SCC.461 Final Assignment

36155273

2023-01-05

Abstract

This report shows how a decision tree classifier was built from scratch using python, exploring the different methods used to construct a tree and adding extra functionality to allow the use of different types and sizes of datasets. It explains how the user can alter stopping criteria such as the maximum depth, minimum samples to split, and the minimum samples per leaf. The implementation also accounted for different criterion for splits (choice between gini and entropy) and was able to handle categorical data as well as continuous. The classifier was directly compared to the high-performing benchmark of the scikit learn library decision tree implementation, resulting in the original model matching the classification scores. However, sci-kit learn was much more computationally efficient in terms of training time and memory. The effects of parameters and the size of train/test splits on the classifiers was also explored further and the distinctions between the two algorithms were discussed in depth.

Introduction

The aim of this report was to show the theory and application of building a decision tree classifier from first principles using python. It explores how the algorithm was built, discussing the criteria for splitting the data, the process of building the tree, and how all the calculations were performed. Extra parameters that define the stopping criteria of a decision tree were also added and shown in depth. To assess the performance of the tree, the class was applied to three different datasets, all with different properties such as size, dimensionality, and the use of categorical features. Furthermore, the 'sci-kit learn' library version of the decision tree classifier was applied to the same data, and used as a benchmark. The report explores the direct comparison between the two models, assessing the differences in classification performance and computational efficiency. It also highlights how different parameters and the size of the training/test sets affect the performance. Ultimately, the aim was to create a decision tree that could compete with the library implementation. The report will first discuss the methodology behind building the decsision tree and choices made in implementation, and will then explore the results of classification on the different datasets from both models. Finally, there is a discussion of the major differences, positive outcomes, and problems faced from the results and application.

Methodology

The methods used to build the decision tree classifier from scratch using python will be discussed here. A decision tree is built in a top-down approach, finding the best split of the data at each level to maximise the information gained right down to the bottom. When new data is passed through, it traverses the tree, picking the left or right branch each time based on one of the features in the data. Eventually, a leaf node is met which returns the predicted label and classifies the data point. It is an extremely logical method of classification, producing usually reliable and accurate results with great intuition. The process in creating the program was a bottom-up approach:

Finding the Best Split

First, the focus was on how an individual split would be decided, starting with the first split and then generalising. At the beginning, all of the data is inputted; this includes all the features and the

corresponding labels. An algorithm was created to find the best possible way to split the data into two different datasets (left and right branches of the tree). This is decided upon a criterion. The first criterion implemented was the 'gini impurity' but later 'entropy' was also added as an option and is discussed later. The idea of the 'best split' is to find a way to maximise the number of points in each subset having the same labels. It is not always possible to split the data so that, for example in a binary case, each subset has all 0s and the other all 1s. This is why a measurement of the purity of these subsets needs to be used to assess which split produces the 'most pure' subsets/branches. The gini impurity is defined by Ayyadevara (2018), as: $Gini = 1 - (\Sigma_{i=1}^{n} p_i^2)$ where n is the number of classes in that dataset and p_i is the probability associated with each class i. A gini of 0.5 implies that there is maximum impurity and the data consists of an equal proportion of each class. This is not ideal and so the aim is to minimise the gini impurity, as a value of 0 represents complete purity, i.e. every data point belongs to the same class.

The algorithm uses a greedy exhaustive approach. For every feature in the dataset, it sorts the data by that feature in ascending order and finds every single possible midpoint (value to split the data up into 2 different subsets). For each midpoint, the data is split up into the two subsets and the gini index is found for each. A weighted sum of the two values is returned and the smallest proportional gini impurity is kept, eventually returning the best possible split point for the feature. This is done for each feature and eventually the best split from the best feature is kept and returned.

There are of course edge cases and problems that can occur in these calculations so the datasets must be checked continuously to ensure the code runs correctly. The first is the problem of the data only having a single label (every point has the same label). This means that the tree no longer needs to split for classification therefore a leaf node is created. A leaf node is the final node of a branch in a tree and has no other branches. It is assigned a value for the class predicted; in this case, the class will be the single class that every point has. Another problem is the case when there is only one value for the feature (every point has the same value) and so no more splits can be done. A leaf node must be created here and the value assigned is the majority class label in the dataset. There were more problems that came after adding more parameters to the class but are discussed in their relevant sections.

Building the Tree

The decision tree can now be built based off this algorithm working. When the data is split, the new datasets (branches) repeat the same process and find the next best way to split the data. Each time the gini impurity should change, as the branches split the new data in different ways, until a stopping criteria is met. To begin with, this criteria is when the gini becomes 0 and all data points are perfectly classified, but more stopping criteria/parameters were added and are to be discussed later. The branches of the trees and all the information they hold must be stored in a node, a class built for this exact purpose. A build function within the decision tree class takes the original data inputted and recursively calls itself, each time splitting the data into left and right branches that are stored as nodes. Beginning at the root node (top of tree), each node is connected as the algorithm builds the tree, meaning that traversal is easy. Every node that splits the data into more branches is called a decision node and eventually every branch ends with a leaf node that has a predicted class based off all the previous criteria met in the splits. A decision tree will now be built and fitted to any dataset inputted. Additionally, another function was created to allow the user to print the tree in the console so that they can easily visualise how the splits have been made, and on what features. This is a great feature allowing the user to get intuition from the model; something that other classification algorithms do not normally have.

