

به نام خدا

عنوان :

گزارش پروژه اول درس یادگیری ماشین

نام و نام خانوادگی :

مهدی فقهی

شماره دانشجویی :

۴۰۱۷۲۲۱۳۶

استاد درس :

دکتر آرش عبدی هجراندوست

## قسمت اول پروژه را با مسئله restaurant شروع می کنیم .

ابتدا در فایل مربوط به restaurant دو فایل به نام های `implement_by_entropy` و `implement_by_gini` را خواهید دید که برای پیدا کردن میزان purity از دو شیوه مختلف کمک گرفته ایم، یکی `entropy` و دیگری استفاده از ضریب جینی .

شکل ساخت درخت تصمیم برای هر دویکی است و شامل مراحل زیر است :

۱. یک کلاس به اسم Decision Tree می سازیم، ورودی آن :

لیستی از نام ویژگی های داده ای که می خواهیم به کمک Decision tree آن را خوشه بندی کنیم . (اسم ستون ها) و حداکثر عمقی که می خواهیم درخت تصمیم پیش رود تا دادگان را دسته بندی کند می باشد .

```
class DecisionTree:
    def __init__(self, list_of_feature_name, max_depth=100):
        self.list_of_feature_name = list_of_feature_name #list of feature our tree have
        self.max_depth = max_depth #maximum deep we can go on tree
        self.first = True #Show us this tree have real node on its root or not
        self.root = Node
```

۲. سپس برای این کلاس توابعی را می سازیم ، با تابع `fit` شروع می کنیم .

```
def fit(self, X, y, map_feature_number):
    #make tree with training data
    map_feature_number = map_feature_number
    father_most_commen_sample = np.argmax(np.bincount(y))
    choice_from_father = "root"
    self.root = self._build_tree(X, y, map_feature_number, father_most_commen_sample, choice_from_father)
```

این تابع ورودی `train` درخت را به صورت `numpy array` می گیرد و به کمک خروجی این ورودی ها درخت تصمیم مربوط را به کمک تابع داخلی `build tree` می سازد .

۳. در تابع `build tree` که یک تابع پیاده سازی شده به صورت بازگشتی است ورودی `X` (دادگان بدون برچسب) و ورودی `Y` (برچسب) را می گیریم، در این تابع لیستی به نام `map feature number` داریم که نشان می دهد هر کدام از ستون های `X` ، نشان دهنده کدام ویژگی از مجموعه ویژگی های درخت اصلی است که در ابتدا برای ساخت درخت از آن استفاده کردیم . همچنین `father most common sample` را داریم که نشان دهنده آن است که اگر این درخت پدری داشته باشد پدر آن از چه برچسبی بیشترین تعداد فرزند را داشته است و همچنین به کمک `choice from father` می توانیم متوجه شویم که با انتخاب کدام شاخه وارد این گره شده ایم .

```

#find most label rapid in one node
most_common_label = self._find_most_common_label(y, father_most_commen_sample)

# stopping criteria
if self._is_finished(depth):
    if len(y) == 0 :
        return Node(choice_from_father = choice_from_father ,value=father_most_commen_sample)
    return Node(choice_from_father = choice_from_father , value=most_common_label)

# get best split
best_feat, list_of_category = self._best_split(X, y, map_feature_number)
map_feature_number.remove(best_feat)
# grow children recursively
dic_of_variable = self._create_split(X[:, best_feat], list_of_category)

list_of_childs = []
for key in dic_of_variable:
    if len(dic_of_variable[key]) != 0 :
        list_of_childs.append(self._build_tree(X[dic_of_variable[key],:],
                                                y[dic_of_variable[key]],
                                                map_feature_number,
                                                most_common_label,
                                                key,
                                                depth + 1))

```

همانطور که در ادامه می بینید ابتدا نگاه می کنیم که بیشترین تعداد را از کدام برچسب در گره فعلی داریم سپس بررسی می کنیم که آیا شرایط پایان ساخت درخت موجود است یا نه .

اگر این شرایط برقرار بود این گره را به عنوان یک برگ در نظر می گیریم و مقداری که برای آن در نظر می گیریم برابر با برچسبی هست که بیشترین تعداد را دارد ، اگر این شرایط برقرار نبود از بین ویژگی های که باقی مانده است یک ویژگی انتخاب می کنیم که بهترین information gain لازم را برای ما بدست آورد، این ویژگی و لیستی از آیتم هایی که این ویژگی دارد را بر می گردانیم که در واقع فرزندان ما برابر خواهند شد با انتخاب هر یک از این آیتم های موجود در ویژگی، پس یک دیکشنری می سازیم که در آن مشخص شده است که با انتخاب هر کدام از آیتم های این ویژگی کدام یک از مجموعه دادگان X به دادگان فرزند انتقال پیدا می کند . پس براساس همین دانش همین تابع را برای فرزندان این گره از درخت صدا می زنیم و لیست فرزندان این گره را می سازیم و در نهایت گره موجود را می سازیم و آن را به پدرش برمی گردانیم .

```

return Node(self.list_of_feature_name[best_feat], choice_from_father, list_of_childs)

```

که شامل اطلاعاتی از این دست است :

(۱) بچه‌های آن با انتخاب کدام ویژگی ساخته شده اند .

(۲) خود این گره بر اساس انتخاب کدام آیتم از ویژگی انتخابی پدرش ساخته شده است .

(۳) چه بچه‌هایی دارد .

۰۴ در قدم بعدی تابع best split را بررسی می‌کنیم، در این تابع یک numpy array X را به عنوان ورودی می‌گیریم و سپس براساس ویژگی‌هایی که داریم یک ویژگی را انتخاب می‌کنیم که information gain بالاتری داشته باشد و آن ویژگی و لیستی از آیتم‌هایی که دارد را بر می‌گردانیم .

```
def _best_split(self, X, y, features):
    #this function find best feature can make greatest
    # difference between father entropy and its weighted
    # size of the its child entropy
    best_split_score = -np.inf
    best_feat=None
    best_list_of_category = None

    for feat in features:
        X_feat = X[:, feat]
        all_data_feat = self.data[:,feat]
        list_of_category = np.unique(all_data_feat)
        score = self._information_gain(X_feat, y, list_of_category)
        if score > best_split_score:
            best_split_score = score
            best_feat = feat
            best_list_of_category = list_of_category

    return best_feat, best_list_of_category

def tree_show(self):
```

• در قسمت بعد تابع `_information_gain`, `_find_Remainder` را خواهیم داشت ، ابتدا ذکر باید کنم که چه بخواهیم حاصل را به کمک `gini` یا `entropy` بدست آوریم ساختار کلی این توابع فرقی نخواهد کرد صرفا باید هر جا که `entropy` دید `gini` قرار داد و برعکس ، پس در ادامه فقط نحوه پیاده سازی این دو تابع را با هم خواهیم دید، ولی اگر بخواهیم خود کارکرد این دو توابع را ببینیم، ابتدا به کمک تابعی که قرار است میزان Purity را حساب کنیم ، `Purity` گرده پدر را حساب می کنیم، سپس حاصل جمع وزن دار Purity فرزندان را هم بدست می آوریم و حاصل پدر را از جمع وزن دار فرزندان کم کرده و `information gain` را بدست می آوریم .

```
def _find_Remainder(self, dic_of_variable: dict, n, y):
    # find the Weighted size of the child entropy
    Remainder = 0
    for key in dic_of_variable:
        Remainder += (len(dic_of_variable[key])/n) * self._entropy(y[dic_of_variable[key]])

    return Remainder

def _information_gain(self, X, y, list_of_category):
    # find the difference between father entropy and its weighted size of the its child entropy
    Entropy = self._entropy(y)
    dic_of_variable = self._create_split(X, list_of_category)
    n = len(y)

    Remainder = self._find_Remainder(dic_of_variable, n, y)
    return Entropy - Remainder
```

در تابع `find Remainder` حاصل جمع وزن دار، `Impurity` فرزندان را بدست می آوریم و آن را برمی گردانیم . فرزندان به کمک ورودی `dic_of_variable` شناخته می شوند که شامل یک دیکشنری است که نشان داده است هر کدام از آیتم های یک ویژگی شامل چه سطری از داده اصلی پدر می شوند و به عنوان فرزند از پدر جدا می شوند .

۶. در این قسمت به بررسی دو تابع gini , entropy که برحسب آن میزان Purity را حساب می کنیم می پردازیم . همانطور که می دانیم فرمول محاسبه آنتروپی برابر است با :

$$E = - \sum_{i=1}^N P_i \log_2 P_i$$

```
def _entropy(self, y):#calculate entropy for use
    proportions = np.bincount(y) / len(y)
    entropy = -np.sum([p * np.log2(p) for p in proportions if p > 0])
    return entropy
```

که به شکل زیر در پایتون پیاده سازی می شود .  
و فرمول gini نیز برابر است با :

$$I_G(p) = \sum_{i=1}^J \left( p_i \sum_{k \neq i} p_k \right) = \sum_{i=1}^J p_i (1 - p_i) :$$

و چون اینجا مسئله ما دو کلاس است نوشتن یکبار آن نیز کافی است که برابر با  $p * q$  است که برابر با  $p * (1-p)$  می باشد البته در این حاصل ما برای سادگی از یک ضرب در ۲ فاکتور گرفتیم که در حقیقت حاصل اصلی می بایست برابر می شد با :

$$p * (1 - p) + q * (1 - q) = 2 * p * (1 - p)$$

که همانطور که می دانیم ضرب در ۲ در پیدا کردن بزرگی حاصل نهایی تاثیری ندارد پس آن را حذف کردیم و حاصل نهایی را برابر با  $p * (1 - p)$  که همان  $p * q$  است در نظر گرفتیم که در بعضی از پیاده سازی ها نیز برای حل مسائل دو کلاسه از این روش استفاده کرده بودند پس :

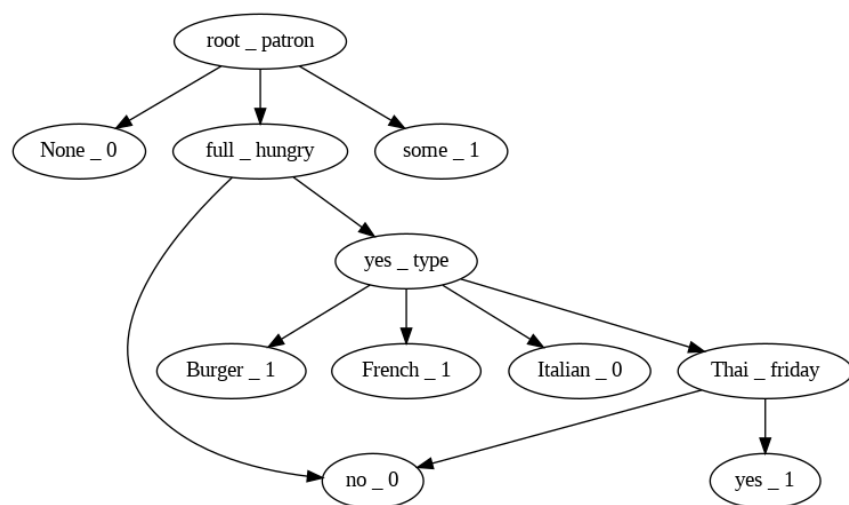
```
def _gini(self, y):#calculate gini for use
    if len(y)==0:
        return 0
    proportions = np.bincount(y) / len(y)
    gini = proportions[0]*(1-proportions[0])
    return gini
```

۷. به کمک تابع `is finished` چگونه شرط اینکه یک گره باید برگ باشد یا نه را بررسی می کنیم. ساده ترین شکل پیاده سازی این تابع به شکل زیر است :

```
def _is_finished(self, depth): # check the situation of continue making tree
    if (depth >= self.max_depth
        or self.n_class_labels == 1):
        return True
    return False
```

که نگاه می کند اگر میزان عمق در حال بیشتر شدن از حد تعیین شده باشد و اگر از بین برچسب ها فقط یک نوع برچسب مانده باشد که به معنی `purity` یا `impurity` صفر است دیگر از ساخت گره جدید دست برمی دارد. در نهایت پس از اجرا این الگوریتم بر روی مسئله `restaurant` درخت آن را به شکل مصور می سازیم.

```
restaurant_decision_tree_model.make_plot()
```



که اگر دقت شود همان شکل کشیده شده در مرجع می باشد.

