15.095 Machine Learning Under a Modern Optimization Lens - Project Report
Language modeling using Tree-Based Methods: Can tree-based methods perform autoregressive language modeling as well as neural networks?

Team Members: Seth Chatterton, Jason Jia

Abstract

We investigate the feasibility of tree-based methods, including XGBoost, Random Forest, and CART, as alternatives to neural networks for autoregressive language modeling. Neural networks are widely favored but computationally intensive, prompting an exploration of tree-based methods as a more efficient option. Using the Cleaned Alpaca Dataset, we found that tree-based methods, particularly Random Forest, exhibit 10-20% higher accuracy than Feedforward Neural Networks in next character prediction within similar training times. The findings suggest that tree-based methods can serve as viable and interpretable alternatives to neural networks in natural language processing tasks on small to medium-sized datasets.

Problem and Motivation

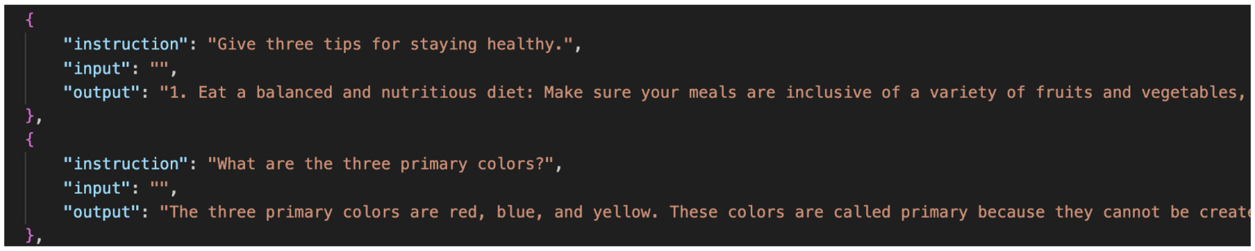
Models based on deep learning methods have gained immense popularity as the model of choice for autoregressive language modeling (predicting the next token/character/word in a given context). Autoregressive language models like ChatGPT can generate interesting and useful text when prompted.

However, these deep learning based methods are computationally expensive to train, and may be overkill for certain sizes of datasets. On the other hand, tree-based models such as XGBoost, RF, and CART can be easier to train and may be able to generate the same predictions. This is also in light of the proof in class that neural networks can be equivalent to trees under certain assumptions. Additionally, tree-based methods like CART can even be interpretable, so we can see why the model generated the text that it did.

We would thus like to experiment with tree-based methods and compare their performance with established neural network-based methods for autoregressive language modeling.

Data

We used the Cleaned Alpaca Dataset[1], which is a slightly modified version of the dataset used to train the Alpaca large language model[2]. Alpaca is a fine-tuned version of the LLaMA[3] large language model. The data consists of instructions for the model, inputs for those instructions, and outputs that the model should try to recreate. We took subsets of the dataset to form datasets of 3 sizes: small (20k rows), medium (400k rows) and large (1M rows).



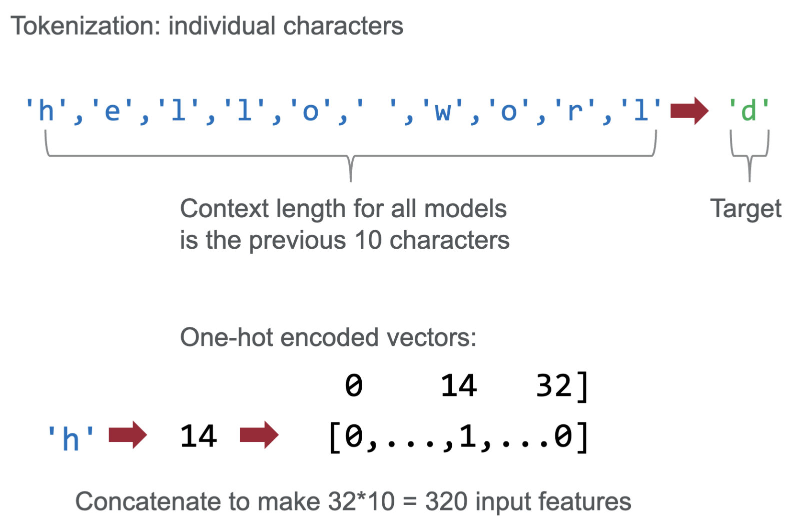
```
{
    "instruction": "Give three tips for staying healthy.",
    "input": "",
    "output": "1. Eat a balanced and nutritious diet: Make sure your meals are inclusive of a variety of fruits and vegetables,
},
{
    "instruction": "What are the three primary colors?",
    "input": "",
    "output": "The three primary colors are red, blue, and yellow. These colors are called primary because they cannot be creat
},
```

Methods

Preprocessing

Our preprocessing involved the following key steps:

- Tokens: We split sequences of concatenated instruction, input, and output by character, then take pairs of (10 consecutive characters, 11th character) as pairs of (X,Y).


Tokenization: individual characters
'h','e','l','l','o',' ','w','o','r','l'
'd'
Context length for all models is the previous 10 characters
Target
One-hot encoded vectors:
0 14 32]
'h'
14
[0,...,1,...0]
Concatenate to make 32*10 = 320 input features

- We add an ‘[EOS]’ (End Of Sequence) token to the end of the sequence, which is the concatenation of the instruction, inputs and outputs. We include this so that at generation time, if a model generates an [EOS] token, this lets us know that the model is done generating an output.
- One-hot encoding of characters: Each character is mapped to an index, and each character is converted to a vector using one-hot encoding. Each of these one-hot encoded vectors is concatenated to make a binary input vector of size (vocabulary size) * (context length)
- Train/test split: We use an 80/20 train-test split.

