به نام خدا

گزارش تمرین عملی درخت تصمیم

اعضا:

فاطمه رفیعی ۹۸۲۳۰۳۹

ریحانه آهنی ۹۸۲۳۰۰۹

پاییز ۱۴۰۱

برای نوشتن الگوریتم درخت از یک کلاس استفاده می‌کنیم. در مقداردهی اولیه خود بردار ورودی X و اسم فیچرهای مورد بررسی را می‌دهیم. همچنین یک attribute دیگر به نام node هم اضافه می‌کنیم که نشان‌دهنده گره‌های موجود در درخت است که در ادامه قرار است ساخته شود.

```
class DecisionTree:
    def __init__(self, X, labels, features):
        self.X = X
        self.features = features
        self.labels = labels
        self.node = None
```

اولین کاری که قرار است انجام دهیم تعریف یک متد برای محاسبه information gain است. در این تابع ورودی‌ها x و اندیس فیچرها هستند. با فرض دانستن مقدار آنتروپی کل داده‌ها از قبل (توسط یک فانکشن دیگر) در حلقه for اول مقدار هر داده ورودی را به ازای آن فیچر خاص در یک لیست قرار می‌دهد سپس در حلقه‌ی بعدی تعداد مربوط به هر خروجی منحصر به فرد شمرده می‌شود و در یک لیست دیگر ذخیره می‌شود. در ادامه یک حلقه دیگر داریم که در آن داده‌هایی که مقادیر یکسانی از آن فیچر را دارند در یک گروه قرار می‌گیرند و در واقع گروه‌بندی می‌شوند. یعنی لیست feature_v_id خودش شامل چند لیست است که هر کدام مربوط به یک دسته از داده‌هاست که بر اساس آن فیچر خاص طبقه‌بندی شده اند.

در ادامه نوبت به محاسبه آنتروپی داده‌های گروه‌بندی شده می‌پردازیم. در نتیجه در for آخر به ازای تمامی دسته‌ها و تعدادشان در آن دسته آنتروپی محاسبه می‌شود و در یک لیست به نام entropy ذخیره می‌شود. نحوه محاسبه هم به این صورت است که نسبت تعداد داده‌های آن دسته بر تعداد کل داده‌ها در مقدار اصلی آنتروپی ضرب می‌شود.

در نهایت تفاضل آنتروپی کل و مجموع تک تک آنتروپی‌هایی که برای هر دسته حساب کردیم را به عنوان خروجی برمی‌گردانیم.

```
def inf_gain_cal(self, x, feature_idx):
    total_entropy = self.entropy_cal(x)
    x_features, feature_cnt = [], []

    for i in x:
        x_features.append(self.X[i][feature_idx])

    for j in list(set(x_features)):
        feature_cnt.append(x_features.count(j))

    feature_v_id = []
    for feature_item in list(set(x_features)):
        tmp = []
        for i, x_i in enumerate(x_features):
            if x_i == feature_item:
                tmp.append(x[i])
        feature_v_id.append(tmp)
    entropy = []
    for n_i, Class in zip(feature_cnt, feature_v_id):
        entropy.append(n_i/len(x)*self.entropy_cal(Class))

    return total_entropy - sum(entropy)
```

حال به سراغ آن تابعی می‌رویم که در بخش قبل اشاره شد، یعنی entropy_cal که برای محاسبه خود آنتروپی طبق فرمول آن عمل می‌کند. برای این کار فقط به بردار x ها نیاز داریم. در حلقه‌ی اول label ها در یک لیست قرار می‌گیرند (به همان ترتیب ورودی‌ها) و سپس در حلقه بعدی تعداد هر label به خصوص در مجموعه‌ی کل label ها شمرده می‌شود و در label_cnt قرار می‌گیرد. در حلقه بعدی به ازای هر label (که تعدادش را شمرده ایم) به وسیله فرمول مربوطه آنتروپی را محاسبه می‌نماییم، به این صورت که این مقدار (cnt) را بر تعداد کل داده‌ها تقسیم و سپس با علامت منفی در لگاریتم مبنای دو همان نسبت ضرب می‌کنیم. این مقادیر را در یک متغیر به نام sigma_entropy که همان مجموع آنتروپی‌هاست جمع می‌کنیم و در نهایت همان را به عنوان خروجی برمی‌گردانیم.

```
def entropy_cal(self, x):
    labels, label_cnt, sigma_entropy = [], [], 0

    for i in x:
        labels.append(self.labels[i])

    for j in list(set(self.labels)):
        label_cnt.append(labels.count(j))

    for cnt in label_cnt:
        if cnt != 0:
            sigma_entropy += -cnt/len(x)*math.log(cnt/len(x), 2)
    return sigma_entropy
```

