

Final Project

Full Name: Yuezhong Yan

NJIT UCID: yy444

Email: yy444@njit.edu

Option 1

Category 3: Decision Tree

Category 5: Naïve Bayes


Programming Language: Python

Dataset: <https://archive.ics.uci.edu/ml/datasets/Iris>

This dataset is about the classification of iris plant based on sepal length, sepal width, petal length, and petal width. It has the total of 150 instances consisting of 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2. The latter are not linearly separable from each other.

It has 5 attributes, sepal length in cm, sepal width in cm, petal length in cm, petal width in cm, and class. The class is split into 3 classifications, Iris Setosa, Iris Versicolor, and Iris Virginica. I use the first 4 attributes to predict which classification a plant belongs to.

To download the dataset, I first click on the URL above, and I see the following Ibsite:



The screenshot shows the UCI Machine Learning Repository website. The header includes the UCI logo, the text "Machine Learning Repository", and the subtitle "Center for Machine Learning and Intelligent Systems". There are links for "About", "Citation Policy", "Donate a Data Set", and "Contact". A search bar is present with a "Search" button. Below the header, the "Iris Data Set" is highlighted. There are links for "Download", "Data Folder", and "Data Set Description". An abstract mentions "Famous database; from Fisher, 1936". A table provides details about the data set characteristics, attributes, and tasks. The source information is listed below the table. The browser's address bar shows the URL "archive.ics.uci.edu/ml/datasets/Iris". The Windows taskbar at the bottom shows the date and time as "2020/4/22 1:27".

UCI
Machine Learning Repository
Center for Machine Learning and Intelligent Systems

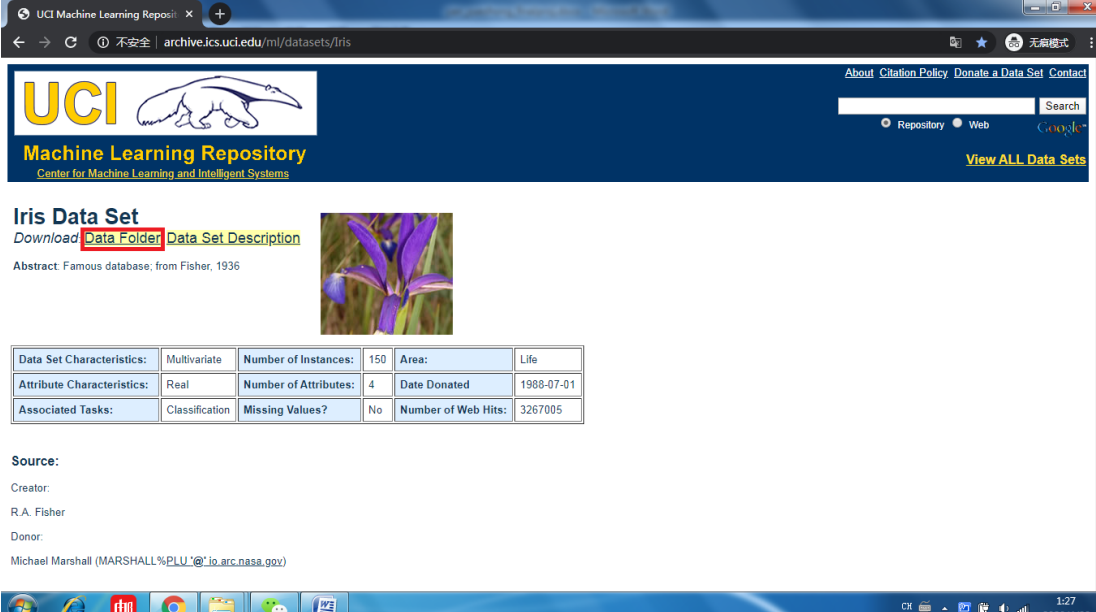
Iris Data Set
Download [Data Folder](#) [Data Set Description](#)

Abstract: Famous database; from Fisher, 1936

Data Set Characteristics:	Multivariate	Number of Instances:	150	Area:	Life
Attribute Characteristics:	Real	Number of Attributes:	4	Date Donated	1988-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	3267005

Source:
Creator:
R.A. Fisher
Donor:
Michael Marshall (MARSHALL%PLU%@*io.arc.nasa.gov)


And then I click on “Data Folder”.



UCI Machine Learning Repository
Center for Machine Learning and Intelligent Systems

Iris Data Set
Download **Data Folder** Data Set Description

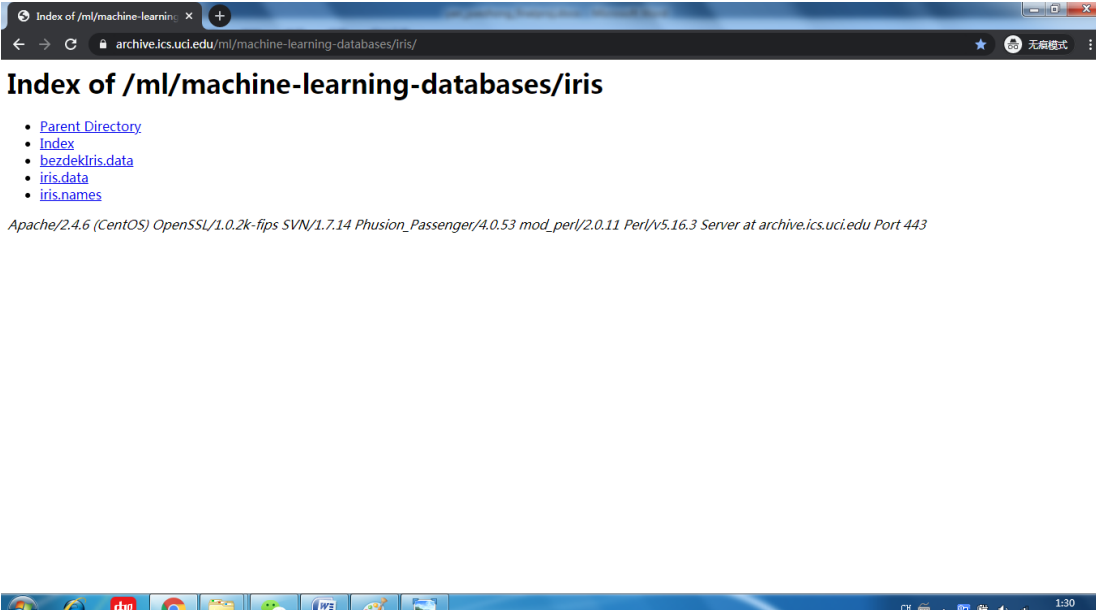
Abstract: Famous database; from Fisher, 1936



Data Set Characteristics:	Multivariate	Number of Instances:	150	Area:	Life
Attribute Characteristics:	Real	Number of Attributes:	4	Date Donated	1988-07-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	3267005

Source:
Creator:
R.A. Fisher
Donor:
Michael Marshall (MARSHALL%PLU%@*io.arc.nasa.gov)

I will enter the Ipage for downloading. In this Ipage, “bezdekIris.data” and “iris.data” have exactly same contents. You can choose either one to download. “Index” is some timestamps about this data. “iris.names” is just some basic information about this dataset. I do not need “Index” or “iris.names” here for our project.



Index of /ml/machine-learning-databases/iris/

- [Parent Directory](#)
- [Index](#)
- [bezdekIris.data](#)
- [iris.data](#)
- [iris.names](#)

Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips SVN/1.7.14 Phusion_Passenger/4.0.53 mod_perl/2.0.11 Perl/v5.16.3 Server at archive.ics.uci.edu Port 443

Here, I click on “iris.data” to download it.



And then, I manually change the extension of the downloaded file into “.csv” to make it easier to read and review in Pycharm.



And finally I got the entire “iris.csv”. To run through this dataset, please put this dataset at the same level as the project.

iris.csv:

Attribute information

1. sepal length in cm
2. sepal width in cm
3. petal length in cm
4. petal width in cm
5. class: Iris Setosa, Iris Versicolor, and Iris Virginica.(Predicted attribute)

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
4.6,3.1,1.5,0.2,Iris-setosa
5.0,3.6,1.4,0.2,Iris-setosa
5.4,3.9,1.7,0.4,Iris-setosa
4.6,3.4,1.4,0.3,Iris-setosa
5.0,3.4,1.5,0.2,Iris-setosa
4.4,2.9,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
5.4,3.7,1.5,0.2,Iris-setosa
```

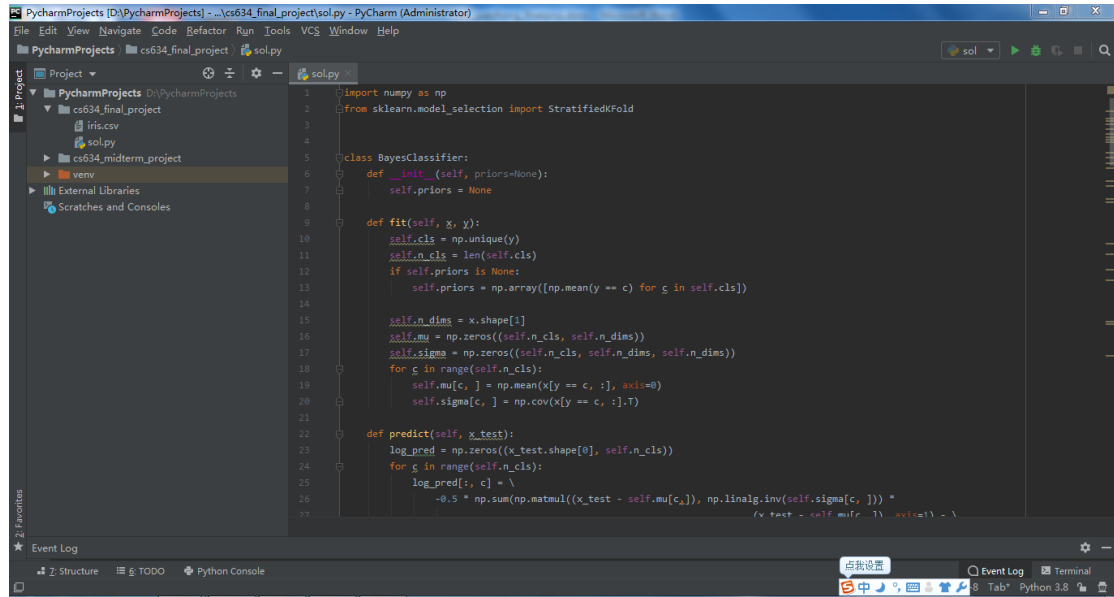
4.8,3.4,1.6,0.2,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
5.4,3.4,1.7,0.2,Iris-setosa
5.1,3.7,1.5,0.4,Iris-setosa
4.6,3.6,1.0,0.2,Iris-setosa
5.1,3.3,1.7,0.5,Iris-setosa
4.8,3.4,1.9,0.2,Iris-setosa
5.0,3.0,1.6,0.2,Iris-setosa
5.0,3.4,1.6,0.4,Iris-setosa
5.2,3.5,1.5,0.2,Iris-setosa
5.2,3.4,1.4,0.2,Iris-setosa
4.7,3.2,1.6,0.2,Iris-setosa
4.8,3.1,1.6,0.2,Iris-setosa
5.4,3.4,1.5,0.4,Iris-setosa
5.2,4.1,1.5,0.1,Iris-setosa
5.5,4.2,1.4,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
5.0,3.2,1.2,0.2,Iris-setosa
5.5,3.5,1.3,0.2,Iris-setosa
4.9,3.1,1.5,0.1,Iris-setosa
4.4,3.0,1.3,0.2,Iris-setosa
5.1,3.4,1.5,0.2,Iris-setosa
5.0,3.5,1.3,0.3,Iris-setosa
4.5,2.3,1.3,0.3,Iris-setosa
4.4,3.2,1.3,0.2,Iris-setosa
5.0,3.5,1.6,0.6,Iris-setosa
5.1,3.8,1.9,0.4,Iris-setosa
4.8,3.0,1.4,0.3,Iris-setosa
5.1,3.8,1.6,0.2,Iris-setosa
4.6,3.2,1.4,0.2,Iris-setosa
5.3,3.7,1.5,0.2,Iris-setosa
5.0,3.3,1.4,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
5.7,2.8,4.5,1.3,Iris-versicolor
6.3,3.3,4.7,1.6,Iris-versicolor
4.9,2.4,3.3,1.0,Iris-versicolor
6.6,2.9,4.6,1.3,Iris-versicolor
5.2,2.7,3.9,1.4,Iris-versicolor
5.0,2.0,3.5,1.0,Iris-versicolor
5.9,3.0,4.2,1.5,Iris-versicolor
6.0,2.2,4.0,1.0,Iris-versicolor
6.1,2.9,4.7,1.4,Iris-versicolor
5.6,2.9,3.6,1.3,Iris-versicolor
6.7,3.1,4.4,1.4,Iris-versicolor
5.6,3.0,4.5,1.5,Iris-versicolor
5.8,2.7,4.1,1.0,Iris-versicolor
6.2,2.2,4.5,1.5,Iris-versicolor
5.6,2.5,3.9,1.1,Iris-versicolor
5.9,3.2,4.8,1.8,Iris-versicolor
6.1,2.8,4.0,1.3,Iris-versicolor

6.3,2.5,4.9,1.5,Iris-versicolor
6.1,2.8,4.7,1.2,Iris-versicolor
6.4,2.9,4.3,1.3,Iris-versicolor
6.6,3.0,4.4,1.4,Iris-versicolor
6.8,2.8,4.8,1.4,Iris-versicolor
6.7,3.0,5.0,1.7,Iris-versicolor
6.0,2.9,4.5,1.5,Iris-versicolor
5.7,2.6,3.5,1.0,Iris-versicolor
5.5,2.4,3.8,1.1,Iris-versicolor
5.5,2.4,3.7,1.0,Iris-versicolor
5.8,2.7,3.9,1.2,Iris-versicolor
6.0,2.7,5.1,1.6,Iris-versicolor
5.4,3.0,4.5,1.5,Iris-versicolor
6.0,3.4,4.5,1.6,Iris-versicolor
6.7,3.1,4.7,1.5,Iris-versicolor
6.3,2.3,4.4,1.3,Iris-versicolor
5.6,3.0,4.1,1.3,Iris-versicolor
5.5,2.5,4.0,1.3,Iris-versicolor
5.5,2.6,4.4,1.2,Iris-versicolor
6.1,3.0,4.6,1.4,Iris-versicolor
5.8,2.6,4.0,1.2,Iris-versicolor
5.0,2.3,3.3,1.0,Iris-versicolor
5.6,2.7,4.2,1.3,Iris-versicolor
5.7,3.0,4.2,1.2,Iris-versicolor
5.7,2.9,4.2,1.3,Iris-versicolor
6.2,2.9,4.3,1.3,Iris-versicolor
5.1,2.5,3.0,1.1,Iris-versicolor
5.7,2.8,4.1,1.3,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
6.3,2.9,5.6,1.8,Iris-virginica
6.5,3.0,5.8,2.2,Iris-virginica
7.6,3.0,6.6,2.1,Iris-virginica
4.9,2.5,4.5,1.7,Iris-virginica
7.3,2.9,6.3,1.8,Iris-virginica
6.7,2.5,5.8,1.8,Iris-virginica
7.2,3.6,6.1,2.5,Iris-virginica
6.5,3.2,5.1,2.0,Iris-virginica
6.4,2.7,5.3,1.9,Iris-virginica
6.8,3.0,5.5,2.1,Iris-virginica
5.7,2.5,5.0,2.0,Iris-virginica
5.8,2.8,5.1,2.4,Iris-virginica
6.4,3.2,5.3,2.3,Iris-virginica
6.5,3.0,5.5,1.8,Iris-virginica
7.7,3.8,6.7,2.2,Iris-virginica
7.7,2.6,6.9,2.3,Iris-virginica
6.0,2.2,5.0,1.5,Iris-virginica
6.9,3.2,5.7,2.3,Iris-virginica
5.6,2.8,4.9,2.0,Iris-virginica
7.7,2.8,6.7,2.0,Iris-virginica
6.3,2.7,4.9,1.8,Iris-virginica
6.7,3.3,5.7,2.1,Iris-virginica
7.2,3.2,6.0,1.8,Iris-virginica
6.2,2.8,4.8,1.8,Iris-virginica
6.1,3.0,4.9,1.8,Iris-virginica
6.4,2.8,5.6,2.1,Iris-virginica
7.2,3.0,5.8,1.6,Iris-virginica
7.4,2.8,6.1,1.9,Iris-virginica
7.9,3.8,6.4,2.0,Iris-virginica
6.4,2.8,5.6,2.2,Iris-virginica

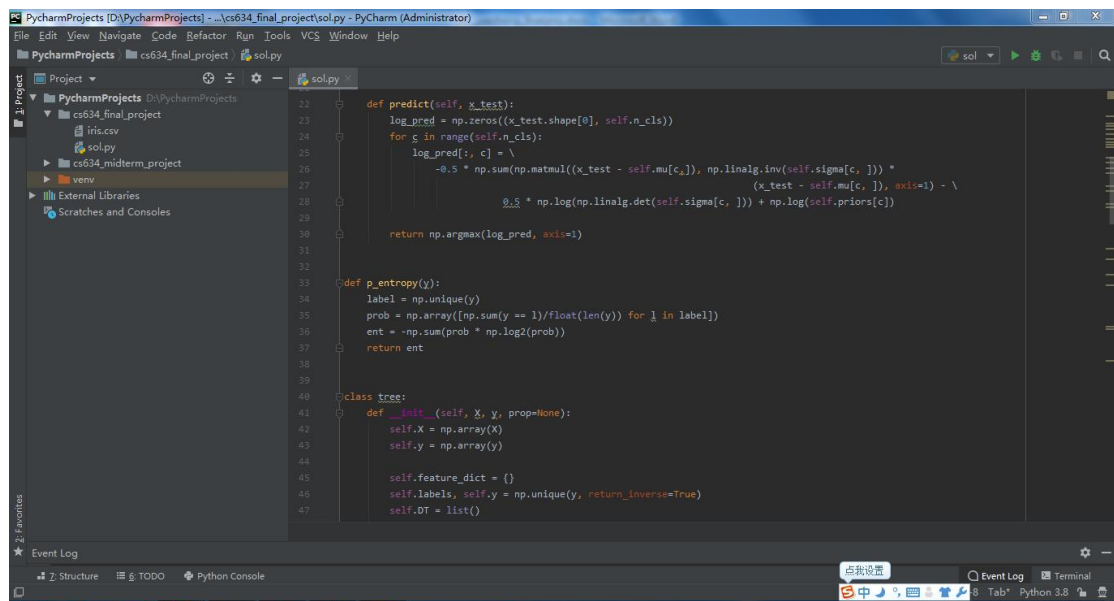
6.3,2.8,5.1,1.5,Iris-virginica
6.1,2.6,5.6,1.4,Iris-virginica
7.7,3.0,6.1,2.3,Iris-virginica
6.3,3.4,5.6,2.4,Iris-virginica
6.4,3.1,5.5,1.8,Iris-virginica
6.0,3.0,4.8,1.8,Iris-virginica
6.9,3.1,5.4,2.1,Iris-virginica
6.7,3.1,5.6,2.4,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
6.7,3.3,5.7,2.5,Iris-virginica
6.7,3.0,5.2,2.3,Iris-virginica
6.3,2.5,5.0,1.9,Iris-virginica
6.5,3.0,5.2,2.0,Iris-virginica
6.2,3.4,5.4,2.3,Iris-virginica
5.9,3.0,5.1,1.8,Iris-virginica

Project

0. This project runs Iris.csv with naïve bayes and decision tree. I will use numpy and StratifiedKFold from sklearn.model_selection. The whole project is shown below.



