COMP9814 assignment2

Student name: Ran Bai
Zid: z5187292

Question 1: Search Algorithms for the 15-Puzzle

a).

	Start10	Start12	Start20	Start30	Start40
UCS	2565	Mem	Mem	Mem	Mem
IDS	2407	13812	5297410	Time	Time
A*	33	26	915	Mem	Mem
IDA*	29	21	952	17297	112571

Mem means algorithm run out of memory.
Time means code runs for five minutes without producing out- put.

b).

Answer: according to the table we got in a),

(1) UCS(Uniform-Cost Search): This algorithm is the least efficient among the four algorithms. Both time complexity and space complexity of it is $O(b^{\lceil C^*/\varepsilon \rceil})$. That is why start12-start40 are out of memory.

(2) IDS(Iterative Deepening Search): The time complexity of IDS is $O(b^d)$ which cause "time out" when it turn to start30-start40 or other more complexity cases， and the space complexity is $O(bd)$ where b is branching factor and d is depth of the shallowest solution.

(3) A*(A star algorithm): A* algorithm is second efficient algorithm among these four algorithms. A* search use the evaluation function f(n) = g(n) + h(n) and try to minimize it, where g(n) = cost from initial node to node n and h(n) = estimated cost of cheapest path from n to goal. Comparing with UCS and IDS, it make a faster alternative to search. However, A* search will out of memory beyond start30. Hence, It is not efficient enough to tackle lengthy and complex searches. The time complexity is $O(b^{\varepsilon d})$, and space complexity can up to $O(b^d)$.

(4) IDA* (Iterative Deepening A- Star Search): IDA* is the most efficient algorithm. It is a low-memory variant of A* which performs a series of depth-first searches but cuts off each search when the sum f (n) = g(n) + h(n) exceeds some pre-defined threshold. The threshold is steadily increased with each successive search. The time complexity is $O(b^{\varepsilon d})$, but space complexity is $O(bd)$, which less than A*. So in the start30 and start40 task, it does not appear to out-of-memory or run-out-of-time.

Question 2: Heuristic Path Search for 15-Puzzle

a) .

Answer:

	start50		start 60		start64	
IDA*	50	14642512	60	321252368	64	1209086782
1.2	52	191438	62	230861	66	431033
1.4	66	116342	82	4432	94	190278
1.6	100	33504	148	55626	162	235848
Greedy	164	5447	166	1617	184	2174

b) .

Answer:according to the formula in week 2 tutorial exercise,

$f(n) = (2-w)g(n) + w\,h(n)$, where $0 \le w \le 2$. , so we get f(n} = 0.8 * g(n) + 1.2 * h(n) with w = 1.2, and the code which should be replaced in idastar.pl is F1 is G1 + H1, as the figure show below.

```
% Otherwise, use Prolog backtracking to explore all successors
% of the current node, in the order returned by s.
% Keep searching until goal is found, or F_limit is exceeded.
depthlim(Path, Node, G, F_limit, Sol, G2)  :-
    nb_getval(counter, N),
    N1 is N + 1,
    nb_setval(counter, N1),
    % write(Node),nl,   % print nodes as they are expanded
    s(Node, Node1, C),
    not(member(Node1, Path)),      % Prevent a cycle
    G1 is G + C,
    h(Node1, H1),
    F1 is G1 + H1,
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

It should be replaced to F1 is 0.8 * G1 + 1.2 * H1, as below