در قسمت بعدی یک تابع دیگر تعریف می‌کنیم که وظیفه‌ی آن انتخاب بهترین feature برای هر گره است. به این صورت که کل داده‌های رسیده به آن گره و فیچرها را دریافت می‌کند. در یک حلقه اقدام به محاسبه information gain هر فیچر موجود می‌کند و آن را در features_entropy قرار می‌دهد. در مرحله بعد ماکزیمم مقدار موجود در این وکتور به عنوان بهترین فیچر انتخاب می‌شود ولی اسم آن برای ما اهمیتی ندارد و اندیس آن است که مهم است، در نتیجه index را به صورت خروجی تابع برمی‌گردانیم.

```
def opt_feature(self, x, features):
    features_entropy = [self.inf_gain_cal(x, feature_id) for feature_id in features]
    max_id = features[features_entropy.index(max(features_entropy))]
    return self.features[max_id], max_id
```

مرحله‌ی آخر fit کردن مدل به داده‌هاست ولی قبل از آن یک تابع بسیار مهم وجود دارد که وظیفه‌ی آن اجرا کردن عملیات تصمیم به صورت recursive است.

این تابع به اندیس x ها و فیچرها و همچنین خود گره‌ها نیاز دارد، چنانچه تا به حال هیچ گرهی ساخته نشده باشد با خط node = Node() این گره ایجاد می‌شود؛ که در واقع کلاسی از گره است چون می‌خواهیم همه‌ی اجزای یک گره به هم مرتبط باشند. در حلقه‌ی اول یک لیست موقت به نام labels_tmp را با مقادیر label های تمامی x ها پر می‌کنیم. سپس چک می‌کنیم که اگر همه داده‌های موجود در آن گره فقط یک label داشته باشند یا به عبارت دیگر آنتروپی‌شان صفر باشد آن‌گاه همان‌جا label زده می‌شود و عملیات دیگر ادامه نمی‌یابد. بعد از آن بررسی می‌شود که اگر هیچ فیچری باقی نمانده باشد باز هم عملیات تمام شود با این تفاوت که در این حالت دیگر آنتروپی صفر نیست بلکه بین تمامی داده‌های موجود در آن گره رای‌گیری انجام می‌شود و هر lable ای که تعداد بیشتری از داده‌ها را شامل می‌شد به عنوان نتیجه‌ی نهایی انتخاب می‌شود. اگر هیچ یک از این دو حالت رخ نداده بود نوبت این است که بهترین فیچر برای آن گره خاص با متد opt_feature حساب شود. دقت داریم که مقدار این گره برابر با خروجی همین تابع است.

```
def rec(self, x_idx, feature_idx, node):
    if not node:
        node = Node()
    labels_tmp = []
    for i in x_idx:
        labels_tmp.append(self.labels[i])

    if len(set(labels_tmp)) == 1:
        node.value = self.labels[x_idx[0]]
        return node

    if len(feature_idx) == 0:
        node.value = max(labels_tmp, key=labels_tmp.count) # voting
        return node

    best_feature_name, best_feature_idx = self.opt_feature(x_idx, feature_idx)
    node.value = best_feature_name
    node.childs = []
```

در ادامه‌ی این تابع یک حلقه داریم که در آن مقدار هر داده‌ی ورودی به ازای فیچر منتخب در یک یک لیست به نام unique_values ذخیره می‌شود که بعداً با تبدیل به set و دوباره list مقادیر تکراری‌اش حذف می‌شوند. حال به ازای هر عنصر در این لیست، یک گره دیگر داریم که به عنوان گره فرزند برای گره فعلی شناخته می‌شود و مقدارهای آن‌ها هم همان عناصر unique_values هستند.

سپس به ازای هر x بررسی می‌کنیم که آیا آن داده مقدارش برابر با مقدار مربوط به آن گره است یا خیر و در صورتی که این طور بود مقدار آن را در یک لیست به نام child_idx اضافه می‌کنیم.