```
1 import numpy as np
2 from sklearn.model_selection import StratifiedKFold
3
4 class BayesClassifier:
5     def __init__(self, priors=None):
6         self.priors = None
7
8     def fit(self, x, y):
9         self.cls = np.unique(y)
10        self.n_cls = len(self.cls)
11        if self.priors is None:
12            self.priors = np.array([np.mean(y == c) for c in self.cls])
13
14        self.n_dims = x.shape[1]
15        self.mu = np.zeros((self.n_cls, self.n_dims))
16        self.sigma = np.zeros((self.n_cls, self.n_dims, self.n_dims))
17        for c in range(self.n_cls):
18            self.mu[c, :] = np.mean(x[y == c, :], axis=0)
19            self.sigma[c, :, :] = np.cov(x[y == c, :].T)
20
21    def predict(self, x_test):
22        log_pred = np.zeros((x_test.shape[0], self.n_cls))
23        for c in range(self.n_cls):
24            log_pred[:, c] = \
25                -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]), np.linalg.inv(self.sigma[c, :, :])) *
26                             (x_test - self.mu[c, :]).T, axis=1) - \
```



```
27                0.5 * np.log(np.linalg.det(self.sigma[c, :, :])) + np.log(self.priors[c])
28
29        return np.argmax(log_pred, axis=1)
30
31
32
33 def p_entropy(y):
34     label = np.unique(y)
35     prob = np.array([np.sum(y == l)/float(len(y)) for l in label])
36     ent = -np.sum(prob * np.log2(prob))
37     return ent
38
39
40 class tree:
41     def __init__(self, X, y, prop=None):
42         self.X = np.array(X)
43         self.y = np.array(y)
44
45         self.feature_dict = {}
46         self.labels, self.y = np.unique(y, return_inverse=True)
47         self.DT = list()
```



```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
Project cs634_final_project sol.py
class tree:
    def __init__(self, X, y, prop=None):
        self.X = np.array(X)
        self.y = np.array(y)

        self.feature_dict = {}
        self.labels, self.y = np.unique(y, return_inverse=True)
        self.DT = list()
        if prop is None:
            self.property = np.zeros((self.X.shape[1]))
        else:
            self.property = prop

        for i in range(self.X.shape[1]):
            self.feature_dict.setdefault(i)
            self.feature_dict[i] = np.unique(self.X[:, i])

    def entropy(self, X, y, k, k_v):
        if self.property[k] == 0:
            c1 = (X[X[:, k] == k_v]).shape[0]
            c2 = (X[X[:, k] != k_v]).shape[0]
            D = y.shape[0]
            return c1 * p_entropy(y[X[:, k] == k_v] / D \
                                + c2 * p_entropy(y[X[:, k] != k_v] / D)
        else:
            c1 = (X[X[:, k] >= k_v]).shape[0]
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
    def entropy(self, X, y, k, k_v):
        if self.property[k] == 0:
            c1 = (X[X[:, k] == k_v]).shape[0]
            c2 = (X[X[:, k] != k_v]).shape[0]
            D = y.shape[0]
            return c1 * p_entropy(y[X[:, k] == k_v] / D \
                                + c2 * p_entropy(y[X[:, k] != k_v] / D)
        else:
            c1 = (X[X[:, k] >= k_v]).shape[0]
            c2 = (X[X[:, k] < k_v]).shape[0]
            D = y.shape[0]
            return c1 * p_entropy(y[X[:, k] >= k_v] / D \
                                + c2 * p_entropy(y[X[:, k] < k_v] / D)

    def makeTree(self, X, y):
        if np.unique(y).size <= 1:
            return y[0]

        minp = 10000.0
        m_i, m_j = 0, 0
        for i in range(self.X.shape[1]):
            for j in self.feature_dict[i]:
                p = self.entropy(X, y, i, j)
                if p < minp:
                    minp = p
                    m_i, m_j = i, j
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
    def makeTree(self, X, y):
        if np.unique(y).size <= 1:
            return y[0]

        minp = 10000.0
        m_i, m_j = 0, 0
        for i in range(self.X.shape[1]):
            for j in self.feature_dict[i]:
                p = self.entropy(X, y, i, j)
                if p < minp:
                    minp = p
                    m_i, m_j = i, j

        if minp == 1:
            return y[0]

        left = []
        right = []
        if self.property[m_i] == 0:
            left = self.makeTree(X[X[:, m_i] == m_j], y[X[:, m_i] == m_j])
            right = self.makeTree(X[X[:, m_i] != m_j], y[X[:, m_i] != m_j])
        else:
            left = self.makeTree(X[X[:, m_i] >= m_j], y[X[:, m_i] >= m_j])
            right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
        return (m_i, m_j), left, right

    def train(self):
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
Project cs634_final_project
  cs634_final_project
  iris.csv
  sol.py
  cs634_midterm_project
  venv
External Libraries
Scratches and Consoles
Event Log
def train(self):
    self.DT = self.makeTree(self.X, self.y)

def predict(self, X):
    result = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        tp = self.DT
        while type(tp) is tuple:
            a, b = tp[0]

            if self.property[a] == 0:
                if X[i][a] == b:
                    tp = tp[1]
                else:
                    tp = tp[2]
            else:
                if X[i][a] >= b:
                    tp = tp[1]
                else:
                    tp = tp[2]
            result[i] = self.labels[tp]
    return result

x = []
y = []
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
Project cs634_final_project
  cs634_final_project
  iris.csv
  sol.py
  cs634_midterm_project
  venv
External Libraries
Scratches and Consoles
Run x
x = np.array(x)
c = np.unique(y)
c = dict([(l, c, i) for i, _c in enumerate(c)])
y = [c[y] for y in y]
y = np.array(y)

acc = 0.
for train_index, test_index in StratifiedFold(n_splits=10).split(x, y):
    x_train, x_test = x[train_index, :], x[test_index, :]
    y_train, y_test = y[train_index], y[test_index]
    model = tree(x_train, y_train)
    model.train()
    z = model.predict(x_test)
    acc += np.mean(z == y_test)
acc /= 10
print('10 fold average accuracy (decision tree) = %.4f' % acc)

acc = 0.
for train_index, test_index in StratifiedFold(n_splits=10).split(x, y):
    x_train, x_test = x[train_index, :], x[test_index, :]
    y_train, y_test = y[train_index], y[test_index]
    model = BayesClassifier()
    model.fit(x_train, y_train)
    z = model.predict(x_test)
    acc += np.mean(z == y_test)
acc /= 10
for train_index, test_index in ...
```

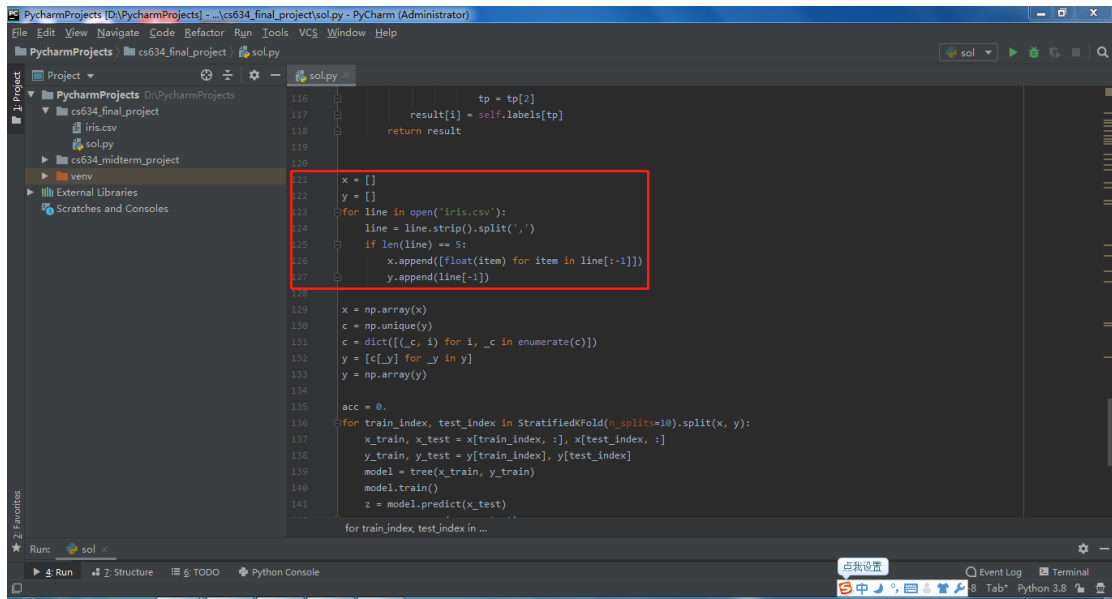
```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
Project cs634_final_project
  cs634_final_project
  iris.csv
  sol.py
  cs634_midterm_project
  venv
External Libraries
Scratches and Consoles
Run x
c = dict([(l, c, i) for i, _c in enumerate(c)])
y = [c[y] for y in y]
y = np.array(y)

acc = 0.
for train_index, test_index in StratifiedFold(n_splits=10).split(x, y):
    x_train, x_test = x[train_index, :], x[test_index, :]
    y_train, y_test = y[train_index], y[test_index]
    model = tree(x_train, y_train)
    model.train()
    z = model.predict(x_test)
    acc += np.mean(z == y_test)
acc /= 10
print('10 fold average accuracy (decision tree) = %.4f' % acc)

acc = 0.
for train_index, test_index in StratifiedFold(n_splits=10).split(x, y):
    x_train, x_test = x[train_index, :], x[test_index, :]
    y_train, y_test = y[train_index], y[test_index]
    model = BayesClassifier()
    model.fit(x_train, y_train)
    z = model.predict(x_test)
    acc += np.mean(z == y_test)
acc /= 10
print('10 fold average accuracy (naive bayes) = %.4f' % acc)

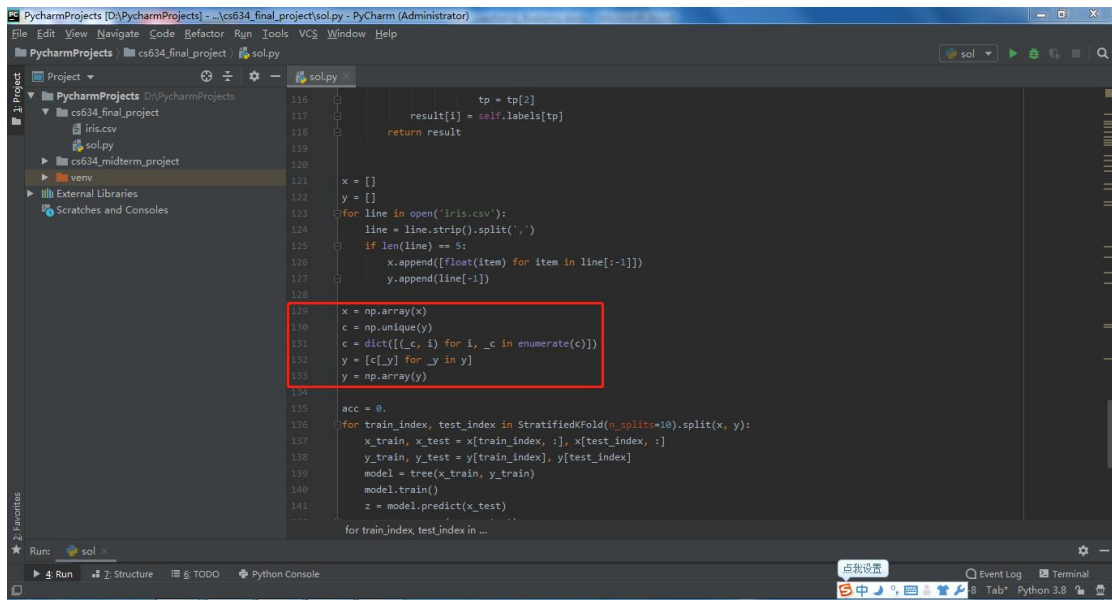
for train_index, test_index in ...
```

1. I first read Iris.csv and split the first 4 attributes and the last one into list variables x and y, respectively. x stores sepal length in cm, sepal width in cm, petal length in cm, and petal width in cm. y stores the 3 classifications Iris Setosa, Iris Versicolor, and Iris Virginica, each having 50 instances.



```
116         tp = tp[2]
117         result[i] = self.labels[tp]
118         return result
119
120
121     x = []
122     y = []
123     for line in open('iris.csv'):
124         line = line.strip().split(',')
125         if len(line) == 5:
126             x.append([float(item) for item in line[:-1]])
127             y.append(line[-1])
128
129     x = np.array(x)
130     c = np.unique(y)
131     c = dict([(c, i) for i, _c in enumerate(c)])
132     y = [c[_y] for _y in y]
133     y = np.array(y)
134
135     acc = 0.
136     for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
137         x_train, x_test = x[train_index, :], x[test_index, :]
138         y_train, y_test = y[train_index], y[test_index]
139         model = tree(x_train, y_train)
140         model.train()
141         z = model.predict(x_test)
142         for train_index, test_index in ...
```

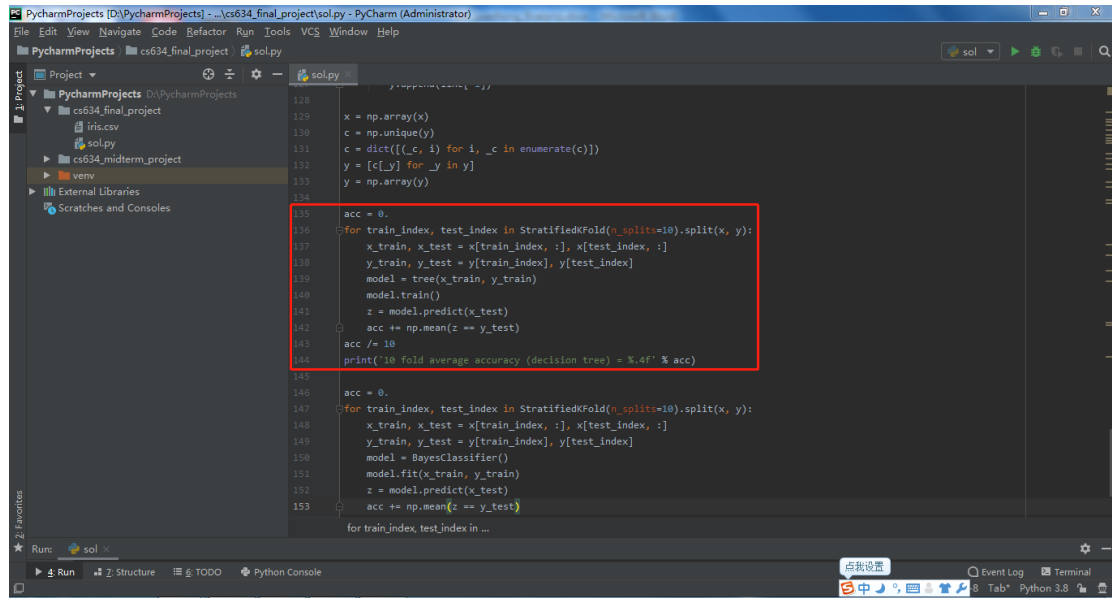
2. Then I put these two variables into another data type ndarray, where x is 2d array and y is 1d array. I put x directly into ndarray, remove the duplicates from y, replace the 3 classifications Iris Setosa, Iris Versicolor, and Iris Virginica from y with 0, 1, 2, respectively.



```
129     x = np.array(x)
130     c = np.unique(y)
131     c = dict([(c, i) for i, _c in enumerate(c)])
132     y = [c[_y] for _y in y]
133     y = np.array(y)
134
135     acc = 0.
136     for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
137         x_train, x_test = x[train_index, :], x[test_index, :]
138         y_train, y_test = y[train_index], y[test_index]
139         model = tree(x_train, y_train)
140         model.train()
141         z = model.predict(x_test)
142         for train_index, test_index in ...
```

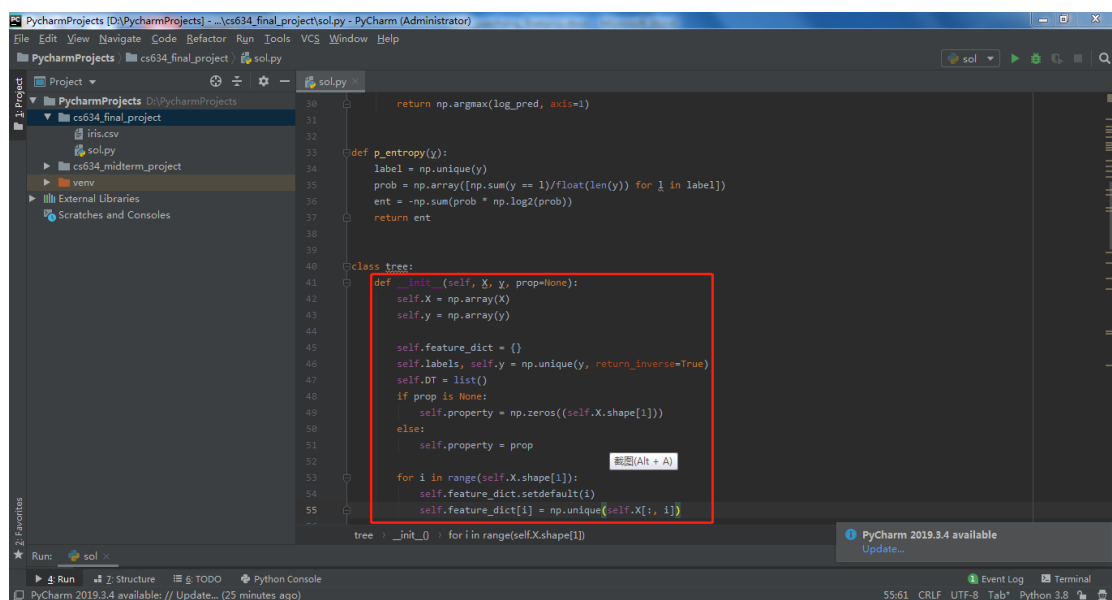
Decision Tree:

1. I use StratifiedKFold to split the data into 10 subsets to perform 10-fold cross validation. I then declare and initialize training data `x_train` and `y_train`, as well as testing data `x_test` and `y_test`, construct `Tree()` object with `x_train` and `y_train`, train the model, predict `x_test` to predict `y`, and finally calculate how accurate the prediction is by taking the average of accuracies of all ten runs. This average is treated as the accuracy of the evaluated classifier.



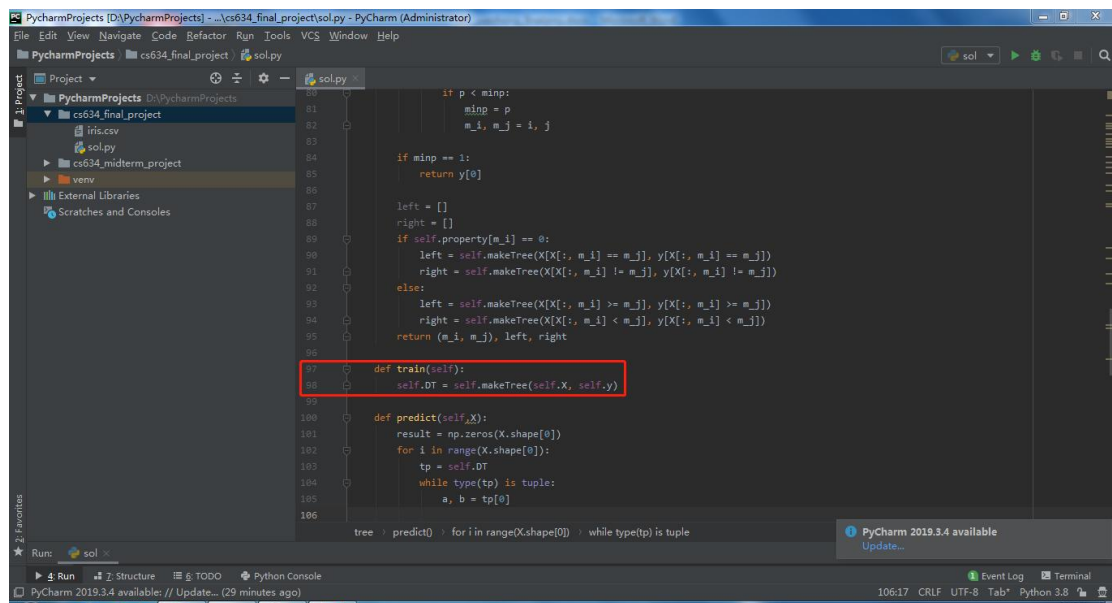
```
128 x = np.array(x)
129 c = np.unique(y)
130 c = dict([(c, i) for i, _c in enumerate(c)])
131 y = [c[_y] for _y in y]
132 y = np.array(y)
133
134 acc = 0.
135 for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
136     x_train, x_test = x[train_index, :], x[test_index, :]
137     y_train, y_test = y[train_index], y[test_index]
138     model = tree(x_train, y_train)
139     model.train()
140     z = model.predict(x_test)
141     acc += np.mean(z == y_test)
142 acc /= 10
143 print('10 fold average accuracy (decision tree) = %.4f' % acc)
144
145 acc = 0.
146 for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
147     x_train, x_test = x[train_index, :], x[test_index, :]
148     y_train, y_test = y[train_index], y[test_index]
149     model = BayesClassifier()
150     model.fit(x_train, y_train)
151     z = model.predict(x_test)
152     acc += np.mean(z == y_test)
153 for train_index, test_index in ...
```

2. `Tree().__init__()` is a constructor for Decision Tree object. `X` creates a 2d array and stores the first 4 attributes of the original dataset. `y` also creates a 2d array and stores the last attribute(classification) of the original dataset. `feature_dict` stores feature values for each column from `X`. `labels` stores the labels for the last attribute(classification) with numbers 0, 1, and 2, with each number representing a classification. `DT` stores a decision tree with a List.



