به نام خدا


# تمرین پیاده‌سازی عملی دوم هوش مصنوعی

## استاد: دکتر عبدی هجراندوست

## پیاده‌سازی دسته‌بند درخت تصمیم

محمدصادق مجیدی یزدی ۹۸۱۰۶۰۰۴

# فهرست عناوین

# نکات کلی و ملزومات اجرای پروژه

در پیاده‌سازی پیش رو از کتابخانه‌های numpy و pandas برای کار با مجموعه‌داده‌ها و انجام محاسبات و عملیات‌های لازم استفاده شده است و در واقع اسکلت اصلی این پروژه بر مبنای این دو کتابخانه و استفاده از توابع پایه‌ای ولی کاربردی آن‌ها بنا شده است. همچنین از کتابخانه‌های جانبی sklearn برای جداسازی داده‌های آموزشی از تست در مجموعه‌داده و همچنین مقایسه دقت مدل با یک مدل مورداستفاده و محبوب استفاده شده است. کتابخانه‌های graphviz و gvgen برای تولید خروجی گرافیکی از درخت و کتابخانه anytree برای ایجاد نمایش متنی درخت در صورت نیاز به کار گرفته شده‌اند. برای نصب این کتابخانه‌ها کافیست پس از ایجاد یک python virtual event و اتصال به آن محیط، دستور زیر را در خط فرمان وارد کنید تا عملیات نصب به‌طور خودکار انجام شود.

pip install -r requirements.txt  [python3 pip install -r requirements.txt]

البته استفاده از یک محیط توسعه پایتون مانند pycharm باعث تسهیل در اجرای این مراحل خواهد شد. در صورت وجود مشکل اجرای دستا این دستورات می‌تواند کمک‌کننده باشد.

pip install numpy

pip install pandas

pip install anytree

pip install graphviz

pip install GvGen

pip install scikit-learn

پس از آماده شدن محیط اجرای برنامه در صورت استفاده از ترمینال برای اجرا کافیست دستور زیر را اجرا کنید.

python3 filename.py

با اجرای برنامه، خروجی برنامه به صورت گرافیکی در قالب فایل "Decision Tree Result.pdf" تولید و به شما نمایش داده می‌شود و همچنین خروجی متنی درخت به فرمتی خوانا در قالب فایل "decision_tree_res.txt" در کنار فایل اصلی پروژه ایجاد می‌شود. همچنین در بخش دوم دقت مدل برای داده‌های آموزشی و تستی در دو خط جداگانه در خروجی پرینت می‌شوند.

دقت کنید که تولیدکننده گرافیکی گراف یا همان graphviz علاوه بر نصب کتابخانه نیاز به نصب موتور رندر روی سیستم شما دارد که در محیط ویندوز با دانلود مستقیم از سایت و تنظیم متغیر PATH و در محیط لینوکس با کامند زیر قابل انجام است:

```
sudo apt install graphviz
```

فرمت خروجی متنی درخت به صورت کلی زیر است:

```
<< Feature i >>  Entropy = x Gain =y Remainder = z
├── label: val 1
│   └── << output >>  Entropy = 0.0000
├── label: val 2
│   └── << output >>  Entropy = 0.0000
└── label: val 3
    └── << Feature j >>  Entropy = x Gain = y Remainder = z
        ├── label: val 1
        │   └── << output >>  Entropy = 0.0000
        └── label: val2
            └── << Feature k >>  Entropy = x Gain =y Remainder = z
                ...
```

در صورت بروز هرگونه مشکلی در تولید خروجی گرافیکی، خروجی متنی همواره در دسترس خواهد بود. فایل decision_tree_restaurant.py مربوط به بخش اول، فایل decision_tree_diabetes.py مربوط به بخش دوم و فایل test.py مربوط به تست مدل کتابخانه‌ای پایتون است.

## بخش اول: مجموعه داده رستوران

این مجموعه‌داده متشکل از ۱۲ داده است که هر یک شامل ۱۰ ویژگی با مقادیر اسمی و گسسته و یک برچسب خروجی باینری هستند.

| Example | Input Attributes | | | | | | | | | | Goal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Alt* | *Bar* | *Fri* | *Hun* | *Pat* | *Price* | *Rain* | *Res* | *Type* | *Est* | *WillWait* |
| $\mathbf{x}_1$ | *Yes* | *No* | *No* | *Yes* | *Some* | *$$$* | *No* | *Yes* | *French* | *0–10* | $y_1 = Yes$ |
| $\mathbf{x}_2$ | *Yes* | *No* | *No* | *Yes* | *Full* | *$* | *No* | *No* | *Thai* | *30–60* | $y_2 = No$ |
| $\mathbf{x}_3$ | *No* | *Yes* | *No* | *No* | *Some* | *$* | *No* | *No* | *Burger* | *0–10* | $y_3 = Yes$ |
| $\mathbf{x}_4$ | *Yes* | *No* | *Yes* | *Yes* | *Full* | *$* | *Yes* | *No* | *Thai* | *10–30* | $y_4 = Yes$ |
| $\mathbf{x}_5$ | *Yes* | *No* | *Yes* | *No* | *Full* | *$$$* | *No* | *Yes* | *French* | *>60* | $y_5 = No$ |
| $\mathbf{x}_6$ | *No* | *Yes* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Italian* | *0–10* | $y_6 = Yes$ |
| $\mathbf{x}_7$ | *No* | *Yes* | *No* | *No* | *None* | *$* | *Yes* | *No* | *Burger* | *0–10* | $y_7 = No$ |
| $\mathbf{x}_8$ | *No* | *No* | *No* | *Yes* | *Some* | *$$* | *Yes* | *Yes* | *Thai* | *0–10* | $y_8 = Yes$ |
| $\mathbf{x}_9$ | *No* | *Yes* | *Yes* | *No* | *Full* | *$* | *Yes* | *No* | *Burger* | *>60* | $y_9 = No$ |
| $\mathbf{x}_{10}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$$$* | *No* | *Yes* | *Italian* | *10–30* | $y_{10} = No$ |
| $\mathbf{x}_{11}$ | *No* | *No* | *No* | *No* | *None* | *$* | *No* | *No* | *Thai* | *0–10* | $y_{11} = No$ |
| $\mathbf{x}_{12}$ | *Yes* | *Yes* | *Yes* | *Yes* | *Full* | *$* | *No* | *No* | *Burger* | *30–60* | $y_{12} = Yes$ |

برای تشکیل درخت تصمیم از این داده‌ها نیاز است که ابتدا داده‌های اسمی را به صورت عددی درآوریم. برای این کار چون سایز مجموعه داده بسیار کوچک است می‌توانیم در حین تشکیل فایل مجموعه داده و بدون نیاز به زدن کد برای این تبدیلات، مقادیر اسمی را به مقادیر عددی گسسته نظیر کنیم.

```
TRANSLATOR = dict.fromkeys(['Alt', 'Bar', 'Fri', 'Hun', 'Rain', 'Res', 'y_label'], {0: 'No', 1: 'Yes'})
TRANSLATOR.update({
    'Pat': {0: 'None', 1: 'Some', 2: 'Full'},
    'Price': {0: '$', 1: '$$', 2: '$$$'},
    'Type': {0: 'French', 1: 'Thai', 2: 'Burger', 3: 'Italian'},
    'Est': {0: '0-10', 1: '10-30', 2: '30-60', 3: '>60'}
})
```

حال که داده‌ها برای تولید درخت تصمیم آماده هستند، مراحل تشکیل درخت را گام به‌گام بررسی می‌کنیم.

```
class Node:
    def __init__(self, entropy):
        self.entropy = entropy


class InteriorNode(Node):
    def __init__(self, attr, entropy, gain, remainder):
        super().__init__(entropy)
        self.attribute = attr
        self.entropy = entropy
        self.gain = gain
        self.remainder = remainder
        self.childs = dict()

    def add_child(self, val, tree):
        self.childs[val] = tree

    def __str__(self):
        return str('<< ' + self.attribute + ' >>\n\n' + 'Entropy = ' + '%.4f' % self.entropy
                   + '\n' + 'Gain = ' + '%.4f' % self.gain
                   + '\n' + 'Remainder = ' + '%.4f' % self.remainder)
```

```
class LeafNode(Node):
    def __init__(self, entropy, label):
        super().__init__(entropy)
        self.label = label

    def __str__(self):
        return '<< ' + TRANSLATOR['y_label'][self.label] + ' >>\n\n' + 'Entropy = ' + '%.4f' %
```

این دو کلاس Node شامل Interior و Leaf داده‌ساختار اصلی ما برای ساخت درخت هستند.