## قسمت دوم پروژه مسئله تشخیص سرطان بدخیم و خوش خیم :

این بخش حاوی دو تا folder است ، folder اول تمامی نتایج برحسب entropy و folder دوم برحسب gini بدست آمده است .

در هر کدام از آن folder ها خود سه folder است : (main & pca & threshold)  
در main داده های پیوسته را به ساده ترین شکل ممکن به ازای دسته بندی ( ۳ ، ۵ ، ۸ و ۱۰ ) به گسسته مورد آزمایش قرار دادم .  
در threshold ، لیست را مرتب کرده و بر حسب یک ویژگی و بر حسب اینکه کدام عدد می تواند معیاری مناسبی برای تقسیم آن به دو دسته (بزرگتر از آن عدد و کوچکتر از آن عدد ) ، مناسب تر باشد ، مورد آزمایش قرار دادم .  
در قسمت pca از روش از دو روش بالا بهترین نتایج بدست آمده را با کاهش ابعاد به ۱۰ بعد مورد مطالعه قرار داده و نتایج رو گزارش کردم.  
از آنجا که منطق کدهای gini , entropy یکی است و فقط در تابع پیدا کردن میزان بی نظمی با هم متفاوت هستند تمامی مراحل را فقط در اینجا برای entropy برحسب کد توضیح می دهم و فقط نتایج آخر بدست آمده حاصل از gini را نشان خواهم .  
کدهای تمامی قسمت ها درون folder ها موجود است .

ابتدا به کمک کتابخانه pandas داده های موجود در part2.csv را می خوانم و نگاهی به دادگام می اندازم .

```
data = pd.read_csv("part2.csv")
data.head(12)
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	conc points_m
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.30010	0.14710
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.08690	0.07017
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.19740	0.12790
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.24140	0.10520
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.19800	0.10430
5	843786	M	12.45	15.70	82.57	477.1	0.12780	0.17000	0.15780	0.08089
6	844359	M	18.25	19.98	119.60	1040.0	0.09463	0.10900	0.11270	0.07400
7	84458202	M	13.71	20.83	90.20	577.9	0.11890	0.16450	0.09366	0.05985
8	844981	M	13.00	21.82	87.50	519.8	0.12730	0.19320	0.18590	0.09353
9	84501001	M	12.46	24.04	83.97	475.9	0.11860	0.23960	0.22730	0.08543
10	845636	M	16.02	23.24	102.70	797.8	0.08206	0.06669	0.03299	0.03323
11	84610002	M	15.78	17.89	103.60	781.0	0.09710	0.12920	0.09954	0.06606

12 rows x 33 columns



- سپس دادگان برحسب را حذف می‌کنم و قسمت‌های حشو جدول را حذف می‌کنم.
- دادگان را به نسبت ۷۵ و ۲۵ برای آموزش و تست تقسیم می‌کنم به صورت رندوم.

```
from sklearn.model_selection import train_test_split# Shuffle and split the data into training and testing subsets
X_train, X_test, y_train, y_test = train_test_split(features,
                                                    diagnosis, test_size=0.25, random_state=42)
```

سپس تابعی به نام `find_data_set_range_for_each_coloumn_not_catogorical` را بوجود می‌آورم که برحسب تعداد برشی که می‌خواهد ایجاد کنم یک دیتافریم می‌گیرد و سپس براساس نام ویژگی‌هایی که پیوسته هستند، بزرگترین و کوچکترین این موجودیت‌ها را پیدا می‌کند و با تقسیم بر تعداد برش مشخص می‌کند طول بازه هر کدام از این ویژگی‌ها چقدر باشد.

```
def find_data_set_range_for_each_coloumn_not_catogorical(data_set_feature, name_of_coloumn_not_catogorical, number_of_cut):

    data_set_feature_details = data_set_feature.describe()
    diff = data_set_feature_details.iloc[7,:] - data_set_feature_details.iloc[3,:]
    diff = diff/number_of_cut
    print(diff)
    return diff
```

تابع دیگری به اسم `make_coloumn_not_catogorical_catogorical` که حاصل تابع قبل که یک دیکشنری هست که در آن اسم ویژگی و طول بازه آن را می‌گیرد و برحسب تعداد برش داده‌های پیوسته را گسسته می‌کند. برای مثال اگر ما ۳ تا دسته بخواهیم یک داده پیوسته را بزنیم بعد از این که طول بازه را برحسب تابع قبل پیدا کرده باشیم.

سه حالت زیر را خواهیم داشت:

در اینجا  $L$  به معنی طول بازه است.

1.  $0 \leq \text{feature} < 1 * L$
2.  $1 * L \leq \text{feature} < 2 * L$
3.  $2 * L \leq \text{feature} < \dots$

از آنجا که برحسب مطالعه ویژگی‌ها متوجه شدم مقادیر پیوسته ما نمی‌توانند کمتر از صفر باشند بازه بندی به شکل زیر در آمده است.

```
def make_column_not_catogorical_catogorical(name_column_with_bais_range, data_set_feature, columns_name, number_of_cut):
    data_set_feature_copy = data_set_feature.copy()
    for column in columns_name:
        for index in data_set_feature.index:
            for c in range(number_of_cut):

                if data_set_feature.loc[index, column] != None and (data_set_feature.loc[index, column] >=
                    name_column_with_bais_range[column]*c) and \
                    (data_set_feature.loc[index, column] < name_column_with_bais_range[column]*(c+1)) and c !=
                        number_of_cut-1:

                    map = c

                elif c == number_of_cut-1 and data_set_feature.loc[index, column] != None and (data_set_feature.loc[index,
                    column] >= name_column_with_bais_range[column]*c):

                    map = c

                if data_set_feature_copy[column][index] != None and map == number_of_cut - 1:

                    data_set_feature_copy[column][index] = f'{column} >= {map*name_column_with_bais_range[column]:.3f}'
                elif data_set_feature_copy[column][index] != None:

                    data_set_feature_copy[column][index] = f'({map+1}*name_column_with_bais_range[column]:.3f) > {column}
                    >= {map*name_column_with_bais_range[column]:.3f}'

    return data_set_feature_copy
```

بعد از این عمل دیتاست ما به شکل زیر خواهد شد :

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean
287	13.613 > radius_mean >= 6.806	19.713 > texture_mean >= 9.857	93.720 > perimeter_mean >= 46.860	776.867 > area_mean >= 0.000	0.074 > smoothness_mean >= 0.037	0.097 > compactness_mean >= 0.000	0.142 > concavity_mean >= 0.000	0.067 > concave points_mean >= 0.000	symmetry_mean >= 0.125
512	13.613 > radius_mean >= 6.806	19.713 > texture_mean >= 19.713	93.720 > perimeter_mean >= 46.860	776.867 > area_mean >= 0.000	smoothness_mean >= 0.074	0.195 > compactness_mean >= 0.097	0.285 > concavity_mean >= 0.142	0.134 > concave points_mean >= 0.067	symmetry_mean >= 0.125
402	13.613 > radius_mean >= 6.806	19.713 > texture_mean >= 9.857	93.720 > perimeter_mean >= 46.860	776.867 > area_mean >= 0.000	0.074 > smoothness_mean >= 0.037	0.097 > compactness_mean >= 0.000	0.142 > concavity_mean >= 0.000	0.067 > concave points_mean >= 0.000	symmetry_mean >= 0.125
446	radius_mean >= 13.613	texture_mean >= 19.713	perimeter_mean >= 93.720	1553.733 > area_mean >= 776.867	smoothness_mean >= 0.074	0.195 > compactness_mean >= 0.097	0.285 > concavity_mean >= 0.142	0.134 > concave points_mean >= 0.067	symmetry_mean >= 0.125

خوب پس ما برحسب داده train داده‌های پیوسته را گسسته می‌کنیم و سپس نتایج بدست آمده را ذخیره و دادگان تست را نیز همانند دادگان train برای گرفتن پیش‌بینی حاصل بر روی آنان بر اساس درخت تصمیم ساخته شده برحسب این نوع دادگان جدید، آن را به فرمت جدید در خواهیم آورد.

حال فقط مانده است که توضیحی درباره آنکه درخت تصمیم در این سوال چه تغییری نسبت به درخت تصمیم قبل کرده است بدهیم .

ابتدا باید با دو اصطلاح به نام **post prune tree** و **pre prune tree** آشنا شویم .

گاهی ما یک درخت کامل را می‌سازیم و آنگاه از نظر حجم درخت ( تعداد برگ‌ها و عمق آن ) ، میزان حداقل خلوص در برگ‌های آن و حداقل دقت هر برگ مورد بررسی قرار می‌دهیم و در صورت پیدا کرد برگ‌ها نامناسب تا جایی که می‌توانیم این برگ‌ها را هرس و گره‌های پدر را تبدیل به برگ می‌کنیم و اینگونه درخت را هرس می‌کنیم .

گاهی نیز در هنگام آموزش این معیارها را مورد بررسی قرار می‌دهیم و اجازه نمی‌دهیم که تعداد برگ‌ها و عمق درخت ما از یک حد بیشتر شوند .

تمامی این روش‌ها برای جلوگیری از **overfit** شدن درخت تصمیم است و افزایش قدرت **generality** مدل است .

روش **post** یک سربار و هزینه زمانی مضاعف ایجاد می‌کند اما به دلیل اینکه درخت به صورت کامل ساخته شده است، می‌توانیم با دید بهتر به یک مدل ایده‌آل تر برسیم .

در روش **pre** درست است که سربار و هزینه زمانی توسط خود مدل کنترل و به صورت حریصانه سعی در پیدا کردن بهترین نتیجه است اما نمی‌تواند تضمین بدهد که درخت بوجود آمده توانایی بهبود دارد .

سعی شده است در ورژن جدید **pre prune tree** به طور کامل انجام شود .

در درخت قبلی صرفاً عمق برای ما مهم بود ولی در درخت فعلی سعی کردیم نگاهی به میزان آشفتگی نیز داشته باشیم و اگر به حد معقولی رسیده باشد از ساخت شاخه‌های بیشتر اجتناب کنیم و همچنین اگر تعداد آیم‌های یک گره به حد خوبی رسیده باشد آن را گسترش ندهیم .

برای اینکه بتوانیم بعد از انجام ساخت درخت عمل **post prune tree** را انجام دهیم نیز از روش **post prune tree** کتابخانه **sklearn** الگوبرداری کردیم که در ادامه آن را با هم بررسی خواهیم کرد .

