تمرینات سری اول

سوال اول :

الف) طراحی دخت تصمیم با استفاده از information gain (IG)

در فرآیند طراحی درخت تصمیم با استفاده از الگوریتم مورد بحث ابتدا تابعی برای محاسبه آنتروپی به صورت زیر تعریف گردیده است:

```
def findEntropy(data, rows):
    yes = 0
    no = 0
    ans = -1
    idx = len(data[0]) - 1
    entropy = 0
    for i in rows:
        if data[i][idx] == 'Yes':
            yes = yes + 1
        else:
            no = no + 1

    x = yes/(yes+no)
    y = no/(yes+no)
    if x != 0 and y != 0:
        entropy = -1 * (x*math.log2(x) + y*math.log2(y))
    if x == 1:
        ans = 1
    if y == 1:
        ans = 0
    return entropy, ans
```

این تابع با دریافت یک مجموعه و تعداد سطر های آن، شروع به انجام محاسبات مربوط به IG میکند و نتیجه را در متغیر entropy ذخیره میکند.

در ادامه کد ، تابعی تحت عنوان findMaxGain تعریف گردیده است که به صورت زیر میباشد :

```
def findMaxGain(data, rows, columns):
    maxGain = 0
    retidx = -1
    entropy, ans = findEntropy(data, rows)
    if entropy == 0:
        return maxGain, retidx, ans
```

```
    for j in columns:
        mydict = {}
        idx = j
        for i in rows:
            key = data[i][idx]
            if key not in mydict:
                mydict[key] = 1
            else:
                mydict[key] = mydict[key] + 1

        gain = entropy

        for key in mydict:
            yes = 0
            no = 0
            for k in rows:
                if data[k][j] == key:
                    if data[k][-1] == 'Yes':
                        yes = yes + 1
                    else:
                        no = no + 1
            x = yes/(yes+no)
            y = no/(yes+no)
            if x != 0 and y != 0:
                gain += (mydict[key] * (x*math.log2(x) + y*math.log2(y)))/14
        if gain > maxGain:
            maxGain = gain
            retidx = j

    return maxGain, retidx, ans
```

این تابع با دریافت دسته داده ها و تعداد سطر و ستون ها شروع به محاسبات میکند . سپس با فراخوانی تابع findEntropy ، آنتروپی مجموعه مادر را محاسبه میکند.

سپس با استفاده از یک حلقه، مقادیری که در هر ستون میتوان داشته باشد را میشمارد. با شمارش این حالت ها و تعداد برچسب های مختلف برای هر کدام از این مقادیر با تعریف پارامتر gain و مقایسه آنها با هم ، ستونی که دارای بیشترین IG میباشد را گزارش میدهد. خروجی این تابع شامل بیشترین IG و ستون متناظر با آن میباشد.

حال در این مرحله با داشتن بیشترین IG و ویژگی مورد نظر میتوان وارد مرحله ایجاد یک درخت تصمیم شد. برای این منظور تابعی به صورت زیر توسعه یافته است:

```
def buildTree(data, rows, columns):
    maxGain, idx, ans = findMaxGain(X, rows, columns)
    root = {
```

```
        'decision': None,
        'decision_condition': None,
        'childs': []
    }

    if maxGain == 0:
        if ans == 1:
            root['decision_condition'] = 'Yes'
        else:
            root['decision_condition'] = 'No'
        return root

    root['decision_condition'] = attribute[idx]
    mydict = {}
    for i in rows:
        key = data[i][idx]
        if key not in mydict:
            mydict[key] = 1
        else:
            mydict[key] += 1

    newcolumns = copy.deepcopy(columns)
    newcolumns.remove(idx)
    for key in mydict:
        newrows = [i for i in rows if data[i][idx] == key]
        temp = buildTree(data, newrows, newcolumns)
        temp['decision'] = key
        root['childs'].append(temp)
    return root
```

این تابع با دریافت دسته داده ها و تعداد سطر و ستون های آن میتواند ساختار یک درخت تصمیم را ایجاد نماید.

این تابع در ابتدا با فراخوانی تابع findMaxGain و دریافت خروجی های این تابع ، اقدام به تعریف یک class شامل ۳ نوع داده ی ‘decision’ ‘decision condition’, ’childs’ میکند.

در مرحله بعد، با کنترل مقدار IG مطمئن میشویم که به تصمیم یا برگ رسیده ایم. در صورت عدم رسیدن به برگ، الگوریتم با گرفتن ویژگی با بیشترین IG شروع به ساخت زیرمجموعه هایی از مجموعه کلی میکند و ستون بررسی شده را حذف می نماید. درآخر یک مجموعه از نوع کلاس تعریف شده ، تشکیل میشود.

در مرحله بعدی یک تابع جدید برای چاپ نتایج تابع ساخت درخت تعریف میگردد. در واقع این تابع وظیفه مرتب نمودن و چاپ برگ ها و شاخه ها را بر عهده دارد.

```
def traverse(root, depth=0):
    indent = "   " * depth
    print(indent + str(root['decision']))
    print(indent + str(root['decision_condition']))

    n = len(root['childs'])
    if n > 0:
        for i in range(0, n):
            traverse(root['childs'][i], depth + 1)
```

حال با تعریف توابع مورد نیاز ، بدنه اصلی الگوریتم معرفی میگردد:

```
def calculate():
    rows = [i for i in range(0, 7)]
    columns = [i for i in range(0, 4)]
    root = buildTree(X, rows, columns)
    root['decision'] = 'Start'
    traverse(root)
    return root

root = calculate()
```

تابع بالا، با تعریف ستون و سطر ها و صدا کردن توابع buildTree و traverse اقدام به ساخت و چاپ درخت تصمیم گیری میکند. نتیجه اجرای کد مورد ارائه برای مجموعه مورد سوال به صورت زیر میباشد.

Start

Colour

Green

Toughness

Hard

Appearance

Smooth

No

Wrinkled

Yes

soft

Yes

Brown

No

Orange

Yes

ب)

در قسمت دوم با تغییر معیار از IG به Gain Ratio (GR) مراحل قبلی تکرار میشود.

