Implementation of Decision Tree

2018008395 박정호

1. Algorithm

기본적인 구조는 강의에서 설명된 알고리즘의 틀을 따라가려 했다. 여기서 성능을 높이기 위해 numpy의 broadcasting, boolean indexing 등을 이용해서 최대한 iteration 횟수를 줄이고, numpy 라이브러리를 사용하는 이점을 최대한 살리려 노력했다.

Decision Tree를 구축하는 과정은 재귀적으로 발생하고, 각 과정은 크게 3단계 정도로 나누어진다. partitioning의 필요성 및 가능 여부 확인, partition의 기준이 될 attribute selection, data set의 partition 및 새로운 노드의 생성이 바로 그것이다. 이 과정은 각 노드의 partition이 불가능 혹은 불필요해질 때까지 재귀적으로 수행된다.

partitioning의 필요성 및 가능 여부 확인은 데이터셋의 엔트로피 확인, major class label이 차지하는 영역의 비율, partitioning을 위한 잔여 attribute의 유무 등을 통해서 판단한다. 우선 엔트로피가 0이라던지, majority voting을 통해 구해진 major class label이 일정 비율 이상을 차지한다면 이미 어느 정도 classify가 되었다고 판단하고, 더 이상 partitioning을 하지 않았다. 또한, 이미 모든 attribute를 partition에 사용했다면 더 이상 partitioning이 불가능하기에, 이때도 partitioning을 하지 않았다.

attribute selection measure로는 information gain을 사용했다. 또한 이렇게 partition된 데이터의 집합 각각을 새로운 노드로 만들 때, 그 크기가 기존 데이터셋의 5% 미만인 것들은 새로운 노드를 만들지 않는 식으로 pruning을 했다. major class label의 비율을 통해 1차적으로 pruning을 했지만, 그럼에도 걸러지지 않은 과도하게 작은 partition은 overfitting을 야기할 수 있기에 이런 결정을 했다. 물론 이로 인한 오차도 분명 발생하겠지만, 전체적인 오차는 줄어들 것이라 생각했다.

여기까지가 Decision Tree를 구축하는 과정이며, 이를 이용하여 새로운 데이터를 classify하는 것은 간단한 트리 탐색 연산을 이용한다. 각 노드에 저장된 test attribute를 이용해서 다음으로 탐색할 노드를 선택하며, 내려갈 노드가 없을 경우(이미 leaf 노드에 도달했거나, pruning 등의 이유로 해당 attribute value에 대응하는 노드가 생성되지 않았을 경우), 현재 노드에 저장된 major class label을 classify의 결과로 선택하는 것이다.

2. Implementation

1. 입력 처리

```
# read input file
# Since DT algorithm in this source code is using vector operation with numpy,
# Dataset is parsed to numpy array
# To eliminate randomness, class labels are sorted
def input_training_set(filename):
    file = open(filename)

    attributes = np.array(file.readline().strip().split('\t'))
    class_labels = []
    training_set = []

    while True:
        line = file.readline()
        if not line:
            break
        data = np.array(line.strip().split('\t'))
        class_labels.append(data[-1])
        training_set.append(data)

    training_set = np.array(training_set)
    class_labels = np.unique(class_labels)
    class_labels.sort()
    file.close()

    return attributes, class_labels, training_set
```

기본적으론 지난 Apriori algorithm 구현에서 사용한 input_process function과 같다. 다만, 후술할 **Decision Tree** 알고리즘에서 사용하기 위해 데이터셋을 3 부분으로 분리했다. 우선 데이터셋이 가지는 attribute의 종류(class label의 이름 포함)를 저장하는 attributes, 데이터셋이 가지는 class label의 종류를 저장하는 class_labels, 마지막으로 실제 학습에 이용될 training_set이 바로 그것이다.

모든 데이터는 numpy array로 만들어지는데, 이는 후술할 Decision Tree 알고리즘에서 수행 시간을 줄이기 위해 벡터 연산을 쓰기 때문이다.

여기서 특이한 점은 class_labels는 중복을 제거하는 unique 연산 이후 정렬된다는 것인데, 기존 구현에서는 class_labels를 python set type으로 두었고, item의 순서를 보장하지 않는 python set type의 특성으로 인해 후술할 majority voting에 일종의 랜덤성이 생기게 되었다. 따라서, 일반 numpy array로 두고, 정렬하는 것으로 같은 데이터에 대해서 같은 순서를 가지도록 했다.

