CS: 4420
Project Report
05/10/2019

Kevin Blicharski - Recorder
Derek Choi - Checker
Ted Paulsen - Coordinator

Decision Tree Learners in OCaml

Abstract

Decision tree learners are a flexible, extensible model for machine learning. Decision tree learners are favored for use as machine learning models because they can be trained quickly and can be represented in a human readable format such as s-expressions. Decision tree models can be tuned by limiting the maximum depth for leaf nodes. Depth limiting is useful to prevent overfitting of the model. Deeper trees may not be necessary for simpler data sets, in this case, building a deeper tree could actually lower its ability to generalize. Complexity of the underlying relationship of a data set greatly affects the optimal depth for a decision tree learner.

Definitions

S-expression: A format for representing a nested tree data structure, as well as our means for serializing and deserializing the decision tree when saving to and loading from files.
K-fold cross validation: A technique for rotating the training and test example sets so that all data is trained and tested on. It is used to improve the assessment of a model's ability to predict values.
Model selection: A technique for minimizing test error via tuning some parameter — in our case this parameter is tree depth.
Overfitting: Occurring in all types of learners, overfitting is what happens as a result of making too rigid of a model. It can be thought of as “memorizing” the given data, thus losing the ability to generalize. It can be reduced by increasing the number of unique examples.

Motivation

Decision trees learners can be used to represent many different categorical data sets. At each node, a single attribute is selected to partition the data to maximize homogeneity of the partitions. While this technique can be applied to any categorical data set, there are surely certain data sets that are better suited for this kind of learner. What qualities of data make a data set better suited for decision tree learners? And what does this say about the underlying relationship? We hope to answer these questions here.

Additionally, we wanted to investigate the importance of depth in decision tree learners. Specifically, we wanted to understand the effect of adding too many levels to a decision tree and how overfitting can be caught in the model building phase to maximize real world performance of the model.

Methods

Due to the rather large nature of the project and our limited time, we began by planning extensively. We created a roadmap of necessary features and identified areas where group members could work independently, maximizing the time spent developing in parallel. For several critical components of the application, namely defining data structures to represent the decision tree and implementing the core recursive algorithm used to build it, we utilized pair programming to reduce the amount of potential mistakes.

Determining how to parse the CSV files was simple enough -- the csv package was very easy to get working. Afterwards, the first step that was taken was calculating entropy, and thus remainders, for each attribute. This was also fairly simple, as it essentially involved encoding a formula and list operations to partition the example set into a positive set and negative set. Parsing command line arguments was also straightforward, and with this ground-work out of the way, we set out to build the actual tree.

To keep track of all necessary information we chose to encode a node with the following information:

```
{
   depth: int;
   characteristic: string;
   decision: string option;
   remainder: float;
   examples: string list list;
 }
```

Where `examples` is a subset of examples available to split on at the current node, `characteristic` is the characteristic selected to partition on in the parent and `decision` is the value of the `characteristic` taken to get to this node. A decision tree was then defined as an algebraic data type where it is either a leaf node, or a decision node with a list of smaller decision trees. We then created each sub-tree by creating a new node at each level.

After constructing the tree, we needed a way to serialize and deserialize it so that we could write it to a file and read it back in when the classification function was run. This proved to be one of the most challenging parts of the project, surprisingly. We discussed using JSON for this encoding, but after some cursory investigation it seemed like s-expressions were better suited to the task. The libraries were very difficult to understand and work with, namely because there was essentially zero user-facing documentation and the library authors had just recently moved from one implementation to another, leading to confusion when parsing online resources. Eventually, it was realized that after adding a preprocessor annotation, all that needed to be done was to write two functions -- one to turn a decision tree into an s-expression, and another for the reverse. The former was simple enough due to the way in which our trees are defined recursively. The latter was much more challenging, involving the use of four mutually recursive functions. However, it worked as intended, and now we could write the classifier.

The classifier was fairly trivial -- essentially recursively traversing the loaded tree, but for all examples we fed to it. This was by all means a far simpler task than constructing the tree or serializing/deserializing it. This task was completed relatively quickly, and with it, part one was completed.