Accounting for Categorical Data

The algorithm for the decision tree works so far with continuous data however it would easily break if categorical features were to be inputted. To account for this, the algorithm was adapted. Firstly, when the model is fit, the type (continuous or categorical) of the features was recorded as an array and to be used throughout as a parameter. Whenever a feature is a string or has less than 20 unique values it is considered to be categorical. This is not completely robust and can cause problems if a feature has more than 20 unique values and is categorical but still works for most cases. When splitting the data on a categorical feature, the possible splits to check are now based on the different unique values in that feature. The splits now produce two datasets: 1. data that has a certain value, 2. they do not have that value. This differs from splitting at a midpoint in the continuous sense and is a lot quicker due to having less values to search through. The split value returned is now simply the value the algorithm

used to split the data based on if the feature was equal to it or not. Whenever a feature is searched for splitting, the algorithm checks the type of feature first based on the array created at the beginning. No other calculations need to be altered as the splits perform the same, and all parameters implemented did not need special consideration for the type of data.

Predicting New Data

The prediction function implemented was simple due to the nature of how the tree was built. Every node was connected from the root node so the predict function passes through each data point into the tree, and the tree is traversed based on the individual feature values. Eventually, a leaf node will be met and the label assigned to that point is the class value of the leaf node. This is completed for all the data points and a list of predicted labels based on the training data is returned. An accuracy score function was also implemented that simply predicts the labels using this method and then compares the labels one by one to the true labels (also inputted), recording how many were correct. A proportion of the total correct predictions is then simply returned.

Parameters

Entropy & Information Gain

The code was adapted such that an additional parameter for the 'criterion' could be passed through. This gives the user a choice of the criteria for splitting the data using gini index, as explained above, or entropy. Entropy is a different calculation to gini and is as follows: $Entropy = -\Sigma_{i=1}^{n} p_i log_2 p_i$. Similar to gini impurity, this value is calculated for the left dataset and the right dataset and a weighted average is taken. This value is useless on its own however and needs to be applied to make a new value called the information gain. This is produced from entropy of the parent node minus the weighted average of its child nodes. It gives a statistic $>= 0$ that resembles the amount of new information gained by taking the split and so the aim is to maximise this value. In the code however, the best split finding algorithm works on the assumption that the criterion (gini) is reducing and aims to minimise this value at each split. To overcome this, the negative of the information gain is passed through instead, keeping the algorithm does not need to be changed. The values stored are negative and so whenever they are printed to the screen, the negative is taken again to show the true gain value. This may not be the most suitable way to program the algorithm but did mean barely any changes needed to be made in what was a perfectly working structure.

Maximum Depth

The depth of a decision tree is the length of the maximum path from the root node to a leaf node. By setting the maximum depth of the tree at the begging of training, as a parameter, the tree will only grow to that desired depth. This was an important feature to add because it allows the user to customize their tree to save on time to train for example, or even give more generalisability by restricting fitting so the data will not be overfit. The maximum depth is a stopping criteria and so was added to the build function. At each new branch, the depth was updated by adding one, and if the depth was greater than the max, no more splits occur and a leaf node is instead created.

Minimum Samples to Split

Another parameter included was the minimum number of samples in a node's dataset for a split to be possible. As this was another stopping criterion, if the number of data points exceded this parameter then no more splits would occur and a leaf node is created. Both the 'max_depth' and 'min_samples_split' criteria must be satisfied to continue splitting.

Minimum Samples per Leaf

Finally, another logical stopping criteria was added for functionality, focusing on the number of samples in a leaf node. This provided more difficulty in implementation because the size of the left and right datasets needed to be taken into account when finding suitable midpoints for splitting. If either dataset was too small then that midpoint was not considered and therefore, if no suitable midpoints were found, the splitting would stop and a leaf node would be created again. This parameter is useful as it allows the

user to make the classification more generalised and ensure that leaf nodes have enough data in them to give suitable confidence of the assignment of a label.

Results

Three different datasets were used to evaluate the classification performance of the decision tree, with a comparison to the 'scikit-learn DecisionTreeClassifier' library version. This library is widely used and considered to be one of the best performing and adaptable implementations of the decision tree. The model implemented from scratch is referred to as the 'original model' and the sci-kit learn version is called the 'sklearn model'. Machine learning aspects of classification such as accuracy, precision, recall and F1 score are compared between the models when fit to the data, as well as computational aspects such as training time and memory use. It is the very high benchmarks of the sklearn library that the model implemented aims to match. The similarities and differences between the two implementations will be thoroughly discussed, highlighting the limitations of each in context of the datasets. The first dataset used was the simple 'Iris' dataset that aims to classify the type of plant, the second was the much larger 'Wine' dataset that aims to classify the region of wine, and the final 'Adult' dataset that aims to classify if a person has a salary over $50k and includes several categorical variables. The datasets cover a range of different criteria and give a thorough representation of performance for analysis. Since a decision tree can take multiple different parameters that will affect the fit of the tree to a dataset, a range of sensible values for each have been passed through and the following results have been recorded. The maximum depth, minimum number of samples per leaf and minimum number of samples to split have all been ranged. The proportion of the data used for training also affects performance therefore different train-test splits were also used and analysed. To evaluate classification performance, sklearn's in-built functions for accuracy, precision, recall and F1 score were used on both datasets after predictions were made. The 'time' and 'malloc' libraries in python were also used for time and memory performance. Each dataset's results is discussed below.

