Final Project

Full Name: Yuezhong Yan

NJIT UCID: yy444

Email: yy444@njit.edu

Option 1

Category 3: Decision Tree

Category 5: Naïve Bayes

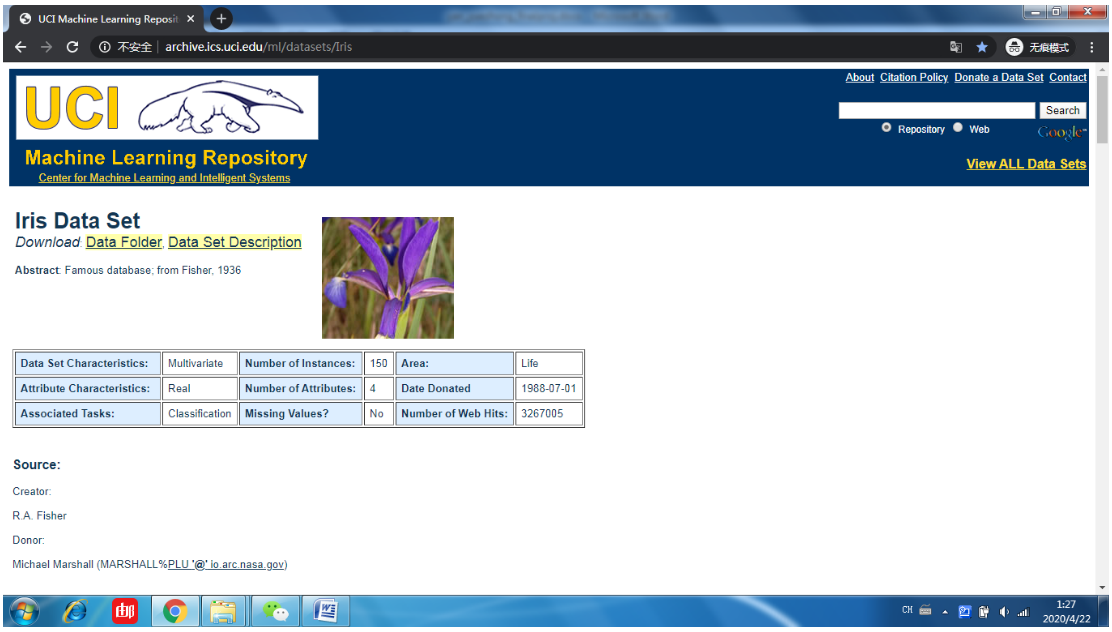
Programming Language: Python

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

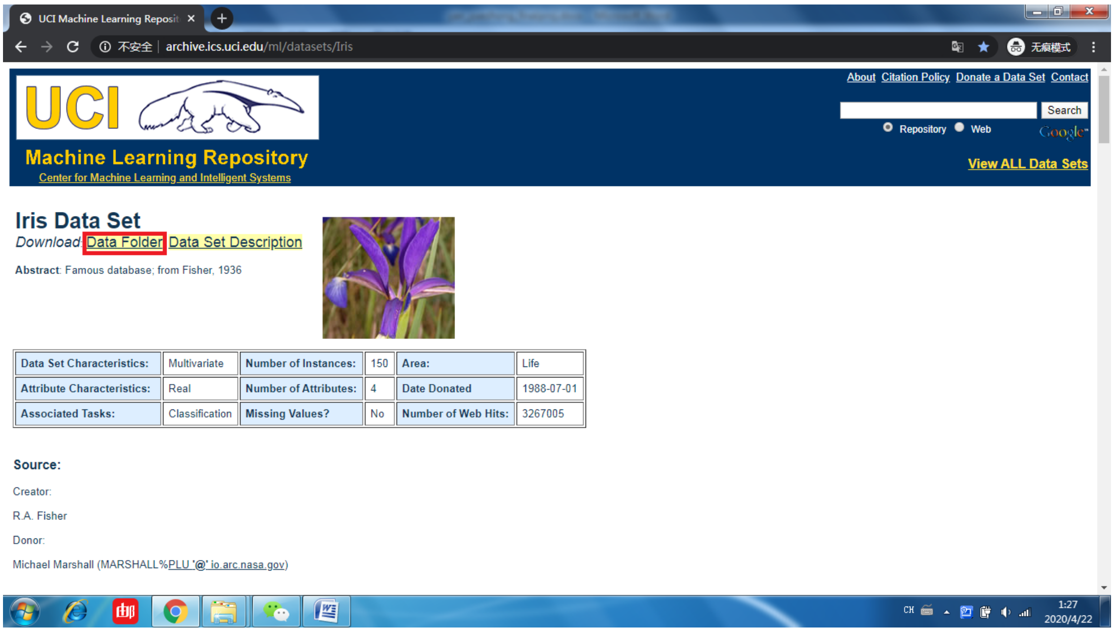
This dataset is about the classification of iris plant based on sepal length, sepal width, petal length, and petal width. It has the total of 150 instances consisting of 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2. 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.

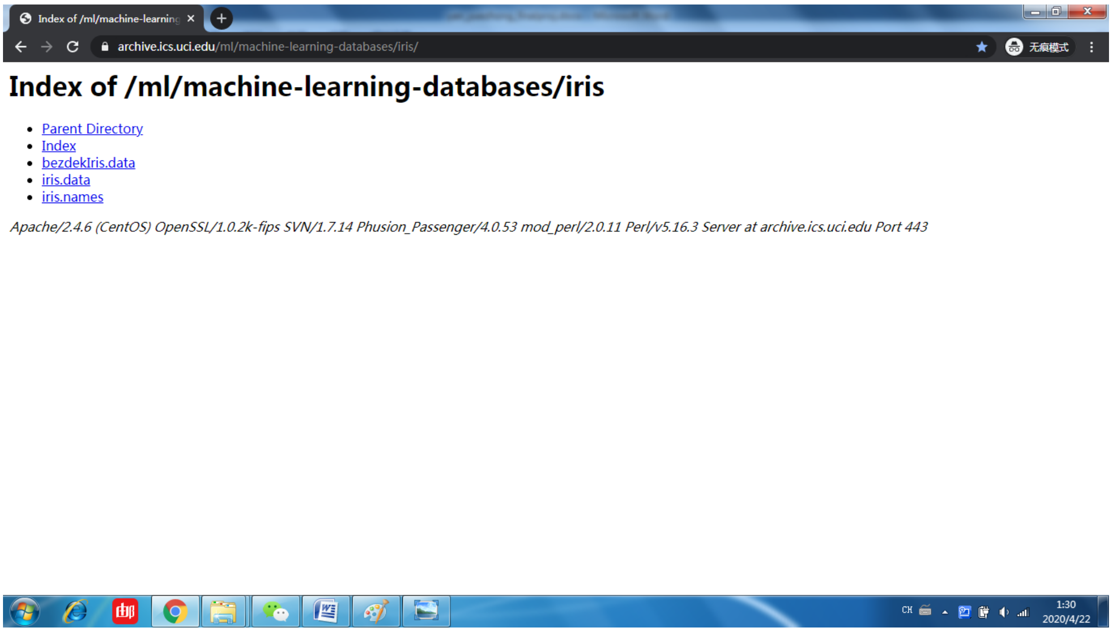
To download the dataset, I first click on the URL above, and I see the following Ibsite:



And then I click on “Data Folder”.

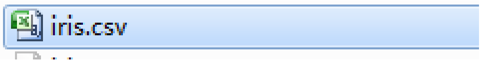


I will enter the Ibpage for downloading. In this Ibpage, “bezdekIris.data” and “iris.data” have exactly same contents. You can choose either one to download. “Index” is some timestamps about this data. “iris.names” is just some basic information about this dataset. I do not need “Index” or “iris.names”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