اگر به یک گره هیچ مقداری assign نشده بود (یعنی در آن گره خاص هیچ داده‌ای وجود نداشت که مقدار خاصی از بهترین فیچر را داشته باشد) آنگاه به لیست unique_values نگاه می‌کنیم و آن مقدار یا لیبیلی که بیشتر تعداد دفعات تکرار را داشته باشد را به عنوان گره بعدی انتخاب می‌نماییم. در غیر این صورت چک می‌کنیم که حتماً آن فیچری که به عنوان بهترین فیچر در آن گره معرفی کرده ایم قبلاً در آن شاخه استفاده نشده باشد، چرا که در الگوریتم ID3 نباید از هر فیچر بیش از یکبار در هر شاخه استفاده کرد. اگر همه چیز درست بود در این جاست که تابع را بازگشتی می‌کنیم، یعنی دوباره خودش را صدا می‌زنیم و به عنوان ورودی child_idx را برای x_idx و child.next را برای node به آن می‌دهیم ولی feature_idx هنوز همان است با این تفاوت که با متد pop اندیس آن فیچری که به عنوان بهترین انتخاب شده بود را حذف می‌کنیم.

```
unique_values = []
for x in x_idx:
    unique_values.append(self.X[x][best_feature_idx])

unique_values = list(set(unique_values))

for value in unique_values:
    child = Node()
    child.value = value
    node.childs.append(child)

    child_idx = []
    for i in x_idx:
        if self.X[i][best_feature_idx] == value:
            child_idx.append(i)

    if not child_idx:
        child.next = max(unique_values, key=unique_values.count)
    else:
        if feature_idx and best_feature_idx in feature_idx:
            feature_idx.pop(feature_idx.index(best_feature_idx))
        child.next = self.rec(child_idx, feature_idx, child.next)
return node
```

در این مرحله قسمت اصلی کار به اتمام رسیده و فقط نوبت فیت کردن و تخمین صحت مدل است. در تابع زیر تعداد داده‌ها و فیچرها را در دو متغیر به نام‌های x_idx و features_idx ذخیره می‌کنیم و با این دو تابع recursive را فراخوانی می‌کنیم. در نهایت خروجی این تابع بازگشتی به صورت کلاسی از Node است که طبیعتاً باید آن را در node attribute قرار دهیم.

```
def fit(self):
    x_idx = list(range(len(self.X)))
    features_idx = list(range(len(self.features)))
    self.node = self.rec(x_idx, features_idx, self.node)
```

بخش زیر برای پیش‌بینی عملکرد درخت برای داده‌های ماست. ابتدا یک لیست به نام result می‌سازیم که قرار است مقادیر نهایی پیش‌بینی را در آن قرار دهیم و چون می‌خواستیم با استفاده از اندیس‌ها مقداردهی را انجام دهیم این لیست را با مقادیر None اینیشیلایز کردیم. در یک حلقه‌ی for به ازای هر x موجود در مجموعه داده‌های تست کار را شروع می‌کنیم. ابتدا یک متغیر به نام nodes از deque می‌سازیم که بتوان حذف مقادیرش را از سمت چپ انجام داد. سپس یک حلقه while داریم که تنها زمانی می‌توانیم وارد آن شویم که حتماً node دارای یک مقدار باشد و به عبارتی تهی نباشد چون قرار است روی nodes ادامه‌ی کار صورت بگیرد. در همان ابتدا اولین گره را (کلاس گره که شامل مقدار فعلی و بعدی و همچنین کلاس گره فرزند) را در متغیر node قرار می‌دهیم. به ازای تک‌تک فیچرها بررسی می‌کنیم که اگر مقدار آن گره با آن فیچر برابر بود نام آن را در یک متغیر به نام current_feature قرار دهد. حالا اگر آن گره فرزند هم داشته باشد یا به عبارتی آخرین گره (برگ) نباشد به ازای تمام child های موجود در آن گره بعدی بررسی می‌کنیم که آیا مقداری که آن x خاص در آن فیچر خاص دارد با مقدار موجود در این گره برابر است یا خیر، اگر برابر بود مقدار گره فرزند را در یک متغیر به نام last_child ذخیره می‌کنیم و بلافاصله گره‌های بعدی را به ابجکت nodes اضافه می‌کنیم که عملیات ادامه پیدا کند. اما اگر گره مورد نظر ما دارای هیچ گره‌ای بعد از خود نبود یعنی به برگ رسیده بودیم، این جاست که result را برای آن اندیس خاص با مقدار last_child مقداردهی می‌کنیم.

```
def predict(self, X_test):
    result = [None for _ in range(len(X_test))]
    for i, x_i in enumerate(X_test):
        nodes = deque()
        nodes.append(self.node)
        current_feature = None
        last_child = None
        while len(nodes) > 0:
            node = nodes.popleft()
            for feature in self.features:
                if node.value == feature:
                    current_feature = feature
            if node.childs:
                for child in node.childs:
                    if x_i[self.features.index(current_feature)] == child.value:
                        last_child = child.next.value
                        nodes.append(child.next)
            else:
                result[i] = last_child
                break
    return result
```

