Decision tree document

محمدصالح پژند 400521171

در این پروژه از سه روش Entropy , Information gain , Gini impurity استفاده شده است.

هر کدام در یک فایل جداگانه قرار دارند و پس از اجرا هر کدام یک فایل json برای نمایش درخت ایجاد می شود.

تقریبا ساختار هر سه الگوریتم یکسان است و فقط در چند تابع باهم تفاوت دارند.

ساختار کلی به این صورت است که ابتدا یک کلاس node برای هریک از تصمیم ها در درخت در نظر میگیریم. این کلاس دارای فرزند راست و چپ است که هر کدام یک node دیگر هستند. اگر این node یک برگ باشد مقدار value در ان ست میشود و اگر مقدار نداشته باشد برگ نیست.

یک کلاس دیگر برای توابع ایجاد میکنیم . برای ساخت درخت کافیست تابع fit را اجرا کنیم . بعد از ان تابع grow tree اجرا میشود . اگر نود شرایط برگ شدن را داشت برگ میشود و اگر نداشت بهترین ویژگی را پیدا میکند و در ان ویژگی اگر بیشتر از 2 حالت وجود داشت بهترین ترکیب را انتخاب میکند و دو نود جدید ایجاد میکند. این انتخاب ترکیب در هریک از الگوریتم ها متفاوت است.

همانطور که گفته شد این درخت تصمیم باینری است و ویژگی هایی که باینری نیستند گسسته در بخش Data Cleaning گسسته شده و در تابع best split به دو بخش تقسیم میشود.

در همین بخش چند ویژگی که تاثیری ندارند یا تاثیر کمی دارند حذف شده اند مانند age, gender ,id ,... . داده ها به صورت رندم انتخاب میشوند .

به طور خلاصه الگوریتم های تقسیم شاخه به این صورت است:

+ Entropy :

```
def _entropy(self, y):
    hist = np.bincount(y)
    ps = hist / len(y)
    return -np.sum([p * np.log(p) for p in ps if p > 0])
```

```
def _Entropy_calculate(self, y, X_column, threshold):
    # create children
    left_idxs, right_idxs = self._split(X_column, threshold)
    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    # calculate the weighted avg. entropy of children
    n = len(y)
    n_l, n_r = len(left_idxs), len(right_idxs)
    e_l, e_r = self._entropy(y[left_idxs]), self._entropy(y[right_idxs])
    child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

    return child_entropy
```

طبق فرمول انتروپی مقدار ان حساب میشود و به تابع split داده میشود تا بر اساس ان تقسیم کند.

+ Information gain : در این الگوریتم از انتروپی استفاده شده تا مقدار نهایی محاسبه شود. تفاوتش این است که برعکس الگوی انتروپی هر چه این مقدار بیشتر باشد ویژگی بهتر است.

```python
def _information_gain(self, y, X_column, threshold):
    # parent entropy
    parent_entropy = self._entropy(y)

    # create children
    left_idxs, right_idxs = self._split(X_column, threshold)

    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    # calculate the weighted avg. entropy of children
    n = len(y)
    n_l, n_r = len(left_idxs), len(right_idxs)
    e_l, e_r = self._entropy(y[left_idxs]), self._entropy(y[right_idxs])
    child_entropy = (n_l / n) * e_l + (n_r / n) * e_r

    # calculate the IG
    information_gain = parent_entropy - child_entropy
    return information_gain
```

: Gini impurity +

```python
def _Gini_imp(self, y, X_column, threshold):
    # create children
    left_idxs, right_idxs = self._split(X_column, threshold)
    if len(left_idxs) == 0 or len(right_idxs) == 0:
        return 0

    def gini_impurity(y):
        p = np.bincount(y) / len(y)
        return 1 - np.sum(p**2)

    left_gini = gini_impurity(y[left_idxs])
    right_gini = gini_impurity(y[right_idxs])
    weighted_gini = (
        left_gini * sum(left_idxs) + right_gini * sum(right_idxs)
    ) / len(y)

    return weighted_gini
```

این روش از انتروپی استفاده نمیکند و تفاوتش بیشتر است .

مقایسه:

به طور کلی با شرایط یکسان هر سه الگوریتم دقت بالای 93–94 درصد دارند (بالاترین دقت متوسط 94.8 درصد از الگوریتم gain است)اما GI مدت زمان اجرایی بیشتری دارد. دو الگوریتم انتروپی و gain زمان و دقت تقریبا مشابه دارند . دلیل این اتفاق هم فرمول انهاست . به این صورت که انتروپی به صورت مستقیم استفاده میشود اما gain در فرمول خودش از یک عدد نسبتا ثابت منهای انتروپی استفاده میکند با order برعکس . پس درخت در هر دو تقریبا مشابه است .

برای تست کردن overfit شدن همان داده train را میتوانیم به درخت بدهیم و میبینیم که دقت به دست آمده مشابه با دقت در بقیه داده های رندوم میباشد.