قسمت اول: آماده‌سازی داده‌ها

برای این کار، فایل 1.DataPreparation.ipynb تهیه شده است. لذا به بررسی آن می‌پردازیم:

```
In [1]: import pandas as pd
```

Loading datasets

```
In [2]: dataset1 = pd.read_csv('Dataset/Dataset1.csv')
        dataset1_unknown = pd.read_csv('Dataset/Dataset1_Unknown.csv')

        dataset2 = pd.read_csv('Dataset/Dataset2.csv')
        dataset2_unknown = pd.read_csv('Dataset/Dataset2_Unknown.csv')

        dataset3 = pd.read_csv('Dataset/Dataset3.csv')
        dataset3_unknown = pd.read_csv('Dataset/Dataset3_Unknown.csv')
```

در ابتدا به کمک کتابخانه pandas، دیتاست های ۱ و ۲و ۳ که به صورت csv می‌باشند را به کمک تابع read_csv بارگذاری و به صورت دیتافریم درمی‌آوریم.

Dataset 1

Concating dataset1 and dataset1_unknown

```
In [3]: dataset = pd.concat([dataset1, dataset1_unknown])
```

```
In [4]: dataset.head()
```

Out[4]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
3	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K
4	37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	0	0	40	United-States	<=50K

```
In [5]: print(f"Maximum capital gain {dataset['capital-gain'].max()}")
        print(f"Maximum capital loss {dataset['capital-loss'].max()}")
        print(f"Maximum hours per week {dataset['hours-per-week'].max()}")
        print(f"Maximum fnlwgt {dataset['fnlwgt'].max()}")
```

```
Maximum capital gain 99999
Maximum capital loss 4356
Maximum hours per week 99
Maximum fnlwgt 1484705
```

با توجه به این که هدف انکود کردن داده ها می‌باشد، بهتر است dataset1 و dataset1_unknown را تحت عنوان یک دیتاست نگاه کنیم. زیراکه برفرض ممکن است در ستون workclass مقداری در dataset1 باشد که در dataset1_unknown نباشد و بالعکس و اگر جدا انکود کنیم تعداد ستون متفاوتی حاصل خواهد شد

که برای مدل یادگیری ماشین دشوار می‌شود (زیرا میخواهیم در قسمت دوم با dataset1 مدل را آموزش دهیم و در دستور کار آمده است که مدل آموزش دیده شده بایستی dataset1_unknown هم پیشبینی کند و اگر ستون ها متفاوت باشد این کار شدنی نیست).
بنابراین به کمک تابع concat، dataset1_unkown به ادامه dataset1 اضافه شده و دیتافریم جدید را در متغیر dataset ذخیره میکنیم.

قبل از انکود کردن، تعدادی ستون هستند که بایستی Categorize شوند. لذا بایستی مقدار ماکسیمم آن ها مشخص باشد که بتوانیم یک تقسیم بندی برابری داشته باشیم. در جلوتر ماکسیمم هر یک پرینت شده اند. با توجه به این مقادیر ماکسیمم تقسیم بندی که خواهیم داشت برابر است با:

Converting numerical data to categorical data

Age

- age <= 19: **Teenager**
- 20 <= age < 50: **Adult**
- 50 <= age < 65: **Middle Age**
- 65 <= age: **Elderly**

Capital Gain

- capital gain < 33,333: **Low**
- 33,333 <= capital gain < 66,666: **Medium**
- 66,666 <= capital gain: **High**

Capital Loss

- capital loss < 1,452: **Low**
- 1,452 <= capital loss < 2,904: **Medium**
- 2,904 <= capital loss: **High**

Final Weight

- fnlwgt < 494,901: **Low**
- 494,901 <= fnlwgt < 989,802: **Medium**
- 989,802 <= fnlwgt: **High**

Hours Per Week

- hours per week < 33: **Short**
- 33 <= hours per week < 66: **Medium**
- 66 <= hours per week: **Large**

لذا به صورت فوق این کار را انجام میدهیم:

In [6]:
```
age = dataset['age'].copy()
age.loc[dataset['age'] <= 19] = "Teenager"
age.loc[(20 <= dataset['age']) & (dataset['age'] < 50)] = "Adult"
age.loc[(50 <= dataset['age']) & (dataset['age'] < 65)] = "Middle Age"
age.loc[dataset['age'] >= 65] = "Elderly"
dataset['age'] = age

capital_gain = dataset['capital-gain'].copy()
capital_gain.loc[dataset['capital-gain'] < 33333] = "Low"
capital_gain.loc[(33333 <= dataset['capital-gain']) & (dataset['capital-gain'] < 66666)] = "Medium"
capital_gain.loc[dataset['capital-gain'] >= 66666] = "High"
dataset['capital-gain'] = capital_gain

capital_loss = dataset['capital-loss'].copy()
capital_loss.loc[dataset['capital-loss'] < 1452] = "Low"
capital_loss.loc[(1452 <= dataset['capital-loss']) & (dataset['capital-loss'] < 2904)] = "Medium"
capital_loss.loc[dataset['capital-loss'] >= 2904] = "High"
dataset['capital-loss'] = capital_loss

final_weight = dataset['fnlwgt'].copy()
final_weight.loc[dataset['fnlwgt'] < 494901] = "Low"
final_weight.loc[(494901 <= dataset['fnlwgt']) & (dataset['fnlwgt'] < 989802)] = "Medium"
final_weight.loc[dataset['fnlwgt'] >= 989802] = "High"
dataset['fnlwgt'] = final_weight

hours_per_week = dataset['hours-per-week'].copy()
hours_per_week.loc[dataset['hours-per-week'] < 33] = "Short"
hours_per_week.loc[(33 <= dataset['hours-per-week']) & (dataset['hours-per-week'] < 66)] = "Medium"
hours_per_week.loc[dataset['hours-per-week'] >= 66] = "Large"
dataset['hours-per-week'] = hours_per_week
```

در ادامه نیاز است که داده ها به کمک One-hot encoder انکود شوند. برای این کار از تابع get_dummies کمک میگیریم و تمامی ستون ها به جز ستون Income (که برچسب مورد نظر ما است) را انکود میکنیم:

Encoding categorical data using one-hot encoder

In [7]:
```
dataset.head()
```

Out[7]:

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	Adult	State-gov	Low	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	Low	Low	Medium	United-States	<=50K
1	Middle Age	Self-emp-not-inc	Low	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	Low	Low	Short	United-States	<=50K
2	Middle Age	Private	Low	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	Low	Low	Medium	United-States	<=50K
3	Adult	Private	Low	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	Low	Low	Medium	Cuba	<=50K
4	Adult	Private	Low	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	Low	Low	Medium	United-States	<=50K

In [8]:
```
categorial_features = [column for column in dataset if column != 'income']

dataset = pd.get_dummies(dataset, columns=categorial_features)

dataset = dataset[[column for column in dataset if column != 'income'] + ['income']]  # move income to the last column
dataset['income'] = pd.factorize(dataset['income'])[0]  # encode <=50k as 0 and > 50k as 1
```

In [9]:
```
dataset1 = dataset.head(dataset1.shape[0]).copy()
dataset1_unknown = dataset.tail(dataset1_unknown.shape[0]).copy()
dataset1_unknown.drop('income', axis=1, inplace=True)
```

لازم به ذکر است که در خط آخر سلول هشتم، ستون Income نیز انکود شده است. اما چون برچسب است، دیگر نبایستی به چند ستون انکود شود و به کمک factorize اگر از 50k کوچکتر باشد کلاس صفر و اگر بیشتر باشد کلاس یک محسوب می‌شود.

در سلول نهم نیز پس از انکود کردن دوباره dataset1 و dataset1_unknown از هم جدا می‌شوند و با توجه به این که dataset1_uknown برچسب ندارد (همگی مقدار NaN دارند)، ستون برچسب از آن حذف می‌شود.

```
In [10]: dataset1_unknown.head()
```

Out[10]:

	age_Adult	age_Elderly	age_Middle Age	age_Teenager	workclass_?	workclass_Federal-gov	workclass_Local-gov	workclass_Never-worked	workclass_Private	workclass_Self-emp-inc
0	1	0	0	0	0	0	0	0	1	0
1	0	0	0	1	1	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0	1
4	1	0	0	0	0	0	0	0	1	0

5 rows × 134 columns

```
In [11]: dataset1.to_csv('DatasetModified/dataset1.csv', index=False)
         dataset1_unknown.to_csv('DatasetModified/dataset1_unknown.csv', index=False)
```

که نمونه ای از ستون ها پس از Encode شدن را مشاهده میکنیم تا از کار اطمینان کسب کنیم و در آخر به کمک to_csv، دیتاست ها را ذخیره میکنیم.