Iris Dataset

This dataset is extremely simple and widely used for classification problems and hence gives an excellent benchmark for the decision tree model built. It is comprised of four continuous features and three different labels, simply recorded as 0,1,2, for the target variable. It is therefore a multi-class classification problem so it tests if the model can handle this data and how well it performs. There are only 150 rows in this dataset and so computation is relatively low, however there are potential problems with such a limited size when it comes to training, especially when parameters on the leaf nodes are changed to higher values. The classes in this dataset are very distinct and seperated hence classification should be very accurate.

Firstly, the models were both built without any restrictions on the parameters. This meant that the default values were assigned (criterion='gini', min_samples_leaf=1, min_samples_split=2, max_depth=unrestricted) and the tree would train the data, classifying it in the best possible way. This gave a good understanding of how the tree was built in both the implementations and would theoretically fit the training data perfectly as the depth of the tree was not limited by any parameter. Despite this, it is by far not the optimal solution because, in general, the data is overfit and so when applied to new test data, classification accuracy can suffer. For this example also, 20% of the data was randomly removed for testing; a universally accepted way to split the data for performance evaluation.

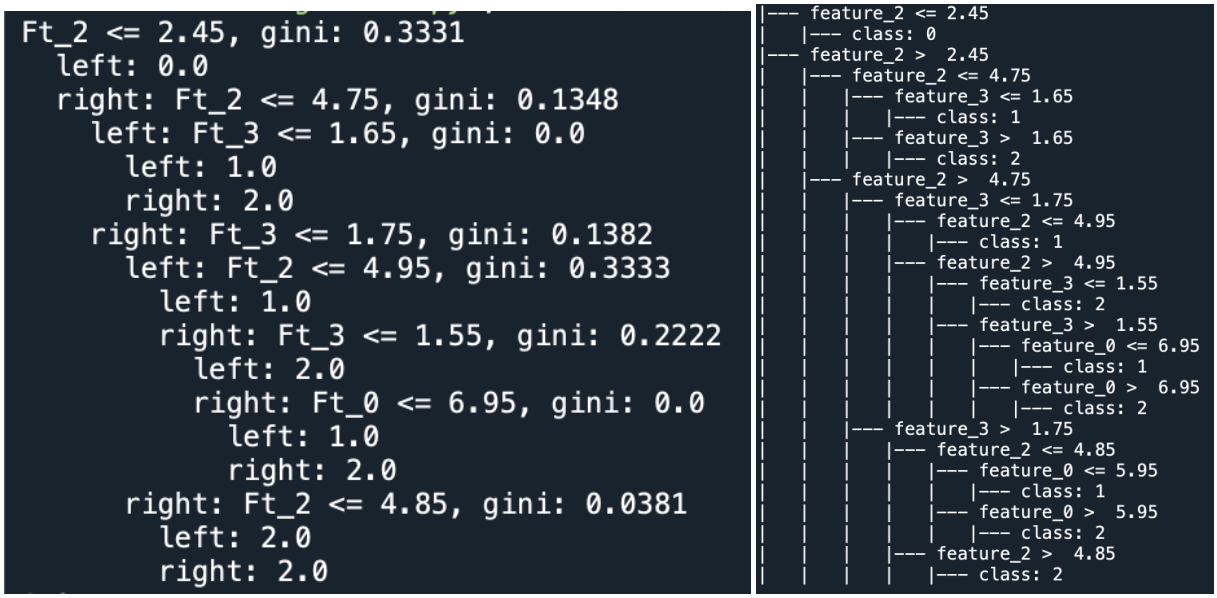

Figure 1: Gini Impurity Trees for Original (left) and sklearn (right))

Despite the slight difference of output from the respective print methods between the classes, the decision trees actually produce the exact same splits. The most important feature was feature 2 at the root node. Both models produced 100% accuracy on the test data also, so they generalised very well. This countered the initial assumptions but it was likely due to the dataset being very small, and so with larger data or the use of k-fold validation, this accuracy would drop. The main difference in results initially, is the notable gap in computational performance. The model implemented took approximately 0.46s to train, taking up 72000kB of memory at its peak, whereas the sklearn version was much faster (0.003s) and memory efficient (11000kB). The original model print_tree function does include the gini impurity at each split which can be very helpful in exploring how the data is split; this is an advantage over sklearn's version.

Furthermore, a paired t-test was performed on the default models to test any difference in accuracy. The dataset was split into 10 different folds randomly and then each fold was used to get an accuracy score for each newly trained model. This was done for both models producing accuracy values from the same k-fold splits to then be used in a paired t-test that evaluated the difference in accuracy. The p-value produced was 0.1679, which was greater than the chosen critical value of 0.05, therefore there is no evidence to suggest that there is significant difference in the accuracy of the two models, meaning that the original model matches the accuracy of the sklearn model on this dataset.