برای این امر تابع قبلی findMaxGain به تابع زیر تغییر پیدا میکند و مراحل تکرار میشود.

```
def findMaxGainRatio(data, rows, columns):
    maxGainRatio = 0
    retidx = -1
    entropy, ans = findEntropy(data, rows)
    if entropy == 0:
        return maxGainRatio, retidx, ans

    for j in columns:
        mydict = {}
        idx = j
        for i in rows:
```

```
            key = data[i][idx]
            if key not in mydict:
                mydict[key] = 1
            else:
                mydict[key] = mydict[key] + 1

        information_gain = entropy
        split_information = 0

        for key in mydict:
            yes = 0
            no = 0
            for k in rows:
                if data[k][j] == key:
                    if data[k][-1] == 'Yes':
                        yes = yes + 1
                    else:
                        no = no + 1
            x = yes / (yes + no)
            y = no / (yes + no)

            if x != 0 and y != 0:
                information_gain += (mydict[key] * (x * math.log2(x) + y * math.log2(y))) / 7
               split_information += -1*(mydict[key] / 7) * math.log2(mydict[key] / 7)

        if split_information != 0:
            gain_ratio = information_gain / split_information
        else:
            gain_ratio = 0

        if gain_ratio > maxGainRatio:
            maxGainRatio = gain_ratio
            retidx = j

    return maxGainRatio, retidx, ans
```

در تابع جدید findMaxGainRatio با تعریف مفهوم split information مقدار GR متناسب با هر ستون به دست می آید.

خروجی بر حسب تغیرات جدید به صورت زیر است.

Start

Colour

Green

Toughness

Hard

Single-branch

t

No

c

No

soft

Yes

Brown

No

Orange

Yes

همان طور که مشاهده میشود، درخت تصمیم خروجی در شاخه اول قبل از این که به برگ برسد با اتمام ستون ها مواجه میشود و به همین دلیل در قسمت مشخص شده دچار مشکل میشود. حدس بنده درباره علت این موضوع کمبود ویژگی ها و تعداد کم داده هاست.

ج)

در این قسمت با معرفی تابع classify و اضافه کردن آن به هرکدام از دو کد بالا میتوان با استفاده از الگوریتم داده های جدید را برچسب گذاری نمود.

```
def classify(instance, root):
    while root['childs']:
        attribute_idx = attribute.index(root['decision_condition'])
        value = instance[attribute_idx]
        child = next((child for child in root['childs'] if child['decision'] == value), None)
        if child:
            root = child
        else:
            break
    return root['decision_condition']
```

این کد با گرفتن نمونه جدید و درخت تصمیم گیری ، خروجی برچسب را گزارش میدهند.

برای نمونه مطرح شده در صورت سوال هر ۲ روش ، برچسب داده را No تشخیص داده اند.

سوال دوم :

برای این بخش از سری اول تمرینات تمام کد های مربوطه در یک کد جامع تجمیع شده است.

الف)

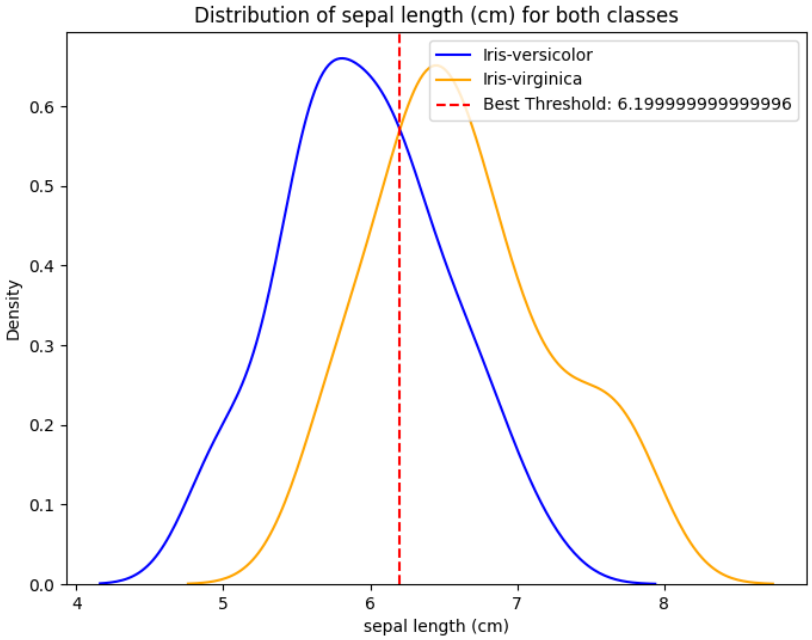
در این قسمت نمودار توزیع پرآکندگی داده ها و همچنین خط مرز جدا شونده ، بر اساس بیشترین IG بدست آمده است.

