گزارش تمرین عملی دوم درس هوش مصنوعی

آریا جلالی
۹۸۱۰۵۶۶۵

۱ کتابخانه‌های استفاده شده:

```
import pandas as pd
import numpy as np
import gvgen
import os
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,
    ConfusionMatrixDisplay
from graphviz import Source
```

از کتابخانه‌ی numpy برای ذخیره کردن ویژگی‌های متفاوت رستوران‌ها و افراد دیابتی استفاده شده است و از کتابخانه‌ی pandas برای خواندن فایل‌های csv استفاده شده است. از کتابخانه‌های gvgen و matplotlib و graphviz برای نشان دادن درخت نهایی استفاده شده است. در نهایت از کتابخانه‌ی sklearn برای انتخاب سمپل‌های رندوم و محاسبه خطای برنامه استفاده شده است که در ادامه کاربرد آن‌ها را به صورت مفصل توضیح خواهم داد.

۲ توضیح بخش‌های مختلف کد:

```
# function for discretizing our continuous data.
def discretize(data: np.array, number_of_bins):
    global bins
    for i in range(8):
        bins.append(pd.cut(data[i], number_of_bins, retbins=True
    )[1])
        data[i] = np.digitize(data[i], bins[i], right=True)
```

این تابع وظیفه‌ی گسسته‌سازی اطلاعات برای افراد دیابتی را دارد و روش استفاده شده. این تابع به این صورت کار می‌کند که برای هر ستون ابتدا تابع pd.cut را صدا می‌زند و این تابع یک لیست را برمی‌گرداند که هر 2 عضو مجاور آن نشان دهنده‌ی یک بازه هستند و در نهایت می‌توان اطلاعات هر ستون را با توجه به مقدار آن‌ها با استفاده از تابع np.digitize که دو ورودی داده‌ها و لیست bin ها را می‌گیرد در بازه‌ی متناظر خود قرار داد.

```
def test(row: np.array, root):
    global Correct_Guess
    if len(root.children) == 0:
        y_pred.append(int(root.plus > root.negative))
        if int(root.plus > root.negative) == int(row[-1]):
            Correct_Guess += 1
        return
    if row[root.attribute] not in root.children.keys():
        if np.random.rand() > 0.5:
            y_pred.append(int(row[-1]))
            Correct_Guess += 1
        else:
```

```
            y_pred.append(1 - int(row[-1]))
        return
    test(row, root.children[row[root.attribute]])
```

تابع test برای تست نهایی برنامه استفاده می‌شود و برای تمامی افراد تابع test را به صورت بازگشتی تا به رسیدن یک برگ صدا می‌زند و نتیجه را بررسی می‌کند. دقت کنید اگر مقداری را برای اولین بار بر روی درخت ببینیم با استفاده از تابع random بررسی می‌کنیم آیا جواب ما درست بوده است یا خیر. با دیدن هر جواب درست متغیر Correct_Guess global را یکی افزایش می‌دهیم و مقدار پیشبینی شده را در لیستی به نام y_pred قرار میدهیم تا در نهایت نتایج را با استفاده از Confusion Matrix بررسی کنیم.

```
# Decision tree
class Tree:
    def __init__(self, data, parent_attribute=None, depth=0):
        if parent_attribute is None:
            parent_attribute = []
        self.children = dict()
        self.attribute = -1
        self.depth = depth
        self.attribute_name = 'Leaf'
        self.information_gain = -1
        self.entropy = -1
        self.plus = 0
        self.negative = 0
        self.parent_attribute = parent_attribute
        self.data = np.array(data)

    def __str__(self):
        if len(self.children) == 0:
            return self.attribute_name + "\n" + "Entropy: " +
    str(self.entropy) + "\n" + "[" + str(
                self.plus) + "/" + str(
                self.negative) + "]"
        return self.attribute_name + "\n" + "Gain: " + str(self.
    information_gain) + "\n" + "[" + str(
            self.plus) + "/" + str(
            self.negative) + "]"
```

در این بخش و ادامه کلاس اصلی درخت تصمیم که Tree نام دارد را بررسی می‌کنیم. این کلاس در constructor خود 3 ورودی می‌گیرد که این ورودی‌ها به ترتیب برابر با data,parent_attributes,depth هستند که در ادامه به توضیح تک تک آن‌ها خواهم پرداخت.

