# ITE4005 – Assignment2

박준영

## 1. Summary

본 프로젝트의 목표는 모든 feature 가 categorical 데이터인 tabular 데이터셋을 학습하는 decision tree 를 구현하는 것이다. 본 프로젝트에선 gain ratio 와 gini index 기반으로 하는 decision tree 여러 개를 ensemble 하는 방식을 이용한다. 세부적인 구현 사항은 아래 기술된 것과 같다.

### 1.1. Data structure

Tabular 데이터를 분석하고 다루는데 유용한 pandas 라이브러리의 DataFrame 을 이용해 내부적으로 tree 의 node 에 속한 데이터를 표현하였다. 구체적으로 decision tree 에 속한 각 node 는 "node 에 속한 데이터", "해당 node 가 leaf 인지 여부", "해당 node 의 label", "해당 node 를 나눈 기준", "해당 node 의 자식" 정보를 저장한다. 해당 node 의 label 은 prediction 과정에서 올바른 자식 node 로 가지 못하는 경우를 대비하여 해당 node 에 속한 sample 중 더 많은 수를 가진 label 로 해당 node 의 label 을 설정하였다. 예컨대 어떤 node 에 A 란 label 을 가진 sample 이 5 개, B 란 label 을 가진 sample 이 3 개인 경우 해당 node 의 label 은 A 가 된다.

### 1.2. Training/Prediction

해당 tree 를 학습하는 과정에선 tree 의 root node 부터 시작하여 자신의 date 를 특정 metric 에 따라 나누어 자식 node 를 만들고, 자식 node 도 같은 작업을 하도록 한다. 이때 자식 node 가 leaf 이면 자식 node 는 더 이상 나누지 않는다. 또한 decision tree 가 overfitting 되는 것을 막기 위해 한 node 에 sample data 가 3 개 미만으로 있을 경우엔 해당 node 를 강제로 leaf node 로 만들고 분리하지 않는다.

비슷하게 prediction 과정에선 root node 부터 시작하여, 주어진 입력 데이터에서 해당 node 를 나눈 기준이 되는 feature data 를 확인하여 그에 맞는 자식 node 로 포인터를 옮긴다. 이 작업은 현재 포인팅 된 node 가 leaf node 이거나 node 에 class 에 해당하는 자식 node 가 없을 경우 현재 포인팅 된 node 에서 tree 의 순회를 멈춘다. 최종적으로 포인팅 된 node 의 label 을 반환함으로써 prediction 과정을 마친다.

### 1.3. Gain ratio

Gain ratio 는 두 가지 버전이 존재한다. 하나는 한 feature 에 클래스가 2 개 이상인 경우 각 클래스 개수만큼의 자식을 만들어 각각의 노드에 하나의 클래스를 할당하는 것이고, 다른 하나는 한 클래스만을 택해 해당하는 클래스에 속한 sample 을 담은 자식과 그렇지 않은 sample 을 담은 자식을 만드는 이진트리(Binary tree) 버전이다.

두 버전 모두 gain ratio 공식을 통해 노드를 나눌 기준을 정한다. 다만 binary tree 버전의 경우 어떤 feature 의 특정 class 를 기준으로 자른 후 gain ratio 를 계산하고, 다른 버전의 경우 어떤 feature 에 속한 각 class 를 기준으로 자른 후 gain ratio 를 계산한다는 차이가 있다. 추가적으로 binary tree 의 경우 나눴을 때 자식 node 가 하나 이하인 경우, 해당 node 를 leaf node 로 지정한다.

### 1.4. Gini index

Gini index 는 binary tree 버전만 존재한다. Gain ratio 의 binary tree 버전과 마찬가지로 어떤 feature 의 특정 class 를 기준으로 node 를 쪼개어 gini index 를 계산하여 가장 큰 값을 갖는 기준으로 노드를 분할한다. 다른 사항은 gain ratio 의 binary tree 와 동일하다.

### 1.5. Ensemble

하나의 decision tree 를 학습시켰을 때 training data 에 overfitting 되는 것을 막고자 ensemble 을 도입하였다. 다만 데이터셋이 충분할 때(100 개 이상)일 때만 ensemble 을 수행하며, 100 개 미만의 train data 가 있을 경우엔 수행하지 않는다.

우선 전체 training data 를 랜덤 샘플링 하여 전체 train data 의 90% 크기를 가진 sub dataset 100 개를 랜덤하게 만든다. 그 후, 100 개의 decision tree 를 각 sub dataset 을 이용해 학습시킨다. 이후 prediction 과정에서는 각 model 별로 공정하게 결과를 수집하여 최종적인 결과를 낸다. 예컨대 A 로 예측한 모델이 49 개, B 로 예측한 모델이 51 개라면 최종적인 출력은 B 가 되는 것이다.

