Intro to AI lab 4 - Random Forest, Implementation and Discussion

Information

- Author: Yan-Tong Lin
- 0712238@NCTU, National Chiao Tung University, Taiwan
- E-mail
  - 0312fs3@gmail.com

- I did the random forest python implemtation all by my self.
- The report statements are developed by my self. (not from wiki or something else)

Task description

- Implementation of random forest from scatch.
- Discussion on topics of one's choice.
- The dataset
  - I developped my code of random forest on the dataset "iris"
  - After realizing the dataset is too trivial for a random forest, "Optical Recognition of Handwritten Digits" was chosen to be my experiment playground
  - the description of the datasets can be found in reference [2]

Description of random forest

Decision Tree

A decision tree is a machine learning model trained on a fixed set of data.
It **recursively devides the given training data** to groups by attributes determined by **a measurement of purity**, and predict coming instances by the devisions trained.
Decision trees are **strong models** since they can do arbitrarily well on a giving trainning set.

CART and Gini's impurity

In the case of CART, the tree devides data to **2** groups at once, and the criterion of (im)purity is **Gini's index**.
$G=1-\sum_{ci=0}p_{2i}$ or $\sum_{ci=0}p_i(1-p_i)$

Random Forest

Bagging(bootstrap aggregating) is a technique/meta-algorithm in machine learning aiming to **decrease variance** and increase stability.
The core idea is that **strong models** tend to overfit noises; However, multiple strong models tend to agree on signals instead of noises.
The random forest algorithm is a classical example of bagging model, and its base models are decision trees.

Please refer to the original paper[1] by Breiman for more details.

Tree bagging

For each **tree**, bootstrap(sample with replacement) $n_{population}$ data from the original dataset.

Feature bagging

Sample (without replacement) k of d attributes of the original data for each **split**.
k is suggested to be $\sqrt{d}$ for classification tasks by the original paper[1].

Implementation

This is a brief decription of my implementation of Node, CART and RandomForest classes.
For more details, refer to the appendix code.

Generally, the main functions of a decision tree is in the “split” method of the “Node” class.
Other classes are basically wrappers.

Node

- Attributes
  - c, the children array, c[0] = left, c[1] = right
  - sf, split feature
  - th, threshold
  - maj, majarity of the training data in the subtree rooted at this node
  - depth, depth of the node(for cutting)
  - leaf, if this node is a leaf node
- Supported methods
  - split(data, options…)

Cart

- Attributes
  - rt, the root node of the decision tree
- Supported methods
  - train(dataset, options…)
  - predict(vector)
  - calc_ac(dataset)

Randon Forest

- Attributes(sk learn style)
  - dtrees, the list of decision trees of the forest
  - n_estimators, number of estimators(decision trees)
  - criterion, “Gini” or “entropy”
  - max_features, “sqrt” or “log” or “one”
  - max_depth, for depth cut
  - min_impurity_decrease, for min impurity cut
  - bootstrap, sample data or not
  - max_samples, number of sampled data for each tree(ratio or integer or None)

- Supported methods
  - train(dataset, options…)
  - predict(vector)
  - calc_ac(dataset)

