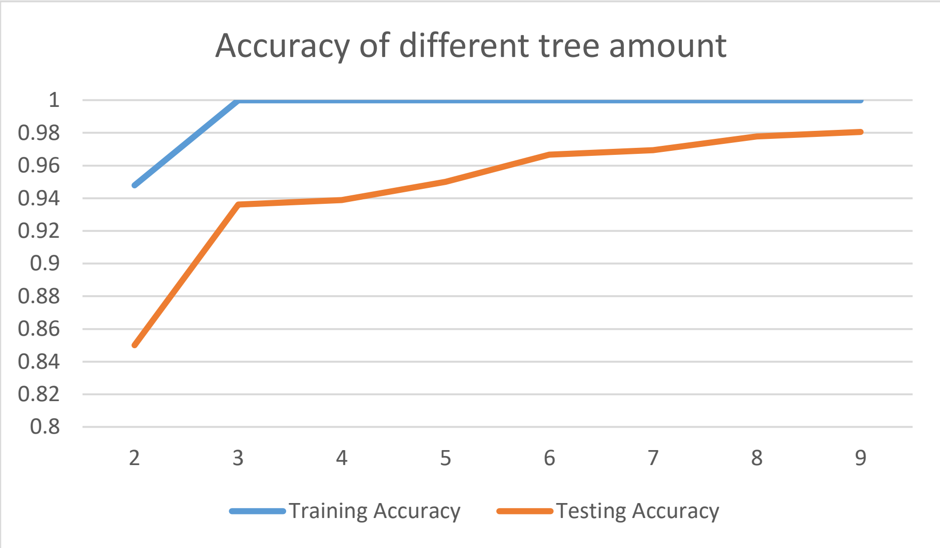
  使用 Holdout-validation 來進行測試 (5 棵樹、深度 4、sample 數無限制)：



可以看到，隨著 training data 的增加，大部分的 dataset 都有著些微的表現提升，沒有顯著差距的原因我認為應該是 datatset 本身不是很複雜的關係。

- **Number of trees in the forest**

  使用 Wine 資料集來進行測試 (Kfold(k=8)、深度 8、sample 數無限制)：



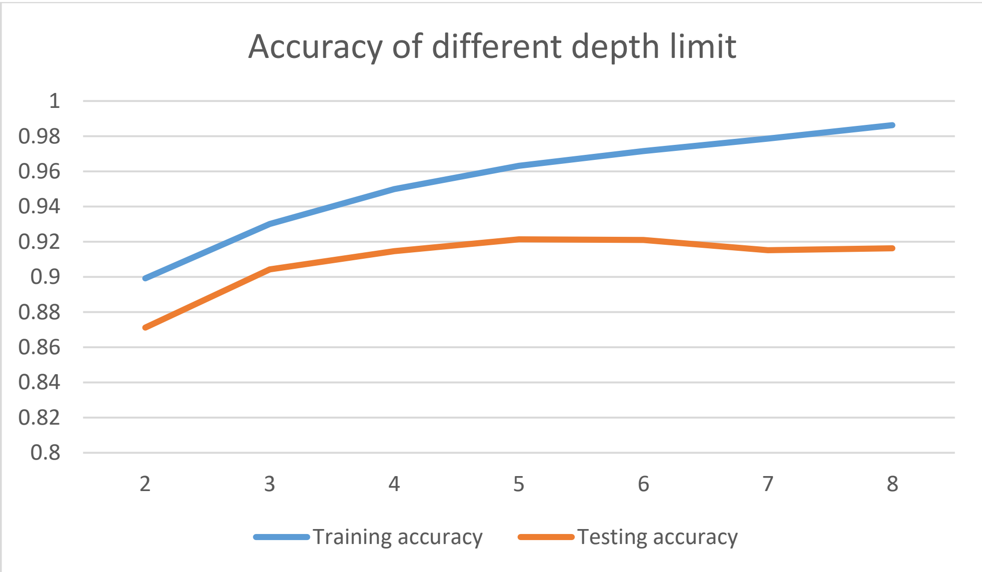
  可以看到，提升樹的數量對於提升 testing accuracy 有著明顯的幫助。