```
% Otherwise, use Prolog backtracking to explore all successors
% of the current node, in the order returned by s.
% Keep searching until goal is found, or F_limit is exceeded.
depthlim(Path, Node, G, F_limit, Sol, G2)  :-
    nb_getval(counter, N),
    N1 is N + 1,
    nb_setval(counter, N1),
    % write(Node),nl,   % print nodes as they are expanded
    s(Node, Node1, C),
    not(member(Node1, Path)),      % Prevent a cycle
    G1 is G + C,
    h(Node1, H1),
    F1 is 0.8 * G1 + 1.2 * H1,
    F1 =< F_limit,
    depthlim([Node|Path], Node1, G1, F_limit, Sol, G2).
```

d) .

Answer: All five algorithms are represented by formula f(n) = (2-w)*g(n) + w*h(n), when w = 2 is greedy search and w = 1 is IDA* search. Use IDA* will use more time to get the optimal path. While use greedy search could get an answer quickly, but it could be stuck in loops. From the table in a), the G(the length of path) is increasing when weight w is increasing. Next, when the weight w becomes larger, the number of nodes expanded(N) decreases. That is, the algorithm runs faster when w from 1 to 2, but the path(G) become longer.

Question 3: Decision tree

a) .

Answer:

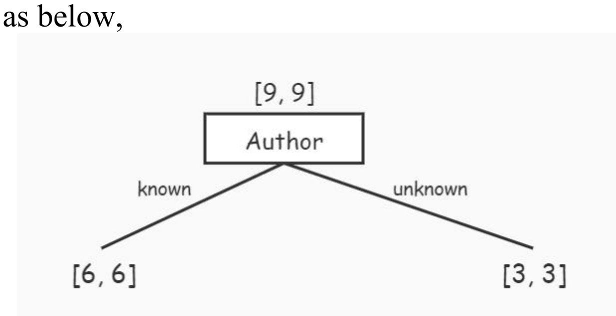
If I change the algorithm to select the first element of the list of feature, as the order of [Author, Thread, Length, WhereRead]. The [X, Y] represents the number of skips action is X, and the number of reads action is Y.

Step 1: splits by Author, the examples can be split into two subsets as below,

Example	Author	Thread	Length	Where_read	User_action
e_1	known	new	long	home	skips
e_4	known	followup	long	home	skips
e_5	known	new	short	home	reads
e_6	known	followup	long	work	skips
e_9	known	followup	long	home	skips
e_{10}	known	new	long	work	skips
e_{12}	known	new	long	work	skips
e_{13}	known	followup	short	home	reads
e_{14}	known	new	short	work	reads
e_{15}	known	new	short	home	reads
e_{16}	known	followup	short	work	reads
e_{17}	known	new	short	home	reads

Example	Author	Thread	Length	Where_read	User_action
e_2	unknown	new	short	work	reads
e_3	unknown	followup	long	work	skips
e_7	unknown	followup	short	work	skips
e_8	unknown	new	short	work	reads
e_{11}	unknown	followup	short	home	skips
e_{18}	unknown	new	short	work	reads

The decision tree is as below,



Step 2: splits by Thread, the two sets we have got can be split into two subsets respectively.

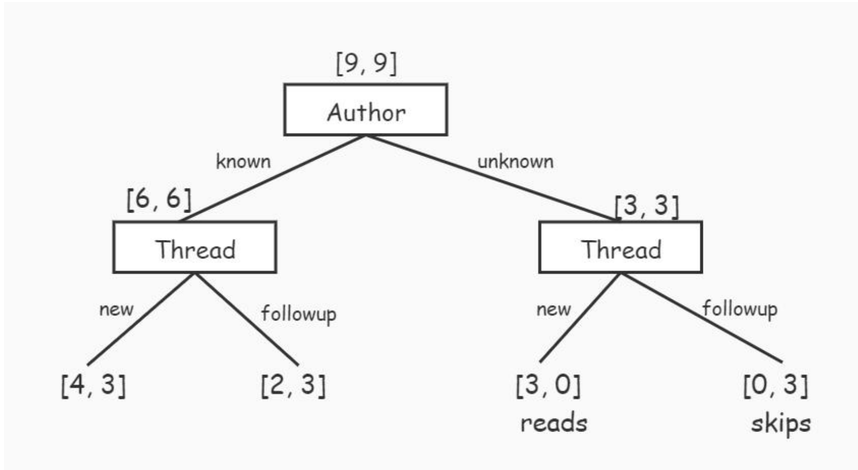
Example	Author	Thread	Length	Where_read	User_action
e_1	known	new	long	home	skips
e_5	known	new	short	home	reads
e_{10}	known	new	long	work	skips
e_{12}	known	new	long	work	skips
e_{14}	known	new	short	work	reads
e_{15}	known	new	short	home	reads
