Name: Oliver Goldstein
Student ID: og14775

Table Of Contents

The Haskell file containing code for the model is located in model.hs and contains ~400 lines of code.

Summary

I will investigate how models in context of functional programming can lead to an improvement in programming accessibility, correctness and potentially efficiency. The specific model which I will formalise is the notion of how decision trees learn models.

Aims and objectives:

1. **(Week 1)** Perform literary review of previous work on DSLs and decision trees. Simultaneously research advanced methods in Haskell.
2. **(Week 2)** Produce a model which describes decision trees in functional notation. Given time, I will implement and execute the model in Haskell.
3. **(Week 3)** Discuss ways in which composition could be used to create a meaningful notion of a language.
4. **(Week 4)** I will conclude by discussing potential improvements with respect to compile time analysis, efficiency, usability, security and extensibility, if DSLs can be adopted.
5. **(Week 5)** Modifications and Improvements.

Deliverables:

1. Construction of a model that describes how decision trees classify.
2. A representation of the model in functional notation and a possible implementation of this model in a functional programming language, specifically Haskell.

Introduction

Individuals without direct knowledge of the specifics of software engineering face challenges when verifying the correctness of code with respect to a given model. This is especially challenging in languages such as C. I contend that languages which are “close to the hardware” force the implementor to think as a machine, rather than a human, restricting the ability of people outside of software engineering to program computers. I advocate that the description of algorithms that exist within our heads should match the implementation of the algorithms as closely as possible. I feel that the functional programming paradigm allows one, versed in small amounts of mathematical literature, to do precisely this.

A concise model for data structures allows one to quickly verify and understand it with respect to a specification, without dealing with hardware related bureaucracies. In order to come up with models in languages such as C can be incredibly time consuming. DSL’s can raise the abstraction level and reduce development time. Models can be further analysed and decomposed into primitive, reusable operations and a DSL can be constructed. The difference between an Application Programming Interface (API) and a DSL is that an API operates on a known data type, whereas a DSL operates on an abstract data type.

The first 10 pages constitutes a literature review, following this is the implementation of a Haskell model.

Analysis of previous work.

When programming languages are discussed, they are often most thought of as general purpose languages, i.e. one capable of programming any application with a similar degree of efficiency. However, for many applications, there are more natural ways to express problem solutions than by using a GPL. A DSL can do so and should encapsulate the semantics of the application domain, in my case decision trees, no more and no less, providing the highest level of abstraction as possible.

Well built DSLs are more concise, quicker to develop, facilitate easier maintenance and they can be reasoned with more effectively. Examples of popular DSL's are SQL for databases, HTML for document markup, Prolog for logic and Open GL for 3D graphics.

Most importantly DSL's can be easily understood and written by non programmers, for instance, people well versed in the domain semantics but not in the specifics of software engineering. An important characteristic of a DSL, therefore, is an effective notation to express that semantics. Although DSL's may have a greater start up cost, ultimately in the long run their initial cost pays off, from the point of view of the software development life cycle.

Moving from DSL's to EDSL's

Conventional DSL's incur large startup costs for a number of reasons, from performance to difficulty in developing the language itself. Often problems with high level DSL's can have far worse performance than low level languages such as C. However optimisations are possible on these high level constructs, such that programming these by hand would be tedious in a low level language. Therefore by hijacking a language such as Haskell and performing a language through embedding often performance criteria can be met. This results in the construction of an *EDSL*. One can distinguish between deep embeddings and shallow embeddings of languages. A deep embedding uses an algebraic data type to represent the abstract syntax tree of the EDSL and is often used to generate code. Shallow embeddings map language constructs directly to semantics. In the model I create, I will generate a shallow embedding.

Programming Languages in Artificial Intelligence: Past to the future

Lisp and its relation to Artificial Intelligence : Functional Programming before Haskell

Before constructing a DSL for a particular language, I am interested in briefly touching upon the past of programming languages in AI and machine learning, perhaps to source inspiration from their analysis. The language I will touch upon is Lisp as it is semantically similar to Haskell, although there are many others such as ProLog etc.

In functional programming, major flow control methods are recursion and conditionals, different from imperative programming which uses sequencing and iteration to dictate control flow. FP also uses higher order functions. Functional programming is highly modular, especially due to the lack of side effects.

Lisp was based on the idea that recursion and conditionals are useful for performing symbolic computation. In addition, notions of abstract mathematical functions defined in lambda calculus are useful for specifying problems in AI. In Lisp, data and functions are represented as s-expressions which are either atoms or lists. With lists, very complex representations of objects can be constructed easily. The semantics of lisp occurs via the evaluation of s-expressions, which when evaluated can result in s-expressions.

In Lisp, the interpreter provides semantics, and it operates according to three rules. The interpreter is a unary function EVAL:

1. Identity, i.e. intrinsic data types with literal meaning that map to themselves.
2. Symbols, which when evaluated returns the s-expression associated to that symbol, i.e. symbols are variables bound to values.

3. Lists, which are interpreted as function calls with prefix notation, where each the first element of a list is a function with arguments.

In order that the interpreter can distinguish between symbols, lists and data the QUOTE function is used which is the identity s-expression. This allows programs to be treated as data. Lisp allows state to be handled via the use of a function called SETQ, which has two arguments; the symbol to which a value is bound and an s-expression which provides that value.