```
38 return np.argmax(log_pred, axis=1)
39
40 def p_entropy(y):
41     label = np.unique(y)
42     prob = np.array([(np.sum(y == l)/float(len(y)) for l in label)])
43     ent = -np.sum(prob * np.log2(prob))
44     return ent
45
46 class tree:
47     def __init__(self, X, y, prop=None):
48         self.X = np.array(X)
49         self.y = np.array(y)
50
51         self.feature_dict = {}
52         self.labels, self.y = np.unique(y, return_inverse=True)
53         self.DT = list()
54         if prop is None:
55             self.property = np.zeros((self.X.shape[1]))
56         else:
57             self.property = prop
58
59         for i in range(self.X.shape[1]):
60             self.feature_dict.setdefault(i)
61             self.feature_dict[i] = np.unique(self.X[:, i])
62
63     tree.__init__ for i in range(self.X.shape[1])
```

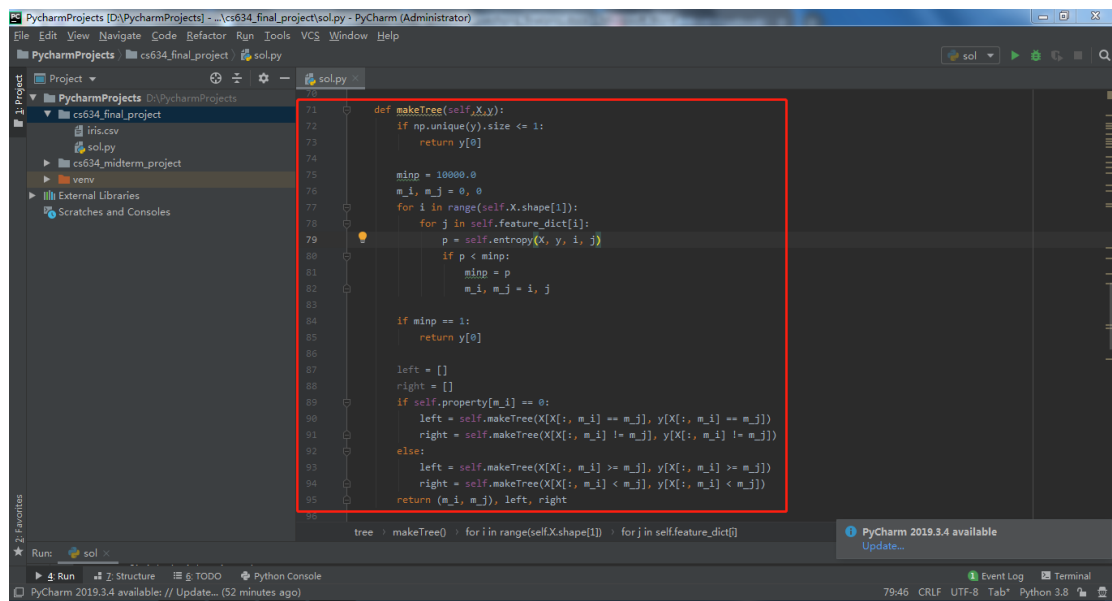
3. `train()` trains the data by making a decision tree with `makeTree()` method.



A screenshot of the PyCharm IDE showing a Python file named `sol.py`. The code defines a `DecisionTree` class. The `train` method (lines 97-106) is highlighted with a red box. It calls `self.makeTree(self.X, self.y)` to build the tree and then uses `self.DT` to predict the class for each instance in `X`.

```
80         if p < minp:
81             minp = p
82             m_i, m_j = i, j
83
84         if minp == 1:
85             return y[0]
86
87         left = []
88         right = []
89         if self.property[m_i] == 0:
90             left = self.makeTree(X[X[:, m_i] == m_j], y[X[:, m_i] == m_j])
91             right = self.makeTree(X[X[:, m_i] != m_j], y[X[:, m_i] != m_j])
92         else:
93             left = self.makeTree(X[X[:, m_i] >= m_j], y[X[:, m_i] >= m_j])
94             right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
95         return (m_i, m_j), left, right
96
97     def train(self):
98         self.DT = self.makeTree(self.X, self.y)
99
100     def predict(self, X):
101         result = np.zeros(X.shape[0])
102         for i in range(X.shape[0]):
103             tp = self.DT
104             while type(tp) is tuple:
105                 a, b = tp[0]
106                 result[i] = b
```

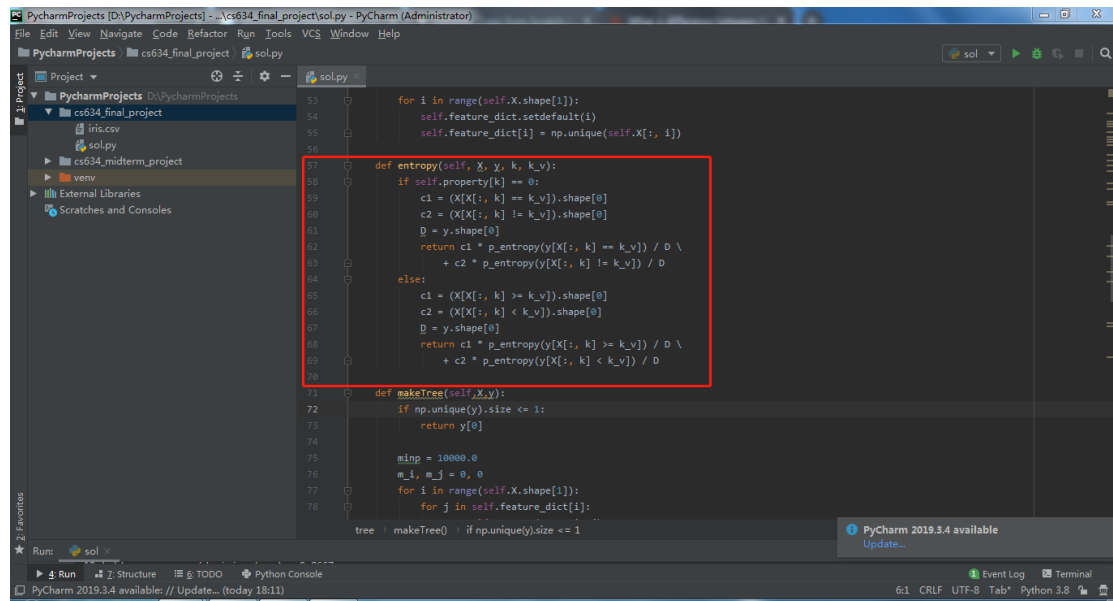
4. In `makeTree()` method, I calculate the information gain, or the reduction in entropy, by keeping track of the 4 attributes from `X` and feature values for each attribute from `feature_dict`, so that I can get the best information gain. Here, `m_i` stores the labels for the last attribute (classification) with numbers 0, 1, and 2, while `m_j` stores the feature value for each label. Then I split the tree with recursion. Finally this function returns `(m_i, m_j)`, left sub-tree, and right sub-tree.



A screenshot of the PyCharm IDE showing the `makeTree` method in `sol.py`. The method (lines 71-95) is highlighted with a red box. It calculates the information gain for each attribute and feature value, then recursively builds the left and right sub-trees.

```
71     def makeTree(self, X, y):
72         if np.unique(y).size <= 1:
73             return y[0]
74
75         minp = 10000.0
76         m_i, m_j = 0, 0
77         for i in range(self.X.shape[1]):
78             for j in self.feature_dict[i]:
79                 p = self.entropy(X, y, i, j)
80                 if p < minp:
81                     minp = p
82                     m_i, m_j = i, j
83
84         if minp == 1:
85             return y[0]
86
87         left = []
88         right = []
89         if self.property[m_i] == 0:
90             left = self.makeTree(X[X[:, m_i] == m_j], y[X[:, m_i] == m_j])
91             right = self.makeTree(X[X[:, m_i] != m_j], y[X[:, m_i] != m_j])
92         else:
93             left = self.makeTree(X[X[:, m_i] >= m_j], y[X[:, m_i] >= m_j])
94             right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
95         return (m_i, m_j), left, right
```

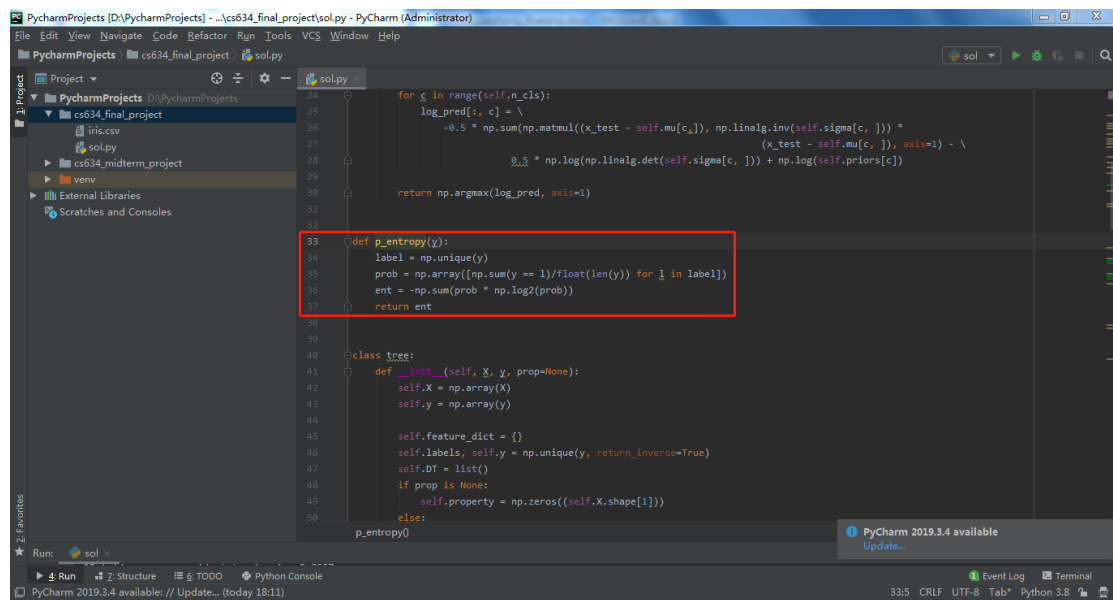
5. `entropy()` at line 79 from `makingTree()` calculates entropies for parent, left-child node, and right-child node with the label for classification and the feature value for each label. Its implementation is shown below.