2. Decision Tree Class

1. 기본 구성

```
# Decision Tree Class
class DT:
    # Constructor
    # mask is set of already used attribute index
    def __init__(self, mask):
        self.child = {}
        self.attr_idx = None
        self.mask = set()
        self.mask.update(mask)
        self.class_label = None
        self.majority_threshold = 0.8
        self.pruning_threshold = 0.05

    # evaluate entropy of dataset
    def entropy_of(self, class_labels, data_set):
        ...

    # evaluate entropy of splited dataset
    def entropy_with_attr_of(self, class_labels, data_set, attr_idx):
        ...

    # majority voting
    # return major class label and its ratio
    def major_label_of(self, class_labels, data_set):
        ...

    # construct Decision Tree recursively
    # pruning with majority_threshold and pruning_threshold
    def construct(self, attributes, class_labels, data_set):
        ...

    # predict unknown class label of given data
    def classify(self, data):
        ...
```

우선 재귀적인 구조를 가지는 Decision Tree를 구현하기 위해 class를 만들었다. 생성자 이외의 함수는 후술하겠으며, 여기서는 기본적인 구조만 다룬다. attr_idx는 해당 노드에서 partition의 기준이 되는 test attribute의 index를 저장하며, class_label은 해당 partition에서 가장 많이 나타난 class label을 저장한다. 이는 classify 중에 leaf 노드까지 탐색하지 못하는 경우(pruning등의 이유로)를 대비한 구현이다. mask는 이전 노드에서 이미 사용한 test attribute의 index를 모두 저장한 집합이다. 이는 이미 사용한 attribute를 다시 partition에 사용하는 것을 피하기 위해 두었다

2. 엔트로피 구하기

이 부분에서 설명할 두 함수는 test attribute selection을 위해 필자가 선택한 방식인 information gain을 구현하기 위해 만든 함수이다. 기본적으로 강의 자료에서 등장한 수식인

$$Info(D) = -\sum_{i=1}^{m} p_i \log_2 p_i$$

$$Info_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} Info(D_j)$$

이 두 수식을 사용했다.

```
    # evaluate entropy of dataset
    def entropy_of(self, class_labels, data_set):
        entropy = 0
        data_labels = data_set.T[-1]
        for label in class_labels:
            p_i = np.sum(data_labels==label)/data_labels.size
            if p_i==0:
                continue
            entropy += p_i*np.log2(p_i)

        return -entropy
```

주어진 데이터셋의 엔트로피를 구하는 함수이다. numpy의 boolean indexing과 broadcasting을 이용해서 iteration 없이 딱 한 번의 iteration으로 전체의 entropy를 구할 수 있도록 했다.

```
    # evaluate entropy of splited dataset
    def entropy_with_attr_of(self, class_labels, data_set, attr_idx):
        entropy = 0
        attr_values = np.unique(data_set.T[attr_idx])

        for attr in attr_values:
            data_subset = data_set[data_set.T[attr_idx]==attr]
            p_i = data_subset.shape[0]/data_set.shape[0]
            entropy += p_i*self.entropy_of(class_labels, data_subset)

        return entropy
```

주어진 데이터셋을 나누었을 때의 엔트로피를 구하는 함수이다. 상술한 entropy_of를 그대로 이용하며, 여기서 split된 데이터셋도 boolean indexing을 이용해서 iteration 없이 구했다. attribute의 종류는 numpy.unique를 사용해서 중복을 제거했고, 오직 데이터셋에 등장하는 값만 걸러냈다.

3. Majority Voting

```
    # majority voting
    # return major class label and its ratio
    def major_label_of(self, class_labels, data_set):
        major_label = None
        max_cnt = 0
        data_labels = data_set.T[-1]
        for label in class_labels:
            cnt = np.sum(data_labels==label)
            if cnt > max_cnt:
                max_cnt = cnt
                major_label = label

        return major_label, max_cnt/data_labels.size
```

단순한 등장 횟수 비교로 데이터 셋의 majority를 구하는 함수이다. 여기서 횟수는 앞에서도 많이 이용한 numpy의 feature를 이용했으며, 후에 pruning에서 사용하기 위해 majority가 데이터셋에서 어느 정도의 비율을 차지하는지도 함께 return했다.