```
class DecisionTree_Binary_Res_entropy:
    def __init__(self, list_of_feature_name, max_depth=100, minmum_diference_entropy=0,
        number_of_child_must_be_in_each_node=2):
        self.list_of_feature_name = list_of_feature_name
        self.max_depth = max_depth
        self.first = True
        self.root = Node
        self.minmum_diference_entropy = minmum_diference_entropy
        self.number_of_child_must_be_in_each_node = number_of_child_must_be_in_each_node
```

۰۱. خوب برای ساخت درخت جدید علاوه بر ورودی‌های قبل نیاز است که ورودی‌هایی مانند حداقل تفاوت ایجاد شده entropy یا gini اضافه شود که یعنی اگر انتخاب بهترین ویژگی برای تقسیم بندی باعث کاهش entropy یا gini به حد دلخواه ما نشود آن گره به صورت یک برگ در نظر گرفته شود و گسترش پیدا نکند .

```
# get best split
best_feat, list_of_category, entropy = self._best_split(X, y, map_feature_number)
if entropy <= self.minmum_diference_entropy :
    return Node(choice_from_father=choice_from_father,
                value=perobabilty_of_pasetive,
                entropy=self._entropy(y),
                leaf=True,
                emprical_error = emperical_erro,
                use_father_value = False,
                )
```

در اینجا نیز به کمک یک کنترلر جدید دیگه در صورتی که تعداد فرزندان یک گره کمتر از میزان کنترلی خواست ما شود آن گره برگ در نظر گرفته می‌شود .

```
def _is_finished(self, depth, number_of_childs_have):
    if (depth >= self.max_depth
        or self.n_class_labels == 1
        or number_of_childs_have < self.number_of_child_must_be_in_each_node):
        return True
    return False
```

از مواردی تغییر دیگه‌ای که در این درخت انجام شده است ، تغییر در ویژگی‌های کلاس Node است :

```
class Node:
    def __init__(self, feature=None,
                  feat_number=None,
                  most_commen_child=None,
                  choice_from_father=None,
                  list_of_childs = None,
                  number_of_each_child_have_childs = None,
                  entropy=None,
                  use_father_value=None,
                  leaf=None,
                  emprical_error = None,
                  value=None):
```

در درخت قبل با ویژگی feature که در واقع مشخص می‌کرد فرزندان این گره بر حسب کدام ویژگی گسترش پیدا کردند و choice from father که نشان دهند آن بود که انتخاب کدام زیر شاخه از پدر باعث بوجود آمدن این گره شده بود آشنایی پیدا کردیم . مواردی مانند most common child در واقع یک روش و هیرستیک است برای هندل کردن برخی اتفاقات خاص، همانند روبه‌رو شدن با یک مقدار گسسته در درخت که قبلاً آن را نداشته‌ایم یا مواردی که ممکن است یک ویژگی مقدار null داشته باشد ما برای حرکت دادن جریان در درخت به سمت فرزندی در این شرایط متمایل می‌شویم که بیشترین تعداد بچه‌ها را به سمت خود کشانده است . در قسمت دیگر لیستی خواهیم داشت که هر کدام از فرزندان این گره چه تعداد فرزند دارند و مقدار entropy یا برای درخت متکی بر gini میزان ناخالصی این درخت چقدر است . ( این موارد برای post prune با توجه به عملکرد ما می‌توانند کمک کننده باشند) همچنین با flage use father value می‌توانیم متوجه شویم که برای پیش بینی نهایی آیا از پیش بینی پدرش برای پیش بینی استفاده می‌کند یا از پیش بینی خود برای اینکار استفاده می‌کند ( این کار برای زمانی خوب است که ممکن است برگی ساخته شده باشد که میزان آشفتگی آن ۵۰ درصد باشد یعنی از دو برچسب به یک اندازه دارد و ما مجبور هستیم برای پیش بینی از پدر کمک بگیریم یا برگی ساخته‌ایم که هیچ یک از دادگان به سمت آن سوق پیدا نکرده است پس مجبور هستیم که برای انتخاب از معیار پدر آن استفاده کنیم برای دسته بندی ) و همچنین پرچم leaf که مشخص می‌کند این Node یک گره است یا برگ که برای پیمایش درخت ساخته شده از آن کمک می‌گیریم . و در نهایت empirical error که مشخص می‌کند چه تعداد خطا در این گره یا برگ داریم در هنگام ساخت درخت با مجموع دادگان آموزش و آخرین مقدار ما که value است، مشخص می‌کند که احتمال مثبت بودن این گره یا برگ چقدر است .

چرا از احتمال استفاده کردیم ؟

به این دلیل از احتمال استفاده کردیم که می‌خواستیم معیاری مثل AUC را پیدا سازی کنیم و دیگر نمی‌توانیم به صورت مطلق همانند سوال قبل بگوییم این گره یا برگ مثبت است یا منفی بلکه مجبور هستیم بگوییم به چه احتمالی مثبت است که به عنوان value در نظر گرفته می‌شود .

خوب حالا که این موارد را توضیح دادیم می‌توانیم عملیات post prune انجام شده توسط خود را انجام دهیم .  
برای کنترل دقت و گستردگی درخت باید فرمولی بسازیم که هم به سبب آن بتوانیم دقت را کنترل کنیم و هم از اینکه درخت بزرگ شود جلوگیری عمل کنیم .  
بزرگ شدن درخت در بسیاری از رفرنس‌ها به عنوان پیچیدگی درخت در نظر گرفته شده است به عبارتی هرچه قدر درخت تعداد برگ زیادی داشته باشد پیچیده تر است و عمق بیشتری نیز دارد .

به همین دلیل یک فرمول به شکل زیر ساخته شده است :

$$\bar{h}(x_i) = \sum_{t=1}^T \bar{\alpha}_t h_t(x_i) \quad \min_{\rho_h \geq \rho \text{ or } \|\bar{\alpha}\|_1 \leq \frac{1}{\rho}} \frac{1}{m} \sum_{i=1}^m e^{-y_i \bar{h}(x_i)} + \lambda \|\bar{\alpha}\|_1 \quad (7.31)$$

convex and differentiable upper bound on the zero-one loss (see Theorem 7.7 slide).

regularization term

قسمت اول که empirical error کل درخت است و قسمت دوم هم تعداد برگ‌ها است که در یک regularization term ضرب می‌شود و هدف ما پیدا کردن min این مقدار بر روی درخت بر حسب regularization term هستیم .

پس ما وقتی درخت را ساختیم برحسب این این فرمول یک درخت بهینه را متناسب با درخت ساخته شده بیرون می‌کشیم که این فرمول را به کمترین حد خود برساند و آن را به عنوان درخت بهینه عرضه می‌کنیم .

اما سوال دیگری پیش می‌آید به ازای چه میزان regularization ؟

خوب این هاپرپارامتر را با انجام آزمایش پیدا می‌کنیم و بهترین میزان برای آن را بر حسب accuracy , f1 score و میزان عمق و تعداد برگ‌ها پیدا می‌کنیم که در ادامه خواهیم داشت .

حال برویم سراغ پیاده سازی این قسمت :

```
def post_prune_tree(decision_tree, alpha):  
  
    while True:  
        k = 0  
        list_of_gain_each_child = []  
        for child in decision_tree.root.childs:  
            k += 1  
            if not child.is_leaf():  
                gain, best_child = child.best_del_achivement(alpha)  
                list_of_gain_each_child.append([gain, k, best_child])  
  
        if len(list_of_gain_each_child) == 0:  
            print("I cant find any good point1")  
            break  
  
        for item in list_of_gain_each_child:  
            number_success = 0  
            if item[0] < 0 :  
                number_success += 1  
                item[2].leaf = True  
        if number_success == 0 :  
            print("I cant find any good point2")  
            break  
  
    return decision_tree, main_tree_complexity, emperical_erro, root_number_depth, root_number_leaves
```

این تابع یک درخت تصمیم میگیرد و یک  $\alpha$  به عنوان regularization سپس تمام فرزندان درخت تصمیم را بررسی می کند و تمام فرزندان گره ای که حذف آنان و تبدیل آن به برگ باعث بوجود آمدن یک achievement می شود با میزان gain حاصل را برمی گرداند و ما این فرزندان و میزان gain حاصل از حذف آنان را در یک لیست نگه می داریم .

سپس اگر این لیست تهی بود که یعنی هیچ فرزندی برای حذف نیست که ارزش حذف داشته باشد ، اگر فرزند نیز باشد ولی حذف آن باعث اثر مثبت نشود یعنی gain منفی داشته باشد نیز عملاً یعنی حذف از آن بی فایده است و عملیات تمام می شود ، در غیر این صورت تمام گره هایی که تبدیل شدند آنان به برگ مفید باشند تبدیل به برگ می شوند که باعث کاهش فرمولی که دنبال نقطه min آن هستیم می شود ، سپس درخت جدید و میزان پیچیدگی آن بر حسب فرمول بالا و مقدار خطای جدید آن و میزان عمق جدید آن و تعداد برگ جدید آن برگشت داده می شود .

- در اینجا یک تابع دیگر داریم که روی فرزندان root صدا زده شده است به نام `best del achievement`.
- این تابع که جز توابع کلاس Node است کار زیر را انجام می دهد.
۱. به دنبال یک گره می گردد که تمامی فرزندان آن برگ باشد.
  ۲. اگر به یک گره رسید که یکی از فرزندان آن خود نیز گره باشد به دنبال فرض بالا در آن گره می گردد.
  ۳. سپس وقتی آن را پیدا کرد `empirical error` تمام فرزندان برگ آن را جمع می کند و حاصل را از `empirical error` گره کم می کند و سپس حاصل را دوباره از ضرب `alpha` در تعداد فرزندانی که حذف می شوند که می شود فرزندان خودش کم می کند و حاصل `gain` را اینگونه بدست می آورد و در نهایت آن گره و میزان `gain` حاصل از تبدیل شدن آن به یک برگ را بر می گرداند.
  ۴. سپس تمام کسانی که از تمامی زیر شاخه ها ویژگی یک را داشته باشند با `gain` حاصل از حذفشان برگشت داده می شوند و از بین آنان کسی انتخاب می شود که بیشترین میزان `gain` را داشته باشد و به عنوان کاندید برای برگ شدن به تابع قبل داده می شود.

