Whitebeam - A simple decision tree library

SCC461 Final Assignment

35762970

Abstract

whitebeam is a python module integrating decision tree classifying algorithms for small to medium scale supervised binary problems. This package focuses on bringing machine learning to non-specialists wishing to implement their own variants of trees using a general-purpose high-level language. Emphasis is put on ease of use and API consistency with established package scikit-learn. Additionally a Particle Swarm Optimisation metaheuristic is used to determine if accuracy can be increased then compared against library implementations from scikit-learn. whitebeam performs well, and achieves similar accuracy scores in less time for some datasets.

1 Introduction

A decision tree is a recursive data partitioning algorithm, where the objective can be to maximise or minimise some criterion such as accuracy or mean squared error. Generally decision trees perform strongly with classification problems and also regression problems, some well known algorithms include CART (Classification and Regression Trees), which is the method applied with `DecisionTreeClassifier` in sklearn, C4.5 and ID3, both created by Ross Quinlan and also Chi-square automatic interaction detection (CHAID), which is a lesser known algorithm based on Bonferroni testing.

2 Methods

In this research there are two frameworks defined; The first method is whitebeam is a decision tree framework, based upon Attribute Value Classes, created in python, with the core created in cython for increased performance. The second is a framework for allowing decision trees with metaheuristic optimisation by particle swarm optimisation, which has been proven to be more competitive in some cases than other genetic algorithms, Sousa, Silva, and Neves (2004) and Falco, Cioppa, and Tarantino (2006).

2.1 Basic Concepts

- Nodes - There are 3 types of nodes,
  - Root Node / Decision Node - A choice that results in a split
  - Internal Node - A possible split
  - Leaf Node - Final classes / events
- Branches - Branches represent outcomes or occurrences that occur from root and internal nodes. A decision tree is formed using a network of branches, connecting the nodes from top root to leaf nodes, where each different path represented a different classification.
- Splitting -
- Stopping - Complexity and robustness are competing characteristics of models. The more complex a model is, the less reliable it can be when used for fitting unseen data. A stopping condition must be implemented to prevent overfitting. Common stopping conditions are (a) maximum depth of the tree (b) the size of the leaf (c) the size of the current root node

Table 1: Comparison of common decision tree algorithms

Method	Splitting Criteria	Pruning Strategy	Dependent Variables	Input Variables	Split Type
CART	Gini Index	Pre-Pruning	Categorical/ Continuous	Categorical/ Continuous	Binary
C4.5	Entropy	Pre-Pruning	Categorical/ Continuous	Categorical/ Continuous	Multiple
CHAID	Chi-square	Pre-Pruning	Categorical	Categorical/ Continuous	Multiple
QUEST	Chi-square, ANOVA for continous	Post-Pruning	Categorical	Categorical/ Continuous	Binary

- Pruning - For some datasets stopping conditions are ineffective, and instead a pruning method can be implemented to fully grow a tree (allow a tree to be overfit) and then prune it to optimal size by removing nodes that provide less information

In the pseudo code block below, `x` and `y` represent a feature matrix and target vector. Then you test if this node has any children, or if it is a leaf node - `is_leaf()`, if this function returns `True` then the recursion ends and the average value of `y` is returned. If `is_leaf()` returns `False` then a splitting point is found using `find_split()`. The best splitting point is found and then the dataset is partitioned into two disjoint sets - less than, and greater than the splitting value. This is performed recursively until all nodes either have a left child and right child or are a leaf node.

```
def decision_tree(x, y):
  # Check if the node is a leaf
  if is_leaf(x, y):
    # If it is a leaf, return the mean
    return mean(y)

  # If it is not, split at this node
  split = find_split(x, y)
  # Get each side as its own new tree
  x_r, x_l, y_r, y_l = partition.data(x, y, split)

  # Save the splitting node, and recurse the left and right tree
  return split, decision_tree(x_r, y_r), decision_tree(x_l, y_l)
```

Some assumptions can be made upon the functions shown in the pseudo-code.`is_leaf()` dictates all stopping conditions, and as such determine when a tree stops growing my modulating the maximum depth or minimum node size. `find_split()` dictates which variable and at which value a split should be created. With C4.5, using information gain to determine splitting. Where as CART uses the gini impurity measure and the minimum variance criterion for classification and regression respectively.

2.2 Learning Metrics

Decision Tree Algorithms take a top-down approach, and work by choosing a variable at each iteration that best splits the current disjoint set. This can be hard to do, since the algorithm is not always informed (it cannot always see the big picture). Generally, the metric is based upon the homogeneity of the target variable within the disjoint set, then combined to provide a measure of the split.

2.2.1 Gini Impurity - CART

The Gini Impurity measures the probability that a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of the labels in the subset. If we have C classes and $p(i)$ is the probability of picking a datapoint of class i, then the gini impurity is calculated by

$$G(p) = \sum_{i=1}^{C} \Big[p_i \cdot (1 - p_i) \Big].$$

However generally, this is applied with

$$G(p) = \sum_{i=1}^{C} \left(p_i \cdot \sum_{k \neq 1} p_k \right)$$

which can be further generalised to

$$G(p) = 1 - \sum_{i=1}^{C} p_i^2$$

as implemented in sklearn's `DecisionTreeClassifier()` (Pedregosa et al. (2011)). ### Information Gain - ID3, C4.5

Information gain, or Shannon Entropy, is the concept of entropy applied to decision tree learning. That is, the expected information gain is the mutual information, therefore, on average, the reduction in the entropy of the parent is the mutual information. Entropy is a complicated topic founded in information theory, and so will not be discussed here, however the general equation that is used

$$G(p) = -\sum_{i=1}^{C} p_i \log_2 p_i - \sum_{a} p(a) \sum_{i=1}^{C} -Pr(i|a) \log_2 Pr(i|a)$$

although it must be noted that the first coefficient of this equation carries more significance than the second coefficient.

2.2.2 Minimum Variance Criterion - CART (Regression)

Although regression is not necessarily considered within this report, some thought is given to the design of metric implementation and some regression will be supported.

3 whitebeam-1.1

whitebeam is a programmable decision tree framework created in python, with the core created in cython for increased performance. Expandability was the focus when creating this framework, with various basic decision tree methods already implemented, and the ease in which more can be added. The project can be obtained from PyPi at `https://pypi.org/project/whitebeam/` and installed through `pip install whitebeam`. It can also be built from source by cloning the github repository `https://github.com/K-Molloy/whitebeam/`. The only requirements for the library are `NumPy`, `Cython` and `joblib`. whitebeam is influenced by Park and Ghosh (2014), Park, Ho, and Ghosh (2016).

3.1 Implementation

The implementation takes heavy influence from Rainforest (Gehrke, Ramakrishnan, and Ganti (2000)) and builds upon the algorithm defined in Sec.3 Pg.135. In short, the greedy top-down classification tree induction schema is refined to a generic Rainforest Tree Induction Schema, then algorithms for missing values is discussed and extension to regression trees.