ورودی data همان افراد مورد بررسی هر Node را نشان می‌دهد که برای root تمامی اعضا را شامل می‌شود و در مراحل بعد با انتخاب ویژگی با توجه به معیارهای متفاوت(information gain یا gini) افراد فیلتر می‌شوند و به مراحل بعدی داده می‌شوند.

parent_attribute برای هر Node نشان دهنده‌ی ویژگی‌هایی است که پدر و اجداد آن تا

الان بررسی کرده‌اند و در این مرحله باید skip شوند و نیازی به بررسی آن‌ها نیست. دقت کنید برای root این ورودی برابر با یک لیست خالی است، زیرا هنوز ویژگی‌ای انتخاب نشده است.

depth نشان دهنده‌ی عمق Node مورد بررسی است که در ابتدا برای root برابر با صفر قرار داده شده است و فرزندان هر Node دارای عمق depth + 1 هستند. این ورودی depth برای محدود کردن سایز درخت و جلوگیری از overfit شدن استفاده شده است که در ادامه در رابطه با آن توضیحاتی خواهم داد.

children فرزندان یک Node را نشان می‌دهد که آن را با استفاده از یک dictionary نشان می‌دهیم که key نشان دهنده‌ی ویژگی مورد بررسی در Node فرزند و value متناظر با آن کلاس فرزند است.

plus & negative به ترتیب نشان دهنده‌ی این هستند که در data فعلی چند نفر از افراد دیابت دارند و چند نفر ندارند. از این ویژگی‌ها برای محاسبه‌ی معیارهایی مانند information gain یا gini و نشان دادن روی درخت نهایی به صورت گرافیکی استفاده خواهیم کرد.

```
def train(self):
    self.plus = np.sum(self.data.T[-1])
    self.negative = self.data.shape[0] - self.plus
    self.attribute_name = str(int(self.plus > self.negative))
    self.entropy = calculate_entropy(self.plus, self.negative)

    if self.depth == max_depth or self.plus == 0 or self.
   negative == 0:
        return

    difference = dict()

    for i in range(len(self.data[0]) - 1):
        if i in self.parent_attribute:
            continue
        for row in self.data:
            if row[i] in difference:
                difference[row[i]].append(row)
            else:
                difference[row[i]] = []
                difference[row[i]].append(row)
        self.choose_attribute(difference, i)
        if self.attribute == i:
            self.children.clear()
            next_parent_attribute = self.parent_attribute.copy()
            next_parent_attribute.append(self.attribute)
            for key, value in difference.items():
                self.children[key] = Tree(value, depth=self.
   depth + 1, parent_attribute=next_parent_attribute)
```

```
        difference.clear()

    for key, value in self.children.items():
        value.train()

    if len(self.children) == 0:
        self.attribute_name = str(int(self.plus > self.negative)
    )
```

تابع train تابع اصلی ما را برای ساخت درخت تصمیم تشکیل می‌دهد. نحوه‌ی کار به این صورت است که ابتدا یک dictionary به نام difference میگیریم و در ادامه روی تمام 8 ویژگی داده شده loop می‌زنیم و هر مقدار جدیدی را که میبینیم آن را به عنوان یک key در difference dictionary قرار میدهیم و برای هر key یک لیست می‌گیریم که نقطه مشترک این افراد این است که ویژگی i ام آن‌ها برابر با key متناظر با لیستی هستند که در آن قرار دارند.
در ادامه تابع choose_attribute با ورودی های i و difference صدا میزنیم که نحوه‌ی کار آن را در ادامه توضیح خواهیم داد در ادامه بررسی می‌کنیم که آیا این ویژگی که آن را مورد بررسی قرار دادیم بهتر از ویژگی‌های پیشین است؟ و در آن صورت dictionary فرزندان را clear میکنیم و فرزندان جدید را از difference میخوانیم و به children اضافه می‌کنیم. در نهایت نیز در صورت داشتن فرزندان و برگ نبودن تابع train را بر روی تک تک فرزندان صدا می‌زنیم.

```
def choose_attribute(self, difference, attribute):
    remainder_A = 0
    for key, value in difference.items():
        plus = 0
        negative = 0
        for each in value:
            if each[-1] == 1:
                plus += 1
            else:
                negative += 1
        remainder_A += ((plus + negative) / (self.plus + self.
    negative)) * calculate_entropy(plus, negative)
    if self.information_gain < self.entropy - remainder_A:
        self.information_gain = self.entropy - remainder_A
        self.attribute = attribute
        self.attribute_name = HEADERS[attribute]
```

در تابع choose_attribute مقدار remainder(A) به صورت مقابل با استفاده از یک for loop محاسبه می‌شود.

$$Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p + n} B(\frac{p_k}{p_k + n_k})$$

که در معادله بالا d نشان دهنده‌ی مقادیر ممکن و p_k و n_k به ترتیب نشان دهنده‌ی افراد دیابتی و غیر دیاتی‌ای هستند که در زیرمجموعه k قرار دارند. و در نهایت تابع Entropy B