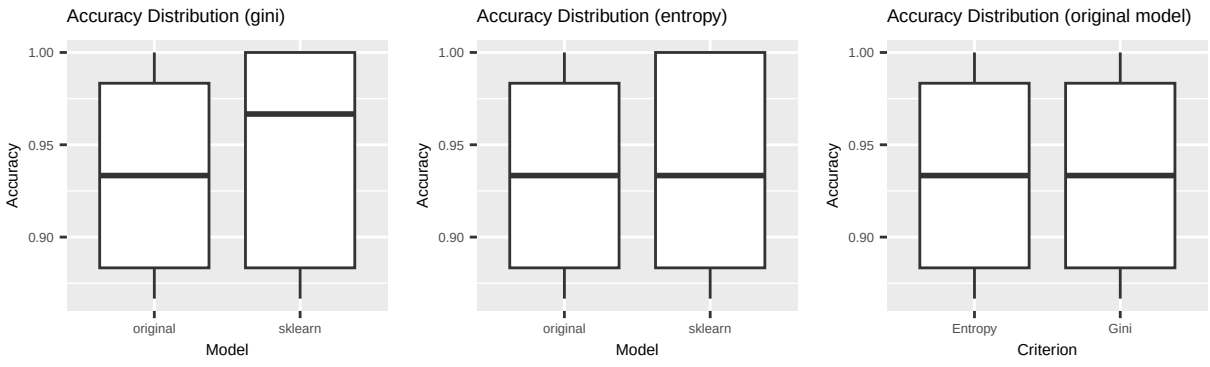

Figure 2: Distribution of Accuracy Scores for t-tests performed

Gini vs Entropy

The criterion parameter can be either 'gini' or 'entropy' and will change the criteria for splitting the data. By running the default parameters again, but with entropy as the criterion, results in the following

decision trees.

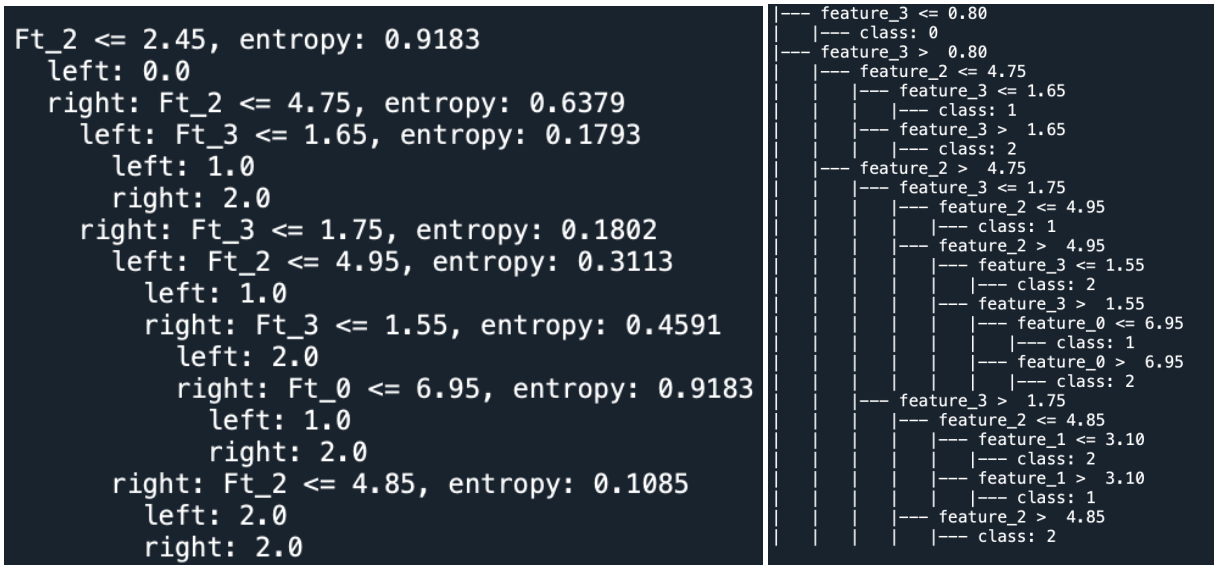

Figure 3: Entropy Trees for Original (left) and sklearn (right))

The accuracy of the classifiers were both again 100% in this case but the most interesting aspect is that the decision trees are produced in the exact same way. In this case, the criterion did not affect the resulting classification however this is not true for all trees; varying the other parameters results in different splits due to the way the different criterion split the data. The original model print_tree function includes an 'entropy' value which represents the information gain at each split and so larger values show extremely decisive splits. The sklearn model replicated the same computational efficiency as in the 'gini' case whereas the original model took double the time to train (0.89s) however did use slightly less memory (67000kB).