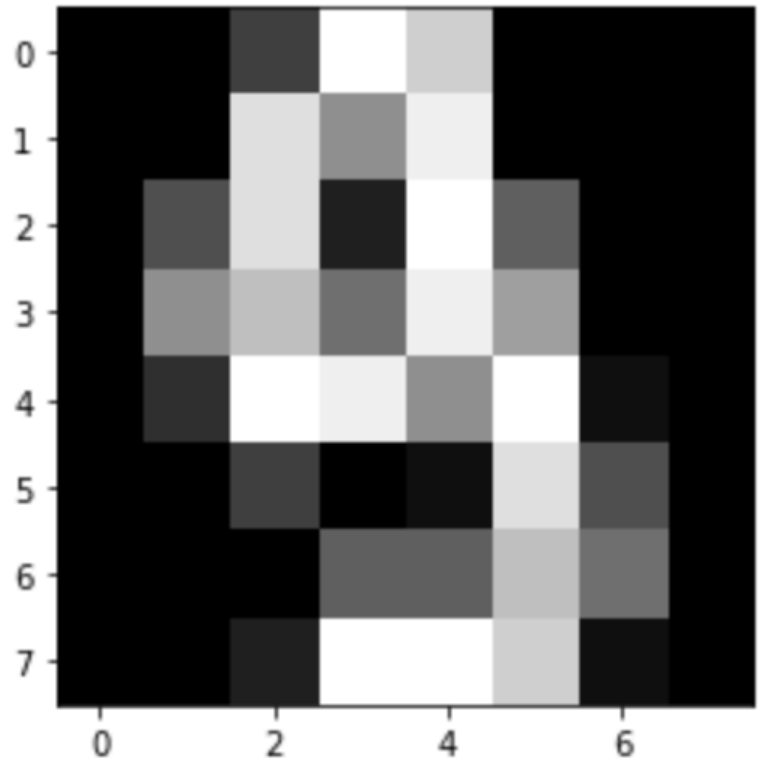
Experimental Results and Discussion

Setting of experients

- we change the param that we want to discuss, and remainings are set to defaults
- the default values for hyper params
  - n_estimators = 10
  - criterion = “Gini”
  - max_features = “sqrt”
  - cuts = No cut
  - bootstrap = True
  - max_examples = 1.0
- An experiment on a setting of a value of discussing param is conducted 10 times
- the following graph shows the confusion matrix of the default setting

```
array([[142,   0,   1,   0,   0,   0,   1,   0,   1,   0],
       [  0, 163,   0,   3,   0,   1,   1,   0,   4,   0],
       [  0,   0, 136,   0,   0,   0,   0,   0,   1,   0],
       [  0,   0,   1, 141,   0,   1,   1,   1,   0,   3],
       [  0,   0,   0,   0, 146,   0,   1,   0,   0,   1],
       [  0,   0,   0,   0,   0, 135,   1,   0,   1,   0],
       [  0,   0,   0,   0,   0,   0, 134,   0,   0,   0],
       [  0,   1,   0,   0,   0,   0,   0, 124,   0,   1],
       [  2,   1,   0,   1,   0,   1,   1,   1, 131,   2],
       [  0,   0,   1,   1,   0,   4,   0,   0,   1, 111]])
```