را از طریق فرمول مقابل محاسبه می‌کند.

$$B(q) = -(q * log_2 * q + (1 - q) * log_2 * (1 - q))$$

در نهایت مقدار information gain را از طریق فرمول مقابل محاسبه می‌کنیم.

$$Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$$

و در صورتی که $Gain$ محاسبه شده بیشتر از مقدار فعلی باشد ویژگی attribute کلاس به ویژگی مورد بررسی تغیر پیدا می‌کند.

```
def calculate_entropy(plus, negative):
    q = plus / (plus + negative)
    if q == 0 or q == 1:
        return 0
    return -(q * np.log2(q) + (1 - q)
             * np.log2(1 - q))
```

در تابع بالا نیز تعداد افراد دیابتی و سالم به تبدیل به متغیر q می‌شوند و entropy طبق فرمول بالا محاسبه می‌شود.

```
def get_edge_name(key, val: Tree):
    attribute_bin = bins[val.attribute]
    key = int(key)
    return str(round(attribute_bin[key - 1], 2)) + " <= x <= " +
    str(round(attribute_bin[key], 2))


def graphMaker(g, my_tree: Tree):
    if len(my_tree.children) == 0:
        myItem = g.newItem(my_tree.__str__())
        return myItem
    else:
        myItem = g.newItem(my_tree.__str__())
        for key, val in my_tree.children.items():
            newTree = graphMaker(g, val)
            link = g.newLink(myItem, newTree)
            g.propertyAppend(link, "color", "darkblue")
            g.propertyAppend(link, "label", get_edge_name(key,
   my_tree))
        return myItem


def makeVisualGraph(my_tree):
    g = gvgen.GvGen()
    graphMaker(g, my_tree)
    string = ""
    my_file = open("output_graphviz.txt", 'w')
    g.dot(my_file)
```

```
    my_file.close()
    my_file = open("output_graphviz.txt", 'r')
    lines = my_file.readlines()[1:]
    for line in lines:
        string = string + line
    src = Source(string)
    src.render(view=True)
```

از 3 تابع بالا برای نمایش گرافیکی درخت تصمیم استفاده کرده‌ام به نحوی که ابتدا تابع makeVisualGraph صدا زده می‌شود و آن نیز تابع graphMaker را صدا می‌زند که آن با اجرای یک dfs و استفاده از کتابخانه‌ی gvgen یک درخت به زبان dot که زبان رسمی موتور گرافیکی graphivz است را در یک فایل txt می‌نویسد و در ادامه آن را تبدیل به یک فایل dot میکنیم و با استفاده از موتور گرافیکی graphivz درخت تصمیم را نمایش می‌دهیم.

```
if __name__ == '__main__':
    # read attributes from file.
    df = pd.read_csv('diabetes.csv')
    HEADERS = df.columns

    # Put all attributes into a numpy matrix.
    attributes = df.to_numpy().T
    discretize(data=attributes, number_of_bins=5)
    attributes_train, attributes_test = train_test_split(
    attributes.T, test_size=0.2)

    root = Tree(attributes_train)
    root.train()
    makeVisualGraph(root)

    for row in attributes_train:
        test(row, root)
    print("Accuracy for train data: " + str(round(Correct_Guess
    / len(attributes_train) * 100, 2)) + "%")
    Correct_Guess = 0

    cm = confusion_matrix(attributes_train.T[-1], y_pred)

    disp = ConfusionMatrixDisplay(confusion_matrix=cm)
    disp.plot()
    plt.show()

    y_pred = []

    for row in attributes_test:
        test(row, root)
    print("Accuracy for test data: " + str(round(Correct_Guess /
     len(attributes_test) * 100, 2)) + "%")

    cm = confusion_matrix(attributes_test.T[-1], y_pred)
```

```
33
34      disp = ConfusionMatrixDisplay(confusion_matrix=cm)
35      disp.plot()
36      plt.show()
37
38      average = 0
39      for i in range(100):
40          attributes_train, attributes_test = train_test_split(
    attributes.T, test_size=0.2)
41
42          root = Tree(attributes_train)
43          root.train()
44
45          for row in attributes_test:
46              test(row, root)
47          average += round(Correct_Guess / len(attributes_test) *
    100, 2)
48          Correct_Guess = 0
49
50      print("Average Accuracy of tree on test data is : " + str(
    round(average / 100, 2)))
```

تکه کد بالا که بخش اصلی کد را نشان می‌دهد ابتدا data را از طریق فایل داده شده می‌خواند و در ادامه با استفاده از تابع‌های گفته شده درخت تصمیم را تشکیل می‌دهد و در ادامه با استفاده از تابع test درخت تصمیم را بر روی attributes_test اجرا می‌کند و دقت برنامه را به کاربر نشان می‌دهد و در نهایت این کار را برای 100 بار تکرار می‌کند تا مقدار متوسطی از عملکرد درخت داشته باشیم.
برای جدا کردن data به 2 بخش train و test از تابع test_train_split کتابخانه‌ی sklearn استفاده میکنیم.