```
def best_del_achievement(self, alpha):

    is_all_of_its_child_leaf = True

    for child in self.chlds :

        if not child.is_leaf():

            is_all_of_its_child_leaf = False
            break

    if is_all_of_its_child_leaf:

        child_error = 0

        for child in self.chlds:
```

```
        for child in self.chlds:
            child_error += child.empirical_error

    return (self.empirical_error - child_error) - (alpha*(len(self.chlds)-1)), self

else:
    best_achievement = np.inf
    best_child_for_del = None
    for child in self.chlds:
        if not child.is_leaf():
            child_achievement, child = child.best_del_achievement(alpha)

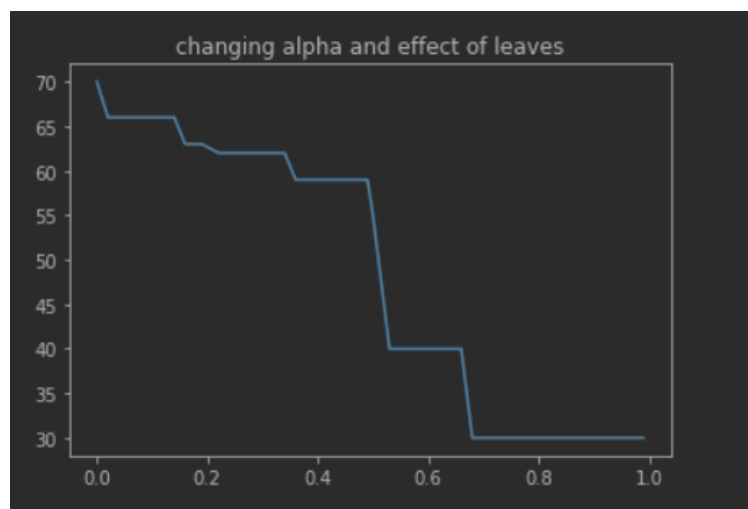
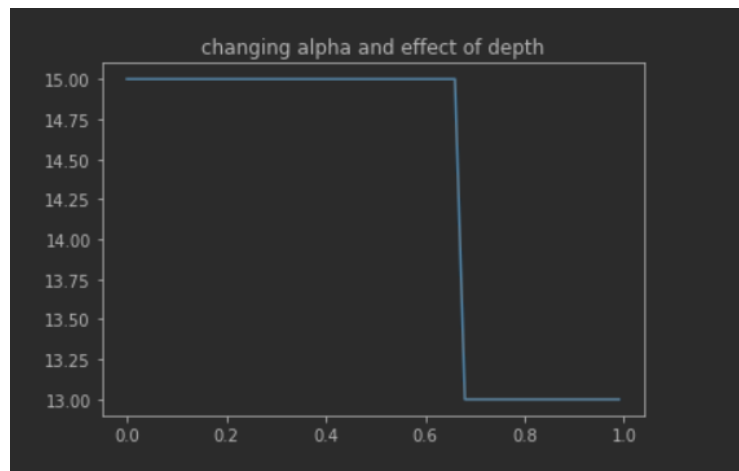
            if best_achievement > child_achievement:
                best_achievement = child_achievement
                best_child_for_del = child
    return best_achievement, best_child_for_del
```

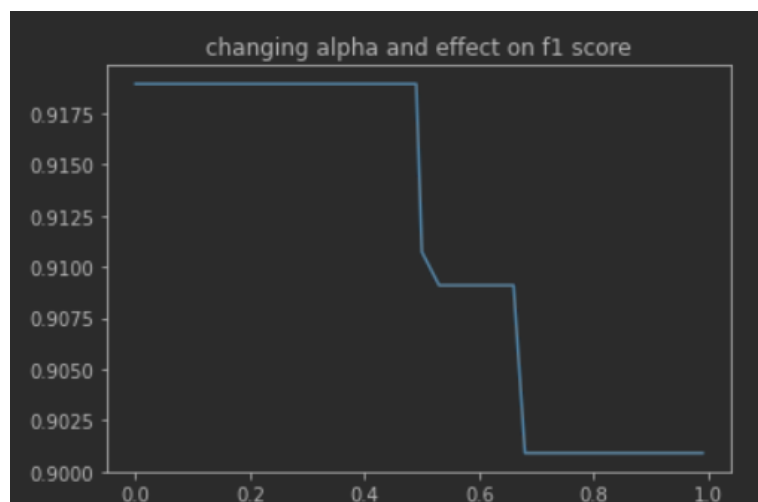
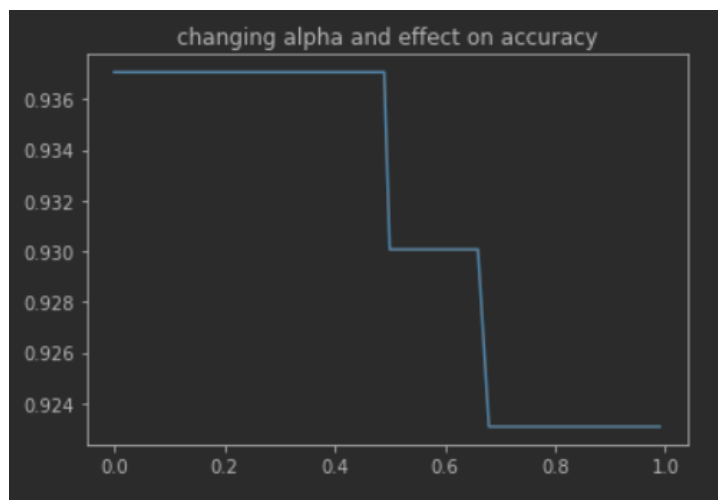
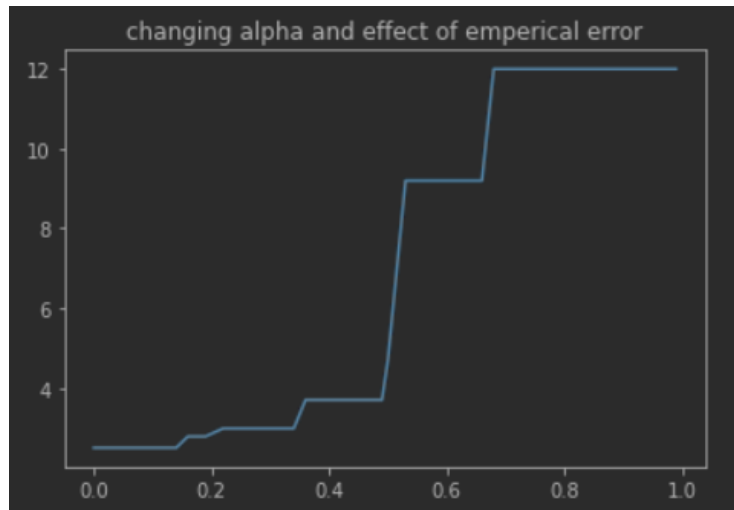


در قدم بعد تابعی ساختیم که به ازای مقادیر مختلف آلفا درخت بهینه را پیدا کند تا بتوانیم بهترین درخت را برحسب نموداری که این تابع به من می‌دهد پیدا کنیم نمودارهایی بر حسب دقت ، عمق ، تعداد برگ .  
این آزمایش را همانطور که در زیر می‌بینید به ازای ۶۰ مقدار آلفا بین ۰ تا ۱ حساب کردم .

```
def make_plot_to_find_best_alpha(model,X_validation,Y_validation):  
  
    random_alpha_numbers = np.sort(np.random.choice(  
        range(1, 100),  
        60,  
        replace=False) * (1/100))  
  
    random_alpha_numbers = np.insert(random_alpha_numbers, 0, 0)  
  
    list_of_model = [copy.deepcopy(model) for item in random_alpha_numbers]  
  