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
```

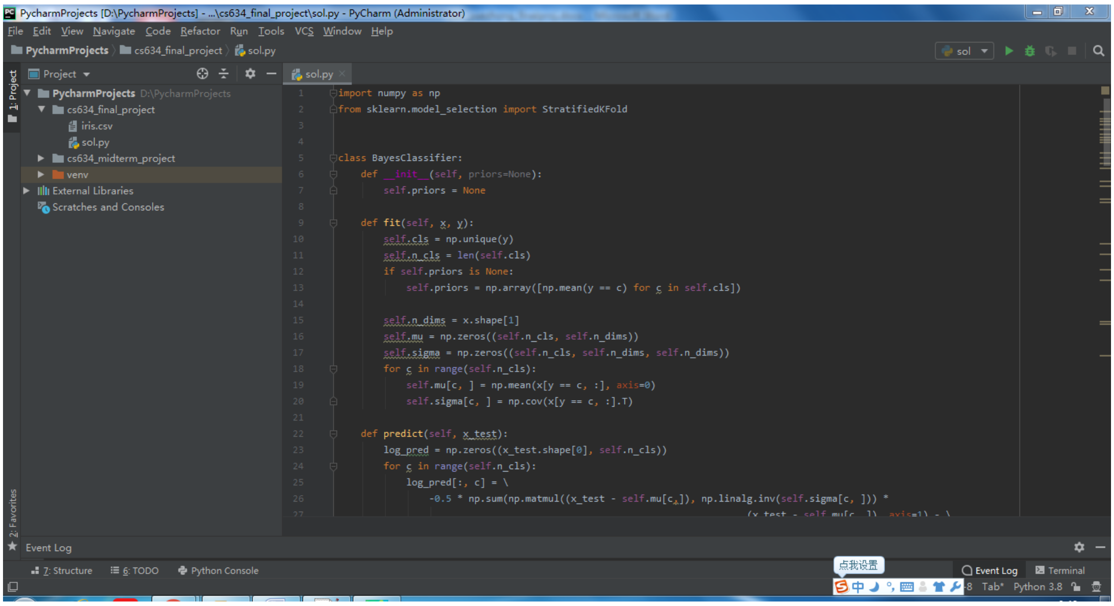
```
4.8,3.4,1.6,0.2,Iris-setosa
4.8,3.0,1.4,0.1,Iris-setosa
4.3,3.0,1.1,0.1,Iris-setosa
5.8,4.0,1.2,0.2,Iris-setosa
5.7,4.4,1.5,0.4,Iris-setosa
5.4,3.9,1.3,0.4,Iris-setosa
5.1,3.5,1.4,0.3,Iris-setosa
5.7,3.8,1.7,0.3,Iris-setosa
5.1,3.8,1.5,0.3,Iris-setosa
5.4,3.4,1.7,0.2,Iris-setosa
5.1,3.7,1.5,0.4,Iris-setosa
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
```

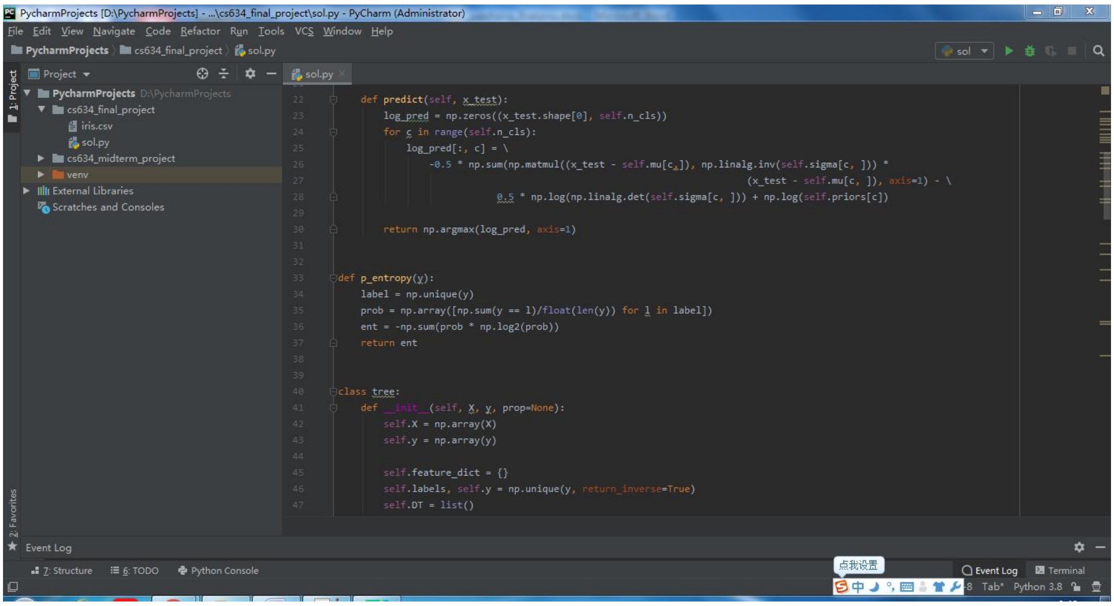
```
6.3,2.5,4.9,1.5,Iris-versicolor
6.1,2.8,4.7,1.2,Iris-versicolor
6.4,2.9,4.3,1.3,Iris-versicolor
6.6,3.0,4.4,1.4,Iris-versicolor
6.8,2.8,4.8,1.4,Iris-versicolor
6.7,3.0,5.0,1.7,Iris-versicolor
6.0,2.9,4.5,1.5,Iris-versicolor
5.7,2.6,3.5,1.0,Iris-versicolor
5.5,2.4,3.8,1.1,Iris-versicolor
5.5,2.4,3.7,1.0,Iris-versicolor
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
```

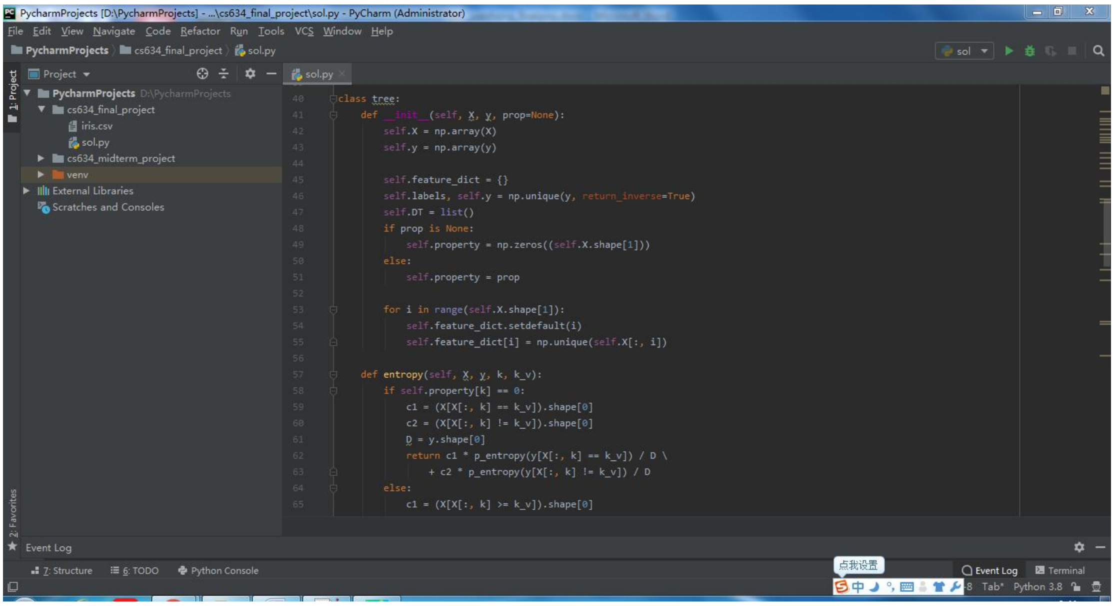
```
6.3,2.8,5.1,1.5,Iris-virginica
6.1,2.6,5.6,1.4,Iris-virginica
7.7,3.0,6.1,2.3,Iris-virginica
6.3,3.4,5.6,2.4,Iris-virginica
6.4,3.1,5.5,1.8,Iris-virginica
6.0,3.0,4.8,1.8,Iris-virginica
6.9,3.1,5.4,2.1,Iris-virginica
6.7,3.1,5.6,2.4,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
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.



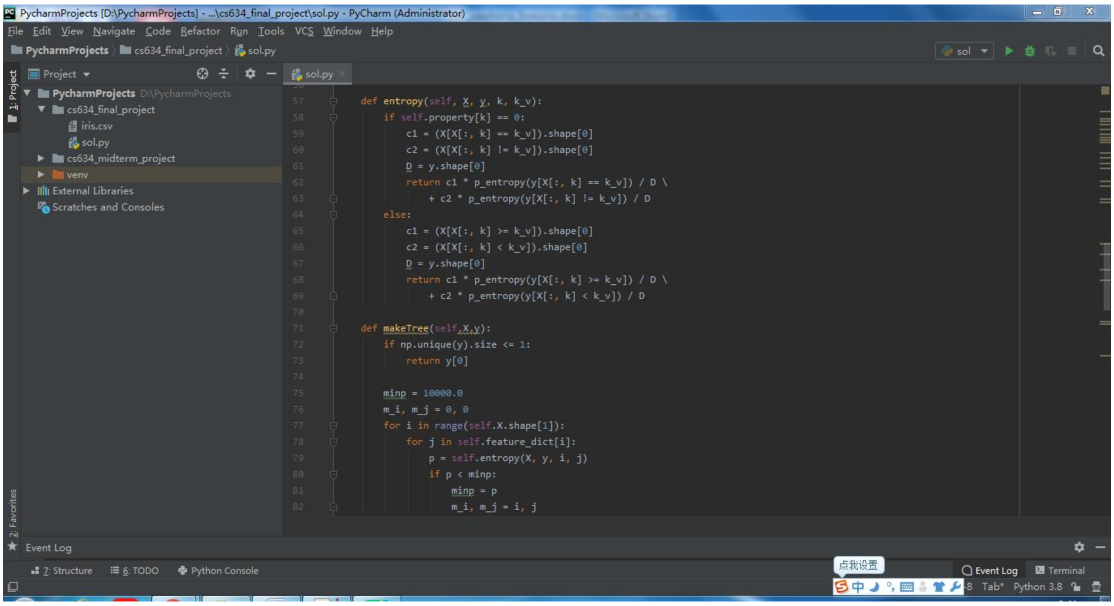



PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
sol.py
class tree:
    def __init__(self, X, y, prop=None):
        self.X = np.array(X)
        self.y = np.array(y)

        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

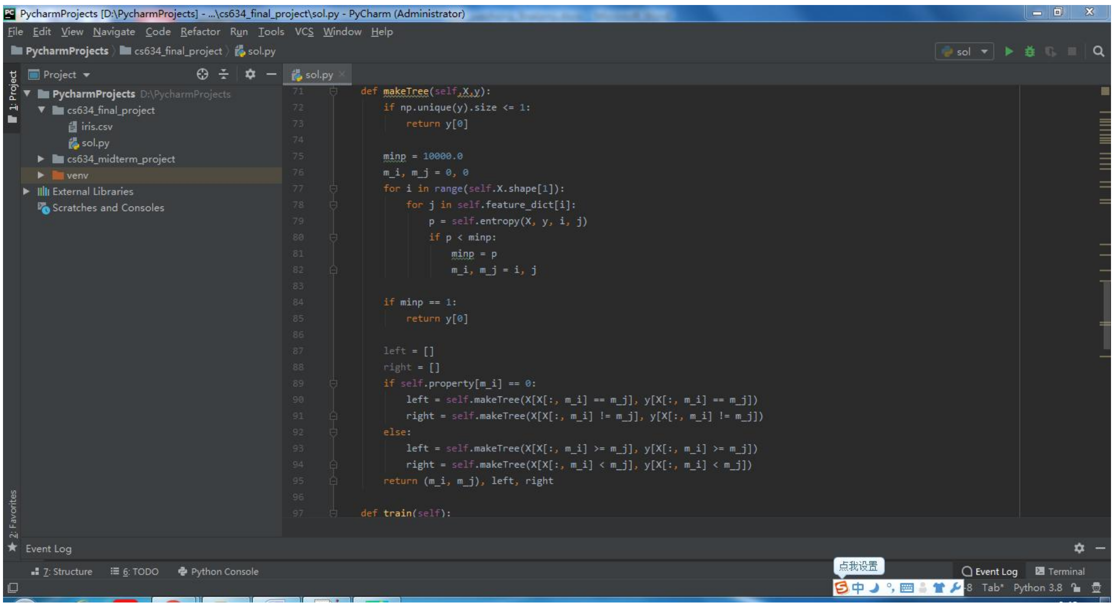
        for i in range(self.X.shape[1]):
            self.feature_dict.setdefault(i)
            self.feature_dict[i] = np.unique(self.X[:, i])