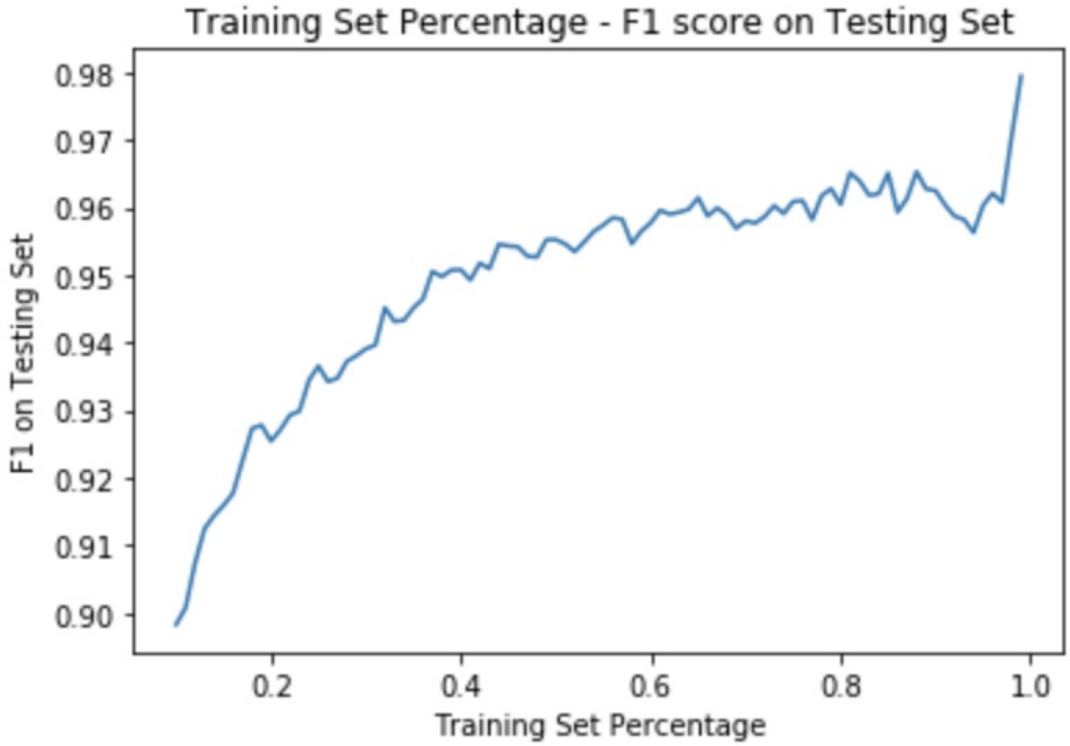
```
(8, 8, 3)
3 9
```



Choice of Order of Experiments

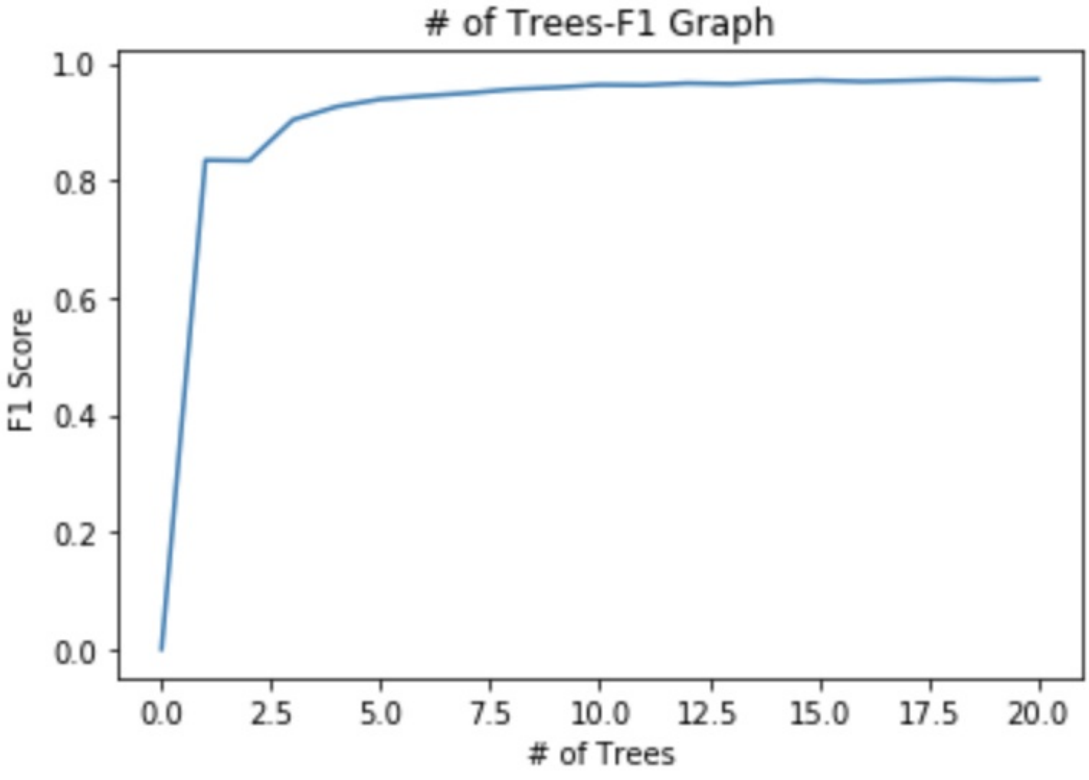
- Trainning/Validation Set Size
  - choose a better measurement
- Population first
  - to choose a suitable population for evaluation
- Others

Training Set Size



- F1 score grows as training set ratio grows bigger
- It is not a surprise since more training data means we have more infromation about the function we want to learn.
- And less validation data means it is less probable that the model get caught on an instance that it does not decribe well.
- Note that in the task 10%(562) of data is enough for achieving 90% F1 score