    list_of_depth = []  
    list_of_leaves = []  
    list_of_accuracy = []  
    list_of_f1_score = []  
    list_of_emperical_error = []  
    list_of_main_tree_complexity = []  
    k = 0  
    for alpha in random_alpha_numbers:  
        # print(alpha)  
        model = list_of_model[k]  
        print("#####")  
        decision_tree,main_tree_complexity,empercal_erro,root_number_depth,root_number_leaves = post_prune_tree  
            (model,alpha)
```

بیا باید تاثیر این عمل را بر روی داده‌هایی که هر کدام از ویژگی‌های پیوسته آنان به سه دسته تقسیم بنده شده است ببینیم .





همانطور که مشاهده می کنید هرچقدر مقدار آلفا بزرگتر میزدن عمق کاهش پیدا می کند و تعداد برگ ها کمتر می شود و به ازای اون میزان دقت و f1 از یک جایی به بعد که مقاومت می کند کاهش پیدا می کند و مقدار خطای داخلی نیز در حال افزایش است . پس باید بر حسب این نمودارها یک آلفا را در نظر بگیریم که ضمن آنکه در آن دقت کاهش پیدا نکند میزان برگ ها تا حد خوبی کمتر شوند که با این حساب می شود آلفا برابر با مثلاً 0.4 .

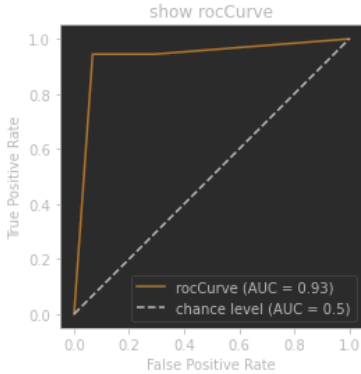
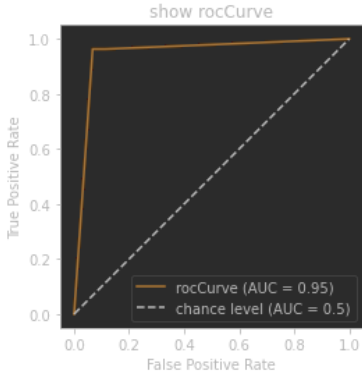
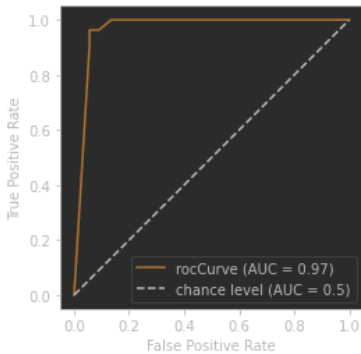
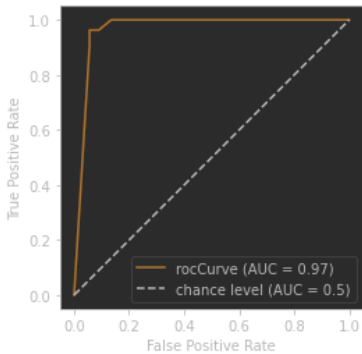
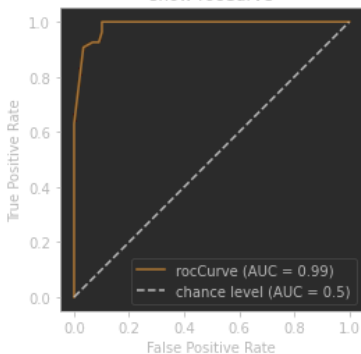
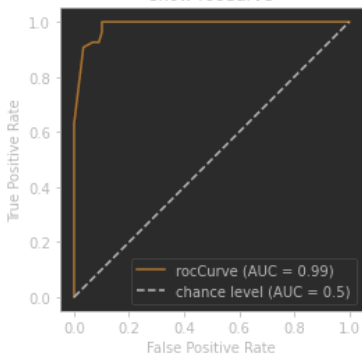
برحسب همین نمودارها من سعی کرده ام بهترین الفاو در نتیجه بهینه ترین درخت را در تمامی مراحل بدست آورم لذا از اینجا به بعد که می خواهم گزارش ارائه کنم بپرض بهینه بودن درخت موجود است .

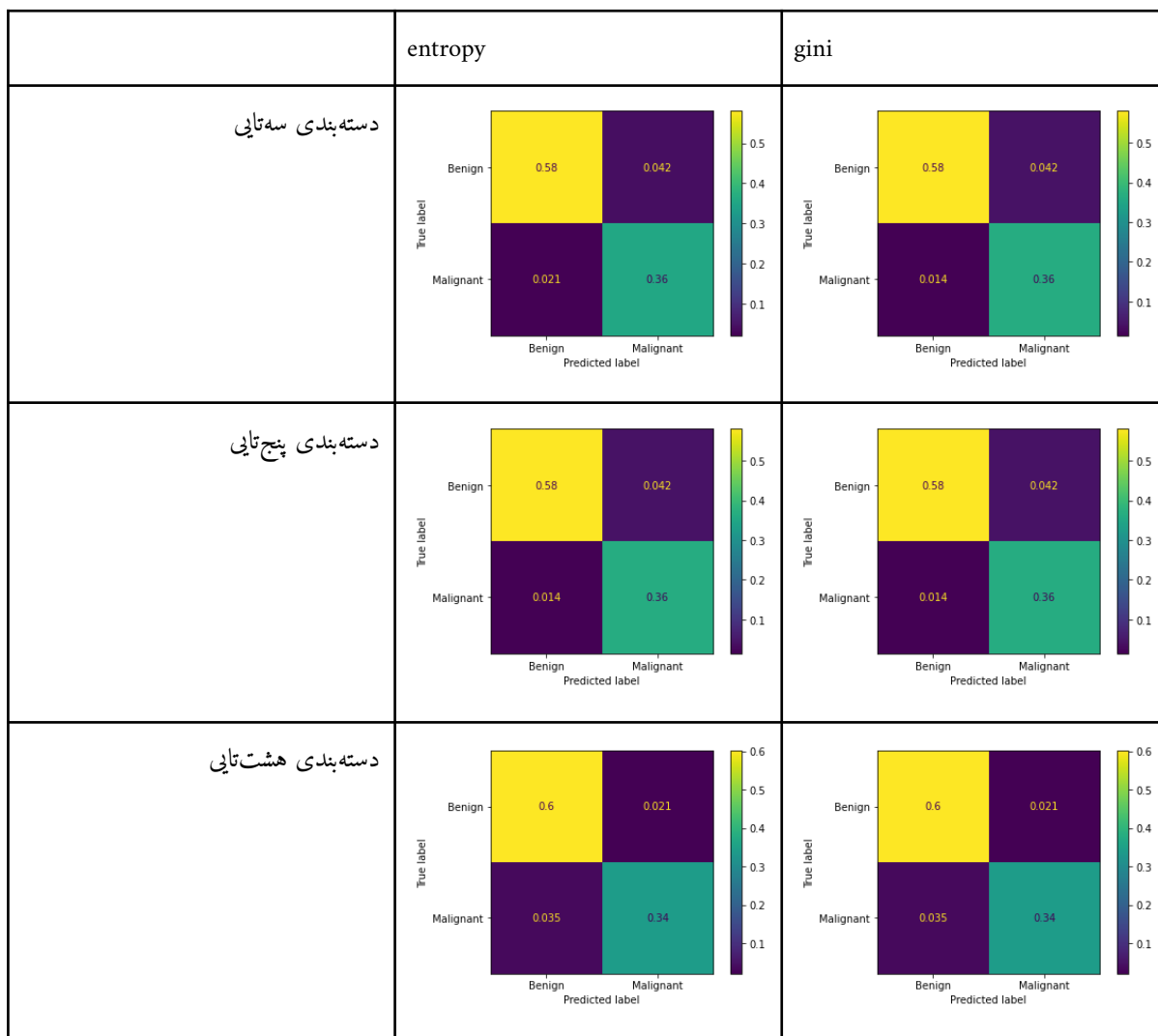
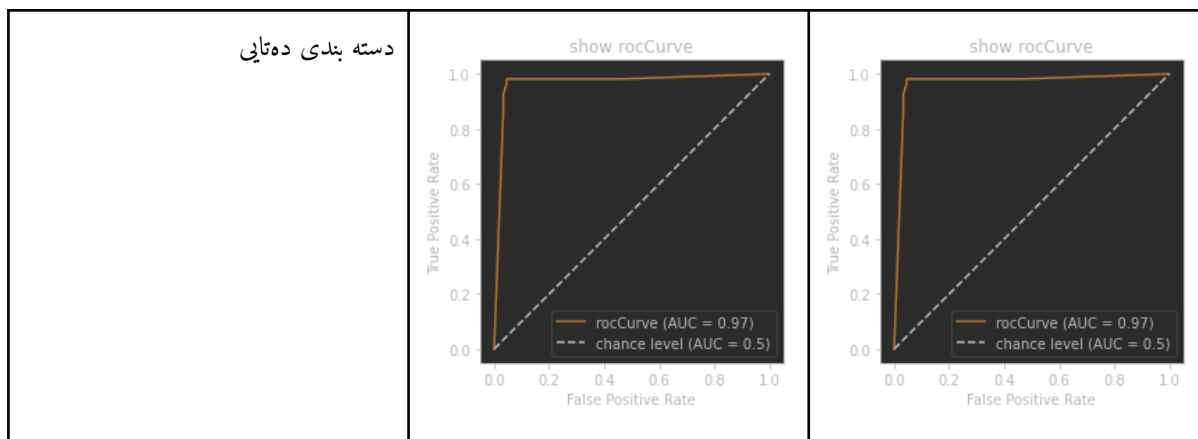
معیارهای اندازه گیری و عمق و تعداد برگ درخت :

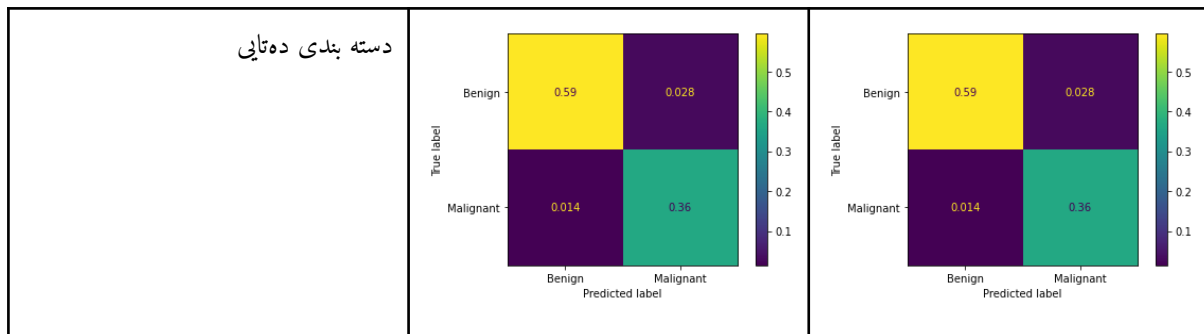
	accuracy	precision	recall	F1 Measure
دسته بندی سه تایی entropy	0.937	0.895	0.944	0.919
دسته بندی سه تایی gini	0.944	0.897	0.963	0.929
دسته بندی پنج تایی entropy	0.944	0.897	0.963	0.929
دسته بندی پنج تایی gini	0.944	0.897	0.963	0.929
دسته بندی هشت تایی entropy	0.944	0.942	0.907	0.925
دسته بندی هشت تایی gini	0.944	0.942	0.907	0.925
دسته بندی ده تایی entropy	0.958	0.929	0.963	0.945
دسته بندی ده تایی gini	0.958	0.929	0.963	0.945

	میزان عمق	تعداد برگ
دسته بندی سه تایی entropy	15	57
دسته بندی سه تایی gini	14	71
دسته بندی پنج تایی entropy	7	43
دسته بندی پنج تایی gini	7	43
دسته بندی هشت تایی entropy	4	24
دسته بندی هشت تایی gini	4	24
دسته بندی ده تایی entropy	5	65
دسته بندی ده تایی gini	5	62

## نمودارهای $Auc$ , Confusion matrix

	entropy	gini
دسته بندی سه تایی	 <p>show rocCurve</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>rocCurve (AUC = 0.93)</p> <p>chance level (AUC = 0.5)</p>	 <p>show rocCurve</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>rocCurve (AUC = 0.95)</p> <p>chance level (AUC = 0.5)</p>
دسته بندی پنج تایی	 <p>show rocCurve</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>rocCurve (AUC = 0.97)</p> <p>chance level (AUC = 0.5)</p>	 <p>show rocCurve</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>rocCurve (AUC = 0.97)</p> <p>chance level (AUC = 0.5)</p>
دسته بندی هشت تایی	 <p>show rocCurve</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>rocCurve (AUC = 0.99)</p> <p>chance level (AUC = 0.5)</p>	 <p>show rocCurve</p> <p>True Positive Rate</p> <p>False Positive Rate</p> <p>rocCurve (AUC = 0.99)</p> <p>chance level (AUC = 0.5)</p>





در ادامه به بررسی یک دیگر از مدل های تبدیل داده های پیوسته به گسسته همراه همدیگر خواهیم بود .

در این روش به جای آنکه کمیت های پیوسته را به صورت preprocess تبدیل به کمیت های گسسته کنیم ، خود عمل گسسته سازی را به صورت یک فرآیند یادگیری ماشین خواهیم داشت .

در اینجا خط برش ویژگی ، خط برشی خواهد بود که بیشترین میزان information gain را درخت مورد نظر بوجود بیاورد .  
ما در اینجا یک درخت باینری بر حسب یک خط برش خواهیم ساخت .

اگر تعداد خط برش بیشتر شود می شود درخت با تعداد فرزندان بیشتر داشت که این نیز خود یک hyperparameter است و از آنجا که سوال از ما فرم با خط برش بالاتر برای این مسئله را نخواسته ما این مسئله را به ساده ترین شکل ممکن حل و نتایج را اعلام می کنیم .  
ما این درخت را یکبار بر حسب entropy , بار دیگر gini مورد مقایسه قرار می دهیم .

برای پیاده سازی درخت با همچنین ویژگی تنها کاری باید انجام دهیم این است که برای پیدا کردن بهترین معیار برای تقسیم بندی وقتی یک ویژگی را مورد بررسی قرار می دهیم علاوه بر آن دادگان را به صورت مرتب از کوچک به بزرگ بررسی کنیم و از میان داده های پیوسته یک نقطه برای خط برش پیدا کنیم که دیتاهای بزرگتر از آن به عنوان فرزندان سمت راست و کوچکتر از آن به عنوان فرزندان سمت چپ تقسیم بندی شوند که این کار به خوبی در زیر انجام شده است . این نقطه می تواند خودش یکی از نقاط درون داده ها باشد . به عبارت دیگر باید برای یک ویژگی یک خط برش پیدا کنیم که معیار را به بیشترین حد خود برساند و دوباره از میان ویژگی ها نیز یک ویژگی را پیدا کنیم که در بهترین خط برش خودش به بیشترین information gain در بین ویژگی ها برسد پس صرفاً یک حلقه به مجموعه برای پیدا کردن خط برش مناسب اضافه شده است و بقیه داستان همانند قبل است .