e_{17}	known	new	short	home	reads

Example	Author	Thread	Length	Where_read	User_action
e_4	known	followup	long	home	skips
e_6	known	followup	long	work	skips
e_9	known	followup	long	home	skips
e_{13}	known	followup	short	home	reads
e_{16}	known	followup	short	work	reads

Example	Author	Thread	Length	Where_read	User_action
e_2	unknown	new	short	work	reads
e_8	unknown	new	short	work	reads
e_{18}	unknown	new	short	work	reads

Example	Author	Thread	Length	Where_read	User_action
e_3	unknown	followup	long	work	skips
e_7	unknown	followup	short	work	skips
e_{11}	unknown	followup	short	home	skips

And the decision tree now is,



Step 3: splits by Length, we only need to focus the left tree now, because the right tree is pure class. The subsets is as below,

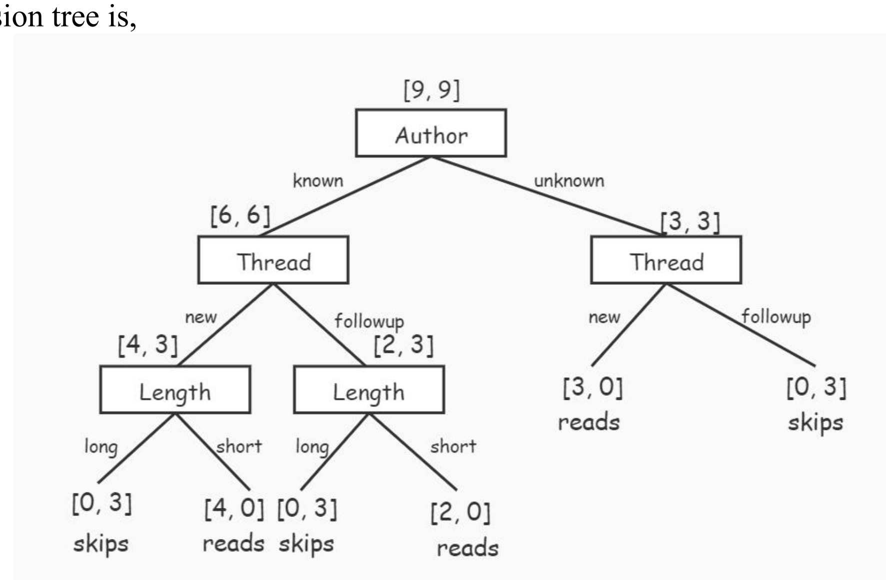
Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_1	*known*	*new*	*long*	*home*	*skips*
e_{10}	*known*	*new*	*long*	*work*	*skips*
e_{12}	*known*	*new*	*long*	*work*	*skips*

Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_5	*known*	*new*	*short*	*home*	*reads*
e_{14}	*known*	*new*	*short*	*work*	*reads*
e_{15}	*known*	*new*	*short*	*home*	*reads*
e_{17}	*known*	*new*	*short*	*home*	*reads*

Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_4	*known*	*followup*	*long*	*home*	*skips*
e_6	*known*	*followup*	*long*	*work*	*skips*
e_9	*known*	*followup*	*long*	*home*	*skips*

Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_{13}	*known*	*followup*	*short*	*home*	*reads*
e_{16}	*known*	*followup*	*short*	*work*	*reads*

The decision tree is,



By comparing the decision tree generated by the maximum entropy principle and the first element principle, we can know they represent <u>different </u>function. This is because the rules that the first elements told us are:

skips <- known ∧ new ∧ long

reads <- known ∧ new ∧ short

skips <- known ∧ followup ∧ long

reads <- known ∧ followup ∧ short

reads <- unknown ∧ new

skips <- unknown ∧ followup

But the rules of decision tree generated by the maximum Information gain principle are:

skips <- long

reads <- short ∧ new

skips <- short ∧ followup ∧ unknown

reads <- short ∧ followup ∧ known

For e19, the first element principle predicts reads, but the maximum Information gain principle predicts skips.

b) .

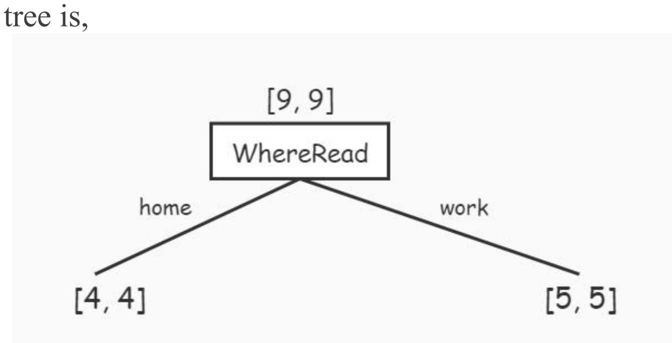
Answer:

The features are in the order of [WhereRead, Thread, Length, Author].

Step 1: split by WhereRead, the examples can be split into two subsets as below,

Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*	*Example*	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_1	*known*	*new*	*long*	*home*	*skips*	e_2	*unknown*	*new*	*short*	*work*	*reads*