The Attibute-Value, Classlabel (AVC) is the key parameter and stores all required information. The formal definition, assume that the domain of the class label is the set $\{1, \ldots, J\}$, formally $\text{dom}(C) = \{1, \ldots, J\}$. Letting $a_{n,X,x,i}$ be the number of records t in F_n with attribute value $t \cdot X = x$ and class label $t \cdot C = i$.

$$a_{n,X,x,i} = \{t \in F_n : t \cdot X = x \wedge t \cdot C = i\}$$

Letting $S = \text{dom}(X) \times \mathbb{N}^J$. Then,

$$\text{AVC}_n(X) = \left\{(x, a_1, \ldots, a_J) \in S : \exists t \in F_n : (t \cdot X = x \forall i \in \{1, \ldots, J\} : a_i = a_{n,X,x,i})\right\}$$

Practically, this is implemented as a `numpy` array that contains 10 columns, and J rows, each column is described in Table 2

Table 2: Attribute Value Classlabel (AVC) indices and description

Index	AVC Description
AVC[:, 0]	AVC indices
AVC[:, 1]	Variable indices
AVC[:, 2]	Split values
AVC[:, 3]	Number of Samples at a hypothetical left node
AVC[:, 4]	Sum of 'y' at left node
AVC[:, 5]	Sum of 'y^2' at left node
AVC[:, 6]	Number of Samples at a hypothetical right node
AVC[:, 7]	Sum of 'y' at right node
AVC[:, 8]	Sum of 'y^2' at right node
AVC[:, 9]	Missing Value Inclusion

3.1.1 whitebeam core

The core of whitebeam is based in cython, which is a complied language that is used to create CPython extension modules which is coded in python-like style, then automatically wrapped in interface code, producing code with significantly less computational overhead at run time. The cython modules have 3 functions; `reorder()`, `create_avc()` and `apply_tree()`. These are all array based functions which are called multiple times per function and benefit from the acclerated performance. The cython functions require handler functions that can be exposed to python in the form of module imports

```
# python exposed function
def reorder(X, y, z, i_start, i_end, j_split, split_value, missing):
    return _reorder(X, y, z, i_start, i_end, j_split, split_value, missing)

# cython private function
cdef size_t _reorder(
        np.ndarray[DTYPE_t, ndim=2] X, # X: 2-d numpy array (n x m)
        np.ndarray[DTYPE_t, ndim=1] y, # y: 1-d numpy array (n)
        np.ndarray[DTYPE_t, ndim=1] z, # z: 1-d numpy array (n)
        size_t i_start, # i_start: row index to start
        size_t i_end,  # i_end: row index to end
        size_t j_split,  # j_split: column index for the splitting variable
        double split_value,  # split_value: threshold
        size_t missing):
```

These functions are imported into the main `Whitebeam()` class - the base tree method, all other trees override this class - which implements the default methods described in Table 3 - this lets the base methods, such as CART, implement the `find_split()` and `is_leaf()` functions and import those into the whitbeam super class at initialisation.

3.1.2 Base Methods

whitebeam has been designed to closely mirror the implementation of sklearn, notably, fitting a decision tree in whitebeam uses the same keywords externally, and mostly also internally. This simplifies the learning curve of the library and also allows some compatibility between them, Buitinck et al. (2013).

```
    model = DecisionTreeClassifier()
    model.fit(X_train, y_train)
    y_hat = model.predict(X_test)
```

However, although endpoints share naming conventions, the method applied within whitebeam is very different to that applied in sklearn whereby sklearn takes the traditional strategy of having a base tree

Table 3: Whitebeam class methods and descriptions

Method	Description
get_avc()	Calculate AVC matrix
split_branch()	Splits the data (X, y) into two children
grow_tree	Grows a tree by recursively partitioning the data (X, y)
fit()	Fit a tree to the data (X, y)
predict()	Predict y by applying the trained tree to X
init_summary()	Initialise the 'image'
set_summary()	Set the 'image'
get_summary()	Get the 'image'
is_stochastic()	Check if subsample is stochastic
get_mask()	Get mask array
get_oob_mask()	Get mask array for OOB samples
get_ttab()	Get tree tables
dump()	return a tree as text
load()	load a tree (from dump)
get_sibling_id()	return sibling id for leaf id
get_sibling_pairs()	return array of sibling pairs
get_feature_importances()	return feature importance
update_feature_importances()	find feature importance

method, whitebeam is based upon AVC. Furthermore, whitebeam provides a series of ready-provided functions such as CART, C4.5, Friedman and XGBoost of which CART and C4.5 are based upon a minimal cost-complexity pruning model. This approach can be used ordinarily to prevent overfitting (as described in Chapter 3 Breiman et al. (1983)) The complexity parameter, α is used to define the cost-complexity measure, $R_\alpha(T)$ of any given tree T:

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|$$

where $|\tilde{T}|$ is the number of terminal nodes in T and $R(T)$ is traditionally defined as the total missclassification rate of the terminal nodes. (Sklearn uses the total sample weighted impurity of the terminal nodes instead).

3.1.3 Ensemble Methods

The goal of ensemble methods is to combine the predictions of several base estimators built with a given algorithm in order to improve generalisability / robustness over a single estimator. There are two categories of ensemble methods:

- Averaging - build several estimators independently, then average the response (this also has the effect of reducing variance)
- Boosting - build sequential estimators, reducing bias in a combined estimator.

whitebeam provides two ensemble methods; random forest and Stochastic Gradient Boosting. A random forest algorithm is a perturb-and-combine technique specifically designed for trees. It is an averaging technique and a diverse set of classifiers are created by introducing randomness and by reducing variance the final prediction is strong. The Stochastic Gradient Boosting approach, as defined by Breiman (1996), originally designed for regression trees, construct additive models by sequentially finding a simple parameterised function and its associated gradient of a loss function. With his randomness addition, Friedman (2002) modified the randomness procedure by taking a random subsample of training data at each iteration instead of the full sample to train the base learner. Letting $\{y_i, X_i\}_1^N$ be the entire training data sample and $\{\pi(i)\}_1^N$ be the random permutation of the integers $\{1, \ldots, N\}$. Then a random subsample of size $\overline{N} < N$ is given by $\{y_{\pi(i)}, X_{\pi(i)}\}_1^{\overline{N}}$.

3.2 Testing

The library was developed with testing fully in mind, with the intention that full version control with continuous integration in place. Additionally, using an approach of written unit tests and integration tests. Enforcing automation in building and testing bridges the gap between development of features and operation activities. This process is commonly known as CI/CD (Continuous Integration and Continuous Delivery).
A full suite of tests have been created to test all functions, including their input and expected output. Each dot in figure 1 represents a test pass, an F would represent a test fail, 106 passes and 54 warnings can be seen (all warnings for the same numpy deprecation warning)

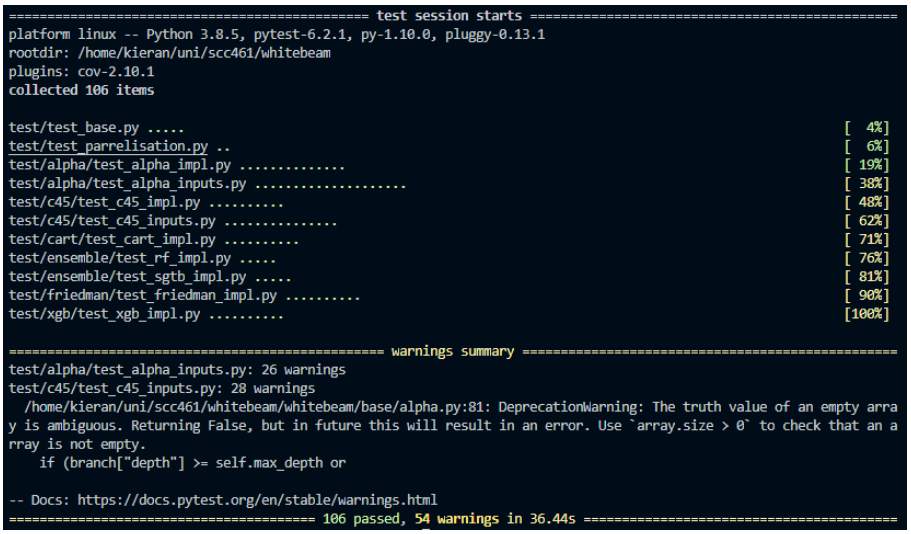

Figure 1: whitebeam pytest console output

3.2.1 Continuous Integration

Continuous testing is the process of periodically executing automated tests as a part of the build pipeline, obtaining immediate feedback on the test output. Usually it encapsulates installation from nothing, to execution of all test packages. whitebeam uses Travis CI to perform continuous integration (`https://travis-ci.com/github/K-Molloy/whitebeam`), this tests the installation procedure and runs all tests for Python versions `3.6` to `3.9-dev`. In Figure 2, the build history for the github repo `K-Molloy/whitebeam` is shown, with the branches main, v1.0.3, v1.0.0 and poetry-conversion all showing as currently passing, although some fails can be seen. It is important that all tests are passed before merging branches as this is a sign of incompatibility.