```

best_entropy = -np.inf
best_feature_index = None |
best_threshold = None
for feature_index in range(n_features):
    thresholds = np.unique(X[:, feature_index])
    for threshold in thresholds:
        left_indices = np.where(X[:, feature_index] <= threshold)[0]
        right_indices = np.where(X[:, feature_index] > threshold)[0]
        if len(left_indices) > 0 and len(right_indices) > 0:
            y_left = y[left_indices]
            y_right = y[right_indices]
            entropy_gain = self._entropy(y) - self._find_Remainder(y_left, y_right)
            if entropy_gain > best_entropy:
                best_entropy = entropy_gain
                best_feature_index = feature_index
                best_threshold = threshold
                best_left_indices = left_indices
                best_right_indices = right_indices

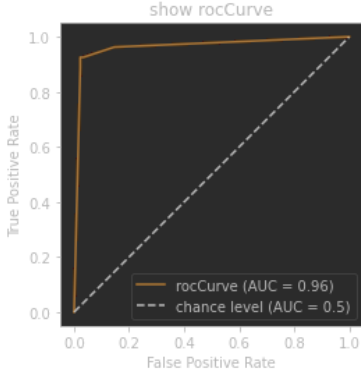
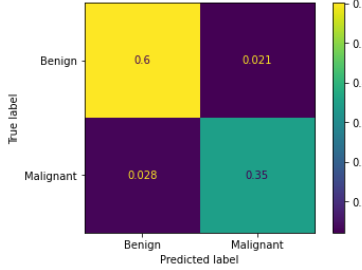
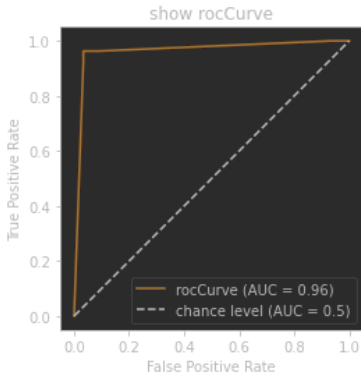
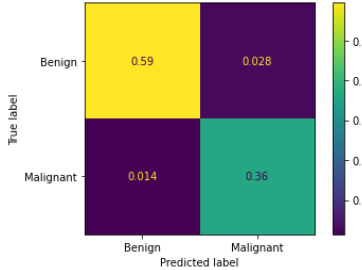
```

نتایج بدست آمده حاصل از دقت، عمق و تعداد برگ :

	accuracy	precision	recall	F1 Measure	depth	leaves
entropy	0.951	0.943	0.926	0.935	4	7
gini	0.958	0.929	0.963	0.945	4	8



برحسب معيار AUC , Confusion Matrix :

	AUC	Confusion Matrix
entropy	 <p>show rocCurve</p> <p>rocCurve (AUC = 0.96) chance level (AUC = 0.5)</p>	 <p>True label</p> <p>Benign</p> <p>Malignant</p> <p>Benign Malignant</p> <p>Predicted label</p>
gini	 <p>show rocCurve</p> <p>rocCurve (AUC = 0.96) chance level (AUC = 0.5)</p>	 <p>True label</p> <p>Benign</p> <p>Malignant</p> <p>Benign Malignant</p> <p>Predicted label</p>

پس از این کار می‌آیم و به کمک تابع PCA که متعلق به sklearn هست ابعاد را به ۱۰ بعد کاهش می‌دهیم و حاصل را بر روی مدل‌هایی که بهترین نتیجه را بر روی آنان گرفته‌ایم گزارش و بررسی می‌کنیم.

بهترین مدل در این لحظه :

- ۱۰ تایی (gini, entropy) و مدل‌های مبتنی بر یادگیری خط برش بوده است.
- برای کاهش ابعاد کافی است داده‌های آموزش را مانند شکل زیر به PCA بدهیم.
- سپس از مدل یادگرفته شده بر روی داده آموزش می‌آیم و داده‌های تست را نیز کاهش ابعاد می‌دهیم.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=10)
X_train = pca.fit_transform(X_train)
X_test = pca.transform(X_test)
```

بعد از انجام این کار و کاهش ابعاد دادگان یک بررسی آنان را می‌کنیم.

```
X_test=pd.DataFrame(data=X_test, # values|
                    columns=['pca_1','pca_2','pca_3','pca_4','pca_5','pca_6','pca_7','pca_8','pca_9','pca_10'])
                    the column names

X_test.head(12)
```

	pca_1	pca_2	pca_3	pca_4	pca_5	pca_6	pca_7	pca_8	pca_9	pca_10
0	-258.173198	-43.844389	5.911875	-1.658788	-2.028851	-0.336213	-0.076233	-0.232543	-0.317070	-0.018207
1	1097.182399	-119.046553	-5.105702	-1.248133	-5.272598	1.459829	-2.489805	-0.307865	-0.661243	0.355032
2	290.160214	-68.852754	-7.522614	-2.361256	-3.834684	-1.599647	0.102061	-0.303803	-0.255038	-0.115519
3	-404.067720	29.579215	-2.319113	-1.518608	-2.730723	-0.472944	-0.039248	0.411966	0.719972	0.061158
4	-487.736099	5.090197	9.308693	5.790706	-2.519784	-1.375474	1.321334	0.414634	0.344384	0.406773
5	1130.900538	19.498984	-22.478816	-24.809998	-2.566227	3.746954	-2.107761	-0.462067	0.432083	0.698759
6	1591.966150	-9.655505	-2.552226	-4.821493	-8.609276	4.152841	-3.949943	-0.249196	0.043038	0.341249
7	460.200494	69.363063	-7.752083	0.457082	-9.953069	0.733416	-0.235700	0.450455	-0.293691	-0.236975
8	-292.331738	21.636395	-11.321615	-1.587118	-4.746743	-0.551340	1.654820	-0.277184	-0.566024	0.008024
9	-162.199725	34.081189	-17.645396	2.074987	-3.323247	0.838778	-0.331089	-0.643228	0.079782	0.041573
10	-267.727408	27.079311	-7.764059	-4.706089	11.147849	-1.220002	0.661487	-0.481828	-0.089248	-0.118764
11	452.027379	-6.598479	0.932606	-1.349339	-2.630085	-0.629107	0.979561	0.268573	-0.425502	0.076025

همانطور که می‌بینید دادگان منفی نیز شده اند در کنار کاهش ابعاد، تغییر در مقادیر نیز داشته ایم.

از آنجا که به دلیل منفی شدن مقادیر ممکن است در گسسته سازی به صورت دستی دچار مشکل شویم، تابع گسسته ساز را به شکل زیر تغییر می‌دهیم.

```
def make_coloumn_not_catogorical_catogorical(name_coloumn_with_bais_range, data_set_feature, columns_name, number_of_cut):
    data_set_feature_copy = data_set_feature.copy()
    for column in columns_name:
        for index in data_set_feature.index:
            if data_set_feature.loc[index, column] != None and data_set_feature.loc[index, column] >= 0:
                for c in range(int(number_of_cut/2)):
                    if data_set_feature.loc[index, column] != None and (data_set_feature.loc[index, column] >=
name_coloumn_with_bais_range[column]*c) and \
(data_set_feature.loc[index, column] < name_coloumn_with_bais_range[column]*(c+1)) and c !=
int(number_of_cut/2)-1:
                        map = c
                    elif c == number_of_cut-1 and data_set_feature.loc[index, column] != None and (data_set_feature
.loc[index, column] >= name_coloumn_with_bais_range[column]*c):
```

```
map = c

if data_set_feature_copy[column][index] != None and map == int(number_of_cut/2) - 1:
    data_set_feature_copy[column][index] = f'{column} >= {map*name_coloumn_with_bais_range[column]:.3f}'
elif data_set_feature_copy[column][index] != None:
    data_set_feature_copy[column][index] = f'{(map+1)*name_coloumn_with_bais_range[column]:.3f} >
{column} >= {map*name_coloumn_with_bais_range[column]:.3f}'

elif data_set_feature.loc[index, column] != None and data_set_feature.loc[index, column] < 0:
    for c in range(int(number_of_cut/2)):
        if data_set_feature.loc[index, column] != None and (data_set_feature.loc[index, column] <=
name_coloumn_with_bais_range[column]*c*-1) and \
(data_set_feature.loc[index, column] > name_coloumn_with_bais_range[column]*(c+1)*-1) and c !=
int(number_of_cut/2)-1:
            map = c
        elif c == number_of_cut-1 and data_set_feature.loc[index, column] != None and (data_set_feature
.loc[index, column] >= name_coloumn_with_bais_range[column]*c*-1):
            map = c
```

```
map = c

if data_set_feature_copy[column][index] != None and map == int(number_of_cut/2) - 1:
    data_set_feature_copy[column][index] = f'{column} <= {-1*map*name_coloumn_with_bais_range[column]:
.3f}'
elif data_set_feature_copy[column][index] != None:
    data_set_feature_copy[column][index] = f'{-1*(map+1)*name_coloumn_with_bais_range[column]:.3f} <
{column} <= {-1*map*name_coloumn_with_bais_range[column]:.3f}'

return data_set_feature_copy
```

دادگان را بررسی می کنیم که آیا منفی هستند یا مثبت .

اگر دادگان مثبت بود، به اندازه نصف تعداد برشی که باید بزنیم را از قسمت مثبت همانند گذشته و به اندازه نصف دیگر برشی

باید بزنیم را از سمت منفی می زنیم .

برای مثال برای ۴ بارش خواهیم داشت :

در اینجا L طول بازه است .

$$1) \quad 0 \leq \text{feature} < 1 * L$$

$$2) \quad 1 * L < \text{feature}$$

$$3) \quad -1 * L < \text{feature} \leq 0$$

$$4) \quad \text{feature} \leq -1 * L$$

کاری که تابع بالا انجام می دهد تقسیم بندی به شکل زیر است .

See the caveats in the documentation: [https://pandas.pydata.org/pandas-docs/stable/user\\_guide/indexing](https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing)