e_4	*known*	*followup*	*long*	*home*	*skips*	e_3	*unknown*	*followup*	*long*	*work*	*skips*
e_5	*known*	*new*	*short*	*home*	*reads*	e_6	*known*	*followup*	*long*	*work*	*skips*
e_9	*known*	*followup*	*long*	*home*	*skips*	e_7	*unknown*	*followup*	*short*	*work*	*skips*
e_{11}	*unknown*	*followup*	*short*	*home*	*skips*	e_8	*unknown*	*new*	*short*	*work*	*reads*
e_{13}	*known*	*followup*	*short*	*home*	*reads*	e_{10}	*known*	*new*	*long*	*work*	*skips*
e_{15}	*known*	*new*	*short*	*home*	*reads*	e_{12}	*known*	*new*	*long*	*work*	*skips*
e_{17}	*known*	*new*	*short*	*home*	*reads*	e_{14}	*known*	*new*	*short*	*work*	*reads*
						e_{16}	*known*	*followup*	*short*	*work*	*reads*
						e_{18}	*unknown*	*new*	*short*	*work*	*reads*

Now the decision tree is,



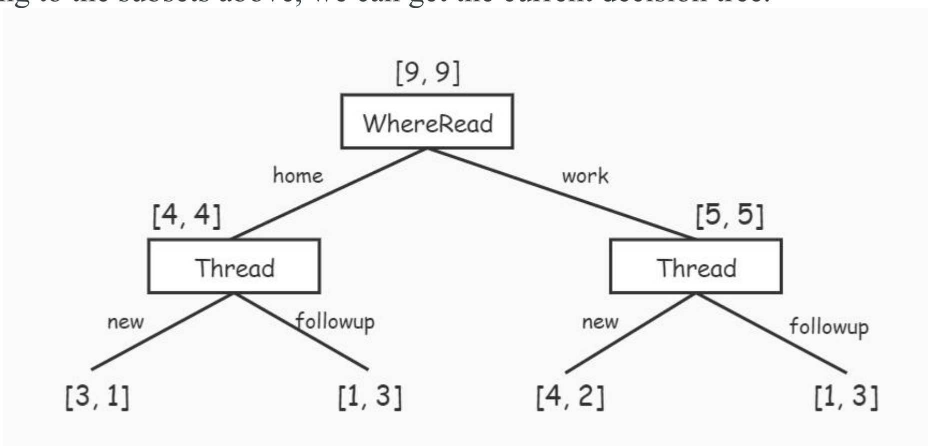
Step 2: next, we split the sets by Thread.

Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*	*Example*	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_1	*known*	*new*	*long*	*home*	*skips*	e_2	*unknown*	*new*	*short*	*work*	*reads*
e_5	*known*	*new*	*short*	*home*	*reads*	e_8	*unknown*	*new*	*short*	*work*	*reads*
e_{15}	*known*	*new*	*short*	*home*	*reads*	e_{10}	*known*	*new*	*long*	*work*	*skips*
e_{17}	*known*	*new*	*short*	*home*	*reads*	e_{12}	*known*	*new*	*long*	*work*	*skips*
						e_{14}	*known*	*new*	*short*	*work*	*reads*
						e_{18}	*unknown*	*new*	*short*	*work*	*reads*

Example	Author	Thread	Length	Where_read	User_action
e_4	known	followup	long	home	skips
e_9	known	followup	long	home	skips
e_{11}	unknown	followup	short	home	skips
e_{13}	known	followup	short	home	reads

Example	Author	Thread	Length	Where_read	User_action
e_3	unknown	followup	long	work	skips
e_6	known	followup	long	work	skips
e_7	unknown	followup	short	work	skips
e_{16}	known	followup	short	work	reads

According to the subsets above, we can get the current decision tree.



Step 3: split by Length,

Example	Author	Thread	Length	Where_read	User_action
e_1	known	new	long	home	skips

Example	Author	Thread	Length	Where_read	User_action
e_5	known	new	short	home	reads
e_{15}	known	new	short	home	reads
e_{17}	known	new	short	home	reads

Example	Author	Thread	Length	Where_read	User_action
e_4	known	followup	long	home	skips
e_9	known	followup	long	home	skips

Example	Author	Thread	Length	Where_read	User_action
e_{11}	unknown	followup	short	home	skips
e_{13}	known	followup	short	home	reads

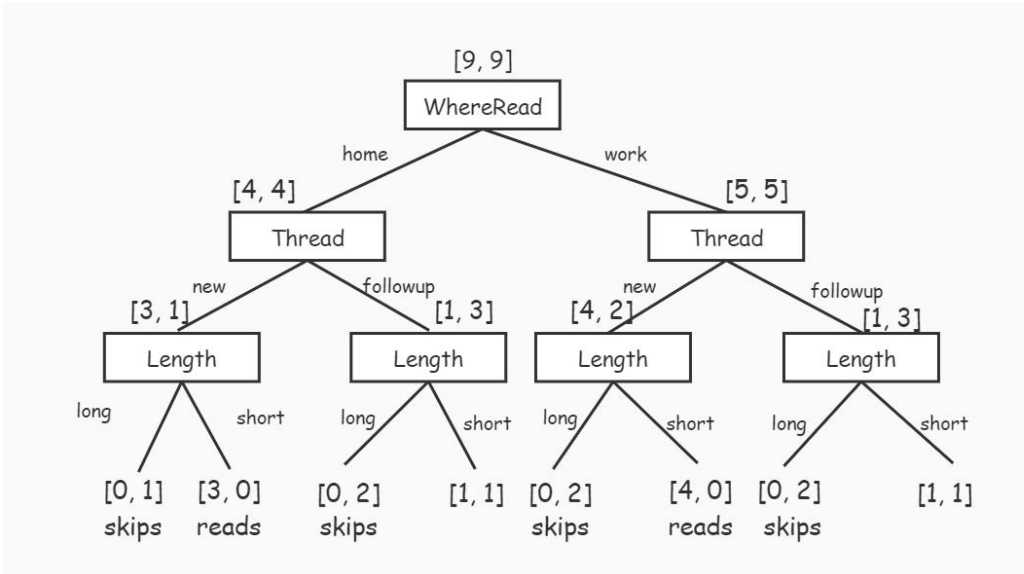
Example	Author	Thread	Length	Where_read	User_action
e_{10}	known	new	long	work	skips
e_{12}	known	new	long	work	skips

Example	Author	Thread	Length	Where_read	User_action
e_2	unknown	new	short	work	reads
e_8	unknown	new	short	work	reads