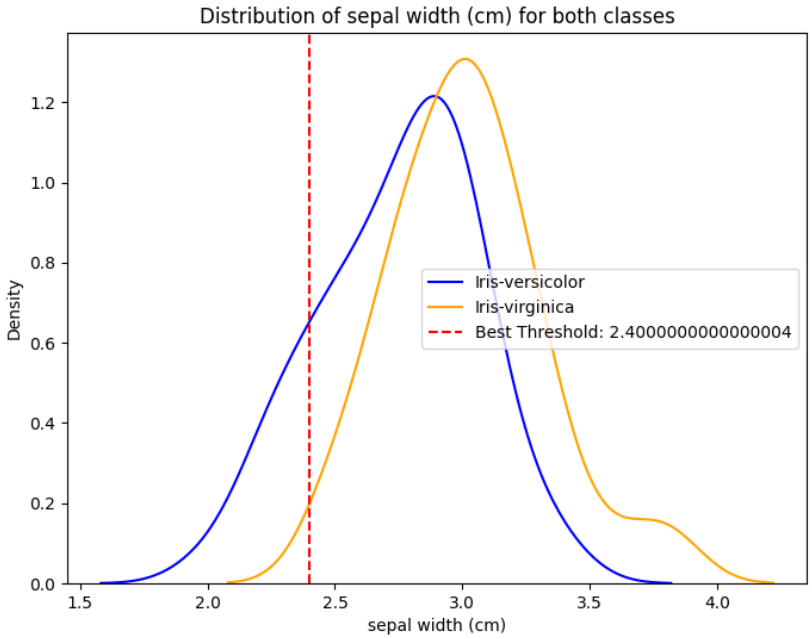
```
def find_best_attribute(data):
    best_attribute = None
    best_threshold = None
    best_info_gain = 0
    max_info_gain = {}
    best_attri_threshold = {}
    best_split_left = None
    best_split_right = None
    decision = 0  # Initialize decision variable

    total_entropy, ans = entropy(data)
    if total_entropy == 0:
        decision = ans
        return best_attribute, max_info_gain, best_threshold, best_attri_threshold, best_split_left,
best_split_right, decision

    for attribute in data.columns[:-1]:
        min_value = data[attribute].min()
        max_value = data[attribute].max()
        max_info_gain[attribute] = 0
        best_attri_threshold[attribute] = 0
        for threshold in np.arange(min_value, max_value, 0.1):
            left_data = data[data[attribute] <= threshold]
            right_data = data[data[attribute] > threshold]

            left_entropy, ans_left = entropy(left_data)
            right_entropy, ans_right = entropy(right_data)

            info_gain = total_entropy - (len(left_data) / len(data) * left_entropy) - (
                    len(right_data) / len(data) * right_entropy)

            if info_gain > max_info_gain[attribute]:
                max_info_gain[attribute] = info_gain
                best_attri_threshold[attribute] = threshold
                best_split_left = left_data
                best_split_right = right_data

        if max_info_gain[attribute] > best_info_gain:
```

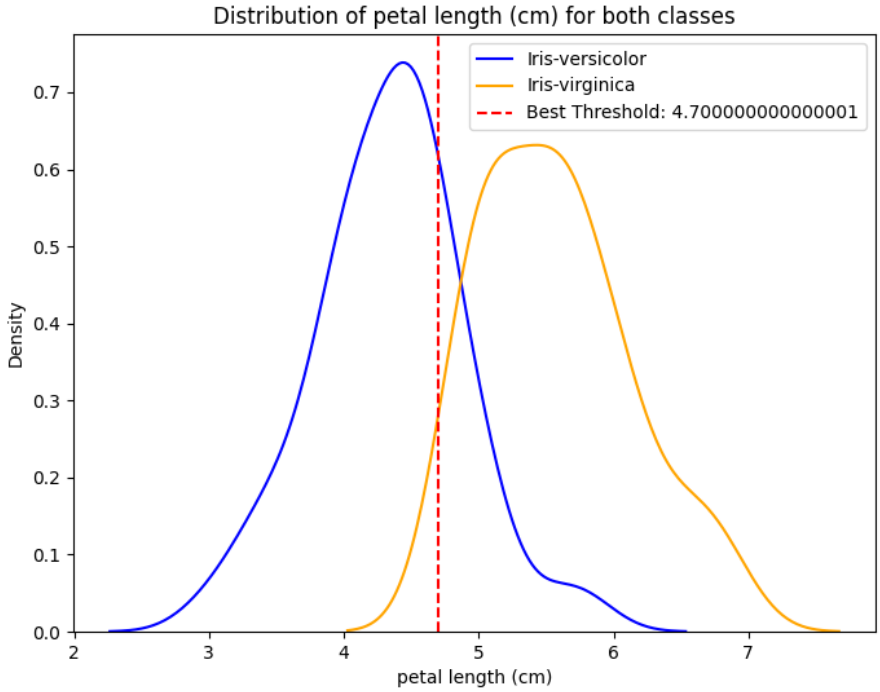
```
            best_info_gain = max_info_gain[attribute]
            best_threshold = best_attri_threshold[attribute]
            best_attribute = attribute

    return best_attribute, max_info_gain, best_threshold, best_attri_threshold, best_split_left,
best_split_right, decision
```

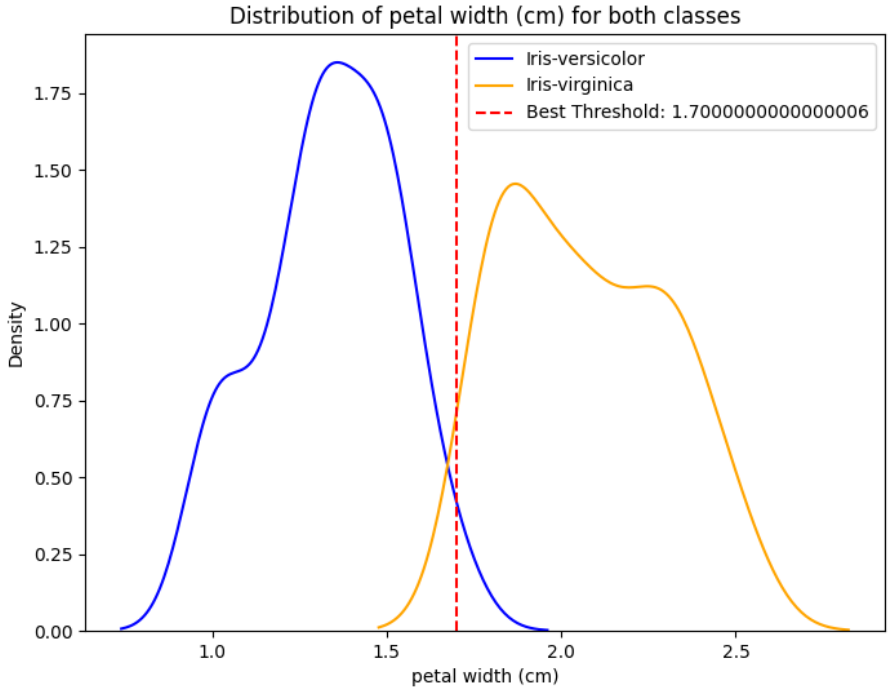
```
# data visualization
def plot_distributions_with_thresholds(data):
    _, max_info_gain, _, best_attri_threshold,best_split_left,best_split_right, _=
find_best_attribute(data)

    for attribute in data.columns[:-1]:
     plt.figure(figsize=(8, 6))

     # Plot distribution lines using KDE for both classes
     sns.kdeplot(best_split_left[attribute], label='Iris-versicolor', color='blue')
     sns.kdeplot(best_split_right[attribute], label='Iris-virginica', color='orange')
     plt.xlabel(attribute)
     plt.ylabel('Density')
     plt.legend()
     plt.title(f'Distribution of {attribute} for both classes')

     # Plot the best threshold as a vertical line on the distribution plot
     plt.axvline(x=best_attri_threshold[attribute], color='r', linestyle='--', label=f'Best Threshold:
{best_attri_threshold[attribute]}')
     plt.legend()
     plt.show()
     # Calculate information gain and best threshold for the attribute
     print(f'Best threshold for {attribute}: {best_attri_threshold[attribute]} (Information Gain:
{max_info_gain[attribute]})')
```