3.2.2 Version Control

whitebeam uses git as its version controller, with regular commits with upcoming features as well as significant release versions and latest stable version - this allows feature prototyping and accountability of introducing bugs / broken code.

3.2.3 Code Coverage

Test coverage is a measure used to describe the degree to which the source code of a program is executed when a particular test suite runs. A program with higher test coverage score has had more of its source code used/executed during testing, which suggests it has a lower chance of containing undetected bugs compared to one with a low test coverage score. whitebeam uses CodeCov (https://codecov.io/gh/K-Molloy/whitebeam/) to test coverage, of which it currently has a 77.36% coverage. This is an okay score for a small project, however aiming for above 95% would be considered good, and above 98% an excellent score.

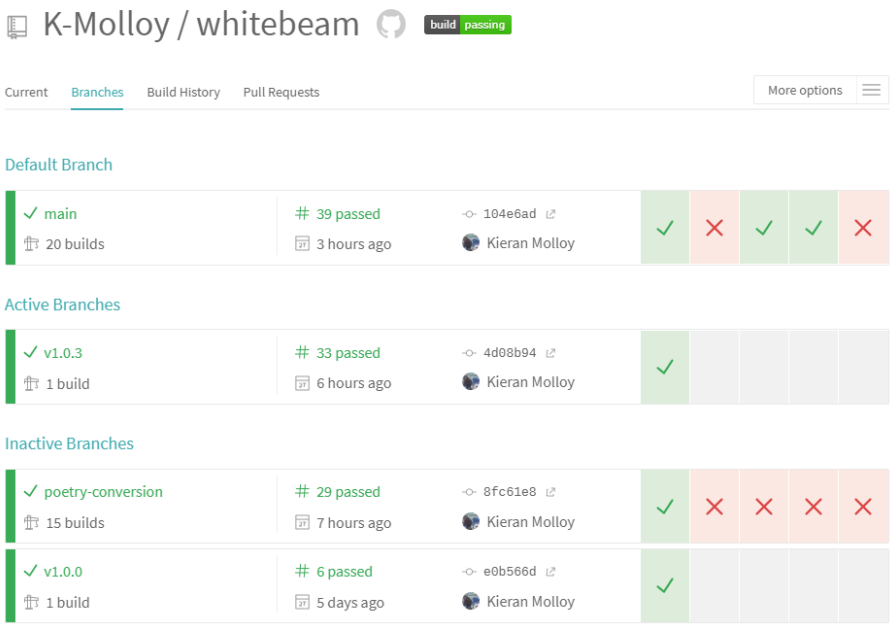

Figure 2: whitebeam travis ci build history

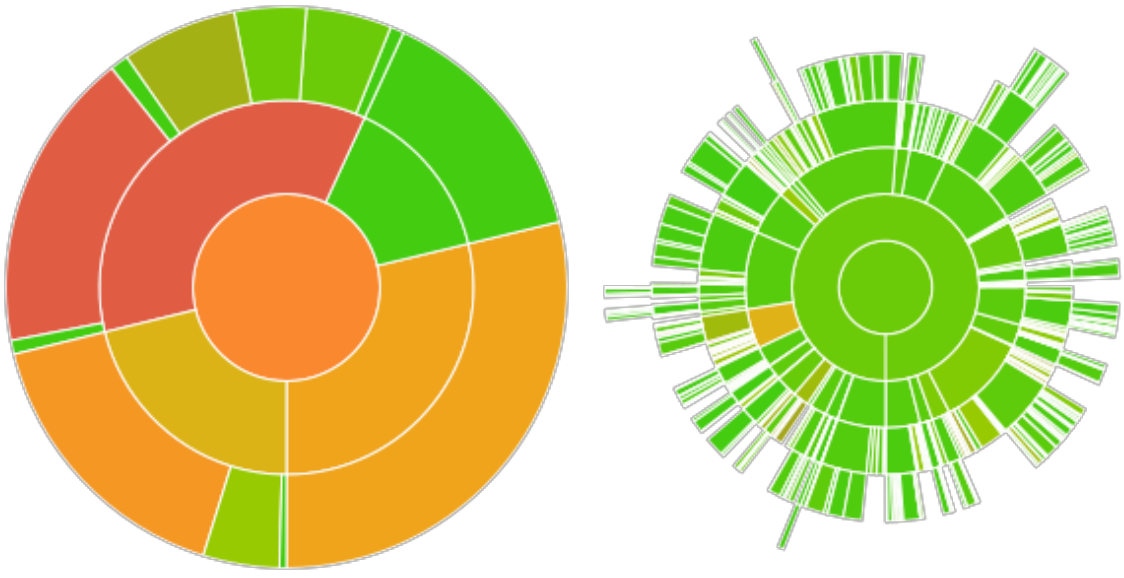

Figure 3: Coverage Sunburst graph for 'K-Molloy/whitebeam' on the left with a coverage of 77.36%, and 'scikit-learn/scikit-learn' on the right with a coverage of 98.18%

3.3 Examples

Following the same design as sklearn, calling the whitebeam methods is simple. For example, with the `DecisionTreeClassifier()` this mirrors the process in sklearn

```
# Initialise
model = DecisionTreeClassifier()
# Fit
model.fit(X_train, y_train)
# Predict
y_hat = model.predict(X_test)
```

Or for an ensemble method, `StochasticGradientBoostedClassifier()` where all parameters are optional but the code snippet demonstrates some user-defined parameters

```
# Initialise
model = StochasticGradientBoostedClassifier(
          distribution="bernoulli",
          n_estimators=200,
          learning_rate=0.4,
          max_depth=5,
          subsample=0.7)
# Fit
model.fit(X_train, y_train)
# Predict
y_hat = model.predict(X_test)
```

All whitebeam methods are applied in the method applied above, below is a list of all methods (as of v1.0.3).

- `DecisionTreeClassifier()`
- `C45TreeClassifier()`
- `CCPTreeClassifier()`
- `FriedmanTreeClassifier()`
- `XGBTreeClassifier()`
- `DecisionTreeRegressor()`
- `RandomForestEnsemble()`
- `StochasticGradientBoostedClassifier()`

4 Particle Swarm Optimisation (PSO)

Evolutionary computation techniques such as genetic algorithms and evolutionary strategies are influenced by traditional evolution seen in nature. The concept of optimisation by a series of non-linear functions using a particle swarm methodology is is introduced in Kennedy and Eberhart (1995) and then built upon in Shi and Eberhart (1998) by adding cooperation and competition among the individuals through generations. Each particle adjusts its 'velocity' according to its own, and other particles history.

A general definition of PSO states two steps, initialisation and interation; Initialisation : a set of particles is initialised with random positions and velocities. This step is equivalent to a random search, Iteration : every particles position is updated based upon its velocity, the particles historically best position and the swarms historical best.

In more detail this becomes the basic outline:

1. Initialisation - randomly generate an initial population
2. Fitness - calculate fitness of each particle `evaluate()`

3. Update Velocity - recalculate velocities `update-velocity()`
4. Update Position - recalculate positions `update-positions()`
5. Repeat

Formally, consider a set p_i^t containing all particles i on iteration t

$$p_i^t = \{p_{(i,1)}^t, p_{(i,2)}^t, \ldots, p_{(i,D)}^t\},$$

which is then used to calculate the best solution of p_i^t, p_g^t

$$p_g^t = \{p_{(g,1)}^t, p_{(g,2)}^t, \ldots, p_{(g,D)}^t\},$$

and can then be used to update the velocities for $d = 1, 2, \ldots, D$ where c_1 is the cognative constant, c_2 is the social constant and r_1, r_2 are independent random numbers.

$$v_{(i,d)}^t = w \cdot v_{(i,d)}^{t-1} + c_1 r_1 \left(p_{(i,d)}^t - x_{(i,d)}^t \right) + c_2 r_2 \left(p_{(g,d)}^t - x_{(i,d)}^t \right).$$

$v_{(i,d)}^t$ are the new velocities for particle i at d, and so updating the positions for the next iteration x^t

$$x_{(i,d)}^{t+1} = x_{(i,d)}^t + v_{(i,d)}^t$$

4.1 Motivation

The overarching motivation for this work is the development of improved convergence and reliability of decision tree classifiers / regressors. Of particular interest are approaches that can be applied to general problems of moderate dimensions, where run-time issues limit the number of function evaluations or model runs that can be undertaken. Whilst there has been some research into this area has been undertaken, compared to research developing new stochastic based ensemble methods such as SGTB (which has been implemented in whitebeam), this field is less developed and has the potential to increase evaluation time by allowing a final model to evaluate a reduced dataset - potentially reducing a dataset by up to 3x without losing any accuracy but this is at the cost of far higher initial computation time.