e_{14}	known	new	short	work	reads
e_{18}	unknown	new	short	work	reads

Example	Author	Thread	Length	Where_read	User_action
e_3	unknown	followup	long	work	skips
e_6	known	followup	long	work	skips

Example	Author	Thread	Length	Where_read	User_action
e_7	unknown	followup	short	work	skips
e_{16}	known	followup	short	work	reads

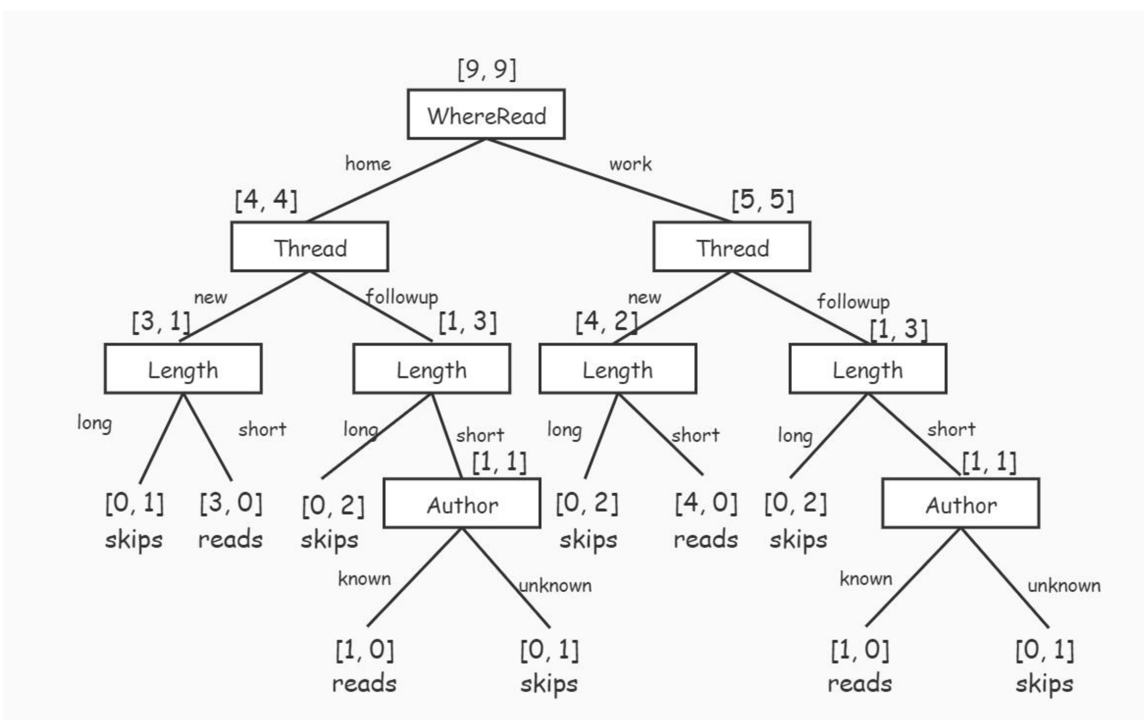
The decision tree currently is,



Step 4: finally, split by Author. From last step, I can know the examples need to analyse are,

Example	*Author*	*Thread*	*Length*	*Where_read*	*User_action*	*Example*	*Author*	*Thread*	*Length*	*Where_read*	*User_action*
e_{11}	*unknown*	*followup*	*short*	*home*	*skips*	e_7	*unknown*	*followup*	*short*	*work*	*skips*
e_{13}	*known*	*followup*	*short*	*home*	*reads*	e_{16}	*known*	*followup*	*short*	*work*	*reads*

So the decision tree generated by the order of [WhereRead, Thread, Length, Author] is,



(1) This tree represent a same function with that found with the maximum information gain split, because they create same rules. The rules of the tree generated in the order of [WhereRead, Thread, Length, Author] as below,

skips <- home ∧ new ∧ long

reads <- home ∧ new ∧ short

skips <- home ∧ followup ∧ long

reads <- home ∧ followup ∧ short ∧ known

skips <- home ∧ followup ∧ short ∧ unknown

skips <- work ∧ new ∧ long

reads <- work ∧ new ∧ short

skips <- work ∧ followup ∧ long

reads <- work ∧ followup ∧ short ∧ known

skips <- work ∧ followup ∧ short ∧ unknown

Then, do some simplification, for example, skips <- home ∧ new ∧ long and skips <- work ∧ new ∧ long can combine to skips <- new ∧ long. So the rules above become to,

skips <- long

reads <- short ∧ new

skips <- short ∧ followup ∧ unknown

reads <- short ∧ followup ∧ known

It's same to the rules of tree found with the maximum information gain, so they represent same function.

(2) Next, according to the result of a) (Tree([Author, Thread, Length, WhereRead]) is different with Tree(the maximum information gain)) and the result in (1) above, so this tree represent a different function with the tree generated in the order of [Author, Thread, Length, WhereRead].

c) .

Answer:

No. Because the rules of other decision can simplify to the two rules above, they represent the same function as one of the trees above.

(1) By asking a) and b), I can know that WhereAt doesn't inference the reads/skips because according to WhereAt, I can not directly classify a class that is completely skips or reads and Its left sub-tree have the same structure as its right sub-tree. So we can consider the tree is found in the order of [Thread, Length, Author]. Then the tree in a) actually represents founding in the order of [Author, Thread, Length], and the tree in figure 7.6 given is in the order of [Length, Thread, Author]. I try all trees founds by threes features, it only exist two different rules as above.