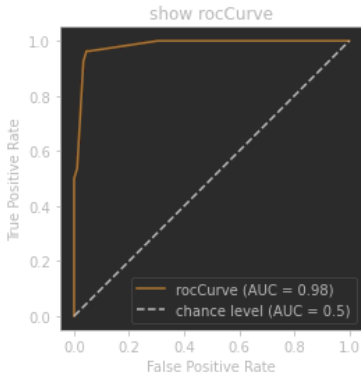
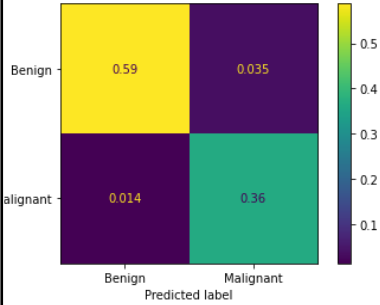
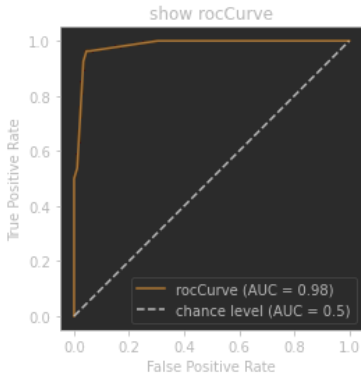
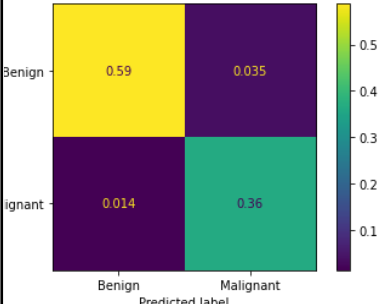
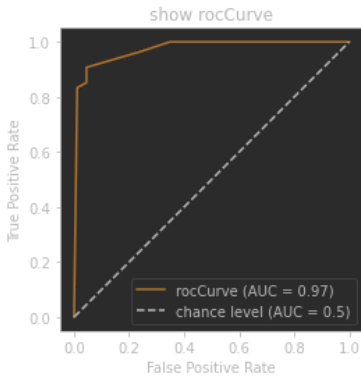
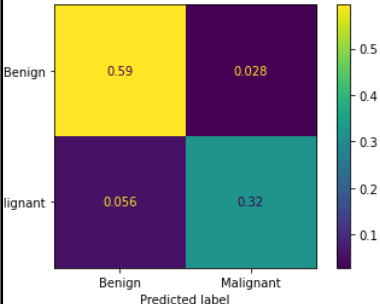
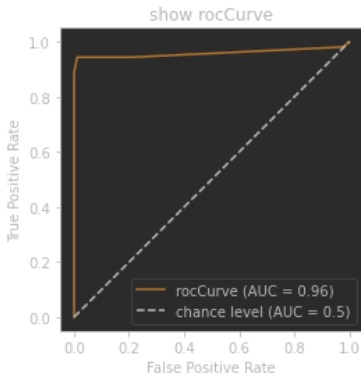
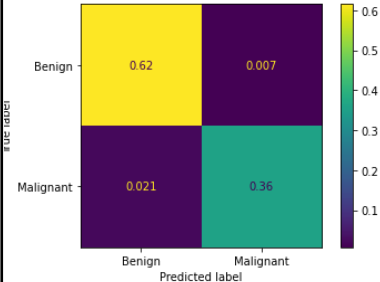
	pca_1	pca_2	pca_3	pca_4	pca_5	pca_6	pca_7	pca_8	pca_9	pca_10
0	-468.263 < pca_1 <= 0.000	121.448 > pca_2 >= 0.000	-40.912 < pca_3 <= 0.000	11.894 > pca_4 >= 5.947	-7.983 < pca_5 <= -3.991	-1.164 < pca_6 <= 0.000	-1.152 < pca_7 <= 0.000	-0.762 < pca_8 <= 0.000	0.400 > pca_9 >= 0.000	-0.350 < pca_10 <= 0.000
1	-468.263 < pca_1 <= 0.000	-121.448 < pca_2 <= 0.000	-40.912 < pca_3 <= 0.000	-11.894 < pca_4 <= -5.947	-3.991 < pca_5 <= 0.000	2.328 > pca_6 >= 1.164	1.152 > pca_7 >= 0.000	-0.762 < pca_8 <= 0.000	0.400 > pca_9 >= 0.000	-0.350 < pca_10 <= 0.000
2	-468.263 < pca_1 <= 0.000	121.448 > pca_2 >= 0.000	-40.912 < pca_3 <= 0.000	-5.947 < pca_4 <= 0.000	-3.991 < pca_5 <= 0.000	1.164 > pca_6 >= 0.000	-1.152 < pca_7 <= 0.000	0.762 > pca_8 >= 0.000	0.400 > pca_9 >= 0.000	0.350 > pca_10 >= 0.000
3	936.526 > pca_1 >= 468.263	-121.448 < pca_2 <= 0.000	-40.912 < pca_3 <= 0.000	-11.894 < pca_4 <= -5.947	7.983 > pca_5 >= 3.991	1.164 > pca_6 >= 0.000	-1.152 < pca_7 <= 0.000	-0.762 < pca_8 <= 0.000	0.400 > pca_9 >= 0.000	-0.350 < pca_10 <= 0.000
4	1404.789 > pca_1 >= 936.526	242.895 > pca_2 >= 121.448	40.912 > pca_3 >= 0.000	-5.947 < pca_4 <= 0.000	-3.991 < pca_5 <= 0.000	2.328 > pca_6 >= 1.164	-1.152 < pca_7 <= 0.000	0.762 > pca_8 >= 0.000	-0.400 < pca_9 <= 0.000	-0.350 < pca_10 <= 0.000

برای پیاده سازی درخت تصمیمی که خودش خط برش را یاد می گیرد اما مشکل خاصی نداریم و همان را به صورت پیش

فرض استفاده می کنیم .

حال بیایم معیارهای اندازه گیری را با یکدیگر مقایسه کنیم :

	accuracy	precision	recall	F1	depth	leaves
۱۰ تایی با entropy	0.951	0.912	0.963	0.937	4	20
۱۰ تایی با gini	0.951	0.912	0.963	0.937	4	20
خط مرتب ساز entropy	0.916	0.920	0.852	0.885	4	7
خط مرتب ساز gini	0.972	0.981	0.944	0.962	4	8

	Auc	confusion Matrix									
۱۰ تایی با entropy		 <table border="1"> <thead> <tr> <th></th> <th>Benign</th> <th>Malignant</th> </tr> </thead> <tbody> <tr> <th>Benign</th> <td>0.59</td> <td>0.035</td> </tr> <tr> <th>Malignant</th> <td>0.014</td> <td>0.36</td> </tr> </tbody> </table>		Benign	Malignant	Benign	0.59	0.035	Malignant	0.014	0.36
	Benign	Malignant									
Benign	0.59	0.035									
Malignant	0.014	0.36									
۱۰ تایی با gini		 <table border="1"> <thead> <tr> <th></th> <th>Benign</th> <th>Malignant</th> </tr> </thead> <tbody> <tr> <th>Benign</th> <td>0.59</td> <td>0.035</td> </tr> <tr> <th>Malignant</th> <td>0.014</td> <td>0.36</td> </tr> </tbody> </table>		Benign	Malignant	Benign	0.59	0.035	Malignant	0.014	0.36
	Benign	Malignant									
Benign	0.59	0.035									
Malignant	0.014	0.36									
خط مرتب ساز entropy		 <table border="1"> <thead> <tr> <th></th> <th>Benign</th> <th>Malignant</th> </tr> </thead> <tbody> <tr> <th>Benign</th> <td>0.59</td> <td>0.028</td> </tr> <tr> <th>Malignant</th> <td>0.056</td> <td>0.32</td> </tr> </tbody> </table>		Benign	Malignant	Benign	0.59	0.028	Malignant	0.056	0.32
	Benign	Malignant									
Benign	0.59	0.028									
Malignant	0.056	0.32									
خط مرتب ساز gini		 <table border="1"> <thead> <tr> <th></th> <th>Benign</th> <th>Malignant</th> </tr> </thead> <tbody> <tr> <th>Benign</th> <td>0.62</td> <td>0.007</td> </tr> <tr> <th>Malignant</th> <td>0.021</td> <td>0.36</td> </tr> </tbody> </table>		Benign	Malignant	Benign	0.62	0.007	Malignant	0.021	0.36
	Benign	Malignant									
Benign	0.62	0.007									
Malignant	0.021	0.36									

خوب همانطور که مشاهده می کنید در نهایت به کمک PCA و استفاده از خط مرتب ساز که توسط خود ماشین گرفته می شد و استفاده از معیار gini توانستیم بهترین دقت به میزان 0.97 درصد را بر روی دادگان تست داشته باشیم .  
به صورت معمول و با کمک توابع آماده sklearn مسئله را حل کنیم به دقت 0.93 خواهیم رسید که این نشان می دهد مدل ما به خوبی به بهبود یافته و پیاده سازی خوبی داشته است .

برای بررسی درستی و صحت تمامی آزمایش ها کدها به صورت impy و اجرا شده در اختیار شما است همچنین برای تمامی درخت های ساخته شده نیز درخت به صورت عکس نیز کشیده شده است که نگار هر کد که مربوط به بخشی از کار است می توانید درخت نهایی بدست آمده را نیز به صورت گرافیکی ببینید .

بسیار دوست داشتم که این عکس را نیز در گزارش نشان دهم منتها به دلیل بزرگی درخت ها و پایین کیفیت وضوح تصویر در صورت آوردن عکس کامل آن در صفحات مجبور شدم عکس ها را صرفا در نگار فایل کد بگذارم که علاقه مندان بتوانند درخت های تصمیم مختلف ساخته شده توسط کد من را ببینند .

خوب در ادامه گزارش برویم سراغ بخش سوم :

برای پیاده سازی یک تابع رگرسیون خطی به کمک درخت تصمیم باید درخت تصمیم ما چند تغییر در خود ببیند .

۱. تابع بدست آوردن purity به کمک entropy , gini تغییر کند و اینجا معیار دیگر به نام واریانس تعریف می کنیم و به دنبال آن هستیم که واریانس بین دادگان یک گروه کمینه شود .

۲. برای بدست آوردن حاصل هر برگ دیگری می توانیم از بیشترین فرکانس به عنوان ارزش برگ استفاده کنیم و باید برای آن جایگزین هایی انتخاب کنیم .

مورد اول به سادگی قابل حل است فقط کافی است که به جای مقدار entropy , gini ، واریانس بین مقادیر مختلف خروجی را پیدا کنیم پس خواهیم داشت :

```
def _variance(self, y):  
  
    if len(y) == 0:  
        return 0  
    return np.var(y)
```

و تمامی توابع داخلی دیگر براساس واریانس کار می کنند .

برای حل مشکل دیگر می توانیم تابعی به نام find value بنویسیم که به دو شکل بر حسب اینکه ما بخواهیم براساس کدام معیار مقدار value را حساب کند، مقدار را برای ما حساب کند ما در ابتدا فقط مقدار را بر حسب میانگین یا میانه داده های ورودی پیدا می کنیم .

```
def _find_value(self, numpy_array_y, father_value, X, map_feature_number):  
  
    if self.how_to_calculate_value == 'mean':  
        return np.mean(numpy_array_y)  
    elif self.how_to_calculate_value == 'median':  
        return np.median(numpy_array_y)  
    else:  
        raise
```

در ادامه برویم و نتایج را بر روی دیتاست شماره سه بررسی کنیم .

ابتدا مانند قبل برویم ببینیم با چه نوع دیتایی سرکار داریم .

ID	Price	Levy	Manufacturer	Model	Prod. year	Category	Leather interior	Fuel type	Engine volume	Mileage	Cylinders	Gear box type	Drive wheels	Doors	Wheel	Color	Airbags
45654403	13328	1399	LEXUS	RX 450	2010	Jeep	Yes	Hybrid	3.5	186005 km	6.0	Automatic	4x4	04-May	Left wheel	Silver	12
44731507	16621	1018	CHEVROLET	Equinox	2011	Jeep	No	Petrol	3	192000 km	6.0	Tiptronic	4x4	04-May	Left wheel	Black	8
457744198467	-	-	HONDA	FIT	2006	Hatchback	No	Petrol	1.3	200000 km	4.0	Variator	Front	04-May	Right-hand drive	Black	2
457691853607	862	862	FORD	Escape	2011	Jeep	Yes	Hybrid	2.5	168966 km	4.0	Automatic	4x4	04-May	Left wheel	White	0
45809263	11726	446	HONDA	FIT	2014	Hatchback	Yes	Petrol	1.3	91901 km	4.0	Automatic	Front	04-May	Left wheel	Silver	4
4580291239493891			HYUNDAI	Santa FE	2016	Jeep	Yes	Diesel	2	160931 km	4.0	Automatic	Front	04-May	Left wheel	White	4

با بررسی داده‌های شماره سه متوجه شدم که با دادگان بسیار بیشتر و البته کثیف‌تری نسبت به دادگان قبلی سرکار دارم . نیاز بود در بعضی از ستون‌ها همانند ستون Mileage اقدام به پیش پردازش کرده و آنان را تمیز کنم مثلاً در اینجا km را باید از میان دادگان پاک کنم یا برای مثال در ستون مربوط به Engine volume متوجه حضور کلمه Turbo در کنار یک سری داده عدد برای بعضی از مثال‌ها شدم که منظور آن بود که از نوع Turbo این نوع از Engine ها هستند پس مجبور شدم ستون دیگری اضافه کنم که در صورتی که Engine ما از نوع Turbo بود مقدار آن yes و در غیر این صورت No برگرداند. همچنین در بعضی از موارد با missing value دست به گریبان هستم که برای حل آن نیز اقدامی‌هایی کردم که در ادامه می‌بینیم و همچنین ویژگی‌های من در این سری از دادگان هم پیوسته است و هم گسسته ، پس نیاز است پیوسته‌ها را گسسته نمایم .