4. 트리 구성

```
    # construct Decision Tree recursively
    # pruning with majority_threshold and pruning_threshold
    def construct(self, attributes, class_labels, data_set):
        if self.entropy_of(class_labels, data_set) == 0:
            self.class_label = data_set[0][-1]
            return

        self.class_label, ratio = self.major_label_of(class_labels, data_set)
        if len(self.mask) == attributes.size - 1 or ratio > self.majority_thre
shold:
            return

        test_attr_idx = None
        max_info_gain = 0

        for attr_idx in range(attributes.size - 1):
            if attr_idx in self.mask:
                continue

            info_gain = self.entropy_of(class_labels, data_set) - self.entropy
_with_attr_of(class_labels, data_set, attr_idx)
            if info_gain > max_info_gain:
                max_info_gain = info_gain
                test_attr_idx = attr_idx
```

```
        new_mask = set()
        new_mask.update(self.mask)
        new_mask.add(test_attr_idx)

        self.attr_idx = test_attr_idx
        attr_values = np.unique(data_set.T[test_attr_idx])

        for attr in attr_values:
            data_subset = data_set[data_set.T[test_attr_idx]==attr]
            subset_ratio = data_subset.shape[0]/data_set.shape[0]
            if subset_ratio < self.pruning_threshold:
                continue
            new_leaf = DT(new_mask)

            new_leaf.construct(attributes, class_labels, data_subset)
            self.child[attr] = new_leaf
```

위에서 구현한 엔트로피 계산 함수, majority voting 함수를 모두 이용해서 Decision Tree를 구축하는 함수이다. 먼저 현재 데이터셋의 엔트로피가 0인지 확인해서 추가적인 classify의 필요성을 확인한다. 이후 노드의 class_label을 현재 데이터셋의 majority로 맞추며, 만약 이 majority의 비율이 미리 정한 threshold보다 크다면 역시 추가적인 classify가 의미 없다고 판단하여 더 이상 진행하지 않는다. 또한 모든 attribute가 test attribute로 사용되었을 경우에도 더 이상 진행하지 않는다.

이후에는 엔트로피 계산 함수들을 이용해서 test attribute를 구한다. 그리고 이 test attribute를 기준으로 데이터셋을 나누는데, 이때 나눠진 데이터셋의 크기가 미리 정한 threshold보다 작다면 이 역시 overfitting의 위험이 있다고 판단하여 새로운 노드를 만들지 않는다. 이 pruning의 대상이 되지 않은 데이터셋들은 모두 새로운 노드로 넘어가게 되며, 이 노드가 다시 재귀적으로 construct 함수를 호출하게 된다. 이 과정에서 데이터셋을 나눌 때도 상술한 함수들과 마찬가지로 numpy의 boolean indexing을 사용했다.

새로운 노드를 만들 때 parameter로 넘기는 new_mask는 기존에 노드가 가지고 있던 mask에 새로 뽑힌 test attribute의 index를 추가한 python set이다. 내부적으로 hash 형식을 사용하는 것으로 알고 있기 때문에 list보다 빠른 탐색이 가능할 것이라 생각했다.

5. Classification

```
    # predict unknown class label of given data
    def classify(self, data):
        if self.attr_idx == None:
            return self.class_label

        attr = data[self.attr_idx]

        if not (attr in self.child):
            return self.class_label

        return self.child[attr].classify(data)
```

생성된 Decision Tree를 이용해 데이터의 class label을 예측하는 함수이다. root에서부터 트리를 탐색해 내려가는 식으로 구현했으며, leaf 노드에 도달했거나, 해당 노드의 child 노드 중 탐색의 대상이 될 노드가 없을 경우 현재 노드의 class label (위에서 구한 majority)를 return하도록 했다.

6. Master Decision Tree Function

```
# master Decision Tree Building Process
# using DT class, it returns a Decision Tree instance
def build_decision_tree(attributes, class_labels, training_set):
    decision_tree = DT(set())

    decision_tree.construct(attributes, class_labels, training_set)

    return decision_tree
```

상술한 Decision Tree class를 이용하는 master function이다. Input_process에서 읽어온 값을 모두 이용한다. Decision Tree의 root에 전달할 mask는 빈 집합으로 한다. 세부적인 동작은 모두 DT.construct에서 수행하기에, 이 함수는 DT.construct만 호출하게 된다.