```

53     for i in range(self.X.shape[1]):
54         self.feature_dict.setdefault(i)
55         self.feature_dict[i] = np.unique(self.X[:, i])
56
57     def entropy(self, X, y, k, k_v):
58         if self.property[k] == 0:
59             c1 = (X[X[:, k] == k_v]).shape[0]
60             c2 = (X[X[:, k] != k_v]).shape[0]
61             D = y.shape[0]
62             return c1 * p_entropy(y[X[:, k] == k_v]) / D \
63                 + c2 * p_entropy(y[X[:, k] != k_v]) / D
64         else:
65             c1 = (X[X[:, k] >= k_v]).shape[0]
66             c2 = (X[X[:, k] < k_v]).shape[0]
67             D = y.shape[0]
68             return c1 * p_entropy(y[X[:, k] >= k_v]) / D \
69                 + c2 * p_entropy(y[X[:, k] < k_v]) / D
70
71     def makeTree(self, X, y):
72         if np.unique(y).size <= 1:
73             return y[0]
74
75         minp = 10000.0
76         m_i, m_j = 0, 0
77         for i in range(self.X.shape[1]):
78             for j in self.feature_dict[i]:
79                 tree = makeTree0(X, y, i, j)
80                 if np.unique(y).size <= 1:
81                     return y[0]
82
83         return y[0]
84
85     def makeTree0(self, X, y, i, j):
86         tree = makeTree(X, y)
87         return tree
88
89     def makeTree1(self, X, y, i, j):
90         tree = makeTree(X, y)
91         return tree
92
93     def makeTree2(self, X, y, i, j):
94         tree = makeTree(X, y)
95         return tree
96
97     def makeTree3(self, X, y, i, j):
98         tree = makeTree(X, y)
99         return tree
100
101     def makeTree4(self, X, y, i, j):
102         tree = makeTree(X, y)
103         return tree
104
105     def makeTree5(self, X, y, i, j):
106         tree = makeTree(X, y)
107         return tree
108
109     def makeTree6(self, X, y, i, j):
110         tree = makeTree(X, y)
111         return tree
112
113     def makeTree7(self, X, y, i, j):
114         tree = makeTree(X, y)
115         return tree
116
117     def makeTree8(self, X, y, i, j):
118         tree = makeTree(X, y)
119         return tree
120
121     def makeTree9(self, X, y, i, j):
122         tree = makeTree(X, y)
123         return tree
124
125     def makeTree10(self, X, y, i, j):
126         tree = makeTree(X, y)
127         return tree
128
129     def makeTree11(self, X, y, i, j):
130         tree = makeTree(X, y)
131         return tree
132
133     def makeTree12(self, X, y, i, j):
134         tree = makeTree(X, y)
135         return tree
136
137     def makeTree13(self, X, y, i, j):
138         tree = makeTree(X, y)
139         return tree
140
141     def makeTree14(self, X, y, i, j):
142         tree = makeTree(X, y)
143         return tree
144
145     def makeTree15(self, X, y, i, j):
146         tree = makeTree(X, y)
147         return tree
148
149     def makeTree16(self, X, y, i, j):
150         tree = makeTree(X, y)
151         return tree
152
153     def makeTree17(self, X, y, i, j):
154         tree = makeTree(X, y)
155         return tree
156
157     def makeTree18(self, X, y, i, j):
158         tree = makeTree(X, y)
159         return tree
160
161     def makeTree19(self, X, y, i, j):
162         tree = makeTree(X, y)
163         return tree
164
165     def makeTree20(self, X, y, i, j):
166         tree = makeTree(X, y)
167         return tree
168
169     def makeTree21(self, X, y, i, j):
170         tree = makeTree(X, y)
171         return tree
172
173     def makeTree22(self, X, y, i, j):
174         tree = makeTree(X, y)
175         return tree
176
177     def makeTree23(self, X, y, i, j):
178         tree = makeTree(X, y)
179         return tree
180
181     def makeTree24(self, X, y, i, j):
182         tree = makeTree(X, y)
183         return tree
184
185     def makeTree25(self, X, y, i, j):
186         tree = makeTree(X, y)
187         return tree
188
189     def makeTree26(self, X, y, i, j):
190         tree = makeTree(X, y)
191         return tree
192
193     def makeTree27(self, X, y, i, j):
194         tree = makeTree(X, y)
195         return tree
196
197     def makeTree28(self, X, y, i, j):
198         tree = makeTree(X, y)
199         return tree
200
201     def makeTree29(self, X, y, i, j):
202         tree = makeTree(X, y)
203         return tree
204
205     def makeTree30(self, X, y, i, j):
206         tree = makeTree(X, y)
207         return tree
208
209     def makeTree31(self, X, y, i, j):
210         tree = makeTree(X, y)
211         return tree
212
213     def makeTree32(self, X, y, i, j):
214         tree = makeTree(X, y)
215         return tree
216
217     def makeTree33(self, X, y, i, j):
218         tree = makeTree(X, y)
219         return tree
220
221     def makeTree34(self, X, y, i, j):
222         tree = makeTree(X, y)
223         return tree
224
225     def makeTree35(self, X, y, i, j):
226         tree = makeTree(X, y)
227         return tree
228
229     def makeTree36(self, X, y, i, j):
230         tree = makeTree(X, y)
231         return tree
232
233     def makeTree37(self, X, y, i, j):
234         tree = makeTree(X, y)
235         return tree
236
237     def makeTree38(self, X, y, i, j):
238         tree = makeTree(X, y)
239         return tree
240
241     def makeTree39(self, X, y, i, j):
242         tree = makeTree(X, y)
243         return tree
244
245     def makeTree40(self, X, y, i, j):
246         tree = makeTree(X, y)
247         return tree
248
249     def makeTree41(self, X, y, i, j):
250         tree = makeTree(X, y)
251         return tree
252
253     def makeTree42(self, X, y, i, j):
254         tree = makeTree(X, y)
255         return tree
256
257     def makeTree43(self, X, y, i, j):
258         tree = makeTree(X, y)
259         return tree
260
261     def makeTree44(self, X, y, i, j):
262         tree = makeTree(X, y)
263         return tree
264
265     def makeTree45(self, X, y, i, j):
266         tree = makeTree(X, y)
267         return tree
268
269     def makeTree46(self, X, y, i, j):
270         tree = makeTree(X, y)
271         return tree
272
273     def makeTree47(self, X, y, i, j):
274         tree = makeTree(X, y)
275         return tree
276
277     def makeTree48(self, X, y, i, j):
278         tree = makeTree(X, y)
279         return tree
280
281     def makeTree49(self, X, y, i, j):
282         tree = makeTree(X, y)
283         return tree
284
285     def makeTree50(self, X, y, i, j):
286         tree = makeTree(X, y)
287         return tree
288
289     def makeTree51(self, X, y, i, j):
290         tree = makeTree(X, y)
291         return tree
292
293     def makeTree52(self, X, y, i, j):
294         tree = makeTree(X, y)
295         return tree
296
297     def makeTree53(self, X, y, i, j):
298         tree = makeTree(X, y)
299         return tree
300
301     def makeTree54(self, X, y, i, j):
302         tree = makeTree(X, y)
303         return tree
304
305     def makeTree55(self, X, y, i, j):
306         tree = makeTree(X, y)
307         return tree
308
309     def makeTree56(self, X, y, i, j):
310         tree = makeTree(X, y)
311         return tree
312
313     def makeTree57(self, X, y, i, j):
314         tree = makeTree(X, y)
315         return tree
316
317     def makeTree58(self, X, y, i, j):
318         tree = makeTree(X, y)
319         return tree
320
321     def makeTree59(self, X, y, i, j):
322         tree = makeTree(X, y)
323         return tree
324
325     def makeTree60(self, X, y, i, j):
326         tree = makeTree(X, y)
327         return tree
328
329     def makeTree61(self, X, y, i, j):
330         tree = makeTree(X, y)
331         return tree
332
333     def makeTree62(self, X, y, i, j):
334         tree = makeTree(X, y)
335         return tree
336
337     def makeTree63(self, X, y, i, j):
338         tree = makeTree(X, y)
339         return tree
340
341     def makeTree64(self, X, y, i, j):
342         tree = makeTree(X, y)
343         return tree
344
345     def makeTree65(self, X, y, i, j):
346         tree = makeTree(X, y)
347         return tree
348
349     def makeTree66(self, X, y, i, j):
350         tree = makeTree(X, y)
351         return tree
352
353     def makeTree67(self, X, y, i, j):
354         tree = makeTree(X, y)
355         return tree
356
357     def makeTree68(self, X, y, i, j):
358         tree = makeTree(X, y)
359         return tree
360
361     def makeTree69(self, X, y, i, j):
362         tree = makeTree(X, y)
363         return tree
364
365     def makeTree70(self, X, y, i, j):
366         tree = makeTree(X, y)
367         return tree
368
369     def makeTree71(self, X, y, i, j):
370         tree = makeTree(X, y)
371         return tree
372
373     def makeTree72(self, X, y, i, j):
374         tree = makeTree(X, y)
375         return tree
376
377     def makeTree73(self, X, y, i, j):
378         tree = makeTree(X, y)
379         return tree
380
381     def makeTree74(self, X, y, i, j):
382         tree = makeTree(X, y)
383         return tree
384
385     def makeTree75(self, X, y, i, j):
386         tree = makeTree(X, y)
387         return tree
388
389     def makeTree76(self, X, y, i, j):
390         tree = makeTree(X, y)
391         return tree
392
393     def makeTree77(self, X, y, i, j):
394         tree = makeTree(X, y)
395         return tree
396
397     def makeTree78(self, X, y, i, j):
398         tree = makeTree(X, y)
399         return tree
400
401     def makeTree79(self, X, y, i, j):
402         tree = makeTree(X, y)
403         return tree
404
405     def makeTree80(self, X, y, i, j):
406         tree = makeTree(X, y)
407         return tree
408
409     def makeTree81(self, X, y, i, j):
410         tree = makeTree(X, y)
411         return tree
412
413     def makeTree82(self, X, y, i, j):
414         tree = makeTree(X, y)
415         return tree
416
417     def makeTree83(self, X, y, i, j):
418         tree = makeTree(X, y)
419         return tree
420
421     def makeTree84(self, X, y, i, j):
422         tree = makeTree(X, y)
423         return tree
424
425     def makeTree85(self, X, y, i, j):
426         tree = makeTree(X, y)
427         return tree
428
429     def makeTree86(self, X, y, i, j):
430         tree = makeTree(X, y)
431         return tree
432
433     def makeTree87(self, X, y, i, j):
434         tree = makeTree(X, y)
435         return tree
436
437     def makeTree88(self, X, y, i, j):
438         tree = makeTree(X, y)
439         return tree
440
441     def makeTree89(self, X, y, i, j):
442         tree = makeTree(X, y)
443         return tree
444
445     def makeTree90(self, X, y, i, j):
446         tree = makeTree(X, y)
447         return tree
448
449     def makeTree91(self, X, y, i, j):
450         tree = makeTree(X, y)
451         return tree
452
453     def makeTree92(self, X, y, i, j):
454         tree = makeTree(X, y)
455         return tree
456
457     def makeTree93(self, X, y, i, j):
458         tree = makeTree(X, y)
459         return tree
460
461     def makeTree94(self, X, y, i, j):
462         tree = makeTree(X, y)
463         return tree
464
465     def makeTree95(self, X, y, i, j):
466         tree = makeTree(X, y)
467         return tree
468
469     def makeTree96(self, X, y, i, j):
470         tree = makeTree(X, y)
471         return tree
472
473     def makeTree97(self, X, y, i, j):
474         tree = makeTree(X, y)
475         return tree
476
477     def makeTree98(self, X, y, i, j):
478         tree = makeTree(X, y)
479         return tree
480
481     def makeTree99(self, X, y, i, j):
482         tree = makeTree(X, y)
483         return tree
484
485     def makeTree100(self, X, y, i, j):
486         tree = makeTree(X, y)
487         return tree
488
489     def makeTree101(self, X, y, i, j):
490         tree = makeTree(X, y)
491         return tree
492
493     def makeTree102(self, X, y, i, j):
494         tree = makeTree(X, y)
495         return tree
496
497     def makeTree103(self, X, y, i, j):
498         tree = makeTree(X, y)
499         return tree
500
501     def makeTree104(self, X, y, i, j):
502         tree = makeTree(X, y)
503         return tree
504
505     def makeTree105(self, X, y, i, j):
506         tree = makeTree(X, y)
507         return tree
508
509     def makeTree106(self, X, y, i, j):
510         tree = makeTree(X, y)
511         return tree
512
513     def makeTree107(self, X, y, i, j):
514         tree = makeTree(X, y)
515         return tree
516
517     def makeTree108(self, X, y, i, j):
518         tree = makeTree(X, y)
519         return tree
520
521     def makeTree109(self, X, y, i, j):
522         tree = makeTree(X, y)
523         return tree
524
525     def makeTree110(self, X, y, i, j):
526         tree = makeTree(X, y)
527         return tree
528
529     def makeTree111(self, X, y, i, j):
530         tree = makeTree(X, y)
531         return tree
532
533     def makeTree112(self, X, y, i, j):
534         tree = makeTree(X, y)
535         return tree
536
537     def makeTree113(self, X, y, i, j):
538         tree = makeTree(X, y)
539         return tree
540
541     def makeTree114(self, X, y, i, j):
542         tree = makeTree(X, y)
543         return tree
544
545     def makeTree115(self, X, y, i, j):
546         tree = makeTree(X, y)
547         return tree
548
549     def makeTree116(self, X, y, i, j):
550         tree = makeTree(X, y)
551         return tree
552
553     def makeTree117(self, X, y, i, j):
554         tree = makeTree(X, y)
555         return tree
556
557     def makeTree118(self, X, y, i, j):
558         tree = makeTree(X, y)
559         return tree
560
561     def makeTree119(self, X, y, i, j):
562         tree = makeTree(X, y)
563         return tree
564
565     def makeTree120(self, X, y, i, j):
566         tree = makeTree(X, y)
567         return tree
568
569     def makeTree121(self, X, y, i, j):
570         tree = makeTree(X, y)
571         return tree
572
573     def makeTree122(self, X, y, i, j):
574         tree = makeTree(X, y)
575         return tree
576
577     def makeTree123(self, X, y, i, j):
578         tree = makeTree(X, y)
579         return tree
580
581     def makeTree124(self, X, y, i, j):
582         tree = makeTree(X, y)
583         return tree
584
585     def makeTree125(self, X, y, i, j):
586         tree = makeTree(X, y)
587         return tree
588
589     def makeTree126(self, X, y, i, j):
590         tree = makeTree(X, y)
591         return tree
592
593     def makeTree127(self, X, y, i, j):
594         tree = makeTree(X, y)
595         return tree
596
597     def makeTree128(self, X, y, i, j):
598         tree = makeTree(X, y)
599         return tree
600
601     def makeTree129(self, X, y, i, j):
602         tree = makeTree(X, y)
603         return tree
604
605     def makeTree130(self, X, y, i, j):
606         tree = makeTree(X, y)
607         return tree
608
609     def makeTree131(self, X, y, i, j):
610         tree = makeTree(X, y)
611         return tree
612
613     def makeTree132(self, X, y, i, j):
614         tree = makeTree(X, y)
615         return tree
616
617     def makeTree133(self, X, y, i, j):
618         tree = makeTree(X, y)
619         return tree
620
621     def makeTree134(self, X, y, i, j):
622         tree = makeTree(X, y)
623         return tree
624
625     def makeTree135(self, X, y, i, j):
626         tree = makeTree(X, y)
627         return tree
628
629     def makeTree136(self, X, y, i, j):
630         tree = makeTree(X, y)
631         return tree
632
633     def makeTree137(self, X, y, i, j):
634         tree = makeTree(X, y)
635         return tree
636
637     def makeTree138(self, X, y, i, j):
638         tree = makeTree(X, y)
639         return tree
640
641     def makeTree139(self, X, y, i, j):
642         tree = makeTree(X, y)
643         return tree
644
645     def makeTree140(self, X, y, i, j):
646         tree = makeTree(X, y)
647         return tree
648
649     def makeTree141(self, X, y, i, j):
650         tree = makeTree(X, y)
651         return tree
652
653     def makeTree142(self, X, y, i, j):
654         tree = makeTree(X, y)
655         return tree
656
657     def makeTree143(self, X, y, i, j):
658         tree = makeTree(X, y)
659         return tree
660
661     def makeTree144(self, X, y, i, j):
662         tree = makeTree(X, y)
663         return tree
664
665     def makeTree145(self, X, y, i, j):
666         tree = makeTree(X, y)
667         return tree
668
669     def makeTree146(self, X, y, i, j):
670         tree = makeTree(X, y)
671         return tree
672
673     def makeTree147(self, X, y, i, j):
674         tree = makeTree(X, y)
675         return tree
676
677     def makeTree148(self, X, y, i, j):
678         tree = makeTree(X, y)
679         return tree
680
681     def makeTree149(self, X, y, i, j):
682         tree = makeTree(X, y)
683         return tree
684
685     def makeTree150(self, X, y, i, j):
686         tree = makeTree(X, y)
687         return tree
688
689     def makeTree151(self, X, y, i, j):
690         tree = makeTree(X, y)
691         return tree
692
693     def makeTree152(self, X, y, i, j):
694         tree = makeTree(X, y)
695         return tree
696
697     def makeTree153(self, X, y, i, j):
698         tree = makeTree(X, y)
699         return tree
700
701     def makeTree154(self, X, y, i, j):
702         tree = makeTree(X, y)
703         return tree
704
705     def makeTree155(self, X, y, i, j):
706         tree = makeTree(X, y)
707         return tree
708
709     def makeTree156(self, X, y, i, j):
710         tree = makeTree(X, y)
711         return tree
712
713     def makeTree157(self, X, y, i, j):
714         tree = makeTree(X, y)
715         return tree
716
717     def makeTree158(self, X, y, i, j):
718         tree = makeTree(X, y)
719         return tree
720
721     def makeTree159(self, X, y, i, j):
722         tree = makeTree(X, y)
723         return tree
724
725     def makeTree160(self, X, y, i, j):
726         tree = makeTree(X, y)
727         return tree
728
729     def makeTree161(self, X, y, i, j):
730         tree = makeTree(X, y)
731         return tree
732
733     def makeTree162(self, X, y, i, j):
734         tree = makeTree(X, y)
735         return tree
736
737     def makeTree163(self, X, y, i, j):
738         tree = makeTree(X, y)
739         return tree
740
741     def makeTree164(self, X, y, i, j):
742         tree = makeTree(X, y)
743         return tree
744
745     def makeTree165(self, X, y, i, j):
746         tree = makeTree(X, y)
747         return tree
748
749     def makeTree166(self, X, y, i, j):
750         tree = makeTree(X, y)
751         return tree
752
753     def makeTree167(self, X, y, i, j):
754         tree = makeTree(X, y)
755         return tree
756
757     def makeTree168(self, X, y, i, j):
758         tree = makeTree(X, y)
759         return tree
760
761     def makeTree169(self, X, y, i, j):
762         tree = makeTree(X, y)
763         return tree
764
765     def makeTree170(self, X, y, i, j):
766         tree = makeTree(X, y)
767         return tree
768
769     def makeTree171(self, X, y, i, j):
770         tree = makeTree(X, y)
771         return tree
772
773     def makeTree172(self, X, y, i, j):
774         tree = makeTree(X, y)
775         return tree
776
777     def makeTree173(self, X, y, i, j):
778         tree = makeTree(X, y)
779         return tree
780
781     def makeTree174(self, X, y, i, j):
782         tree = makeTree(X, y)
783         return tree
784
785     def makeTree175(self, X, y, i, j):
786         tree = makeTree(X, y)
787         return tree
788
789     def makeTree176(self, X, y, i, j):
790         tree = makeTree(X, y)
791         return tree
792
793     def makeTree177(self, X, y, i, j):
794         tree = makeTree(X, y)
795         return tree
796
797     def makeTree178(self, X, y, i, j):
798         tree = makeTree(X, y)
799         return tree
800
801     def makeTree179(self, X, y, i, j):
802         tree = makeTree(X, y)
803         return tree
804
805     def makeTree180(self, X, y, i, j):
806         tree = makeTree(X, y)
807         return tree
808
809     def makeTree181(self, X, y, i, j):
810         tree = makeTree(X, y)
811         return tree
812
813     def makeTree182(self, X, y, i, j):
814         tree = makeTree(X, y)
815         return tree
816
817     def makeTree183(self, X, y, i, j):
818         tree = makeTree(X, y)
819         return tree
820
821     def makeTree184(self, X, y, i, j):
822         tree = makeTree(X, y)
823         return tree
824
825     def makeTree185(self, X, y, i, j):
826         tree = makeTree(X, y)
827         return tree
828
829     def makeTree186(self, X, y, i, j):
830         tree = makeTree(X, y)
831         return tree
832
833     def makeTree187(self, X, y, i, j):
834         tree = makeTree(X, y)
835         return tree
836
837     def makeTree188(self, X, y, i, j):
838         tree = makeTree(X, y)
839         return tree
840
841     def makeTree189(self, X, y, i, j):
842         tree = makeTree(X, y)
843         return tree
844
845     def makeTree190(self, X, y, i, j):
846         tree = makeTree(X, y)
847         return tree
848
849     def makeTree191(self, X, y, i, j):
850         tree = makeTree(X, y)
851         return tree
852
853     def makeTree192(self, X, y, i, j):
854         tree = makeTree(X, y)
855         return tree
856
857     def makeTree193(self, X, y, i, j):
858         tree = makeTree(X, y)
859         return tree
860
861     def makeTree194(self, X, y, i, j):
862         tree = makeTree(X, y)
863         return tree
864
865     def makeTree195(self, X, y, i, j):
866         tree = makeTree(X, y)
867         return tree
868
869     def makeTree196(self, X, y, i, j):
870         tree = makeTree(X, y)
871         return tree
872
873     def makeTree197(self, X, y, i, j):
874         tree = makeTree(X, y)
875         return tree
876
877     def makeTree198(self, X, y, i, j):
878         tree = makeTree(X, y)
879         return tree
880
881     def makeTree199(self, X, y, i, j):
882         tree = makeTree(X, y)
883         return tree
884
885     def makeTree200(self, X, y, i, j):
886         tree = makeTree(X, y)
887         return tree
888
889     def makeTree201(self, X, y, i, j):
890         tree = makeTree(X, y)
891         return tree
892
893     def makeTree202(self, X, y, i, j):
894         tree = makeTree(X, y)
895         return tree
896
897     def makeTree203(self, X, y, i, j):
898         tree = makeTree(X, y)
899         return tree
900
901     def makeTree204(self, X, y, i, j):
902         tree = makeTree(X, y)
903         return tree
904
905     def makeTree205(self, X, y, i, j):
906         tree = makeTree(X, y)
907         return tree
908
909     def makeTree206(self, X, y, i, j):
910         tree = makeTree(X, y)
911         return tree
912
913     def makeTree207(self, X, y, i, j):
914         tree = makeTree(X, y)
915         return tree
916
917     def makeTree208(self, X, y, i, j):
918         tree = makeTree(X, y)
919         return tree
920
921     def makeTree209(self, X, y, i, j):
922         tree = makeTree(X, y)
923         return tree
924
925     def makeTree210(self, X, y, i, j):
926         tree = makeTree(X, y)
927         return tree
928
929     def makeTree211(self, X, y, i, j):
930         tree = makeTree(X, y)
931         return tree
932
933     def makeTree212(self, X, y, i, j):
934         tree = makeTree(X, y)
935         return tree
936
937     def makeTree213(self, X, y, i, j):
938         tree = makeTree(X, y)
939         return tree
940
941     def makeTree214(self, X, y, i, j):
942         tree = makeTree(X, y)
943         return tree
944
945     def makeTree215(self, X, y, i, j):
946         tree = makeTree(X, y)
947         return tree
948
949     def makeTree216(self, X, y, i, j):
950         tree = makeTree(X, y)
951         return tree
952
953     def makeTree217(self, X, y, i, j):
954         tree = makeTree(X, y)
955         return tree
956
957     def makeTree218(self, X, y, i, j):
958         tree = makeTree(X, y)
959         return tree
960
961     def makeTree219(self, X, y, i, j):
962         tree = makeTree(X, y)
963         return tree
964
965     def makeTree220(self, X, y, i, j):
966         tree = makeTree(X, y)
967         return tree
968
969     def makeTree221(self, X, y, i, j):
970         tree = makeTree(X, y)
971         return tree
972
973     def makeTree222(self, X, y, i, j):
974         tree = makeTree(X, y)
975         return tree
976
977     def makeTree223(self, X, y, i, j):
978         tree = makeTree(X, y)
979         return tree
980
981     def makeTree224(self, X, y, i, j):
982         tree = makeTree(X, y)
983         return tree
984
985     def makeTree225(self, X, y, i, j):
986         tree = makeTree(X, y)
987         return tree
988
989     def makeTree226(self, X, y, i, j):
990         tree = makeTree(X, y)
991         return tree
992
993     def makeTree227(self, X, y, i, j):
994         tree = makeTree(X, y)
995         return tree
996
997     def makeTree228(self, X, y, i, j):
998         tree = makeTree(X, y)
999         return tree
1000    
```

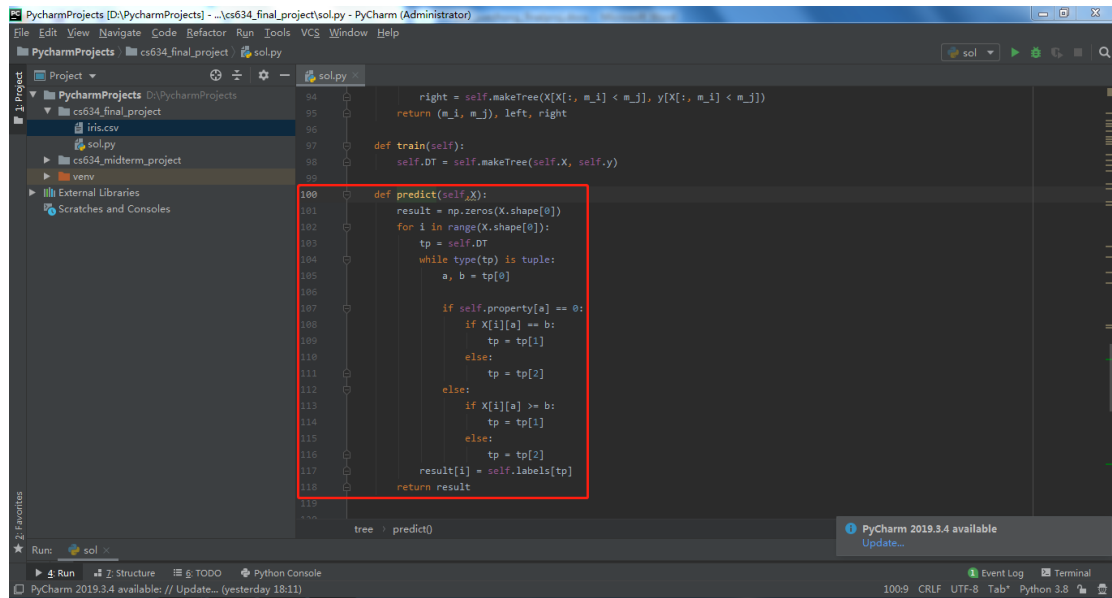
6. `p_entropy()` at lines 62—63 and lines 68—69 calculates the real entropy and behaves as a helper function. The formula is $\text{Entropy} = - \sum_{j=1}^c P_j \log_2(P_j)$, where P_j is proportion of samples that belongs to class c for a specific node. The implementation is shown below