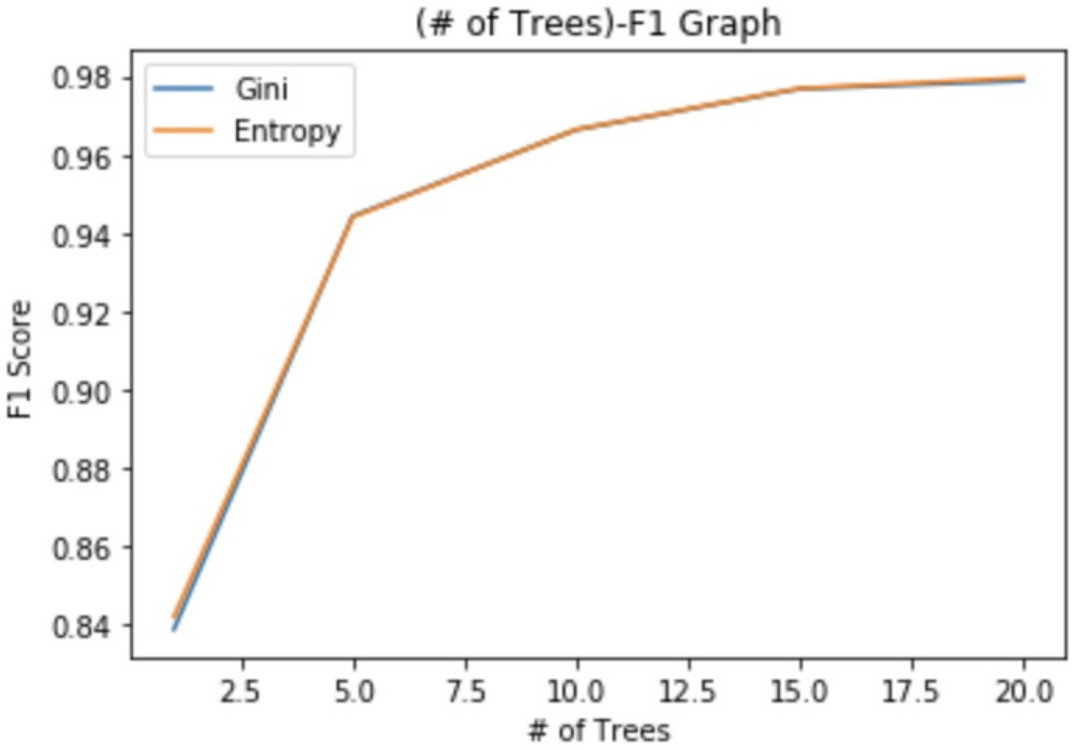
Number of trees and F1 score



- F1 score grows as training set ratio grows bigger

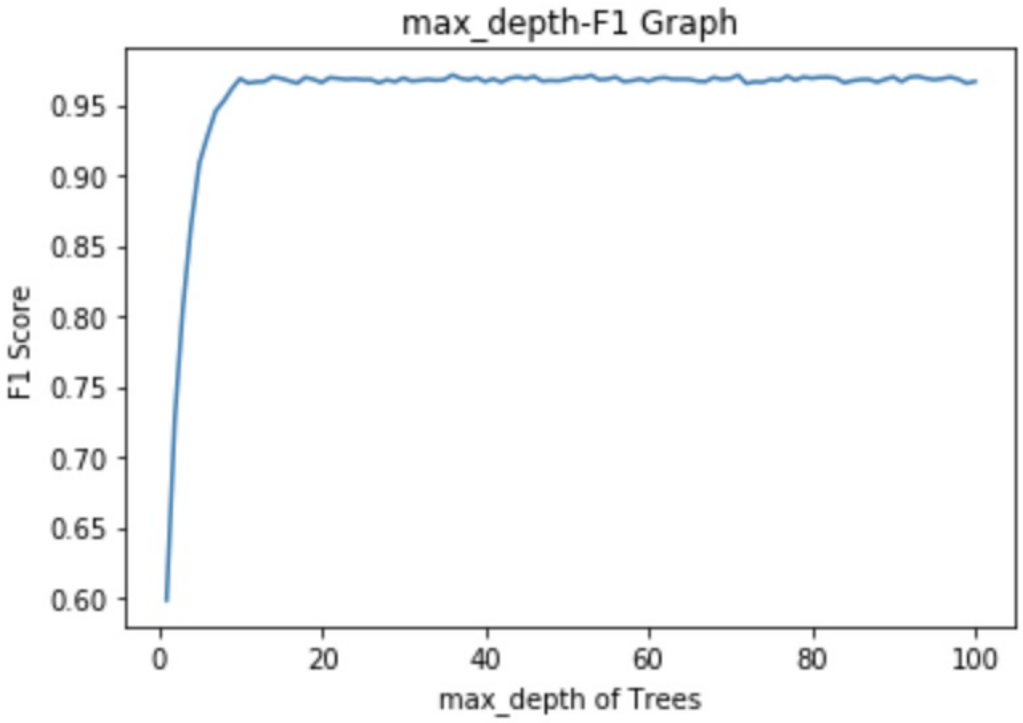
- This is not surprising either since aggregating more strong models means more powerful!
- An advantage of random forest is that it does not overfit for using more trees to vote