(2) By observating, I find that It cannot determine reads/skips by no more than two features.

So, I think there is no tree represent the different rules with two above.

Question 4:

Answer:

As show in the file, the decision_tree.py build a model by decision tree, and the naive_bayers.py build a model by Gaussian naive bayers. There is accuracy of decision_tree model and naive_bayers model on test dataset as below,

	Decision tree	Naive bayers
Accuracy	85.695%	79.510%

The model of the decision tree I used some methods of pre-pruning to prevent overfitting and improve the accuracy.

(1) Importing data: use load_csv() function in pandas import data into memory.

(2) Cleaning data: In the real data, the data we got may contain a large number of missing values, may contain a lot of noise, or there may be abnormal points due to manual entry errors, which caused some trouble for us to mine valid information, so we Some methods need to be adopted to improve the quality of the data as much as possible.

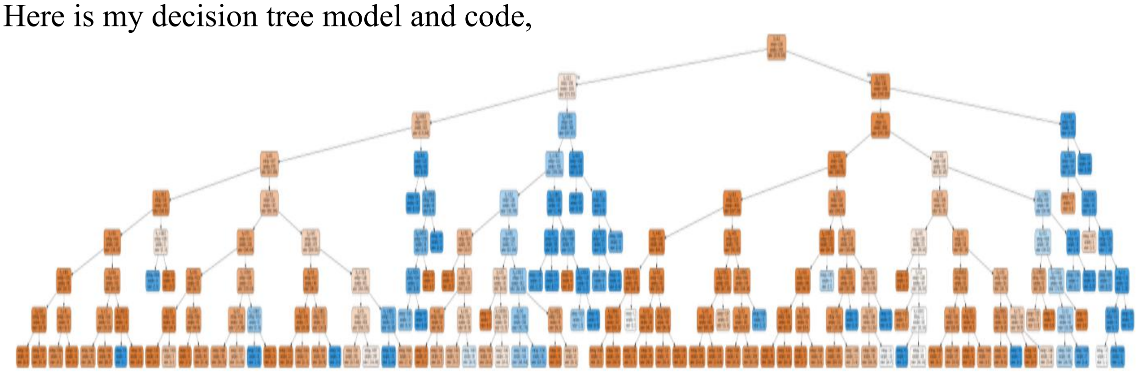
(3) Splitting it into train/test or cross-validation sets: For machine learning tasks, we need to divide the data set into two parts in proportion: the training set and the test set. The training set is used to train the model, and the test set is used to test the performance of our model. This is more applicable when no test set is provided. For this problem, I used adult.data as the training set and adult.test as the test set.

(4) Pre-processing: It concludes dealing with missing data, duplicates data, data standardization and regularization. Three methods for missing value processing: directly use features with missing values; delete features with missing values (this method is effective when attributes containing missing values contain a large number of missing values but only a small number of valid values); missing values Completion. I choose to complete the missing value by filling with most frequency data. Because data is valuable, even if it is incomplete, it has some value.

(5) Transformations: In this task, I use label-encoder to transfer String to integer.

(6) Feature engineering: the transformation of raw data into features suitable for modeling, or removing unnecessary features.

Here is my decision tree model and code,



-*- coding: utf-8 -*-
"""decision_tree.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/12ZpP-vs5o35y1VWXlVx62f6vg3cWtuKc
"""