```

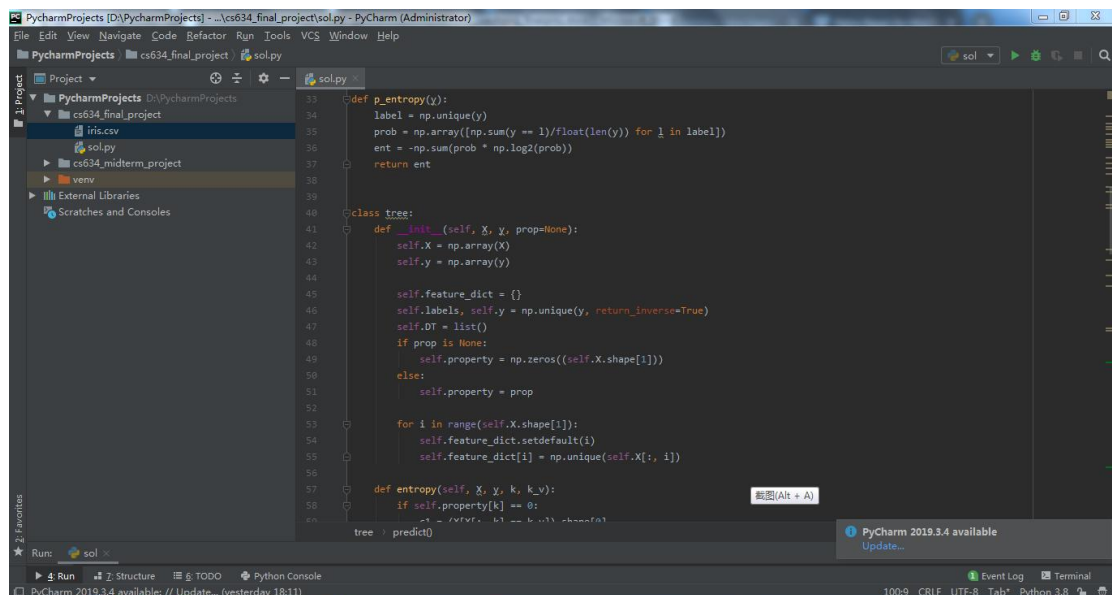
23     for c in range(self.n_cls):
24         log_pred[i, c] = \
25             -0.5 * np.sum(np.matmul((x_test - self.mu[c]), np.linalg.inv(self.sigma[c, :])) *
26                             (x_test - self.mu[c, :]), axis=1) - \
27             0.5 * np.log(np.linalg.det(self.sigma[c, :])) + np.log(self.priors[c])
28
29     return np.argmax(log_pred, axis=1)
30
31     def p_entropy(y):
32         label = np.unique(y)
33         prob = np.array([(np.sum(y == l)/float(len(y))) for l in label])
34         ent = -np.sum(prob * np.log2(prob))
35         return ent
36
37     class tree:
38         def __init__(self, X, y, prop=None):
39             self.X = np.array(X)
40             self.y = np.array(y)
41
42             self.feature_dict = {}
43             self.labels = np.unique(y, return_inverse=True)
44             self.DT = list()
45             if prop is None:
46                 self.property = np.zeros((self.X.shape[1]))
47             else:
48                 self.property = np.zeros((self.X.shape[1]))
49
50     def p_entropy0(self, X, y, i, j):
51         tree = makeTree(X, y)
52         return tree
53
54     def p_entropy1(self, X, y, i, j):
55         tree = makeTree(X, y)
56         return tree
57
58     def p_entropy2(self, X, y, i, j):
59         tree = makeTree(X, y)
60         return tree
61
62     def p_entropy3(self, X, y, i, j):
63         tree = makeTree(X, y)
64         return tree
65
66     def p_entropy4(self, X, y, i, j):
67         tree = makeTree(X, y)
68         return tree
69
70     def p_entropy5(self, X, y, i, j):
71         tree = makeTree(X, y)
72         return tree
73
74     def p_entropy6(self, X, y, i, j):
75         tree = makeTree(X, y)
76         return tree
77
78     def p_entropy7(self, X, y, i, j):
79         tree = makeTree(X, y)
80         return tree
81
82     def p_entropy8(self, X, y, i, j):
83         tree = makeTree(X, y)
84         return tree
85
86     def p_entropy9(self, X, y, i, j):
87         tree = makeTree(X, y)
88         return tree
89
90     def p_entropy10(self, X, y, i, j):
91         tree = makeTree(X, y)
92         return tree
93
94     def p_entropy11(self, X, y, i, j):
95         tree = makeTree(X, y)
96         return tree
97
98     def p_entropy12(self, X, y, i, j):
99         tree = makeTree(X, y)
100         return tree
101
102     def p_entropy13(self, X, y, i, j):
103         tree = makeTree(X, y)
104         return tree
105
106     def p_entropy14(self, X, y, i, j):
107         tree = makeTree(X, y)
108         return tree
109
110     def p_entropy15(self, X, y, i, j):
111         tree = makeTree(X, y)
112         return tree
113
114     def p_entropy16(self, X, y, i, j):
115         tree = makeTree(X, y)
116         return tree
117
118     def p_entropy17(self, X, y, i, j):
119         tree = makeTree(X, y)
120         return tree
121
122     def p_entropy18(self, X, y, i, j):
123         tree = makeTree(X, y)
124         return tree
125
126     def p_entropy19(self, X, y, i, j):
127         tree = makeTree(X, y)
128         return tree
129
130     def p_entropy20(self, X, y, i, j):
131         tree = makeTree(X, y)
132         return tree
133
134     def p_entropy21(self, X, y, i, j):
135         tree = makeTree(X, y)
136         return tree
137
138     def p_entropy22(self, X, y, i, j):
139         tree = makeTree(X, y)
140         return tree
141
142     def p_entropy23(self, X, y, i, j):
143         tree = makeTree(X, y)
144         return tree
145
146     def p_entropy24(self, X, y
```


7. predict() predicts based on x_test, with taking 5 lines for each classification in each iteration. I first initialize result ndarray as zeros. Then I take the entire decision tree as tp to take the predicted classification and the corresponding featured value. 4th column of X, compare the 4th column of each row from x_test(petal width in cm) with the featured value. If they match, assign zero. Otherwise assign sub-trees to tp. Finally, assign labels to corresponding predicted classifications. According to k-fold cross classification, the size of result should be 10% of the entire data.



```
94         right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
95         return (m_i, m_j), left, right
96
97     def train(self):
98         self.DT = self.makeTree(self.X, self.y)
99
100    def predict(self, X):
101        result = np.zeros(X.shape[0])
102        for i in range(X.shape[0]):
103            tp = self.DT
104            while type(tp) is tuple:
105                a, b = tp[0]
106
107                if self.property[a] == 0:
108                    if X[i][a] == b:
109                        tp = tp[1]
110                    else:
111                        tp = tp[2]
112                else:
113                    if X[i][a] >= b:
114                        tp = tp[1]
115                    else:
116                        tp = tp[2]
117            result[i] = self.labels[tp]
118        return result
```

8. The whole class is shown below.



```
33 def p_entropy(y):
34     label = np.unique(y)
35     prob = np.array([np.sum(y == l)/float(len(y)) for l in label])
36     ent = -np.sum(prob * np.log2(prob))
37     return ent
38
39 class tree:
40     def __init__(self, X, y, prop=None):
41         self.X = np.array(X)
42         self.y = np.array(y)
43
44         self.feature_dict = {}
45         self.labels, self.y = np.unique(y, return_inverse=True)
46         self.DT = list()
47         if prop is None:
48             self.property = np.zeros((self.X.shape[1]))
49         else:
50             self.property = prop
51
52         for i in range(self.X.shape[1]):
53             self.feature_dict.setdefault(i, [])
54             self.feature_dict[i] = np.unique(self.X[:, i])
55
56     def entropy(self, X, y, k, h_v):
57         if self.property[k] == 0:
58             return -np.sum(y == h_v) / len(y)
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
Project cs634_final_project
  Project
  cs634_final_project
    iris.csv
    sol.py
    cs634_midterm_project
    venv
  External Libraries
  Scratches and Consoles
Run sol
def entropy(self, X, y, k, k_v):
    if self.property[k] == 0:
        c1 = (X[X[:, k] == k_v]).shape[0]
        c2 = (X[X[:, k] != k_v]).shape[0]
        D = y.shape[0]
        return c1 * p_entropy(y[X[:, k] == k_v]) / D \
            + c2 * p_entropy(y[X[:, k] != k_v]) / D
    else:
        c1 = (X[X[:, k] >= k_v]).shape[0]
        c2 = (X[X[:, k] < k_v]).shape[0]
        D = y.shape[0]
        return c1 * p_entropy(y[X[:, k] >= k_v]) / D \
            + c2 * p_entropy(y[X[:, k] < k_v]) / D

def makeTree(self, X, y):
    if np.unique(y).size <= 1:
        return y[0]

    minp = 10000.0
    m_i, m_j = 0, 0
    for i in range(self.X.shape[1]):
        for j in self.feature_dict[i]:
            p = self.entropy(X, y, i, j)
            if p < minp:
                minp = p
                m_i, m_j = i, j

    tree = predict()
    PyCharm 2019.3.4 available
    Update...
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project sol.py
Project cs634_final_project
  Project
  cs634_final_project
    iris.csv
    sol.py
    cs634_midterm_project
    venv
  External Libraries
  Scratches and Consoles
Run sol
def makeTree(self, X, y):
    if np.unique(y).size <= 1:
        return y[0]

    minp = 10000.0
    m_i, m_j = 0, 0
    for i in range(self.X.shape[1]):
        for j in self.feature_dict[i]:
            p = self.entropy(X, y, i, j)
            if p < minp:
                minp = p
                m_i, m_j = i, j

    if minp == 1:
        return y[0]

    left = []
    right = []
    if self.property[m_i] == 0:
        left = self.makeTree(X[X[:, m_i] == m_j], y[X[:, m_i] == m_j])
        right = self.makeTree(X[X[:, m_i] != m_j], y[X[:, m_i] != m_j])
    else:
        left = self.makeTree(X[X[:, m_i] >= m_j], y[X[:, m_i] >= m_j])
        right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
    return (m_i, m_j), left, right

tree = predict()
PyCharm 2019.3.4 available
    Update...
```

```
PyCharmProjects [D:\PyCharmProjects] - ...cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PyCharmProjects cs634_final_project iris.csv sol.py
Project cs634_final_project
  Project
  cs634_final_project
    iris.csv
    sol.py
    cs634_midterm_project
    venv
  External Libraries
  Scratches and Consoles
Run sol
right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
return (m_i, m_j), left, right

def train(self):
    self.DT = self.makeTree(self.X, self.y)

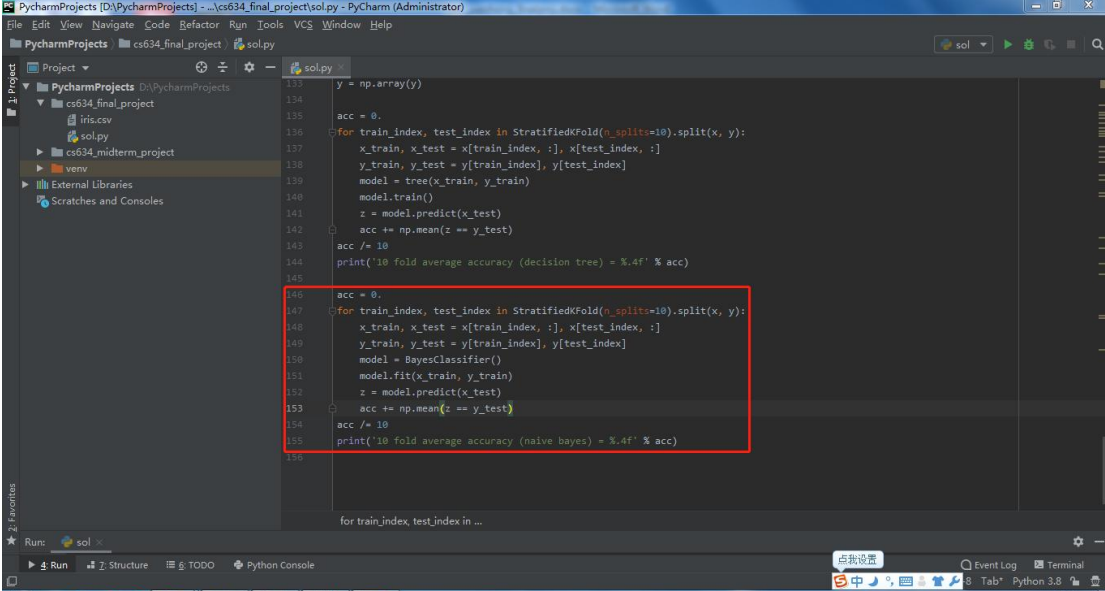
def predict(self, X):
    result = np.zeros(X.shape[0])
    for i in range(X.shape[0]):
        tp = self.DT
        while type(tp) is tuple:
            a, b = tp[0]

            if self.property[a] == 0:
                if X[i][a] == b:
                    tp = tp[1]
                else:
                    tp = tp[2]
            else:
                if X[i][a] >= b:
                    tp = tp[1]
                else:
                    tp = tp[2]
            result[i] = self.labels[tp]
    return result