List operations are central to Lisp, as well as support for lambda functions, higher order functions and recursive definitions. Ultimately, Lisp is useful as it is quick to prototype ideas and it supports symbolic AI well. In addition the weak distinction between data and programs allowed for quick hacking However, the confusing syntax and other factors contributed to its decline.

The main point that I wish to make is to highlight the remarkable similarity between Lisp and Haskell, which is the general purpose language I chose to implement the EDSL.

Fig 1.0

Decision Trees

$$(\mathbf{x}, Y) = (x_1, x_2, x_3, \ldots, x_k, Y)$$

Fundamentally learning a decision tree classifier is analogous to understanding or generalising the dependent variable Y (Fig 1.0). Y is understood using the input vector x. A relation is constructed between members of the cartesian product in Fig 1.0 such that the relation matches as closely as possible the underlying distribution of the correspondence between x and Y in reality. In the top down induction of decision trees the order in which data is presented is not important, which is in contrast to more incremental methods such as MARVIN (1985).

Decision Trees have a variety of uses from categorical classification to regression, clustering and probability estimation with respect to real valued features. Although classification appears to be a small part of procedural knowledge, there exist diverse activities such as robotic planning that can be recast as classification problems (Dechter and Michie, 1985). The construction of decision trees is also NP complete (Rivest Hyafil 1976). Decision trees have evolved from the CLS algorithm in the 1960's through the various algorithms proposed by Quinlan e.g. ID3 and many others most notably the CART and C4.5/C5.0 algorithms. They are interpretable and often generalise well assuming the features are correlated with the target concept. There are many different learning algorithms here, one can also use meta data about the data sets to choose the optimal learning algorithm in a field called meta learning. Decision trees cannot compactly represent simple functions such as linear functions. In addition, for discrete models, for regression trees, a continuous variable is predicted.

Different forms of decision trees

Before constructing a model, I wish to explore other decision tree models, splitting criteria and new forms of the concept. If the model constructed is meant to capture the essence of the domain of decision trees then in some sense, it should be able to generalise the following concepts. Although the no free lunch theorem states that if one model outperforms a model on one data set, then it must underperform that same model on another data set. I will try and abstract away from this principle, in the sense that the model I eventually construct must be invariant to different inductive biases.

Regression Trees

This concept can be simply generalised by replacing the standard grow tree algorithms impurity measure by the variance of the set. A concept similar to regression trees are model trees which were pioneered (yet again) by Karalic in 1991 with RETIS and Quinlan (1992) with the M5 algorithm. The difference with M5 is that when the tree is pruned, interior nodes are replaced by a regression plane rather than a constant value. Regression trees are also very similar

to clustering trees. One can adjust the CART algorithm to change the impurity measure to be cluster dissimilarity, using any abstract measure such as euclidean distance etc.

Tree based regression models are known for their simplicity and efficiency when dealing with domains with large numbers of variables and cases. However they can be unstable and non smooth. Regression trees can be of the form of piecewise constant regression models, as they partition the instance space into a set of regions and fit a constant value in each region.

Often regression trees are built by trying to minimise the least squares error. At each leaf, the constant that minimises the expected value of the squared error is the mean of the target variable. Some models use linear polynomials instead of averages at each leaf (M5 mentioned above).

Instead of using the least squared error as the splitting criteria, some have raised the possibility that least absolute deviation is a more robust error criterion as it weights differences linearly rather than with a squared relationship. LAD trees have medians at their leaves rather than means, which is a better measure of centrality than mean for skewed distributions. LAD and LSE trees are similar conceptually, but LAD trees have a very high computational cost due to the necessity to do a lot of sorting for each split.

Regression by rule induction (Weiss and Indurkhya 1995)

Tree models find solutions which are conjunctive expressions disjunctive normal form. The conjunctive expressions may represent non disjoint regions. Several rules may be satisfied by a single sample in higher dimensional spaces. Some mechanism is needed to resolve conflicts. One method orders rules (Weiss & Indurkhya 1993a). These are often called decision lists and the rule that is chosen is the first one that is chosen. An extension in this paper is proposed by turning the regression covering problem into a classification problem.

In order to classify, there is an expectation that there is some pattern within the data. Is there therefore an expectation that classes formed by an ordering of the input feature to be a reasonable classification problem. The authors argue that there is.

Local regression trees

The problem with models mentioned above is that they are very sharp at boundaries between nodes and this is often highly unlikely to reflect the underlying distribution of data. One possibility to solve this is by using local regression at the leaves. Local regression is non parametric and does not assume anything about the underlying global distribution of data. However, local regression techniques have a high computational cost and low comprehensibility. Combining them with trees may have advantages.

Non parametric regression consists of obtaining the prediction of a data point x by fitting a parametric function in the neighbourhood of x. Work of this kind starts in the 1950s with kernel methods, which is effectively fitting a polynomial of degree 0 (constant) within a neighbourhood. Often these kind of regression methods are classed as lazy learning methods rather than eager methods. In addition, for regression in higher dimensions the models are not comprehensible. In addition, with many attributes, training cases can be so sparse that one could hardly describe the regression techniques as being local! In response additive models were created, using the idea that complicated regression functions can be decomposed by simpler functions, which are hopefully more comprehensible and have fewer dimensions. These simple additive functions can be created with the iterative backfitting algorithm.