```
from sklearn import datasets
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.tree.export import export_text
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn import metrics
import graphviz

dataset = pd.read_csv('adult.data', header=None)                    # load data
from adult.data file
array = dataset.iloc[: , :-1].values
tags = dataset.iloc[:, -1].values

imp = SimpleImputer(missing_values=' ?', strategy="most_frequent")           # deal
with missing value by replacing with most frequent value
array[:, ::] = imp.fit_transform(array[:, ::])

labelencoder_X = LabelEncoder()                                              #
convert char type to integer type using laberEncoder
labelencoder_X.fit(array[: , 1])

array[: , 1] = labelencoder_X.fit_transform(array[: , 1])
array[: , 3] = labelencoder_X.fit_transform(array[: , 3])
array[: , 5] = labelencoder_X.fit_transform(array[: , 5])
array[: , 6] = labelencoder_X.fit_transform(array[: , 6])
array[: , 7] = labelencoder_X.fit_transform(array[: , 7])
array[: , 8] = labelencoder_X.fit_transform(array[: , 8])
array[: , 9] = labelencoder_X.fit_transform(array[: , 9])
array[: , 13] = labelencoder_X.fit_transform(array[: , 13])

# using decision tree to classify (and do some prunching to avoid overfitting)
clf = tree.DecisionTreeClassifier(random_state=0, criterion='entropy',
                                  max_depth=8, splitter='best',
                                  min_samples_split=30)

clf = clf.fit(array, tags)
#tree.plot_tree(clf.fit(array, tags))
r = export_text(clf)
print(r)
```

```
feature_name = ['age','workclass','fnlwgt','education','education-num','marital-status',
                'occupation','relationship','race','sex','capital-gain',
                'capital-loss','hours-per-week', 'native-country']
target_name = ['>50k','<=50k']

dot_data = tree.export_graphviz(clf, out_file=None,
                    feature_names=None,
                    class_names=None,
                    filled=True, rounded=True,
                    special_characters=True)
graph = graphviz.Source(dot_data)
graph.format='png'
graph.render()

from IPython.display import Image
Image(filename="Source.gv.png", width = 1000, height=300)