کلاس InteriorNode مربوط به راس‌های میانی درخت که دارای فرزند هستند و در واقع محل تصمیم گیری بر اساس یک ویژگی در درخت تصمیم می‌باشند، است. این کلاس ویژگی‌ای که قرار است تصمیم گیری بر اساس آن صورت بگیرد را به همراه مقدار آنتروپی، دست‌آورد اطلاعات و Remainder آن ویژگی در بین داده‌های آن مرحله در خود ذخیره می‌کند. همچنین به ازای هر مقدار آن ویژگی، زیر درختی که به سمت آن حرکت خواهیم کرد را به عنوان فرزند ذخیره می‌کنیم. کلاس

LeafNode نیز مربوط به برگ‌هایی است که تمام داده‌ها در آن مرحله خروجی مشخص ۰ یا ۱ خواهند داشت و در واقع در این مرحله از درخت تصمیم گیری کامل می‌شود. این درخت تصمیم، شامل مسیرهای تصمیمی که از طریق داده‌های آموزش قابل استخراج نیستند، نخواهد بود و آن مسیرها در درخت نهایی نمایش داده نمی‌شوند(چون در این بخش از ما صرفا درخت تصمیم بر حسب کل داده‌ها خواسته شده و نیازی به پیش‌بینی داده تستی نداریم تصمیم گیری در مورد حالاتی که در درخت حاضر نیستند لزومی ندارد).

کلاس DecisionTreeClassifier مدل اصلی ما برای یاد گرفتن درخت تصمیم است. این مدل علاوه بر یک هایپر پارامتر max_depth که بعدا توضیح داده خواهد شد، شامل توابع مهمی است که در ادامه کاربرد هر یک را توضیح می‌دهیم:

```python
def calc_binary_entropy(self, p):
    if p == 0 or p == 1:
        return 0
    return -(p * np.log2(p) + (1 - p) * np.log2(1 - p))
```

این تابع مقدار آنتروپی متغیر تصادفی برنولی(دودویی) با احتمال p را محاسبه می کند. چون که خروجی مجموعه داده ما دودویی است این تابع برای محاسبه آنتروپی مناسب است.

```python
def calc_remainder(self, X, feature, y_label):
    vals = X[feature].unique()
    p = len(X[X[y_label] == 1])
    n = len(X[X[y_label] == 0])
    rem = 0.0

    for k in vals:
        pk = len(X[(X[feature] == k) & (X[y_label] == 1)])
        nk = len(X[(X[feature] == k) & (X[y_label] == 0)])
        rem += np.divide(pk + nk, p + n) * self.calc_binary_entropy(np.divide(pk, pk + nk))

    return rem
```

این تابع مقدار remainder را برای استفاده در محاسبه دست‌آورد اطلاعات بر اساس رابطه اشاره‌شده در اسلاید محاسبه می‌کند. این تابع دیتاست و ویژگی مورد بررسی در آن مرحله را به عنوان ورودی دریافت و به ازای هر یک از مقادیر ممکن (موجود در دیتاست) آن ویژگی، مجموع حاصل‌ضرب نسبت تعداد داده‌های دارای آن مقدار برای ویژگی موردنظر به کل داده‌ها در آنتروپی آن داده‌ها به دست می‌آورد. مقادیری از ویژگی که در دیتاست وجود ندارند عملا به صورت ۰ با حاصل جمع شده و تاثیری ندارند. ورودی‌های y_label در این توابع در واقع برچسب یا اسم ستون خروجی دیتاست هستند که برای جامعیت بیشتر کد به صورت پارامتری در نظر گرفته شده‌اند.

```
def calc_info_gain(self, X, feature, y_label):
    p = len(X[X[y_label] == 1])
    n = len(X[X[y_label] == 0])
    entropy = self.calc_binary_entropy(np.divide(p, p + n))
    remainder = self.calc_remainder(X, feature, y_label)
    return entropy - remainder, remainder
```

این تابع وظیفه محاسبه مقدار دست‌آورد اطلاعات به ازای یک ویژگی روی مجموعه داده را دارد. این تابع دیتاست و ویژگی را به عنوان ورودی می‌گیرد. سپس از طریق فرمول $Gain = Entropy - Remainder$ مقدار دست‌آورد را محاسبه کرده و آن را به همراه remainder (صرفا برای ذخیره‌سازی) خروجی می‌دهیم.

```
def select_feature(self, X, features, y_label):
    selected_feature = None
    gain = -np.inf
    remainder = 0

    for feature in features:
        _gain, _remainder = self.calc_info_gain(X, feature, y_label)
        if _gain > gain:
            selected_feature = feature
            gain = _gain
            remainder = _remainder

    return selected_feature, gain, remainder
```

این تابع با ورودی گرفتن دیتاست و ویژگی‌های موجود در آن مرحله، به ازای تمام ویژگی‌ها دست‌آورد اطلاعات را محاسبه کرده و نهایتا ویژگی دارای بیشینه دست‌آورد را به همراه مقدار دست‌آورد و remainder آن ویژگی خروجی می‌دهد تا به عنوان ملاکی تصمیم گیری این راس از درخت استفاده شود.

```
@staticmethod
def check_all_have_same_classification(x):
    a = x.to_numpy()
    return (a[0] == a).all()
```

این تابع بررسی میکند که آیا برچسب خروجی تمام داده‌های موجود در مجموعه‌داده داده‌شده به عنوان ورودی یکسان هستند یا خیر.

الگوریتم کلی ساخت درخت تصمیم در اسلاید به صورت زیر بود:

**function** DECISION-TREE-LEARNING(*examples*, *attributes*, *parent_examples*) **returns** a tree

**if** *examples* is empty **then return** PLURALITY-VALUE(*parent_examples*)
**else if** all *examples* have the same classification **then return** the classification
**else if** *attributes* is empty **then return** PLURALITY-VALUE(*examples*)
**else**
  $A \leftarrow \text{argmax}_{a \in attributes}$ IMPORTANCE($a$, *examples*)
  *tree* $\leftarrow$ a new decision tree with root test $A$
  **for each** value $v_k$ of $A$ **do**
    $exs \leftarrow \{e : e \in examples$ **and** $e.A = v_k\}$
    *subtree* $\leftarrow$ DECISION-TREE-LEARNING($exs$, $attributes - A$, *examples*)
    add a branch to *tree* with label $(A = v_k)$ and subtree *subtree*
  **return** *tree*

حال تابع زیر را دقیقا به همین صورت با یک تغییر جزیی پیاده می کنیم:

```
def create_decision_tree(self, X, features, y_label, parent_Xs, depth):
    if X.empty:
        entropy, label = self.calc_plurality_value(parent_Xs, y_label)
        return LeafNode(entropy, label)

    if DecisionTreeClassifier.check_all_have_same_classification(X[y_label]):
        return LeafNode(0, X[y_label].iloc[0])

    if not features or (self.max_depth and depth == self.max_depth):
        entropy, label = self.calc_plurality_value(X, y_label)
        return LeafNode(entropy, label)

    feature, gain, remainder = self.select_feature(X, features, y_label)
    root = InteriorNode(feature, gain + remainder, gain, remainder)
    vals = X[feature].unique()

    for val in vals:
        new_feats = features.copy()
        new_feats.remove(feature)
        subtree = self.create_decision_tree(X[X[feature] == val], new_feats, y_label, X, depth + 1)
        root.add_child(val, subtree)

    return root
```