The trend was that gini impurity was a more efficient method in the model implemented but machine learning performance was difficult to assess with such a simple dataset, as both algorithms performed very well. In fact, taking the average accuracy of all models, with ranges of different parameters, over reliable test_sizes of 0.2 and 0.3, resulted in gini and entropy having similar accuracies of 67.6% and 67.9% respectively. This shows that there was no affect from the criterion on accuracy. The sklearn model also had the same but higher average accuracies (72.2% for both) showing that the sklearn model was better in general but classification was entirely down to the simplicity of the dataset. The average times for the original model further proved that entropy took twice as long to calculate but memory was equal, although still much worse than sklearn.

Another paired t-test was performed using entropy as the criterion and assessing the difference in accuracy. Again, the null hypothesis was rejected as the p-value was 0.3434, meaning there was no evidence to suggest that the two models had different accuracy when the criterion was entropy. Moreover, another pared t-test was performed between the gini and entropy versions of the original model, to assess if there was any difference in accuracy. The results were exactly the same therefore the mean difference was 0 and there was no difference in accuracy between the two criterions.

Maximum Depth

The maximum depth parameter was also ranged over values from 1 to 5 incrementally, keeping the other parameters fixed to their default values.

In terms of classification scores, the original model matched the scores of the sklearn model at every depth. The scores all rise sharply with increased maximum depth but once the models reached 3, the scores stayed constant at 100% for every max_depth after. This is due to the depth of the tree clearly only needing to be 3 for perfect classification and further increase of the max_depth parameter has no effect. The precision initially starts extremely low at less than 50% however increases dramatically as the depth increments to 2. The match in the scores between the two models shows that the original model

Figure 4: Metrics for Max Depth

has signs of excellent machine learning performance. However, the computational efficiency of the two algorithms is vastly different. For the original model, the training times and memory show much higher values in comparison. The times are still very quick but the sklearn library clearly has extremely efficient processes behind it. The maximum depth does, as expected, increase training time as more processes are being run as the tree gets larger. Interestingly, the sklearn model has no increase in memory or time to train, and the original model's memory use still oscillates about a mean showing memory use is not affected as much as training time. A max_depth of 3 was clearly the best choice here if optimisation was the priority, as it maximises classification for better computational efficiency.

Minimum Samples to Split

Again, to show the effect of this paramater, all others were left to their default (except max_depth which was left at the maximum of 5) and the min_samples_split was ranged over the values of 2, 10, 20, 50, and 100.

Figure 5: Metrics for min_samples_split

The original model again had the exact same classification performance as sklearn and so varying the min_samples_split parameter worked perfectly and as expected. This paramater is inversely proportional to the max_depth in the sense that increasing it, decreases the classification scores. All scores follow the same trend of sharp decreases as the parameter input gets larger. This is because the parameter is a stopping criteria for the build function in the decision tree class, and so when this value is large, the number of samples needed to make a split is large and so this criteria is met much earlier in the tree splits. The value of 100 gives very low scores in comparison and this is very dependent on the size of the original dataset. Since the number of samples originally in the tree is 130, after splitting into training and testing sets, the stopping criteria is met very quickly, resulting in a smaller depth tree. This parameter can be varied to give better generalisation performance in larger datasets however the user must be careful to not make the value too large and resulting in underfitting the data completely, creating a very poor classification performance. The increase in this parameter results in faster training times due to less calculations being performed. The memory usage is again six times that of the sklearn model and the training time is infinitely worse (although still a quick process to humans). This was a continuous pattern throughout comparison. The memory again does not change much, showing that the storage needed in real time does not change depending on the min_samples_split parameter.

Minimum Samples per Leaf

This parameter was altered and tested in the exact same way as previous, keeping parameters default and varying the min_samples_leaf over 1, 10, 20, 50 and 100.

Figure 6: Metrics for min_samples_leaf

The resulting classification scores between the two models were again the same which further shows that the classification power of the model implemented was extremely good. Increasing this parameter massively decreases the scores due to the stopping criteria being met earlier in fitting with higher values. The accuracy drops to 30%, the worst performance after varying other parameters but this is because the value of 100 takes a very large proportion of the dataset and so a split is very unlikely to occur more than once. The guidance on selecting this parameter for optimisation should be the same as the min_samples_split as they are very closely related and both determine leaf sizes. The increase of the parameter does decrease training time again as stopping criteria is met earlier. The time does actually meet the same as sklearns implementation at 100 but this is likely due to no splits actually being made, and min_samples_split would also reach this stage if increased further. Unlike other parameters, the memory usage of the model decreases linearly which is likely due to the placement of the checking criteria within the code. This parameter is checked when deciding possible splits so the actual storage of variables is affected whereas other stopping criteria such as max_depth and min_samples_split are only checked during building and so the possible splits are still always created at each iteration.

Test Size

The proportion of the data that is assigned to testing is of great importance to the performance of the classifier also, as a dataset with more data to train on is likely to perform better on newer data. Equally, too much data can lead to it overfitting and not applying well to new unseen data. To show the effects of test sizes, the proportion was varied on the default parameters of the models. It was important to keep the random seed for splitting the data constant throughout to ensure repeatability in measurements.

