classification-toy-example

March 11, 2024

1 Lab 3: Introducing Classification

Objectives: - To gain hands-on experience classifying small dataset - To implement concepts related to Decision Tree classifier (i.e. Entropy, Information Gain), along with the Decision Tree algorithm

[2]:
```
import pandas as pd
import math

# Read the data
df = pd.read_csv('../store/toy_data.csv')
df
```

[2]:

	age	income	student	credit rating	buys computer
0	<=30	high	no	fair	no
1	<=30	high	no	excellent	no
2	31-40	high	no	fair	yes
3	>40	medium	no	fair	yes
4	>40	low	yes	fair	yes
5	>40	low	yes	excellent	no
6	31-40	low	yes	excellent	yes
7	<=30	medium	no	fair	no
8	<=30	low	yes	fair	yes
9	>40	medium	yes	fair	yes
10	<=30	medium	yes	excellent	yes
11	31-40	medium	no	excellent	yes
12	31-40	high	yes	fair	yes
13	>40	medium	no	excellent	no

[3]:
```
print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 14 entries, 0 to 13
Data columns (total 5 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             14 non-null     object
 1   income          14 non-null     object
 2   student         14 non-null     object
```

```
 3   credit rating  14 non-null     object
 4   buys computer  14 non-null     object
dtypes: object(5)
memory usage: 692.0+ bytes
None
```

2 Solution

1. Find *Probability* of target variable (buys computer)

[8]:
```python
# Probability of target variable
n = df.shape[0]
p_yes = nStudent = df[df['buys computer'] == 'yes'].shape[0]
p_no = n - p_yes

print("Yes: ", p_yes, "No: ", p_no)
```

```
Yes:  9 No:  5
```

2. Find *Entropy* of target variable (buys computer)

[16]:
```python
entropy_t = - (p_yes/n) * math.log2(p_yes/n) - (p_no/n) * math.log2(p_no/n)
print("Entropy of target variable (buys computer): ", entropy_t)
```

```
Entropy of target variable (buys computer):  0.9402859586706311
```

2.0.1 Question 0: What is Gain Split of age

[10]:
```python
target_col = "age"

partitions = df[target_col].value_counts().to_dict()
bought_com_count = df[df['buys computer'] == 'yes'].groupby(target_col).size().
 ↪to_dict()


print("Partitions: ",partitions)

#Find P(student|t) (array)
probArr = {}
for key in partitions:
    probArr[key] = { "buy": bought_com_count[key]/partitions[key], "not_buy": 1␣
 ↪- (bought_com_count[key]/partitions[key])}

print("Probability of each partition: ",probArr)

#Find entropy of student
ent = {}
```

```python
for key in probArr:
    p_buy = probArr[key]["buy"]
    p_not_buy = probArr[key]["not_buy"]

    # Calculate entropy
    ent_buy = -p_buy * math.log2(p_buy) if p_buy > 0 else 0
    ent_not_buy = -p_not_buy * math.log2(p_not_buy) if p_not_buy > 0 else 0
    ent[key] = (partitions[key]/n) * (ent_buy + ent_not_buy)

sumEnt = sum(ent.values())

print("Sum of Entropy: ",sumEnt)
print("Entropy of target variable: ", entropy_t)
print("Gain Split: ", entropy_t - sumEnt)
```

```
Partitions:  {'<=30': 5, '>40': 5, '31-40': 4}
Probability of each partition:  {'<=30': {'buy': 0.4, 'not_buy': 0.6}, '>40':
{'buy': 0.6, 'not_buy': 0.4}, '31-40': {'buy': 1.0, 'not_buy': 0.0}}
Sum of Entropy:  0.6935361388961918
Gain Split:  0.24674981977443933
```

2.0.2 Question 1: What is Gain Split of income

[12]:
```python
target_col = "income"

partitions = df[target_col].value_counts().to_dict()
bought_com_count = df[df['buys computer'] == 'yes'].groupby(target_col).size().
 ↪to_dict()



print("Partitions: ",partitions)

#Find P(student|t) (array)
probArr = {}
for key in partitions:
    probArr[key] = { "buy": bought_com_count[key]/partitions[key], "not_buy": 1␣
 ↪- (bought_com_count[key]/partitions[key])}

print("Probability of each partition: ",probArr)

#Find entropy of student
ent = {}

for key in probArr:
    p_buy = probArr[key]["buy"]
    p_not_buy = probArr[key]["not_buy"]
```

```
    # Calculate entropy
    ent_buy = -p_buy * math.log2(p_buy) if p_buy > 0 else 0
    ent_not_buy = -p_not_buy * math.log2(p_not_buy) if p_not_buy > 0 else 0
    ent[key] = (partitions[key]/n) * (ent_buy + ent_not_buy)

sumEnt = sum(ent.values())

print("Sum of Entropy: ",sumEnt)
print("Entropy of target variable: ", entropy_t)
print("Gain Split: ", entropy_t - sumEnt)
```

```
Partitions:  {'medium': 6, 'high': 4, 'low': 4}
Probability of each partition:  {'medium': {'buy': 0.6666666666666666,
'not_buy': 0.33333333333333337}, 'high': {'buy': 0.5, 'not_buy': 0.5}, 'low':
{'buy': 0.75, 'not_buy': 0.25}}
Sum of Entropy:  0.9110633930116763
Entropy of target variable:  0.9402859586706311
Gain Split:  0.02922256565895487
```