For part two, the depth-limiting was essentially already implemented as each node stored its depth. To implement depth-limiting then all that was required was to add a case to the if statement responsible for creating leaf nodes and an option type for the depth being passed into the tree creation function. Afterwards, the only thing left was to implement the k-fold cross validation on multiple depths. First we took the examples and then split them into 4 different folds. We first took the whole group and split it in half by taking the list of examples and an empty list, then taking examples one at a time and putting them in the empty list until the empty list had as many examples as the initial list. Then we applied this same halving algorithm to both halves to get the 4 groups.

Then we iterated through each depth up to the given maximum depth. For each depth, we also iterated through each fold and accumulated errors. We put each of the 4 groups into a list and took the first group out of the list to use as the validation set. Then the rest of the groups were used as the training set. We then trained a tree off the training set up to a certain depth, and then used our classifier from part 1 to find the total number of misclassifications both in the training set and the validation set. Training the tree sometimes created an issue where since we were validating off examples that were not trained on, there would sometimes not be a branch in the tree that fit the example. This issue was resolved by classifying the examples that found no suitable branch as misclassifications. Then we take the group used as validation and move it to the back of the list, and repeat this until every group has been used as validation.

For each tree trained off a different set, we accumulate the total number of misclassifications in the different training sets and validation sets. Then we get the average error by dividing the total number of misclassifications by the total number of examples that these misclassifications come from. More specifically, we divide the number of misclassifications in the validation sets by the total number of examples, since every example was used as part of the validation set once. We divide the number of misclassifications in the training sets by the total number of examples, multiplied by 3 since every example was used as training three separate times. This gives us the average error of each fold for both validation and training sets. Then we iterate through each depth up to the given maximum depth to find which depth has the least average validation error.

This iteration was done through a for loop and the least average validation error and the depth that produced it were kept track of by using references, or mutable variables. After we iterated through all the depths, we took the depth that produced the least average validation error and train it on the entire set of examples. We then use our classifier on the resulting tree to determine how many training examples are misclassified and print out the result.

Results

The results for testing KFold for the 5 different data sets can be seen below. These graphs were created by extracting the data printed out for doing KFold cross-validation for depths up to 10 and then putting the data into an Excel spreadsheet and creating these line graphs from there. For each graph, the x axis represents the maximum allowed depth of the tree and the y axis shows the average error taken over each fold in the 4-fold training. The blue line indicates the training error while the orange line indicates the validation error.

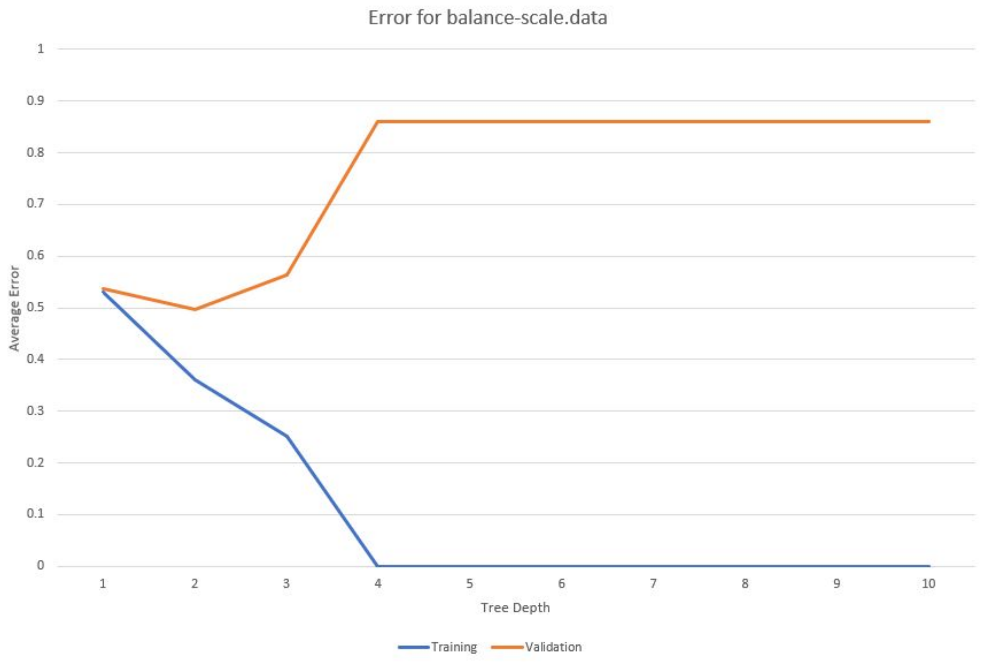


Figure 1: KFold error for the balance scale dataset.