تغییر جزیی این است که یک شرط دیگر برای ایجاد برگ در درخت تصمیم، علاوه بر حالتی که تمام داده‌ها برچسب یکسان داشته باشند و یا هیچ ویژگی دیگری برای بررسی باقی نمانده باشد، حالتی که درخت به یک عمق خاصی رسیده باشد هم در نظر گرفته و حتی اگر ویژگی‌های دیگری هم باقی‌مانده باشد آن زیردرخت را به دلیل رسیدن به حداکثر عمق مجاز پایان داده و مقدار مد (plurality value) را به عنوان برگ درخت قرار می‌دهیم. با تنظیم کردن یک مقدار مناسب عمق برای درخت بسته به شرایط مسئله و مجموعه داده می‌تواند تاثیر مثبتی بر دقت مدل درخت تصمیم گذاشته و از بیش‌برازش یا همان overfit شدن درخت تا حد خوبی جلوگیری کند.

```
def calc_plurality_value(self, X, y_label):
    p = len(X[X[y_label] == 1])
    n = len(X[X[y_label] == 0])
    entropy = self.calc_binary_entropy(np.divide(p, p + n))
    mode = p if p >= n else n
    return mode, entropy
```

این تابع برای محاسبه plurality_value استفاده شده و با گرفتن مجموعه‌داده به عنوان ورودی، مقدار مد در بین خروجی‌ها را به عنوان plurality_value به همراه مقدار آنتروپی در حالت فعلی برای ذخیره‌سازی در برگ درخت خروجی می‌دهد.

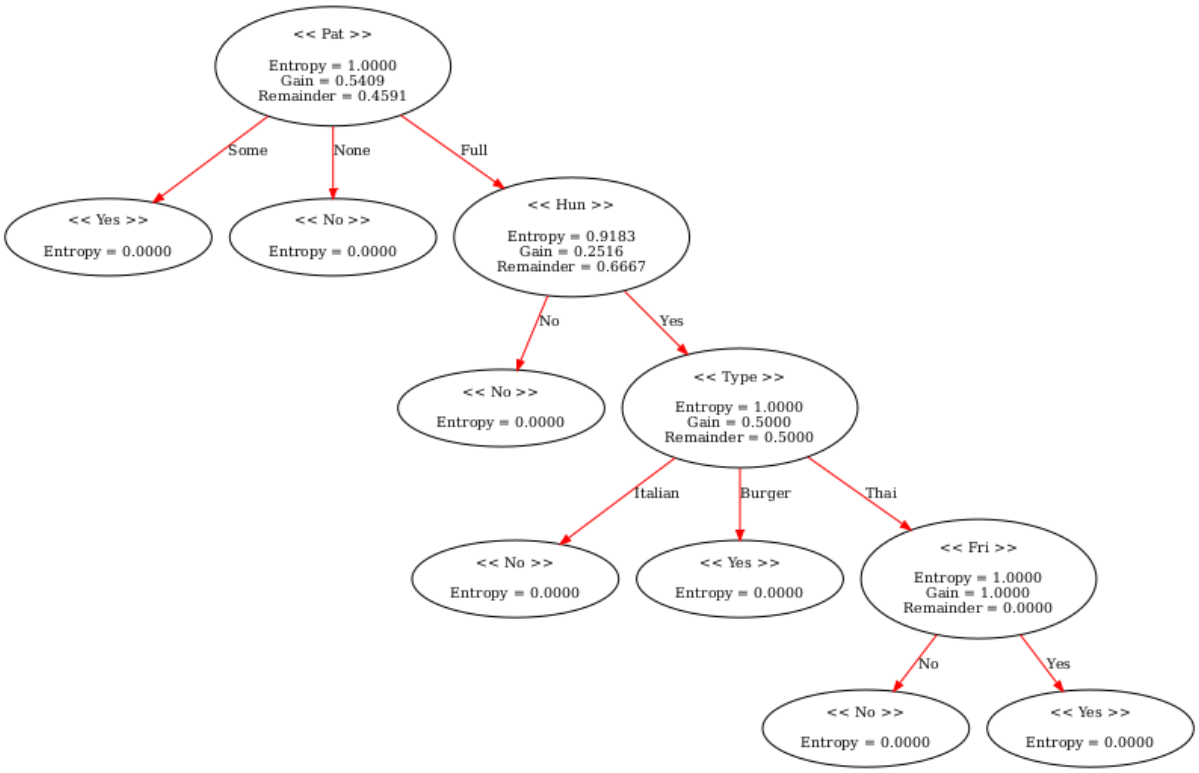
تابع fit به عنوان یک wrapper روی تابع بالا برای ساخت درخت نوشته شده است و در عمل این تابع از بیرون برای تشکیل درخت صدا زده ی‌شود. دو تابع make_tree_graphviz و show_graphical_deciosion_tree_result برای تولید خروجی گرافیکی از درخت یادگرفته‌شده توسط مدل به کار می‌روند. همچنین دو تابع draw_tree و show_decision_tree برای تولید خروجی متنی درخت استفاده می‌شوند. در هر دو حالت با یک پیمایش اول عمق درخت را با دیتا استراکچر مورد استفاده در هر یک از کتابخانه‌ها بازسازی کرده و آن را رندر می کنیم.

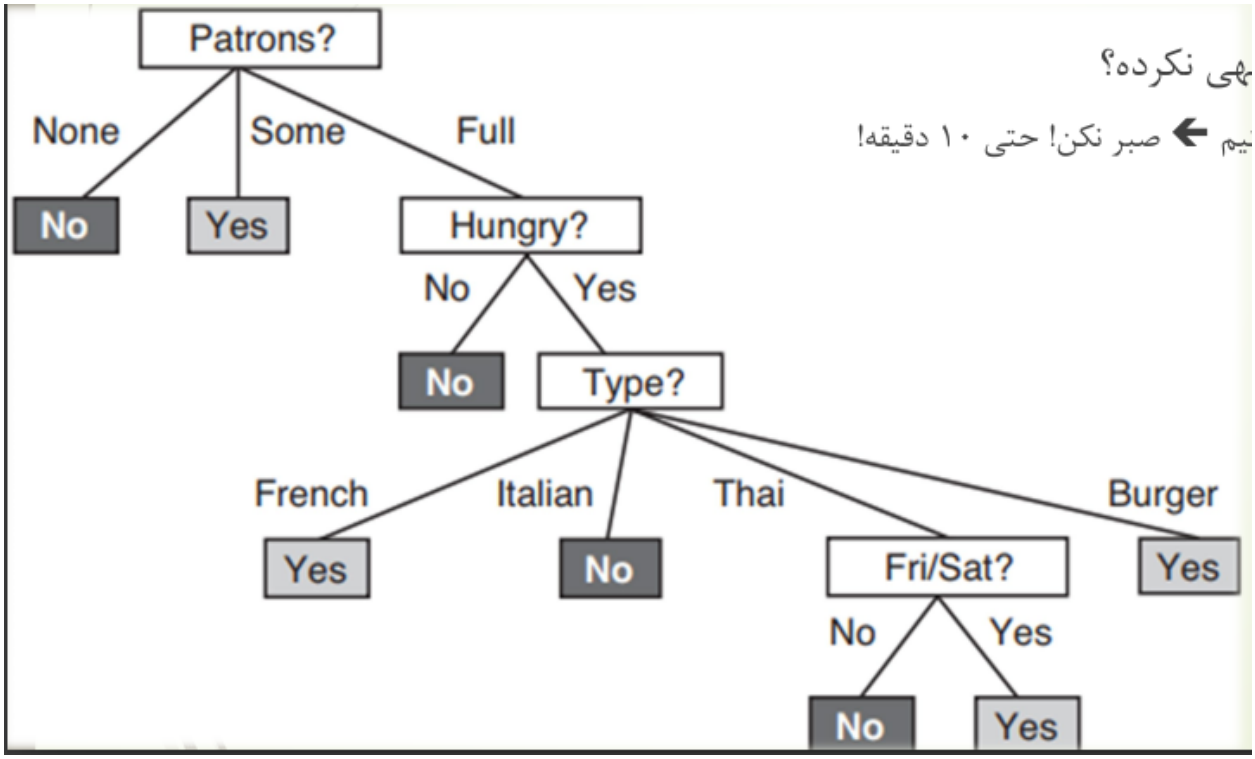
حال کافیست مجموعه داده را با استفاده از ابزار کتابخانه pandas لود کرده و مدل نوشته‌شده را با استفاده از کل داده‌ها آموزش دهیم. تکه کد مربوط به این قسمت در زیر آمده است. (به دلیل کوچک بودن دیتاست و مشاهده صرف عملکرد درخت تصمیم و عدم اهمیت بیش‌برازش در این بخش مقداری برای عمق بیشینه در نظر نگرفته و محدودیتی روی عمق درخت نداریم)

```
rest_df = pd.read_csv('restaurant.csv')
dtc = DecisionTreeClassifier()
dtc.fit(rest_df, 'Goal')
dtc.show_decision_tree()
dtc.show_graphical_decision_tree_result()
```

که به ترتیب پس از لود شدن دیتاست، نمونه‌ای از مدل ساخته شده و روی داده‌ها آموزش داده می‌شود و نهایتا خروجی متنی و سپس گرافیکی تولید می‌شود.