    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)
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

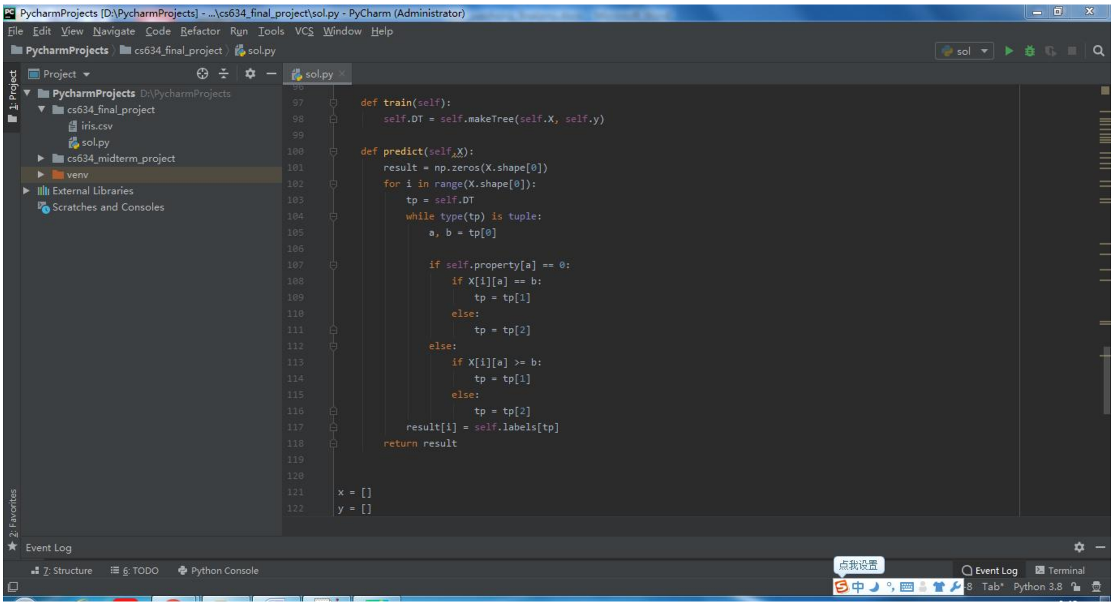

PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
sol.py
    def makeTree(self,X,y):
        if np.unique(y).size <= 1:
            return y[0]

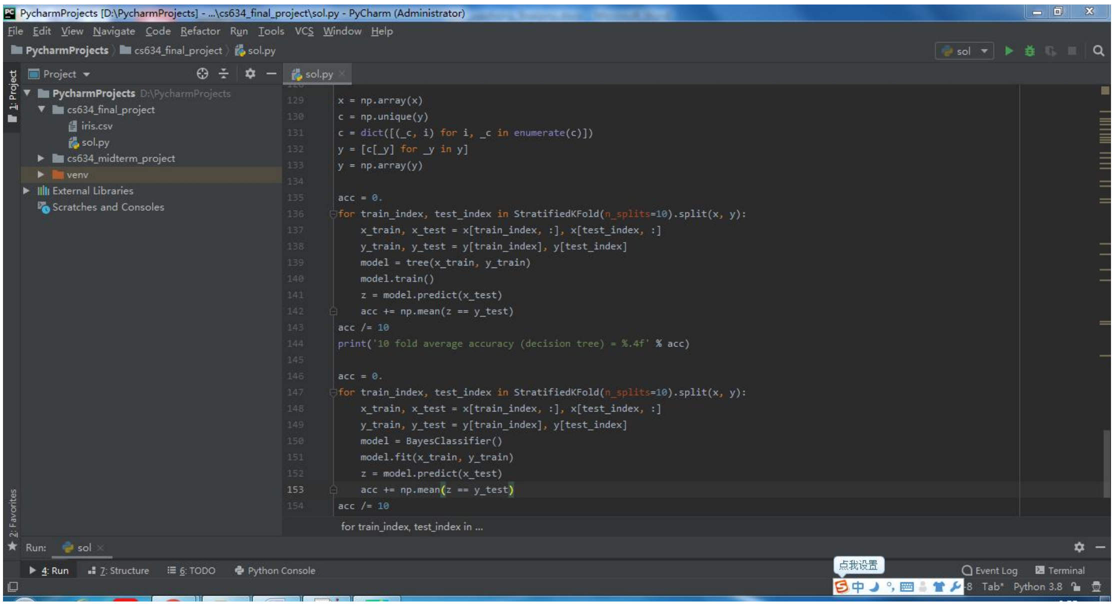
        minp = 10000.0
        m_i, m_j = 0, 0
        for i in range(self.X.shape[1]):
            for j in self.feature_dict[i]:
                p = self.entropy(X, y, i, j)
                if p < minp:
                    minp = p
                    m_i, m_j = i, j

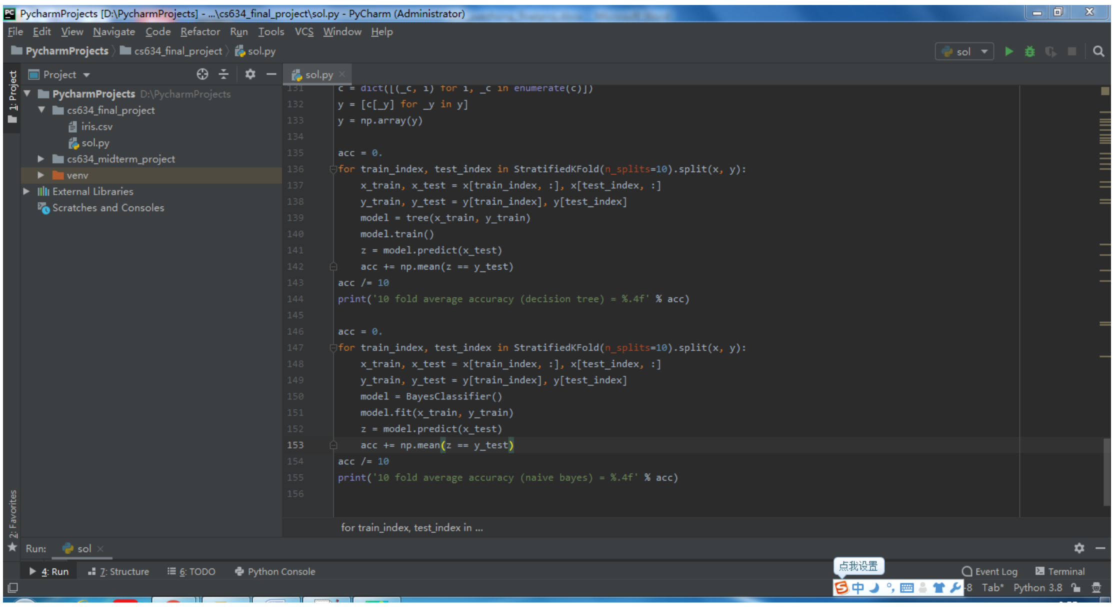
        if minp == 1:
            return y[0]

        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

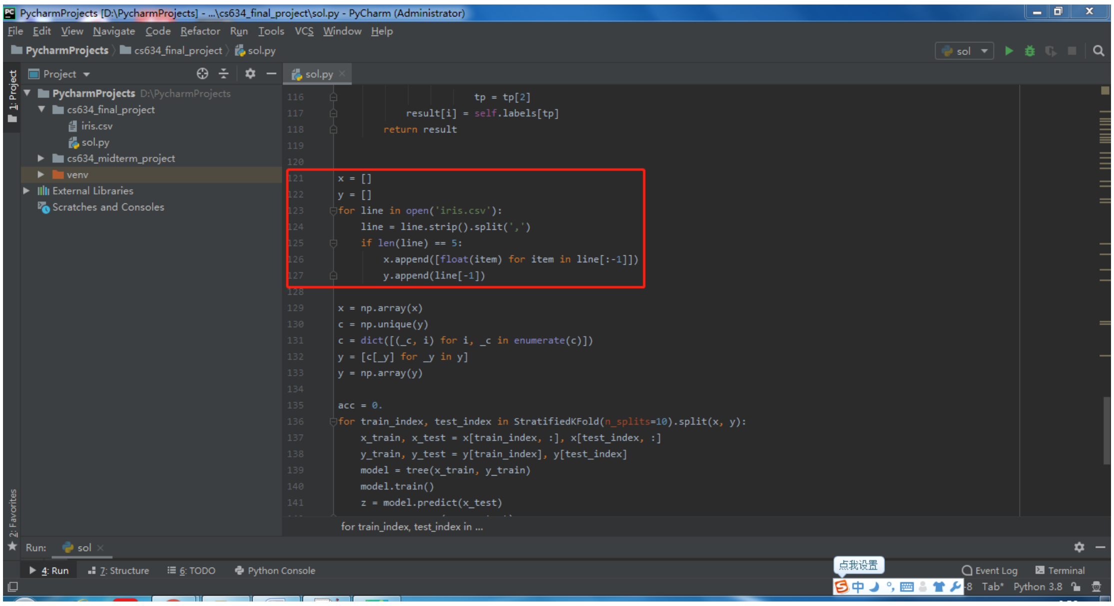
    def train(self):


PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
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 = []

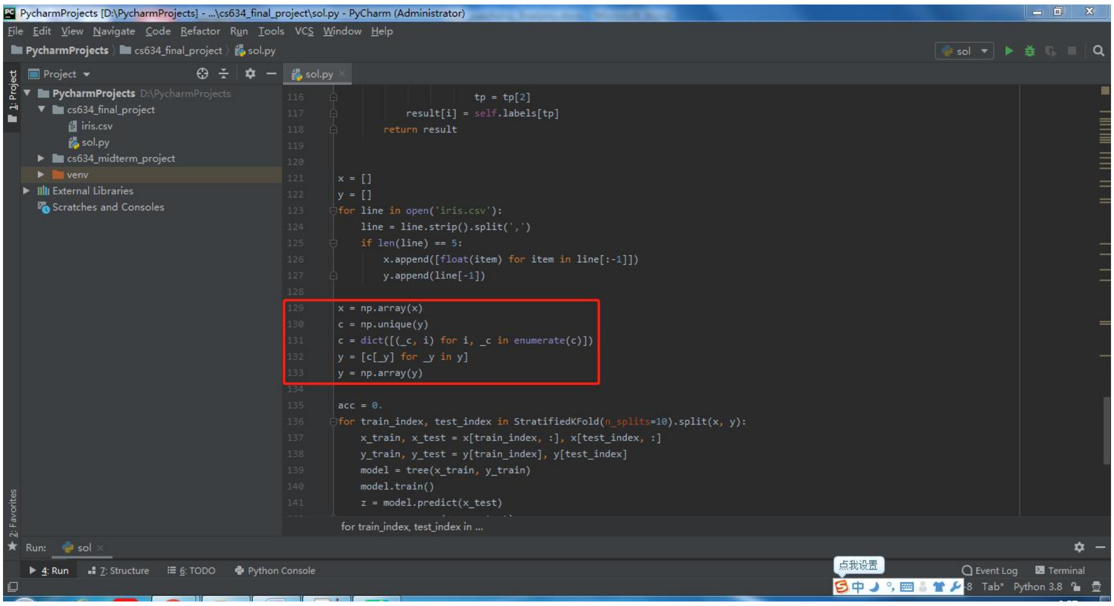

PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
x = np.array(x)
c = np.unique(y)
c = dict([(_c, i) for i, _c in enumerate(c)])
y = [c[_y] for _y in y]
y = np.array(y)
acc = 0.
for train_index, test_index in StratifiedKFold(n_splits=10).split(x, y):
    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


PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
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)

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.

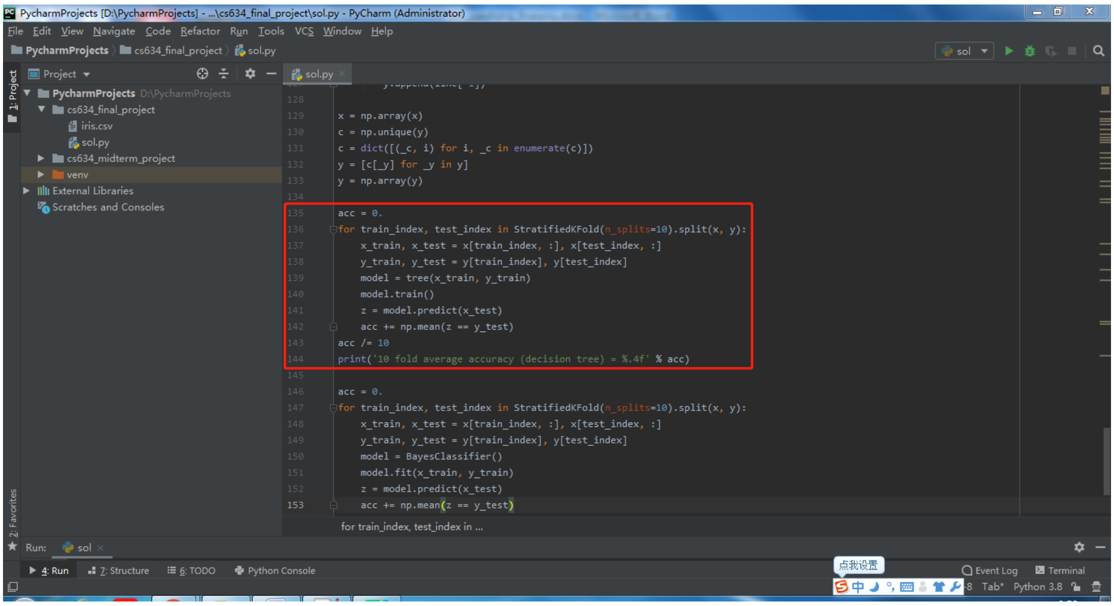


2. Then I put these two variables into another data type ndarray, where x is 2d array and y is 1d array. I put x directly into ndarray, remove the duplicates from y, replace the 3 classifications Iris Setosa, Iris Versicolor, and Iris Virginica from y with 0, 1, 2, respectively.

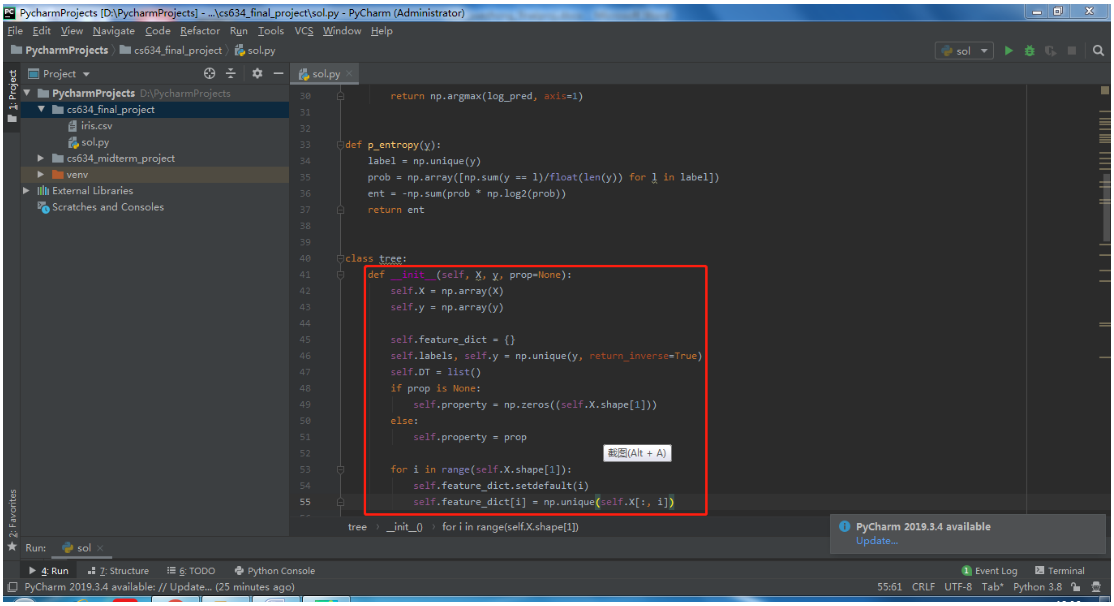


Decision Tree:

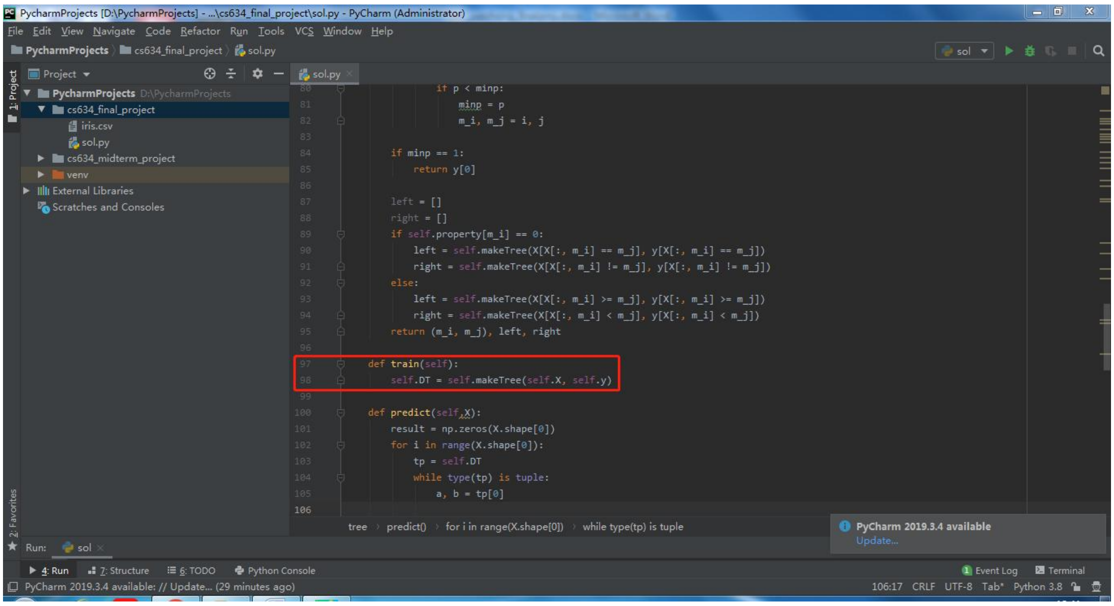
1. I use StratifiedKFold to split the data into 10 subsets to perform 10-fold cross validation. I then declare and initialize training data x_train and y_train, as Ill as testing data x_test and y_test, construct Tree() object with x_train and y_train, train the model, predict x_test to predict y, and 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.