3. 출력 처리

1. 각 데이터의 string formatting

```
# formatting function for numpy array with string
def array_to_str(arr):
    result = ''
    for s in arr:
        result += s
        result += '\t'
    result = result[:-1]
    result += '\n'

    return result
```

테스트를 거친 결과를 저장한 numpy array를 올바른 형식의 string으로 변환하는 함수이다. numpy array의 각 element를 하나의 문자열에 붙여나가면서 중간중간에 '₩t'를 추가하고, 맨 마지막의 '₩t'를 제거한 다음 개행 문자를 추가하는 식으로 구현했다.

2. 테스트와 파일 출력

```
# read test file and write test result to output file
def test_and_output(attributes, decision_tree, test_filename, output_filename)
:
    test_file = open(test_filename)
    output_file = open(output_filename, mode='w')

    test_file.readline()
    output_file.write(array_to_str(attributes))

    while True:
        line = test_file.readline()
        if not line:
            break
        data = np.array(line.strip().split('\t'))

        class_label = decision_tree.classify(data)
        data = np.append(data, class_label)
        output_file.write(array_to_str(data))

    test_file.close()
    output_file.close()
```

구축한 Decision Tree를 이용해서 testset file의 각 데이터를 classify하고, 그 결과를 상술한 array_to_str 함수를 통해 파싱해서 output file에 적는 함수이다. 파일을 한줄 한줄 읽어야 하기에 기본 구조는 input_training_set과 같으며, 한줄을 읽을 때마다 classify해서 바로 output file에 적었다.

4. main 함수

```
def main(argv):
    if len(argv)<4:
        print('PLEASE, give 3 arguments (training filename, test filename, out
put filename)')
        return
    attributes, class_labels, training_set = input_training_set(argv[1])

    sys.setrecursionlimit(max(attributes.size*10, 10000))
    train_start = time.time()
    decision_tree = build_decision_tree(attributes, class_labels, training_set
)
    train_end = time.time()
    print('Building Time : %f sec' % (train_end-train_start))
    test_start = time.time()
    test_and_output(attributes, decision_tree, argv[2], argv[3])
    test_end = time.time()
    print('Testing Time : %f sec' % (test_end-test_start))
```

위에서 구현된 모든 함수를 이용하는 함수이다. Decision Tree를 구축하는 시간은 I/O 시간을 배제한 상태로 측정했으며, testset을 classify하는 시간은 소스코드의 구조 상, I/O 시간을 포함한 상태로 측정했다.

3. Usage

우선 전체 코드는 python으로 작성되었으며, 버전은 3.8.2이다.

numpy 라이브러리를 활용했으며, numpy 버전은 1.18.2이다.

또한 python은 인터프리터 언어이기에 별도의 컴파일 명령어나 Makefile을 쓰지 않았으며, 아래의 밑줄 그어진 부분의 명령어로 실행할 수 있다. 필자는 .py 파일에 대한 환경변수 설정이 되어있지 않았기 때문에 파일명만으로 실행하는 것이 불가능했음을 밝힌다.

```
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2> python dt.py .\data\dt_train.txt .\data\dt_test.txt .\result\dt_result.txt
Building Time : 0.001984 sec
Testing Time : 0.002993 sec
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2>
```

스크린샷의 크기 문제로 인해 명령어만을 포함한 스크린샷을 하나 더 첨부하겠다.
여기서 command line argument에 전달된 filename에 디렉토리 정보도 포함된 것은 단순히 필자의 디렉토리 구조 때문이므로, 실제 실행에는

`python dt.py (training filename) (test filename) (output filename)`

의 구조만 지키면 된다.

```
python dt.py dt_train.txt dt_test.txt dt_result.txt
```

4. Result

```
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2> python dt.py dt_train.txt dt_test.txt dt_result.txt
Building Time : 0.001996 sec
Testing Time : 0.000963 sec
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2> python dt.py dt_train1.txt dt_test1.txt dt_result1.txt
Building Time : 0.069869 sec
Testing Time : 0.012919 sec
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2> .\dt_test.exe dt_answer.txt dt_result.txt
5 / 5
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2> .\dt_test.exe dt_answer1.txt dt_result1.txt
320 / 346
PS C:\Users\pch68\2021_ite4005_2018008395\assignment2>
```

dt_test.txt는 모든 데이터를 정확하게 classify하는 것을 확인할 수 있고, dt_test1.txt는 346개 중 320개의 데이터에 대해서 정확하게 classify하는 것을 확인할 수 있다.