حال خروجی مدل نوشته‌شده را با خروجی مورد نظر در اسلاید مقایسه می کنیم:


<< Pat >>
Entropy = 1.0000
Gain = 0.5409
Remainder = 0.4591
Some
None
Full
<< Yes >>
Entropy = 0.0000
<< No >>
Entropy = 0.0000
<< Hun >>
Entropy = 0.9183
Gain = 0.2516
Remainder = 0.6667
No
Yes
<< No >>
Entropy = 0.0000
<< Type >>
Entropy = 1.0000
Gain = 0.5000
Remainder = 0.5000
Italian
Burger
Thai
<< No >>
Entropy = 0.0000
<< Yes >>
Entropy = 0.0000
<< Fri >>
Entropy = 1.0000
Gain = 1.0000
Remainder = 0.0000
No
Yes
<< No >>
Entropy = 0.0000
<< Yes >>
Entropy = 0.0000


Patrons?
None
Some
Full
No
Yes
Hungry?
No
Yes
No
Type?
French
Italian
Thai
Burger
Yes
No
Fri/Sat?
Yes
No
Yes
No
Yes

که مشاهده می‌کنیم هر دو خروجی برای درخت تصمیم کاملا مشابه هستند. درباره‌ی شاخه French از ویژگی Type در درخت تصمیم که در اسلاید آمده ولی در خروجی ما وجود ندارد، طبق بررسی دستی و مرحله به مرحله‌ای که در تولید درخت کردم، در مجموعه داده راس تصمیم Type داده‌ای با نوع French وجود ندارد و بنابراین در خروجی درخت تصمیم مشاهده نخواهد شد یا به بیان دیگر برگی مجزا برای آن در نظر نمی‌گیریم و در هنگام پیش‌بینی هم در صورت برخورد به چنین موردی صرفا مقدار plurality value ذخیره‌شده در آن راس تصمیم را به عنوان خروجی مدل می‌دهیم.

```
In [3]: import pandas as pd
        rdf = pd.read_csv('restaurant.csv')
        rdf[(rdf['Pat'] == 2) & (rdf['Hun'] == 1)]
```

Out[3]:

| | Alt | Bar | Fri | Hun | Pat | Price | Rain | Res | Type | Est | Goal |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 1 | 2 | 0 |
| **3** | 1 | 0 | 1 | 1 | 2 | 0 | 1 | 0 | 1 | 1 | 1 |
| **9** | 1 | 1 | 1 | 1 | 2 | 2 | 0 | 1 | 3 | 1 | 0 |
| **11** | 1 | 1 | 1 | 1 | 2 | 0 | 0 | 0 | 2 | 2 | 1 |

مشاهده کردیم که در بین داده‌های این مرحله غذایی با $Type = 0\ (French)$ وجود ندارد و امکان تولید برگ مجزا برای آن نیست.

## بخش دوم: مجموعه داده دیابت

این مجموعه‌داده شامل ۷۶۸ داده با ۸ ستون فیچر عددی پیوسته و یک ستون باینری به عنوان برچسب خروجی است. برای تشکیل درخت تصمیم از روی این داده‌ها نیاز داریم به فیچرها مقادیر گسسته‌ای نسبت دهیم. برای این منظور از همان روش ساده دسته‌بندی داده‌های آموزش در هر فیچر به تعدادی بازه با طول مساوی استفاده می‌کنیم.

در ضمن ابتدا داده‌ها را با استفاده از تابع train_test_split کتابخانه sklearn به صورت کاملا رندوم و به نسبت ۰.۸ به ۰.۲ به دو بخش آموزش (train) و آزمایش (test) تقسیم می‌کنیم.

```
X_train, X_test, y_train, y_test = train_test_split(
    diabetes_df.drop(columns=[y_label]), diabetes_df[y_label], test_size=test_size, shuffle=True)
X_train[y_label] = y_train
X_test[y_label] = y_test
```

حال به صورت دستی به ازای یک ستون بازه‌های مورد نیاز برای گسسته‌سازی داده‌ها بین تعداد ثابتی سبد را تولید می‌کنیم.

```
def create_bins_boundaries(df, feature, bins_num: int):
    min_val = df[feature].min()
    max_val = df[feature].max()
    step = np.divide(max_val - min_val, bins_num)
    bins = [-np.inf]
    for i in range(bins_num):
        bins.append(round(min_val + step * i, 3))
    bins.append(round(max_val, 3))
    bins.append(np.inf)
    return bins
```

این تابع با ورودی گرفتن دیتاست و ویژگی مورد نظر و تعداد سبد خواسته شده، طول بازه بین مقادیر کمینه و بیشینه آن ویژگی در دیتاست را به تعداد سبد تقسیم می‌کند تا طول بازه‌ها مشخص شود. نهایتا بازه‌ها به صورت زیر تشکیل می‌شوند:

(-infinity, min], (min, min+step], (min+step, min+2*step], … , (max, infinity)

که علاوه بر تعداد بازه‌های خواسته‌شده، شامل دو بازه اضافی‌تر مربوط به مقادیر کمتر از مینیمم و بیشتر از ماکزیمم است تا مقادیر احتمالی خارج از بازه مینیمم و ماکزیمم در بین داده‌های تست مشکل‌ساز نشود و درخت کارایی خود را حفظ کند.

```
def make_feature_discrete(df, feature, bins: list):
    labels = [i for i in range(len(bins) - 1)]
    df[feature + '-discrete'] = pd.cut(df[feature], bins, labels=labels)
```

این تابع با ورودی گرفتن مرزهای بازه‌های جداسازی، یک ستون خاص از دیتاست را گسسته‌سازی کرده و آن را با اسمی جدید به دیتاست اضافه می‌کند.

```
def make_data_discrete(df: pd.DataFrame, test_df: pd.DataFrame):
    original_cols = list(X_train.columns)[:-1]
    for feature in list(df.columns)[:-1]:
        bins = create_bins_boundaries(df, feature, NUM_OF_BINS)
        make_feature_discrete(df, feature, bins)
        make_feature_discrete(test_df, feature, bins)

    return df.drop(columns=original_cols)
```

این تابع با دریافت هر دو مجموعه داده‌های آموزشی و آزمایشی، به ازای هر ستون ویژگی، ابتدا مرزهای جداسازی را بر اساس داده‌های آموزشی تعیین کرده و سپس داده‌های هر دو دیتاست را بر اساس این داده‌ها جداسازی می‌کند و تحت ستون‌های جدید به آن‌ها اضافه می‌کند.

کلاس‌های نوشته‌شده برای پیاده‌سازی داده‌ساختار درخت، مشابه بخش قبل هستند. با این تفاوت که برای هر یک از راس‌های میانی یک مقدار به عنوان مقدار پیش‌بینی default نگه‌داری می‌شود که در هنگام انجام پیش‌بینی با این درخت، وقتی به این راس برسیم و نتوانیم از هیچ یک از مسیرهای موجود به مسیر پیش‌بینی ادامه دهیم،این مقدار پیش‌فرض به عنوان نتیجه پیش‌بینی درخت برخواهد گشت. این مقدار برابر با مد(خروجی با تعداد بیشتر) در بین داده‌های آموزش در آن مرحله از یادگیری درخت است.