Kernel regression methods can also be improved by using a weighting function with respect to similar instances contributing to more of the decision of the classification of the test data. These can be gaussian kernels or distance measures or even k nearest neighbour models (Cleveland and Loader 1995).

The problems with crisp regression trees is that they are highly discontinuous and often non smooth (Friedman 1991). Local models in the leaves can be beneficial as they provide smoother approximations to the data. However, there are disadvantages to these methods with respect to the fact that they store far more of the training data and the model they do create is far less comprehensible. In addition, when they underlying distribution in discontinuous there will clearly be problems with assuming a smooth underlying distribution. Finally Deng & Moore, 1995 show that there can be a great computational cost associated with lots of data with local modelling.

Random Forests These are trees where the training data, on which the feature vectors, on which the tree is grown is randomly selected (with replacement) either through bagging (Breiman et al. 1996) or random split selection (Dietterich 1998). At each node, the split is selected randomly from the k best splits. Random forests are seen as highly accurate on large data sets as they are generally observed to not overfit data. Random forests are a response to the high variance and high bias problems of single decision trees.

I have added random forests to the model produced below in less than 15 or so lines of simple code.

Oblique decision trees

Often, decision trees, at each node, only test the value of one attribute relative to a constant. This results in axis parallel splits. One can also test a linear combination of attributes at each internal node, resulting in a polygonal partitioning of the attribute space. In some domains, a set of axis parallel splits approximates the training data using a stair case like split set, but in an oblique model, only one or two splits may be required, and the model is both smaller and more accurate. By using a randomised hill climbing algorithm such as OC1, the runtime can also be guaranteed to be only O(log n) times greater than the worst case for inducing axis parallel trees. The two important computations here are the method for finding coefficients of a hyperplane at each node and methods for computing the impurity of a hyperplane. The essence of the function here is it takes a data set and returns a set of attributes, a threshold and a linear combination of weights.

Multivariate Decision Trees (Brodley & Utgoff 1992)

Closely related to oblique decision trees are multivariate decision trees. Multivariate trees overcome some of the representational limits of univariate decision trees. Decision trees that test multiple attributes at a node can be smaller than those that only test one node at a time.

To construct a multivariate decision tree, the following problems must be overcome:
1. Including symbolic features in multivariate tests.
2. Handling missing values
3. Representing a linear combination test
4. Learning the coefficients of a linear combination test
5. Selecting features to use in a test
6. Avoiding Overfitting

There are aspects of a decision tree that are advantageous. Firstly, only the features that are required to reach a leaf are required to be tested, if there is a cost of obtaining a feature then this is advantageous. There may be a tradeoff; simple tests may result in large trees that are difficult to understand, yet multivariate tests may obtain complex tests in small trees that are equally un interpretable. Univariate trees map ordered features to unordered boolean test features. In a linear combination test, one can form linear combinations using only the ordered features. Alternatively you could map unordered features to ordered features.

A note with noisy data; one can either remove incomplete data elements or one can estimate it using the sample mean which is an unbiased estimator of the expected value.

There are different ways in which a multivariate numeric test can be represented. For two classes, a multivariate test can be represented by a linear threshold. Increasing the number of classes allows for the use of a linear threshold unit or a linear machine (Nilsson 1965).

Given n features, one wants to find the set of coefficients which result in the best partition of the tree. In CART, it uses a maximisation of an impurity measure.

With respect to choosing the features to select, one wants to minimise the number of features in the test to increase understandability and decrease the number of features in the data. The number of tests will be exponential in the number of features to get the best set. Sequential backward elimination and sequential forward selection are two approaches to choosing features. SBE begins with all features and removes those that are not helpful in splitting. SBE assumes it is better to start from a informed location and work downwards, in contrast to SFS which starts from zero features which is an uninformed state and works upwards, gradually adding features. The criterion function can be a combination of both quality and cost of a feature.

In addition, one must be careful not to underfit the training data. Where there are many hyperplanes to choose from but very few data points, then there may be no basis for choosing one over another. Brodley and Utgoff argue that pruning a multivariate tree results in a greater number of classification errors and believe that instead, removing features from the test adds generality.

Finally, there needs to be a method to choose the coefficients for a linear combination test. The method I will analyse, is the recursive least squares algorithm (Young 1984). This seeks to minimise mean squared error over the training data. For discrete classification problems, the true value of the feature is the class which can be 1 for our purposes. To find the coefficients of the linear function which minimise the mean squared error. The RLS algorithm iteratively coefficients using error between the estimated and true value. One must initialise the weights and the covariance of this matrix, such that the covariance matrix reflects the uncertainty in your initial a priori choice of weight values.

There are three other methods for choosing coefficients, that I will briefly talk about. The pocket algorithm minimises the number of errors rather than the mean squared error. The thermal training procedure, identifies a problem with linear machines, such that they may never end if instances are not linearly separable. To solve this, decreasing attention is applied to large weights. CART does the same explicitly searching for weights that minimise impurity.

