Lucas Summers (lsumme01@calpoly.edu)
Xiuyuan Qiu (xiqiu@calpoly.edu)

CSC 466 Lab 2 Report

Introduction:

Our work implements C4.5 in python and the requested files, c45.py, InduceC45.py, predict.py, crossVal.py, crossValSKL.py, and a README. We also include a small utility file csv_reader.py that reads the special csv format into a pandas dataframe.

The c45 class is fairly straightforward, implementing the .fit() and .predict() methods along with all helper functions such as calculating entropy, info gain and gain ratio. build_tree() recursively builds the tree, using best_split() to find the best attribute to split on at each node. The tree is represented internally as a simple python dict which directly matches the required output format, so .read_tree() and .save_tree() only have to use json.dump()and json.load().

Our evaluation was largely done with custom scripts in eval_cache.py and eval_grapher.py, where trees are stored in the evaltrees/ directory. This is because some of our larger trees were taking upwards of 15 minutes to generate.

Our csvs were stored in a csv/ dir and miscellaneous json outputs in the tree/ dir, though the code can be used without using those dirs.

Implementation Notes:

There is a requirements.txt file for all the libraries we used. There are example run commands in the README for testing purposes.

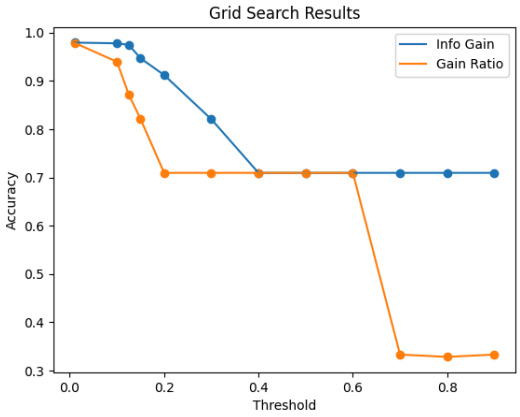
Best Models:

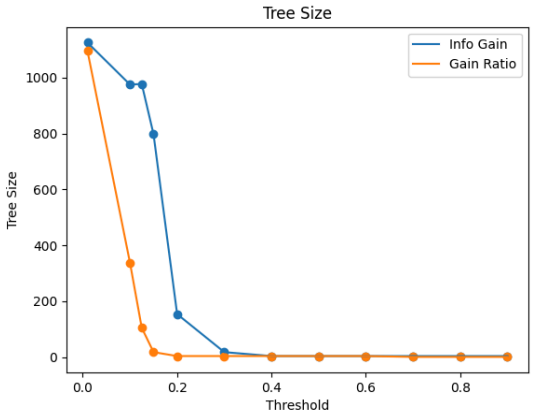
We generally found that larger trees produced higher accuracy scores. In our analysis, we rate hyperparameters by balancing between tree size and accuracy score, with high accuracies and high tree sizes indicating overfitting. Info gain and Info gain ratio performed fairly similar to each other and follow similar trends during cross validation. While getting decent accuracy scores on each dataset, overly large tree sizes lend to concerns of overfitting, especially for the larger nursery and letter recognition datasets.

Nursery

Most of the accurate nursery trees (> 90%) were above 800 in tree size, which ended up seeming a bit bulky; the best tradeoff between accuracy and size happened at info gain=0.2, where tree size was

small (154), and accuracy was still above 90% (91.23%). The most accurate was at info gain=0.01, with 97.96% accuracy, but the tree was more than seven times bigger (1125), and looked overfit.






```
Info gain = 0.2 (acc = 91%)
Actual      not_recom  priority  recommend  spec_prior  very_recom
Predicted
not_recom        4320         0          0           0           0
priority            0      3511          2          18         328
spec_prior          0       755          0        4026           0
```