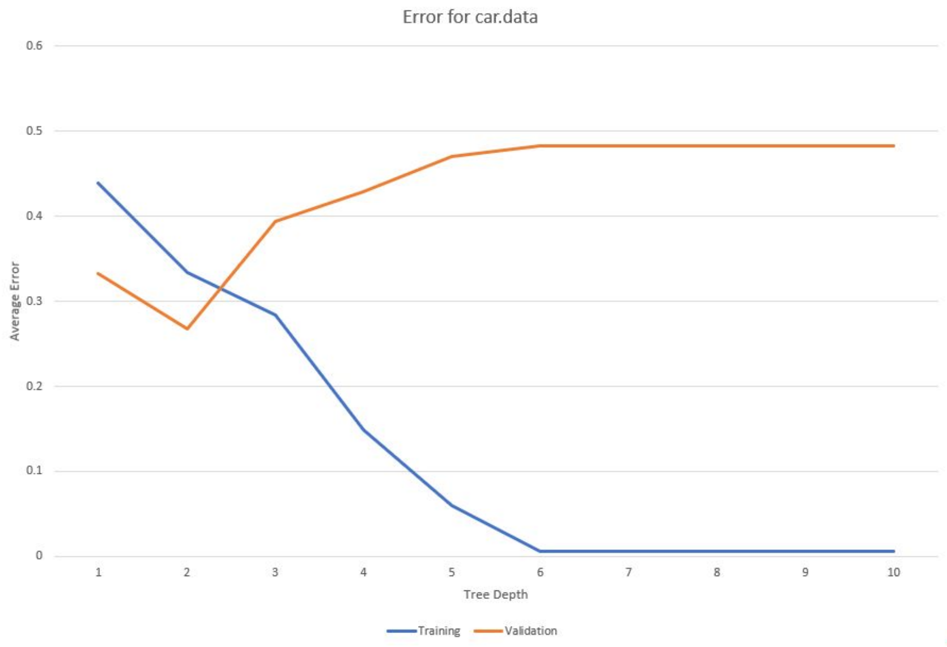


Figure 2: KFold error for the car dataset.

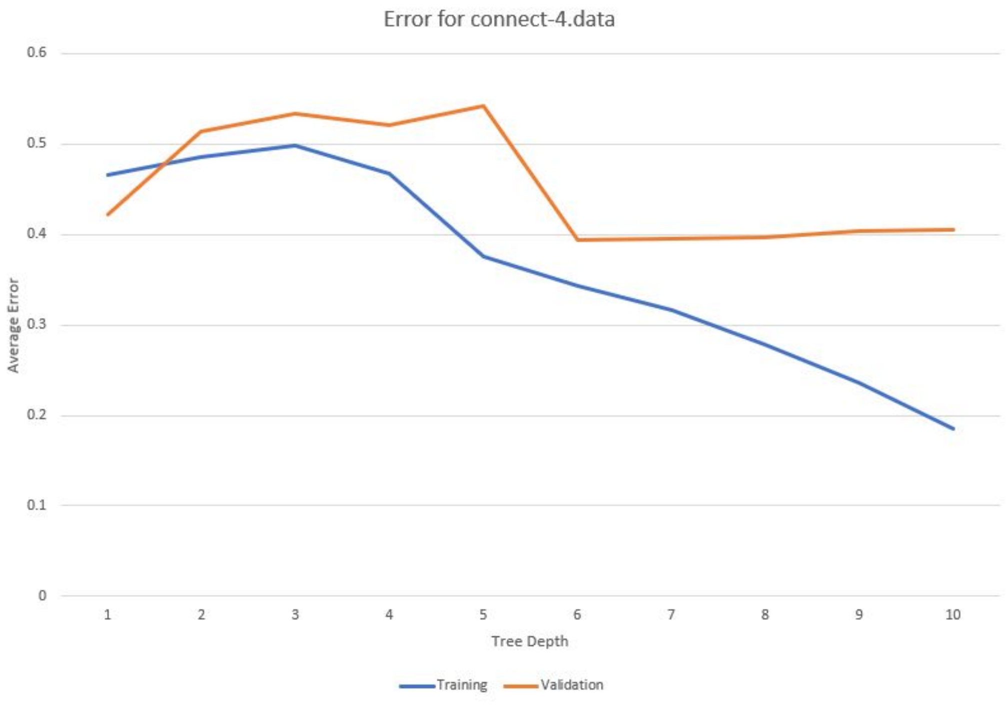


Figure 3: KFold error for the Connect 4 dataset.