- **Parameters used during tree induction**

  在選擇 feature 的部分，就如**實作簡介**部分提到的，我預設是從可切割的 m 個 feature 中隨機挑選$\sqrt{m}$個出來選擇最佳 threshold。至於 extremely random forest 的部分我認為和 depth limit 有些許關係，所以我將其放在後面討論。

- **Depth Limit**

  深度限制會直接影響較為複雜的資料集的表現。舉 Ionosphere 資料集為例 (5 棵樹、Kfold(k=8)、5 個 episode 平均)：

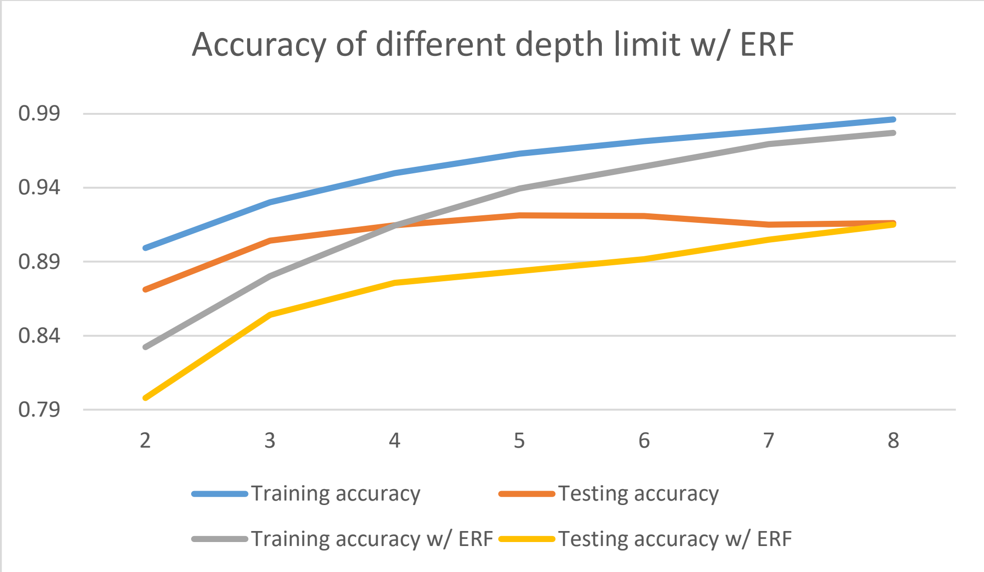


  可以看到，增加限制深度可以很好的提升準確度，尤其是在 training accuracy 方面，深度的增加讓模型可以更好的切割資料，使 out-of-bag error 下降。

- ✓ **Extremely Random Forest**

  Extremely Random Forest 因為在選擇 feature 時完全隨機，所以深度會直接影響其能夠選到有意義的 feature 的機率 (單個 node 的機率不變，但選擇的次數變多了)