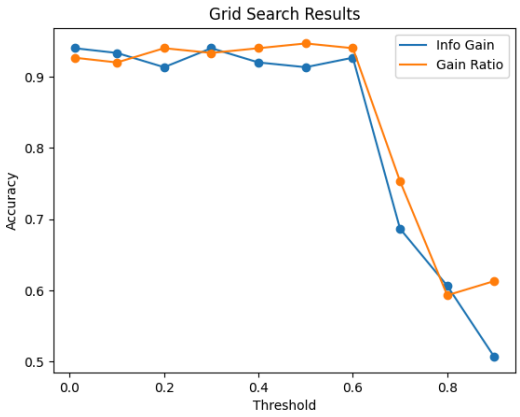
Interestingly enough, the 0.2 tree does not classify anything as recommend or very_recom, since the proportion of the data that has that class is relatively small.

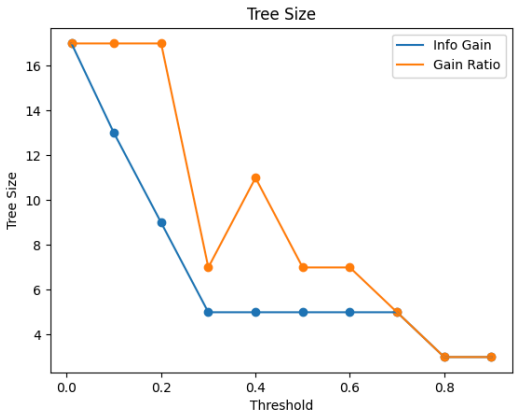
```
Info gain = 0.01 (acc = 98%)
Actual      not_recom  priority  recommend  spec_prior  very_recom
Predicted
not_recom        4320         0          0           0           0
priority            0      4194          0          41          38
recommend           0         0          0           0           2
spec_prior          0        28          0        4003           0
very_recom          0        44          2           0         288
```

The larger tree can have the smaller classes, which makes it more accurate, but the tree must be much larger to handle the edge cases.

IRIS

This one was more straightforward, the most accurate was 94.67% at gain ratio=0.5, with a tree size of 7. The larger trees at lower thresholds did not improve accuracy.



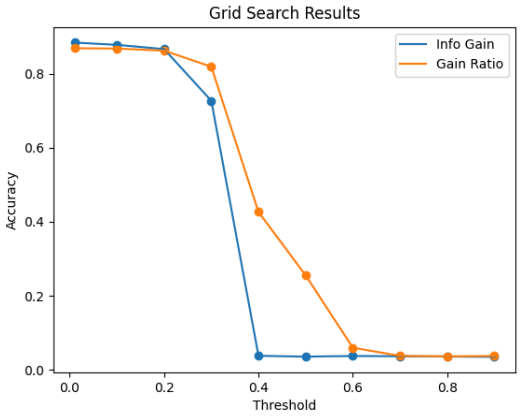


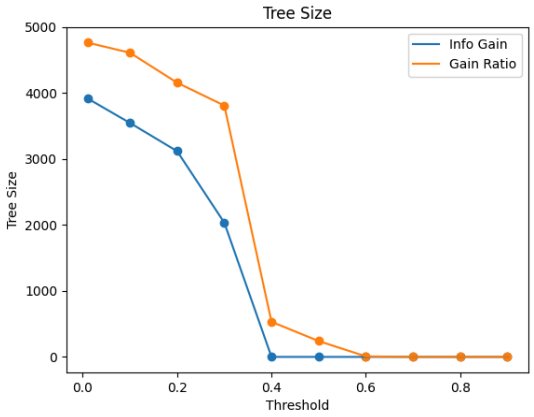
Actual	Iris-setosa	Iris-versicolor	Iris-virginica
Predicted			
Iris-setosa	50	0	0
Iris-versicolor	0	46	4
Iris-virginica	0	4	46

Setosa was easiest to classify, with versicolor and virginica having some minor misidentifications.