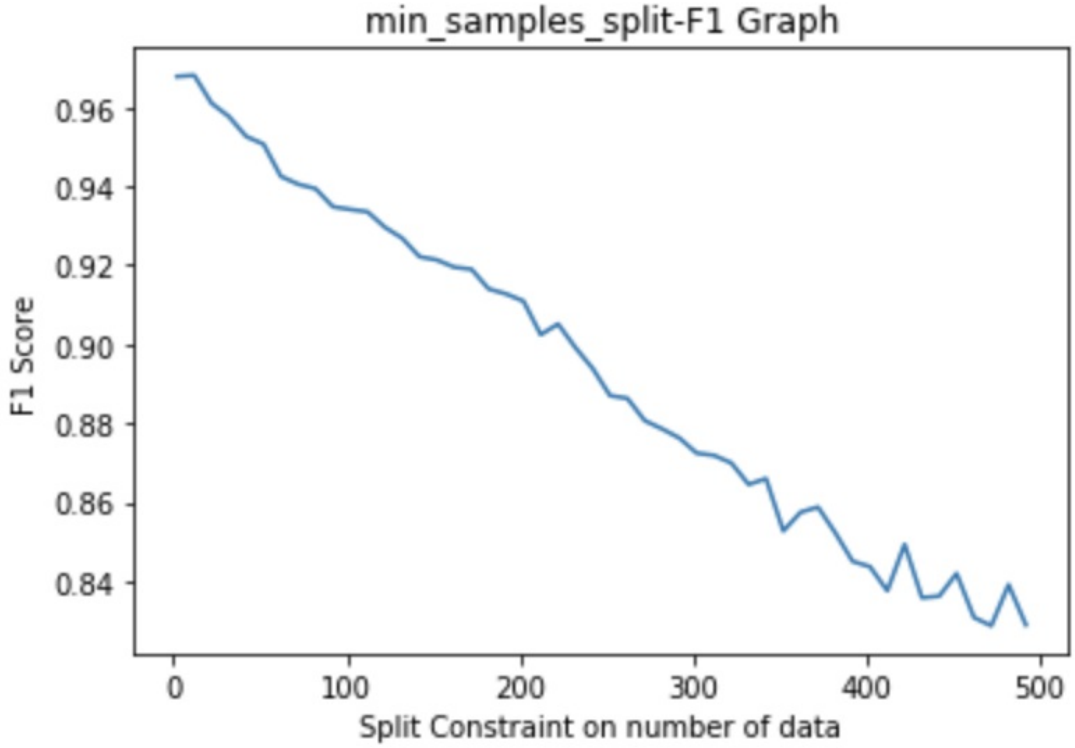
Criterion for spliting

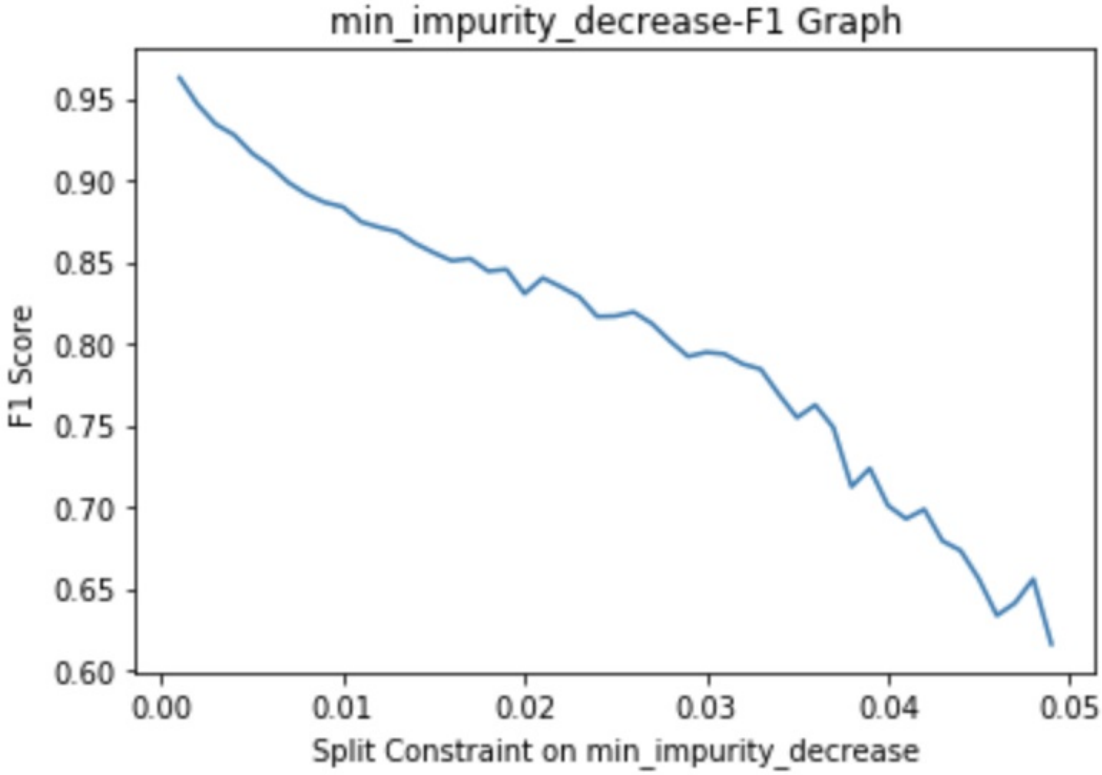


- redo the last experiment with both criterion
- the two criterions(Gini's impurity and Shannon's entropy) act almost identical
- But Gini's impurity should has a better time constant, maybe this is why CART choose Gini's instead of Shannon's.

Early Cut - Depth of Trees/Split Constraint on number of data/Min Impurity Gain







- The test F1 score rises as capacity of a single model grows
- Bagging method works on low bias, high variance model, limiting the capacity of a simgle component is not a good idea.

Conclusion and Future investigations

Random forest is indead a powerful model. Utilizing the bagging meta-algorithm, the model is able to take advantages of the model strength of decision trees while maintaining an acceptable variance. No wonder there are so many tasks on Kaggles that random forest outperforms many other algorithms. Even with neural networks rising in popularity, I think that it will remain a robust and efficient choice for

some problems.

In this assignment, I found it relatively easy to implement this model considering how powerful it is. And by conducting experiments, I am able to varify the properties of random forest.

A future work for my implementation may be optimizing the time and memory usage. Also, an investigation for me to pose is that - is there an “online” version of decision trees?