  以下是上面 Ionosphere 資料集的實驗加上 Extremely RF 的結果：



  可以看出，depth limit 越大，Extremely Random Forest 和普通 Random Forest 的表現差距就越小。

Appendix 附錄

目錄

Appendix.A Github 連結

https://github.com/hm-ysjiang/ArtificialIntelligence-HW2

Appendix.B pip-requirements

```
numpy>=1.20.1
pandas>=1.2.3
requests
tqdm
```

Appendix.C main.py

```
import numpy as np
import tqdm

from models import DecisionTree, RandomForest
from utils import Dataset, compute_accuracy


def model_provider(use_forest=True):
    n_trees = 5
    tree_bagging = True
    feature_bagging = True
    depth_lim = 8
    min_samples = 0

    if use_forest:
        return RandomForest(n_trees, tree_bagging, feature_bagging, d
epth_lim, min_samples)
    return DecisionTree(feature_bagging, depth_lim, min_samples)


def run(dataset, use_forest=True, use_holdout=False, holdout_ratio=0.
8, kfold=8, episodes=5):
    assert use_holdout or kfold >= 1

    accu_scores_train = []
    accu_scores = []
    with tqdm.tqdm(total=episodes * (1 if use_holdout else kfold)) as
 pg:
        for _ in range(episodes):
            if use_holdout:
                train_data, train_label, test_data, test_label = data
set.holdout(
                    holdout_ratio)
                model = model_provider(use_forest)
                model.train(train_data, train_label)
                accu_scores_train.append(
```

```
                    compute_accuracy(train_label, model.predict(train
_data)))
                accu_scores.append(
                    compute_accuracy(test_label, model.predict(test_d
ata)))
                pg.update(1)
            else:
                for train_data, train_label, test_data, test_label in
 dataset.kfold(kfold):
                    model = model_provider(use_forest)
                    model.train(train_data, train_label)
                    accu_scores_train.append(
                        compute_accuracy(train_label, model.predict(t
rain_data)))
                    accu_scores.append(
                        compute_accuracy(test_label, model.predict(te
st_data)))
                    pg.update(1)
    print('Training data accuracy: %.2f%%' %
          (100 * np.average(accu_scores_train)))
    print('Testing data accuracy: %.2f%%' %
          (100 * np.average(accu_scores)))


if __name__ == '__main__':
    # Uncomment this line to enable extreme DecisionTree
    # DecisionTree._F_bagging_policy = lambda x: 1

    run(Dataset.Wine, use_forest=True, use_holdout=False, holdout_rat
io=0.8, kfold=8, episodes=10)
```