4.2 Implementation

The implementation consists of two classes `PSO` and `Particle`. `PSO` is the main body, it controls the correct setup of the algorithm and initiates a series of `Particle` objects which contain the functions `evaluate()`, `update_velocity()` and `update_position()`. Additionally, it is shipped with a handling script which contains the function that will optimised and allows command line arguments such as filename, iterations and verbosity. It must be noted, the `PSO` object is created by passing a filename and a function, the filename is passed by commandline (the filename function is entirely custom to this pipeline, and not general beyond hdf5 keywords) and is used to setup particle sizes and the function must be defined within the script. This approach allows this optimiser to be entirely general to any decision tree, and requires a single line change to solve any function.[1]

They key distinction of this algorithm is that the number of dimensions is intrisically based on the shape of the data and each dimension of the particle represents a dimension of the data. Implied bounds of 0 and 1 are given which represent including the column in a model. This is performed by the optimisation function

```
# optimisation function for decision tree classification
def decisionTree(position, Xtrain, Xtest, ytrain, ytest):

    # Extract positions
```

[1]Initially a full rust implementation was the plan, with a c4.5 and PSO created entirely in rust. However a numpy equivalent for rust has not been created, and due to the nature of the project, Array operations must be used, and the time scope of the project does not allow for creating an Array library from scratch.

```
    pos = np.array(position)
    pos = pos.astype(bool)

    # Find and extract datasets with those positions
    X_train=dataFrame(pos,Xtrain)
    X_test=dataFrame(pos,Xtest)
    y_train=ytrain.values
    y_test=ytest.values

    # Create a model
    clf = DecisionTreeClassifier()
    # Fit the model
    clf.fit(X_train, y_train)

    # Return negative score as pso minimises
    return -clf.score(X_test, y_test)
```

The stopping conditions are based on error convergence or a maximum iterations met.

Without inundating this report with source code, further source can be found at https://github.com/K-Molloy/scc461-pso

5 Results

This section focuses on the results of whitebeam and the proposed PSO method. First the datasets were described and experimental setups, after which the results of the experiments are presented and compared with other well-known classification algorithms.

5.1 Dataset Selection and Processing

The datasets selected are presented in Table 4, they have been selected for their wide array of features:

- Attribute Type : Categorical / Real / Integer
- Size : Small / Large / Super-Large
- Preprocessing Required : None / Significant
- Problem Type : Binary / Multiclass

The hope is the massively varying requirements can test the proposed algorithms with their different features / attributes. All datasets are pre-processed using the same template, the pipeline is described in Figure 4, this outputs all sklearn methods and provides a dataset.h5 file for reproducability for other methods - this ensures the tests are performed on the exact same data.

5.2 Environment

The performance of the methods are evaluated by conducting numerical experiments using 10 datasets with diverse feature types, sizes and classes. The datasets are described in 4.
All tests are created in python 3.8.3 with sections created in cython and compiled to C at runtime, they are then run on WSL2 (Windows Subsystem for Linux 2) operating Ubuntu 20.02 paired with an Intel i7-7700K @ 3.2GHz x 4 cores (8 threads), with 16GB 3000MHz (roughly 13GB available) physical memory (RAM) and 53GB virtual memory available. Models are loaded from an M2 SSD with an average speed of 400 MB/s.
Models have been run concurrently to attempt and keep uncontrollable system variables similar, such as background activities, loaded packages, etc.

Table 4: Comparison Dataset Descriptions

	Type	Attribute Type	Task	Instances	Attibutes	Processing
breast-cancer	Multivariate	Categorical	Binary Classification	286	9	Heavy
breast-cancer-ws	Multivariate	Real	Binary Classification	569	32	Minimal
egypt-hcv	Multivariate	Integer, Real	Multilabel Classification	1385	29	Heavy
glass	Multivariate	Real	Multilabel Classification	214	10	Minimal
heart-disease	Multivariate	Categorical, Integer, Real	Multilabel Classification	303	75	Heavy
hepmass	Multivariate	Real	Binary Classification	1050000	28	Minimal
htru2	Multivariate	Real	Binary Classification	17898	9	Minimal
lsvt	Multivariate	Real	Binary Classification	126	309	Minimal
ionosphere	Multivariate	Integer, Real	Binary Classification	351	34	Minimal
dota2	Multivariate	Boolean	Binary Classification	102944	116	Minimal

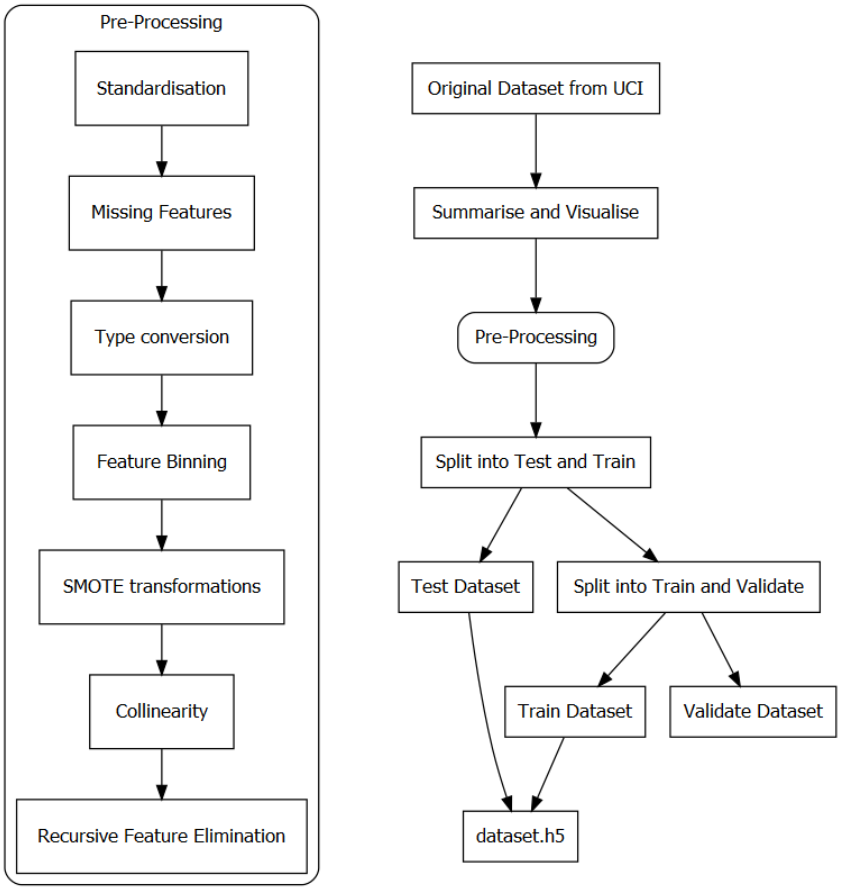

Figure 4: Dataset Processing Flow, from UCI Download to .h5 redistribution

Table 5: evaluation function model descriptions

ID	Model
sklearn	
LDA	Linear Discriminant Analysis
CRT1	CART Tree
KNN	K-Nearest Neighbours
BGT	Bagging Classifier
RNF1	Random Forest
EXT	Extra Trees
GBM	Gradient Boosting
whitebeam	
CRT2	CART Tree
C4.5	C4.5 Tree
ALPH	Cost-Complexity Tree
FRID	Friedman Tree
XGBT	XG Boosted Tree
SGTB	Stochastic Gradient Tree Boost
RNF2	Random Forest
pso	
PSSK	PSO + sklearn
PSWH	PSO + whitebeam