Also, an interesting counterpart to the bagging technique is the “boosting” technique, and gradient-boosted tree is a well-known example. I am looking forward to explore the algorithm in the summer vacation, with my ability to self-learn and implement AI algorithms that obtained and strengthened in this class.

Reference

- [1] Breiman, L. Random Forests. Machine Learning 45, 5–32 (2001).
  - The original paper of random forest
- [2] Dua, D. and Graff, C. (2019). UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science.
  - The datasets come from this repository
- [3] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.
  - sk learn tools - for F1 scores, confusion matrices, etc.

Appendix A - Python Implementation of Random Forest

RandomForest.py

```
#!/usr/bin/env python
# coding: utf-8
# author = 0712238, Yan-Tong Lin

import numpy  as np
import pandas as pd
import random
from collections import defaultdict


# # Hyper Params and other

fname = "./dat/iris.data"
TEST = True
n_feature = 64
n_class = 10
LAB = n_feature
epsilon = 0.00000001

# # preprocessing

def load_data(fname, partition=(8,1,1)):
    # read in, df to nparray
    df = pd.read_csv(fname, header=None)
    npd = np.array(df)
    n = npd.shape[0]

    # tag to id, id to tag, transform
    tag_map = defaultdict(int)
```

```
    tag_to_id = defaultdict(int)
    id_to_tag = defaultdict(int)
    for d in npd:
        tag_map[d[LAB]] += 1

    cnt = 0
    for k, v in tag_map.items():
        id_to_tag[cnt] = k
        tag_to_id[k] = cnt
        cnt += 1

    for i in range(n):
        npd[i][LAB] = tag_to_id[npd[i][LAB]]
    #print(tag_to_id)
    #print(id_to_tag)

    # shuffle(no when testing)
    np.random.shuffle(npd)
    if not TEST:
        np.random.shuffle(npd)

    # partition
    cut1 = int(n*partition[0]/sum(partition))
    cut2 = int((n*(partition[0] + partition[1])/sum(partition)))
    # print(cut1, cut2)
    train_set = npd[:cut1, :]
    valid_set = npd[cut1:cut2, :]
    test_set = npd[cut2:, :]

    return train_set, valid_set, test_set, tag_to_id, id_to_tag


def Gini(data, feature_id = -1, threshold = None):
    n = len(data)
    sigma = 0.0
    # first dimension, lc or rc
    # second dimension, map label type to cnt, map[n_class] = total
    group_labels = [defaultdict(int), defaultdict(int)] # group 1 stat, group 2 stat
    for d in data:
        if feature_id == -1:
            group_labels[0][d[LAB]] += 1
            group_labels[0][n_class] += 1
        else:
            group_labels[d[feature_id] < threshold][d[LAB]] += 1
            group_labels[d[feature_id] < threshold][n_class] += 1 #total
    #print(group_labels)
    for g in group_labels:
        w = g[n_class]/n
        cur = 0.0
        if(g[n_class] == 0):
            continue
        for i in range(n_class):
            p = g[i]/g[n_class]
            cur += p*p
        sigma += w*cur
    return 1 - sigma


class Node(): # subtree
    def __init__(self):
        self.c = [None, None] # child, c[0] => < threshold , c[1]
        self.sf = None # split feature
        self.th = None # threshold
        self.maj = None # majority label in this subtree
        self.depth = None #depth
        self.leaf = True
```

```
        self.leaf = True

    def split(self, data, criterion="gini", max_features=None, max_depth=None, min_im
        n = len(data)
        self.depth = depth

        # deal with min_impurity_decrease
        if min_impurity_decrease == None:
            min_impurity_decrease = epsilon # inf small to represent any gain > epsil

        # criterion => func
        if criterion == "gini":
            func = Gini
        elif criterion == "entropy":
            func = None
        else:
            print("not a valid critirien, use Gini's impurity")
            func = Gini

        # calc majority
        vote = defaultdict(int)
        for d in data:
            vote[d[LAB]] += 1
        self.maj = max(vote, key=vote.get)

        # cut if maxd exceed
        if(max_depth and self.depth >= max_depth):
            print("maxd_cut")
            self.leaf = True
            return

        # feature_mask (max_feature)
        mask = np.zeros(n_feature)
        if max_features == None :
            n_sample_f = n_feature