Appendix.D models.py

```
import numpy as np

from utils import compute_gini, kfold_indices


class DecisionTree:
    class Node:
        def __init__(self, depth=0):
            self.depth = depth
            self.feature = None
            self.threshold = None
            self.label = None
            self.child1 = None
            self.child2 = None

        def split(self, data, label, feature_bagging, depth_lim, min_
samples):
            # Set node label if only one label presents
            if np.unique(label).shape[0] == 1:
                self.label = label[0]
                return
            # Set node label if limit reached
            if self.depth >= depth_lim or label.shape[0] <= min_sampl
es:
                self.label = np.argmax(np.bincount(label))
                return
            n_samples, n_features = data.shape
            # Consider only sqrt(n_features) features
            n_consider = DecisionTree._F_bagging_policy(n_features)
            # Find splitable features
            splitable = list(filter(lambda feature: np.unique(data[:,
 feature]).shape[0] > 1,
                                    list(range(n_features))))
            n_splitable = len(splitable)
            if n_splitable > 0:
                min_gini = None
```

```
                # Iterate through the features chosen
                for feature in (np.random.choice(splitable, min(n_consider, n_splitable), replace=False) if feature_bagging else range(n_features)):
                    # Sorted unique elements
                    values = np.unique(data[:, feature])
                    for idx in range(len(values) - 1):
                        # Try midpoints between each two unique values
                        threshold = (values[idx] + values[idx+1]) / 2

                        # Compute total Gini impurity
                        g1 = data[:, feature] <= threshold
                        g2 = np.invert(g1)
                        n1 = g1.sum()
                        n2 = n_samples - n1
                        gini = n1 * compute_gini(label[g1]) \
                            + n2 * compute_gini(label[g2])

                        # Updates
                        if self.feature is None or min_gini is None or gini < min_gini:
                            min_gini = gini
                            self.feature = feature
                            self.threshold = threshold
            else:   # If no splitable feature
                self.label = np.argmax(np.bincount(label))
                return

            # Split the data into two groups and continue the split of children
            g1 = data[:, self.feature] <= self.threshold
            g2 = np.invert(g1)
            self.child1 = DecisionTree.Node(self.depth+1)
            self.child1.split(data[g1], label[g1],
                              feature_bagging, depth_lim, min_samples)
            self.child2 = DecisionTree.Node(self.depth+1)
```

```
            self.child2.split(data[g2], label[g2],
                              feature_bagging, depth_lim, min_samples
)

        def __call__(self, data):
            if self.label is not None:
                return self
            else:
                return self.child1 if data[self.feature] <= self.thre
shold else self.child2

    def _F_bagging_policy(x):
        return max(1, round(np.sqrt(x)))

    def __init__(self, feature_bagging=True, depth_lim=8, min_samples
=0):
        """Initialize a Decision Tree Classifier

        Args:
            feature_bagging (bool, optional): Enable Feature-
bagging or not. Defaults to True.
            depth_lim (int, optional): The depth limit of the tree. D
efaults to 8.
            min_samples (int, optional): The minimum amount of sample
s in each node. Defaults to 0.
        """
        assert depth_lim > 0
        assert min_samples >= 0

        self.feature_bagging = feature_bagging
        self.depth_lim = depth_lim
        self.min_samples = min_samples
        self.root = DecisionTree.Node()
        self.trained = False

    def train(self, data, label):
        """
        Args:
```

```
            data (Iterable): An iterable contains training data, the
dimension should be (samples, features)
            label (Iterable): An iterable contains training data, the
 dimension should be (samples, )
        """
        assert not self.trained, 'This tree has already been trained!
'
        self.root.split(data, label, self.feature_bagging,
                        self.depth_lim, self.min_samples)
        self.trained = True

    def predict(self, data):
        """
        Args:
            data (Iterable): An iterable contains testing data, the d
imension should be (samples, features)

        Returns:
            numpy.ndarray: An array of dim (samples, ), contains the
predictions of each input
        """
        assert self.trained, 'This tree has not been trained yet!'

        res = [self.root] * data.shape[0]
        for _ in range(self.depth_lim):
            res = [node(data[idx]) for idx, node in enumerate(res)]
        return np.array([node.label for node in res])


class RandomForest:
    def __init__(self, n_tree=5, tree_bagging=True, feature_bagging=T
rue, depth_lim=8, min_samples=0):
        """Initialize a Random Forest Classfier

        Args:
            n_tree (int, optional): The number of trees in this fores
t. Defaults to 5.
            tree_bagging (bool, optional): Enable Tree-
bagging or not. Defaults to True.
```

```
            feature_bagging (bool, optional): Enable Feature-bagging or not. Defaults to True.
            depth_lim (int, optional): The depth limit of each tree. Defaults to 8.
            min_samples (int, optional): The minimum amount of samples in each tree's node. Defaults to 0.
        """
        assert n_tree >= 1
        assert tree_bagging or feature_bagging

        self.n_tree = n_tree
        self.tree_bagging = tree_bagging
        self.trees = [DecisionTree(feature_bagging, depth_lim, min_samples)
                      for _ in range(n_tree)]
        self.trained = False

    def train(self, data, label):
        """
        Args:
            data (Iterable): An iterable contains training data, the dimension should be (samples, features)
            label (Iterable): An iterable contains training data, the dimension should be (samples, )
        """
        assert not self.trained, 'This forest has already been trained!'
        if not type(data) is np.ndarray:
            data = np.array(data)
        if not type(label) is np.ndarray:
            label = np.array(label)

        if self.tree_bagging:
            for idx, fold_idx in enumerate(kfold_indices(self.n_tree, label.shape[0])):
                self.trees[idx].train(data[fold_idx[0]], (label[fold_idx[0]]))
        else:
```

```
            [tree.train(data, label) for tree in self.trees]
        self.trained = True

    def predict(self, data):
        """
        Args:
            data (Iterable): An iterable contains testing data, the dimension should be (samples, features)

        Returns:
            numpy.ndarray: An array of dim (samples, ), contains the predictions of each input
        """
        assert self.trained, 'This forest has not been trained yet!'
        if not type(data) is np.ndarray:
            data = np.array(data)
        return np.array([np.argmax(np.bincount(votes)) for votes in np.array([tree.predict(data) for tree in self.trees]).T])
```