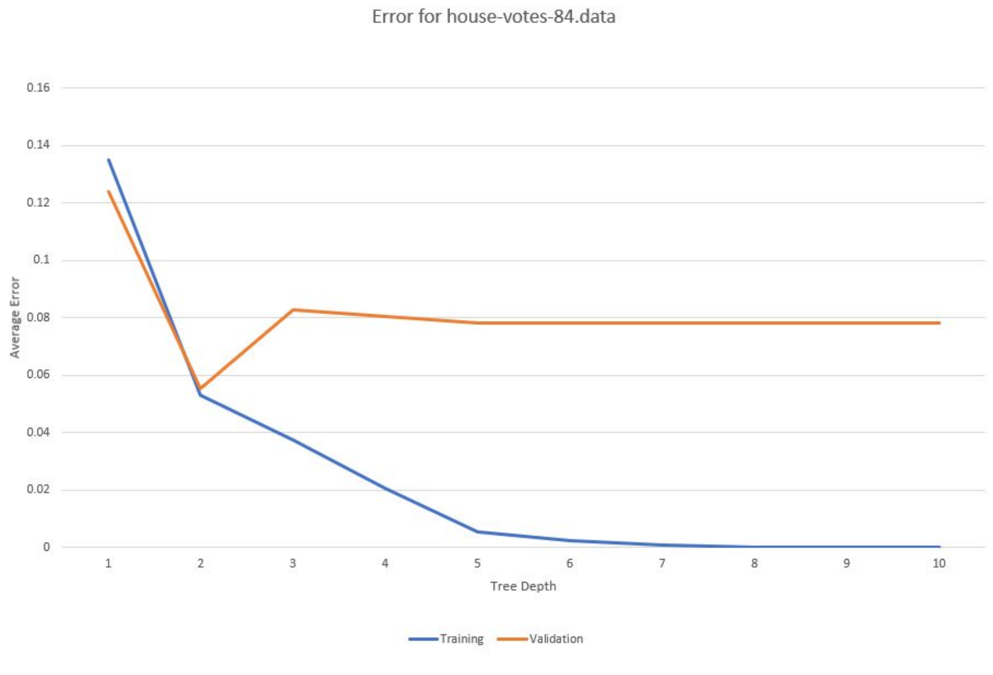


Figure 4: KFold error for the 1984 House votes dataset.