5.3 Numerical Experiments

The effectiveness of whitebeam and PSO are evaluated by comparing method accuracy with library tree functions and other popular classification methods, namely, KNN, LDA, BGT.
Based on the experiments, the performance of the algorithms are evaluated, in terms of accuracy rate calculated using the Jaccard Index,

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

and by design, $0 \leq J(A,B) \leq 1$.
Table 6 lists the accuracy of whitebeam and PSO methods in addition to the baseline library methods. It is split into 3 category-columns, sklearn, whitebeam and PSO. sklearn are the library functions for comparison, whitebeam are all methods from the whitebeam library and PSO are implementations with Particle Swarm Optimisation fitting. The acronyms in the table are given their respective modelling names in Table 5. Figure 5 shows the memory usage for the runtime of the python script for breast-cancer, htru and ionosphere for both sklearn library implementation and whitebeam implementation.[2]

[2]graphs for all datasets are available in the documentation for whitebeam

Table 6: library implementation dataset comparisons

	sklearn							whitebeam							pso	
model.descriptions	LDA	CRT1	KNN	BGT	RNF1	EXT	GBM	CRT2	C4.5	ALPH	FRID	XGBT	SGTB	RNF2	PSSK	PSWH
breast-cancer																
CV Score	0.816937	0.717346	0.679263	0.768141	0.813350	0.809697	0.815184	0.706896	0.706896	0.706896	0.706896	0.706896	0.706896	0.706896	0.793103	0.706897
CV std	0.031102	0.040219	0.019984	0.013111	0.027630	0.03034	0.022056	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	0.015433	0.012062	0.026631	0.090479	0.664595	0.522923	0.410072	0.075842	0.076907	0.074184	0.077594	0.071795	22.6465	6.584919	1.51602	8.69754
breast-cancer-ws																
CV Score	0.954325	0.967598	0.973367	0.958128	0.965747	0.975254	0.97529	0.931428	0.948571	0.948571	0.925714	0.925714	0.925714	0.931428	0.971429	0.954286
CV std	0.016428	0.014819	0.012385	0.025089	0.014005	0.012095	0.018909	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	0.012932	0.012062	0.033169	0.156517	1.056037	0.862741	0.581069	0.104807	0.104807	0.104233	0.103796	0.102898	20.13498	20.26456	0.73153	22.5839
dota2																
CV Score	0.981256	0.931301	0.961702	0.963806	0.963950	0.96395	0.963537	0.96395	0.96395	0.96395	0.96395	0.96395	0.963734	0.96395	0.936158	0.96395
CV std	0.001185	0.002473	0.000937	0.000875	0.000860	0.00086	0.00098	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	12.621177	5.288830	81.438744	32.603538	39.893312	59.553106	304.741255	5.4375321	3.5080125	3.2729027	2.8173942	3.3374693	537.469331	137.818933	472.83923	368.92424
egypt-hcv																
CV Score	0.282801	0.240632	0.247861	0.252666	0.275557	0.25268	0.269548								0.306859	
CV std	0.011622	0.027042	0.021589	0.028842	0.017376	0.026921	0.023824								1	
Training Time	0.662838	0.062971	0.121962	0.181725	0.769000	0.734849	2.237237								2.5268	
glass																
CV Score	0.655385	0.640615	0.719692	0.688308	0.797231	0.804615	0.719077								0.936751	0.706896
CV std	0.051349	0.014053	0.084933	0.05033	0.026512	0.049288	0.067001									
Training Time	0.016308	0.009770	0.014242	0.077978	0.586757	0.437822	1.75542								480.05181	24.57183
heart-disease																
CV Score	0.816937	0.717346	0.679263	0.768141	0.813350	0.809697	0.815184								0.766304	0.706896
CV std	0.031102	0.040219	0.019984	0.013111	0.027630	0.03034	0.022056									
Training Time	0.015433	0.012062	0.026631	0.090479	0.664595	0.522923	0.410072								2.42979	24.57183
hepmass																
CV Score	0.835401	0.816164			0.813350			0.806909	0.79875	0.79875	0.806909	0.806909	0.806913	0.806909	0.769116	0.813868
CV std	0.000375	0.000425			0.027630			1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	118.350246	2056.796522			0.664595			89.425366	90.690338	92.529593	85.367174	84.701086	611.3449	2266.810721		7853.94292
htru																
CV Score	0.975321	0.967498	0.973086	0.979232	0.980257	0.97886	0.978301	0.978491	0.97877	0.97877	0.97877	0.97877	0.979888	0.978492	0.982123	0.971831
CV std	0.002248	0.003275	0.002662	0.002077	0.001736	0.001606	0.003238	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	0.199972	0.342312	0.351883	1.797237	5.138928	1.515634	8.016993	0.1934251	0.2585043	0.2112766	0.2080175	0.1942343	4.896769	30.80292296	36.38298	29.3213
ionosphere																
CV Score	0.857143	0.866667	0.9	0.890476	0.980257	0.904762	0.904762	0.957746	0.943662	0.943662	0.943662	0.943662	0.929577	0.957746	0.971831	0.971831
CV std	0.052164	0.041513	0.040963	0.035635	0.001736	0.054294	0.026082	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	0.165357	0.102315	0.062733	0.401235	5.138928	0.651162	0.793946	0.180321455	0.17241549	0.1698842	0.16264414	0.16504883	19.57314	26.94411	1.81634	29.3213
lsvt																
CV Score	0.773333	0.786667	0.6	0.813333	0.840000	0.786667	0.8	0.692307	0.846153	0.846153	0.730769	0.730769	0.846153	0.692307	0.923077	0.923077
CV std	0.067987	0.077746	0.10328	0.114698	0.099778	0.077746	0.094281	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06	1e-06		
Training Time	0.118908	0.064658	0.018642	0.18496	0.537266	0.38369	1.232596	0.5536897	0.4789657	0.5096256	0.4903507	0.5120443	21.983395	43.43459272	38.93655	98.40017

* whitebeam currently unable to process multilabel

† whitebeam currently unable to process kfold cross_val_score(), so instead a brute force approach is taken

‡ hepmass causes python memory failure / timeout beyond 30 minutes for some models

5.4 Performance Comparison - Computation

Comparing the performance computationally considers average memory usage, peak memory usage and CPU processing time, which considers multi-threaded applications - which some of the implementations are.
Figure 5 demonstrates the current memory usage for 3 selected datasets, breast-cancer, htru and ionosphere (chosen for varying accuracy and fast computation), for both sklearn and whitebeam.[3] The sklearn implementations appear to be more memory-stable throughout the script, making somewhat discrete jumps where as whitebeam is always at a positive gradient. The regression lines, shown in blue, would suggest all solutions are not stable, however the scripts with a longer runtime generally have lower gradients, where as those with short runtimes have higher gradients. If the script was artificially slowed, it would be expected the gradient would eventually decrease to near 0. The computation time are included in the main results table, as well as all auxillary tables.