مدل نوشته‌شده برای یادگیری درخت تصمیم در این بخش هم مانند بخش قبل است و فقط یک تابع برای اجرای پیش‌بینی خروجی برای داده‌های جدید تست به آن اضافه شده و همچنین در هنگام یادگیری مقدار پیش‌فرض به نودهای میانی تصمیم گیری اضافه می‌شود.

```
@staticmethod
def predict_single_sample(sample, tree_node):
    if isinstance(tree_node, LeafNode):
        return tree_node.label

    if isinstance(tree_node, InteriorNode):
        val = sample[tree_node.attribute]
        if val in tree_node.childs:
            return DecisionTreeClassifier.predict_single_sample(sample, tree_node.childs[val])
        else:
            return tree_node.default

def predict(self, test_X):
    result = []

    for i in range(len(test_X)):
        result.append(DecisionTreeClassifier.predict_single_sample(test_X.iloc[i], self.trained_tree))

    return np.array(result)
```

توابع بالا برای پیش‌بینی کردن برچسب خروجی روی داده‌های تست داده‌شده به درخت تصمیم پس از یادگیری نوشته شده است. تابع predict_single_sample با گرفتن یک نمونه ورودی تست، بر اساس ویژگی‌های این نمونه درخت تصمیم را پیمایش می‌کند و هرجا به نتیجه مورد نظر برسد، برچسب پیش‌بینی‌شده را خروجی می‌دهد. در این پیمایش اگر به یک راس برگ رسیدیم مقدار برگ را مستقیما خروجی می‌دهیم. اگر در یک راس میانی بودیم بر اساس مقدار ویژگی تصمیم آن برگ به سمت زیردرخت درست حرکت می‌کنیم و اگر زیردرختی برای آن مقدار در آن راس پیدا نشد، مقدار پیش‌فرض آن راس را به عنوان برچسب خروجی می‌دهیم.

تابع accuracy_score مقدار صحت پیش‌بینی را محاسبه می‌کند. به این ترتیب که مجموع تعداد برچسب‌های درست پیش‌بینی‌شده را بر تعداد کل داده‌های تست تقسیم کرده و آن را به صورت درصد خروجی می‌دهد.

## بررسی‌ها، اجراها، نتایج و چالش‌ها

در ابتدا کمی داده‌ها و توزیع آن‌ها در بین ویژگی‌ها را بررسی می‌کنیم.

```
In [86]: X_train.describe()
```

Out[86]:

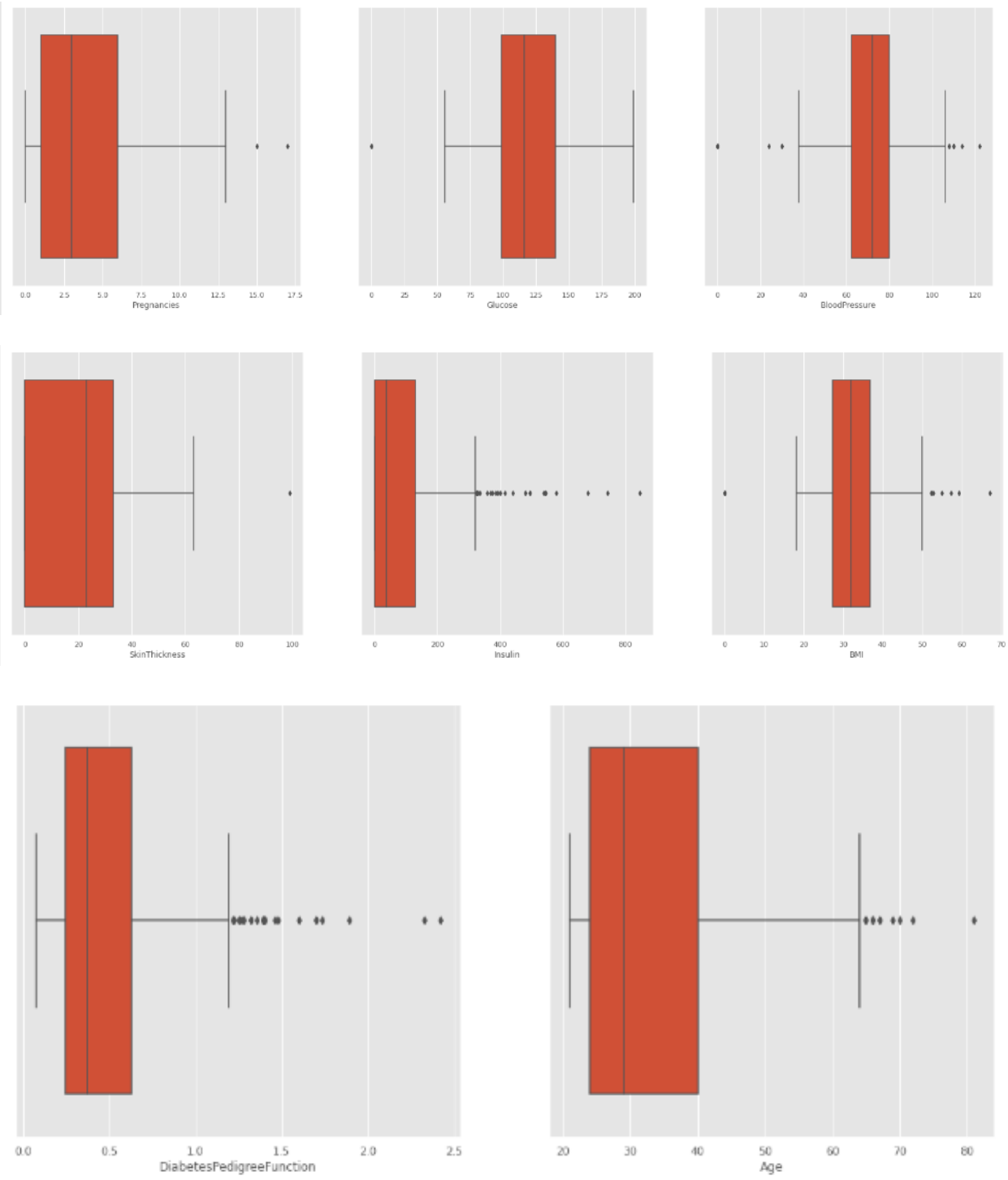
| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| count | 614.000000 | 614.000000 | 614.000000 | 614.000000 | 614.000000 | 614.000000 | 614.000000 | 614.000000 | 614.000000 |
| mean | 3.809446 | 120.762215 | 69.057003 | 21.001629 | 81.221498 | 32.116612 | 0.472601 | 33.135179 | 0.348534 |
| std | 3.368287 | 31.867468 | 19.972453 | 16.005454 | 115.724330 | 7.871292 | 0.324980 | 11.819039 | 0.476895 |
| min | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.078000 | 21.000000 | 0.000000 |
| 25% | 1.000000 | 99.000000 | 62.500000 | 0.000000 | 0.000000 | 27.400000 | 0.244250 | 24.000000 | 0.000000 |
| 50% | 3.000000 | 116.000000 | 72.000000 | 23.000000 | 37.500000 | 32.050000 | 0.372500 | 29.000000 | 0.000000 |
| 75% | 6.000000 | 140.000000 | 80.000000 | 33.000000 | 129.750000 | 36.775000 | 0.629750 | 40.000000 | 1.000000 |
| max | 17.000000 | 199.000000 | 122.000000 | 99.000000 | 846.000000 | 67.100000 | 2.420000 | 81.000000 | 1.000000 |

اطلاعات آماری داده‌های هر یک از ستون‌های دیتاست در بالا نمایش داده شده است. وجود داده‌های میسینگ(ناموجود یا غیرقابل‌قبول) را در دیتاست بررسی می‌کنیم.

```
In [87]: print(f'number of records: {len(X_train.index)}\n')
         print('number of missing values in each column:')
         print(X_train.isna().sum())
```