به عنوان آخرین متد در تابع زیر قرار است نتیجه صحت را محاسبه نماییم. به این صورت که مقادیر x ها و لیبل‌ها را به شکل ورودی دریافت می‌کند. در ابتدا یک متغیر به نام misclassified تعریف می‌کنیم که قرار است تعداد دفعاتی که مدل اشتباه پیش‌بینی کرده است را در آن قرار دهیم. با استفاده از متد predict (که در صفحه قبل توضیح داده شد) y_pred که تخمین مدل است را ایجاد می‌کنیم سپس در یک حلقه مقادیر اصلی را با آن‌ها مقایسه می‌نماییم و در صورتی که با یکدیگر برابر نباشند به misclassified یک واحد اضافه می‌کنیم و در آخر هم درصدی از درستی را به عنوان خروجی برمی‌گردانیم.

```
def accuracy(self, X_test, y_test):
    misclassified = 0
    y_pred = self.predict(X_test)
    for h in range(len(y_pred)):
        if y_pred[h] != y_test[h]:
            misclassified += 1

    return (len(y_pred) - misclassified) / len(y_pred)
```

از این قسمت به بعد فقط لود کردن دیتاست و اعمال کردن مدل است.

```
df = pd.DataFrame(pd.read_csv("nursery.csv"))
df
```

	parents	has_nurs	form	children	housing	finance	social	health	final evaluation
0	usual	proper	complete	1	convenient	convenient	nonprob	recommended	recommend
1	usual	proper	complete	1	convenient	convenient	nonprob	priority	priority
2	usual	proper	complete	1	convenient	convenient	nonprob	not_recom	not_recom
3	usual	proper	complete	1	convenient	convenient	slightly_prob	recommended	recommend
4	usual	proper	complete	1	convenient	convenient	slightly_prob	priority	priority

همانطور که مشخص است تارگت ما در این‌جا ۵ مقدار دارد که تعداد یکی از لیبل‌ها بسیار کم است، در نتیجه می‌توان از آن صرف نظر نمود. در نتیجه می‌توانیم این دو سمپل را به شکل زیر حذف نماییم.

```
1  df.value_counts('final evaluation')
```

```
final evaluation
not_recom     4320
priority      4266
spec_prior    4044
very_recom     328
recommend        2
dtype: int64
```

```
df.drop(df[(df.final_evaluation == 'recommend')].index)
```

در نهایت نیز با جداسازی داده‌های train و test و ایجاد یک instance از مدل DecisionTree به نام dt، فیت می‌کنیم و بعد درصد درستی را مشاهده می‌کنیم.

```
X = np.array(df.drop('final_evaluation', axis=1).copy())
y = np.array(df['final_evaluation'].copy())
# using the train test split function
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=18, test_size=0.2)

features = ['parents', 'has_nurs', 'form', 'children', 'housing', 'finance', 'social', 'health']
dt = DecisionTree(X_train, y_train, features)
```

```
dt.fit()
dt.accuracy(X_test, y_test)
```

```
0.8113425925925926
```

در بخش بعدی، از ما خواسته شده است تا داده ها را یکبار ۵۰ درصد و یکبار ۷۵ درصد بصورت تصادفی انتخاب کنیم و درخت را با آن آموزش دهیم. (یکبار با عمق ۶ و یکبار با عمق ۸) و سپس یکبار با انتروپی و یک بار با ضریب جینی و در نهایت صحت طبقه بندی داده ها تست کنیم.

```
class DecisionTree:
    def __init__(self, X, labels, features, max_depth=8, measure='entropy'):
        self.X = X
        self.features = features
        self.labels = labels
        self.node = None
        self.measure = measure
        self.max_depth = max_depth
        self.depth = 0
```