برای حل مشکل missing value از روش زیر استفاده کردم .

```
def find_replacing_for_null(X_train, list_of_categorical_column, list_of_continous_column):
    dic_replacing_for_null = {}
    for column in list_of_continous_column:
        print(column)
        dic_replacing_for_null[column] = find_mean_column(X_train.loc[:, column])

    for column in list_of_categorical_column:
        dic_replacing_for_null[column] = X_train[column].mode()[0]

    return dic_replacing_for_null
```



برای داده‌های پیوسته از میانگین و برای دادگان گسسته از داده‌هایی که بیشترین تکرار را داشت برای پرکردن جای خالی استفاده کردم .

البته این مقادیر بر حسب داده‌های train ساخته شده است لذا آن را درون یک دیکشنری ذخیره کردم که بتوانم برای پرکردن جای خالی در دادگان تست نیز از آنان استفاده کنم .

برای تبدیل داده‌های پیوسته به گسسته نیز همانند مسئله قبل ستون‌هایی که متعلق به داده‌های پیوسته بودند را گسسته کردم . در نهایت ستون دادگان من به شکل زیر درآمد و به شکلی قابل قبول برای یادگیری .

	Levy	Manufacturer	Model	Prod. year	Category	Leather interior	Fuel type	Engine volume	Mileage	Cylinders	Gear box type	Drive wheels	Doors	Wheel	Color
13330	> 2323.800 >= 0.000	MERCEDES-BENZ	ML 300	Prod. year >= 64.800	Jeep	Yes	Petrol	4.000 > Engine volume >= 0.000	429496729.400 > Mileage >= 0.000	Cylinders >= 12.000	Tiptronic	4x4	04-May	Left wheel	Black
7499	> 2323.800 >= 0.000	KIA	Sportage	Prod. year >= 64.800	Jeep	Yes	Diesel	4.000 > Engine volume >= 0.000	429496729.400 > Mileage >= 0.000	6.000 > Cylinders >= 3.000	Automatic	Front	04-May	Left wheel	Silver
	2323.800			Prod. year				4.000 > Engine	429496729.400	6.000 >			04-	Left	

برای اندازه‌گیری دقت و یا خوبی مدل درخت خود دو نوع خطا تعریف کردم به شکل زیر :

```
def lossfun(y_true,y_pred):
    loss = np.sum((y_true - y_pred)**2)
    return loss

def lossmean(y_true,y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
```

اولی مجموعه تفاضلات به توان دو است .

دومی جذر mean squared error بین مقدار اصلی و مقدار پیش بینی شده توسط درخت است .

خوب ابتدا درخت بر اساس میانگین بر روی دادگان آموزش ساختم .

```
list_of_feature_name = list(X_train_all_catogorical.columns.values)
map_feature_number = [i for i in range(len(list_of_feature_name))]

np_train_x = X_train_all_catogorical.to_numpy()
np_label_train = y_train.to_numpy()

machine_decision_tree_model = DecisionTreeRegression(list_of_feature_name,how_to_calculate_value='mean',max_depth=10)
machine_decision_tree_model.fit(np_train_x,np_label_train,map_feature_number)

machine_decision_tree_model.tree_show()
```

- و سپس دادگان تست را برحسب درخت ساخته شده، مقادیر خروجی آنان را حدس زدم.
- و سپس میزان تفاوت میان مقدار واقعی و پیش بینی شده را حساب کردم.

```
print(lossfun(y_true,y_pred_reg))
```

```
22707780937902.38
```

```
print(lossmean(y_true,y_pred_reg))
```

```
68709.18699811753
```

همانطور که مشاهده می‌فرمایید میزان تفاوت بالا است .

- همین کار را انجام دادم و اینبار مقدار را برحسب میانه حساب کردم .
- میزان تفاوت به شکل زیر بود :

```
print(lossfun(y_true,y_pred_reg))
```

```
1064942881875.25
```

```
print(lossmean(y_true,y_pred_reg))
```

```
14879.57817089285
```

- همانطور که مشاهده می‌کنید میانه نسبت به حالت میانگین بسیار بهتر عمل می‌کند ولی باز نیز خطا آن بسیار بالا است .
- باید برای تقریب سراغ روش دیگری رفت .
- در ادامه ما با روش دیگر مقدار میزان را تقریب می‌زنیم .

در این روش به جای آنکه مقدار را به شکل میانگین یا میانه از داده‌های آموزش بیرون بکشیم، میایم و یک تابع خطی بر حسب آنان می‌سازیم و مقدار خروجی را بدست می‌آوریم .  
اما چگونه ؟

برای ساخت یک تابع خطی یکی از نیازهای ما این است که مقادیر ما از جنس پیوسته باشند .  
در بین این نمونه‌ها ما شش عدد ویژگی پیوسته داشتیم که آنان را تبدیل به گسسته کردیم، اما برای رسیدن به یک تقریب بهتر نیاز است که این دادگان را به صورت پیوسته نیز داشته باشیم ، یعنی آنکه برای ساخت درخت آنان را به گسسته تبدیل کنیم ولی برای ساخت فانکشن خطی که مقدار پیش‌بینی کند از مقادیر پیوسته آنان برای ساخت این تابع استفاده کنیم .

به همین علت کلاسی ساختیم به اسم کلاس Approximation که یک ضابطه خطی میان این شش تا ویژگی پیوسته در هر جا پیدا می‌کند .

حل معادله خط در دستگاه جبری را همه بلدیم . به کمک np.linalg.solve ما می‌توانیم ضرایب یک معادله خطی را بیابیم که به کمک این ضرایب می‌توانیم یک معادله خطی بین چند متغیر بنویسیم .  
اما برای حل یک معادله خطی چند شرط داریم که :  
۱. تعداد مجهولات و تعداد معادلات برابر باشند.  
۲. وارون پذیر باشد که یعنی ماتریس دترمینان صفر نباشد .

گاهی ممکن است تعداد نمونه‌های موجود در برگ بیشتر از تعداد مجهولات باشد در این جا ما می‌آییم و به تعداد مورد نیاز شش تا را به صورت رندوم حساب می‌کنیم و تابع تقریب را می‌سازیم .  
گاهی تعداد نمونه کمتر از تعداد مجهولات است . در اینجا مجبور هستیم براساس تعداد نمونه‌هایی که داریم یک تابع خطی بسازیم و از در نظر گرفتن بقیه مجهولات صرف نظر کنیم برای مثال ما در مسئله فعلی شش تا مجهول داریم ولی در برگ چهار نمونه داریم . راهکار ما این است که به صورت رندوم چهار مجهول را انتخاب می‌کنیم و برحسب آن چهار نمونه یک ضریبی برای آنان در نظر می‌گیریم و آن دو ۲ مجهول که استفاده نکرده‌ایم را به کار نمی‌بریم برای تقریب تابع البته در دیکشنری dic of continuous use مشخص می‌کنیم از کدام ویژگی‌ها برای تقریب تابع استفاده کرده ایم .

```

class Aproximation:

    def __init__(self,y,X,list_of_continous_column):

        self.dic_of_continous_use = {}
        self.fit_function = self._fit(y,X,list_of_continous_column)
    def _fit(self,y,X,list_of_continous_column):

        if len(X) > len(list_of_continous_column) :

            rng = np.random.default_rng()

            size = len(list_of_continous_column)
            idx = rng.choice(len(X), size=size,replace=False)
            for i in range(len(X)):
                size = len(list_of_continous_column)
                idx = rng.choice(len(X), size=size,replace=False)
                X_selected = X[idx,:]

                Y_selected =y[idx]
                try :
                    res = np.linalg.solve(X_selected, Y_selected)
                    k = 0
                    for item in list_of_continous_column:
                        self.dic_of_continous_use[item] = k
                        k+=1
                return res

```

```

            except :

                X_selected = self._add_noise(X_selected)
                res = np.linalg.solve(X_selected, Y_selected)
                k = 0
                for item in list_of_continous_column:
                    self.dic_of_continous_use[item] = k
                    k+=1
                return res

        elif len(X) == len(list_of_continous_column) :
            k = 0
            for item in list_of_continous_column:
                self.dic_of_continous_use[item] = k
                k+=1
            try:
                return np.linalg.solve(X, y)
            except:
                x_selected = self._add_noise(X)
                return np.linalg.solve(x_selected, y)

        elif len(X) < len(list_of_continous_column):

```

```

        size = len(X)
        rng = np.random.default_rng()

        idx = rng.choice(len(list_of_continous_column), size=size,replace=False)
        ix_sort = np.sort(idx)
        for item in ix_sort :
            self.dic_of_continous_use[list_of_continous_column[item]] = item

        x_selected = X[:,ix_sort]
        try:
            return np.linalg.solve(x_selected, y)
        except:
            x_selected = self._add_noise(x_selected)
            return np.linalg.solve(x_selected, y)

```

گاهی وقت‌ها ماتریس وارون پذیر نیست پس نمی‌توان جواب نیز برای آن بدست آورد مجبوریم در این جا چندین بار به صورت تصادفی از نمونه‌های دیگر استفاده می‌کنیم برای تقریب تابع اما اگر تمام تیرها به سنگ خورد از تابع add noise استفاده می‌کنیم . در این جا به نمونه‌های یک مقدار رندوم بسیار کوچک اضافه می‌کنیم تا بتوانیم تابع تقریب را بدست آوریم .

```
def value(self,X):
    x_can_give = np.array([x[self.dic_of_continous_use[key]] for key in self.dic_of_continous_use])
    return np.dot(self.fit_function,x_can_give)

def _add_noise(self,X):
    return X + np.random.randn(X.shape[0], X.shape[1])*0.001
```

در نهایت ما به ازای هر برگ یک کلاس Approximation می‌سازیم که اطلاعات بالا را در خود دارد و هر وقت بخواهیم آن برگ می‌تواند مقداری برای ما حساب کند کافی است تابع value این کلاس را صدا کنیم . ویژگی‌هایی که بر حسب آن تابع تقریب را ساخته است از ورودی را ضرب داخلی در مجهولاتی که برای معادله خط کرده است (ضرایب ویژگی‌ها) می‌کند و یک مقدار برمی‌گرداند . حال ببینیم به کمک این نوع تقریب تابع خطا ما چگونه خواهد شد .

```
print(lossfun(y_true,y_pred_reg))

8.786359446650179e+39

print(lossmean(y_true,y_pred_reg))

1.351549464742052e+18
```

واقعا حیرت انگیز است .

انجام همین کار ساده خطا ما را هزار برابر کوچکتر از حالت قبل کرد .

پیاده سازی این دو حالت را می‌توانید در machin\_price\_by\_fit\_function و machin\_price\_by\_medium\_mean در قسمت machin\_price\_regrassion\_part3 مشاهده بفرمایید تمامی کدها و نتایج موجود است .

ممنون از توجه شما خدا نگه دار