خروجی متناظر با مجموعه مورد سوال برای ۴ ویژگی به صورت زیر میباشد:






Distribution of petal length (cm) for both classes
Iris-versicolor
Iris-virginica
Best Threshold: 4.700000000000001
Density
petal length (cm)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
2
3
4
5
6
7


Distribution of petal width (cm) for both classes
Iris-versicolor
Iris-virginica
Best Threshold: 1.7000000000000006
Density
petal width (cm)
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
1.0
1.5
2.0
2.5

ب) برای توسعه یک درخت تصمیم برای مجموعه ی مورد سوال، از الگوریتم توسعه داده شده در بخش های گذشته استفاده شده است. برای گزارش ماتریس در هم ریختگی از تابع زیر استفاده گردیده است :

```
def findconmat(data, root):
    tp = 0
    tn = 0
    fn = 0
    fp = 0
    columns =   ['sepal length (cm)','sepal width (cm)','petal length (cm)', 'petal width (cm)']
    final_X = [[0, 0], [0, 0]]  # Initialize the confusion matrix

    for i in range(len(data)):
        new_instance = {'sepal length (cm)': 0, 'sepal width (cm)': 0, 'petal length (cm)': 0, 'petal
width (cm)': 0}  # Use a dictionary to represent the instance
        for j in columns:
           new_instance[j] = data.loc[filtered_data.index[i], j]
           classification_result = classify_instance(new_instance, root)
           if data.loc[filtered_data.index[i], 'Species'] == 1:
               actual_result = 'I. versicolor'
           if data.loc[filtered_data.index[i], 'Species'] == 2:
               actual_result = 'I. virginica'

        if classification_result == actual_result == 'I. versicolor':
            tp += 1
        elif classification_result == actual_result == 'I. virginica':
            tn += 1
        elif classification_result != actual_result and classification_result == 'I. versicolor':
            fp += 1
        elif classification_result != actual_result and classification_result == 'I. virginica':
            fn += 1

    final_X = [[tp, fn],
               [fp, tn]]  # Update confusion matrix with counts
    print(final_X)
    precision = tp / (tp + fp) if (tp + fp) != 0 else 0  # Calculate precision
    recall = tp / (tp + fn) if (tp + fn) != 0 else 0  # Calculate recall
    print('precision:', precision)
    print('recall:', recall)
```

درخت تصمیم نهایی و نتیجه ماتریس در هم ریختگی به صورت زیر گزارش میشود.

Start

petal width (cm)

<= 1.7

petal length (cm)

<= 4.9

petal width (cm)

<= 1.6

I. versicolor

> 1.6

I. virginica

>4.9

petal width (cm)

<=1.5

I. virginica

>1.5

sepal length (cm)

<= 6.79

I. versicolor

>6.79

I. virginica

>1.7

petal length (cm)

<= 4.8

sepal length (cm)

<= 5.9

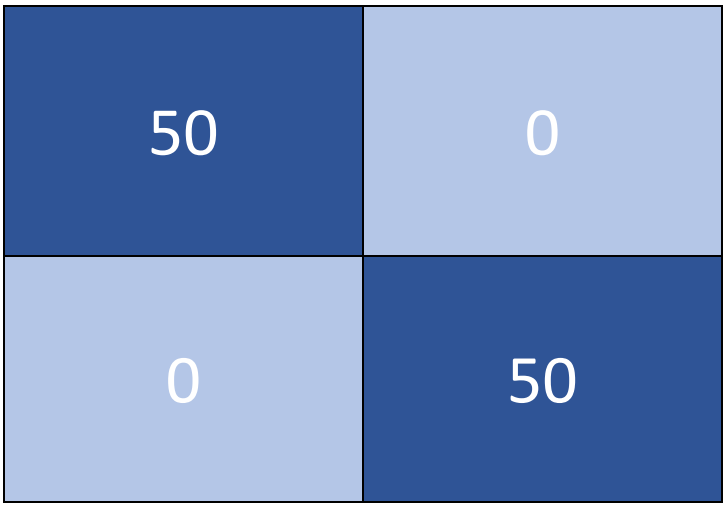
I. versicolor

> 5.9

I. virginica

> 4.8

I. virginica



precision: 1.0

recall: 1.0

با توجه به توسعه کد مربوط به درخت تصمیم و عدم تعریف محدوده ای برای جلوگیری از بروز overfit ، مدل مورد نظر overfit میباشد.
توسعه درخت تصمیم به گونه ای که دارای حد مرزی ثابت نباشد و در هر مرحله حد مرزی جدیدی تعریف بشود ، با وجود افزایش دقت اما مدل را دچار overfit میکند.