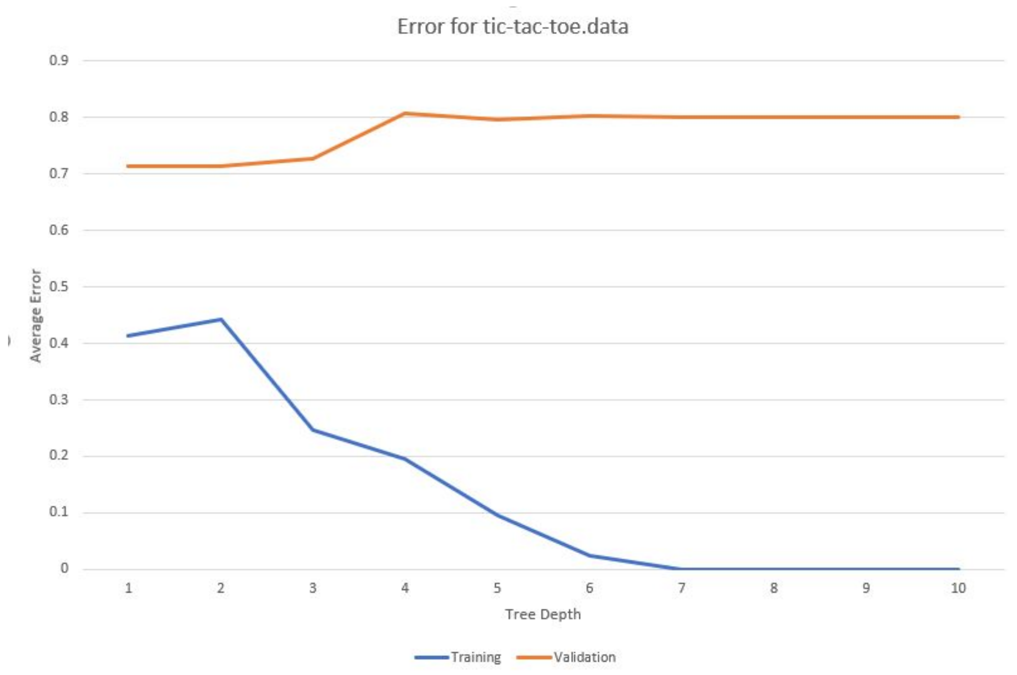


Figure 5: KFold error for the tic tac toe dataset.

Discussion

We can learn things by looking into some trends that can be seen in the KFold error graphs. For example, as depth increases, training error decreases, usually approaching zero and staying there once it reaches that point. On the other hand, validation error was much less predictable, but for several data sets the validation error would initially decrease before increasing again later on. These two trends are likely due to the decision tree overfitting the training data as max depth allowed increases, such that rather than learning the patterns in the data, the tree is mostly just memorizing examples. Overfitting will almost always decrease training error while increasing validation error.

Another thing to notice is that for several datasets, the errors stop changing after a certain depth. This happens for each dataset except for the Connect 4 dataset which can be seen in figure 3, which is a significantly larger dataset. Since this issue is only not seen in the much larger Connect 4 dataset, it is most likely the case that the rest of the datasets had fit the training data as well as possible at a depth less than 10. This is supported by the fact that generally, the errors stopped changing after the depth where the training error became 0 or got very close to it. The reason why the trees produced from the Connect 4 dataset doesn't have this issue would be because it is significantly larger and more complex, and so likely requires a tree of depth greater than 10 before it can perfectly fit the training data.

It is interesting that the only dataset for which our decision tree worked very well on was the House voting records. This seems to make intuitive sense -- representatives often vote along partisan lines, so it seems natural that one could deduce party membership by looking at voting histories. On the other hand, it is intuitive that it would perform poorly for the more complicated problems. For tic tac toe, even though it performs rather poorly at determining whether X will win a given game (and the authors of the dataset state as much regarding decision trees in the .name file they provide), it is interesting that the best decision tree for this problem picks the middle square as the sole characteristic to branch on. For anyone who is familiar with tic-tac-toe, this also makes intuitive sense as this is the most "important" square in a sense. These results seem to confirm that our decision tree construction works as expected.

Acknowledgements

Jane Street for providing us with the s-expression libraries
Christophe Troestler for the csv library

References

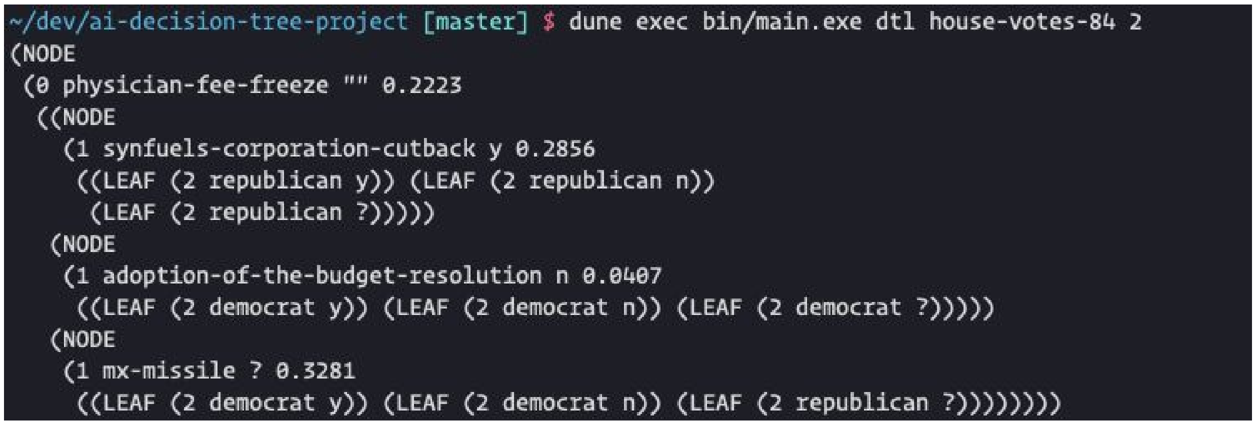
OCaml Documentation
https://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html
https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html
https://caml.inria.fr/pub/docs/manual-ocaml/libref/Printf.html