## 2. Detailed description

### main entry

```python
if __name__ == '__main__':
    if len(sys.argv) != 4:
        print('usage: python {} <training file> <test file> <output file>'.format(sys.argv[0]))
        sys.exit(-1)

    training_file, test_file, output_file = sys.argv[1:]

    df_train, orig_label = parse_dataset(training_file, True)
    df_test, _ = parse_dataset(test_file, False)

    if len(df_train) < 100:
        model = DecisionTree()
        model.fit(df_train, calc_gain_ratio)
    else:
        model = RandomForest(n_estimators=100)
        model.fit(df_train)

    for i in range(len(df_test)):
        df_test.loc[i, orig_label] = model.predict(df_test.iloc[i])

    df_test.to_csv(output_file, sep='\t', index=False)
```

프로그램이 실행되면 제일 처음으로 실행되는 코드로, 데이터를 읽어오고 학습한 후 test data 에 대해 prediction 을 하여 결과를 내보내는 것을 수행한다. 각 작업은 해당하는 subroutine 을 호출하는 방식으로 수행된다.

### parse_dataset function

```python
def parse_dataset(filename: str, training: bool) -> Tuple[pd.DataFrame, str]:
    df = pd.read_csv(filename, sep='\t')
    orig_label = df.columns[-1]
    if training:
        df.rename(columns = { orig_label : 'label' }, inplace = True)

    return df, orig_label
```

데이터를 파싱하는 함수이다. pandas 의 api 를 활용하였으며, 구현의 편의를 위해 중간에 label 에 해당하는 column 명을 label 로 변경한다. 또한 이후에 다시 원래 이름으로 복구하기 위해 원본 이름도 같이 반환한다.

**decision tree class**

```
class DecisionTree:
    def __init__(self, backend):
        self.backend = backend

    def fit(self, df: pd.DataFrame):
        self.tree = self.backend(df)
        self.tree.fit()

    def predict(self, df: pd.DataFrame) -> str:
        return self.tree.predict(df)
```

decision tree 를 나타내는 클래스이다. gain ratio, gini index 를 편하게 스위칭 하고, 이후 ensemble 등을 편하게 수행하기 위해 같은 interface 를 제공하는 것을 목표로 설계되었다.

**random forest class**

```
class RandomForest:
    def __init__(self, n_estimators: int):
        self.n_estimators: int = n_estimators
        self.estimators: List[DecisionTree] = []

    def fit(self, df: pd.DataFrame):
        indices = list(range(len(df)))

        dfs = []
        stride = len(indices)//self.n_estimators
        for i in range(self.n_estimators):
            random.shuffle(indices)
            dfs.append(df.iloc[indices[:-stride]])

        for i in range(self.n_estimators//2):
            tree = DecisionTree(DTNode2)
            tree.fit(dfs[i])
            self.estimators.append(tree)

        for i in range(self.n_estimators//2):
            tree = DecisionTree(BDTNode)
            tree.fit(dfs[i])
            self.estimators.append(tree)
```

ensemble 을 수행하는 클래스이다. tree 의 개수를 입력 받아 해당 개수만큼의 sub dataset 을 만들고 decision tree 도 만들어 학습하는 코드다. 다만 tree 의 다양성을 극대화하기 위해 만든

decision tree 의 절반은 gain ratio 방식으로, 나머지 절반은 gini index 방식으로 학습하였다. gain ratio 의 non-binary tree 버전의 경우 binary tree 버전보다 정확도가 낮아 학습에선 배제하였다.

```
def predict(self, df: pd.DataFrame) -> str:
    votes = dict()

    for i, estimator in enumerate(self.estimators):
        result = estimator.predict(df)
        votes[result] = votes.get(result, 0) + 1

    max_result = None
    max_value = -999
    for res, val in votes.items():
        if val > max_value:
            max_value = val
            max_result = res
    return max_result
```

ensemble 방식을 이용하여 predict 하는 method 이다. 각 decision tree 의 prediction 결과를 모두 수집하여 가장 많이 선택된 label 을 반환하는 방식으로 구현되었다.