Oblivious Decision Trees (Langley & Sage 1994)

Case based reasoning relies on identifying a subset of features that are relevant to the learning task at hand. It is normally assumed that these are labelled, but when little domain knowledge is available, this becomes difficult. Cardie (1993) used a decision tree to identify those attributes relevant to the task at hand, passing on the features selected to a k nearest neighbour algorithm. The basis for oblivious decision trees is that when attributes interact, the greedy approach of the C4.5 algorithm suffers. In this case, a relevant feature seems to be no more discriminating than an irrelevant feature. In addition, irrelevant features degrade the performance of concept learners with respect to speed, as high dimensionality exists, and predictive accuracy (due to irrelevant information). Concept learning decision trees such as ID3 try to attempt to solve this by looking at information gain with respect to features, but feature interaction may result in little information gain. For example, if the concept is f1 XOR f2 and the feature values are uniform over {0,1} the probability of an instance being positive when f1 = 1 is 50% (Kira and Rendell 1993)

The oblivion algorithm is built on oblivious trees where nodes on the same level of the tree always test the same attribute.

Oblivion works as follows in polynomial time O(n^2) where n is the number of features:

1. Start with a full oblivious tree which includes all relevant attributes and estimate accuracy using n way cross validation.
2. Remove each attribute in turn, estimates accuracy of resulting tree and chooses most accurate tree.

3. If the new tree makes no more errors than the initial one then the old tree is replaced.
4. This continues until the accuracy of the best pruned tree is less than the current one.

Exhaustive search solves the problem giving an optimal solution, but as usual is exponential in time.

Information Fuzzy Networks

An addition to decision trees comes in the form of Last et al. (2002) who propose a new algorithm for creating decision trees called an information fuzzy network, based on information theory. Often, as stated many times already, decision trees can be very unstable which presents itself in cross validation results which have correspondingly high variance.

Stability is formally defined by Turney (1995) as the degree to which an algorithm generates repeatable results, given different batches of data from the same process. If it is phrased mathematically, it is the expected agreement of two models on a random sample of the original data, i.e. they both assign examples the same class.

There are some ways to deal with instability: Breitman et al (1984) recommend that after the CART algorithm completes, one should get a group of experts to review the trees and produce the best one on the efficiency frontier. The C4.5 algorithm built by Quinlan (1993) uses a windowing approach, which generates a single classifier from several samples. Meta learning approaches such as bagging (Breitman) and boosting (Freund) which generates multiple classifiers and weights them make the model more stable but lose comprehensibility. Domingos (1997a) attempts to use a method called combined multiple models to recover the lost comprehensibility. Here, a classifier is trained on a set of random examples, whose classes are predicted by the combined models. The resulting model is then used.

The IFN method uses statistical significance testing with dimensionality reduction. Ultimately the IFN method aims to solve the problem of tree instability. Whilst it is a network not a tree, it is structured in a similar way. Unfortunately due to time constraints and the complexity of the method I was not able to write further material on this subject.

Fast decision & Hoeffding trees

I will discuss VDFT, a system that can build decision trees using constant memory and constant time per example. If new examples of data arrive faster than they can be mined, then the quantity of unused data grows without bounds. The VDFT uses Hoeffding trees, which can learn examples in constant time. The probability that Hoeffding and conventional learners will choose different tests at any given node decreases exponentially with the number of nodes.

The VDFT algorithm proposed by Domingos and Hulten in 2000 is an I/O bound computation process. Classic decision tree algorithms such as ID3, C4.5 and CART assume that all training examples can be kept in main memory. SLIQ and SPRINT assume the examples can be stored on disk, and can be computationally expensive. VDFT does not store any examples, only requiring storage proportional to the size of the tree.

The authors note that in order to find the best attribute to test at a given node, one can consider a small subset of the training examples. Once the root test is chosen, examples are streamed to corresponding leaves and this procedure is applied recursively. The number of examples necessary at each node is calculated using a statistical result called the Hoeffding bound or Chernoff bound. The algorithm is a form of online machine learning.

Fuzzy/Soft Decision Trees

Decision trees are interpretable, efficient, problem independent and can treat large scale applications, but they are also recognised as highly unstable. Decision trees can be susceptible to noise and often have high variance. Fuzzy logic can improve this flaw of decision tree models and improve prediction accuracy and increase stability by a factor of 2 at the parameter level against classical decision trees (Wehenkel, Olaru, 2003). Here each input/output pair used to train the tree is characterised by a membership function from [0,1] to some set from the possible universe of objects

(input/output pairs). Within each internal node, the test occurs on a certain attribute (input) and uses two parameters, which characterise the discriminator function which fuzzily splits the local set of objects.

A widely used discriminator function is piecewise linear and contains two parameters, ***A***, which is the location of the cut point and ***B*** which is the width of the transition region on the attribute chosen. Some objects are directed to the left successor in a tree and others to the right. Objects which exist in the overlap region go to both successors.

The larger the transition region ***B***, the larger the overlap and the decision is correspondingly softer. Instances may propagate through all paths or only one. An instance has a membership degree attached to a node. The node can be seen as a fuzzy set. When the instance reaches the leaves, output estimations are aggregated by all the leaves and put through a defuzzification function to obtain the final membership degree to a target class.