Dataset2

```
In [12]: dataset = pd.concat([dataset2, dataset2_unknown])
```

```
In [13]: dataset.head()
```

Out[13]:

	poisonous	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil-type	veil-color	ring-number	ring-type	spore-print-color	populat
0	p	x	s	n	t	p	f	c	n	k	...	s	w	w	p	w	o	p	k	
1	e	b	s	w	t	l	f	c	b	n	...	s	w	w	p	w	o	p	n	
2	p	x	y	w	t	p	f	c	n	n	...	s	w	w	p	w	o	p	k	
3	e	x	s	g	f	n	f	w	b	k	...	s	w	w	p	w	o	e	n	
4	e	x	y	y	t	a	f	c	b	n	...	s	w	w	p	w	o	p	k	

5 rows × 23 columns

```
In [14]: categorial_features = [column for column in dataset if column != 'poisonous']

         dataset = pd.get_dummies(dataset, columns=categorial_features)

         dataset = dataset[[column for column in dataset
                            if column != 'poisonous'] + ['poisonous']]  # move poisonous to the last column
         dataset['poisonous'] = pd.factorize(dataset['poisonous'])[0]  # convert poisonous values to classes
```

```
In [15]: dataset2 = dataset.head(dataset2.shape[0]).copy()
         dataset2_unknown = dataset.tail(dataset2_unknown.shape[0]).copy()
         dataset2_unknown.drop('poisonous', axis=1, inplace=True)
```

```
In [16]: dataset2.to_csv('DatasetModified/dataset2.csv', index=False)
         dataset2_unknown.to_csv('DatasetModified/dataset2_unknown.csv', index=False)
```

برای دیتاست دوم روند بسیار ساده تر است. زیرا تمامی ستون ها به صورت گسسته می‌باشند و تنها کافیست مشابه قبل به کمک get_dummies عمل one-hot encoding صورت پذیرد و در آخر دو دیتاست را ذخیره کنیم.

با توجه به مشابه بودن دیتاست سوم و اول از نظر نوع داده، از توضیح آن صرف نظر میکنیم.

قسمت دوم: کلاس‌بندی داده‌ها

در ابتدا برای تقسیم‌بندی داده به دو قسمت مجزا، در فایل model_selection.py یک تابع تحت عنوان train_test_split تهیه شده است که به توضیح آن می‌پردازیم:

```python
def train_test_split(data, train_ratio, seed=None):
    if isinstance(data, pd.DataFrame):
        data = data.to_numpy()
    if train_ratio > 1 or train_ratio < 0:
        print("ERROR: TRAIN RATIO SHOULD BE BETWEEN 0 AND 1")
        exit()

    data_length = data.shape[0]
    train_size = int(train_ratio * data_length)

    if seed:
        np.random.seed(seed)
    indices = np.random.permutation(data_length)
    train_indices, test_indices = indices[:train_size], indices[train_size:]
    return data[train_indices, :], data[test_indices, :]
```

در تابع فوق داده و train_ratio که یک عددی بین ۰ و ۱ می‌باشد دریافت می‌شود. اگر داده از نوع دیتافریم باشد به نامپای تبدیل میشود و اگر train_ratio بین ۰ و ۱ نباشد خطا اعلام میگردد.

در تمرین با توجه به این که ۸۰ درصد داده بایستی آموزشی باشند، train_ratio همواره برابر با ۰.۸ می‌باشد. اما برای این که مشخص شود ۸۰ درصد داده چه اندازه می‌باشد، ابتدا تعداد سطر ها در data_length قرار گرفته می‌شود و سپس train_size محاسبه می‌شود.

سپس به اندازه train_size تا داده میخواهیم از داده های اصلی تحت داده آموزشی به صورت رندم انتخاب کنیم. بنابراین یک ورودی seed هم گذاشته شده است که اگر نخواهیم در هر اجرا train_data متفاوت باشد، یک مقدار seed نیز اختیار کنیم.

در ادامه همانطور که در stackoverflow توضیح داده شده است، ابتدا به کمک permutation ایندکس هارا به صورت رندم به تعداد سایز سطر ها انتخاب میکنیم، و از بین این ایندکس های رندم train_size تای اولی را برای داده آموزشی و باقی را برای تست در نظر میگیریم و در آخر داده های مربوطه را باز میگردانیم.

برای سه مدل خواسته شده نیز در پوشه models، سه فایل وجود دارد که هرکدام یک مدل میباشد که اجرای آن ها 2.DataClassification.py صورت میگیرد که به توضیح آن میپردازیم:

۱) در ابتدای امر به Visualize کردن درخت با دو criterion متفاوت میپردازیم:

Decision Tree

```
In [2]: dataset1 = pd.read_csv('DatasetModified/dataset1.csv')
        dataset1_unknown = pd.read_csv('DatasetModified/dataset1_unknown.csv')

In [3]: train_set, test_set = train_test_split(dataset1, train_ratio=0.8)
```

در دو سلول فوق ابتدا دیتاست ها خوانده شده و سپس به کمک train_test_split که توضیح داده شد، داده آموزشی و تست تشکیل میشوند. سپس برای Visualize کردن داریم:

Visualize Tree with different criterion

```
In [4]: decision_tree = DecisionTree(train_set, test_set, max_depth=3, min_size=10, criterion='GINI')
        decision_tree.visualize_tree()
```

```
|--- feature_50 == 0
|  |--- feature_26 == 0
|  |  |--- feature_59 == 0
|  |  |  |--- class 0
|  |  |--- feature_59 == 1
|  |  |  |--- class 0
|  |--- feature_26 == 1
|  |  |--- feature_59 == 0
|  |  |  |--- class 1
|  |  |--- feature_59 == 1
|  |  |  |--- class 1
|--- feature_50 == 1
|  |--- feature_25 == 0
|  |  |--- feature_65 == 0
|  |  |  |--- class 0
|  |  |--- feature_65 == 1
|  |  |  |--- class 1
|  |--- feature_25 == 1
|  |  |--- feature_91 == 0
|  |  |  |--- class 1
|  |  |--- feature_91 == 1
|  |  |  |--- class 0
```

ایده رسم این از این وبسایت آمده است. برای رسم در ابتدا بایستی خود درخت ساخته شود که به توضیح آن میپردازیم:

```
class DecisionTree:
    def __init__(self, train_set, test_set, max_depth=3, min_size=10,
criterion='GINI'):
        """
        :param train_set: train set of our dataset which is a numpy array
        :param test_set: test set of our dataset which is a numpy array
        :param max_depth: maximum depth that we want to expand. Default is 3
        :param min_size: minimum size of each node. Default is 10
        :param criterion: Either GINI or ENTROPY. Default is GINI
        """
        self.max_depth = max_depth
        self.min_size = min_size
        self.criterion = criterion

        self.train_set = train_set
        self.test_set = test_set

        self.tree = self.build_tree(self.train_set)
```

در کلاس DecisionTree، همانطور که مشاهده میکنیم پارامتر هایی دریافت میشود که در کامنت توضیح داده شده اند. پس از ذخیره این پارامتر ها، برای ساخت درخت تابعی تحت عنوان build_tree فراخوانی میشود که ایده ایجاد آن از این وبسایت می‌باشد (در این وبسایت درخت تصمیم بدون هیچ کتابخانه ای ساخته

میشود ولی در این تمرین به کمک نامپای انجام گرفته و دارای سرعت بیشتر و از لحاظ سینتکسی خلاصه تر است).

```
def build_tree(self, train_set):
    root = self.get_split(train_set)
    self.split(root, 1)
    return root
```

متد build_tree ابتدا به کمک متد get_split ریشه درخت را ساخته و سپس به کمک split به صورت بازگشتی فرزندان را می‌سازد. لذا به بررسی این دو میپردازیم:

```
def get_split(self, data):
    best_index, best_score, best_groups = float('inf'), float('inf'), None
    for feature_index in range(data.shape[1] - 1):  # except last column
        groups = self.binary_split(feature_index, data)
        cost_score = self.cost(groups)
        if cost_score < best_score:
            best_index, best_score, best_groups = feature_index, cost_score,
groups
    return {
        'index': best_index,
        'groups': best_groups
    }
```

در متد get_split هدف انتخاب بهترین ستون از بین ستون های موجود و انتخاب فرزند چپ و راست برای آن ستون می‌باشد.

بنابراین در ابتدا سه متغیر تعریف شده که best_index نشان میدهد بهترین ستون چه ایندکسی را دارد، best_score امتیاز آن ستون با معیار criterion (GINI یا ENTROPY) و best_groups نیز فرزندان چپ و راست آن ستون می باشد.

پس در یک حلقه به ازای تمامی ستون ها به جز ستون آخر(که برچسب می‌باشد)، عمل binary_split صورت میگیرد که فرزندان چپ و راست برای آن ستون را تشکیل می‌دهد (توضیح در ادامه). سپس به کمک متد cost نیز بسته به Criterion هزینه آن فرزندان محاسبه شده و اگر این هزینه از بهترین هزینه فعلی بهتر باشد، آنگاه پس بهتر است ستون مورد نظر انتخاب گردد و در آخر ایندکس آن ستون و فرزندان آن برگردانده می‌شوند.

```
@staticmethod
def binary_split(feature_index, data):
    left = data[data[:, feature_index] == 0]
    right = data[data[:, feature_index] == 1]
    return np.array([left, right], dtype=object)
```