**metrics**

```
def calc_info(children: List[pd.DataFrame]):
    col_frac = np.zeros(len(children))
    infos = np.zeros(len(children))

    for i, df in enumerate(children):
        frac = df['label'].value_counts(normalize=True).to_numpy()
        infos[i] = -np.sum(frac * np.log2(frac))
        col_frac[i] = len(df)

    col_frac /= np.sum(col_frac)
    return np.sum(np.multiply(col_frac, infos))
```

```
def calc_gini(children: List[pd.DataFrame]):
    col_frac = np.zeros(len(children))
    ginis = np.zeros(len(children))

    for i, df in enumerate(children):
        frac = df['label'].value_counts(normalize=True).to_numpy()
        ginis[i] = 1 - np.sum(frac * frac)
        col_frac[i] = len(df)

    col_frac /= np.sum(col_frac)
    return np.sum(np.multiply(col_frac, ginis))
```

```python
def calc_gain(parent: pd.DataFrame, children: List[pd.DataFrame]):
    if len(children) == 1:
        return -np.inf

    return calc_info([parent]) - calc_info(children)
```

```python
def calc_gain_ratio(parent: pd.DataFrame, children: List[pd.DataFrame]):
    if len(children) == 1:
        return -np.inf

    gain = calc_gain(parent, children)
    split_info = np.array([len(child) for child in children])
    split_info = split_info / np.sum(split_info)
    split_info = -np.sum(split_info * np.log2(split_info))

    return gain / split_info
```

```python
def calc_gini_index(parent: pd.DataFrame, children: List[pd.DataFrame]):
    if len(children) == 1:
        return -np.inf

    return calc_gini([parent]) - calc_gini(children)
```

gain ratio 과 gini index 를 계산하기 위한 utility 함수들이다.

**gain ratio non-binary tree version**

```python
class DTNode:
    def __init__(self, df: pd.DataFrame):
        self.df = df
        self.is_leaf = (len(df['label'].unique()) == 1)

        label_counts = df['label'].value_counts(normalize=True)
        max_label = np.argmax(label_counts.to_numpy())

        self.label = label_counts.index[max_label]

        self.children: Dict[str, DTNode] = None
        self.split_pivot = None
```

non-binary tree version 의 생성자이다. 이 알고리즘의 경우엔 노드를 나누는 기준이 feature 밖에 없기 때문에 split_pivot field 하나만 가지고 있다.

```python
def _split(self, column: str) -> Dict[str, pd.DataFrame]:
    children = dict()

    types = self.df[column].unique()
    for t in types:
        children[t] = self.df[self.df[column] == t]

    return children
```

노드를 쪼개는 method 이다. column 이 주어지면 해당 column 을 기준으로 노드를 쪼개어 리스트로 반환한다. 이 method 의 경우 해당하는 node 를 바꾸지는 않는다.

```python
def predict(self, df: pd.DataFrame) -> str:
    if self.is_leaf:
        return self.label

    key = df[self.split_pivot]
    if key not in self.children:
        return self.label

    return self.children[key].predict(df)
```

predict 를 수행하는 method 다. leaf 이거나 해당하는 key 를 가진 자식 node 가 없다면 해당 node 의 label 을 반환하고, 그렇지 않다면 자식 node 의 predict method 를 호출한다.

```python
def fit(self):
    columns = self.df.columns[:-1]
    scores = np.zeros(len(columns))

    if len(self.df) < 3:
        self.is_leaf = True
        return

    for i, column in enumerate(columns):
        children = self._split(column)
        scores[i] = calc_gain_ratio(self.df, list(children.values()))

    max_score = np.max(scores)
    if np.isneginf(max_score) or max_score < 1e-1:
        self.is_leaf = True
        return

    column = columns[np.argmax(scores)]
    self.split_pivot = column
    self.children = {t: DTNode(child) for t, child in self._split(column).items()}

    for child in self.children.values():
        if not child.is_leaf:
            child.fit()
```

node 에 해당하는 샘플의 개수가 3 개 미만인 경우 pruning 을 하고, 각 column 별로 gain ratio 를 구해 최대로 하는 column 을 선택해 자식 node 를 만든다. 이때 어떠한 사유로 쪼갤 방법이 없거나 쪼갰을 때 충분히 improve 되지 않는다면 마찬가지로 pruning 을 수행한다.