Soft decision trees are grown, added in a top down fashion until stopping criteria are met, the grown tree is pruned using a pruning set, followed by a refitting or backfitting step. A final test set is used to evaluate the predictive accuracy of these trees.
Growing:
1. A method to select a fuzzy split.
2. A rule for determining when a node should be considered a leaf.
3. A rule for assigning a label to every leaf.

At each internal node, only one of the attributes is chosen to be split on. These attributes alternate successively.

UC(0) is the degree of membership of an object O to the class C.
UNC(0) is the membership degree as estimated by a tree.

Automatic fuzzy partitioning of a node

There exist fuzzy decision tree induction methods that assume the discriminator functions are defined a priori. Here, growing the tree also implies the automatic generation of the best fuzzy split.

Objective: Given S, a fuzzy set in a soft decision tree:
1. Find attribute a
2. Threshold ***A***
3. Width ***B***
4. successor labels Ll and Lr.

The root is allocated to have a membership degree of one since in most problems all objects are equally weighted at the beginning. The membership degree to a class C, for a certain object o is the average of all the labels L attached to the leaves, weighted by the membership degrees of the object to the fuzzy subsets of these leaves.

Probability estimating decision trees

Obtaining the frequency from class counts may not give optimal probability estimates as a small leaf can potentially give optimistic estimates. In order to solve this Provost & Domingos (2003), smooth frequency based estimates by taking into account the relative coverage of the leaves and the original class distribution. Laplace smoothing at the leaves can also improve probability estimates.

Pruning Trees

Schafer (1993) observes that there is no statistical analysis which identifies pruning as a correct way to reduce the true prediction error. Instead, pruning is simply a bias for simpler models. Understanding what exactly this bias is, can be helpful in choosing the right pruning strategy.

Post pruning can be inefficient, as often trees can be very large and reducing them afterwards takes computational time. Alternatively, pre-pruning the tree can be more efficient, but there is a risk of choosing a sub optimal tree.

Pruning can be seen as a search problem for finding the most optimal tree. In CART (Breiman et al. 1984), the error complexity algorithm is used, whereby a cost is imposed for additional leaves within a tree. Alternatively, a holdout set can be used to obtain estimates of the prediction error. Trees with the smallest error are chosen, or the smallest trees with error in a certain range are chosen.

Alternatively to the cart method is the m estimate method combined with RETIS. This runs in a bottom up fashion through all of the nodes of the tree and at each node compares the error of each inner node and the weighted error of the subtree at T. If the error of the node is less than the error of the subtree then the tree is pruned. One question that is raised is how to discover the error. Cestnik, 1990, uses a measure called an *m estimator* which takes into account the prior and posterior knowledge of the data distributions. M5 (Quinlan 1992) uses a different pruning strategy as it contains multivariate linear models in its leaves. M5 calculates the mean absolute deviation of the linear model and is multiplied by a heuristic factor. This is compared to the root of the subtree and pruned accordingly.

Pruning can also occur via tree selection. There exist an exponential number of potential trees and exhaustive search is far beyond the limits of current computation power. So far, I have examined pruning trees as a two step process. In the first step, a set of pruned trees are generated and evaluated, and in the second step one is chosen. Single step methods decide on a local level whether to prune a node or not, whereas two step models choose over a wide set of different trees. This gives two step processes flexibility.

Within the generation of the tree set, there is a further distinction between *optimal* pruning algorithms and *nested* pruning algorithms. (Breiman et al. 1984) produce a set of trees that decrease in size by one node at a time, ensuring the tree in a sequence is the highest accuracy for a tree of that size. The algorithm is based on dynamic programming and occurs efficiently in non noisy domains. Here a noisy domain is one in which either the class or/and the attribute is incorrectly labelled. It is an example of an optimal algorithm. A nested algorithm generates a sequence of trees, where each tree is constructed through taking a previous tree and pruning some node off it.

The generation of trees is the first step, previously, we saw that CART uses an error complexity algorithm to generate a sequence of nested pruned trees.

Another method is weakest link pruning (Breiman et al. 1984). Here, a tree is recursively pruned such that the ratio delta/n is minimised where n is the number of pruned nodes and delta is the increase in error. This is advantageous as it only uses training cases and it is compatible with resampling.

Stopping Criteria

A brief summary of commonly used stopping criteria are listed here:
All instances in the training set belong to some instance y. The maximum tree depth has been reached. The number of cases in the terminal node is too small. If the node is split, the number of instances in a child is less than the minimum allowed. The splitting criteria is not above a certain threshold.

Categorical features
The model produced, at some point should support ordered (numeric) and unordered (categorical) features.

Production of a model

In order to produce the following model, I used an ad hoc empirical approach i.e. observing many decision tree types and deducing their structure. Given more time, I would have researched mathematical fields such as detection theory or model theory and found a more abstract way to represent decision trees. From the start, I found there was no easy

way to capture the generalisation of decision trees to account for all of the variation that I have witnessed. I suspect that a priori analytical methods within mathematics are easier to generalise, as they are simpler to formally specify in advance.