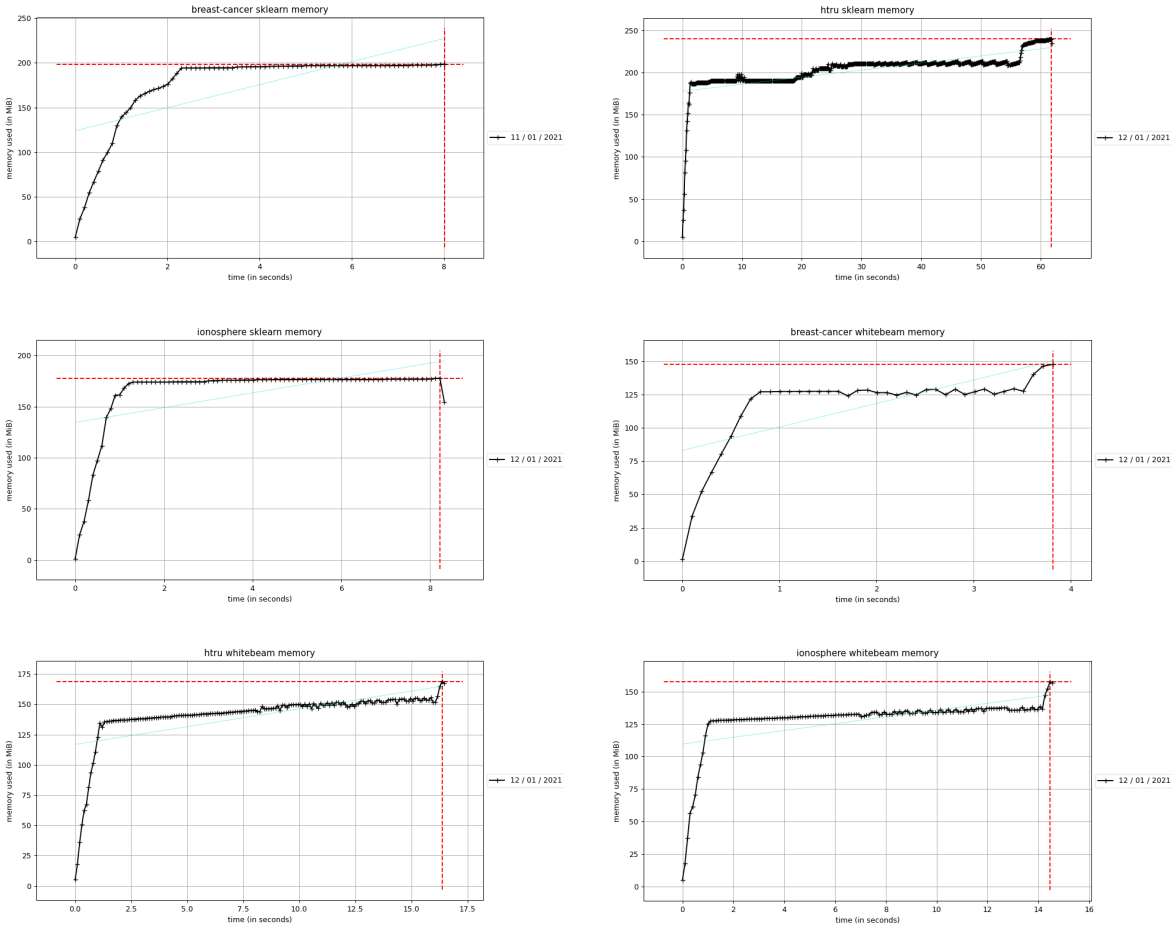

Figure 5: Memory Plots for Sklearn Library Implementations

5.5 Performance Comparison - Accuracy

Comparing accuracy considers average score and score standard deviation, Figure 6 demonstrates key characteristics for the breast-cancer dataset, notably sklearn's CART implementation has the highest std, and second lowest library accuracy however it is the fastest calculation of all models.[4] whitebeam has no deviance for all models, this could be due to a blunt implementation of kfold-cross-validation. The

[3] the other plots can be found in the appendices
[4] Accuracy Plots use discrete y axis to show discrete accuracy differences, where as Time plots use continuous y axis to clearly show execution time comparisons

highest accuracy for breast-cancer is LDA, which is likely due to pre-processing methods transforming the dataset into a format which is strong for LDA.

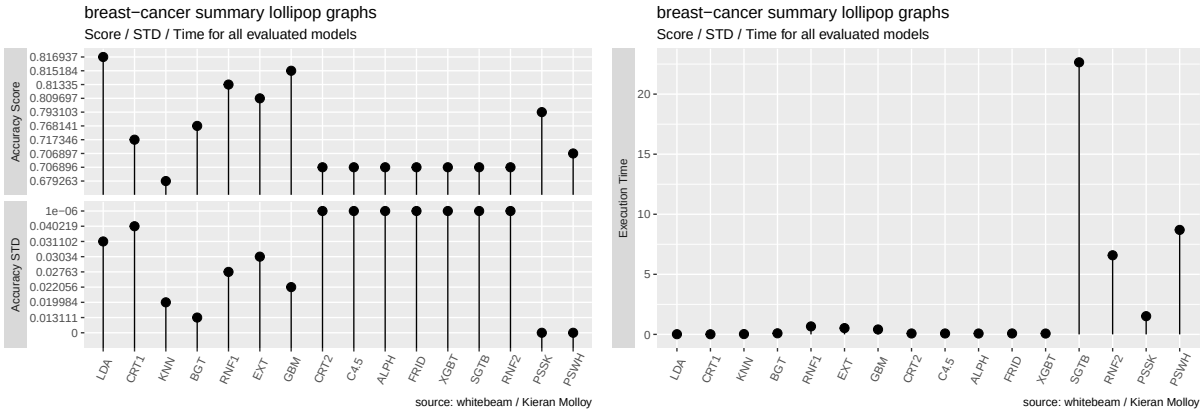

Figure 6: breast-cancer summary plots

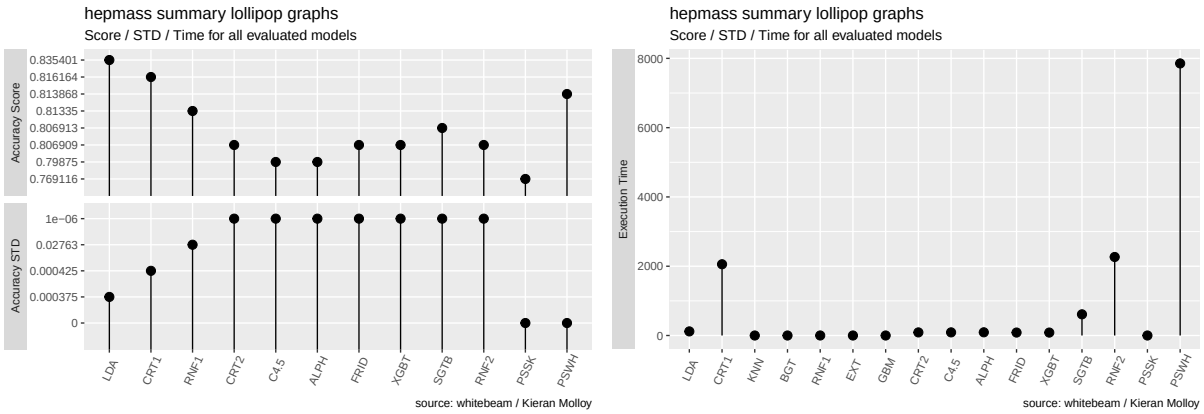

Figure 7: hepmass summary plots

Figure 7 demonstrates key plots for hepmass, the hyper-large dataset, the methods which did not complete are removed, those are KNN, BGT, RNF1, GBM and PSSK. Also removed from the time plot is the PSWH model, as it skewed the graph to not be useful. All whitebeam methods have accuracy scores lower than sklearn's LDA and CART, however the remaining library functions were not able to run, and whitebeam methods required far less computation time and memory across the board. This could be due to more complicated method checking in sklearn, where excess copies of data are created instead of implicit referencing. It is clear both methods benefit from core implementation in cython, with function calls into the 1000's for the core methods.

Figure 8 shows the plots for htru, all whitebeam methods attain higher accuracy and lower execution time whilst using less memory than sklearn's CART implementation. sklearn's Random Trees outperforms whitebeam though, whilst also attaining the 2nd highest observed accuracy - with PSO + Sklearn being the highest but also coming with a runtime of nearly 45s, which is over 45x longer. It must be noted this dataset sees the smallest deviation of all scores - with all accuracy scores between $[0.967, 0.982]$.