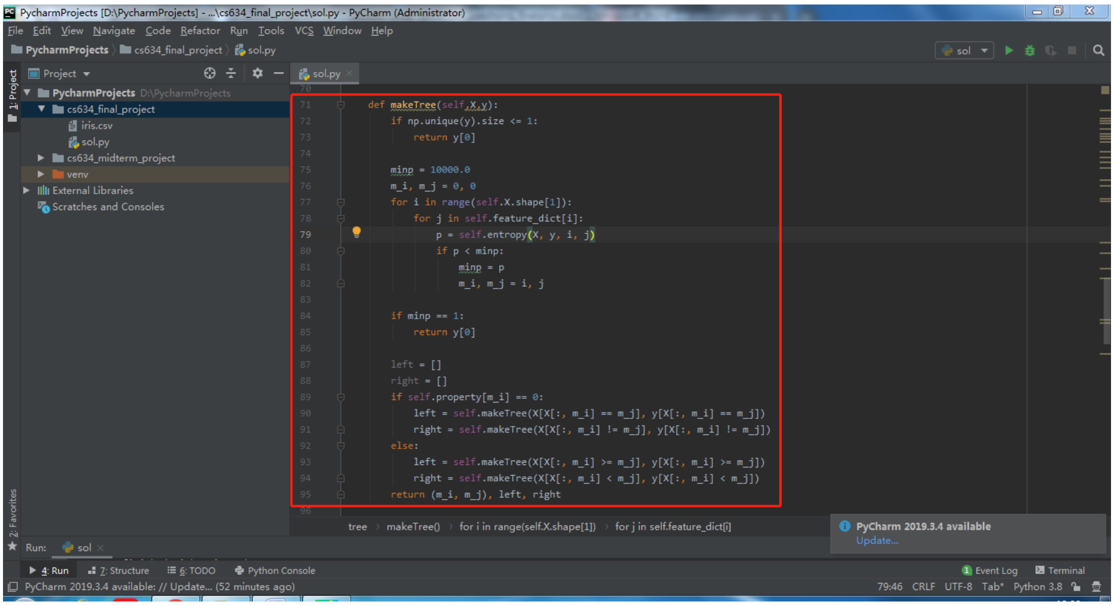
2. Tree().__init__() is a constructor for Decision Tree object. X creates a 2d array and stores the first 4 attributes of the original dataset. y also creates a 2d array and stores the last attribute(classification) of the original dataset. feature_dict stores feature values for each column from X. labels 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.



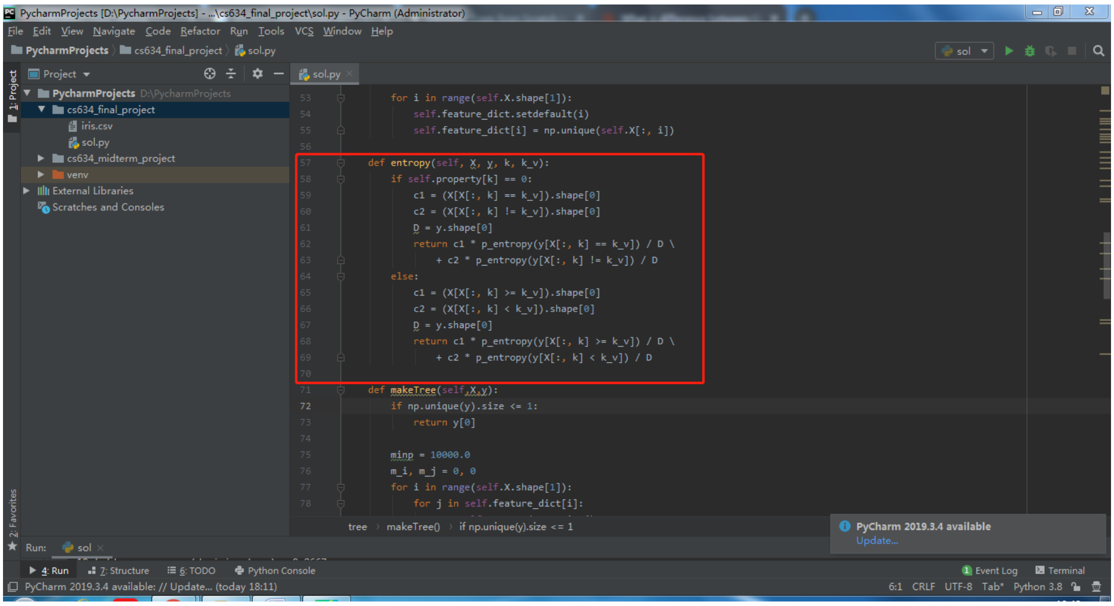
3. train() trains the data by making a decision tree with makeTree() method.



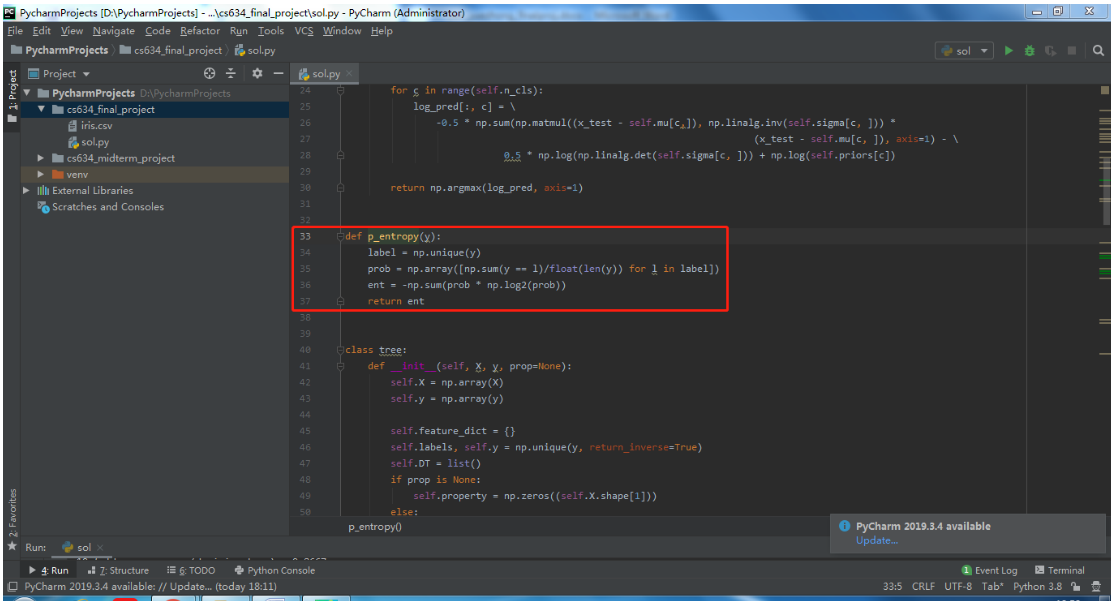
4. In makeTree() method, I calculate the information gain, or the reduction in entropy, by keeping track of the 4 attributes from X and feature values for each attribute from feature_dict, so that I can get the best information gain. Here, m_i stores the labels for the last attribute(classification) 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.



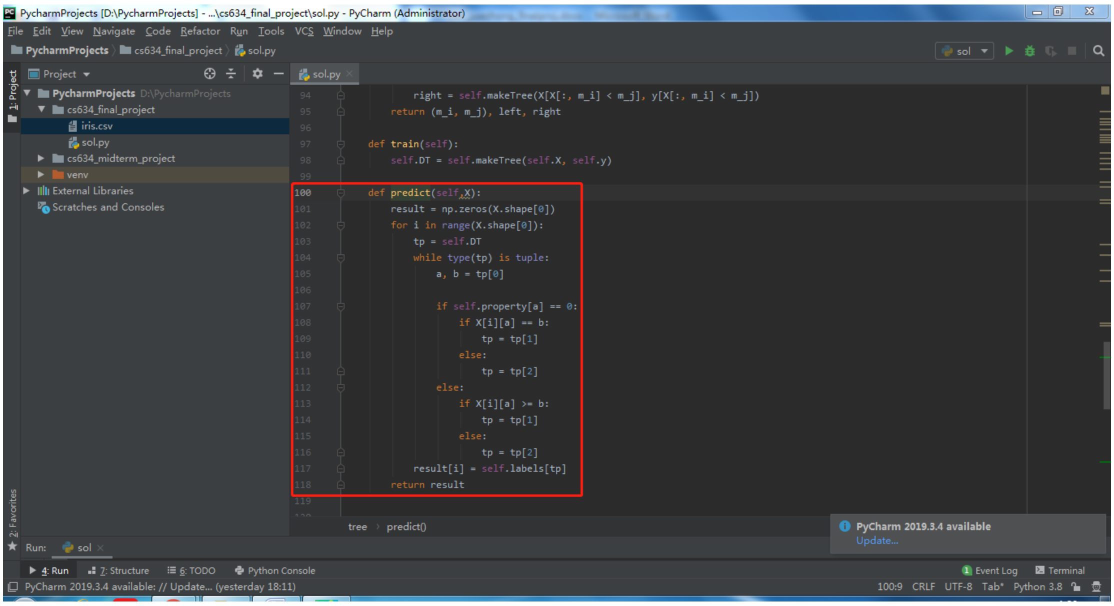
5. entropy() at line 79 from makingTree() calculates entropies for parent, left-child node, and right-child node with the label for classification and the feature value for each label. Its implementation is shown below.