۳ بررسی معیارهای استفاده شده:

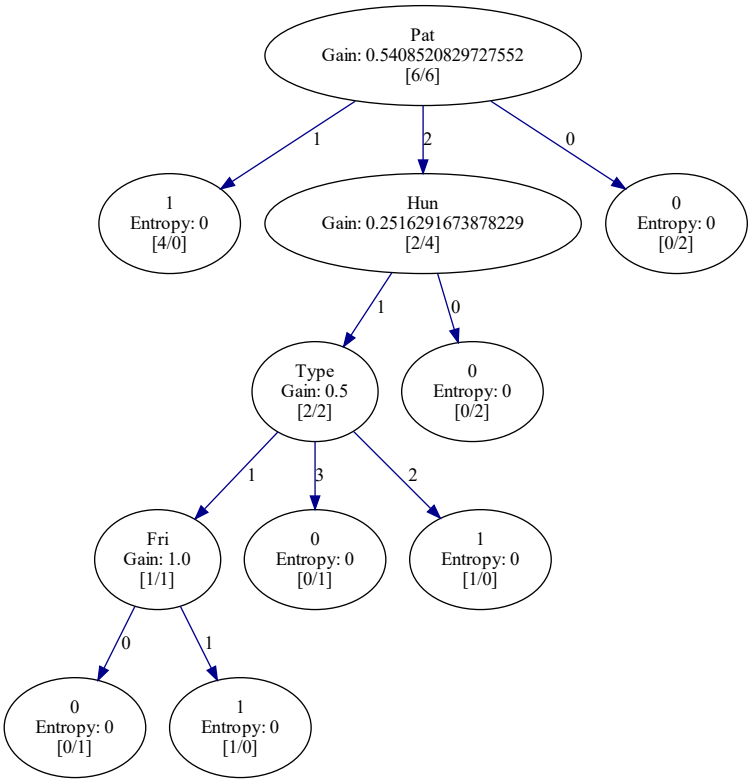
همانطور که در بخش‌های قبل‌تر نیز گفته شد علاوه بر معیار information gain از معیار gini index نیز استفاده شده است که محاسبه‌ی آن به صورت مقابل است:

$$Gini\ Impurity(k) = 1 - (\frac{p_k}{p_k + n_k})^2 - (\frac{n_k}{p_k + n_k})^2$$

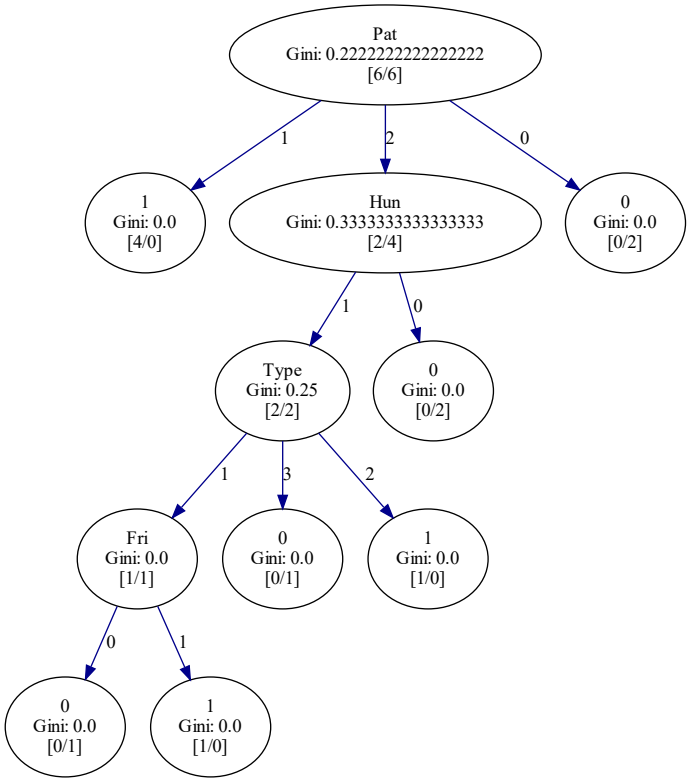
$$Gini\ Index = \sum_{k=1}^{d} Gini\ Impurity(k) \times \frac{p_k + n_k}{p + n}$$

دقت کنید معیار gini index برخلاف معیار information gain در مقادیر پایین نشان دهنده‌ی بهتر بودن ویژگی مورد بررسی است و از طرفی محاسبه‌ی آن نسبت به information gain از لحاظ محاسباتی آسان‌تر است و در ادامه نتایج این 2 معیار را مقایسه خواهیم کرد.