Due to time constraints, the initial model in initial_attempt.hs was created. This is a very simple, un optimal model that I made to ensure a code base existed by the end of the project. Later attempts to generalise occurred in final_attempt.hs

The algorithm that I start with is the GrowTree and PruneTree algorithms given by Peter Flach (Introduction to machine learning), which is similar to CART (Breiman et al. 1984)

In the following model, the classification use case of decision trees is examined. Given more time, I will investigate other forms of trees. Training and test data sets are assumed to be representative, and I provide an implementation of the Gini Impurity measure rather than the square root of the Gini Impurity as a splitting criterion. If at any point the data sampling is unrepresentative, one can account for an observed distorted class ratio by dividing positive counts by the true class ratio. The reduced error pruning algorithm is also modelled, which aims to reduce overfitting i.e. the existence of a low bias but a very high variance.

The best splits are found at points where numeric features and categorical features change on the sorted training data which are equivalent to points on the convex hull of the ROC curve, albeit in this very simplistic example. Breiman et al. (1984) and Fayyad and Irani (1992a) proved that this results in the best split, which means that one only has to look for O(n-1) splits rather than O(2^(n-1)) splits in the naive case.

This can be identified by examining the function splittingPoints:

```
splittingPoints input_data = intersect (filterMaybe diffAdjacent classes) (filterMaybe diffAdjacent values)
```

The file decision_trees.hs represents an embedding of a language used to describe decision trees.

The decision tree model created has the following a priori limitations:

1) Binary splits
2) Binary classes (A and B)
3) Integer number features only
4) If training set has identical values for two different attribute belonging to different classes, an exception is raised.
5) The number of splits is bounded above by the number of classes.

Some examples of functions provided include:

```
growTree :: [DataPoint] -> ImpurityMeasure -> DTree Split LeafData

pruneTree :: DTree Split LeafData -> [DataPoint] -> DTree Split LeafData

bestSplit :: [DataPoint] -> ImpurityMeasure -> DataSplit

predict :: [Data1D] -> DTree Split LeafData -> [Class]

giniIndex ::ImpurityMeasure

type ImpurityMeasure = DataSplit -> Double

printTree :: DTree Split LeafData -> IO()
```

The homogeneity criterion is that there are no A's in a leaf and more than or equal to one B class and vice versa.

After loading the module; simply type aPrint then bPrint then cPrint then dPrint in GHC's interactive mode to see two trees and their pruned versions. b is the identity pruning, i.e. pruning on the same set as the training data and therefore exhibits no change. Type predict [1,2] a to see the prediction of numbers 1 and 2 on tree a.

In addition another function implemented is printComprehension

```
printComprehension :: DTree Split LeafData -> Data1D -> IO()
```

This will, given a tree and numeric data, predict a class and 'explain' how it got there.

e.g. `printComprehension a (1)`

yields:

```
1 is smaller than 3 (Left)
1 is larger than -1 (Right)
1 is smaller than 2 (Left)


----------------------
=>Predicted Class:B
```

An example of a printed tree is as follows. I did not have time to construct a GUI.

```
  |Br <= 3.0
  |       |Br <= -1.0
  |       |       |Leaf : A                 as:1 bs:0
  |       |       |Br <= 1.5
  |       |       |       |Leaf : B         as:0 bs:1
  |       |       |       |Leaf : A         as:1 bs:0
  |       |Leaf : B                         as:0 bs:2
```

An extension to random forests.

One can see how easy it is to extend the model to include something like random forests with simple functional composition.

On the following page, in only a few lines can the bagging method (Breiman et al. 1996) be implemented.

Unfortunately I did not have time to implement the uniform method which would simply randomly choose a uniform distribution over the training data so all of the trees give precisely the same result.

By simply typing betterAcc at the ghci prompt, you will see the result. Typing averages will show you the result for each decision tree.

I feel the line that captures the simplicity is:

```
randomForest dps imp = map f uniformSample
```

```
rFS = randomForest training_data_one giniIndex
betterAcc = average (map average (map (predict dps) rFS))
averages = (map average (map (predict dps) rFS))

randomForest :: [DataPoint] -> ImpurityMeasure -> [DTree Split LeafData]
randomForest dps imp = map f uniformSample
    where uniformSample = uniform dps 100
          f             = (flip growTree) imp

uniform :: [DataPoint] -> Int -> [[DataPoint]]
uniform dps x = replicate x dps

average :: [Class] -> Class
average xs = if as < bs then B else A
    where as = length (filter (\x -> x == A) xs)
          bs = length (filter (\x -> x == B) xs)
```

Advantages of the approach

I challenge the reader to inspect the code base, a code base, I feel can be easily understood and be examined to be correct (or not!), albeit without formal verification of correctness with respect to pre and post conditions. Each function is clear, even from it's function definition, in part, due to the strong type system in Haskell, which also ensures efficient compile time error checking, and in part due to sensible variable names. Each function is also modular and self contained, especially as conventional side effects do not exist. The equivalent implementation in the machine learning framework Sci-Kit Learn is far longer, far more difficult to understand and has side effects, albeit with a greater degree of task generality. In addition, I feel the documentation provided to the user of Sci-Kit Learn is highly obfuscated and fails to capture the essence of decision trees. During experiences I have had with the package, significant amounts of time were spent dealing with how to display graphs or ensure lists, of types I could not identify, were of a correct layout or size and very little time was spent implementing the abstract ideas I had in my head. Although I do not have an empirical verification of the idea that the implementation I created is more understandable, conversations with close friends yield a high degree of agreement. A human computer interaction study would have to follow in order to deduce a meaningful result