Appendix.E utils.py

```
from pathlib import Path
import numpy as np
import pandas as pd
import requests as req


def compute_gini(clazz):
    return 1 - sum([(c/clazz.shape[0]) ** 2 for c in np.bincount(claz
z)])


def compute_accuracy(a, b):
    try:
        iter(a)
        a = np.array(a)
    except TypeError:
        a = np.array([a])
    try:
        iter(b)
        b = np.array(b)
    except TypeError:
        b = np.array([b])
    assert len(a.shape) == 1 and len(b.shape) == 1 and a.shape == b.s
hape, \
        'a and b should have the same dimension of (n, )'
    return (a == b).sum() / a.shape[0]


def kfold_indices(k, dimension):
    fold_size = dimension // k
    remainer = dimension % k
    fold_sizes = [fold_size + 1 if _ < remainer else fold_size for _
in range(k)]
    counter = 0
    for fold in fold_sizes:
        test_fold = np.array([True if counter <= x < counter + fold e
lse False
```

```
                            for x in range(dimension)])
        counter += fold
        yield np.invert(test_fold), test_fold


class Dataset:
    FTYPE_REAL = 0
    FTYPE_CATEGORICAL = 1
    FTYPE_UNUSED = 2
    FTYPE_CLASS = 3

    BezdekIris: 'Dataset' = None
    BreastCancer: 'Dataset' = None
    Glass: 'Dataset' = None
    Ionosphere: 'Dataset' = None
    Iris: 'Dataset' = None
    Wine: 'Dataset' = None

    def __init__(self, filepath, feature_types, header=None, delim=',
', dl_url=None):
        """Initialize a Dataset

        Args:
            filepath (str): path to the csv file
            feature_types (Iterable / Callable): An iterable contains
 feature type of each column, or a callable that gives corresponding
feature type from column index
            header (Iterable, optional): The header of the csv file.
Defaults to None.
            delim (str, optional): The string used to seperate column
s in the csv file. Defaults to ','.
            dl_url (str, optional): The url of the data file to downl
oad if the file does not present. Defaults to None.
        """
        # Read csv in
        fp = Path(filepath)
        if not fp.exists():
            if dl_url is not None:
                print('Downloading dataset %s' % filepath)
```

```
                res = req.get(dl_url)
                if not fp.parent.exists():
                    fp.parent.mkdir(parents=True)
                with open(filepath, 'wb') as file:      # Write conte
nt with LF instead of CRLF
                    file.write(res.text.encode())
            else:
                raise FileNotFoundError(
                    'Cannot find file %s, and no dl_url provided.' %
filepath)
        df = pd.read_csv(filepath, sep=delim, header=header)
        # Sanity check the feature types
        feature_types = Dataset._sanity_check_ftypes(
            df.shape[1], feature_types)
        # Drop unused columns
        unused_features = [x[0] for x in filter(
            lambda x: x[1] == Dataset.FTYPE_UNUSED, enumerate(feature
_types))]
        df.drop(columns=unused_features, inplace=True)
        # Split feature and class label
        class_label = [x[0] for x in filter(
            lambda x: x[1] == Dataset.FTYPE_CLASS, enumerate(feature_
types))]
        target_classes = df[class_label].to_numpy().reshape(-1)
        df.drop(columns=class_label, inplace=True)
        # One-hot categorical features
        cate_features = [x[0] for x in filter(
            lambda x: x[1] == Dataset.FTYPE_CATEGORICAL, enumerate(fe
ature_types))]
        df = pd.get_dummies(df, columns=cate_features).astype('float3
2')
        # Encode the class labels
        self._class_dict = {}
        self._r_class_dict = []
        _class = []
        for clazz in target_classes:
            if clazz not in self._class_dict:
                self._class_dict[clazz] = len(self._r_class_dict)
```

```
                self._r_class_dict.append(clazz)
            _class.append(self._class_dict[clazz])
        # Set final results
        self._class = np.array(_class)
        self._data = df.to_numpy()

    def holdout(self, train_test_ratio=0.7, shuffle=True):
        """Holdout validation

        Args:
            train_test_ratio (float, optional): The ratio of train da
ta to split the dataset. Defaults to 0.7.
            shuffle (bool, optional): Should the data be shuffled. De
faults to True.

        Returns:
            tuple: (train_data, train_labels, test_data, test_labels)