        elif max_features == "sqrt":
            n_sample_f = int(np.ceil(np.sqrt(n_feature)))
        elif max_features == "log":
            n_sample_f = int(np.ceil(np.log2(n_feature)))
        elif max_features == "one":
            n_sample_f = 1
        else:
            print("Error: No Such Max Feature")
            assert(False)

        # random.choices may with repeat
        # random.sample without
        sampled_features = random.sample(list(range(n_feature)), k=n_sample_f)
        for sampled_feature in sampled_features:
            mask[sampled_feature] = 1

        # select feature
        best_gain = 0.0
        best_branch = (None, None)
        cur_func = func(data)
        # cut if impurity is low enough
        if(cur_func < min_impurity_decrease):
            # print("min_impuraity cut"), left element
            self.leaf = True
            return

        for f in range(n_feature):
            if(not mask[f]):
                continue
            #print(data.shape)
            sort by f = data[data[:,f].argsort()]
```

```
            for i in range(n-1):
                th = (sort_by_f[i+1][f] + sort_by_f[i][f])/2
                #print(th)
                cur_gain = abs(func(sort_by_f, f, th) - cur_func)
                if(cur_gain > best_gain):
                    best_branch = (f, th)
                    best_gain = cur_gain

        # cut if not enough gain
        if(best_gain < min_impurity_decrease):
            print("min_impuraity_gain cut")
            self.leaf = True
            return
        else:
            self.leaf = False

        # actually split by best
        self.sf = best_branch[0]
        self.th = best_branch[1]

        c_data = [[], []]
        for d in data:
            c_data[d[best_branch[0]] < best_branch[1]].append(d)
        # print(c_data)
        self.c = [None, None]
        for i in range(2):
            self.c[i] = Node()
            self.c[i].split(np.array(c_data[i]), criterion, max_features, max_depth,
        return


class CART:
    def __init__(self):
        self.rt = Node()
    def train(self, dataset, criterion="gini", max_features=None, max_depth=None, min
        self.rt.split(dataset, criterion, max_features, max_depth, min_impurity_decre
    def predict(self, v):
        cur = self.rt
        while(cur.leaf == False):
            cur = cur.c[v[cur.sf] < cur.th]
        return cur.maj
    def calc_ac(self, testd):
        total = 0.0
        succ = 0.0
        for d in testd:
            succ += int(self.predict(d) == d[LAB])
            total += 1
        return succ/total

class RandomForest:
    def __init__(self, n_estimators=100, criterion="gini", max_features="sqrt", max_d
        self.n_estimators = n_estimators
        self.criterion = criterion
        self.max_features = max_features
        self.max_depth = max_depth
        self.min_impurity_decrease = min_impurity_decrease
        self.max_samples = max_samples
        self.dtrees = []
        self.bootstrap = bootstrap

    def train(self, data):
        n = data.shape[0]
        if self.max_samples == None:
            n_population = n
        elif type(self.max_samples) == float:
```

```
            n_population = int(n*self.max_samples)
        elif type(self.max_samples) == int:
            self.n_polulation = self.max_samples
        else:
            print("MAX SAMPLE TYPE ERROR")
            assert(False)
            return

        for i in range(self.n_estimators):
            ti = CART()
            if self.bootstrap == True:
                sampled_ids = np.random.choice(n, n_population, replace=True)
            else:
                sampled_ids = list(range(n))
            bootstrap_data = data[sampled_ids]
            ti.train(bootstrap_data, criterion=self.criterion, max_features=self.max_
            self.dtrees.append(ti)
        return

    def predict(self, v):
        cnt = np.zeros(n_class)
        for i in range(self.n_estimators):
            cnt[self.dtrees[i].predict(v)] += 1
        return np.argmax(cnt)

    def calc_ac(self, testd):
        total = 0.0
        succ = 0.0
        for d in testd:
            succ += int(self.predict(d) == d[LAB])
            total += 1
        return succ/total
```