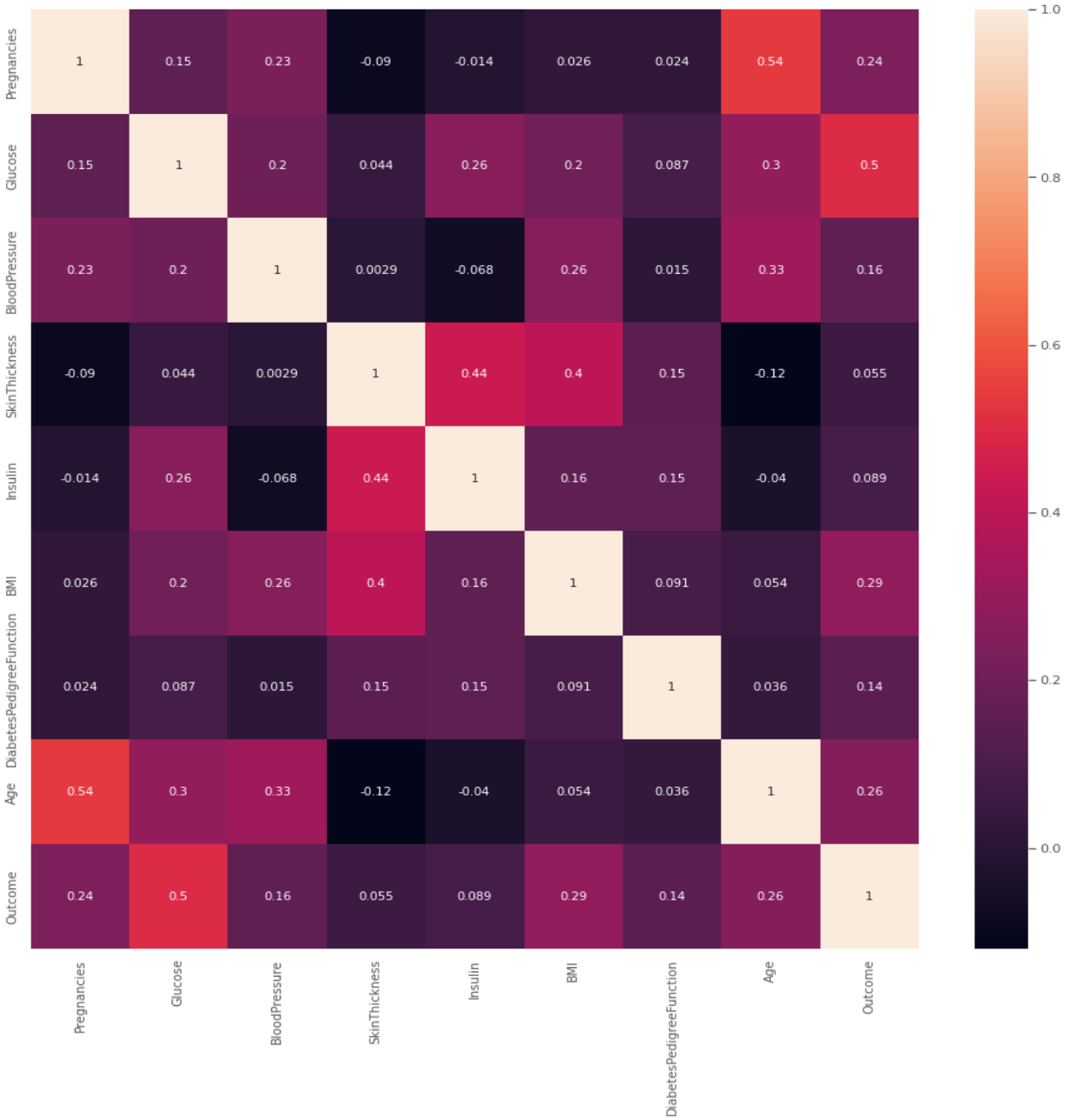
```
number of records: 614

number of missing values in each column:
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

که همانطور که مشخص است مقدار ناموجودی در بین داده‌های دیتاست وجود ندارد. حال خلاصه‌ای مفید از توزیع داده‌ها حول هر ویژگی بررسی می‌کنیم. از نمودار جعبه‌ای استفاده می‌کنیم تا میانه و چارک اول و سوم و مرزهای IQR و داده‌های خارج از این مرزها را بررسی کنیم.




همانطور که از نمودارها مشخص است، در برخی از ستون‌ها داده‌هایی با مقدار ۰ و یا خیلی کم وجود دارند که دور از واقعیت هستند و نشان می‌دهند داده‌های جمع‌آوری‌شده در دیتاست مقداری داده غیرقابل قبول دارند و صحت کامل ندارد. می‌توان به مواردی مثل 0 بودن BMI یا BloodPressure یا

Glucose و یا ستون‌ای دیگر اشاره کرد (که در این موارد عملا طرف مرده است!). همچنین در مواردی مانند Insulin تعداد زیادی داده خارج از مرزهای IQR وجود دارند که البته شاید با توجه به ماهیت دیتاست که مربوط به دیابت است نتوان آن‌ها را داده‌های خیلی پرتی برشمرد! ولی در کل چنین مواردی در یک دیتاست موجب می‌شود مدل درخت تصمیم یادگرفته‌شده روی آن اندکی کاهش دقت داشته باشد و بر عملکرد تاثیرگذار است.



این نمودار هیت‌مپ رسم‌شده نیز میزان همبستگی ویژگی‌های مختلف و خروجی با یکدیگر را نشان می‌دهد. با توجه به این نمودار، برچسب خروجی Outcome بیشترین همبستگی را با ویژگی Glucose دارد. پس انتظار داریم در ابتدا و ریشه درخت داده‌ها بر اساس این ویژگی دسته‌بندی شوند چون بیشترین دست‌آورد اطلاعات از این ویژگی خواهد بود.

حال اجراهای مختلفی را با ست کردن پارامترها انجام و نتایج را تحلیل می‌کنیم. ابتدا چندین اجرا با مقادیر $NUM\ OF\ BINS\ =\ 7$ و بدون تایین محدودیتی برای حداکثر عمق انجام می‌دهیم و نتایج به دست‌آمده غالبا به این صورت است که دقت روی داده‌های آموزش تقریبا ۱۰۰ (غالبا عددی نزدیک ۹۹) و روی داده‌های تست در اکثر اجراها در بازه ۵۷ تا ۷۰ درصد متغیر است و غالبا نزدیک به ۶۴ است. این موضوع نشان می‌دهد مقدار بسیار زیادی بیش‌برازش روی داده‌های آموزش به وجود آمده است. البته طبیعت درخت تصمیم به گونه‌ای است که مدلی با واریانس بالا و بایاس کم است و نتیجه یادگیری آن بسیار حساس به داده‌های آموزش است و طبیعی است در این مدل غالبا اگر حواسمان نباشد بیش‌برازش رخ دهد.

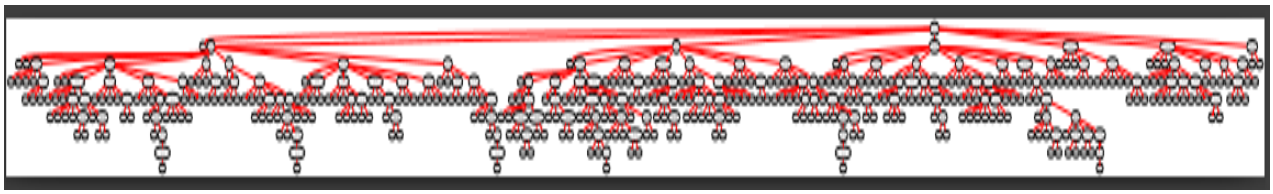
```
Test Set Accuracy(0): 61.68831168831169
Test Set Accuracy(1): 64.28571428571429
Test Set Accuracy(2): 55.1948051948052
Test Set Accuracy(3): 61.68831168831169
Test Set Accuracy(4): 57.14285714285714
Test Set Accuracy(5): 64.28571428571429
```

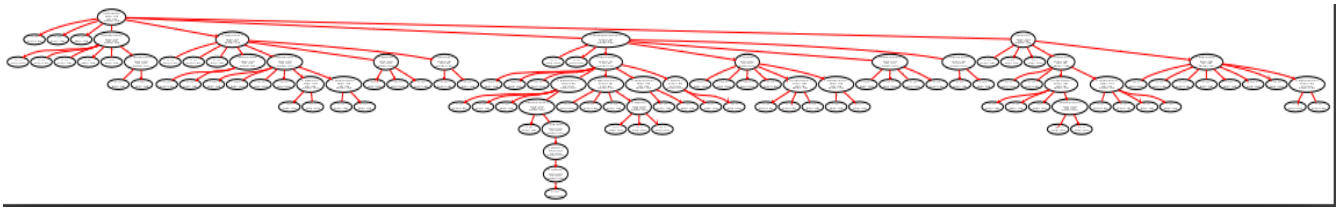
حال اگر با همان تعداد سبد برای جداسازی داده‌ها یعنی ۷، حداکثر عمق max_depth را برابر با ۵ قرار دهیم، دقت روی داده‌های آموزشی به حدود ۹۰ درصد کاهش می‌یابد و اندکی از بیش‌برازش کم می‌شود ولی دقت داده‌های تست در همان بازه ۵۷ تا ۷۰ درصد باقی می‌ماند و تغییر چندانی نمی‌کند. حال اگر با همین عمق ۵ تعداد سبد را به ۱۲ افزایش دهیم مجددا مقدار دقت روی داده‌های آموزش به ۱۰۰ نزدیک می‌شود و بیش‌برازش نمود بیشتری می‌کند. در صورتی که هم مقدار عمق کم مثلا ۴ و تعداد سبد کم مثلا ۳ به مدل داده شود، دقت هم روی داده‌های آموزش و هم روی داده‌های آزمایش کاهش چشمگیری می‌کند و در بازه ۳۰ تا ۴۰ قرار می‌گیرد.

از طرفی اگر روی همان مدل اولیه با تعداد سبد ۷ و عمق نامحدود، مقدار نسبت داده‌های تست و آموزش را تغییر دهیم، مشاهده می‌کنیم که حتی با قرار دادن نسبت کمی از داده‌ها در قسمت آموزش می‌توان درخت تصمیمی با دقت قابل‌قبول یاد گرفت. این مورد یکی از مزایای درخت تصمیم است که با تعداد داده کم برای یادگیری هم می‌تواند تطابق خوبی داشته باشد و دقت قابل قبولی از خود ارائه کند. البته کاهش تعداد داده‌های آموزشی باعث کم کردن مقدار بیش‌برازش نمی‌شود و درخت طبیعت خود را حفظ می‌کند. برای مثال با همان ستینگ اولیه و قرار دادن نسبت ۰.۸ داده‌ها برای آزمایش و فقط ۰.۲ برای آموزش، دقت روی داده‌های تست غالبا در بازه ۵۸ تا ۶۸ بود که قابل قبول است.