Letters

As with nursery, larger trees tended to produce better results, but in this case it seems to make more sense as there are 16 different features to split upon. Accuracy seems to generally plateau around the 0.2 mark for both metrics, the tree that presents the best size/accuracy tradeoff is info_gain=0.2, with an accuracy of 86.62% and a respectable tree size of 3119.





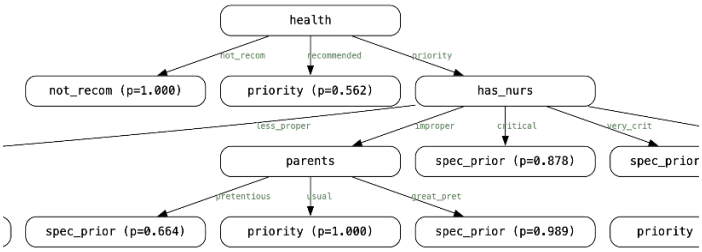
Actual	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Predicted																										
A	746	0	0	4	0	1	2	2	2	5	3	2	2	0	0	1	4	1	4	2	2	1	0	1	1	2
B	1	647	0	15	4	5	10	12	3	7	8	2	2	2	5	3	1	32	16	3	4	14	0	5	2	8
C	1	1	628	0	10	2	45	1	1	0	3	3	1	1	1	0	4	2	0	5	4	4	1	0	0	0
D	2	10	0	667	1	4	11	31	6	7	7	1	3	16	18	1	3	7	3	5	9	0	0	6	1	4
E	1	5	14	0	648	3	15	3	0	0	10	6	1	0	1	3	10	2	12	7	1	1	1	7	1	8
F	3	6	8	5	2	629	5	1	10	4	2	1	1	2	1	42	2	1	15	19	0	7	0	2	10	3
G	3	5	31	3	19	3	599	5	2	1	1	6	3	1	9	5	18	4	7	1	0	2	1	0	1	4
H	1	9	2	16	3	2	3	562	1	1	18	2	3	8	6	3	3	12	4	1	6	0	1	4	5	0
I	0	4	1	3	1	10	1	2	670	25	1	1	0	2	0	4	5	1	8	1	2	1	0	3	1	1
J	2	3	0	3	0	4	0	4	26	657	4	5	0	3	1	0	1	2	0	1	3	1	0	4	0	4
K	0	3	6	3	9	3	3	29	3	3	619	1	4	1	2	0	1	15	0	7	2	0	0	10	0	0
L	4	2	5	4	5	0	2	3	0	4	3	705	0	0	1	0	3	3	6	2	2	0	0	5	0	1
M	3	1	0	0	1	1	4	2	0	0	0	0	736	11	1	1	0	1	0	1	12	2	16	0	1	0
N	0	2	1	13	0	2	1	3	1	1	1	1	11	683	8	1	1	13	2	1	23	7	20	3	2	1
O	2	2	18	9	0	1	9	13	0	6	0	2	5	4	656	2	26	8	3	1	18	3	3	0	2	1
P	0	6	0	18	2	41	3	9	6	2	2	1	2	7	3	712	4	4	0	1	0	11	1	1	4	1
Q	4	3	4	2	14	1	41	2	2	2	3	7	2	2	12	3	670	5	4	1	4	1	1	2	3	8
R	0	21	0	21	6	5	6	21	2	1	20	0	1	9	7	3	5	630	7	0	3	0	3	1	0	2
S	5	10	2	2	7	8	2	4	13	2	4	6	0	1	3	3	4	7	635	7	1	3	0	5	3	22
T	1	1	4	6	0	14	4	1	1	3	1	2	0	1	2	0	5	2	2	705	3	1	0	3	34	3
U	1	3	4	2	0	2	2	15	0	3	1	0	2	7	5	1	1	2	1	2	706	11	2	0	5	0
V	1	8	1	0	1	8	2	1	0	0	0	1	3	5	2	1	2	2	2	5	3	670	10	0	5	0
W	0	1	3	1	0	1	2	0	0	1	1	0	8	14	6	3	1	0	2	2	0	16	689	0	3	0
X	3	6	1	5	16	4	0	4	5	6	26	4	0	0	2	3	1	2	7	4	2	0	0	713	5	7
Y	4	1	2	2	0	18	0	3	1	0	0	0	2	3	0	5	1	0	2	12	3	8	3	3	690	2
Z	1	6	1	1	19	3	1	1	0	6	1	2	0	0	1	3	7	0	6	0	0	0	0	9	7	652