"""TEST"""

# use test dataset to valify accuracy
test_dataset = pd.read_csv('adult.test', header=None ,skip_blank_lines=True, skiprows=1)
# load data from adult.test file
test_features = test_dataset.iloc[: , :-1].values
test_tags = test_dataset.iloc[:, -1].values

test_features[:, ::] = imp.fit_transform(test_features[:, ::])
labelencoder = labelencoder_X
# convert char type to integer type using laberEncoder

test_features[: , 1] = labelencoder_X.fit_transform(test_features[: , 1])
test_features[: , 3] = labelencoder_X.fit_transform(test_features[: , 3])
test_features[: , 5] = labelencoder_X.fit_transform(test_features[: , 5])
test_features[: , 6] = labelencoder_X.fit_transform(test_features[: , 6])
test_features[: , 7] = labelencoder_X.fit_transform(test_features[: , 7])
test_features[: , 8] = labelencoder_X.fit_transform(test_features[: , 8])
test_features[: , 9] = labelencoder_X.fit_transform(test_features[: , 9])
test_features[: , 13] = labelencoder_X.fit_transform(test_features[: , 13])

pred = clf.predict(test_features)
test_tags = [i.replace(".", "") for i in test_tags]
print("The accury is: " + str(metrics.accuracy_score(test_tags, pred)))
```

```
# -*- coding: utf-8 -*-
```

```
"""naive_bayers

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1fAabYhnz1Hn511JVD337iPymJLB2RJxX
"""

from sklearn import datasets
import numpy as np
import pandas as pd
from sklearn.naive_bayes import GaussianNB, CategoricalNB
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.impute import SimpleImputer

dataset = pd.read_csv('adult.data', header=None, skip_blank_lines=True)
# load data from adult.data file
array = dataset.iloc[: , :-1].values
tags = dataset.iloc[:, -1].values

imp = SimpleImputer(missing_values=' ?', strategy="most_frequent")                    # deal
with missing value by replacing with most frequent value
array[:, ::] = imp.fit_transform(array[:, ::])

labelencoder_X = LabelEncoder()                                                          #
convert char type to integer type using laberEncoder

labelencoder_X.fit(array[: , 1])

array[: , 1] = labelencoder_X.fit_transform(array[: , 1])
array[: , 3] = labelencoder_X.fit_transform(array[: , 3])
array[: , 5] = labelencoder_X.fit_transform(array[: , 5])
array[: , 6] = labelencoder_X.fit_transform(array[: , 6])
array[: , 7] = labelencoder_X.fit_transform(array[: , 7])
array[: , 8] = labelencoder_X.fit_transform(array[: , 8])
array[: , 9] = labelencoder_X.fit_transform(array[: , 9])
array[: , 13] = labelencoder_X.fit_transform(array[: , 13])
print(array)

# using decision tree to classify (and do some prunching to avoid overfitting)
gnb = GaussianNB()
gnb = gnb.fit(array, tags)

print(gnb)

# use test dataset to valify accuracy
```

```
test_dataset = pd.read_csv('adult.test', header=None ,skip_blank_lines=True, skiprows=1)
# load data from adult.test file
test_features = test_dataset.iloc[: , :-1].values
test_tags = test_dataset.iloc[:, -1].values

test_features[:, ::] = imp.fit_transform(test_features[:, ::])

# convert char type to integer type using laberEncoder

test_features[: , 1] = labelencoder_X.fit_transform(test_features[: , 1])
test_features[: , 3] = labelencoder_X.fit_transform(test_features[: , 3])
test_features[: , 5] = labelencoder_X.fit_transform(test_features[: , 5])
test_features[: , 6] = labelencoder_X.fit_transform(test_features[: , 6])
test_features[: , 7] = labelencoder_X.fit_transform(test_features[: , 7])
test_features[: , 8] = labelencoder_X.fit_transform(test_features[: , 8])
test_features[: , 9] = labelencoder_X.fit_transform(test_features[: , 9])
test_features[: , 13] = labelencoder_X.fit_transform(test_features[: , 13])

pred = gnb.predict(test_features)

same = 0
for i in range(0, len(pred)):
	if pred[i] + "." == test_tags[i]:
		same += 1

print(pred.size)
print(test_tags.size)
print("The accury is: " + str(same / pred.size))
```