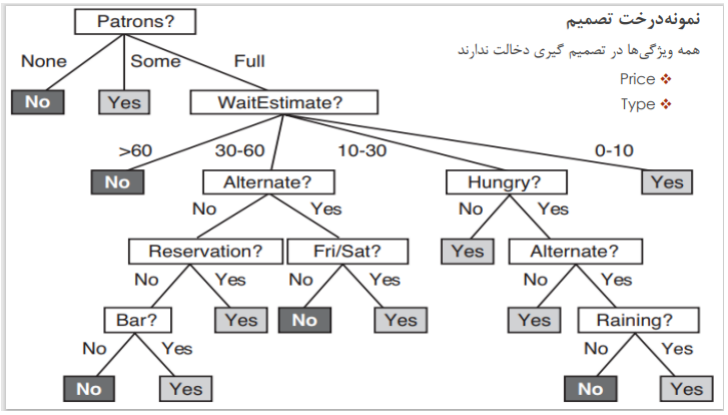
۴ درخت تولید شده برای داده‌های رستوران:



شکل ۱: درخت تصمیم تولید شده با معیار information gain



شکل ۲: درخت تصمیم تولید شده با معیار gini index



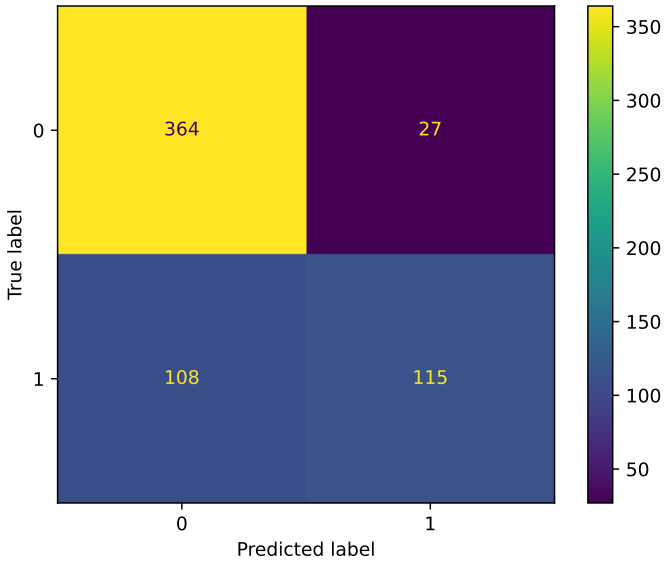
شکل ۳: نمونه درخت تولید شده در اسلاید درس

همانطور که از تصاویر بالا نیز مشخص است هر 3 درخت تولید شده یکسان شده‌اند و دلیل این عمل برای درخت اول و سوم واضح است زیرا هر 2 یک الگوریتم را پیاده‌سازی کرده‌اند ولی جالب است که معیار gini index نیز همان درخت را تشکیل داده است. دلیل اینکار را می‌توان به این حقیقت که تمام داده‌ها برای ساخت درخت استفاده شده‌اند و باید دقت آن‌ها 1 شود نسبت داد.
دقت کنید در معیار gini index روی هر Node مقدار gini impurity آن نود روی آن نوشته شده است و در معیار information gain روی Node های میانی مقدار Gain و بر روی برگ‌ها مقدار entropy نوشته شده است و در هر دو معیار ویژگی مورد بررسی در هر Node روی آن نوشته شده است و در نهایت به ترتیب تعداد افراد دیابتی و سالم در هر Node به فرم [Plus,Negative] نوشته شده است.

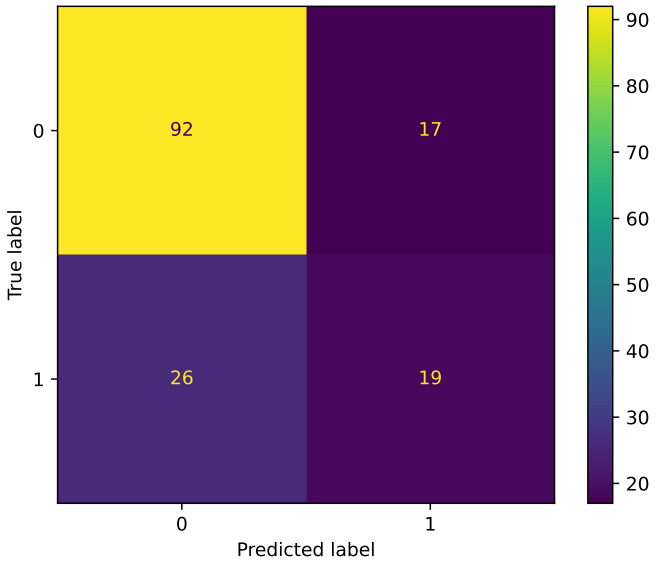
۵ بررسی دقت درخت‌های تولید شده بر روی داده‌های تست:

شکل ۴: درخت تولید شده با استفاده از معیار gini index

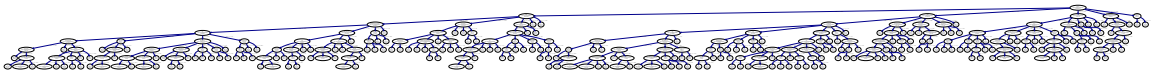
دقت کنید به دلیل بزرگ بودن درخت تصویر بالا کوچک نشان داده شده است ولی به دلیل svg بودن فرمت عکس می‌توانید هر مقدار که بخواهید روی آن zoom کنید از طرفی تمامی عکس‌ها در فایل‌های فرستاده شده قرار داده شده‌اند.
درخت بالا با حداکثر عمق 3 و تعداد 5 bin طراحی شده است و دقت آن برای داده‌های train برابر با 78.01% و برای داده‌های test برابر با 72.08% است و مقدار متوسط آن پس از 100 بار اجرا روی داده‌های تست برابر با 72.7% است.



شکل ۵: ماتریکس confusion برای داده‌های train

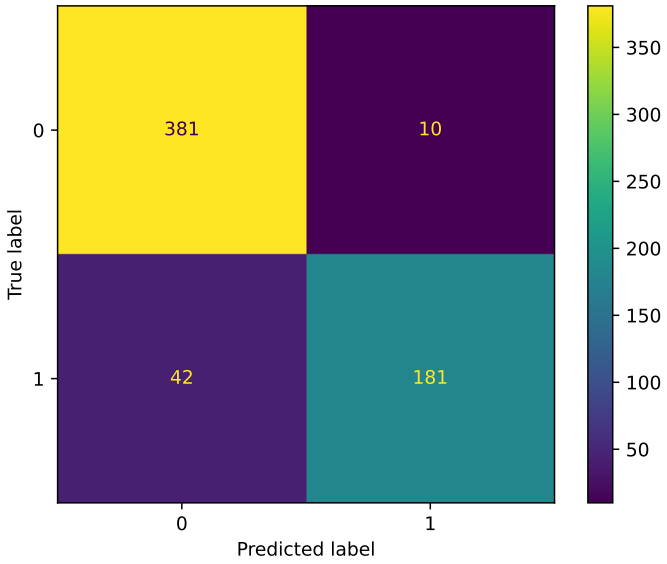


شکل ۶: ماتریکس confusion برای داده‌های test

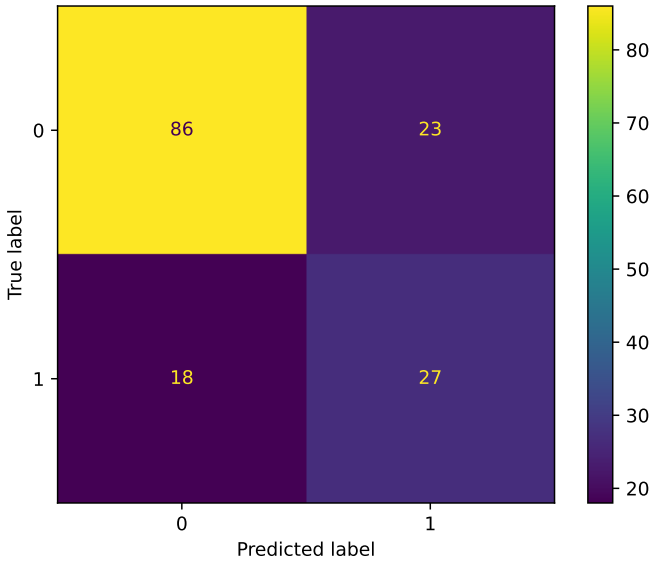


شکل ۷: درخت تولید شده با معیار gini index

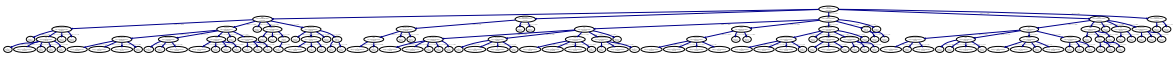
درخت بالا با حداکثر عمق 10 و تعداد 5 bin طراحی شده است و دقت آن برای داده‌های train برابر با 91.53% و برای داده‌های test برابر با 73.38% است و مقدار متوسط آن پس از 100 بار اجرا روی داده‌های تست برابر با 69.28% است.