```
Test Set Accuracy: 66.01626016260163
Train Set Accuracy: 99.34640522875817
```

در ادامه تصویر خروجی ۳ اجرا از کد درخت تصمیم با سه مقدار دهی متفاوت به پارامترها آمده شده است. در تصویر اول مقدار NUM_OF_BINS برابر با ۷ و محدودیت عمق تعیین‌نشده و نسبت داده‌های تست ۰.۲ است. در تصویر دوم مقدار NUM_OF_BINS همان ۷ و محدودیت عمق max_depth برابر با ۵ و نسبت داده‌های تست برابر با ۰.۲ است. در تصویر آخر نیز مقدار پارامترها مانند حالت اول است و فقط نسبت داده‌های تست به ۰.۸ تغییر کرده است. توجه کنید که با توجه به رندوم بودن انتخاب داده‌های تست از دیتاست اصلی حتی با یک ستینگ ثابت نیز خروجی‌های متفاوتی در هر بار اجرا به دست می‌آید. همچنین تصاویر آورده شده از درخت‌ها به دلیل بزرگی درخت دارای کیفیت مناسبی نیستند و می‌توانید از فایل‌های ضمیمه‌شده از درخت‌های رسم‌شده برای زوم و حفظ کیفیت استفاده کنید.





با توجه به اجراهای متعدد انجام‌شده روی کد با پارامترهای مختلف، می‌توانیم نتایجی را به دست آوریم. اول اینکه مدل درخت تصمیم در کل مدلی با استعداد زیاد برای دچار شدن به بیش‌برازش است و یادگیری آن بسیار حساس به داده‌های آموزش انتخاب‌شده برای فرایند یادگیری است. برای کاهش این بیش‌برازش می‌توانیم عمق را بیشتر محدود کنیم و یا تعداد سبدها یا همان بازه‌های گسسته‌سازی مقادیر ویژگی‌های داده‌ها را کمتر کنیم. در کل هرچه عمق کمتر باشد بیش‌برازش کمتر و هرچه بیشتر باشد امکان رخداد بیش‌برازش بیشتر است. به همین صورت با افزایش تعداد بازه‌های گسسته‌سازی مقدار بیش‌برازش افزایش داشته و با کاهش آن کمتر می‌شود. همچنین به این نتیجه رسیدیم که درخت تصمیم با تعداد کمی داده آموزش هم می‌تواند با دقت خوبی یاد گرفته شود و در نتیجه در مواقعی که با داده‌های کمی می‌خواهیم به دقت نسبتا معقولی برسیم این مدل یک گزینه مناسب خواهد بود. همچنین در تمام حالات اولین راس تصمیم گیری مربوط به ویژگی Glucose بود که با شهود به دست‌آمده از نمودار Heatmap ما همخوانی دارد.

وقتی مشکلی به وجود می‌آید، راه‌حل‌های احتمالی هم در کنار آن خودنمایی می‌کنند. برای افزایش دقت مدل درخت تصمیم بر روی داده‌های تست و همچنین کمی کاستن از مقدار بیش‌برازش راه‌حل‌هایی مختلف به نظر من رسید که بعضی را آزمایش کردم و نتیجه‌ای نگرفتم و بعضی را صرفا ذکر می‌کنم. اولین راه‌حل من این بود که سعی کنم داده‌های پرت را که بالاتر در بررسی کلی دیتاست به

آن اشاره کردم حذف کرده و با مقادیر معقول‌تری جایگزین کنم. به این صورت عمل کردم که برای هر ستون با توجه به نمودار جعبه‌ای آن، بازه‌ای را به عنوان مقادیر قابل قبول گرفتم و سایر مقادیر خارج از این بازه را به عنوان outlier محسوب کرده و آن را با میانگین داده‌های آن ستون که مقدار مناسبی برای داده‌های عددی پیوسته است، جایگزین کردم.

```
def remove_outliers_by_IQR(df, column, lower_coef=1.5, upper_coef=1.5):
    q1 = df[column].quantile(0.25)
    q3 = df[column].quantile(0.75)
    IQR = q3 - q1
    df.where(df[[column]] <= q3 + (upper_coef * IQR), inplace=True)
    df.where(df[[column]] >= q1 - (lower_coef * IQR), inplace=True)


def pre_process_data(df: pd.DataFrame):
    df.where(df[list(df.columns)] >= 0, inplace=True)

    remove_outliers_by_IQR(df, 'Pregnancies')
    remove_outliers_by_IQR(df, 'SkinThickness')
    remove_outliers_by_IQR(df, 'BMI')
    remove_outliers_by_IQR(df, 'Glucose')
    remove_outliers_by_IQR(df, 'DiabetesPedigreeFunction', upper_coef=3)
    remove_outliers_by_IQR(df, 'Insulin', upper_coef=3)
    remove_outliers_by_IQR(df, 'BloodPressure', lower_coef=2.2, upper_coef=2.5)

    def fill_missing(x):
        x.fillna(x.mean(), inplace=True)