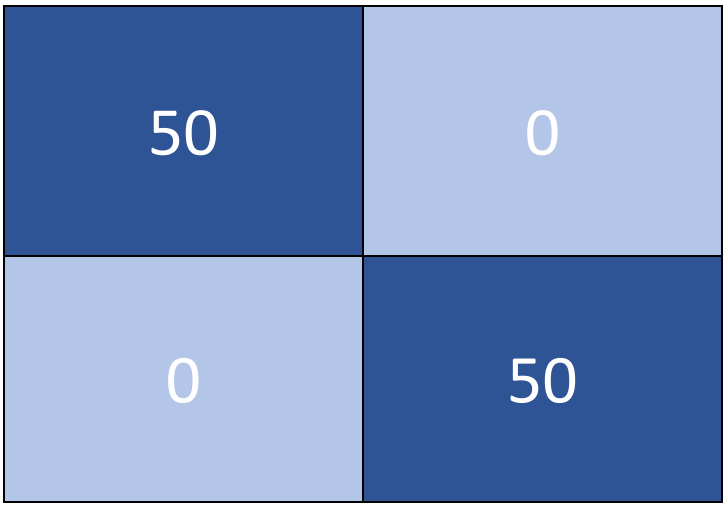
ج)

یک مدل KNN با k متغییر از ۱ تا ۵ گسترش داده شده است. از کد زیر و فاصله اقلیدسی استفاده گردیده شده.

```
# KNN
def euclidean_dist(new_instance, instance):
    distance = 0
    for i in range(len(new_instance)):
        distance += np.sqrt((new_instance[i] - instance[i]) ** 2)
    return distance
def knn(data, new_instance, k):
    distances = []
    for instance in data:
        distance = euclidean_dist(new_instance, instance[:-1])  # Assuming last column is the class label
        distances.append(distance)

    # Get indices of k nearest neighbors
    nearest_indices = np.argsort(distances)[:k]

    # Retrieve class labels of k nearest neighbors
```

```
    classes = [data[i][-1] for i in nearest_indices]

    # Choose the most common class among the nearest neighbors
    prediction = max(set(classes), key=classes.count)
    return prediction
def calculate_KNN(data, k):
    tp = 0
    tn = 0
    fn = 0
    fp = 0
    for i in range(len(data)):
        new_instance = data.iloc[i, :-1]  # Assuming last column is the class label
        prediction = knn(data.values, new_instance, k)
        actual_result = data.iloc[i, -1]  # Assuming last column is the class label

        if prediction == actual_result == 1:
            tp += 1
        elif prediction == actual_result == 2:
            tn += 1
        elif prediction != actual_result and prediction == 1:
            fp += 1
        elif prediction != actual_result and prediction == 2:
            fn += 1

    precision = tp / (tp + fp) if (tp + fp) != 0 else 0  # Calculate precision
    recall = tp / (tp + fn) if (tp + fn) != 0 else 0  # Calculate recall
    print('Confusion Matrix for KNN:')
    print([[tp, fn], [fp, tn]])
    print('Precision:', precision)
    print('Recall:', recall)
```

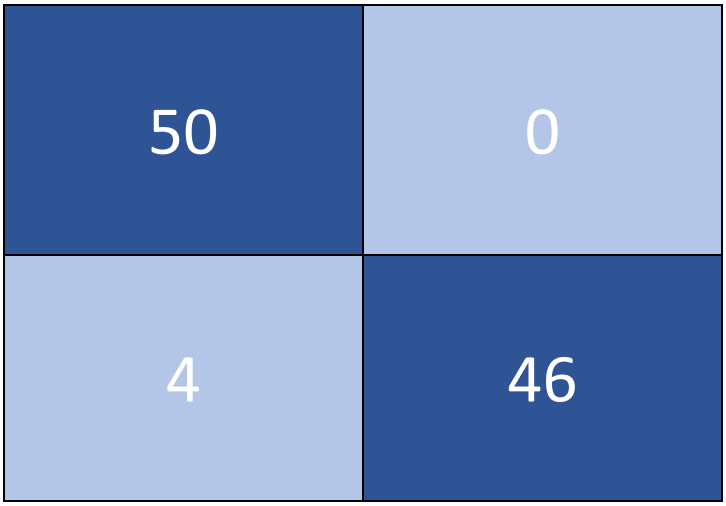
برای مقادیر k مختلف نتایج به صورت زیر میباشد:



K : 1

precision: 1.0

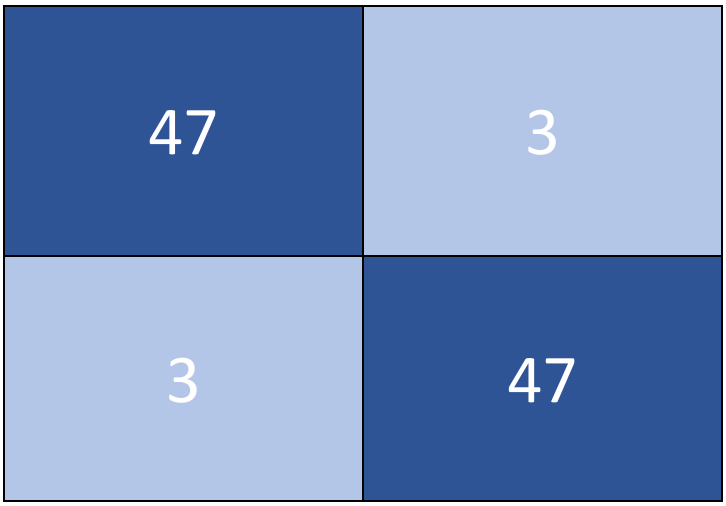
recall: 1.0



K : 2

precision: 0.925

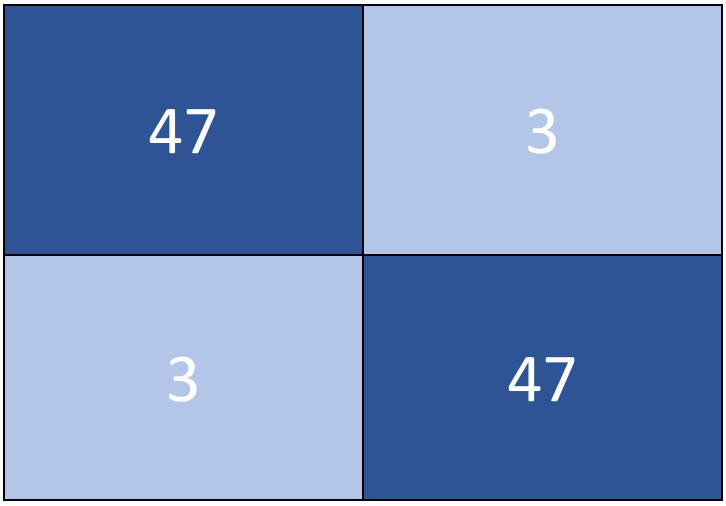
recall: 1.0



K : 3

precision: 0.94

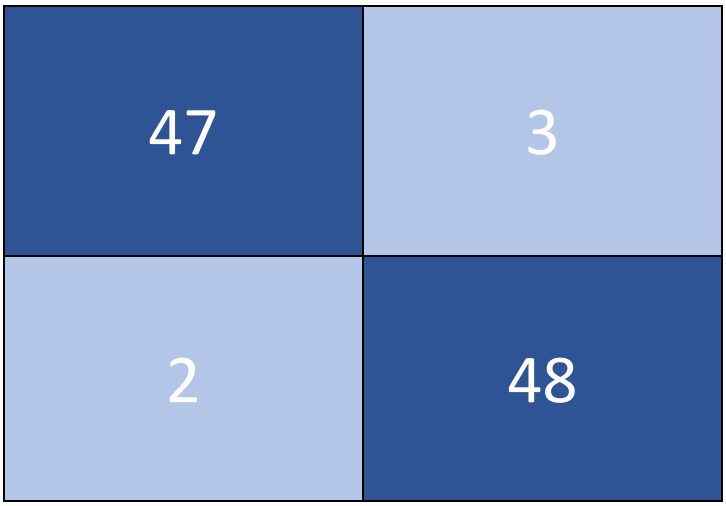
recall: 0.94



K : 4

precision: 0.94

recall: 0.94



K : 5

precision: 0.959

recall: 0.94

با وجود دقت عددی بالاتر در مورد k = 1 اما در این حالت مدل دچار overfit شده است. با مقایسه مقادیر مربوط به precision و recall میتوان نتیجه گرفت بالاترین دقت بدون overfit مربوط به حالتی است که k = 5 میباشد.

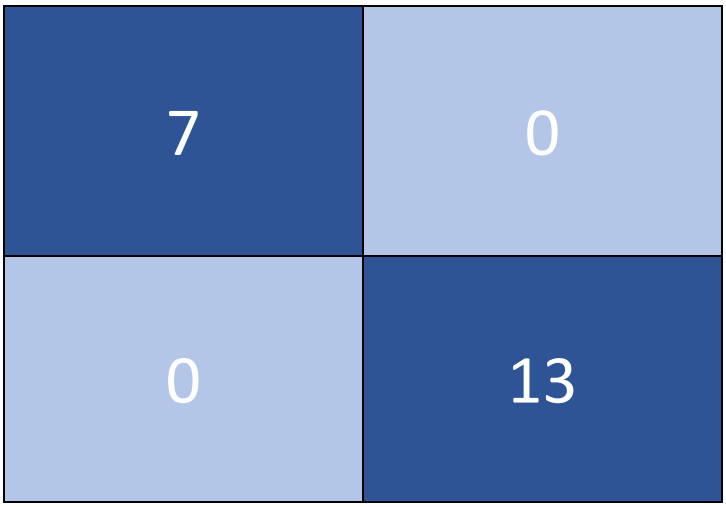
د)

در این قسمت با استفاده از الگوریتم K-fold روی کد موجود برای درخت تصمیم گیری نتایج را یکبار دیگر تکرار میکنیم.

از کد زیر برای تشکیل الگوریتم K-fold استفاده شده است.

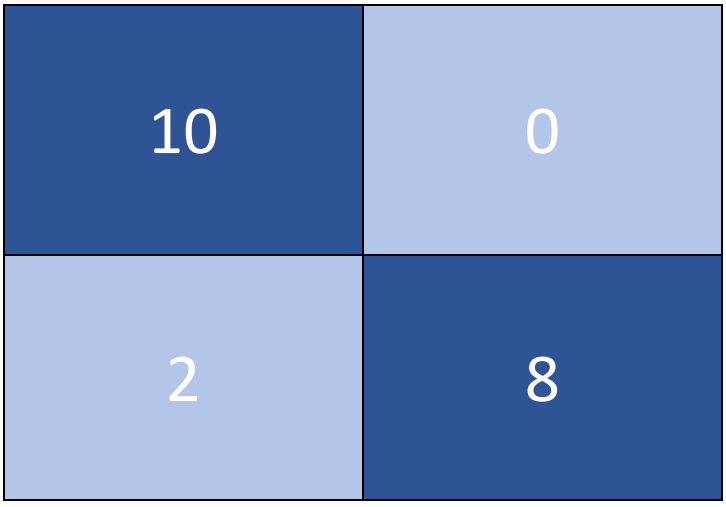
```
def kfold_decision_tree(data):
    precision_max = 0
    recall_max = 0
    final_X_max = 0
    choice3 = input("how mant fold do you want? ")
    kf = int(choice3)
    # Creating 4 subsets
    shuffled_data = data.sample(frac=1).reset_index(drop=True)
    subset_size = len(shuffled_data) // kf
    # Creating 4 subsets
    subsets = []
    start_index = 0
    for i in range(kf - 1):
        subset = shuffled_data.iloc[start_index:start_index + subset_size]
        subsets.append(subset)
        start_index += subset_size
    subsets.append(shuffled_data.iloc[start_index:])

    for j in range(kf):
        train_data = pd.DataFrame()
        test_data = pd.DataFrame()
        test_data = pd.concat([test_data, subsets[j]], ignore_index=True)
        for i in range(kf):
            if i!=j:
                train_data = pd.concat([train_data, subsets[i]], ignore_index=True)
        root=build_tree(train_data)
        final_X,precision,recall = find_con_matrix(shuffled_data,test_data, root)
        if precision > precision_max and recall > recall_max :
             precision_max=precision
             recall_max=recall
             final_X_max = final_X
    return precision_max,recall_max,final_X_max
```