شکل ۸: ماتریکس confusion برای داده‌های train

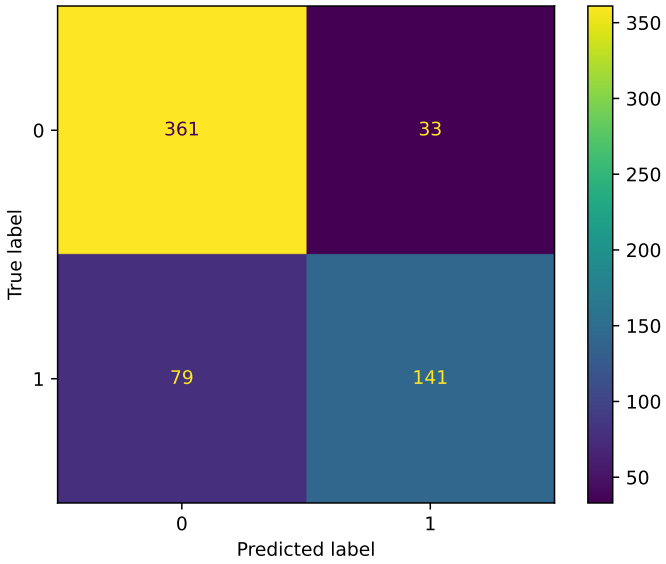


شکل ۹: ماتریکس confusion برای داده‌های test

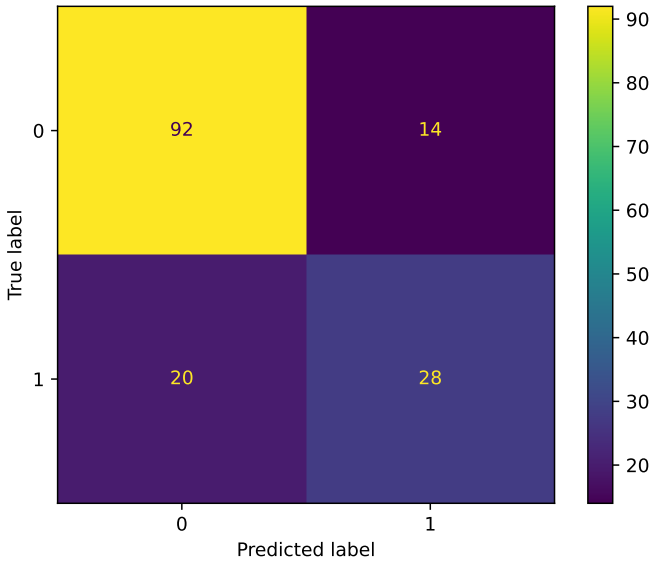


شکل ۱۰: درخت تولید شده با استفاده از معیار information gain

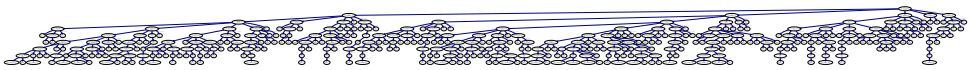
درخت بالا با حداکثر عمق 4 و تعداد 5 bin طراحی شده است و دقت آن برای داده‌های train برابر با 81.76% و برای داده‌های test برابر با 77.92% است و مقدار متوسط آن پس از 100 بار اجرا روی داده‌های تست برابر با 71.5% است.