Figure 9 shows the summary plots for ionosphere, as with htru, ionosphere sees whitebeam methods having higher accuracy scores than sklearn CART, however whitebeam CART gets the 3rd highest score - only topped by both PSO methods, and sklearn Random Forest, all taking multiple factors higher computation time. Sklearn Random Forest does see the lowest standard deviation of all models, suggesting if robustness is required then Random Forest would be the best model to use. sklearn CART has a lower computation time than whitebeam CART - which is exacerbated in the PSO implementations

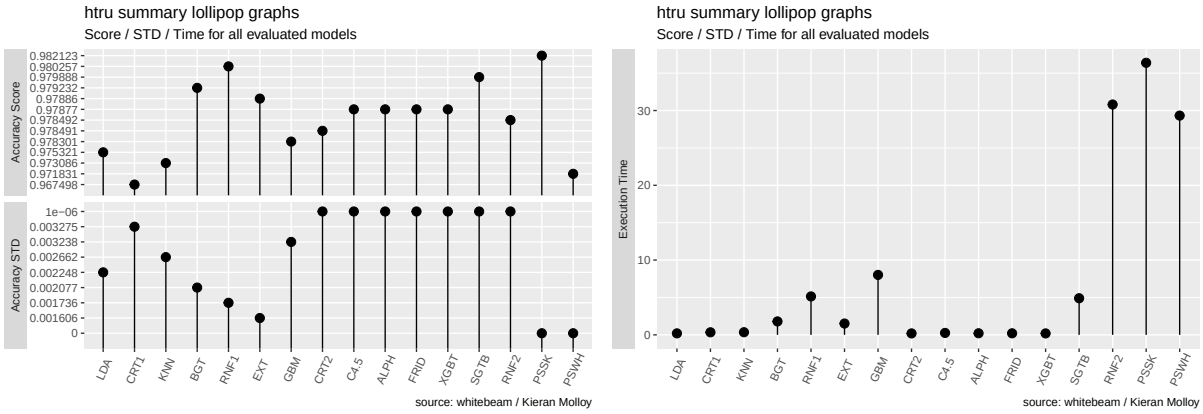

Figure 8: htru summary plots

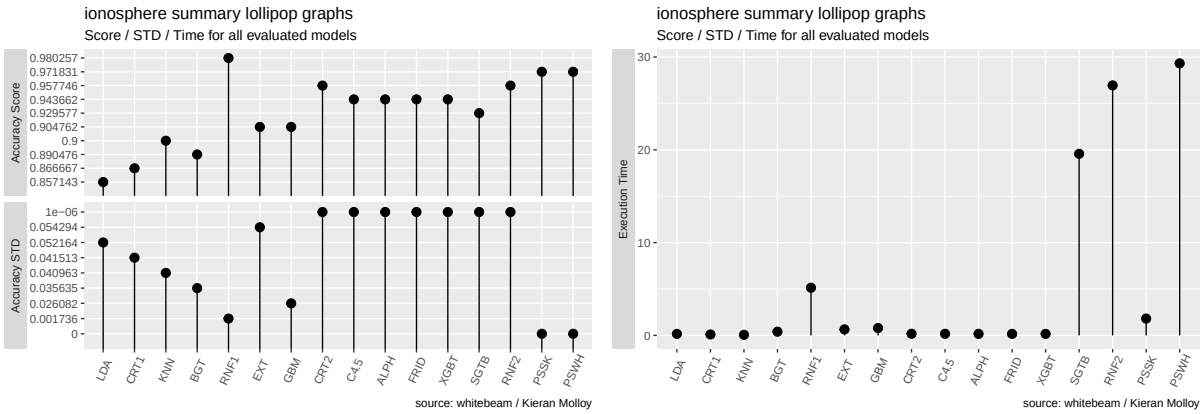

Figure 9: ionosphere summary plots

where sklearn takes 1.81634 CPU/seconds and whitebeam takes 29.3213 CPU/seconds to achieve the same accuracy.

5.6 Depth Comparisons

When evaluating the best `max_depth` parameter for whitebeam - sklearn has an automatic method of allocating `max_depth` when it is not defined explicity, so is not compared - using two plots for depth comparisons, `max_depth` vs `computation time` and `max_depth` vs `score`. As previous sections, the depth comparisons are shown for the breast-cancer, htru and ionosphere. The right-hand plot in Figure 10, representing maximum depth vs computation time shows that the maximum depth changing makes no difference to the accuracy score attained beyond a maximum depth of 1, the minimum split depth, this suggests the split found is highly informative and the data has very clear binary classes based on one or multiple features. Observing the F-Score, Precision, Recall and Support in left-hand plot in Figure 13 show no deviation reinforcing the previouus statement, therefore the deciding factor is the computation time, which is seemingly random for any method and maximum depth. Hence, for this dataset, any method with any depth would be optimal - perhaps with a different preprocesssing process this would not be the case.

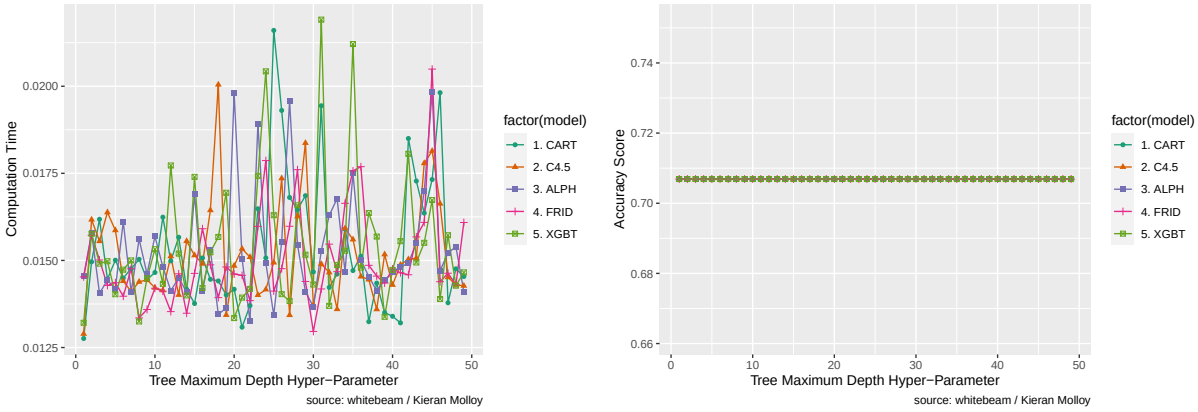

Figure 10: ionosphere summary plots

The left hand plot in Figure 11 follows the trend which is expected from increasing maximum depth - in that the computation time increases following a sigmoid function - it increases slowly until depth=10, then faster until depth=20, then slowly again. All methods have the same computation time from depth=1 to 10 then high deviance can be observed, those lines closer the the x-axis are more desirable functions due to their lower computation times. That is relying on its accuracy being acceptable, as in the right hand plot, which shows a peak around depth=7 is seen, with accuracy dropping as depth approaches 50 - this is due to model overfitting, this is reflected in the classification metrics shown in the middle plot in Figure 13, where all values see a clear drop after depth=10 - again a sign of overfitting. CART has the highest average accuracy, but not the peak, Friedman tree and XGB Tree attain the peak accuracy at depth=6 and as shown in the left-hand plot, there is no deviance in computation time at depth=6 - for htru Friedman Tree or XGB Tree should be used with depth=6 for maximum accuracy and minimum computation time.

The computation time observed for ionosphere, Figure 12, demonstrates a more pronounced sigmoid curve than htru and all methods do not deviate beyond 0.05 at any maximum depth, the right plot shows clear maximums for CART and other models, with CART scoring 0.957746 for all depths beyond depth=4 and the other models all scoring 0.943662 beyond depth=3. Comparing this to the computation curve, it is clear that the optimal results are obtained from CART at depth=4. The classification metrics, in the right plot of Figure 13, reinforce this sentiment approaching maxima after depth=3 and achieving maxima of all metrics at depth=4.