The existence of higher order functions in Haskell allows for simple extensibility of the impurity measures provided. For instance, to add another impurity measure, as long as it is a function of type ImpurityMeasure (`DataSplit -> Double`) then it can be added. Decision trees can represent different things from trees that reduce variance, to trees that cluster data together, to regression trees. Many of these different tree styles are created when the impurity measure changes, for instance the weighted average cluster dissimilarity yields the split dissimilarity, which can be used to form clustering trees. In addition there also exist splitting criterion that choose splits based on highest local AUC, i.e. the area under the ROC curve, rather than accuracy criteria. When decision trees are learned, often there are costs involved with respect to misclassification of different features. In order to deal with this, the training algorithm could incorporate costs into the splitting criteria. However, when the cost of misclassification is not known at training time, the AUC measure is more effective as a splitting mechanism (Flach et al. 2002). If a measure is to be maximised, then one can feed in 1/measure into the model and it is the same as minimising this.

The equivalent Sci Kit Learn source code for decision trees inherits from this.

https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/tree/tree.py

This code has more flexibility, but also it is far longer, includes far more hidden function calls, far more frameworks and is far harder to understand then the code I have written.

Evaluation of the model

Due to a lack of familiarity with the concept of monadicity, which models stateful computation, something that is taken for granted in languages such as C, a full and proper implementation was not created. Pragmatically, the DTree data type can be extracted from the module created and it's structure can be violated, which in turn would violate the correctness of the model. In future, I would spend more time analysing the abstraction of decision trees and testing the models flexibility with respect to new concepts. There are many other functions, impurity measures and concepts that could be implemented. I would also explore what it means to have composition within the language and how that interfaces with other areas of machine learning. In addition by understanding the DSL and it's necessary restrictions, domain specific optimisations can be added, further improving performance.

In addition to the above, I would have liked to construct a rich algebra describing decision trees with simple modular operations that are in some sense axiomatic. I was unable to formulate these operations as I do not feel I had enough time to really understand decision trees on a level that was deep enough.

A hypothesis that I would like to investigate in the future is whether, if, the scope of the language is widened to account for the kinds of functions that a package such as Sci-Kit Learn provides, is it necessary that the language becomes far more complicated and less understandable.

In future, I would have looked at more tree models, included more generic types into Haskell, such that the model trivially works with more types of data, subject to a set of minimal mathematical and abstract requirements that I would have understood and defined, for instance:

```
f :: (Num a) => a -> a -> a
```

Here, as long as a is a descendent of the number class, then the function works. This would be more effective than specifying int as it can be more general. This would have enabled me to have more attributes and a greater degree of extensibility.

Not only would the abstraction occur within Haskell's type system, I would also attempt to fit the model to the examples of decision trees listed above (and hopefully avoid overfitting :)). Due to time constraints, I was unable to complete this.

Given time I would have extended the model to be able to deal with all real valued values. This should not increase the time complexity of the model. It would still run in O(n-1) where n is the number of training examples.
Another point I would like to make is in light of various reproducibility crises within computer science, such that often performance improvements of models are difficult to replicate, DSLs which allow quick and efficient implementations of various models can allow the models to be tested with minimal effort.

Random forests in the model I have constructed are incredibly easy to create and would consist of a few function compositions. In Sci Kit Learn,

Language Composition

Given time I would re-implement the function definitions of the tree growing algorithm and try and ensure it generalises all of the concepts that I discussed in the first section. In addition, I would discuss abstract mathematical models related to algebras. A multi sorted algebra is a collection of sets called carriers together with functions which contain operations and constants. A rough intuition of what they are may allow for a further level of abstraction.

For instance, the ImpurityMeasure type defined in my program has the type:

```
type ImpurityMeasure = DataSplit -> Double
```

However, one implementation of an algorithm may return, instead of a double, a tuple of two doubles:

```
(Double, Double)
```

The model I have clearly cannot cope with this. I posit that this is due to an improper understanding of decision trees and abstract maths such as algebras.

Understanding how to add compositionally to a language requires an intimate understanding of algebras, monads (Moggi) and perhaps even monadic transformers. The reason why monads are useful is because they are a good example of data abstraction. With further time I would investigate this further. I would also investigate functional reactive programming and how the modelling of data streams could potentially improve machine learning.

Can dependent type programming improve the state of machine learning even further.

Above, I have emphasised the use of abstraction to improve programming design and to bridge the gap between programmers who often write things that do stateful computation and mathematicians who often write things that state things (some say tautologies :)), although with algorithms and monads etc, this difference is not concrete obviously. A question that I ask here is can we abstract even more, from the point of view of programming language design.

The Curry Howard Isomorphism proves there is a direct correspondence between propositions and types. Recent advances in homotopy type theory can improve the state of programming. The Curry Howard Isomorphism states:
"In Logic, evert proposition can be viewed as a type whose members are the proofs of the propositions"

Types can also be viewed by sets and have many of the same operations. We can posit that when two different things have the same structure, this implies that they are the same thing, i.e. types are propositions themselves. Martin Lof Type Theory implements this on the level of propositions and even predicate logic. This system enables further abstraction. Types can be defined in terms of natural numbers, within this domain, for instance:

```
Type x :: x % 2 == 0
```

Haskell does not yet have so called dependent types.