شکل ۱۱: ماتریکس confusion برای داده‌های train

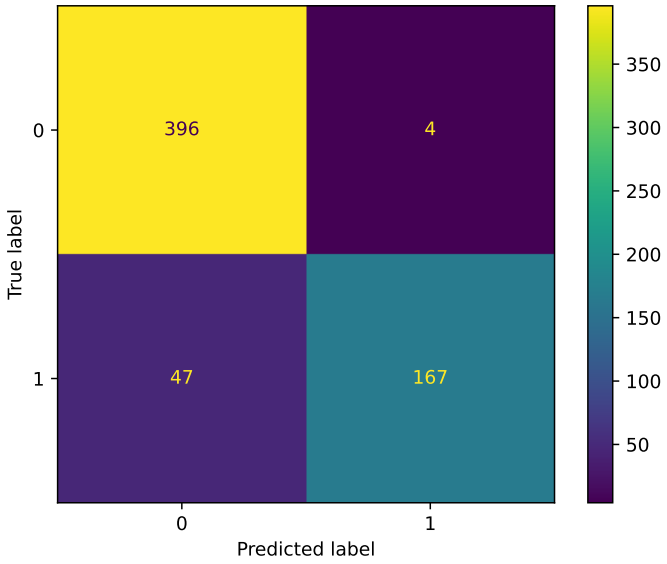


شکل ۱۲: ماتریکس confusion برای داده‌های test

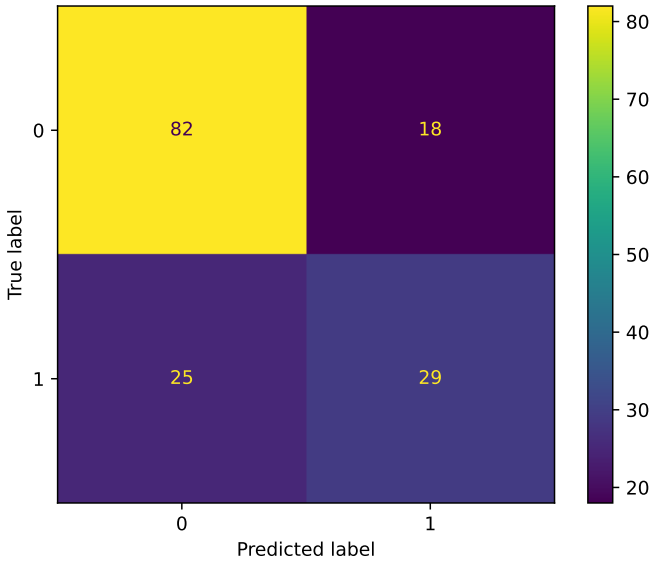


شکل ۱۳: درخت تولید شده با استفاده از معیار information gain

درخت بالا با حداکثر عمق 10 و تعداد 5 bin طراحی شده است و دقت آن برای داده‌های train برابر با 91.69% و برای داده‌های test برابر با 72.08% است و مقدار متوسط آن پس از 100 بار اجرا روی داده‌های تست برابر با 69.92% است.



شکل ۱۴: ماتریکس confusion برای داده‌های train



شکل ۱۵: ماتریکس confusion برای داده‌های test

۶ نتایج نهایی:

با چندین بار اجرای برنامه برای هر 2 معیار gini index و information gain متوجه شدم گرفتن دقت بالای 80% برای داده‌های test تقریبا با روش decision tree ناممکن است و حتی کتابخانه‌ی معتبر و قدیمی sklearn نیز در بهترین حالت دقتی برابر با 79% درصد داشت و به نظر می‌رسد دقت موردانتظار این الگوریتم با معیارهایی مانند gini index و information gain باید حدود 70% باشد. با این حال با نگاهی به داده‌ی دیابت متوجه می‌شویم بسیاری از داده‌ها ثبت نشده‌اند و به عنوان مثال افراد زیادی دارای فشار خونی و ضخامت پوست و BMI صفر هستند و می‌توانستم این مقادیر را در نظر نگیرم و یا مقادیر 0 را با میانگین آن‌ها جایگزین کنم ولی در آن صورت درخت من general نخواهد بود و برای همین dataset خاص طراحی شده و باید برای هر dataset یک کد جدید یا تغییراتی روی کد قبلی داد.

در نهایت راجع به موضوع overfit متوجه شدم هرچه تعداد bin ها و حداکثر عمق ممکن بیشتر باشد پدیده‌ی overfit بیشتر رخ می‌دهد که این اتفاق از نتیجه‌های بالا مشخص است که تنها افزایش حداکثر عمق ممکن باعث شد دقت بر روی داده‌های train افزایش چشمگیری داشته باشد و در طرف دیگر دقت بر روی داده‌های test کاهش یابد. برای کاهش مشکل overfit حداکثر عمق ممکن را برای هر دو معیار را برابر با 3 قرار دادم و تعداد bin ها را نیز برابر با 5 قرار دادم. این 2 hyper parameter نسبت به بقیه نتایج بهتری کسب کردند.

در ادامه با مقایسه بین 2 معیار gini index و information gain متوجه شدم با اینکه ممکن

است دقت information gain تا حد کمی بیشتر باشد ولی هزینه‌ی محاسبات آن نیز بیشتر از معیار gini index است و باید به نحوی بین دقت بالاتر و محاسبات بیشتر trade-off انجام شود. البته با توجه به نتایج بالا متوجه می‌شویم انجام دادن محاسبات بیشتر برای افزایش دقت به اندازه‌ی 1 یا 2 درصد ارزش ندارد.

۷ فایل‌های ارسال شده:

تمام کدهای اجرایی همراه با عکس‌ها و فایل README برای اجرای کد در فایل زیپ فرستاده شده قرار داده شده‌اند.