Libraries
https://github.com/janestreet/sexplib
https://github.com/janestreet/ppx_sexp_conv
https://github.com/Chris00/ocaml-csv

Datasets
http://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe+Endgame
http://archive.ics.uci.edu/ml/datasets/Balance+Scale
http://archive.ics.uci.edu/ml/datasets/Congressional+Voting+Records
http://archive.ics.uci.edu/ml/datasets/Car+Evaluation
http://archive.ics.uci.edu/ml/datasets/Connect-4

Appendices

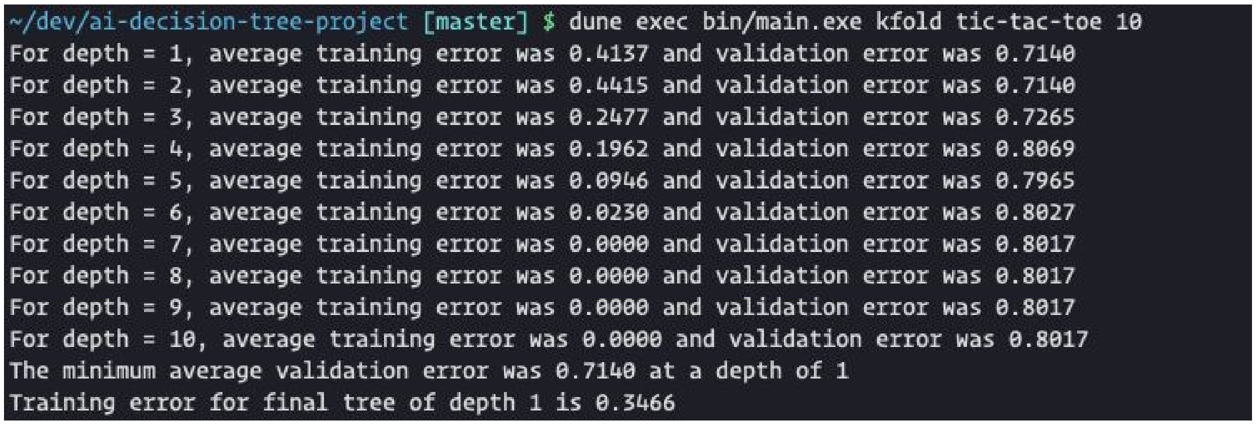
Appendix A: Sample decision tree



```
~/dev/ai-decision-tree-project [master] $ dune exec bin/main.exe dtl house-votes-84 2
(NODE
 (0 physician-fee-freeze "" 0.2223
  ((NODE
    (1 synfuels-corporation-cutback y 0.2856
     ((LEAF (2 republican y)) (LEAF (2 republican n))
      (LEAF (2 republican ?)))))
   (NODE
    (1 adoption-of-the-budget-resolution n 0.0407
     ((LEAF (2 democrat y)) (LEAF (2 democrat n)) (LEAF (2 democrat ?)))))
   (NODE
    (1 mx-missile ? 0.3281
     ((LEAF (2 democrat y)) (LEAF (2 democrat n)) (LEAF (2 republican ?))))))))
```