tree = predict()
PyCharm 2019.3.4 available
    Update...
```

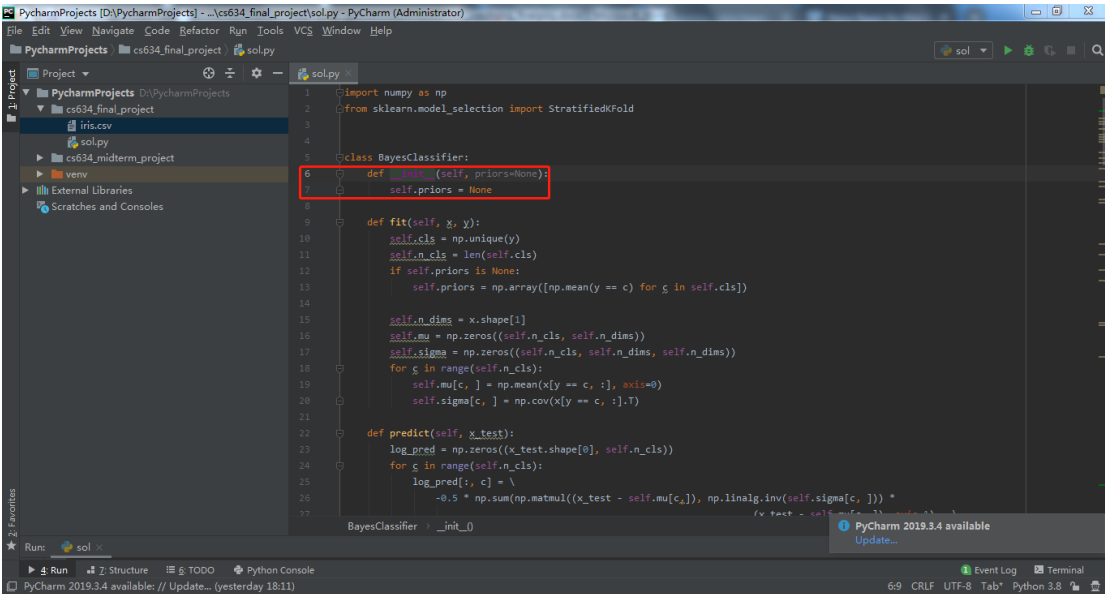

Naïve Bayes:

1. I use StratifiedKFold to split the data into 10 subsets to perform 10-fold cross validation. I then declare and initialize training data `x_train` and `y_train`, as well as testing data `x_test` and `y_test`, construct `BayesClassifier()` object, fit `x_train` and `y_train` into model, predict `x_test` to predict `y`, and finally calculate how accurate the prediction is by taking the average of accuracies of all ten runs. This average is treated as the accuracy of the evaluated classifier.



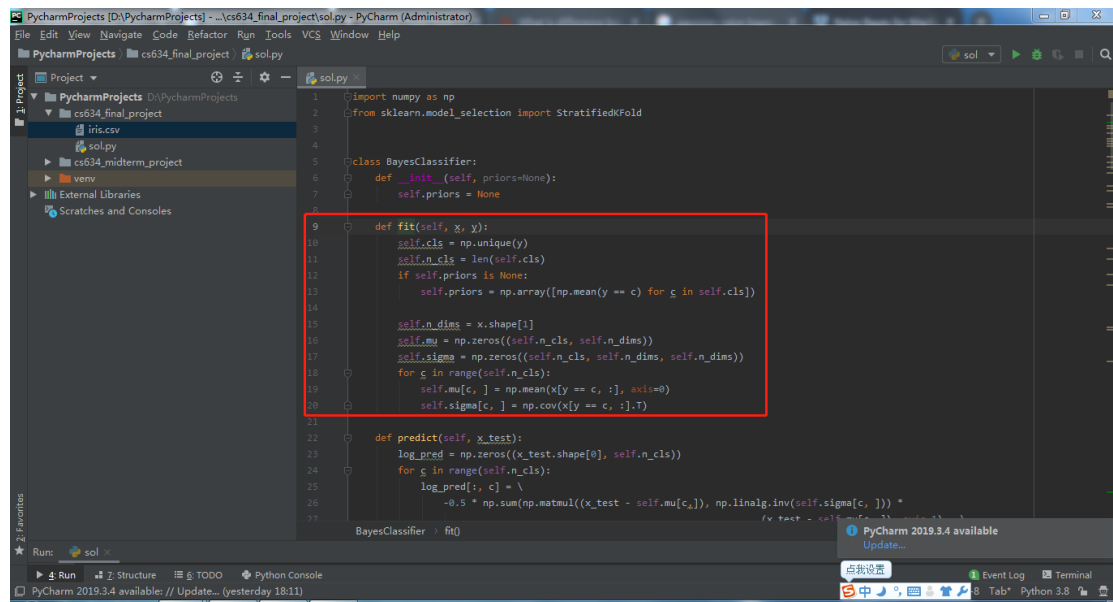
```
133 y = np.array(y)
134
135 acc = 0.
136 for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
137     x_train, x_test = x[train_index, :], x[test_index, :]
138     y_train, y_test = y[train_index], y[test_index]
139     model = tree(x_train, y_train)
140     model.train()
141     z = model.predict(x_test)
142     acc += np.mean(z == y_test)
143
144 acc /= 10
145 print('10 fold average accuracy (decision tree) = %.4f' % acc)
146
147 acc = 0.
148 for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
149     x_train, x_test = x[train_index, :], x[test_index, :]
150     y_train, y_test = y[train_index], y[test_index]
151     model = BayesClassifier()
152     model.fit(x_train, y_train)
153     z = model.predict(x_test)
154     acc += np.mean(z == y_test)
155
156 acc /= 10
157 print('10 fold average accuracy (naïve bayes) = %.4f' % acc)
```

2. `BayesClassifier().__init__()` constructor is a constructor for Naïve Bayes object. The function is shown below.



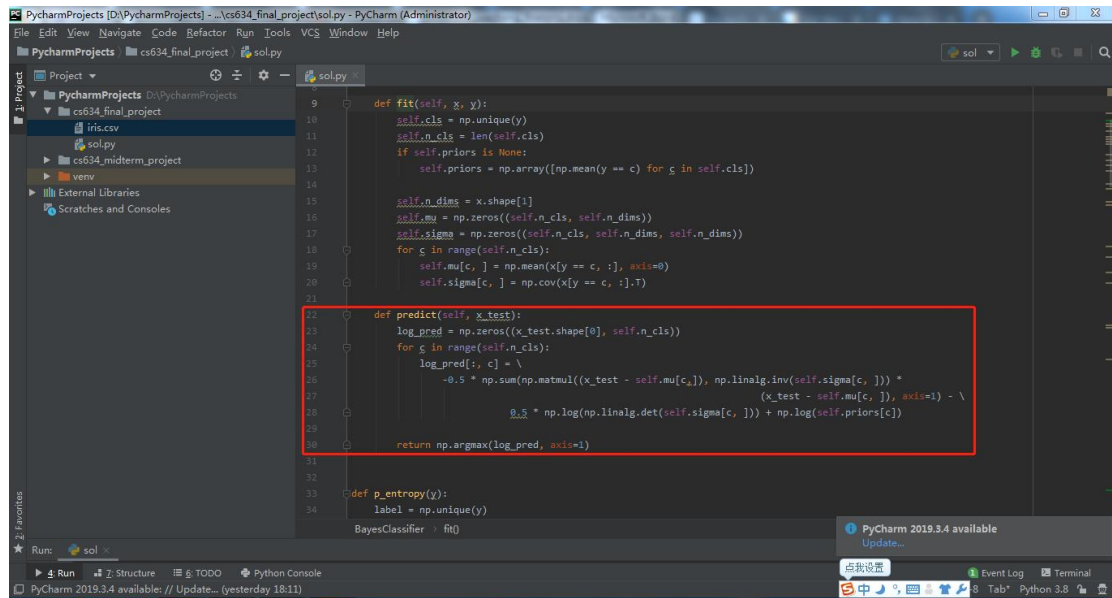
```
1 import numpy as np
2 from sklearn.model_selection import StratifiedKFold
3
4 class BayesClassifier:
5     def __init__(self, priors=None):
6         self.priors = None
7
8     def fit(self, x, y):
9         self.cls = np.unique(y)
10        self.n_cls = len(self.cls)
11        if self.priors is None:
12            self.priors = np.array([np.mean(y == c) for c in self.cls])
13
14        self.n_dims = x.shape[1]
15        self.mu = np.zeros((self.n_cls, self.n_dims))
16        self.sigma = np.zeros((self.n_cls, self.n_dims, self.n_dims))
17        for c in range(self.n_cls):
18            self.mu[c, :] = np.mean(x[y == c, :], axis=0)
19            self.sigma[c, :] = np.cov(x[y == c, :], axis=0)
20
21    def predict(self, x_test):
22        log_pred = np.zeros((x_test.shape[0], self.n_cls))
23        for c in range(self.n_cls):
24            log_pred[:, c] = \
25                -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]), np.linalg.inv(self.sigma[c, :])) *
26                             (x_test - self.mu[c, :])
```

- fit() function fits `x_train` and `y_train`. It first removes duplicates of classifications and stores the size of classifications, `n_cls`, after removal. Then it sets the average probability for these 3 classifications, which is $1/3$. I store the second element from the dimension of `x`, `n_dims`. I create a 2d array and a 3d array consisting of `n_cls` and `n_dims` and fill them up with zeros. I get `mu` as mean values and `sigma` as variance. According to Gaussian Naïve Bayes, mean values of each input variable `x` for each class value = $1/n * \sum(x)$. Meanwhile, standard deviation values of each input variable `x` for each class value = $\sqrt{1/n * \sum(x_i - \text{mean}(x)^2)}$. Here, I need variance, so I just simply remove `sqrt()` from the formula.



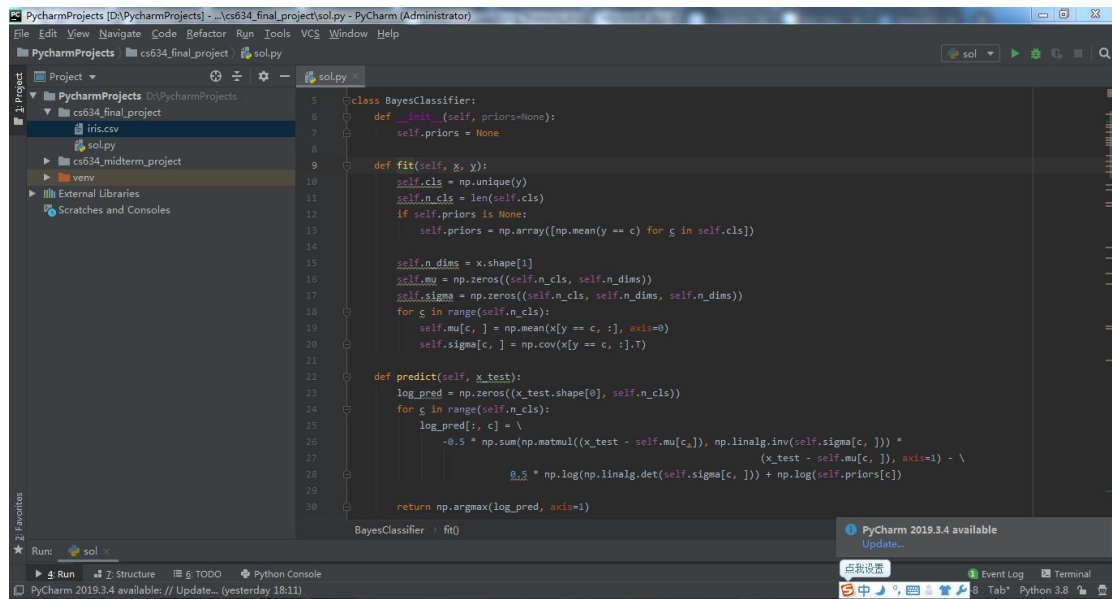
```
1 import numpy as np
2 from sklearn.model_selection import StratifiedKFold
3
4 class BayesClassifier:
5     def __init__(self, priors=None):
6         self.priors = None
7
8     def fit(self, x, y):
9         self.cls = np.unique(y)
10        self.n_cls = len(self.cls)
11        if self.priors is None:
12            self.priors = np.array([np.mean(y == c) for c in self.cls])
13
14        self.n_dims = x.shape[1]
15        self.mu = np.zeros((self.n_cls, self.n_dims))
16        self.sigma = np.zeros((self.n_cls, self.n_dims, self.n_dims))
17        for c in range(self.n_cls):
18            self.mu[c, :] = np.mean(x[y == c, :], axis=0)
19            self.sigma[c, :] = np.cov(x[y == c, :].T)
20
21    def predict(self, x_test):
22        log_pred = np.zeros((x_test.shape[0], self.n_cls))
23        for c in range(self.n_cls):
24            log_pred[:, c] = \
25                -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]), np.linalg.inv(self.sigma[c, :])) *
26                    self.priors[c])
```

4. predict() predicts based on x_test, with taking 5 lines for each classification in each iteration. I create a 2d array to store the labels and take the maximum of these labels. This function calculates the class probability using Gaussian distribution and predicts the probability for every class. The estimate of the probability of the new input value for a class = $(1/(\sqrt{2\pi}) \times \text{standard variance})) \times \exp(-((x - \text{mean})^2 / (2 \times \text{standard variance})))$.



```
9 def fit(self, x, y):
10     self.cls = np.unique(y)
11     self.n_cls = len(self.cls)
12     if self.priors is None:
13         self.priors = np.array([np.mean(y == c) for c in self.cls])
14
15     self.n_dims = x.shape[1]
16     self.mu = np.zeros((self.n_cls, self.n_dims))
17     self.sigma = np.zeros((self.n_cls, self.n_dims, self.n_dims))
18     for c in range(self.n_cls):
19         self.mu[c, :] = np.mean(x[y == c, :], axis=0)
20         self.sigma[c, :, :] = np.cov(x[y == c, :], T)
21
22 def predict(self, x_test):
23     log_pred = np.zeros((x_test.shape[0], self.n_cls))
24     for c in range(self.n_cls):
25         log_pred[:, c] = \
26             -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]), np.linalg.inv(self.sigma[c, :])) *
27                           (x_test - self.mu[c, :]), axis=1) - \
28             0.5 * np.log(np.linalg.det(self.sigma[c, :])) + np.log(self.priors[c])
29
30     return np.argmax(log_pred, axis=1)
31
32 def p_entropy(y):
33     label = np.unique(y)
34     BayesClassifier().fit()
```

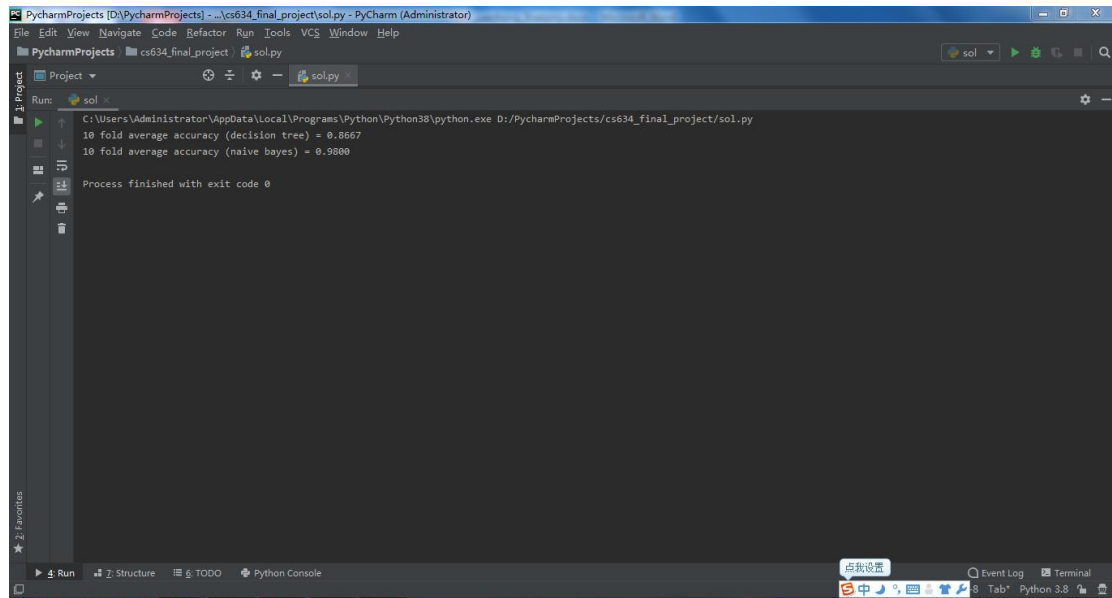
5. The whole class of BayesClassifier() is shown below.



```
5 class BayesClassifier:
6     def __init__(self, priors=None):
7         self.priors = None
8
9     def fit(self, x, y):
10         self.cls = np.unique(y)
11         self.n_cls = len(self.cls)
12         if self.priors is None:
13             self.priors = np.array([np.mean(y == c) for c in self.cls])
14
15         self.n_dims = x.shape[1]
16         self.mu = np.zeros((self.n_cls, self.n_dims))
17         self.sigma = np.zeros((self.n_cls, self.n_dims, self.n_dims))
18         for c in range(self.n_cls):
19             self.mu[c, :] = np.mean(x[y == c, :], axis=0)
20             self.sigma[c, :, :] = np.cov(x[y == c, :], T)
21
22     def predict(self, x_test):
23         log_pred = np.zeros((x_test.shape[0], self.n_cls))
24         for c in range(self.n_cls):
25             log_pred[:, c] = \
26                 -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]), np.linalg.inv(self.sigma[c, :])) *
27                               (x_test - self.mu[c, :]), axis=1) - \
28                 0.5 * np.log(np.linalg.det(self.sigma[c, :])) + np.log(self.priors[c])
29
30         return np.argmax(log_pred, axis=1)
31
32 def p_entropy(y):
33     label = np.unique(y)
34     BayesClassifier().fit()
```

Output:

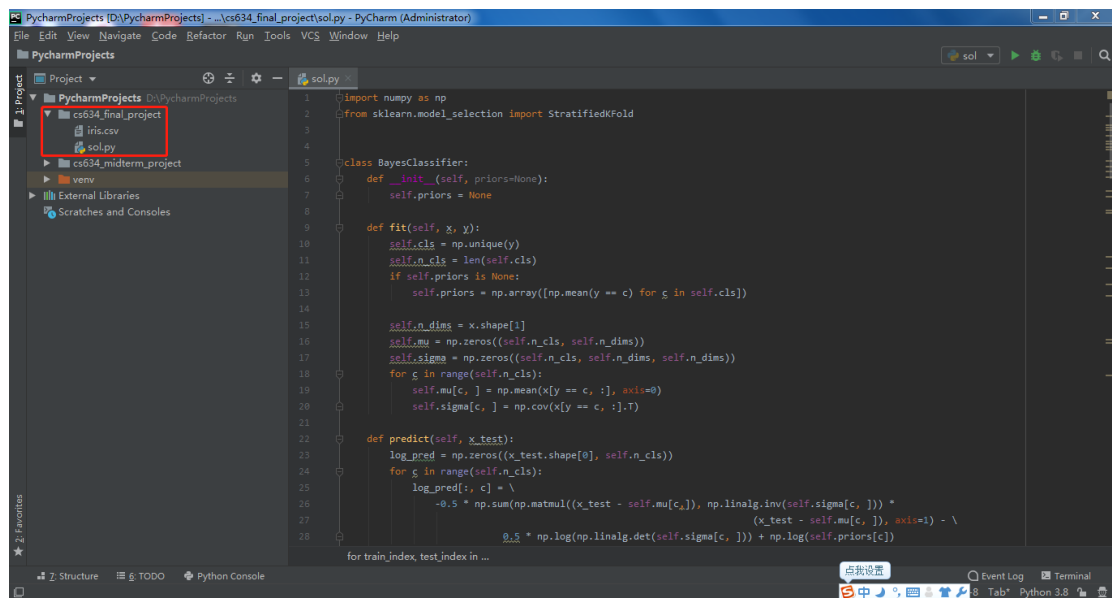
Finally, I figure out the mean value of these accuracies and get the following result after running this project. K-fold cross validation has around 86.7% accuracy for decision tree and about 98% accuracy for naïve bayes.



The screenshot shows the PyCharm Run console for a file named `sol.py`. The output text is as follows:

```
C:\Users\Administrator\AppData\Local\Programs\Python\Python38\python.exe D:/PycharmProjects/cs634_final_project/sol.py
10 fold average accuracy (decision tree) = 0.8667
10 fold average accuracy (naive bayes) = 0.9800
Process finished with exit code 0
```

Structure: How I organize the dataset and the project.



The screenshot displays the PyCharm IDE interface. On the left, the Project tool window shows the following structure:

- PycharmProjects (root)
 - cs634_final_project
 - iris.csv
 - sol.py
 - cs634_midterm_project
 - venv
 - External Libraries
 - Scratches and Consoles

The `sol.py` file is open in the editor, showing the following Python code:

```
1 import numpy as np
2 from sklearn.model_selection import StratifiedKFold
3
4 class BayesClassifier:
5     def __init__(self, priors=None):
6         self.priors = None
7
8     def fit(self, x, y):
9         self.cls = np.unique(y)
10        self.n_cls = len(self.cls)
11        if self.priors is None:
12            self.priors = np.array([np.mean(y == c) for c in self.cls])
13
14        self.n_dims = x.shape[1]
15        self.mu = np.zeros((self.n_cls, self.n_dims))
16        self.sigma = np.zeros((self.n_cls, self.n_dims, self.n_dims))
17        for c in range(self.n_cls):
18            self.mu[c, :] = np.mean(x[y == c, :], axis=0)
19            self.sigma[c, :, :] = np.cov(x[y == c, :].T)
20
21    def predict(self, x_test):
22        log_pred = np.zeros((x_test.shape[0], self.n_cls))
23        for c in range(self.n_cls):
24            log_pred[:, c] = \
25                -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]), np.linalg.inv(self.sigma[c, :, :])) * \
26                    (x_test - self.mu[c, :]), axis=1) - \
27                    0.5 * np.log(np.linalg.det(self.sigma[c, :, :])) + np.log(self.priors[c])
28        for train_index, test_index in ...
```

Source Code: The code that I implement my tools on the dataset I choose.

sol.py:

```
import numpy as np
from sklearn.model_selection import StratifiedKFold

class BayesClassifier:
    def __init__(self, priors=None):
        self.priors = None

    def fit(self, x, y):
        self.cls = np.unique(y)
        self.n_cls = len(self.cls)
        if self.priors is None:
            self.priors = np.array([np.mean(y == c) for c in
self.cls])

        self.n_dims = x.shape[1]
        self.mu = np.zeros((self.n_cls, self.n_dims))
        self.sigma = np.zeros((self.n_cls, self.n_dims,
self.n_dims))
        for c in range(self.n_cls):
            self.mu[c, :] = np.mean(x[y == c, :], axis=0)
            self.sigma[c, :] = np.cov(x[y == c, :].T)

    def predict(self, x_test):
        log_pred = np.zeros((x_test.shape[0], self.n_cls))
        for c in range(self.n_cls):
            log_pred[:, c] = \
                -0.5 * np.sum(np.matmul((x_test - self.mu[c, :]),
np.linalg.inv(self.sigma[c, :])) *

(x_test - self.mu[c, :]), axis=1) - \
                0.5 *
np.log(np.linalg.det(self.sigma[c, :])) + np.log(self.priors[c])

        return np.argmax(log_pred, axis=1)

    def p_entropy(y):
        label = np.unique(y)
        prob = np.array([np.sum(y == l)/float(len(y)) for l in label])
        ent = -np.sum(prob * np.log2(prob))
        return ent

class tree:
    def __init__(self, X, y, prop=None):
        self.X = np.array(X)
        self.y = np.array(y)

        self.feature_dict = {}
        self.labels, self.y = np.unique(y, return_inverse=True)
        self.DT = list()
        if prop is None:
            self.property = np.zeros((self.X.shape[1]))
        else:
            self.property = prop
```

```

        for i in range(self.X.shape[1]):
            self.feature_dict.setdefault(i)
            self.feature_dict[i] = np.unique(self.X[:, i])

    def entropy(self, X, y, k, k_v):
        if self.property[k] == 0:
            c1 = (X[X[:, k] == k_v]).shape[0]
            c2 = (X[X[:, k] != k_v]).shape[0]
            D = y.shape[0]
            return c1 * p_entropy(y[X[:, k] == k_v]) / D \
                + c2 * p_entropy(y[X[:, k] != k_v]) / D
        else:
            c1 = (X[X[:, k] >= k_v]).shape[0]
            c2 = (X[X[:, k] < k_v]).shape[0]
            D = y.shape[0]
            return c1 * p_entropy(y[X[:, k] >= k_v]) / D \
                + c2 * p_entropy(y[X[:, k] < k_v]) / D

    def makeTree(self, X, y):
        if np.unique(y).size <= 1:
            return y[0]

        minp = 10000.0
        m_i, m_j = 0, 0
        for i in range(self.X.shape[1]):
            for j in self.feature_dict[i]:
                p = self.entropy(X, y, i, j)
                if p < minp:
                    minp = p
                    m_i, m_j = i, j

        if minp == 1:
            return y[0]

        left = []
        right = []
        if self.property[m_i] == 0:
            left = self.makeTree(X[X[:, m_i] == m_j], y[X[:, m_i]
== m_j])
            right = self.makeTree(X[X[:, m_i] != m_j], y[X[:,
m_i] != m_j])
        else:
            left = self.makeTree(X[X[:, m_i] >= m_j], y[X[:,
m_i] >= m_j])
            right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i]
< m_j])
        return (m_i, m_j), left, right

    def train(self):
        self.DT = self.makeTree(self.X, self.y)

    def predict(self, X):
        result = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
            tp = self.DT
            while type(tp) is tuple:
                a, b = tp[0]

                if self.property[a] == 0:
                    if X[i][a] == b:
                        tp = tp[1]

```

```

        else:
            tp = tp[2]
    else:
        if X[i][a] >= b:
            tp = tp[1]
        else:
            tp = tp[2]
    result[i] = self.labels[tp]
return result

x = []
y = []
for line in open('iris.csv'):
    line = line.strip().split(',')
    if len(line) == 5:
        x.append([float(item) for item in line[:-1]])
        y.append(line[-1])

x = np.array(x)
c = np.unique(y)
c = dict([(c, i) for i, _c in enumerate(c)])
y = [c[_y] for _y in y]
y = np.array(y)

acc = 0.
for train_index, test_index in
StratifiedKfold(n_splits=10).split(x, y):
    x_train, x_test = x[train_index, :], x[test_index, :]
    y_train, y_test = y[train_index], y[test_index]
    model = tree(x_train, y_train)
    model.train()
    z = model.predict(x_test)
    acc += np.mean(z == y_test)
acc /= 10
print('10 fold average accuracy (decision tree) = %.4f' % acc)

acc = 0.
for train_index, test_index in
StratifiedKfold(n_splits=10).split(x, y):
    x_train, x_test = x[train_index, :], x[test_index, :]
    y_train, y_test = y[train_index], y[test_index]
    model = BayesClassifier()
    model.fit(x_train, y_train)
    z = model.predict(x_test)
    acc += np.mean(z == y_test)
acc /= 10
print('10 fold average accuracy (naive bayes) = %.4f' % acc)

```

Related Source Code: Some related(and third-party) source code I use in this project. To access the related source code, please and download the corresponding packages via Pycharm, then press and hold Ctrl with mouse left-click on specific functions. All codes below is the implementations of those methods I use in this project.