در متد best_split نیز از داده های ورودی در آن ستون مربوطه، درایه هایی که مقدار صفر دارند در فرزند چپ و درایه هایی که مقدار یک دارند در فرزند راست قرار میگیرند.

```
def cost(self, groups):
    if self.criterion == "GINI":
        return self.gini_index(groups)
```

```
    elif self.criterion == "ENTROPY":
        return self.entropy(groups)
    else:
        print("ERROR: INVALID CRITERION")
        exit()
```

در متد cost نیز به criterion توجه میشود و اگر GINI باشد gini_index و اگر ENTROPY باشد entropy فرزندان محاسبه می‌شود. لذا به بررسی این دو میپردازیم:

```
@staticmethod
def gini_index(groups):
    number_of_instances = np.sum([group.shape[0] for group in groups])

    gini = 0
    for group in groups:
        group_size = group.shape[0]
        if group_size == 0:
            continue

        group_classes = group[:, -1]
        _, counts = np.unique(group_classes, return_counts=True)
        p = counts / group_size
        sigma = np.sum(p * p)
        group_weight = group_size / number_of_instances
        gini += (1 - sigma) * group_weight

    return gini
```

میدانیم GINI برابر است با:

$$GINI(t) = 1 - \sum_{j} [p(j|t)]^2$$

که در آن p(j|t) نیز برابر است با تعداد کلاس j در آن Node تقسیم بر کل درایه های آن Node. و بعد $GINI_{split}$ نیز برابر است با:

$$GINI_{split} = \sum_{i=1}^{k} \frac{n_i}{n} GINI(i)$$

پس ابتدا gini = 0 که گذاشته شده همان $GINI_{split}$ می باشد که قرار است در هر حلقه زیاد شود و گویی حلقه اصلی همان سیگما در رابطه $GINI_{split}$ می‌باشد.

در داخل حلقه ابتدا تعداد کل درایه های داخل Node در group_size ریخته میشود که در p(j|t) حکم مخرج و در $GINI_{split}$ نیز همان n_i می‌باشد. سپس درایه های ستون آخر را در group_classes ذخیره میکنیم که برچسب ها میباشند و به کمک np.unqiue تعداد هر برچسب را در داخل counts ذخیره میکنیم. در رابطه p(j|t) نیز بایستی تعداد هر برچسب تقسیم بر group_size میشد لذا این کار را انجام

میدهیم و یک آرایه p تشکیل میشود که هر درایه آن یک p(j|t) می‌باشد.
سپس سیگمای GINI(t) را محاسبه کرده و در داخل sigma ذخیره میکنیم. در گام بعد n_i/n را انجام داده و در group_weight ذخیره میکنیم و $GINI_{split}$ را که همان gini است با مقدار بدست آمده جمع میکنیم.

برای آنتروپی نیز داریم:

```
@staticmethod
def entropy(groups):
    number_of_instances = np.sum([group.shape[0] for group in groups])

    entropy = 0
    for group in groups:
        group_size = group.shape[0]
        if group_size == 0:
            continue

        group_classes = group[:, -1]
        _, counts = np.unique(group_classes, return_counts=True)
        p = counts / group_size
        sigma = np.sum(p * np.log2(p))
        group_weight = group_size / number_of_instances
        entropy += (- sigma) * group_weight

    return entropy
```

محاسبه این نیز همانند GINI می باشد و تنها تفاوت آن رابطه Entropy(t) می باشد که به صورت زیر است:

$$Entropy(t) = -\sum_{j} p(j|t) log p(j|t)$$

بنابراین متد get_split به کمک توضیحات فوق بهترین ستون را انتخاب کرده، فرزند چپ و راست آن را درست کرده و Node ایجاد شده به همراه فرزندان آن را در root قرار میدهد:

```
def build_tree(self, train_set):
    root = self.get_split(train_set)
    self.split(root, 1)
    return root
```

در گام بعد از build_tree تابع split فراخوانی میشود که به صورت بازگشتی درخت را تشکیل دهد:

```
def split(self, node, depth):
    left, right = node['groups']
    del (node['groups'])

    if left.shape[0] == 0 or right.shape[0] == 0:
        node['left'] = node['right'] = self.to_terminal(np.vstack((left,
right)))
        return
    if depth >= self.max_depth:
        node['left'], node['right'] = self.to_terminal(left),
self.to_terminal(right)
```

```
        return

    if left.shape[0] <= self.min_size:
        node['left'] = self.to_terminal(left)
    else:
        node['left'] = self.get_split(left)
        self.split(node['left'], depth + 1)

    if right.shape[0] <= self.min_size:
        node['right'] = self.to_terminal(right)
    else:
        node['right'] = self.get_split(right)
        self.split(node['right'], depth + 1)
```

در این تابع node فعلی و عمق آن دریافت میشود. علت دریافت عمق نیز این است که اگر از ماکسیمم عمق تعریف شده بیشتر شود عملیات به پایان برسد.