2.0.3 Question 2: What is Gain Split of Student

[13]:
```
target_col = "student"

partitions = df[target_col].value_counts().to_dict()
bought_com_count = df[df['buys computer'] == 'yes'].groupby(target_col).size().
 ↪to_dict()



print("Partitions: ",partitions)

#Find P(student|t) (array)
probArr = {}
for key in partitions:
    probArr[key] = { "buy": bought_com_count[key]/partitions[key], "not_buy": 1␣
 ↪- (bought_com_count[key]/partitions[key])}

print("Probability of each partition: ",probArr)

#Find entropy of student
ent = {}

for key in probArr:
    p_buy = probArr[key]["buy"]
    p_not_buy = probArr[key]["not_buy"]

    # Calculate entropy
    ent_buy = -p_buy * math.log2(p_buy) if p_buy > 0 else 0
```

```
    ent_not_buy = -p_not_buy * math.log2(p_not_buy) if p_not_buy > 0 else 0
    ent[key] = (partitions[key]/n) * (ent_buy + ent_not_buy)

sumEnt = sum(ent.values())

print("Sum of Entropy: ",sumEnt)
print("Entropy of target variable: ", entropy_t)
print("Gain Split: ", entropy_t - sumEnt)
```

```
Partitions:  {'no': 7, 'yes': 7}
Probability of each partition:  {'no': {'buy': 0.42857142857142855, 'not_buy':
0.5714285714285714}, 'yes': {'buy': 0.8571428571428571, 'not_buy':
0.1428571428571429}}
Sum of Entropy:  0.7884504573082896
Entropy of target variable:  0.9402859586706311
Gain Split:  0.15183550136234159
```

2.0.4 Question 3: What is Gain Split of credit rating

[15]:
```
target_col = "credit rating"

partitions = df[target_col].value_counts().to_dict()
bought_com_count = df[df['buys computer'] == 'yes'].groupby(target_col).size().
 ↪to_dict()



print("Partitions: ",partitions)

#Find P(student|t) (array)
probArr = {}
for key in partitions:
    probArr[key] = { "buy": bought_com_count[key]/partitions[key], "not_buy": 1␣
 ↪- (bought_com_count[key]/partitions[key])}

print("Probability of each partition: ",probArr)

#Find entropy of student
ent = {}

for key in probArr:
    p_buy = probArr[key]["buy"]
    p_not_buy = probArr[key]["not_buy"]

    # Calculate entropy
    ent_buy = -p_buy * math.log2(p_buy) if p_buy > 0 else 0
    ent_not_buy = -p_not_buy * math.log2(p_not_buy) if p_not_buy > 0 else 0
    ent[key] = (partitions[key]/n) * (ent_buy + ent_not_buy)
```

```
sumEnt = sum(ent.values())

print("Sum of Entropy: ",sumEnt)
print("Entropy of target variable: ", entropy_t)
print("Gain Split: ", entropy_t - sumEnt)
```

```
Partitions:  {'fair': 8, 'excellent': 6}
Probability of each partition:  {'fair': {'buy': 0.75, 'not_buy': 0.25},
'excellent': {'buy': 0.5, 'not_buy': 0.5}}
Sum of Entropy:  0.8921589282623617
Entropy of target variable:  0.9402859586706311
Gain Split:  0.04812703040826949
```