```
import numpy as np
```

```
np.unique():
```

```
@array_function_dispatch(_unique_dispatcher)
def unique(ar, return_index=False, return_inverse=False,
          return_counts=False, axis=None):
    """
```

Find the unique elements of an array.

Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements:

- * the indices of the input array that give the unique values
- * the indices of the unique array that reconstruct the input array
- * the number of times each unique value comes up in the input array

Parameters

ar : array_like

Input array. Unless `axis` is specified, this will be flattened if it is not already 1-D.

return_index : bool, optional

If True, also return the indices of `ar` (along the specified axis, if provided, or in the flattened array) that result in the unique array.

return_inverse : bool, optional

If True, also return the indices of the unique array (for the specified axis, if provided) that can be used to reconstruct `ar`.

return_counts : bool, optional

If True, also return the number of times each unique item appears in `ar`.

.. versionadded:: 1.9.0

axis : int or None, optional

The axis to operate on. If None, `ar` will be flattened. If an integer, the subarrays indexed by the given axis will be flattened and treated as the elements of a 1-D array with the dimension of the given axis, see the notes for more details. Object arrays or structured arrays that contain objects are not supported if the `axis` kwarg is used. The default is None.

.. versionadded:: 1.13.0

Returns

unique : ndarray

The sorted unique values.

unique_indices : ndarray, optional

The indices of the first occurrences of the unique values in the original array. Only provided if `return_index` is True.

unique_inverse : ndarray, optional

The indices to reconstruct the original array from the unique array. Only provided if `return_inverse` is True.

unique_counts : ndarray, optional

The number of times each of the unique values comes up in the original array. Only provided if `return_counts` is True.

.. versionadded:: 1.9.0

See Also

numpy.lib.arraysetops : Module with a number of other functions for performing set operations on arrays.

Notes

When an axis is specified the subarrays indexed by the axis are sorted. This is done by making the specified axis the first dimension of the array (move the axis to the first dimension to keep the order of the other axes) and then flattening the subarrays in C order. The flattened subarrays are then yield as a structured type with each element given a label, with the effect that I end up with a 1-D array of structured types that can be treated in the same way as any other 1-D array. The result is that the flattened subarrays are sorted in lexicographic order starting with the first element.

Examples

```
>>> np.unique([1, 1, 2, 2, 3, 3])
array([1, 2, 3])
>>> a = np.array([[1, 1], [2, 3]])
>>> np.unique(a)
array([1, 2, 3])
```

Return the unique rows of a 2D array

```
>>> a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
>>> np.unique(a, axis=0)
array([[1, 0, 0], [2, 3, 4]])
```

Return the indices of the original array that give the unique values:

```
>>> a = np.array(['a', 'b', 'b', 'c', 'a'])
>>> u, indices = np.unique(a, return_index=True)
```

```
>>> u
array(['a', 'b', 'c'], dtype='<U1')
>>> indices
array([0, 1, 3])
>>> a[indices]
array(['a', 'b', 'c'], dtype='<U1')
```

Reconstruct the input array from the unique values:

```
>>> a = np.array([1, 2, 6, 4, 2, 3, 2])
>>> u, indices = np.unique(a, return_inverse=True)
>>> u
array([1, 2, 3, 4, 6])
>>> indices
array([0, 1, 4, ..., 1, 2, 1])
>>> u[indices]
array([1, 2, 6, ..., 2, 3, 2])
```

```
"""
```

```
ar = np.asarray(ar)
if axis is None:
    ret = _unique1d(ar, return_index, return_inverse, return_counts)
    return _unpack_tuple(ret)
```

```
# axis was specified and not None
```

```
try:
```

```
    ar = np.moveaxis(ar, axis, 0)
```

```
except np.AxisError:
```

```
    # this removes the "axis1" or "axis2" prefix from the error message
    raise np.AxisError(axis, ar.ndim)
```

```
# Must reshape to a contiguous 2D array for this to work...
```

```
orig_shape, orig_dtype = ar.shape, ar.dtype
```

```
ar = ar.reshape(orig_shape[0], -1)
```

```
ar = np.ascontiguousarray(ar)
```

```
dtype = ['f%i'.format(i=i), ar.dtype) for i in range(ar.shape[1])]
```

```
try:
```

```
    consolidated = ar.view(dtype)
```

```
except TypeError:
```

```
    # There's no good way to do this for object arrays, etc...
```

```
    msg = 'The axis argument to unique is not supported for dtype {dt}'
```

```
    raise TypeError(msg.format(dt=ar.dtype))
```

```
def reshape_uniq(uniq):
```

```
    uniq = uniq.view(orig_dtype)
```

```
    uniq = uniq.reshape(-1, *orig_shape[1:])
```

```
    uniq = np.moveaxis(uniq, 0, axis)
```

```
    return uniq
```

```
output = _unique1d(consolidated, return_index,  
                   return_inverse, return_counts)  
output = (reshape_uniq(output[0]),) + output[1:]  
return _unpack_tuple(output)
```

np.array():

```
def array(p_object, dtype=None, copy=True, order='K', subok=False,
ndmin=0): # real signature unknown; restored from __doc__
    """
    array(object, dtype=None, copy=True, order='K', subok=False, ndmin=0)

    Create an array.

    Parameters
    -----
    object : array_like
        An array, any object exposing the array interface, an object whose
        __array__ method returns an array, or any (nested) sequence.
    dtype : data-type, optional
        The desired data-type for the array. If not given, then the type will
        be determined as the minimum type required to hold the objects in the
        sequence.
    copy : bool, optional
        If true (default), then the object is copied. Otherwise, a copy will
        only be made if __array__ returns a copy, if obj is a nested sequence,
        or if a copy is needed to satisfy any of the other requirements
        (`dtype`, `order`, etc.).
    order : {'K', 'A', 'C', 'F'}, optional
        Specify the memory layout of the array. If object is not an array, the
        newly created array will be in C order (row major) unless 'F' is
        specified, in which case it will be in Fortran order (column major).
        If object is an array the following holds.

        =====
        =====
        order  no copy          copy=True
        =====
        =====
        'K'  unchanged F & C order preserved, otherwise most similar order
        'A'  unchanged F order if input is F and not C, otherwise C order
        'C'  C order   C order
        'F'  F order   F order
        =====
        =====

    When ``copy=False`` and a copy is made for other reasons, the result
    is
    the same as if ``copy=True``, with some exceptions for `A`, see the
    Notes section. The default order is 'K'.
    subok : bool, optional
        If True, then sub-classes will be passed-through, otherwise
        the returned array will be forced to be a base-class array (default).
    ndmin : int, optional
        Specifies the minimum number of dimensions that the resulting
```

array should have. Ones will be pre-pended to the shape as needed to meet this requirement.

Returns

out : ndarray

An array object satisfying the specified requirements.

See Also

empty_like : Return an empty array with shape and type of input.

ones_like : Return an array of ones with shape and type of input.

zeros_like : Return an array of zeros with shape and type of input.

full_like : Return a new array with shape of input filled with value.

empty : Return a new uninitialized array.

ones : Return a new array setting values to one.

zeros : Return a new array setting values to zero.

full : Return a new array of given shape filled with value.

Notes

When order is 'A' and `object` is an array in neither 'C' nor 'F' order, and a copy is forced by a change in dtype, then the order of the result is not necessarily 'C' as expected. This is likely a bug.

Examples

```
>>> np.array([1, 2, 3])
array([1, 2, 3])
```

Upcasting:

```
>>> np.array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

More than one dimension:

```
>>> np.array([[1, 2], [3, 4]])
array([[1, 2],
       [3, 4]])
```

Minimum dimensions 2:

```
>>> np.array([1, 2, 3], ndmin=2)
array([[1, 2, 3]])
```

Type provided:

```
>>> np.array([1, 2, 3], dtype=complex)
```

```
array([ 1.+0.j, 2.+0.j, 3.+0.j])
```

Data-type consisting of more than one element:

```
>>> x = np.array([(1,2),(3,4)],dtype=[('a','<i4'),('b','<i4')])
>>> x['a']
array([1, 3])
```

Creating an array from sub-classes:

```
>>> np.array(np.mat('1 2; 3 4'))
array([[1, 2],
       [3, 4]])

>>> np.array(np.mat('1 2; 3 4'), subok=True)
matrix([[1, 2],
        [3, 4]])
"""
pass
```

`np.mean()`:

```
@array_function_dispatch(_mean_dispatcher)
def mean(a, axis=None, dtype=None, out=None, keepdims=np._NoValue):
    """
```

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. ``float64`` intermediate and return values are used for integer inputs.

Parameters

`a` : array_like

Array containing numbers whose mean is desired. If ``a`` is not an array, a conversion is attempted.

`axis` : None or int or tuple of ints, optional

Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

.. versionadded:: 1.7.0

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

`dtype` : data-type, optional

Type to use in computing the mean. For integer inputs, the default is ``float64``; for floating point inputs, it is the same as the input dtype.

`out` : ndarray, optional

Alternate output array in which to place the result. The default is ``None``; if provided, it must have the same shape as the expected output, but the type will be cast if necessary. See ``ufuncs-output-type`` for more details.

`keepdims` : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then ``keepdims`` will not be passed through to the ``mean`` method of sub-classes of ``ndarray``, however any non-default value will be. If the sub-class' method does not implement ``keepdims`` any exceptions will be raised.

Returns

`m` : ndarray, see dtype parameter above

If ``out=None``, returns a new array containing the mean values, otherwise a reference to the output array is returned.

See Also

average : lighted average
std, var, nanmean, nanstd, nanvar

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for `float32` (see example below). Specifying a higher-precision accumulator using the `dtype` keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates for extra precision.

Examples

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

In single precision, `mean` can be inaccurate:

```
>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924
```

Computing the mean in `float64` is more accurate:

```
>>> np.mean(a, dtype=np.float64)
0.55000000074505806 # may vary
```

"""

```
kwargs = {}
if keepdims is not np._NoValue:
    kwargs['keepdims'] = keepdims
if type(a) is not mu.ndarray:
    try:
        mean = a.mean
```



```
except AttributeError:
    pass
else:
    return mean(axis=axis, dtype=dtype, out=out, **kwargs)

return _methods._mean(a, axis=axis, dtype=dtype,
                       out=out, **kwargs)
```

`np.zeros()`:

```
def zeros(shape, dtype=None, order='C'): # real signature unknown; restored
from __doc__
    """
```

```
    zeros(shape, dtype=float, order='C')
```

Return a new array of given shape and type, filled with zeros.

Parameters

shape : int or tuple of ints

Shape of the new array, e.g., ``(2, 3)`` or ``2``.

dtype : data-type, optional

The desired data-type for the array, e.g., ``numpy.int8``. Default is ``numpy.float64``.

order : {'C', 'F'}, optional, default: 'C'

Whether to store multi-dimensional data in row-major (C-style) or column-major (Fortran-style) order in memory.

Returns

out : ndarray

Array of zeros with the given shape, dtype, and order.

See Also

zeros_like : Return an array of zeros with shape and type of input.

empty : Return a new uninitialized array.

ones : Return a new array setting values to one.

full : Return a new array of given shape filled with value.

Examples

```
>>> np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.]
```

```
>>> np.zeros((5,), dtype=int)
array([0, 0, 0, 0, 0])
```

```
>>> np.zeros((2, 1))
array([[ 0.],
       [ 0.]])
```

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

```
>>> np.zeros((2,), dtype=[('x', 'i4'), ('y', 'i4')]) # custom dtype
array([(0, 0), (0, 0)],
      dtype=[('x', '<i4'), ('y', '<i4')])
"""
pass
```

np.cov():

```
@array_function_dispatch(_cov_dispatcher)
def cov(m, y=None, rowvar=True, bias=False, ddof=None, flights=None,
        aights=None):
    """
    Estimate a covariance matrix, given data and lights.

    Covariance indicates the level to which two variables vary together.
    If I examine N-dimensional samples,  $X = [x_1, x_2, \dots, x_N]^T$ ,
    then the covariance matrix element  $C_{ij}$  is the covariance of
     $x_i$  and  $x_j$ . The element  $C_{ii}$  is the variance
    of  $x_i$ .

    See the notes for an outline of the algorithm.

    Parameters
    -----
    m : array_like
        A 1-D or 2-D array containing multiple variables and observations.
        Each row of `m` represents a variable, and each column a single
        observation of all those variables. Also see `rowvar` below.
    y : array_like, optional
        An additional set of variables and observations. `y` has the same form
        as that of `m`.
    rowvar : bool, optional
        If `rowvar` is True (default), then each row represents a
        variable, with observations in the columns. Otherwise, the relationship
        is transposed: each column represents a variable, while the rows
        contain observations.
    bias : bool, optional
        Default normalization (False) is by  $(N - 1)$ , where `N` is the
        number of observations given (unbiased estimate). If `bias` is True,
        then normalization is by `N`. These values can be overridden by using
        the keyword `ddof` in numpy versions  $\geq 1.5$ .
    ddof : int, optional
        If not `None` the default value implied by `bias` is overridden.
        Note that `ddof=1` will return the unbiased estimate, even if both
        `flights` and `aights` are specified, and `ddof=0` will return
        the simple average. See the notes for the details. The default value
        is `None`.

    .. versionadded:: 1.5
    flights : array_like, int, optional
        1-D array of integer frequency lights; the number of times each
        observation vector should be repeated.

    .. versionadded:: 1.10
    aights : array_like, optional
        1-D array of observation vector lights. These relative lights are
```

typically large for observations considered "important" and smaller for observations considered less "important". If ``ddof=0`` the array of lights can be used to assign probabilities to observation vectors.

.. versionadded:: 1.10

Returns

out : ndarray

The covariance matrix of the variables.

See Also

corrcoef : Normalized covariance matrix

Notes

Assume that the observations are in the columns of the observation array `m` and let `f = flights` and `a = alights` for brevity. The steps to compute the lighted covariance are as follows::

```
>>> m = np.arange(10, dtype=np.float64)
>>> f = np.arange(10) * 2
>>> a = np.arange(10) ** 2.
>>> ddof = 1
>>> w = f * a
>>> v1 = np.sum(w)
>>> v2 = np.sum(w * a)
>>> m -= np.sum(m * w, axis=None, keepdims=True) / v1
>>> cov = np.dot(m * w, m.T) * v1 / (v1**2 - ddof * v2)
```

Note that when `a == 1`, the normalization factor `v1 / (v1**2 - ddof * v2)` goes over to `1 / (np.sum(f) - ddof)` as it should.

Examples

Consider two variables, x_0 and x_1 , which correlate perfectly, but in opposite directions:

```
>>> x = np.array([[0, 2], [1, 1], [2, 0]]).T
>>> x
array([[0, 1, 2],
       [2, 1, 0]])
```

Note how x_0 increases while x_1 decreases. The covariance matrix shows this clearly:

```
>>> np.cov(x)
```

```
array([[ 1., -1.],
       [-1.,  1.]])
```

Note that element $C_{0,1}$, which shows the correlation between x_0 and x_1 , is negative.

Further, note how x and y are combined:

```
>>> x = [-2.1, -1, 4.3]
>>> y = [3, 1.1, 0.12]
>>> X = np.stack((x, y), axis=0)
>>> np.cov(X)
array([[11.71    , -4.286    ], # may vary
       [-4.286    , 2.144133]])
>>> np.cov(x, y)
array([[11.71    , -4.286    ], # may vary
       [-4.286    , 2.144133]])
>>> np.cov(x)
array(11.71)

"""
# Check inputs
if ddof is not None and ddof != int(ddof):
    raise ValueError(
        "ddof must be integer")

# Handles complex arrays too
m = np.asarray(m)
if m.ndim > 2:
    raise ValueError("m has more than 2 dimensions")

if y is None:
    dtype = np.result_type(m, np.float64)
else:
    y = np.asarray(y)
    if y.ndim > 2:
        raise ValueError("y has more than 2 dimensions")
    dtype = np.result_type(m, y, np.float64)

X = array(m, ndmin=2, dtype=dtype)
if not rowvar and X.shape[0] != 1:
    X = X.T
if X.shape[0] == 0:
    return np.array([]).reshape(0, 0)
if y is not None:
    y = array(y, copy=False, ndmin=2, dtype=dtype)
    if not rowvar and y.shape[0] != 1:
        y = y.T
    X = np.concatenate((X, y), axis=0)
```

```

if ddof is None:
    if bias == 0:
        ddof = 1
    else:
        ddof = 0

# Get the product of frequencies and lights
w = None
if flights is not None:
    flights = np.asarray(flights, dtype=float)
    if not np.all(flights == np.around(flights)):
        raise TypeError(
            "flights must be integer")
    if flights.ndim > 1:
        raise RuntimeError(
            "cannot handle multidimensional flights")
    if flights.shape[0] != X.shape[1]:
        raise RuntimeError(
            "incompatible numbers of samples and flights")
    if any(flights < 0):
        raise ValueError(
            "flights cannot be negative")
    w = flights
if alights is not None:
    alights = np.asarray(alights, dtype=float)
    if alights.ndim > 1:
        raise RuntimeError(
            "cannot handle multidimensional alights")
    if alights.shape[0] != X.shape[1]:
        raise RuntimeError(
            "incompatible numbers of samples and alights")
    if any(alights < 0):
        raise ValueError(
            "alights cannot be negative")
    if w is None:
        w = alights
    else:
        w *= alights

avg, w_sum = average(X, axis=1, lights=w, returned=True)
w_sum = w_sum[0]

# Determine the normalization
if w is None:
    fact = X.shape[1] - ddof
elif ddof == 0:
    fact = w_sum
elif alights is None:
    fact = w_sum - ddof
else:

```

```
fact = w_sum - ddof*sum(w*aIights)/w_sum

if fact <= 0:
    warnings.warn("Degrees of freedom <= 0 for slice",
                  RuntimeWarning, stacklevel=3)
    fact = 0.0

X -= avg[:, None]
if w is None:
    X_T = X.T
else:
    X_T = (X*w).T
c = dot(X, X_T.conj())
c *= np.true_divide(1, fact)
return c.squeeze()
```


np.sum():

```
@array_function_dispatch(_sum_dispatcher)
def sum(a, axis=None, dtype=None, out=None, keepdims=np._NoValue,
        initial=np._NoValue, where=np._NoValue):
    """
```

Sum of array elements over a given axis.