نتایج این مرحله به صورت زیر میباشد:



precision: 1.0

recall: 1.0



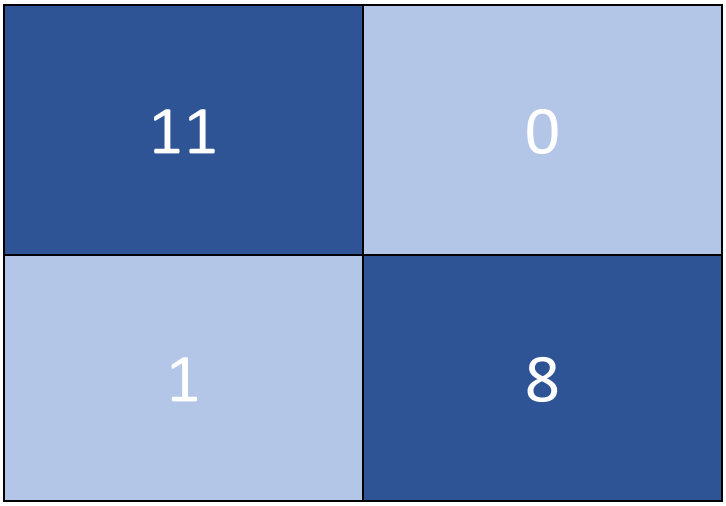
precision: 0.83

recall: 1.0

با توجه به اختلاف نتیجه در دو بار اجرای کد ، استفاده از الگوریتم فوق دقت مدل را کاهش نداده است و حتی با تشکیل k-fold و انتخاب مجموعه های تصادفی در هر بار اجرای کد از overfit شدن جلوگیری میشود.

ه)

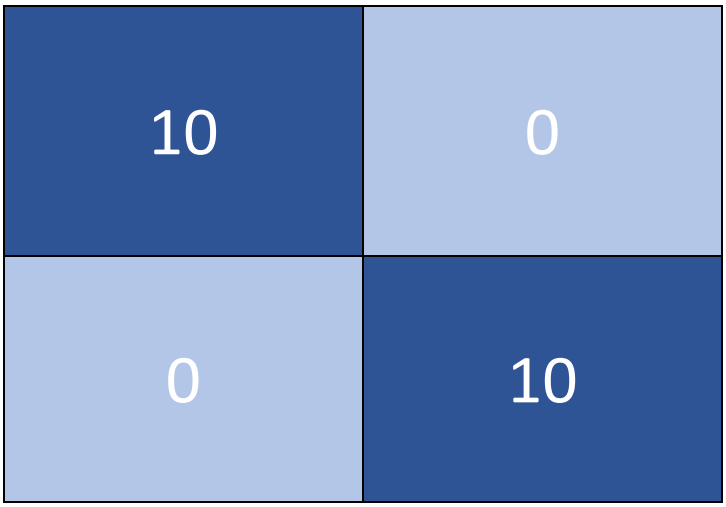
برای مقادیر k مختلف و 5-fold نتایج به صورت زیر میباشد:



K : 1

max Precision: 1.0

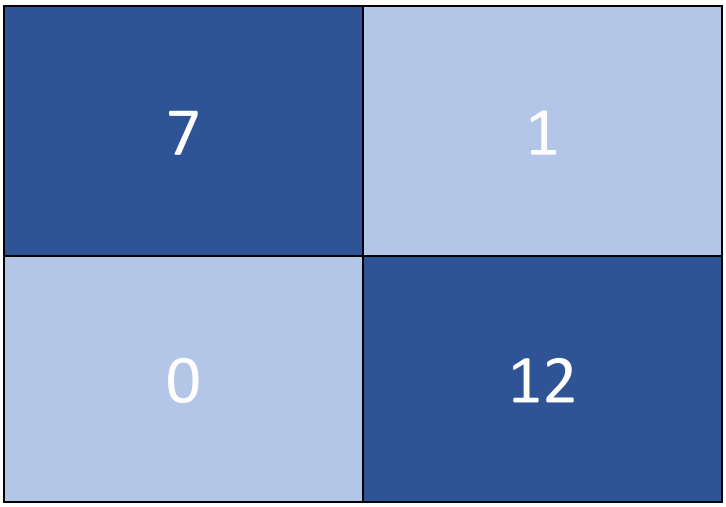
Recall: 0.8888888888888888



K : 2

max Precision: 1.0

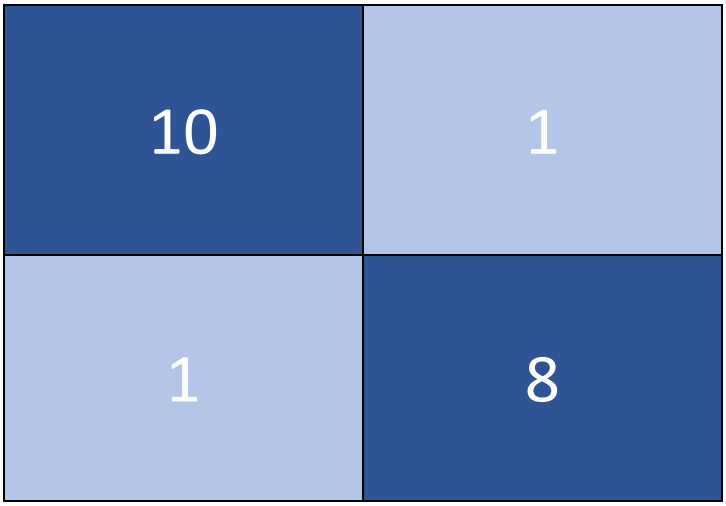
Recall: 1.0



K : 3

max Precision: 0.9230769230769231

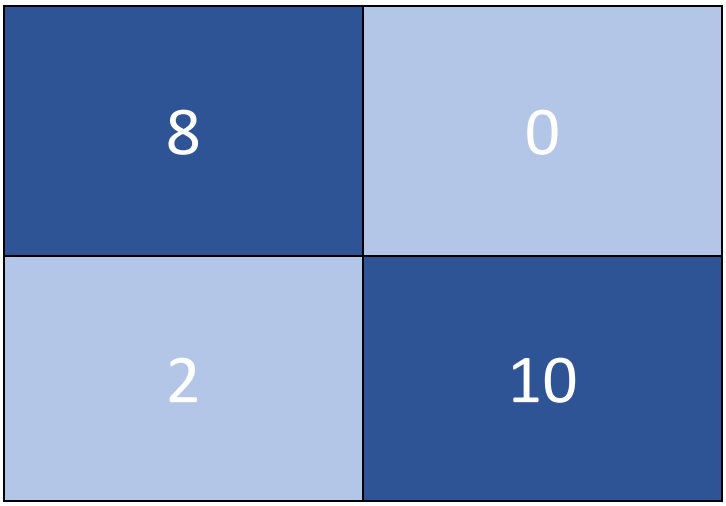
Recall: 1.0



K : 4

max Precision: 0.8888888888888888

Recall: 0.8888888888888888



K : 5

max Precision: 1.0

Recall: 0.8333333333333334

به علت تصادفی بودن مجموعه های انتخاب شده در K-fold نمیتوان به طور قطعی نظر داد اما میتوان گفت با وجود این اصل که دقت مدل ها در بعضی از موارد کمتر شده است اما به طور کلی الگوریتم k-fold با دور کردن مدل از ناحیه ی overfit یک مدل واقعی تر برای دسته بندی ارائه میدهد.

و)

با مقایسه نتایج دو قسمت قبلی با وجود الگوریتم k-fold ، متوجه میشویم که هر دو مدل از ناحیه ی overfit دور میشوند. اما با توجه به مفاهیم الگوریتم KNN میتوان انتظار داشت که K-fold در کنار این الگوریتم عملکرد بهتری خواهد داشت که نتایج عددی این مسئله را تایید میکنند.

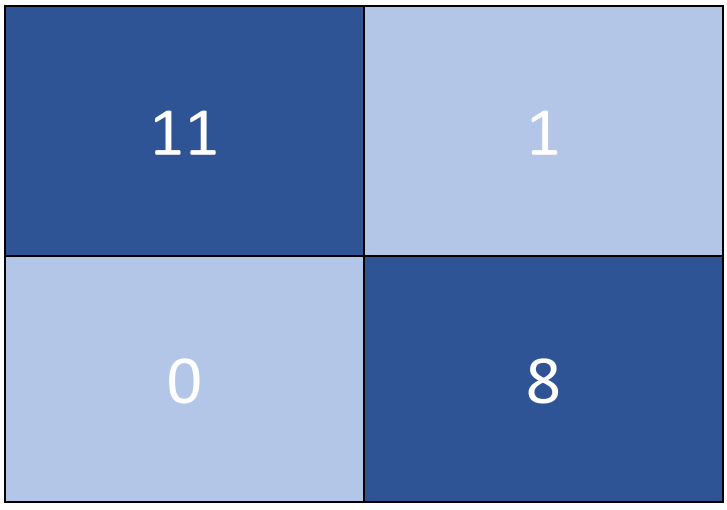
ز)

با استفاده از کتابخانه sikit و استفاده از کد زیر ، مدل های درخت تصمیم و KNN برای دسته داده iris توسعه داده شده است :

```
X_train, X_test, y_train, y_test = train_test_split(X, y_binary, test_size=0.2, random_state=42)

# Decision Tree model
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train, y_train)

# Predictions and metrics for Decision Tree
dt_predictions = dt_model.predict(X_test)
```

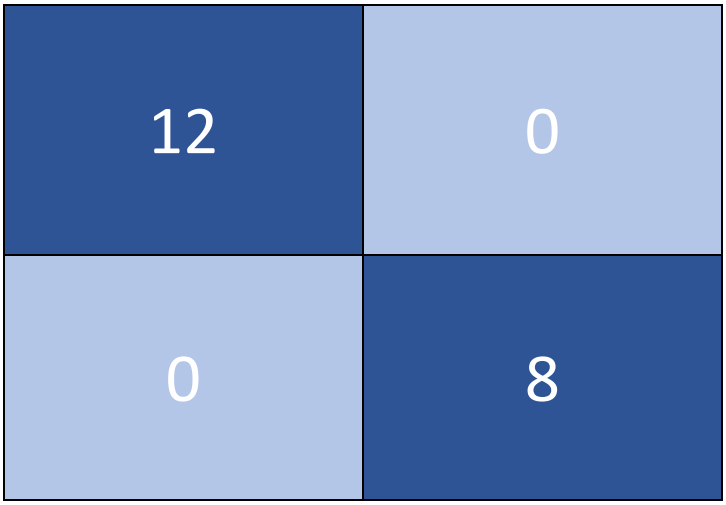
```
dt_conf_matrix = confusion_matrix(y_test, dt_predictions)
dt_precision = precision_score(y_test, dt_predictions)
dt_recall = recall_score(y_test, dt_predictions)

print("Decision Tree Confusion Matrix:")
print(dt_conf_matrix)
print("Decision Tree Precision:", dt_precision)
print("Decision Tree Recall:", dt_recall)
print("\n")

# k-Nearest Neighbors model
knn_model = KNeighborsClassifier(n_neighbors=3)
knn_model.fit(X_train, y_train)

# Predictions and metrics for k-Nearest Neighbors
knn_predictions = knn_model.predict(X_test)

knn_conf_matrix = confusion_matrix(y_test, knn_predictions)
knn_precision = precision_score(y_test, knn_predictions)
knn_recall = recall_score(y_test, knn_predictions)

print("k-Nearest Neighbors Confusion Matrix:")
print(knn_conf_matrix)
print("k-Nearest Neighbors Precision:", knn_precision)
print("k-Nearest Neighbors Recall:", knn_recall)
```

نتیجه دقت مدل های موجود به صورت زیر میباشد:



Decision Tree Precision: 0.8888888888888888

Decision Tree Recall: 1.0



K : 3

k-Nearest Neighbors Precision: 1.0

k-Nearest Neighbors Recall: 1.0

با توجه به نتایج بدست آمده، قابل مشاهده است که دقت مدل توسعه یافته توسط کتابخانه آماده و به صورت دستی دارای تفاوت فاحش نمیباشد.