The most confused letter pairs (highlighted above) are:

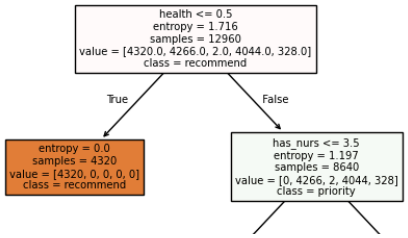
- G seen as a C (45)
- P seen as an F (42)
- F seen as a P (41)
- G seen as a Q (41)

Comparison to SKLearn:

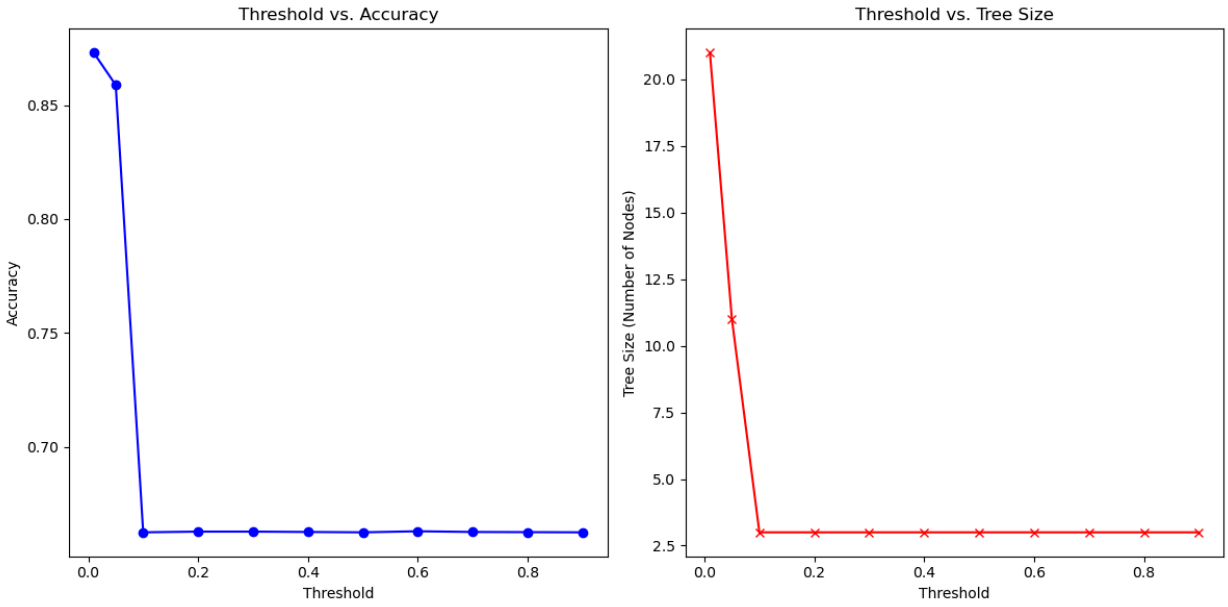
Overall, SKLearn had comparable or worse performance than our implementation overall, with smaller tree sizes. We attribute this to the fact to differences in implementation (SKLearn implements CART), with the most noticeable one being that all of SKLearn's decision trees are binary. For example, on the nursery dataset, instead of splitting on all branches for a category:



It splits in binary on the category indices (note the rightmost node splitting on has_nurs <= 3.5):

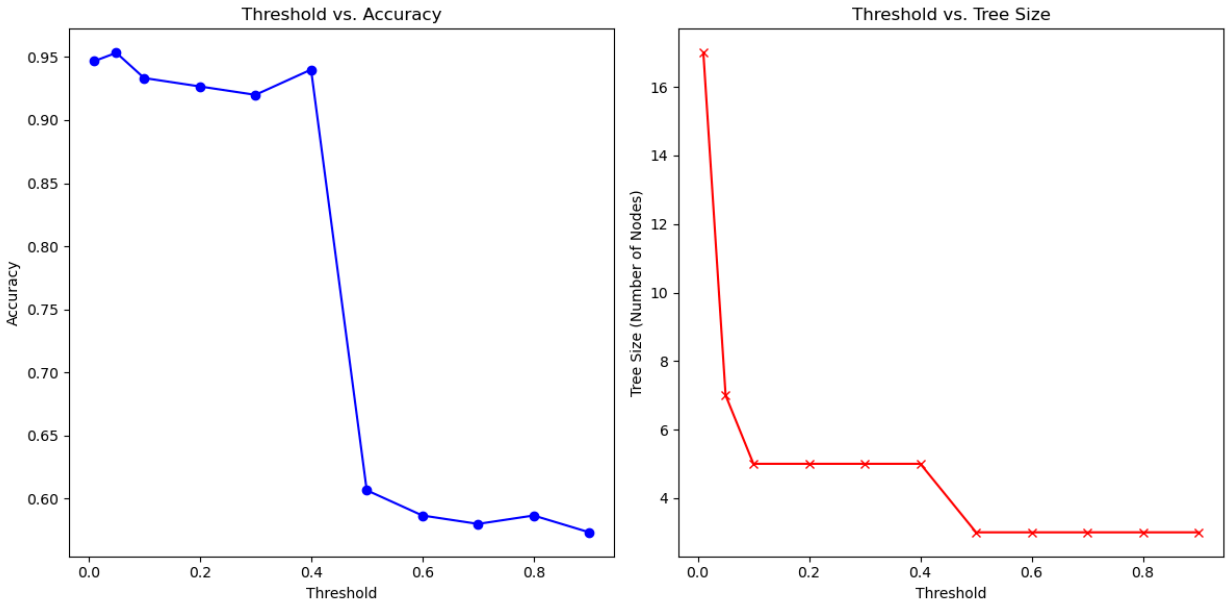


Nursery



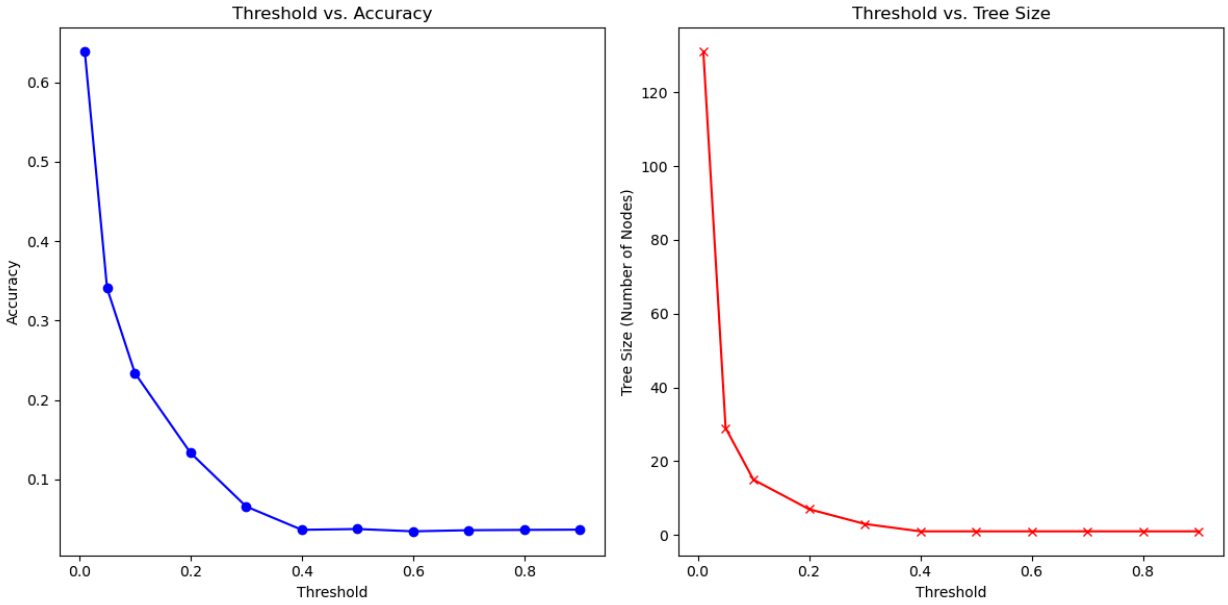
Similar to our implementation, tree size drops very quickly as threshold increases, plateauing at 3 nodes after a 0.1 threshold. Accuracy follows a very similar trend, dropping to a steady 67% a lot quicker than our implementation as the trees get overly simple. This indicates aggressive tree pruning that leads to a lack of model complexity at higher thresholds. Also, note the dramatic difference in tree sizes, with the largest tree produced being around 20 nodes at the lowest thresholds, while our implementation had trees with over 1000 nodes. So while less accurate overall, sklearn implementation is probably much more resistant to overfitting due to its enforcing of low tree sizes even at low thresholds. That being said, the best model has to be at info_gain=0.01, which had a respectable accuracy of 87.3% and a tree size of 22.

IRIS



Similar to our implementation, accuracy remains over 90% until a threshold around 0.5. Tree size is more comparable between implementations than the nursery dataset, but drops much quicker to simple 3 node trees, which is probably what causes the accuracy drop off. Overall, the best tree that presents the best size/accuracy tradeoff is info_gain=0.4, with an accuracy of 94.46% and a tree size of only 5 nodes. This is very similar to our implementation's best model in terms of both metrics.

Letters