Parameters

a : array_like

Elements to sum.

axis : None or int or tuple of ints, optional

Axis or axes along which a sum is performed. The default, axis=None, will sum all of the elements of the input array. If axis is negative it counts from the last to the first axis.

.. versionadded:: 1.7.0

If axis is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

dtype : dtype, optional

The type of the returned array and of the accumulator in which the elements are summed. The dtype of `a` is used by default unless `a` has an integer dtype of less precision than the default platform integer. In that case, if `a` is signed then the platform integer is used while if `a` is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : ndarray, optional

Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

keepdims : bool, optional

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.

If the default value is passed, then `keepdims` will not be passed through to the `sum` method of sub-classes of `ndarray`, however any non-default value will be. If the sub-class' method does not implement `keepdims` any exceptions will be raised.

initial : scalar, optional

Starting value for the sum. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.15.0

where : array_like of bool, optional

Elements to include in the sum. See `~numpy.ufunc.reduce` for details.

.. versionadded:: 1.17.0

Returns

sum_along_axis : ndarray

An array with the same shape as `a`, with the specified axis removed. If `a` is a 0-d array, or if `axis` is `None`, a scalar is returned. If an output array is specified, a reference to `out` is returned.

See Also

ndarray.sum : Equivalent method.

add.reduce : Equivalent functionality of `add`.

cumsum : Cumulative sum of array elements.

trapz : Integration of array values using the composite trapezoidal rule.

mean, average

Notes

Arithmetic is modular when using integer types, and no error is raised on overflow.

The sum of an empty array is the neutral element 0:

```
>>> np.sum([])
0.0
```

For floating point numbers the numerical precision of sum (and `np.add.reduce`) is in general limited by directly adding each number individually to the result causing rounding errors in every step.

However, often numpy will use a numerically better approach (partial pairwise summation) leading to improved precision in many use-cases.

This improved precision is always provided when no `axis` is given.

When `axis` is given, it will depend on which axis is summed.

Technically, to provide the best speed possible, the improved precision is only used when the summation is along the fast axis in memory.

Note that the exact precision may vary depending on other parameters.

In contrast to NumPy, Python's `math.fsum` function uses a slower but more precise approach to summation.

Especially when summing a large number of lower precision floating point numbers, such as `float32`, numerical errors can become significant.

In such cases it can be advisable to use `dtype="float64"` to use a higher precision for the output.

Examples

```
>>> np.sum([0.5, 1.5])
2.0
>>> np.sum([0.5, 0.7, 0.2, 1.5], dtype=np.int32)
1
>>> np.sum([[0, 1], [0, 5]])
6
>>> np.sum([[0, 1], [0, 5]], axis=0)
array([0, 6])
>>> np.sum([[0, 1], [0, 5]], axis=1)
array([1, 5])
>>> np.sum([[0, 1], [np.nan, 5]], where=[False, True], axis=1)
array([1., 5.]
```

If the accumulator is too small, overflow occurs:

```
>>> np.ones(128, dtype=np.int8).sum(dtype=np.int8)
-128
```

You can also start the sum with a value other than zero:

```
>>> np.sum([10], initial=5)
15
"""
```

```
if isinstance(a, _gentype):
    # 2018-02-25, 1.15.0
    warnings.warn(
        "Calling np.sum(generator) is deprecated, and in the future will give a
different result. "
        "Use np.sum(np.fromiter(generator)) or the python sum builtin
instead.",
        DeprecationWarning, stacklevel=3)

    res = _sum_(a)
    if out is not None:
        out[...] = res
    return out
    return res

return _wrapreduction(a, np.add, 'sum', axis, dtype, out,
    keepdims=keepdims,
        initial=initial, where=where)
```

np.matmul():

```
def matmul(x1, x2, *args, **kwargs): # real signature unknown; NOTE:
unreliably restored from __doc__
    """
```

```
    matmul(x1, x2, /, out=None, *, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])
```

Matrix product of two arrays.

Parameters

x1, x2 : array_like

Input arrays, scalars not allowed.

out : ndarray, optional

A location into which the result is stored. If provided, it must have a shape that matches the signature `(n,k),(k,m)->(n,m)`. If not provided or None, a freshly-allocated array is returned.

**kwargs

For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

.. versionadded:: 1.16

Now handles ufunc kwargs

Returns

y : ndarray

The matrix product of the inputs.

This is a scalar only when both x1, x2 are 1-d vectors.

Raises

ValueError

If the last dimension of `a` is not the same size as the second-to-last dimension of `b`.

If a scalar value is passed in.

See Also

vdot : Complex-conjugating dot product.

tensordot : Sum products over arbitrary axes.

einsum : Einstein summation convention.

dot : alternative matrix product with different broadcasting rules.

Notes

The behavior depends on the arguments in the following way.

- If both arguments are 2-D they are multiplied like conventional matrices.
- If either argument is N-D, $N > 2$, it is treated as a stack of matrices residing in the last two indexes and broadcast accordingly.
- If the first argument is 1-D, it is promoted to a matrix by prepending a 1 to its dimensions. After matrix multiplication the prepended 1 is removed.
- If the second argument is 1-D, it is promoted to a matrix by appending a 1 to its dimensions. After matrix multiplication the appended 1 is removed.

`np.matmul` differs from `np.dot` in two important ways:

- Multiplication by scalars is not allowed, use `np.multiply` instead.
- Stacks of matrices are broadcast together as if the matrices were elements, respecting the signature `(n,k),(k,m) -> (n,m)`:

```
>>> a = np.ones([9, 5, 7, 4])
>>> c = np.ones([9, 5, 4, 3])
>>> np.dot(a, c).shape
(9, 5, 7, 9, 5, 3)
>>> np.matmul(a, c).shape
(9, 5, 7, 3)
>>> # n is 7, k is 4, m is 3
```

The `matmul` function implements the semantics of the `@` operator introduced in Python 3.5 following PEP465.

Examples

For 2-D arrays it is the matrix product:

```
>>> a = np.array([[1, 0],
...               [0, 1]])
>>> b = np.array([[4, 1],
...               [2, 2]])
>>> np.matmul(a, b)
array([[4, 1],
       [2, 2]])
```

For 2-D mixed with 1-D, the result is the usual.

```
>>> a = np.array([[1, 0],
...               [0, 1]])
>>> b = np.array([1, 2])
>>> np.matmul(a, b)
array([1, 2])
>>> np.matmul(b, a)
```

```
array([1, 2])
```

Broadcasting is conventional for stacks of arrays

```
>>> a = np.arange(2 * 2 * 4).reshape((2, 2, 4))
>>> b = np.arange(2 * 2 * 4).reshape((2, 4, 2))
>>> np.matmul(a,b).shape
(2, 2, 2)
>>> np.matmul(a, b)[0, 1, 1]
98
>>> sum(a[0, 1, :] * b[0, :, 1])
98
```

Vector, vector returns the scalar inner product, but neither argument is complex-conjugated:

```
>>> np.matmul([2j, 3j], [2j, 3j])
(-13+0j)
```

Scalar multiplication raises an error.

```
>>> np.matmul([1,2], 3)
Traceback (most recent call last):
...
ValueError: matmul: Input operand 1 does not have enough dimensions ...

.. versionadded:: 1.10.0
"""
pass
```

`np.linalg.inv()`:

```
@array_function_dispatch(_unary_dispatcher)
def inv(a):
    """
    Compute the (multiplicative) inverse of a matrix.

    Given a square matrix `a`, return the matrix `ainv` satisfying
    ``dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])``.

    Parameters
    -----
    a : (..., M, M) array_like
        Matrix to be inverted.

    Returns
    -----
    ainv : (..., M, M) ndarray or matrix
        (Multiplicative) inverse of the matrix `a`.

    Raises
    -----
    LinAlgError
        If `a` is not square or inversion fails.

    Notes
    -----
    .. versionadded:: 1.8.0

    Broadcasting rules apply, see the `numpy.linalg` documentation for
    details.

    Examples
    -----
    >>> from numpy.linalg import inv
    >>> a = np.array([[1., 2.], [3., 4.]])
    >>> ainv = inv(a)
    >>> np.allclose(np.dot(a, ainv), np.eye(2))
    True
    >>> np.allclose(np.dot(ainv, a), np.eye(2))
    True

    If a is a matrix object, then the return value is a matrix as well:

    >>> ainv = inv(np.matrix(a))
    >>> ainv
    matrix([[ -2. ,  1. ],
            [ 1.5, -0.5]])
```

Inverses of several matrices can be computed at once:

```
>>> a = np.array([[[1., 2.], [3., 4.]], [[1, 3], [3, 5]]])
>>> inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]],
      [[-1.25,  0.75],
       [ 0.75, -0.25]])

"""
a, wrap = _makearray(a)
_assert_stacked_2d(a)
_assert_stacked_square(a)
t, result_t = _commonType(a)

signature = 'D->D' if isComplexType(t) else 'd->d'
extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
ainv = _umath_linalg.inv(a, signature=signature, extobj=extobj)
return wrap(ainv.astype(result_t, copy=False))
```


np.log():

```
def log(x, *args, **kwargs): # real signature unknown; NOTE: unreliably
restored from __doc__
    """
```

```
    log(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])
```

Natural logarithm, element-wise.

The natural logarithm `log` is the inverse of the exponential function, so that `log(exp(x)) = x`. The natural logarithm is logarithm in base `e`.

Parameters

`x` : array_like

Input value.

`out` : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

`where` : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result.

Elsewhere, the `out` array will retain its original value.

Note that if an uninitialized `out` array is created via the default `out=None`, locations within it where the condition is False will remain uninitialized.

`**kwargs`

For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

Returns

`y` : ndarray

The natural logarithm of `x`, element-wise.

This is a scalar if `x` is a scalar.

See Also

`log10`, `log2`, `log1p`, `emath.log`

Notes

Logarithm is a multivalued function: for each `x` there is an infinite number of `z` such that `exp(z) = x`. The convention is to return the `z` whose imaginary part lies in `[-pi, pi]`.

For real-valued input data types, `log` always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the `invalid` floating point error flag.

For complex-valued input, `log` is a complex analytical function that has a branch cut `[-inf, 0]` and is continuous from above on it. `log` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

- .. [1] M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", 10th printing, 1964, pp. 67. <http://www.math.sfu.ca/~cbm/aands/>
- .. [2] Wikipedia, "Logarithm". <https://en.wikipedia.org/wiki/Logarithm>

Examples

```
>>> np.log([1, np.e, np.e**2, 0])
array([ 0.,  1.,  2., -Inf])
"""
pass
```

np.linalg.det():

```
@array_function_dispatch(_unary_dispatcher)
def det(a):
    """
    Compute the determinant of an array.

    Parameters
    -----
    a : (... , M, M) array_like
        Input array to compute determinants for.

    Returns
    -----
    det : (...) array_like
        Determinant of `a`.

    See Also
    -----
    slogdet : Another way to represent the determinant, more suitable
        for large matrices where underflow/overflow may occur.

    Notes
    -----
    .. versionadded:: 1.8.0

    Broadcasting rules apply, see the `numpy.linalg` documentation for
    details.

    The determinant is computed via LU factorization using the LAPACK
    routine ``z/dgetrf``.

    Examples
    -----
    The determinant of a 2-D array  $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$  is  $ad - bc$ :

    >>> a = np.array([[1, 2], [3, 4]])
    >>> np.linalg.det(a)
    -2.0 # may vary

    Computing determinants for a stack of matrices:

    >>> a = np.array([ [1, 2], [3, 4]], [ [1, 2], [2, 1]], [ [1, 3], [3, 1]] )
    >>> a.shape
    (3, 2, 2)
    >>> np.linalg.det(a)
    array([-2., -3., -8.])

    """
```

```
a = asarray(a)
_assert_stacked_2d(a)
_assert_stacked_square(a)
t, result_t = _commonType(a)
signature = 'D->D' if isComplexType(t) else 'd->d'
r = _umath_linalg.det(a, signature=signature)
r = r.astype(result_t, copy=False)
return r
```

`np.argmax()`:

```
@array_function_dispatch(_argmax_dispatcher)
```

```
def argmax(a, axis=None, out=None):
```

```
    """
```

Returns the indices of the maximum values along an axis.

Parameters

`a` : array_like

Input array.

`axis` : int, optional

By default, the index is into the flattened array, otherwise along the specified axis.

`out` : array, optional

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

Returns

`index_array` : ndarray of ints

Array of indices into the array. It has the same shape as ``a.shape`` with the dimension along ``axis`` removed.

See Also

`ndarray.argmax`, `argmin`

`amax` : The maximum value along a given axis.

`unravel_index` : Convert a flat index into an index tuple.

`take_along_axis` : Apply ``np.expand_dims(index_array, axis)`` from `argmax` to an array as if by calling `max`.

Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> a = np.arange(6).reshape(2,3) + 10
```

```
>>> a
```

```
array([[10, 11, 12],
       [13, 14, 15]])
```

```
>>> np.argmax(a)
```

```
5
```

```
>>> np.argmax(a, axis=0)
```

```
array([1, 1, 1])
```

```
>>> np.argmax(a, axis=1)
```

```
array([2, 2])
```

Indexes of the maximal elements of a N-dimensional array:

```
>>> ind = np.unravel_index(np.argmax(a, axis=None), a.shape)
>>> ind
(1, 2)
>>> a[ind]
15
```

```
>>> b = np.arange(6)
>>> b[1] = 5
>>> b
array([0, 5, 2, 3, 4, 5])
>>> np.argmax(b) # Only the first occurrence is returned.
1
```

```
>>> x = np.array([[4,2,3], [1,0,3]])
>>> index_array = np.argmax(x, axis=-1)
>>> # Same as np.max(x, axis=-1, keepdims=True)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-
```

1)

```
array([[4],
       [3]])
>>> # Same as np.max(x, axis=-1)
>>> np.take_along_axis(x, np.expand_dims(index_array, axis=-1), axis=-
```

1).squeeze(axis=-1)

```
array([4, 3])
```

```
"""
```

```
return _wrapfunc(a, 'argmax', axis=axis, out=out)
```

np.log2():

```
def log2(x, *args, **kwargs): # real signature unknown; NOTE: unreliably
restored from __doc__
    """
```

```
    log2(x, /, out=None, *, where=True, casting='same_kind', order='K',
dtype=None, subok=True[, signature, extobj])
```

Base-2 logarithm of `x`.

Parameters

x : array_like

Input values.

out : ndarray, None, or tuple of ndarray and None, optional

A location into which the result is stored. If provided, it must have a shape that the inputs broadcast to. If not provided or None, a freshly-allocated array is returned. A tuple (possible only as a keyword argument) must have length equal to the number of outputs.

where : array_like, optional

This condition is broadcast over the input. At locations where the condition is True, the `out` array will be set to the ufunc result. Elsewhere, the `out` array will retain its original value.

Note that if an uninitialized `out` array is created via the default

`out=None`, locations within it where the condition is False will remain uninitialized.

**kwargs

For other keyword-only arguments, see the :ref:`ufunc docs <ufuncs.kwargs>`.

Returns

y : ndarray

Base-2 logarithm of `x`.

This is a scalar if `x` is a scalar.

See Also

log, log10, log1p, math.log2

Notes

.. versionadded:: 1.3.0

Logarithm is a multivalued function: for each `x` there is an infinite number of `z` such that $2^{**z} = x$. The convention is to return the `z` whose imaginary part lies in $[-\pi, \pi]$.

For real-valued input data types, `log2` always returns real output.

For each value that cannot be expressed as a real number or infinity,

it yields ``nan`` and sets the `invalid` floating point error flag.

For complex-valued input, `log2` is a complex analytical function that has a branch cut $[-\infty, 0]$ and is continuous from above on it. `log2` handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

Examples

```
>>> x = np.array([0, 1, 2, 2**4])
```

```
>>> np.log2(x)
```

```
array([-Inf,  0.,  1.,  4.])
```

```
>>> xi = np.array([0+1.j, 1, 2+0.j, 4.j])
```

```
>>> np.log2(xi)
```

```
array([ 0.+2.26618007j, 0.+0.j      , 1.+0.j      , 2.+2.26618007j])
```

```
"""
```

```
pass
```



```
from sklearn.model_selection import StratifiedKFold
```

StratifiedKFold():

```
def __init__(self, n_splits=5, shuffle=False, random_state=None):
    super().__init__(n_splits, shuffle, random_state)
```

StratifiedKFold().split():

```
def split(self, X, y, groups=None):
    """Generate indices to split data into training and test set.

    Parameters
    -----
    X : array-like, shape (n_samples, n_features)
        Training data, where n_samples is the number of samples
        and n_features is the number of features.

        Note that providing ``y`` is sufficient to generate the splits and
        hence ``np.zeros(n_samples)`` may be used as a placeholder for
        ``X`` instead of actual training data.

    y : array-like, shape (n_samples,)
        The target variable for supervised learning problems.
        Stratification is done based on the y labels.

    groups : object
        Always ignored, exists for compatibility.

    Yields
    -----
    train : ndarray
        The training set indices for that split.

    test : ndarray
        The testing set indices for that split.

    Notes
    -----
    Randomized CV splitters may return different results for each call of
    split. You can make the results identical by setting ``random_state``
    to an integer.
    """
    y = check_array(y, ensure_2d=False, dtype=None)
    return super().split(X, y, groups)
```

Neither numpy nor sklearn.model_selection.StratifiedKFold(i.e. internal packages):

ndarray.shape:

```
shape = property(lambda self: object(), lambda self, v: None, lambda self: None)
# default
```

property() at the right of equal sign:

```
def __init__(self, fget=None, fset=None, fdel=None, doc=None): # known
special case of property.__init__
```

```
"""
```

```
Property attribute.
```

```
fget
```

```
function to be used for getting an attribute value
```

```
fset
```

```
function to be used for setting an attribute value
```

```
fdel
```

```
function to be used for del'ing an attribute
```

```
doc
```

```
docstring
```

Typical use is to define a managed attribute x:

```
class C(object):
```

```
    def getx(self): return self._x
```

```
    def setx(self, value): self._x = value
```

```
    def delx(self): del self._x
```

```
    x = property(getx, setx, delx, "I'm the 'x' property.")
```

Decorators make defining new properties or modifying existing ones easy:

```
class C(object):
```

```
    @property
```

```
    def x(self):
```

```
        "I am the 'x' property."
```

```
        return self._x
```

```
    @x.setter
```

```
    def x(self, value):
```

```
        self._x = value
```

```
    @x.deleter
```

```
    def x(self):
```

```
        del self._x
```

```
# (copied from class doc)
```

```
"""
```

```
pass
```

--

`dict.setdefault():`

```
def setdefault(self, *args, **kwargs): # real signature unknown
    """
    Insert key with a value of default if key is not in the dictionary.

    Return the value for key if key is in the dictionary, else default.
    """
    pass
```