در گام اول فرزند چپ و راست از group آن node استخراج میشود و از دیکشنری مورد نظر حذف میشود. چرا که یا میخواهیم این هارا تبدیل به internal node کنیم و یا terminal node. پس ابتدا بررسی میکنیم که آیا فرزند چپ و یا فرزند راست تهی هستند یا خیر. اگر تهی باشند با توجه به این که GINI از پیش چک شده است، بدین معنی است که impurity در آن یک Node باقی مانده بسیار کم است که از بین بقیه ستون ها اکنون انتخاب شده، پس به کمک متد to_terminal، در فرزند چپ و راست Node کنونی تنها class قرار میگیرد.

اگر هم عمق از ماکسیمم عمق مشخص شده بیشتر باشد، آنگاه باز بایستی از ادامه دادن کار بایستیم و کلاس هارا مشخص کنیم.

اگر هر Node تعداد درایه ها از min_sizeکوچکتر باشند نیز بایستی Terminal باشند، اما در غیر این صورت به کمک get_split فرزند چپ و راست آن ها مشخص شده و به صورت بازگشتی split فراخوانی میشود.

```
@staticmethod
def to_terminal(group):
    outcomes = group[:, -1]
    unique_outcomes, counts = np.unique(outcomes, return_counts=True)
    most_frequent_index = np.argmax(counts)
    return unique_outcomes[most_frequent_index]
```

متند to_terminal نیز به این صورت است که از بین برچسب ها، برچسبی که بیشترین تکرار را داشته است انتخاب شده و برگردانده میشود.

رسم درخت با معیار آنتروپی نیز به شکل زیر است:

```
In [5]: decision_tree2 = DecisionTree(train_set, test_set, max_depth=3, min_size=10, criterion='ENTROPY')
        decision_tree2.visualize_tree()
```

```
|--- feature_50 == 0
|  |--- feature_73 == 0
|  |  |--- feature_59 == 0
|  |  |  |--- class 0
|  |  |--- feature_59 == 1
|  |  |  |--- class 0
|  |--- feature_73 == 1
|  |  |--- feature_91 == 0
|  |  |  |--- class 0
|  |  |--- feature_91 == 1
|  |  |  |--- class 0
|--- feature_50 == 1
|  |--- feature_25 == 0
|  |  |--- feature_65 == 0
|  |  |  |--- class 0
|  |  |--- feature_65 == 1
|  |  |  |--- class 1
|  |--- feature_25 == 1
|  |  |--- feature_91 == 0
|  |  |  |--- class 1
|  |  |--- feature_91 == 1
|  |  |  |--- class 0
```

برای محاسبه Accuracy نیز به نحو زیر عمل میکنیم:

Calculate Accuracy

```
In [6]: decision_tree.get_accuracy()
Out[6]: 80.69097888675624

In [7]: decision_tree2.get_accuracy()
Out[7]: 80.7869481765835
```

که همانطور که مشاهده میشود، معیار ENTROPY در این دیتاست عملکرد بهتری را نشان داده است.

```
def get_accuracy(self):
    predictions = self.predict()
    labels = self.test_set[:, -1]

    accuracy = ((predictions == labels).sum() / labels.shape[0]) * 100
    return accuracy
```

برای محاسبه accuracy نیز ابتدا به کمک متد predict یک پیشبینی صورت گرفته، سپس برچسب ها از ستون آخر test_set برداشته شده و accuracy حساب شده است. برای محاسبه accuracy همانطور که مشاهده میشود دو بردار مقایسه شده است که در صورت تساوی دو درایه True و در غیر این صورت False میشود. سپس ()sum فراخوانی شده است چرا که True به ۱ و False به صفر Cast میشود. پس تعداد درست محاسبه شده و تقسیم بر کل تعداد شده و خروجی در ۱۰۰ ضرب شده است.

```
def predict(self):
    predictions = []
    for row in self.test_set:
        predictions.append(self.predict_row(self.tree, row))
    return np.array(predictions)
```

در متد predict نیز به ازای هر ردیف از test set متد predict_row فراخوانی شده و نتیجه در لیستی اضافه شده است.

```
def predict_row(self, node, row):
    if row[node['index']] == 0:
        if isinstance(node['left'], dict):
            return self.predict_row(node['left'], row)
        else:
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return self.predict_row(node['right'], row)
        else:
            return node['right']
```

در متد predict_row نیز یک node دریافت میشود که ابتدا root می باشد و row نیز همان ردیف test set برای پیشبینی است. پس ابتدا در ریشه درخت میشاهده میشود که index انتخاب شده است و در ردیف test set آن ویژگی را نگاه میکنیم که آیا صفر است یا یک. اگر صفر باشد به فرزند چپ و اگر یک باشد به فرزند راست میرویم.