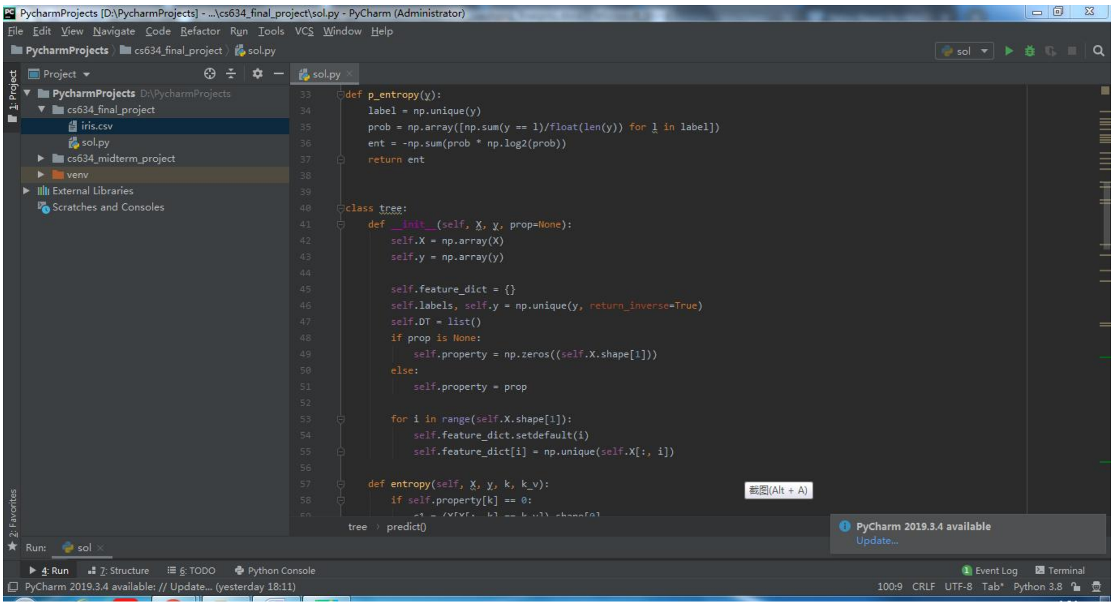
6. p_entropy() at lines 62—63 and lines 68—69 calculates the real entropy and behaves as a helper function. The formula is Entropy $= -\sum_{j=1}^{c} P_j \log_2(P_j)$, where P_j is proportion of samples that belongs to class c for a specific node. The implementation is shown below

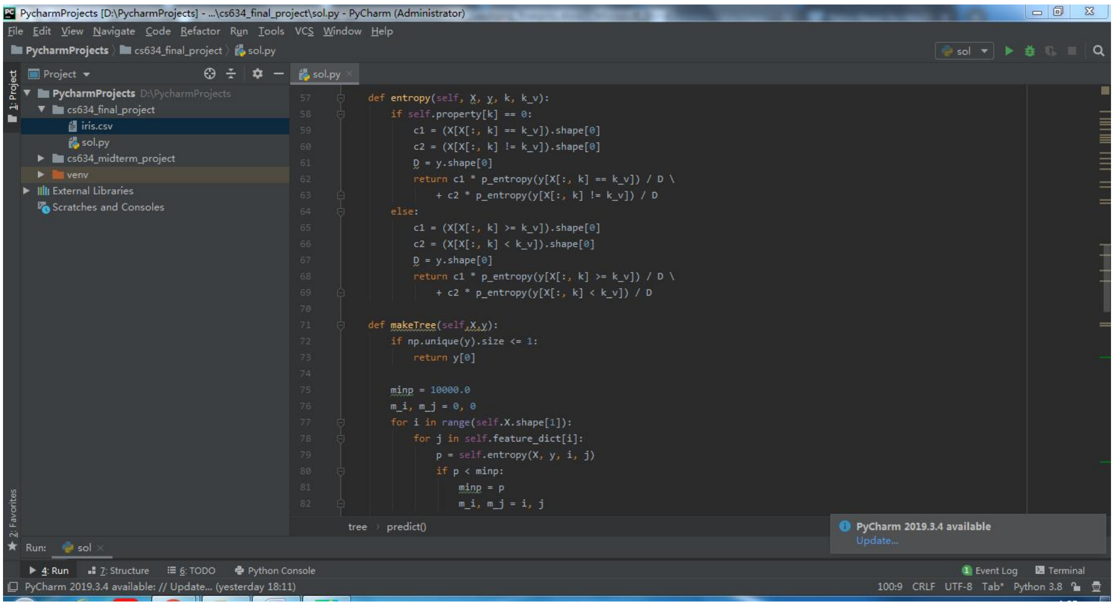


7. predict() predicts based on x_test, with taking 5 lines for each classification in each iteration. I first initialize result ndarray as zeros. Then I take the entire decision tree as tp to take the predicted classification and the corresponding featured value. 4th column of X, compare the 4th column 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.



8. The whole class is shown below.

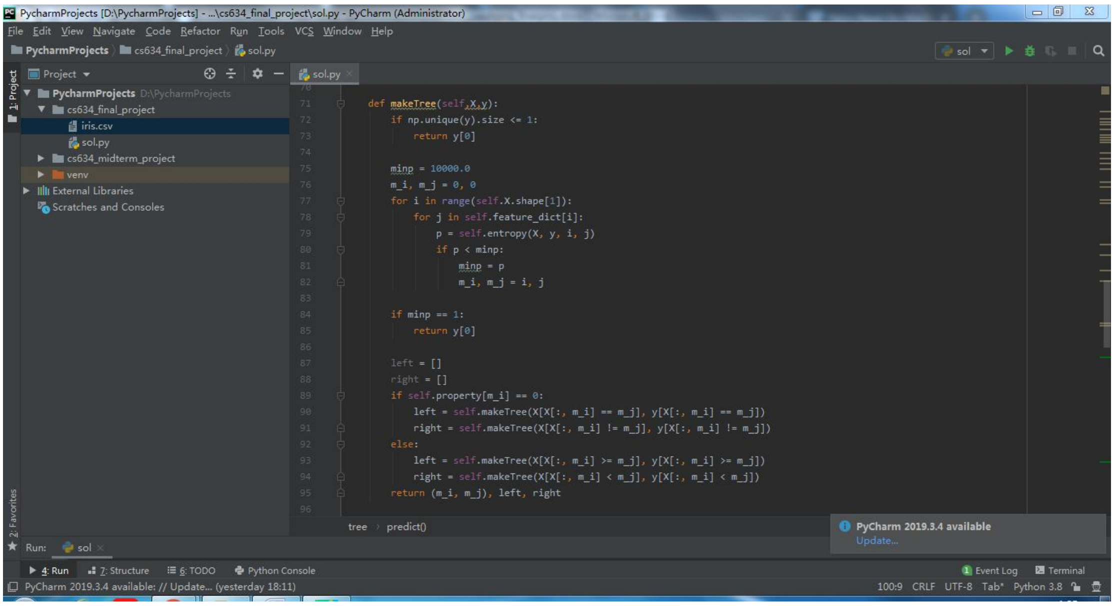




PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PycharmProjects cs634_final_project sol.py
Project
PycharmProjects D:\PycharmProjects
cs634_final_project
iris.csv
sol.py
cs634_midterm_project
venv
External Libraries
Scratches and Consoles
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
tree › predict()
PyCharm 2019.3.4 available
Update...
Run: sol
4: Run 7: Structure 6: TODO Python Console
Event Log Terminal
PyCharm 2019.3.4 available: // Update... (yesterday 18:11)
100:9 CRLF UTF-8 Tab* Python 3.8

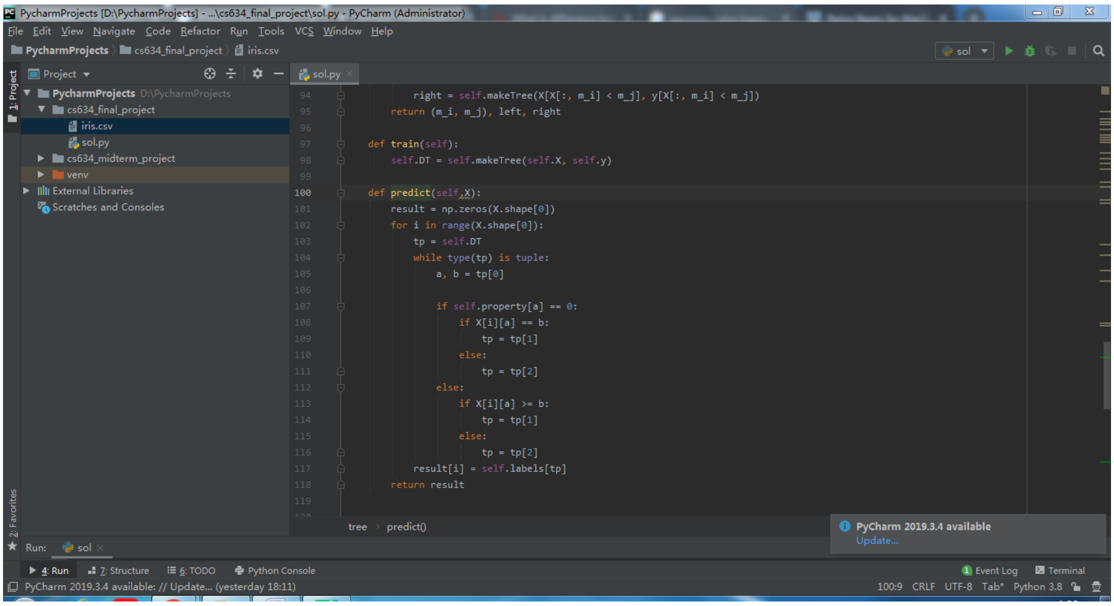


PycharmProjects [D:\PycharmProjects] - ...\cs634_final_project\sol.py - PyCharm (Administrator)
File Edit View Navigate Code Refactor Run Tools VCS Window Help
PycharmProjects cs634_final_project sol.py
Project
PycharmProjects D:\PycharmProjects
cs634_final_project
iris.csv
sol.py
cs634_midterm_project
venv
External Libraries
Scratches and Consoles
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
tree › predict()
PyCharm 2019.3.4 available
Update...
Run: sol
4: Run 7: Structure 6: TODO Python Console
Event Log Terminal
PyCharm 2019.3.4 available: // Update... (yesterday 18:11)
100:9 CRLF UTF-8 Tab* Python 3.8