Figure 7: Metrics for different test set sizes

The classification scores for the models were the same yet again and did decrease slightly with high test data sizes, although all metrics remained very high showing excellent classification even with less training data. The training times also linearly decreased in the original model as the amount of training data decreased. The memory usage did begin to decrease also like the sklearn model shows however a large

spike was produced at 0.4. This was very unusual but a possible reasoning is that if there less training data then potentially the splits found are not as efficient in terms of information gain. This would mean more splits need to be assessed at lower branches than previous, needing more memory storage. This is just a hypothesis and shows that higher proportions of test data could result in worse performance for the original model. Clearly the test_size of 0.2 or 0.3 give the best trade-off of accuracy and efficiency, and is why these values are often seen as the standard in data science.

Since this dataset was a very easy classification problem, 100% accuracy should be easily possible. After running 1250 models with varying parameters and test sizes (10%, 20%, 30%, 40%, 50%), 126 of the models achieved perfect accuracy (10% of models- shown in Table 1). Sklearn's model had 169 models of maximum accuracy (displayed in Table 2) showing there is some differences in classification scores between the two models for different parameter combinations and test sizes.

Table 1: Original

test_size	n	proportion
0.1	96	0.7619048
0.2	18	0.1428571
0.3	12	0.0952381

Table 2: Sklearn

test_size	n	proportion
0.1	96	0.7619048
0.2	18	0.1428571
0.3	61	0.4841270
0.5	3	0.0238095

For smaller test sizes of 0.1 and 0.2, the original model produced the same number of perfect models due to the training data proportion being larger. However, the sklearn model has more perfect models for larger test sizes showing it has better performance on less data. Taking the models from this list of perfect classifiers with the fastest training times returns a good estimation of the parameters that give the 'optimal' solution to this dataset (shown in Table 3). The memory usage does not vary much for these models hence it was not considered. The max_depth varies and is inconsequential as the actual depth of the top performers were all 2 s taking any value over 2 was fine. The min_samples_leaf is consistently 20 and min_samples_split is consistently 50. These values are the best from the values that were inputted but obviously could be further changed for optimisation. The model clearly does not need small samples of leaves for classification.

Table 3: Best paramaters for optimal performance on original model

criterion	max_depth	Depth	min_leaf	min_split	test_size	Time	Memory	Accuracy
gini	3	2	20	50	0.1	0.2394571	59320	1
gini	4	2	20	50	0.1	0.2398162	59520	1
gini	2	2	20	20	0.1	0.2402010	58816	1

Furthermore, it was interesting to note that the majority of the fastest models were trained on test sizes of 0.1 and so more data can lead to faster training. This can seem confusing at first but the training time is dependent on the quality of splits and, with more data, the splits of the decision trees will be found faster. Although times of 0.24s seem very fast, they are still incomparable to the rapid times of sklearn's implementation. No model took longer than one second to train which meant that the original decision tree implemented had overall very good performance on this dataset. Varying the size of the data is something that is next explored.

Wine Dataset

The wine dataset was used next to assess performance on a different dimension of dataset. It contained a similar number of rows (177) but with 14 features instead of 4. This tested the performance of the models on a larger feature set but the same pattern of results as the iris dataset were observed (plots all very similar to figures 4, 5 and 6). The only difference was that training times and memory use were larger in comparison, and no tree attained 100% accuracy. The sklearn implementation was again immensely quick and only needed a storage of 42000kB throughout. No parameter changes affected it's computational efficiency yet again. As done on the previous dataset, taking the average accuracy of all models, with ranges of different parameters, over reliable test sizes of 0.2 and 0.3, resulted in gini and entropy having

differing accuracy of 67.2% and 62.2% respectively. This shows that gini impurity was the better classifier on this data and suggests that it performs better on more features. The sklearn model also had the same trend but lower average accuracy (66.5% and 62.0%), showing that the sklearn model was worse in general. The average times for the original model further proved that entropy took twice as long (0.73s & 1.45s) to calculate but memory was equal (100000kB), although still much worse than sklearn (36000kB). The impact of other parameters was also explored but graphs are not included due to them all matching the same exact trends as on the iris dataset.

To begin with, increasing the max_depth in turn increased classification performance up to a maximum of 93%. This leveled out at a max_depth of 3, as the tree with depth of 3 was the best performer possible so it did not exceed this. The training times also increased but leveled out again at max_depth=3, and the memory use was a much larger constant at around 175000kB. In comparison, the training times for the different max_depth's varied from 1 to 2 seconds and so computational load was massively increased due to the size of the dataset.

Secondly, the exact same procedures were done for the minimum samples per split and per leaf for this dataset and resulted in the same trends as discussed in the iris dataset but with increased training time and memory use, as expected. The graphs of these parameter changes are not shown since they simply confirmed the theory from the iris dataset. The only observable difference between the sklearn and original model classification scores was surprising. The original model did outperform the sklearn implementation on all metrics on small min_samples_split and max_depth's greater than 3. The differences were marginal (only a few % in accuracy) but evidenced how well the original model had been built for classification.

The test size was also varied for this dataset, keeping all parameters default, but produced more unique results.