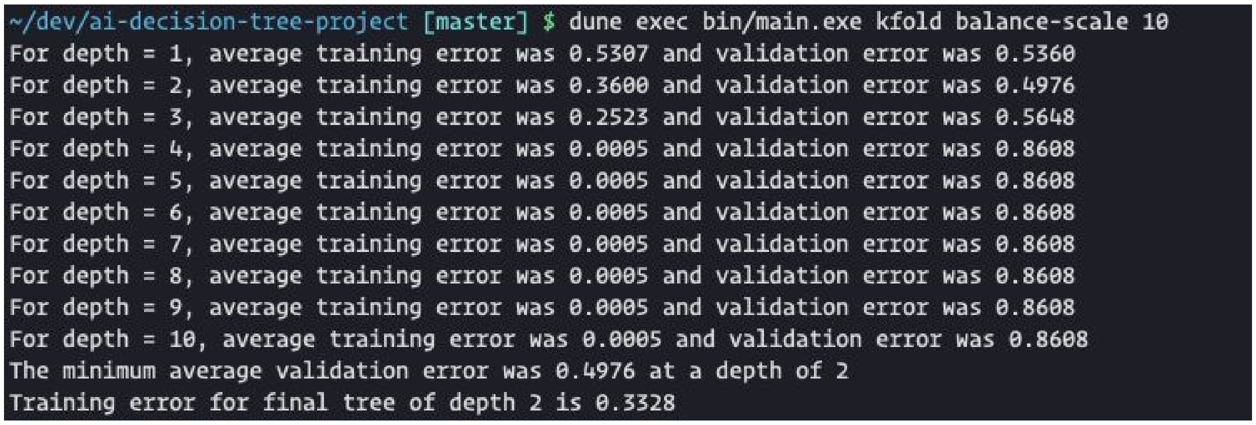
Appendix B: K-Fold

1: Tic-tac-toe



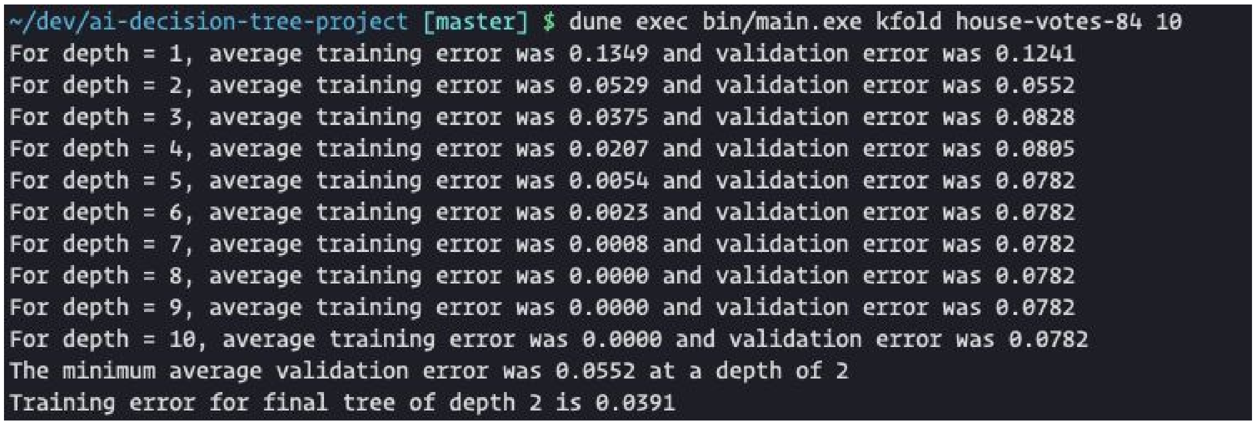
```
~/dev/ai-decision-tree-project [master] $ dune exec bin/main.exe kfold tic-tac-toe 10
For depth = 1, average training error was 0.4137 and validation error was 0.7140
For depth = 2, average training error was 0.4415 and validation error was 0.7140
For depth = 3, average training error was 0.2477 and validation error was 0.7265
For depth = 4, average training error was 0.1962 and validation error was 0.8069
For depth = 5, average training error was 0.0946 and validation error was 0.7965
For depth = 6, average training error was 0.0230 and validation error was 0.8027
For depth = 7, average training error was 0.0000 and validation error was 0.8017
For depth = 8, average training error was 0.0000 and validation error was 0.8017
For depth = 9, average training error was 0.0000 and validation error was 0.8017
For depth = 10, average training error was 0.0000 and validation error was 0.8017
The minimum average validation error was 0.7140 at a depth of 1
Training error for final tree of depth 1 is 0.3466
```