        """
        assert 0 <= train_test_ratio <= 1, 'Train-
Test ratio should be in [0, 1]'
        data, class_ = self._shuffle() \
            if shuffle else (self._data.copy(), self._class.copy())
        sep = int(self._class.shape[0] * train_test_ratio)
        return data[:sep, :], class_[:sep], data[sep:, :], class_[sep
:]

    def kfold(self, k=3, shuffle=True):
        """K-Fold validation

        Args:
            k (int, optional): The 'K' in kfold. Defaults to 3.
            shuffle (bool, optional): Should the data be shuffled. De
faults to True.

        Yields:
            tuple: (train_data, train_labels, test_data, test_labels)
        """
        data, class_ = self._shuffle() \
```

```
            if shuffle else (self._data.copy(), self._class.copy())
        res = []
        for train, test in kfold_indices(k, class_.shape[0]):
            res.append((data[train, :], class_[train],
                        data[test, :], class_[test]))
        return res

    def convert_label(self, x):
        try:
            iter(x)
            # Handle x as a list
            return np.array([self._r_class_dict[_] for _ in x])
        except TypeError:
            # Handle x as a single value
            return self._r_class_dict[x]

    @property
    def data(self):
        return self._data.copy()

    @property
    def clazz(self):
        return self._class.copy()

    def _shuffle(self):
        tmp = np.concatenate((self._class.reshape(-
1, 1), self._data), axis=1)
        np.random.shuffle(tmp)
        return tmp[:, 1:], tmp[:, 0].astype(np.int32)

    def _sanity_check_ftypes(n_features, ftypes):
        if hasattr(ftypes, '__call__'):
            ftypes = [ftypes(x) for x in range(n_features)]
        else:
            assert len(ftypes) == n_features, \
                'The feature_types length does not match with the inp
ut data. Expecting %d, got %d' \
                % (n_features, len(ftypes))
```

```
            ftypes = list(ftypes)
        for idx, x in enumerate(ftypes):
            if not 0 <= x <= 3:
                ftypes[idx] = Dataset.FTYPE_REAL
                print('Detect an undefined feature type: %d, this value will be changed to FTYPE_REAL. Check Dataset.FTYPE_* for proper usage.' % x)
        return tuple(ftypes)

    def __len__(self):
        return self._class.shape[0]

    def __getitem__(self, idx):
        return self._data[idx].reshape(1, -1), self._class[idx]


Dataset.BezdekIris = Dataset('./data/iris/bezdekiris.data',
                             lambda x: Dataset.FTYPE_REAL if x != 4 else Dataset.FTYPE_CLASS,
                             dl_url='https://archive.ics.uci.edu/ml/machine-learning-databases/iris/bezdekIris.data')

Dataset.BreastCancer = Dataset('./data/breast-cancer/breast-cancer.data',
                               [Dataset.FTYPE_CATEGORICAL] * 6 +
                               [Dataset.FTYPE_REAL] +
                               [Dataset.FTYPE_CATEGORICAL] * 2 +
                               [Dataset.FTYPE_CLASS],
                               dl_url='https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer/breast-cancer.data')

Dataset.Glass = Dataset('./data/glass/glass.data',
                        [Dataset.FTYPE_UNUSED] +
                        [Dataset.FTYPE_REAL] * 9 +
                        [Dataset.FTYPE_CLASS],
                        dl_url='https://archive.ics.uci.edu/ml/machine-learning-databases/glass/glass.data')
```

```
Dataset.Ionosphere = Dataset('./data/ionosphere/ionosphere.data',
                             lambda x: Dataset.FTYPE_CLASS if x == 34
 else Dataset.FTYPE_REAL,
                             dl_url='https://archive.ics.uci.edu/ml/m
achine-learning-databases/ionosphere/ionosphere.data')

Dataset.Iris = Dataset('./data/iris/iris.data',
                       [Dataset.FTYPE_REAL,
                        Dataset.FTYPE_REAL,
                        Dataset.FTYPE_REAL,
                        Dataset.FTYPE_REAL,
                        Dataset.FTYPE_CLASS],
                       dl_url='https://archive.ics.uci.edu/ml/machine
-learning-databases/iris/iris.data')

Dataset.Wine = Dataset('./data/wine/wine.data',
                       lambda x: Dataset.FTYPE_CLASS if x == 0 else D
ataset.FTYPE_REAL,
                       dl_url='https://archive.ics.uci.edu/ml/machine
-learning-databases/wine/wine.data')
```