در هر فرزند که میرویم چک میکنیم که آیا فرزند از نوع dictionary میباشد یا خیر. اگر باشد پس فرزند internal node میباشد و بایستی ادامه بدهیم و در غیر این صورت به terminal node رسیده ایم و مقدار class موجود در آن را باز میگردانیم.

predict dataset_unknown

```
In [8]: decision_tree.change_test_set(dataset1_unknown)
        y_predicted = decision_tree.predict()
        _, counts = np.unique(y_predicted, return_counts=True)
        print(f"class 0: {counts[0]}\nclass 1: {counts[1]}")

        class 0: 5691
        class 1: 821
```

```
In [9]: decision_tree2.change_test_set(dataset1_unknown)
        y_predicted = decision_tree2.predict()
        _, counts = np.unique(y_predicted, return_counts=True)
        print(f"class 0: {counts[0]}\nclass 1: {counts[1]}")

        class 0: 5724
        class 1: 788
```

در آخر نیز برای مشاهده عملکرد مدل روی dataset1_unknown، به کمک متد change_test_set داده تست را عوض میکنیم و با کمک متد predict بردار خروجی برای داده تست جدید را تولید میکنیم.

```
def change_test_set(self, new_test_set):
    if isinstance(new_test_set, pd.DataFrame):
        new_test_set = new_test_set.to_numpy()
    self.test_set = new_test_set
```

در متد change_test_set نیز با توجه به این که dataset1_unkownاز فیلتر
train_test_splitنگذشته است، از نوع دیتافریم می‌باشد و لذا در ابتدا به صورت نامپای در می آوریم و سپس در test_set قرار می‌دهیم.

۲) برای توضیح ابتدا متد های کلاس KNearestNeighbor را بررسی میکنیم:

```
@staticmethod
def euclidean_distance(row1, row2):
    return np.sqrt(np.sum(np.power(row1 - row2, 2)))

def get_k_neighbors(self, target_row):
    distances = []
    for train_row in self.train_data:
        if train_row.size == target_row.size:
            distance = self.euclidean_distance(train_row[:-1], target_row[:-
1])
        else:
            distance = self.euclidean_distance(train_row[:-1], target_row)
        distances.append(distance)

    sorted_indices = sorted(range(len(self.train_data)), key=lambda i:
distances[i])
```

```
    neighbors = self.train_data[sorted_indices[:self.k]]
    return neighbors
```

در متد eculidean_distance فاصله اقلیدسی دو سطر محاسبه میشود.

در متد get_k_neighbors، k تا همسایه با نزدیک ترین فاصله اقلیدسی دریافت می‌شوند. به صورتی که در داده آموزشی به ازای هر ردیف، فاصله ردیف با ردیفی که میخواهیم کلاس آن را پیشبینی کنیم اندازه گیری میشود. برای محاسبه یک شرط گذاشته شده است که اگر سایز این ۲ ردیف یکسان باشند، یعنی هردو برچسب دارند و آنگاه از هردو برچسب برداشته شده و مقایسه میشود. ولی اگر سایزشان برابر نباشد، انگاه یعنی dataset2_unkown داریم که برچسب ندارد و برای target_row دیگری ستونی حذف نمیشود. پس در حلقه فاصله سطر هدف با تمامی سطر های آموزشی محاسبه میشود.

سپس این فاصله ها sort میشود و ایندکس ردیف های متناظر در sorted_indices قرار میگیرد. پس تنها کافیست از بین این sorted_indices، k تای نزدیک انتخاب و تحت همسایه برگردانده شوند.

```
def predict_class(self, target_row):
    neighbors = self.get_k_neighbors(target_row)
    output_values = neighbors[:, -1]

    unique_output_values, counts = np.unique(output_values,
return_counts=True)
    most_frequent_index = np.argmax(counts)
    most_frequent_output_value = unique_output_values[most_frequent_index]
    return most_frequent_output_value

def predict(self):
    predictions = []
    for test_row in self.test_data:
        prediction = self.predict_class(test_row)
        predictions.append(prediction)
    predictions = np.array(predictions)
    return predictions
```

در متد predict_class نیز پس ابتدا k تا همسایه نزدیک انتخاب میشود، سپس تعداد برچسب آن ها محاسبه و برچسبی که بیشترین تکرار دارد تحت عنوان کلاس داده تست انتخاب میشود.

متد predict نیز به ازای تمامی سطر های داخل داده های تست این کار را انجام میدهد ویک یک بردار از نتایج را برمیگرداند.