This was the most different between sklearn and our implementation. Tree size and accuracy roughly follow the same downward trend, which quickly drops to very simple trees and less than 10% accuracy. The max tree size is only around 130 at a threshold of 0.01, while the max tree size of our implementation is much larger at around 5000. Despite this, max accuracy is around 64% compared to an accuracy of 86% for our implementation. This indicates sklearns implementation is more focused on limiting tree depth/complexity to prevent overfitting, while our implementation has no guards against this and thus prone to overly large trees. The best tree was probably info_gain=0.01, because while having a larger tree size of 125, it had the highest accuracy at 64.2%. This is much different from our implementation's model, which had better accuracy but a tree size of 3119.

Conclusion:

We learned to implement a c4.5 decision tree in python and compared our implementation to SKLearn's, which used CART. Our trees were equal in accuracy or more accurate than SKLearn's, at the expense of larger trees.

Sklearn's implementation was also noticeably faster in terms of runtime, especially when fitting on smaller thresholds (0.05, 0.01, 0.001, etc.). This was expected, due to the SKLearn models being precompiled Cython code and utilizing optimized algorithms and parallel processing.

Despite this, our C4.5 implementation could be significantly sped up in the future. For example, we used pandas dataframes for all operations, with operations like value_counts(), sort_values(), iloc, and unique() all able to be replaced by more efficient NumPy vectorized operations with 5x-10x faster runtimes. While not part of the lab, we also could have implemented some kind of tree pruning or an enforced max tree depth like SKLearn does, which would avoid unnecessary or redundant computations and overly large trees that take a long time to process.