What does this imply?

1. Less time debugging. Languages which do not have the full type system, benefit from the type theory paradigm. Haskell has a type system, albeit less strict than those of say Asda or Idris. Djinn is a Haskell program that attempts to reverse engineer algorithms from their type definition, e.g.

```
—> f ? a->a
f :: a -> a
f x1 = x1
```

If we are given function types and dependent types, we can also specify properties of the function as part of the type for instance:

```
f :: Mat -> Mat -> Mat : f(x,f(y,z)) = f(f(x,y),z)}
```

i.e. here we specify the associativity of matrix multiplication. A program such as Djinn can then go ahead find a function that does this. If we start increasing the complexity of the algorithms, maybe the algorithm can give us some crazy new ideas.

2. Provably correct code. We spend our lives writing and debugging programs, in many cases hoping they are correct. Why can we not prove our programs correct subject to some formal specification. In future, unit testing may be seen as a primitive way to verify program correctness. If Martin Lof Type systems were implemented, more time would be spent proving function properties than specifying algorithms for computing them.

3. Cross language datatype compatibility. Ints in C are different from ints in javascript. Formal and overly complex specifications such as:

https://en.wikipedia.org/wiki/IEEE_floating_point

would be abolished and replaced.

Further work to MLTT exists within the interpretation of types as, instead of sets, as categories, which are similar to sets but include more structure, i.e. instead of just boxes containing elements they are the boxes and processes relating these things. Homotopy Type Theory as this is known gives a more concrete way to understand if two types are identical or not.

Given time on the project, I would implement a language within a dependently typed language such as Agda or Coq and explore in a very pragmatic way, how programming language design for machine learning can be improved.

A brief note on computational learning theory

By attempting to abstract, I will analyse the abstraction of the learning mechanism itself. Learning divides into induction, abduction, deduction and more. There is both a philosophy and science that studies these methods. Computational learning theory devotes itself to studying the design of machine learning algorithms. Differences in approaches to learning theory make assumptions on the inference techniques with respect to the data.

Probably approximately correct learning: The learner receives samples and must select a generalisation function or hypothesis from a class of functions.

VC Theory: This theory concerns itself with questions such as:

1. Do learning algorithms necessarily use the empirical minimisation principle.
2. What characterises the learning convergence rate.
3. How is the generalising ability of a learning process limited.

Bayesian Inference: This refers to the update hypothesis probabilities as more information becomes available.

Algorithmic Learning Theory: This method assumes that data points are not random samples and was introduced by E. Mark Gold. The theory assumes that evidence is given and after each piece of evidence, the hypothesis is correct for all instances of the given evidence. If there exists a certain number of steps after which the hypothesis does not change then the learner can “learn a language in a limit”.

Can higher level EDSLs match the efficiency of “close to the metal” languages like C

The Haskell community, in recent years, has enjoyed an improvement in optimisations for GHC, and in addition now has several high performance libraries. It is becoming more possible to write efficient code in Haskell. Languages of the future within machine learning will have to support as many forms of parallelism as possible, from efficient SIMD vectorisation to effective domain decomposition of work items, in order to have strong performance. Currently languages that are most used within the spheres of HPC are CUDA, C and perhaps now, TensorFlow.

In order to exploit the memory hierarchy, some of the most important optimisations one can do are serial, for instance changing non contiguous memory regions e.g. array of structs to a contiguous memory region such as a struct of arrays, which allows for efficient vectorisation, or loop fusion.

I will explain one approach which is automatic SIMD vectorisation for Haskell. Programmers or preferably compilers are responsible for finding parallel computations and expressing them in terms of fixed width vectors. Parallelism occurs here over the loop level and so here this corresponds to identifying loops for which many iterations can occur at once.

A major problem faced by programmers who wish to vectorise their code is one of data dependence. If loops are dependent between each other, such the result of one loop iteration affects another, results can be rendered incorrect. In C, explicit pragmas which the programmer asserts such as `#pragma ivdep` allow the compiler to vectorise code, trusting the programmer. In a functional language, the absence of side effects, eliminates large classes of dependence violations that stop vectorisation from occurring. The Vector library for GHC provide efficient high level abstractions for generating immutable arrays. Functions such as map, fold and zip can compute arrays from other arrays (Orchard ICFP 2013). Intel has taken this idea and produced a compiler for Haskell called the Intel Labs Haskell Research Compiler (Peterson 2013).

In addition, the functional programming paradigm has been incredibly successful for Google in using map reduce to split computation across multiple computers or multiple cpus. This can occur simply and efficiently for statistical techniques that can be written in summation form (Olukotun NIPS 2006).

Conclusion:
Further work has been extensively discussed in the evaluation section of the report. I feel that on balance:

- Functional programming can massively improve the software engineering cycle within machine learning, with respect to a multitude of factors mentioned in detail in the report. Monadicity and monadic transformations and the associated semantic effects of these can be explored further within the literature.
- Model construction can lead to improvements in usability, efficiency, correctness and greater understanding of the field.
- Dependent type programming can perhaps solve ethical issues related to correctness of machine learning algorithms, i.e. if a feature of spam emails is “Black person” for instance. This occurs by showing the model correct to some specification.
- More research is needed on language compositionality and monadicity and how this can create more effective languages.