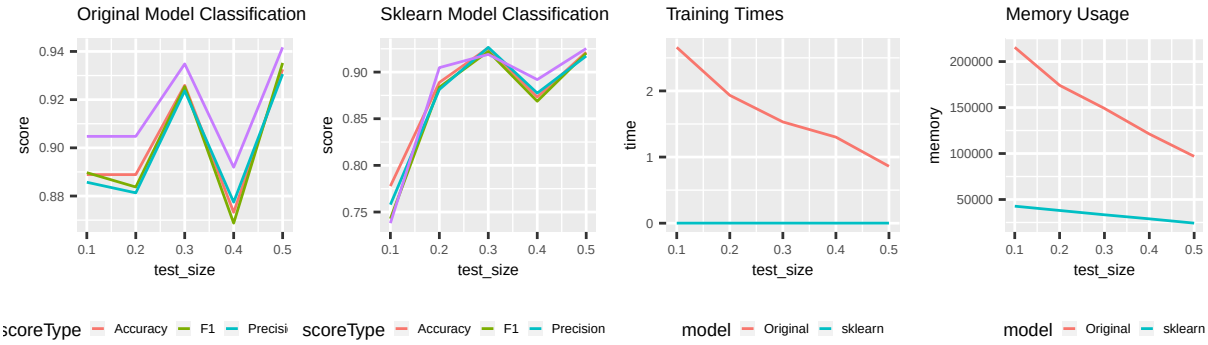

Figure 8: Metrics for different test set sizes

Varying the test_size results in rather random differences in classification scores for the original model. These scores are still very high, showing test size does not have much of an impact with this dataset, probably due to having a larger amount of features. The recall in particular, for the original model stays much higher than other measures (most significantly, greater than the precision), suggesting the model correctly identifies the true classes in this dataset at a high rate. On the other hand, the sklearn model has much lower scores for smaller test sizes and shows an increase as the test_size increases. Again, like the iris data, the time and memory of the models decrease with larger test sizes but the original model was considerably worse again in efficiency.

By taking the models with accuracy greater than 90%, 54 of the original models were found whereas 82 models from sklearn had this high performance. Table 4 and 5 show there was a clear bias towards higher test sizes in this dataset to give better performances.

Furthermore, the table 6 shows the top performing classifiers and gives an excellent idea of the optimal parameters from those tested. It is clear that the depth of the tree does not need to exceed 3, the min_samples_leaf is best at 1 and the min_samples_split is best at 10 or 20. Test sizes of 0.5 also produced of the fastest implementations. Sklearn's implementation (table 7) has major similarities however test sizes of 0.4 are generally better performing and the criterion of 'entropy' is preferred. The maximum accuracy of the model is also slightly lower, despite having more high performing models.

Table 4: Original

test_size	n	proportion
0.3	30	0.5555556
0.4	12	0.2222222
0.5	12	0.2222222

Table 5: Sklearn

test_size	n	proportion
0.2	1	0.0185185
0.3	30	0.5555556
0.4	15	0.2777778
0.5	38	0.7037037

Table 6: Optimal paramaters for original model

criterion	max_depth	Depth	min_leaf	min_split	test_size	Time	Memory	Accuracy
gini	5	3	1	20	0.5	0.8456969	87071	0.9325843
gini	5	3	1	10	0.5	0.8472271	86228	0.9325843
gini	4	3	1	10	0.5	0.8481169	87323	0.9325843

Table 7: Optimal paramaters for sklearn model

criterion	max_depth	Depth	min_leaf	min_split	test_size	Time	Memory	Accuracy
gini	4	3	1	10	0.5	0.0009630	24280	0.9438202
gini	4	3	1	2	0.5	0.0009351	24280	0.9213483
gini	3	3	1	2	0.5	0.0009100	24280	0.9213483

Adult Dataset (Categorical)

The final dataset looked at was the adult dataset. It is a binary classification problem with 14 different features and over 32000 rows. This is by far the largest dataset but it's purpose was to show how the models dealt with categorical features. Sklearn's implementation does not have an adaptation to categorical data so the model could not run at all (unless the user pre-processed the data themselves but this wasn't explored). The original model, as mentioned, could handle it however the model was clearly not very computationally efficient and struggled with even a few hundred rows with this many features (wine dataset example). Therefore, only the first 1000 rows of the adult dataset were taken to reduce run time but still making it a larger dataset. The following trees were produced for gini and entropy models with default parameters:

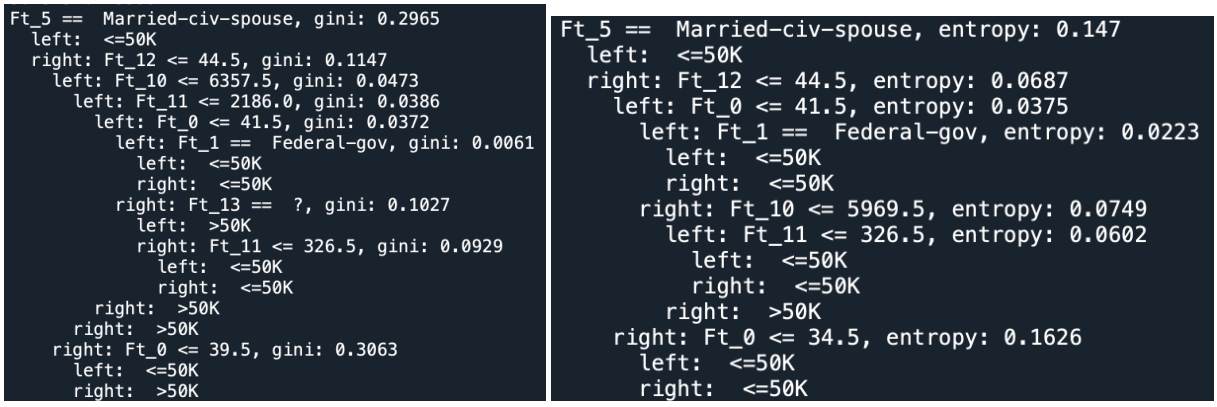

Figure 9: Trees for Original with gini (left) and entropy (right))

There are clear uses of categorical features at different splits, such as feature 5, 1 and 13, proving the model is adaptable to different types of datasets. Despite this adaptability, the training time for this model was around 30 seconds and the memory taking up a ridiculous 5.5GB for the 'gini' version showing that the original model is not efficient with larger datasets. The 'entropy' model took even longer (47s) with similar memory use. The accuracy produced for 'gini' was 76% whereas the 'entropy' model performed

better with 81% accuracy. These seem like good scores but it is difficult to understand if this is truly the best on the data when there is no benchmark possible from sklearn. Repeatability and testing of other parameters was difficult due to the long training times, so this must be improved first if more exploration was wanted.

Discussion

The main objective of this report was to build a decision tree classifier from scratch using python and compare it to the known library version (sklearn) to evaluate how it performed against this benchmark. Overall, classification on the datasets used was excellent, keeping the same standard, if not higher, accuracy scores as sklearn's benchmark. Precision, recall and F1 scores for most part also matched sklearn's performance showing the algorithm was well written and exhaustive in its training of the data. It was shown that on the Iris dataset for example, the exact same splits on features were taken as sklearn and produced identical trees at times. Of course this was not always the case, as parameters changed, showing there were different nuances between the algorithms that resulted in different constructions. This would need to be explored further to see if it was due to the way sklearn's implementation was written or potential bugs and problems in the original code.

In terms of computational performance, there was no competition between the algorithms. Sklearn clearly has an extremely efficient algorithm as all datasets were trained in less than 0.01 seconds. In comparison, the original model took 1 to 2 seconds even on small datasets and as the datasets scaled in size, performance became much worse which is consequential for any data scientist wishing to use it. The memory use of the original became very large again with more data however was not too inefficient at smaller scales in comparison. Both models generally kept constant memory and was based off the size of the dataset generally and not on the tree however sklearn still was much more efficient. The efficiency of sklearn was something that could be researched further, to understand how it performs so well and take ideas to improve the original implementation.

Furthermore, different ranges of input parameters impacted the two models in the exact same way. In general, increasing the the max depth increased the classification scores up to a limit where they plateaued as the depth of the tree was no longer affected by the max depth parameter. The opposite was the case for the min_samples_split and min_samples_leaf variables, both affecting the sizes of leaf nodes, where increasing them in turn decreased the classification scores exponentially. Decreasing depth or increasing the leaf parameters both decreased the number of calculations needed in the algorithm therefore the training time for the original model decreased also, however sklearn was not affected by this. The memory use of the original model was only affected by the size of the dataset, decreasing with less training data. This could be because the algorithm was efficient in memory use or could be a problem with the recording, and so more analysis would be required.

Conclusion

Overall, the original model was excellent on small datasets, especially in terms of classification, however, as this scaled, the training times and memory usage becomes too much of a problem. To overcome this hurdle, the code must be looked at again to analyse where there are memory leaks or potential problems in the implementation. Immediate theories are that there are too many for loops in the code leading to high order time complexities when functions are called iteratively. With more time, an analysis of the functions could be done to remove repeated calculations and streamline the splitting process. There were also clear inefficiencies in the code that were produced by patching up certain problems during testing, which could be improved and could help performance. Examples of this was switching continuously between numpy arrays and 2D arrays, and 'if' statements throughout to make sure the model did not break at certain cases. There are many more questions to be asked: does the model need to check every single possible midpoint for splitting?, can the same accuracy be achieved using a different method?, and can the 2D arrays used for sorting be replaced with more efficient storage such as numpy arrays or pandas data frames? The implementation tried not to rely too heavily on other libraries to ensure it was original but this could be something explored in order to improve the efficiency also. In conclusion, the difference between the two models was clearly down to their computational efficiency as the choice of parameters were optimised in the same way and the test size did not have much effect on classification. The algorithm itself created must be optimised to become a viable alternative to sklearn.

Acknowledgements

I would like to acknowledge the help of several different online resources to help guide the build of the python code for the decision tree. Help was also gained for Leandro Marcolino's SCC461 Coursework 9 Decision Tree solutions to begin the structure of the code. All links to the resources used are contained in the seperate statement.pdf file submitted.

Bibliography

Ayyadevara, V Kishore. 2018. "Decision Tree." In *Pro Machine Learning Algorithms*, 71–103. Berkeley, CA: Apress.