    df.apply(fill_missing)
```

با وجود صدا زدن pre_process_data روی دیتاست اصلی و حذف داده‌های پرت دقت مدل تغییر خاص زیادی نکرد و شاهد بهبودی نبودم. اوضاع آماری داده‌ها پس از پیش‌پردازش در زیر آمده:

| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| count | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 | 768.000000 |
| mean | 3.786649 | 121.686763 | 72.405184 | 20.434159 | 73.384717 | 32.204005 | 0.458731 | 33.240885 | 0.348958 |
| std | 3.270153 | 30.435949 | 12.096346 | 15.698281 | 98.352356 | 6.410480 | 0.295600 | 11.760232 | 0.476951 |
| min | 0.000000 | 44.000000 | 24.000000 | 0.000000 | 0.000000 | 18.200000 | 0.078000 | 21.000000 | 0.000000 |
| 25% | 1.000000 | 99.750000 | 64.000000 | 0.000000 | 0.000000 | 27.500000 | 0.243750 | 24.000000 | 0.000000 |
| 50% | 3.000000 | 117.000000 | 72.202592 | 23.000000 | 30.500000 | 32.204005 | 0.372500 | 29.000000 | 0.000000 |
| 75% | 6.000000 | 140.250000 | 80.000000 | 32.000000 | 122.000000 | 36.300000 | 0.612250 | 41.000000 | 1.000000 |
| max | 13.000000 | 199.000000 | 122.000000 | 63.000000 | 495.000000 | 50.000000 | 1.731000 | 81.000000 | 1.000000 |

البته شاید این پیش‌پردازش مناسب نبوده و باید تصمیم دیگری برای پیش‌پردازش داده‌ها اتخاذ می‌شد تا دقت مدل بهبود یابد.

ایده دیگری که می‌توان برای افزایش دقت مدل استفاده کرد، استفاده از تعداد سبدهای متفاوت برای هر کدام از ویژگی‌های دیتاست است. در مدل فعلی تمام ویژگی‌ها به تعداد ثابت NUM_OF_BINS دسته با طول مساوی تقسیم می‌شوند. متمایز کردن تعداد دسته‌ها می‌تواند انعطاف و قدرت مدل را افزایش داده و منجر به افزایش احتمالی دقت شود.

ایده‌های دیگری نیز به ذهن می‌آیند که می‌توان به تغییر روش گسسته‌سازی داده‌های پیوسته این مجموعه‌داده اشاره کرد. ایده استفاده از نقاط برش که در متن تمرین نیز به آن اشاره شده است می‌تواند ایده‌ای مناسب‌تر برای گسسته‌سازی داده‌ها و ایجاد درخت تصمیم باشد که البته در این گزارش آن را آزمایش نکردم. با توجه به بررسی اولیه‌ای که انجام دادم به نظر مدل درخت تصمیم کتابخانه sklearn هم از این روش برای گسسته‌کردن داده‌ها در هنگام تشکیل درخت استفاده می‌کند که می‌تواند مناسب‌تر بودن آن را بیشتر توجیه کند. همچنین یک ایده دیگر می‌تواند استفاده از روش‌های هرس کردن درخت تصمیم که در اسلاید هم گفته شده باشد که مقداری بیش‌برازش را کاهش دهیم. همچنین گذاشتن وقت زیاد بر روی یاد گرفتن هایپر پارامترهای عمق و تعداد سبد و آزمایش تعداد زیادی حالت مختلف برای هر ویژگی می‌تواند تاثیر مثبتی بر دقت داشته باشد.

یک روش بسیار مرسوم برای کاهش بیش‌برازش و واریانس مدل درخت تصمیم و افزایش بایاس و دقت روی داده‌های تست، استفاده از روش‌های یادگیری تقویتی و ترکیب کردن چندین درخت تصمیم در کنار همدیگر به صورت خطی است. این روش‌ها معمولا باعث می‌شوند که هر درخت بر

روی داده‌های که در درخت قبلی اشتباه پیش‌بینی شده‌اند بیشتر تمرکز کند و به آن‌ها اصطلاحا وزن بیشتری اختصاص دهد. تجربه‌ای که به شخصه در تمرین درس یادگیری ماشین با پیاده‌سازی روش AdaBoost با مدل پایه‌ای درخت تصمیم رشد چشمگیر آن در دقت (بالای ۹۲ درصد) و سایر score های مدل را نسبت به یک درخت تصمیم ساده مشاهده کردم.

چالش اصلی من در این تمرین، دقت کمی پایین‌تر از مدل نوشته‌شده من نسبت به مدل درخت تصمیم کتابخانه sklearn و درخت چند نفر دیگر از دوستان بود که به طور میانگین حدود ۵ ۶ درصد فاصله داشتم. در کل پیاده‌سازی خود درخت تصمیم و سایر موارد چالش خاصی نداشت و بخش سخت یافتن راهی برای افزایش دقت بود. در تصویر پایین کد درخت تصمیم کتابخانه‌ای پایتون و ۱۰ بار اجرای آن را مشاهده می‌کنید:

```
diabetes_df = pd.read_csv('diabetes.csv')
y_label = 'Outcome'
test_size = 0.2

for i in range(10):
    X_train, X_test, y_train, y_test = train_test_split(diabetes_df.drop(columns=[y_label]), diabetes_df[y_label],
                                                        test_size=test_size, shuffle=True)
    clf = DecisionTreeClassifier(criterion='entropy')

    clf = clf.fit(X_train, y_train)

    y_pred = clf.predict(X_test)
    time.sleep(0.5)
    print(f"Accuracy {i} :", metrics.accuracy_score(y_test, y_pred))
```

```
Accuracy 0 : 0.7272727272727273
Accuracy 1 : 0.6883116883116883
Accuracy 2 : 0.6883116883116883
Accuracy 3 : 0.6623376623376623
Accuracy 4 : 0.6493506493506493
Accuracy 5 : 0.7207792207792207
Accuracy 6 : 0.7857142857142857
Accuracy 7 : 0.7402597402597403
Accuracy 8 : 0.7727272727272727
Accuracy 9 : 0.6948051948051948
```

در اجراهای مختلف گرفته‌شده از این کتابخانه دقت‌هایی غالبا از بازی ۶۵ تا ۷۵ درصد به دست می‌آمد و برخی مواقع حتی تا ۸۰ درصد رشد و یا تا ۶۰ درصد افت می‌کرد. به طور میانگین دقت ۷۴ را از

این مدل می‌توانستیم به دست آوریم که حدودا ۱۰ درصد بیشتر از دقت میانگین مدل من بود (البته با یک ستینگ خاص و نه برای تمام شرایط). تلاش‌های زیادی هم برای افزایش دقت مدل انجام دادم که برخی در این گزارش ذکر شدند ولی در کل نتوانستم پیشرفت زیادی بکنم. بیشترین ظن من در این اختلاف دقت‌ها، به تفاوت نحوه گسسته‌سازی داده‌ها در مدل من با این مدل کتابخانه‌ای برمی‌گردد.
همچنین در نمایش گرافیکی درخت تصمیم اندکی مشکل وجود داشت که با کمک گرفتن از چند نفر از دوستان این مشکل برطرف شد. برای نمایش متنی هم از روشی که در نمایش درخت پارس گرامر در درس کامپایلر استفاده کردم کمک گرفتم.
تعدادی تصویر استفاده‌شده در گزارش به همراه فایل‌های pdf خروجی گرافیکی درخت‌های ذکر شده در گزارش به پیوست آمده‌اند. همچنین یک jupyter notebook هم در کنار فایل‌ها وجود دارد که در اصل نقش چرک‌نویس آزمایش‌ها و بازی با داده‌ها و بخش‌های مختلف کد اصلی را دارد که شاید بررسی آن خالی از لطف نباشد. در هنگام اجرای کد، تعدادی فایل اضافه‌تر هم در کنار خروجی‌ها تولید می‌شود که به ناچار در فرایند رسم درخت مجبور به استفاده از آن‌ها بوده‌ایم که نیازی به بررسی این فایل‌ها نیست و صرفا برای مقاصد پیاده‌سازی تولید می‌شوند.

ممنون از حوصله و توجه شما