ابتدا باید به درخت قابلیت حداکثر عمق را اضافه کنیم، برای اینکار در تابع init ورودی به نام max_depth دریافت می کنیم، سپس در تابع rec که محلی که رای گیری برای انتهای feature ها انجام میشود، بررسی میکنیم depth فعلی بزرگتر از max_depth است یا خیر، و در این صورت بین feature ها رای گیری می شود.

```
    def rec(self, x_idx, feature_idx, node):
        self.depth += 1

        if not node:
            node = Node()
        labels_tmp = []
        for i in x_idx:
            labels_tmp.append(self.labels[i])

        if len(set(labels_tmp)) == 1:
            node.value = self.labels[x_idx[0]]
            return node

        if len(feature_idx) == 0 or self.max_depth < self.depth:
            node.value = max(labels_tmp, key=labels_tmp.count) # voting
            return node
```

برای افزودن قابلیت انتخاب میان انتروپی و ضریب جینی در تابع init ورودی جدیدی به نام measure اضافه میکنیم. سپس تابعی جدید برای محاسبه ضریب gini اضافه می کنیم:

```
    def gini_cal(self, x):
        labels, label_cnt, sigma_gini = [], [], 0

        for i in x:
            labels.append(self.labels[i])

        for j in list(set(self.labels)):
            label_cnt.append(labels.count(j))

        for cnt in label_cnt:
            if cnt != 0:
                sigma_gini += (cnt/len(x)) ** 2
        return 1 - sigma_gini
```

$$Gini = 1 - \sum_{i=1}^{C} (p_i)^2$$

سپس در تابع inf_gain_cal از تابع مورد انتخاب استفاده می کنیم:

```
    def inf_gain_cal(self, x, feature_idx):
        if self.measure == 'entropy':
            total_entropy = self.entropy_cal(x)
        elif self.measure == 'gini':
            total_entropy = self.gini_cal(x)

        x_features, feature_cnt = [], []

        for i in x:
            x_features.append(self.X[i][feature_idx])

        for j in list(set(x_features)):
            feature_cnt.append(x_features.count(j))

        feature_v_id = []
        for feature_item in list(set(x_features)):
            tmp = []
            for i, x_i in enumerate(x_features):
                if x_i == feature_item:
                    tmp.append(x[i])
            feature_v_id.append(tmp)
        entropy = []
        for n_i, Class in zip(feature_cnt, feature_v_id):
            entropy.append(n_i/len(x)*self.entropy_cal(Class))

        return total_entropy - sum(entropy)
```

سپس برای اجرای تمام حالت های خواسته شده، از کد زیر استفاده می کنیم:

```
cases = [
    {'measure': 'entropy', 'max_depth': 6, 'test': 0.5},
    {'measure': 'entropy', 'max_depth': 8, 'test': 0.5},
    {'measure': 'entropy', 'max_depth': 6, 'test': 0.25},
    {'measure': 'entropy', 'max_depth': 8, 'test': 0.25},
    {'measure': 'gini', 'max_depth': 6, 'test': 0.5},
    {'measure': 'gini', 'max_depth': 8, 'test': 0.5},
    {'measure': 'gini', 'max_depth': 6, 'test': 0.25},
    {'measure': 'gini', 'max_depth': 8, 'test': 0.25},
]
```

```
for case in cases:
    print(f"measure: {case['measure']}, test: {case['test']}, max_depth: {case['max_depth']}")
    # using the train test split function
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=3, test_size=case['test'])

    dt = DecisionTree(X_train, y_train, features, max_depth=case['max_depth'], measure=case['measure'])
    dt.fit()
    print(dt.accuracy(X_test, y_test))
```

نتایج به شکل زیر است:

```
measure: entropy, test: 0.5, max_depth: 6
0.7709876543209877

measure: entropy, test: 0.5, max_depth: 8
0.7708333333333334

measure: entropy, test: 0.25, max depth: 6
0.769753086419753

measure: entropy, test: 0.25, max_depth: 8
0.7700617283950617

measure: gini, test: 0.5, max_depth: 6
0.7709876543209877

measure: gini, test: 0.5, max_depth: 8
0.7708333333333334

measure: gini, test: 0.25, max_depth: 6
0.769753086419753

measure: gini, test: 0.25, max depth: 8
0.7700617283950617
```