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
Project
PycharmProjects D:\PycharmProjects
cs634_final_project
iris.csv
sol.py
cs634_midterm_project
venv
External Libraries
Scratches and Consoles
sol.py
            right = self.makeTree(X[X[:, m_i] < m_j], y[X[:, m_i] < m_j])
        return (m_i, m_j), left, right

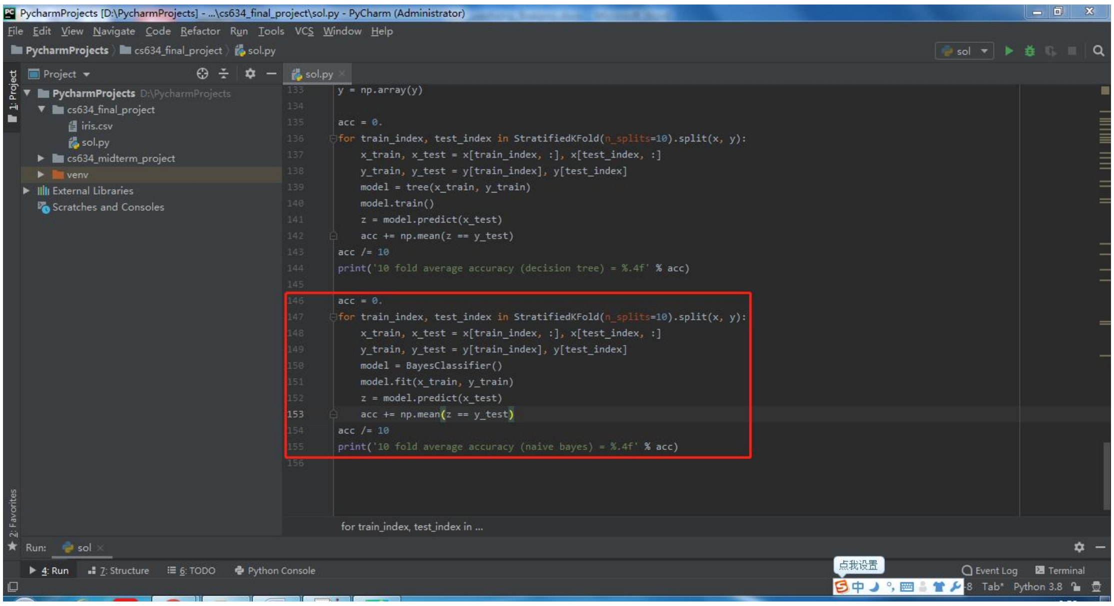
    def train(self):
        self.DT = self.makeTree(self.X, self.y)

    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]

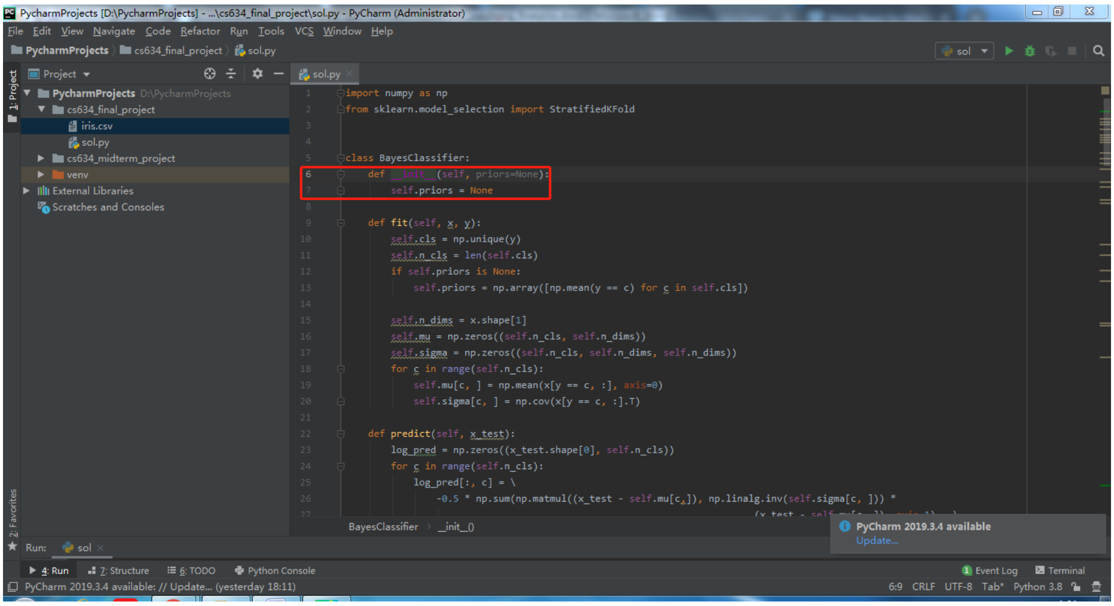
                if self.property[a] == 0:
                    if X[i][a] == b:
                        tp = tp[1]
                    else:
                        tp = tp[2]
                else:
                    if X[i][a] >= b:
                        tp = tp[1]
                    else:
                        tp = tp[2]
            result[i] = self.labels[tp]
        return result
tree › predict()
PyCharm 2019.3.4 available
Update...
Run: sol
4: Run 7: Structure 6: TODO Python Console
Event Log Terminal
PyCharm 2019.3.4 available: // Update... (yesterday 18:11)
100:9 CRLF UTF-8 Tab* Python 3.8

Naïve Bayes:

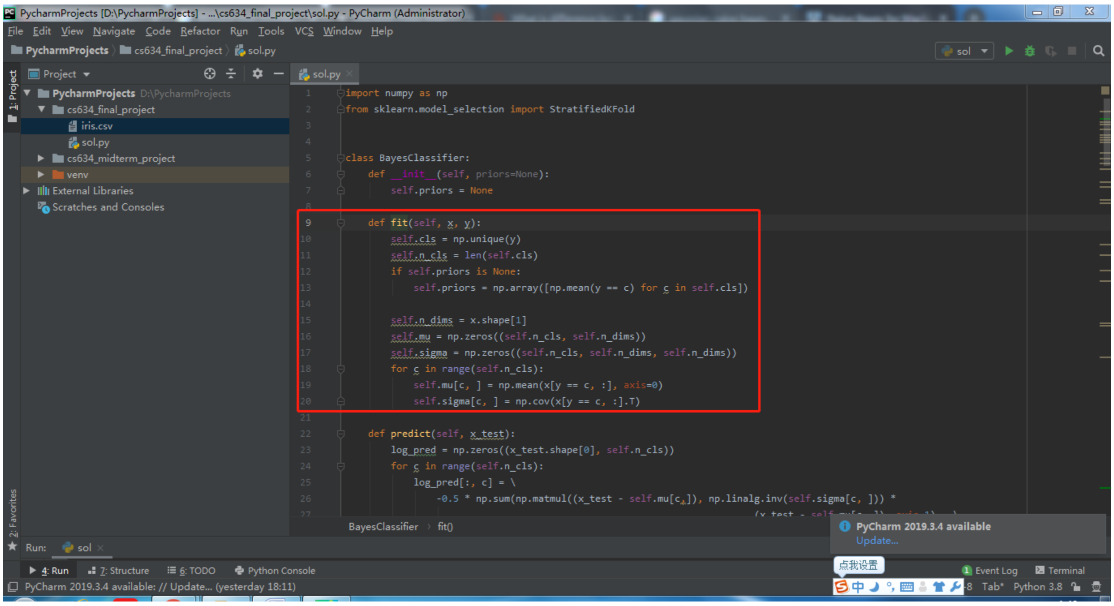
1. I use StratifiedKFold to split the data into 10 subsets to perform 10-fold cross validation. I then declare and initialize training data x_train and y_train, as Ill as testing data x_test and y_test, construct BayesClassfier() object, fit x_train and y_train into model, predict x_test to predict y, 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.



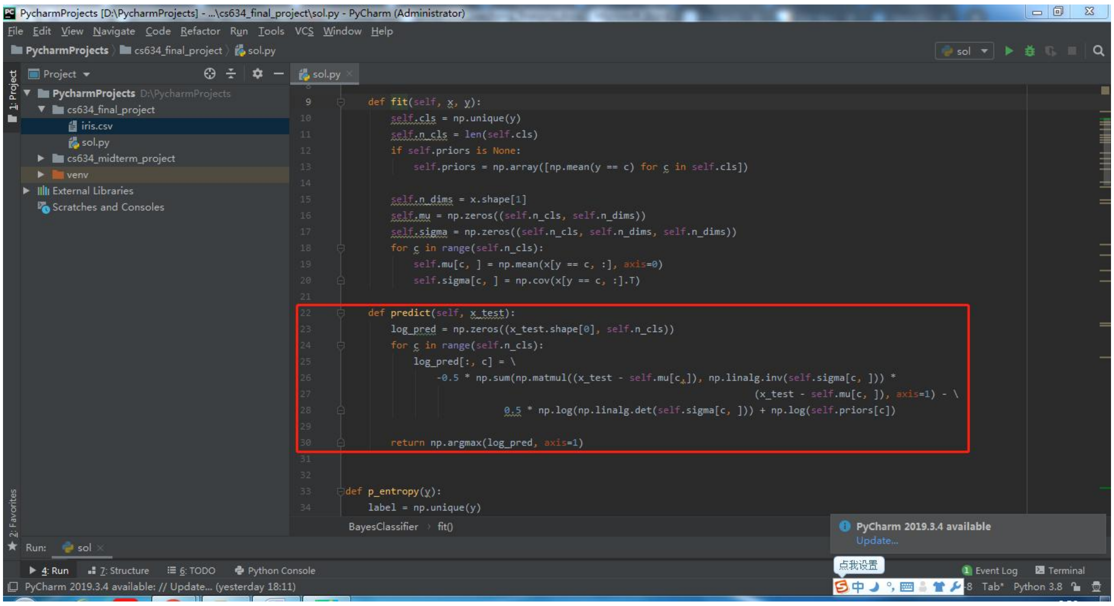
2. BayesClassfier().__init__() constructor is a constructor for Naïve Bayes object. The function is shown below.