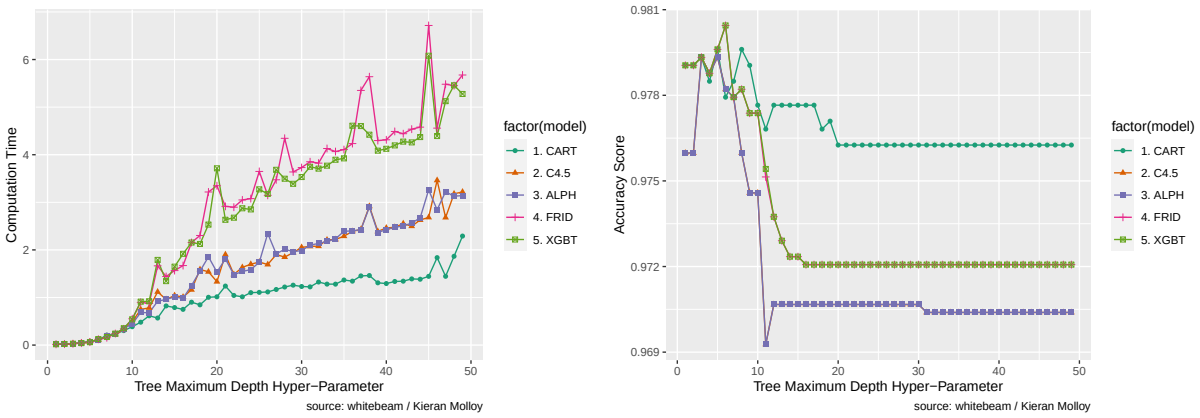

Figure 11: htru summary plots

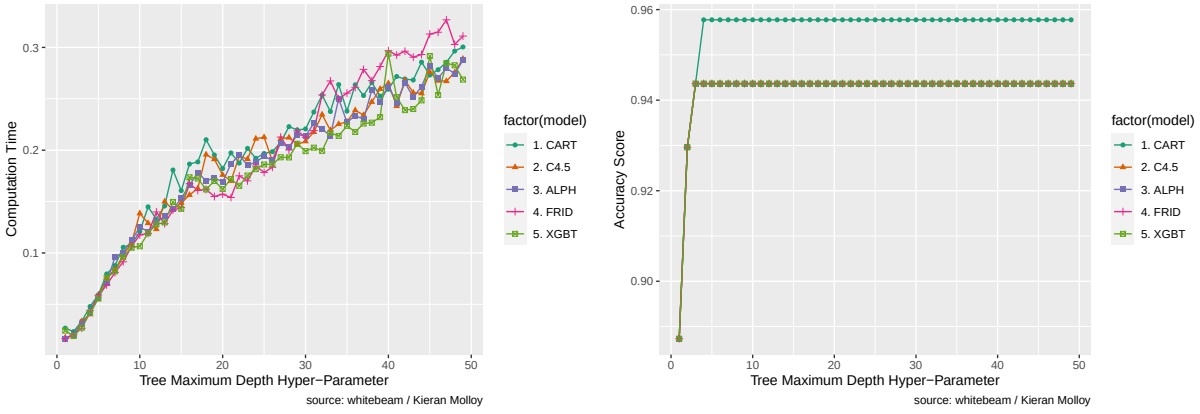

Figure 12: ionosphere summary plots

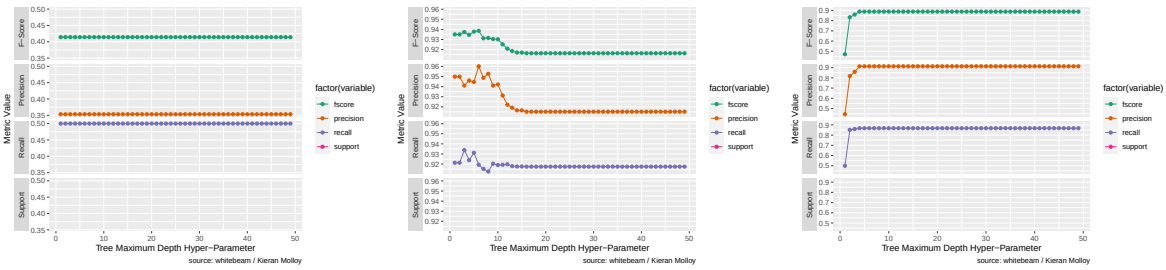

Figure 13: htru fscore, precision, recall and support plots

6 Conclusions

In this report, a novel method in feature selection was proposed using a PSO algorithm combined with a CART decision tree classifier, as well as a library for creating new decision tree-based classifiers. Ten datasets were used to test the performance of the proposed methods, sklearn library functions uses k-fold cross-validation methods to justify the performance of the models with whitebeam and PSO lacking this feature. whitebeam performed strongly, especially for htru, ionosphere and lsvt datasets where it outperformed sklearn both in computation time and accuracy however for most other datasets sklearn is more accurate and faster. The exception to this is hepmass, the huge dataset, sklearn's decision tree alternatives were unable to cope with such a large dataset with LDA taking 4x longer than CART and other methods running out of time or memory. whitebeam's ensemble methods were generally unable to improve models by more than +0.1 score, at a cost of more than 2000x computation time however this may be due to dataset selection and processing methods. The PSO method was able to achieve high accuracy scores for all datasets, but as expected, came at the cost of computation time This research has two key contributions. Firstly, this study has created a decision tree library with comparable performance with sklearn providing popular classification methods such as CART, C4.5 and Friedman tree and standard ensemble methods SGTB and Random Forests. Second, the PSO algorithm was successfully applied to the binary applications and combined with a classifier, providing a solution for feature selectio. Third, in this report, the proposed PSO methods and whitebeam are compared against other well-known library classifiers using a variety of datasets with diverse sizes and feature types.
Further development for whitebeam is required, including support for cross-validation, in-built metrics and more ensemble methods, the PSO requires further investigation into hyper-parameters and ensuring it is distributionally robust. Furthermore, a hybrid method such as Black Widow Optimisation, where a new stage of cannibalism is added, or standard genetic algorithm, where a mutation operator is added, could help diversity and randomness in the solution set.

References

Breiman, Leo. 1996. "Bagging Predictors." *Machine Learning* 24 (2): 123–40. https://doi.org/10.1023/A:1018054314350.

Breiman, L., J. Friedman, R. Olshen, and C. J. Stone. 1983. "Classification and Regression Trees." In.

Buitinck, Lars, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, et al. 2013. "API Design for Machine Learning Software: Experiences from the Scikit-Learn Project." In *ECML Pkdd Workshop: Languages for Data Mining and Machine Learning*, 108–22.

Falco, I. De, A. Della Cioppa, and E. Tarantino. 2006. "Evaluation of Particle Swarm Optimization Effectiveness in Classification." *Fuzzy Logic and Applications Lecture Notes in Computer Science*, 164–71. https://doi.org/10.1007/11676935_20.

Friedman, Jerome. 2002. "Stochastic Gradient Boosting." *Computational Statistics & Data Analysis* 38 (February): 367–78. https://doi.org/10.1016/S0167-9473(01)00065-2.

Gehrke, Johannes, Raghu Ramakrishnan, and Venkatesh Ganti. 2000. "RainForest—a Framework for Fast Decision Tree Construction of Large Datasets." *Data Mining and Knowledge Discovery* 4 (2): 127–62. https://doi.org/10.1023/A:1009839829793.

Kennedy, J., and R. Eberhart. 1995. "Particle Swarm Optimization." In *Proceedings of Icnn'95 - International Conference on Neural Networks*, 4:1942–8 vol.4. https://doi.org/10.1109/ICNN.1995.488968.

Park, Y., and J. Ghosh. 2014. "Ensembles of (α)-Trees for Imbalanced Classification Problems." *IEEE Transactions on Knowledge and Data Engineering* 26 (1): 131–43. https://doi.org/10.1109/TKDE.2012.255.

Park, Yubin, Joyce Ho, and Joydeep Ghosh. 2016. "ACDC: α-Carving Decision Chain for Risk Stratification." http://arxiv.org/abs/1606.05325.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12 (85): 2825–30. http://jmlr.org/papers/v12/pedregosa11a.html.

Shi, Y., and R. Eberhart. 1998. "A Modified Particle Swarm Optimizer." In *1998 Ieee International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, 69–73. https://doi.org/10.1109/ICEC.1998.699146.

Sousa, Tiago, Arlindo Silva, and Ana Neves. 2004. "Particle Swarm Based Data Mining Algorithms for Classification Tasks." *Parallel Computing* 30 (5): 767–83. https://doi.org/https://doi.org/10.1016/j.parco.2003.12.015.