[21]:
```
# Decision Tree
from collections import Counter

def entropy(y):
    hist = np.bincount(y)
    ps = hist / len(y)
    return -np.sum([p * np.log2(p) for p in ps if p > 0])

class Node:
    def __init__(
        self, feature=None, threshold=None, left=None, right=None, *, value=None
    ):
        self.feature = feature
        self.threshold = threshold
        self.left = left
        self.right = right
        self.value = value

    def is_leaf_node(self):
        return self.value is not None

class DecisionTree:
    def __init__(self, min_samples_split=2, max_depth=100, n_feats=None):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_feats = n_feats
        self.root = None

    def fit(self, X, y):
        self.n_feats = X.shape[1] if not self.n_feats else min(self.n_feats, X.
 ↪shape[1])
        self.root = self._grow_tree(X, y)
```

```python
    def predict(self, X):
        return np.array([self._traverse_tree(x, self.root) for x in X])

    def _grow_tree(self, X, y, depth=0):
        n_samples, n_features = X.shape
        n_labels = len(np.unique(y))

        if (
            depth >= self.max_depth
            or n_labels == 1
            or n_samples < self.min_samples_split
        ):
            leaf_value = self._most_common_label(y)
            return Node(value=leaf_value)

        feat_idxs = np.random.choice(n_features, self.n_feats, replace=False)

        best_feat, best_thresh = self._best_criteria(X, y, feat_idxs)

        left_idxs, right_idxs = self._split(X[:, best_feat], best_thresh)
        left = self._grow_tree(X[left_idxs, :], y[left_idxs], depth + 1)
        right = self._grow_tree(X[right_idxs, :], y[right_idxs], depth + 1)
        return Node(best_feat, best_thresh, left, right)

    def _best_criteria(self, X, y, feat_idxs):
        best_gain = -1
        split_idx, split_thresh = None, None
        for feat_idx in feat_idxs:
            X_column = X[:, feat_idx]
            thresholds = np.unique(X_column)
            for threshold in thresholds:
                gain = self._information_gain(y, X_column, threshold)
                if gain > best_gain:
                    best_gain = gain
                    split_idx = feat_idx
                    split_thresh = threshold
        return split_idx, split_thresh

    def _information_gain(self, y, X_column, split_thresh):
        parent_entropy = entropy(y)
        left_idxs, right_idxs = self._split(X_column, split_thresh)
        if len(left_idxs) == 0 or len(right_idxs) == 0:
            return 0

        n = len(y)
        n_l, n_r = len(left_idxs), len(right_idxs)
        e_l, e_r = entropy(y[left_idxs]), entropy(y[right_idxs])
```

```python
        child_entropy = (n_l / n) * e_l + (n_r / n) * e_r
        ig = parent_entropy - child_entropy
        return ig

    def _split(self, X_column, split_thresh):
        left_idxs = np.argwhere(X_column <= split_thresh).flatten()
        right_idxs = np.argwhere(X_column > split_thresh).flatten()
        return left_idxs, right_idxs

    def _traverse_tree(self, x, node):
        if node.is_leaf_node():
            return node.value

        if x[node.feature] <= node.threshold:
            return self._traverse_tree(x, node.left)
        return self._traverse_tree(x, node.right)

    def _most_common_label(self, y):
        counter = Counter(y)
        most_common = counter.most_common(1)[0][0]
        return most_common

mapping_buys_computer = {'yes': 1, 'no': 0}
mapping_income = {'high': 2, 'medium': 1, 'low': 0}
mapping_credit_rating = {'fair': 1, 'excellent': 0}
mapping_age = {'<=30': 0, '31-40': 1, '>40': 2}

df['buys computer'] = df['buys computer'].map(mapping_buys_computer)
df['income'] = df['income'].map(mapping_income)
df['credit rating'] = df['credit rating'].map(mapping_credit_rating)
df['age'] = df['age'].map(mapping_age)
df['student'] = df['student'].map(mapping_buys_computer)
print(df)
print()

X_train = df.drop('buys computer', axis=1).values
y_train = df['buys computer'].values


tree = DecisionTree(min_samples_split=2, max_depth=4, n_feats=None)
tree.fit(X_train, y_train)


X_test = np.array([
    [0, 1, 1, 1],
    [1, 0, 0, 0],
    [2, 0, 0, 1],
```

```
    [0, 0, 1, 0],
    [0, 1, 0, 1]
])


predictions = tree.predict(X_test)
prediction_list = []

print("Predictions:")
for i in predictions:
    if i == 1:
        prediction_list.append('buy')
    else:
        prediction_list.append('not buy')
print(prediction_list)
```

```
    age  income  student  credit rating  buys computer
0     0       2        0              1              0
1     0       2        0              0              0
2     1       2        0              1              1
3     2       1        0              1              1
4     2       0        1              1              1
5     2       0        1              0              0
6     1       0        1              0              1
7     0       1        0              1              0
8     0       0        1              1              1
9     2       1        1              1              1
10    0       1        1              0              1
11    1       1        0              0              1
12    1       2        1              1              1
13    2       1        0              0              0

Predictions:
['buy', 'buy', 'buy', 'buy', 'not buy']
```