2. Balance scale:



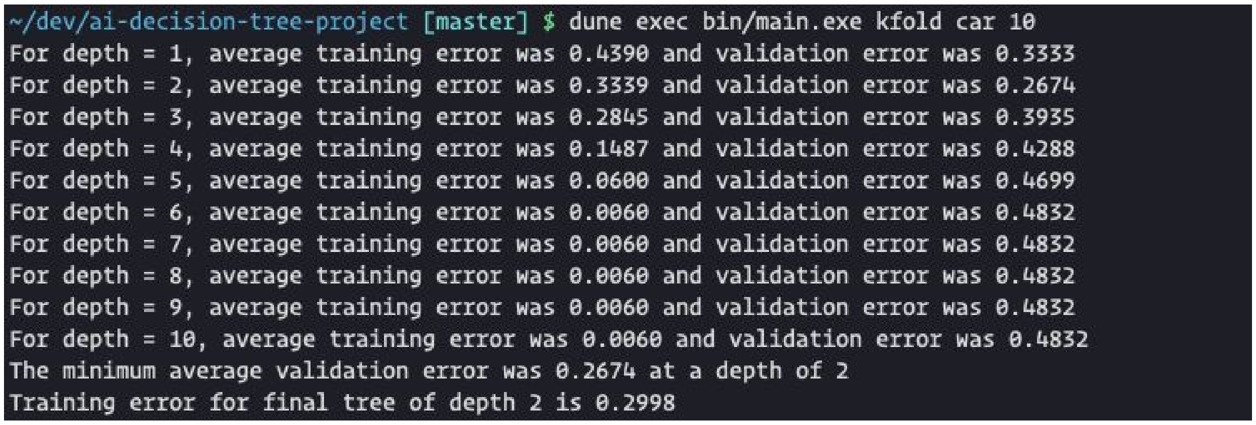
```
~/dev/ai-decision-tree-project [master] $ dune exec bin/main.exe kfold balance-scale 10
For depth = 1, average training error was 0.5307 and validation error was 0.5360
For depth = 2, average training error was 0.3600 and validation error was 0.4976
For depth = 3, average training error was 0.2523 and validation error was 0.5648
For depth = 4, average training error was 0.0005 and validation error was 0.8608
For depth = 5, average training error was 0.0005 and validation error was 0.8608
For depth = 6, average training error was 0.0005 and validation error was 0.8608
For depth = 7, average training error was 0.0005 and validation error was 0.8608
For depth = 8, average training error was 0.0005 and validation error was 0.8608
For depth = 9, average training error was 0.0005 and validation error was 0.8608
For depth = 10, average training error was 0.0005 and validation error was 0.8608
The minimum average validation error was 0.4976 at a depth of 2
Training error for final tree of depth 2 is 0.3328
```

3. Congressional votes '84:



```
~/dev/ai-decision-tree-project [master] $ dune exec bin/main.exe kfold house-votes-84 10
For depth = 1, average training error was 0.1349 and validation error was 0.1241
For depth = 2, average training error was 0.0529 and validation error was 0.0552
For depth = 3, average training error was 0.0375 and validation error was 0.0828
For depth = 4, average training error was 0.0207 and validation error was 0.0805
For depth = 5, average training error was 0.0054 and validation error was 0.0782
For depth = 6, average training error was 0.0023 and validation error was 0.0782
For depth = 7, average training error was 0.0008 and validation error was 0.0782
For depth = 8, average training error was 0.0000 and validation error was 0.0782
For depth = 9, average training error was 0.0000 and validation error was 0.0782
For depth = 10, average training error was 0.0000 and validation error was 0.0782
The minimum average validation error was 0.0552 at a depth of 2
Training error for final tree of depth 2 is 0.0391
```

4. Car evaluation



```
~/dev/ai-decision-tree-project [master] $ dune exec bin/main.exe kfold car 10
For depth = 1, average training error was 0.4390 and validation error was 0.3333
For depth = 2, average training error was 0.3339 and validation error was 0.2674
For depth = 3, average training error was 0.2845 and validation error was 0.3935
For depth = 4, average training error was 0.1487 and validation error was 0.4288
For depth = 5, average training error was 0.0600 and validation error was 0.4699
For depth = 6, average training error was 0.0060 and validation error was 0.4832
For depth = 7, average training error was 0.0060 and validation error was 0.4832
For depth = 8, average training error was 0.0060 and validation error was 0.4832
For depth = 9, average training error was 0.0060 and validation error was 0.4832
For depth = 10, average training error was 0.0060 and validation error was 0.4832
The minimum average validation error was 0.2674 at a depth of 2
Training error for final tree of depth 2 is 0.2998
```