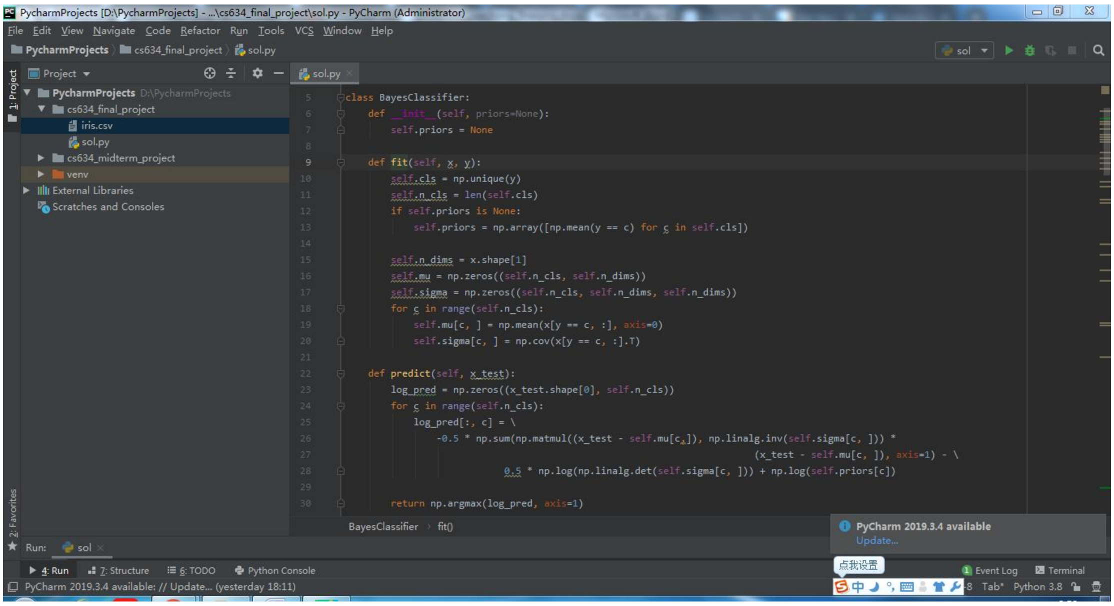
3. fit() function fits x_train and y_train. It first removes duplicates of classifications and stores the size of classifications, n_cls, after removal. Then it sets the average probability for these 3 classifications, which is 1/3. I store the second element from the dimension of x, n_dims. I create 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 – mean(x)^2)). Here, I need variance, so I just simply remove sqrt() from the formula.



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)*standard variance)) * exp(-((x-mean^2)/(2*standard variance^2))).

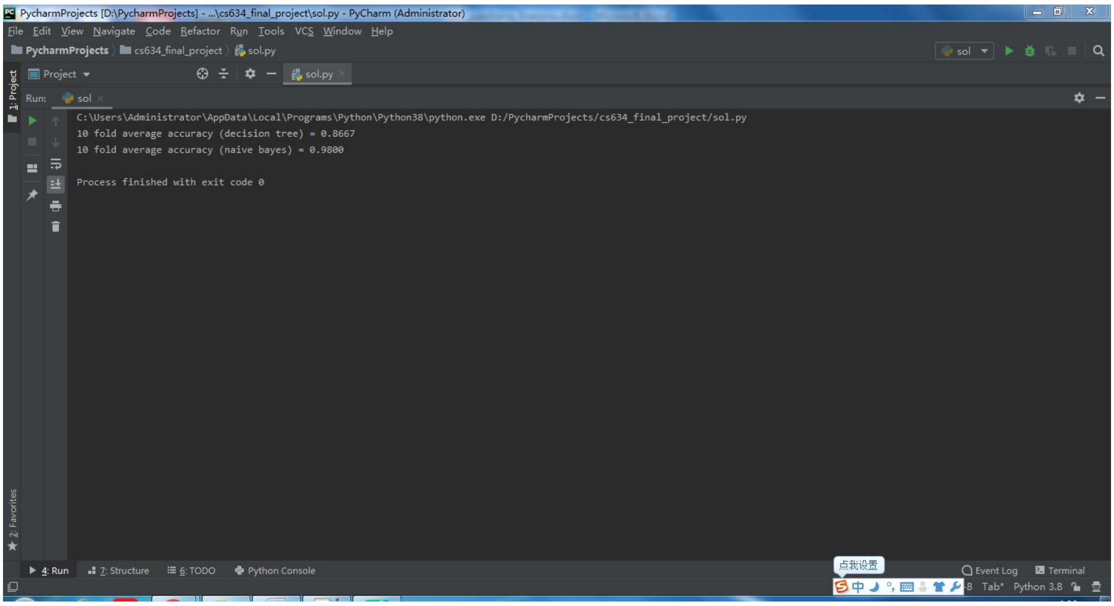


5. The whole class of BayesClassfier() is shown below.

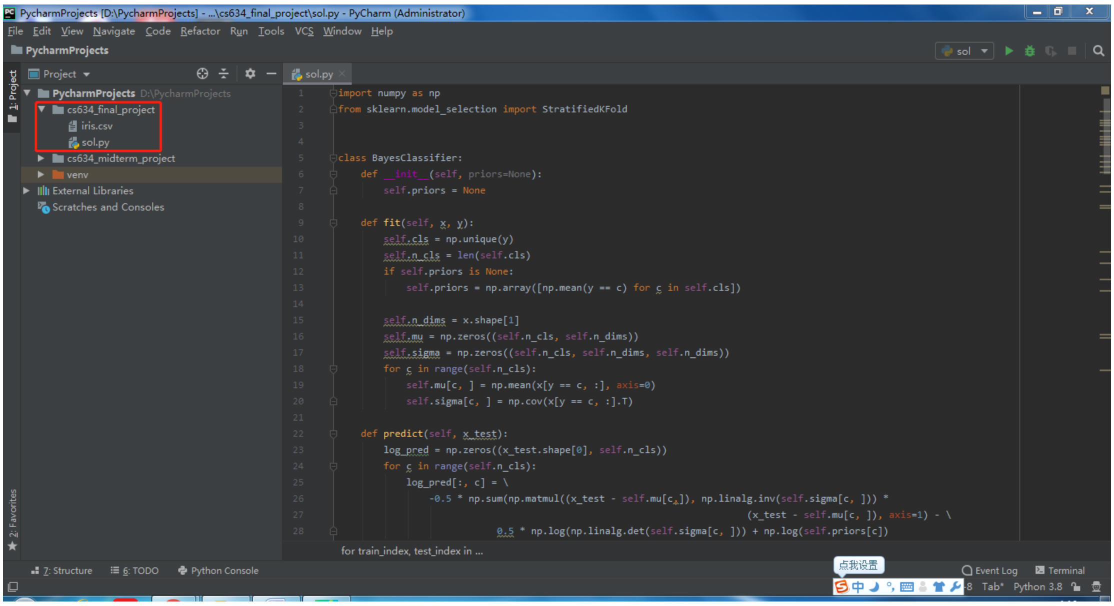


Output:

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.



Structure: How I organize the dataset and the project.



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 vieId 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.asanyarray(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`, hoIver 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 : Iighted 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, fIights=None,
        aIights=None):
    """
    Estimate a covariance matrix, given data and Iights.

    Covariance indicates the level to which two variables vary together.
    If I examine N-dimensional samples, :math:`X = [x_1, x_2, ... x_N]^T`,
    then the covariance matrix element :math:`C_{ij}` is the covariance of
    :math:`x_i` and :math:`x_j`. The element :math:`C_{ii}` is the variance
    of :math:`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 >= 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
        `fIights` and `aIights` are specified, and ``ddof=0`` will return
        the simple average. See the notes for the details. The default value
        is ``None``.

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

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

typically large for observations considered "important" and smaller for
observations considered less "important". If ``ddof=0`` the array of
Iights 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 = fIights`` and ``a = aIights`` for brevity. The
steps to compute the Iighted 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, :math:`x_0` and :math:`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 :math:`x_0` increases while :math:`x_1` decreases. The
covariance
matrix shows this clearly:

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

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

Note that element :math:`C_{0,1}`, which shows the correlation betIen
:math:`x_0` and :math:`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 Iights
    w = None
    if fIights is not None:
        fIights = np.asarray(fIights, dtype=float)
        if not np.all(fIights == np.around(fIights)):
            raise TypeError(
                "fIights must be integer")
        if fIights.ndim > 1:
            raise RuntimeError(
                "cannot handle multidimensional fIights")
        if fIights.shape[0] != X.shape[1]:
            raise RuntimeError(
                "incompatible numbers of samples and fIights")
        if any(fIights < 0):
            raise ValueError(
                "fIights cannot be negative")
        w = fIights
    if aIights is not None:
        aIights = np.asarray(aIights, dtype=float)
        if aIights.ndim > 1:
            raise RuntimeError(
                "cannot handle multidimensional aIights")
        if aIights.shape[0] != X.shape[1]:
            raise RuntimeError(
                "incompatible numbers of samples and aIights")
        if any(aIights < 0):
            raise ValueError(
                "aIights cannot be negative")
        if w is None:
            w = aIights
        else:
            w *= aIights

    avg, w_sum = average(X, axis=1, Iights=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 aIights 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`, hoIver 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. HoIver, 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 sloIr but more precise approach to summation.
Especially when summing a large number of loIr 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 alloId.
            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.

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

- Multiplication by scalars is not alloId, use ``*`` instead.
- Stacks of matrices are broadcast together as if the matrices
  Ire 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 Ill:

    >>> 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 [[a, b], [c, d]] 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, emath.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 `[-inf, 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 StratefiedKFold

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
```