K-Nearest Neighbors

```
In [10]: dataset2 = pd.read_csv('DatasetModified/dataset2.csv')
         dataset2_unknown = pd.read_csv('DatasetModified/dataset2_unknown.csv')

In [11]: train_set, test_set = train_test_split(dataset2, train_ratio=0.8)

In [12]: knn = KNearestNeighbor(train_set, test_set, k=10)
         knn.get_accuracy()

Out[12]: 100.0

In [13]: knn.change_test_set(dataset2_unknown)
         y_predicted = knn.predict()
         _, counts = np.unique(y_predicted, return_counts=True)
         print(f"class 0: {counts[0]}\nclass 1: {counts[1]}")

         class 0: 800
         class 1: 825
```

پس همانطور که در تصویر فوق مشاهده میکنیم، به ازای k=10، مدل دقت ۱۰۰ درصد را میدهد.

۳) توضیح NaiveBayes:

```
class NaiveBayes:
    def __init__(self, train_data, test_data):
        self.train_data, self.test_data = train_data, test_data
        self.classes = np.unique(self.train_data[:, -1])  # unique is always
sorted (Important for argmax)
```

در Constructor این کلاس علاوه بر ذخیره داده آموزشی و تست، کلاس های مورد نظر نیز در self.classes ذخیره میشود.

```
def predict(self):
    predictions = []
    for test_row in self.test_data:
        prediction = self.predict_class(test_row)
        predictions.append(prediction)
    return predictions

def predict_class(self, test_row):
    """
    P(y|x1, x2, ..., xn) = P(x1|y)P(x2|y)...P(xn|y)P(y)
    We don't consider denominator because it's the same for different classes
and we want to compare them.
    """
    P_y_x = []
    for y in self.classes:
        class_data = self.train_data[np.where(self.train_data[:, -1] == y)]

        P_y = class_data.size / self.test_data.size
```

```
        P_x_y = 1
        for col in range(self.test_data.shape[1] - 1):  # except the last
column
            P_x_y *= np.where(class_data[:, col] == test_row[col])[0].size /
class_data.size

        P_y_x.append(P_y * P_x_y)

    return np.argmax(P_y_x)
```

متد predict نیز مشابه قبل به ازای هر سطر predict_class را فراخوانی میکند.

در متد predict_class همانطور که در کامنت توضیح داده شده تنها صورت کسر اهمیت دارد. بنابراین ابتدا یک لیست P_y_x میسازیم که همان P(y|x1, x2, ..., xn) به ازای y های متفاوت را در خود ذخیره میکند. سپس در حلقه به ازای هر کلاس ابتدا از داده آموزشی آن ردیف هایی که برچسبشان y می باشد جدا شده و در class_data ذخیره میشوند. P(y) طبق رابطه برابر است با تعداد ردیف هایی که برچسبشان y است تقسیم بر کل.

سپس میخواهیم P(x1|y)P(x2|y)...P(xn|y) را حساب کنیم. پس در یک حلقه به ازای ستون های متفاوت این کار را انجام میدهیم. به طوری که از بین ردیف هایی که برچسبشان y است، ردیف هایی را انتخاب میکنیم که مقدار درایه ستون col اشان برابر با مقدار ستون col ردیف داده تست باشد و تعداشان را تقسیم بر تعداد داده های class_data میکنیم.
پس در انتها یک لیست P_y_x داریم که صورت کسر رابطه نایو بیز را در خود ذخیره میکند. با توجه به این که در self.classes کلاس ها به صورت صعودی مرتب شده اند، اگر argmax بگیریم خود کلاس دریافت شده و لذا argmax را برمیگردانیم.

Naive Bayes

In [14]:
```
dataset3 = pd.read_csv('DatasetModified/dataset3.csv')
dataset3_unknown = pd.read_csv('DatasetModified/dataset3_unknown.csv')
```

In [15]:
```
train_set, test_set = train_test_split(dataset3, train_ratio=0.8)
```

In [16]:
```
naive_bayes = NaiveBayes(train_set, test_set)
naive_bayes.get_accuracy()
```

Out[16]:
```
77.55102040816327
```

In [17]:
```
naive_bayes.change_test_set(dataset3_unknown)
y_predicted = naive_bayes.predict()
_, counts = np.unique(y_predicted, return_counts=True)
print(f"class 0: {counts[0]}\nclass 1: {counts[1]}")
```

```
class 0: 35
class 1: 26
```

در اجرا نیز مشاهده میشود که روی دیتاست با دقت ۷۷.۵۵ درصد این پیشبینی صورت میگیرد.