**gain ratio binary tree version**

```
class DTNode2:
    def __init__(self, df: pd.DataFrame):
        self.df = df
        self.is_leaf = (len(df['label'].unique()) == 1)

        label_counts = df['label'].value_counts(normalize=True)
        max_label = np.argmax(label_counts.to_numpy())

        self.label = label_counts.index[max_label]

        self.children: List[DTNode2] = None
        self.split_pivot = None
        self.split_value = None
```

gain ratio 의 binary tree 버전의 경우 쪼개는 기준이 feature 와 value 이므로 추가적으로 split_value field 를 추가하였다.

```
def _split(self, column: str, value: str) -> List[pd.DataFrame]:
    children = list()
    children.append(self.df[self.df[column] == value])
    children.append(self.df[self.df[column] != value])

    return children
```

node 를 쪼개는 method 의 경우 위 non-binary tree 버전과 다르게 어떤 value 인 sample 과 그렇지 않은 sample 을 따로 하여 두 개의 child node 를 만든다.

```
def predict(self, df: pd.DataFrame) -> str:
    if self.is_leaf:
        return self.label

    key = 0 if df[self.split_pivot] == self.split_value else 1
    return self.children[key].predict(df)
```

predict 의 경우엔 value 를 비교하여 해당하는 child node 의 predict method 를 호출하는 방식으로 구현하였다. 다만 leaf node 인 경우 해당 node 의 label 을 반환한다. 만약 해당하는 child node 가 없었다면 애초에 leaf node 로 지정이 되었을 것이므로 이에 대한 예외 처리는 하지 않았다.

```
def fit(self):
    features = []
    for column in self.df.columns[:-1]:
        values = self.df[column].unique()
        for value in values:
            features.append([column, value])

    scores = np.zeros(len(features))
```

위와는 다르게 feature, value tuple 을 모두 만들어 각 tuple 의 gain ratio 값을 구해 위 non-binary tree version 과 유사하게 학습한다. 다만 gain ratio value 에 대한 pruning 은 수행하지 않았다.

**gini index**

```
class BDTNode:
    def __init__(self, df: pd.DataFrame):
        self.df = df
        self.is_leaf = (len(df['label'].unique()) == 1)

        label_counts = df['label'].value_counts(normalize=True)
        max_label = np.argmax(label_counts.to_numpy())

        self.label = label_counts.index[max_label]

        self.children: List[BDTNode] = None
        self.split_pivot = None
        self.split_value = None
```

위 gain ratio 의 binary tree version 과 마찬가지로 node 를 나누는 기준이 두 개이므로 각각 split_pivot, split_value 로 기준을 저장하도록 하였다.

```
def _split(self, column: str, value: str) -> List[pd.DataFrame]:
    children = list()
    children.append(self.df[self.df[column] == value])
    children.append(self.df[self.df[column] != value])

    return children
def predict(self, df: pd.DataFrame) -> str:
    if self.is_leaf:
        return self.label

    key = 0 if df[self.split_pivot] == self.split_value else 1
    return self.children[key].predict(df)
```

노드를 분할하는 method 와 predict method 도 gain ratio 의 binary tree version 의 것과 동일하다.

```python
def fit(self):
    features = []
    for column in self.df.columns[:-1]:
        values = self.df[column].unique()
        for value in values:
            features.append([column, value])

    scores = np.zeros(len(features))

    if len(self.df) < 3:
        self.is_leaf = True
        return

    for i, feat in enumerate(features):
        column, value = feat
        children = self._split(column, value)



        if len(children[0]) == 0 or len(children[1]) == 0:
            scores[i] = -np.inf
        else:
            scores[i] = calc_gini_index(self.df, children)

    max_score = np.max(scores)
    if np.isneginf(max_score):
        self.is_leaf = True
        return

    column, value = features[np.argmax(scores)]
    self.split_pivot = column
    self.split_value = value
    self.children = [BDTNode(df) for df in self._split(column, value)]

    for child in self.children:
        if not child.is_leaf:
            child.fit()
```

fit method 는 gain ratio 의 binary tree version 과 동일하지만, metric 이 gini index 로 변경되었다.

## 3. Instructions for executing the program

우선 다음 명령을 통해 본 프로젝트를 받는다.

```
git clone https://github.com/frechele/ITE4005
```

이후 다음 명령을 통해 본 프로그램의 위치로 이동한다.

```
cd ITE4005/assignment2
```

마지막으로, 다음의 명령을 통해 프로그램을 실행시킨다.

```
python dt.py <train file> <test file> <output file>
ex) python dt.py dt_train.txt dt_test.txt dt_output.txt
```

## 4. Additional information

본 프로젝트는 다음의 환경에서 테스트되었다.

- OS: macOS Monterey 12.3.1
- CPU: Apple M1 Pro
- RAM: 16GB
- Python 3.9.7
- Numpy 1.20.3
- Pandas 1.3.4