Machine Learning Algorithms for Binary Classification

Deep Learning vs. Decision Tree Learning

Connor Claypool

Machine Learning for Binary Classification

- Deep learning and decision tree learning
- Each algorithm trains a predictive model using a dataset of samples which include one or more independent input variables and a binary dependent variable
- The model is then validated on new data to determine its accuracy
- This application uses the Banknote Authentication Data Set from the UCI Machine Learning Repository, consisting of four measurements from images of banknotes along with whether the banknote is authentic

Deep Learning: Use of Data Structures

For storing data samples, a `std::vector` was chosen because:

- Data samples are accessed sequentially, and `std::vector` stores its elements contiguously in memory, allowing an advantage over e.g. linked lists and hash tables in terms of locality and the cache
- No lookup or search operations are used, so data structures like sets, maps and hash tables make little sense
- The size of `std::vector` is not fixed at compile time, which would be a cumbersome code requirement
- No insertions or deletions occur once the data is loaded, meaning no expensive reallocations after this

Deep Learning: Use of Data Structures

The independent variables are stored in a custom `Matrix` class template backed by a two-dimensional `std::array` because:

- Linear algebra operations such as dot product, transposition follow defined patterns based on element position so a `std::array`'s $O(1)$ index-based lookup is the best fit
- A two-dimensional `std::array`'s elements are contiguous which provides a locality advantage for the above operations and keeps the dataset contiguous
- Sizes are determined by the number of independent variables and the architecture of the model, and compile-time size definitions mean it is impossible to pass an invalid-sized `Matrix` to e.g. a dot product method

Deep Learning: Use of Data Structures

Matrices of parameters and intermediate results are stored in a custom `GradMatrix` class template which includes another `Matrix` of gradients as a property because:

- Elements and their gradients are often accessed in short succession, so storing the gradients contiguously with the elements provides a locality advantage
- The median training time in nanoseconds across 100 measurements was 1.552e+06 with gradients stored contiguously and 2.522e+06 with storage via a pointer
- Comparing these samples with a Wilcoxon rank-sum test yields a p-value of 1.47578e-30 and an effect size estimate of 959200 nanoseconds

Decision Tree Learning: Use of Data Structures

For storing data samples, a `std::vector` was chosen because:

- Each sample is accessed multiple times but not in sequential order, `std::vector` benefits from contiguous storage and $O(1)$ random access
- No lookup or search operations, so maps, sets & hash tables would not make sense
- Compile-time fixed size would be a cumbersome requirement as for deep learning

Decision Tree Learning: Use of Data Structures

For storing rows within the data table, a `std::array` was used, because:

- Variables within a row are accessed both sequentially and randomly, `std::array` benefits from $O(1)$ random access
- Keys, e.g. strings would make some sense, but they would be duplicated for each row, and integer indices are perfectly suitable
- `std::array` allows contiguity across parent container
- Number of variables is known at compile time

Decision Tree Learning: Use of Data Structures

For storing the indices of the data samples, a `std::vector` was used because:

- This allows each node to maintain an iterator to the start and end points of its group, and partition (sort) its group into two and pass the relevant iterators to its child nodes' constructors
- This means the data can be stored contiguously (as opposed to sorting a collection of pointers to rows) with no need to rearrange whole rows in memory
- `std::vector` supports sorting (unlike sets, maps & hash tables), a dynamic size and is stored contiguously

Decision Tree Learning: Use of Data Structures

- The median training time in nanoseconds across 100 measurements was 2.74747e+07 with rows stored contiguously and 3.06321e+07 with rows stored via pointers
- Comparing these samples with a Wilcoxon rank-sum test yields a p-value of 4.21177e-27 and an effect size estimate of 3.15947e+06 nanoseconds

Decision Tree Learning: Use of Data Structures

The decision tree model is stored in a binary tree-like structure because:

- The decision process is binary: a node’s split point partitions a group of samples into two, with leaf nodes representing a prediction value
- No searching, sorting, or updating is needed - training and inference both start at root and create/find leaf node(s)
- Small tree size and the low complexity of validation means the code simplicity of storing child nodes via pointers outweighs the possible locality advantages of using arrays

Deep Learning: Time Complexity

Holding the architecture of the model and the number of epochs constant:

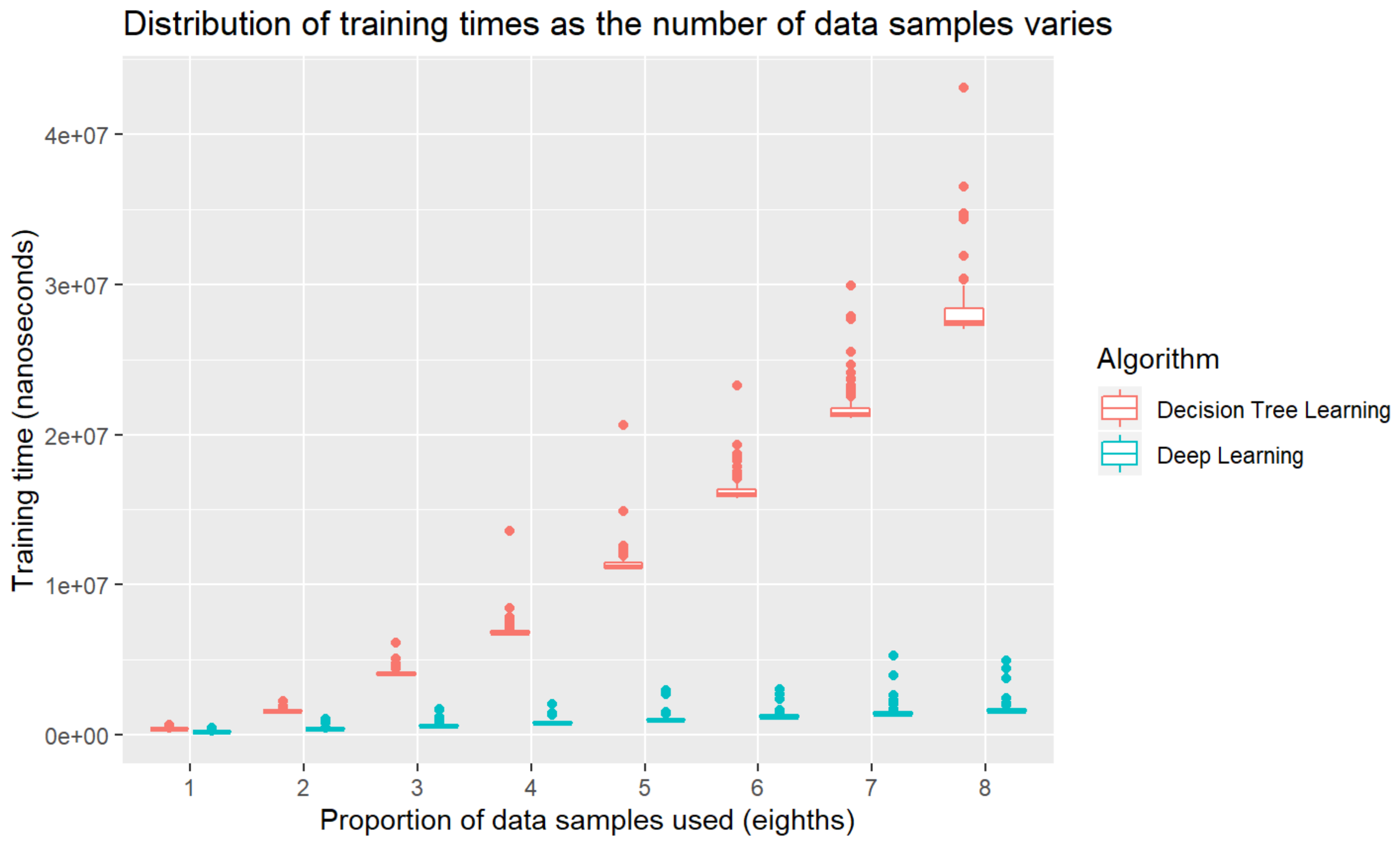
- The time complexity of training the deep learning model is $O(s \times v)$, where s is the number of data samples and v is the number of independent variables per sample
- The time complexity of inference with the deep learning model is also $O(s \times v)$ as the forward pass is the same, without backward pass or update

Decision Tree Learning: Time Complexity

Holding the maximum depth of the model (and thus the number of nodes) constant:

- The time complexity of training the decision tree model is $O(s^2 \times v)$, where s is the number of training samples and v is the number of independent variables per sample
- The time complexity of inference with the decision tree model is $O(s)$, where s is the number of validation samples

Performance Comparison

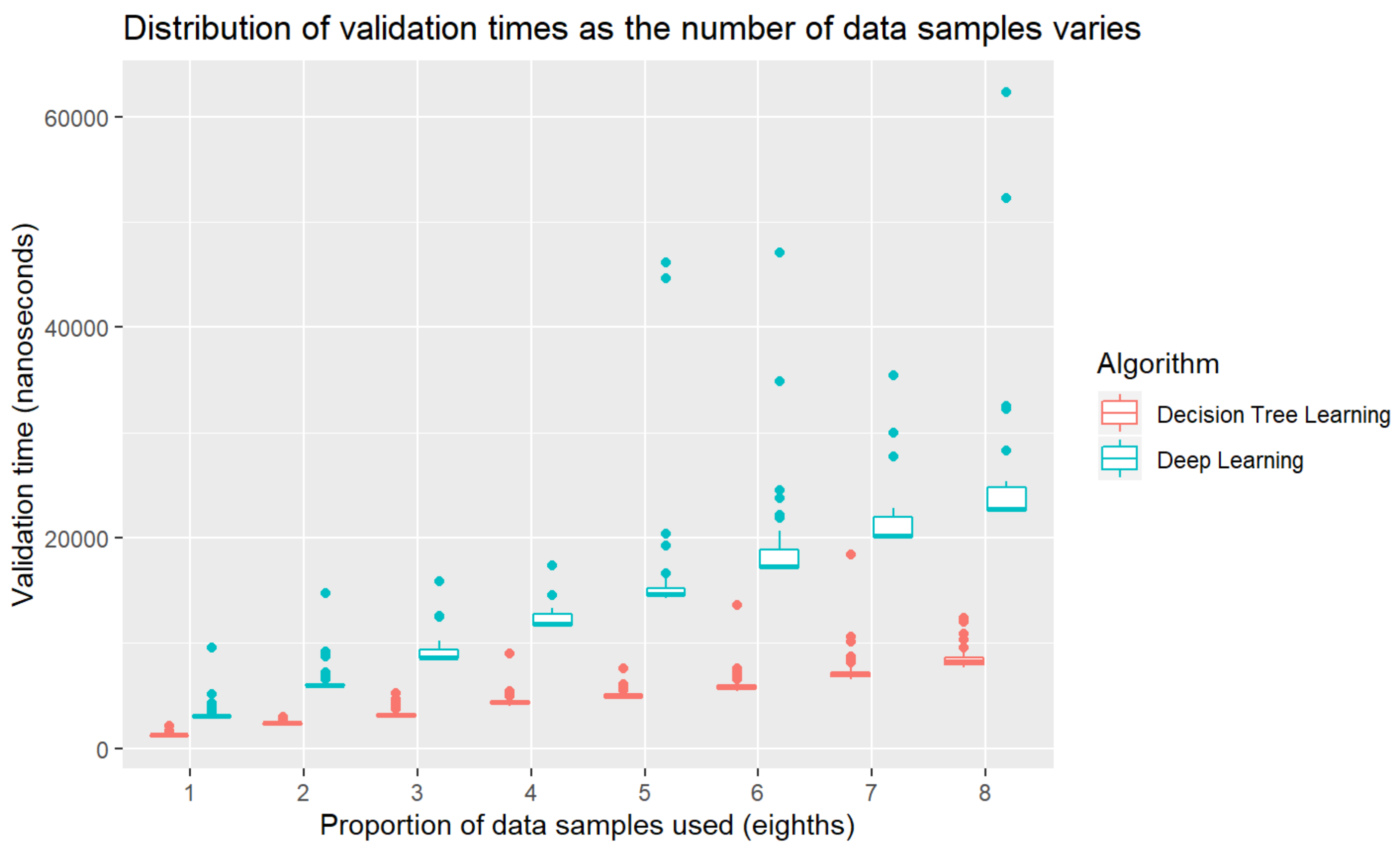


Performance Comparison

Comparing these distributions by algorithm with a Wilcoxon rank-sum test yields the following results:

Samples proportion	p-value	Effect size estimate (nanoseconds)
1	0	202800
2	0	1174900
3	0	3443800
4	0	6007131
5	0	10216400
6	0	14797500
7	0	19943200
8	0	25866608

Performance Comparison

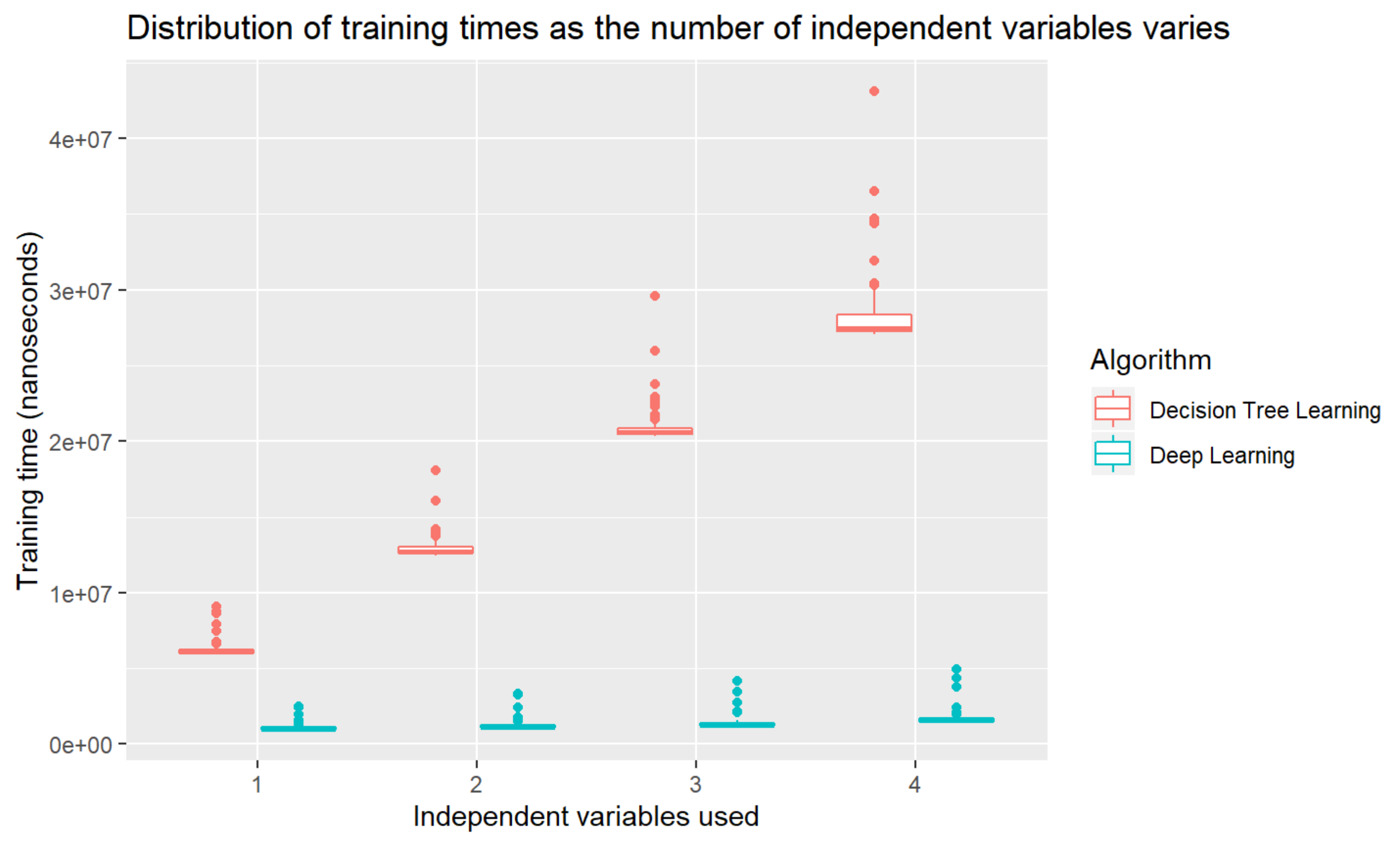


Performance Comparison

Comparing these distributions by algorithm with a Wilcoxon rank-sum test yields the following results:

Samples proportion	p-value	Effect size estimate (nanoseconds)
1	0	-1800
2	0	-3700
3	0	-5400
4	0	-7500
5	0	-9700
6	0	-11600
7	0	-13200
8	0	-14600

Performance Comparison

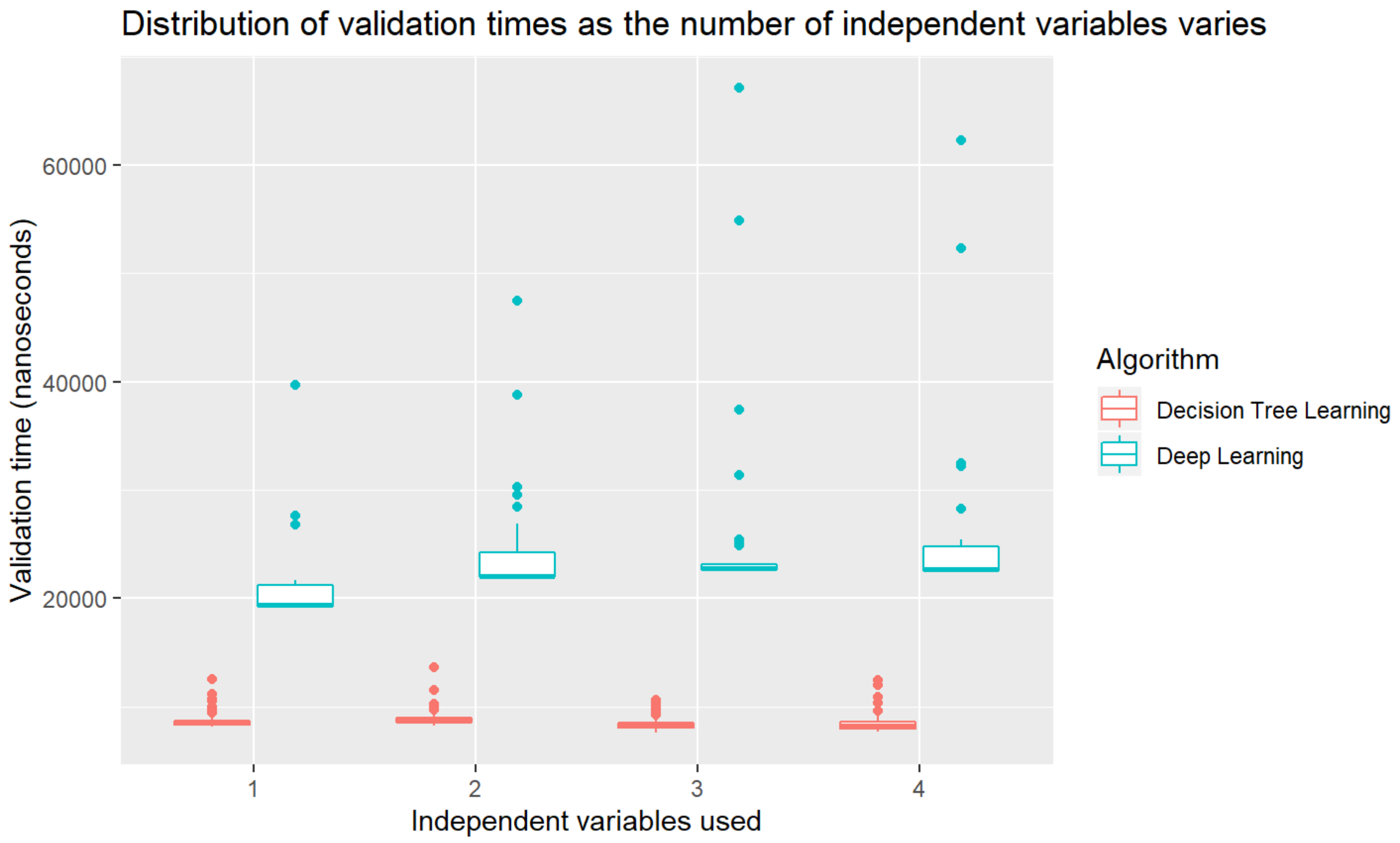


Performance Comparison

Comparing these distributions by algorithm with a Wilcoxon rank-sum test yields the following results:

Independent variables	p-value	Effect size estimate (nanoseconds)
1	0	5072700
2	0	11511937
3	0	19281800
4	0	25866608

Performance Comparison

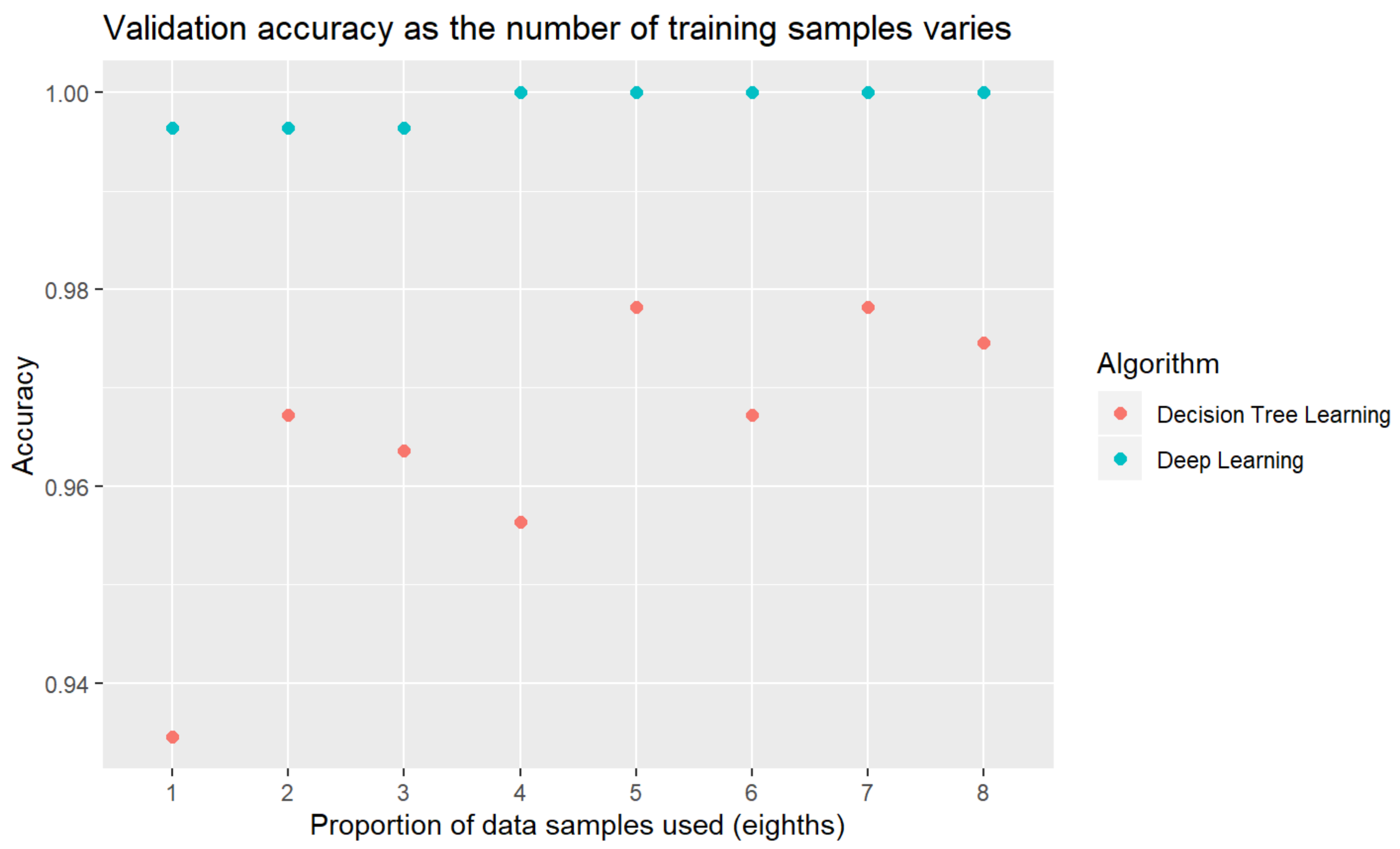


Performance Comparison

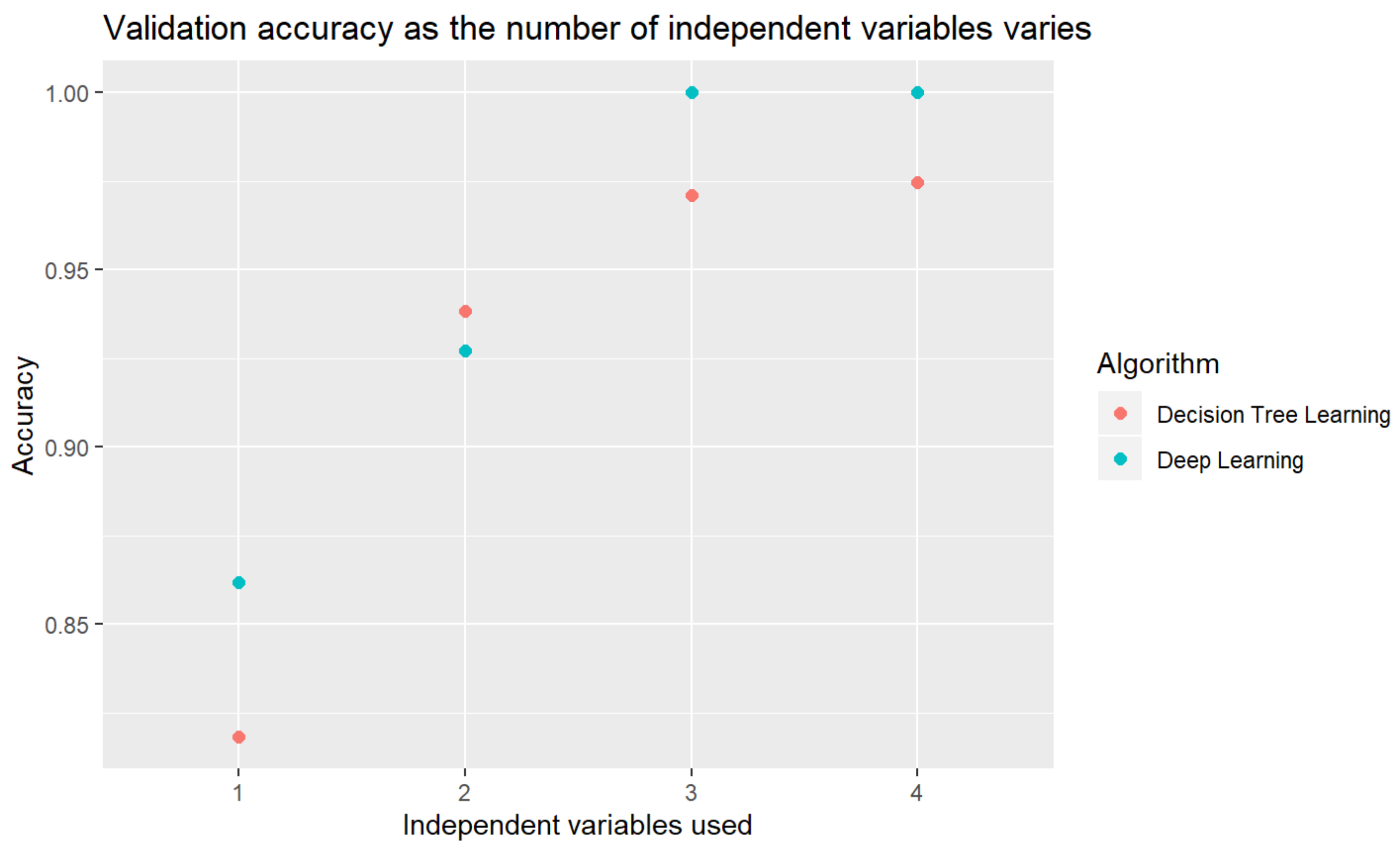
Comparing these distributions by algorithm with a Wilcoxon rank-sum test yields the following results:

Independent variables	p-value	Effect size estimate (nanoseconds)
1	0	-10900
2	0	-13400
3	0	-14600
4	0	-14600

Accuracy Comparison



Accuracy Comparison


Validation accuracy as the number of independent variables varies
Accuracy
Independent variables used
Algorithm
Decision Tree Learning
Deep Learning