Tree-based methods

We tested several tree-based methods for autoregressive language modeling, including:

- CART
- XGBoost
- Random Forest

One of the other avenues we explored was to try to make token embeddings using these tree based methods. Token embeddings can be created by using techniques such as word2vec[4], or by learning them jointly in the first layer of deep learning methods. We wanted to explore methods of creating token embeddings without the use of deep learning methods. This would allow us to train end-to-end an autoregressive tree based model that can jointly learn semantically dense token representations, and provide non-deep learning methods with the potential for greater generalization to text.

We tried a number of different methods, but the core idea of the algorithm is as follows:

1) Initialize embeddings for every token randomly, with elements in the range [-1, 1] and normalized to unit length.
    - These take the place of the one-hot encoded vectors, but are still concatenated together for every token in the context window
2) Train an initial model to predict the next character from the embeddings of the characters in the context window using these random embeddings
3) Keeping the model fixed, adjust the token embeddings in a direction which minimizes the in-sample error
4) After every adjustment of the embeddings, use the new token embeddings to train a new model
5) Repeat steps 3 through 5 until the model and embedding error no longer improves by some threshold, or does not improve for some number of iterations

The key problem we face in the above algorithm however is how to update the embeddings. We cannot backpropagate through a tree like we can in a neural network. We tried three different methods of adjusting the embeddings at every iteration.

- Gradient Estimation
    - For every element in the current embedding vector, add a small positive or a small negative delta to that embedding vector element.
    - Evaluate the loss of the model when the change is included. If it improves when the delta is included, then we can accumulate that change in a vector of the estimated gradient with respect to the loss.
    - Adjust the embedding vectors by the accumulated estimated gradient direction to improve the loss after checking all vector deltas.
    - This is similar to a procedure known as gradient checking, which was more common for verifying the gradients of neural networks before frameworks like PyTorch and Tensorflow included automatic differentiation of model weights throughout an entire network
    - One problem with this approach is that tree based methods have discontinuous changes in them, so moving in a direction specified by a vector consisting of accumulated estimated gradients may no longer be a direction which decreases the loss compared to that of an individual element change.
- Noise Local Search
    - Produce a small random noise for every element of every vector
    - Add that noise to the best performing embeddings
    - Retrain the model using these new embeddings. If the loss improves, then we can keep these new embeddings and assume that they moved in a direction that is a more useful representation
    - This turned out to be the only method that could make any improvements at all from iteration to iteration
- Mean Embedding Collection

  - For every prediction made by the model, move the embedding vector in a direction that more closely aligns with the vector prediction made by the model, defined by the weighted average of all other token embeddings using the predicted probability of that token.
  - Move incorrect tokens in a direction away from the predicted weighted average embedding vector
  - The idea behind this update method was to try to consolidate the embeddings of similar vectors, and move dissimilar vectors away when they are incorrect
  - One problem with this is that we do not take into account changes in the input, only changes in the output, so there is no guarantee that we are moving in a direction that will actually improve the loss in retraining.

Each of the above techniques for updating the embeddings also has to contend with the fact that retraining a model with the methods we have chosen produces new trees. The splits on those trees are highly unstable from iteration to iteration, so we have no guarantee that after retraining the model we have similar splits that make use of the adjusted embeddings that we created.

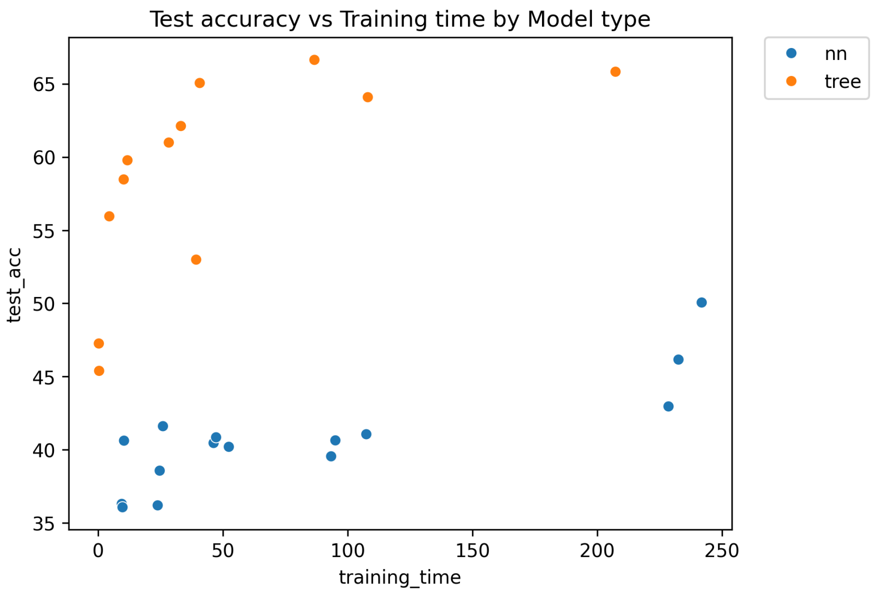
Neural Network methods (Benchmark)
We use feedforward neural networks (FFNs) as our benchmark. We choose a straightforward model architecture comprising an input layer, a hidden layer and an output layer. Here, embedding size is the number of rows of the weight matrix representing the hidden layer. The input to this model is the one-hot encoded context vector, which is the same input as the tree-based methods receive and makes results more comparable between methods.

While we have also tried Transformers with positional encoding, it gave a training accuracy of about 26.2% and test accuracy of about 27.5%, which are notably lower than expected. As the implementation may not be fully correct, we chose to use FFNs as our main benchmark.

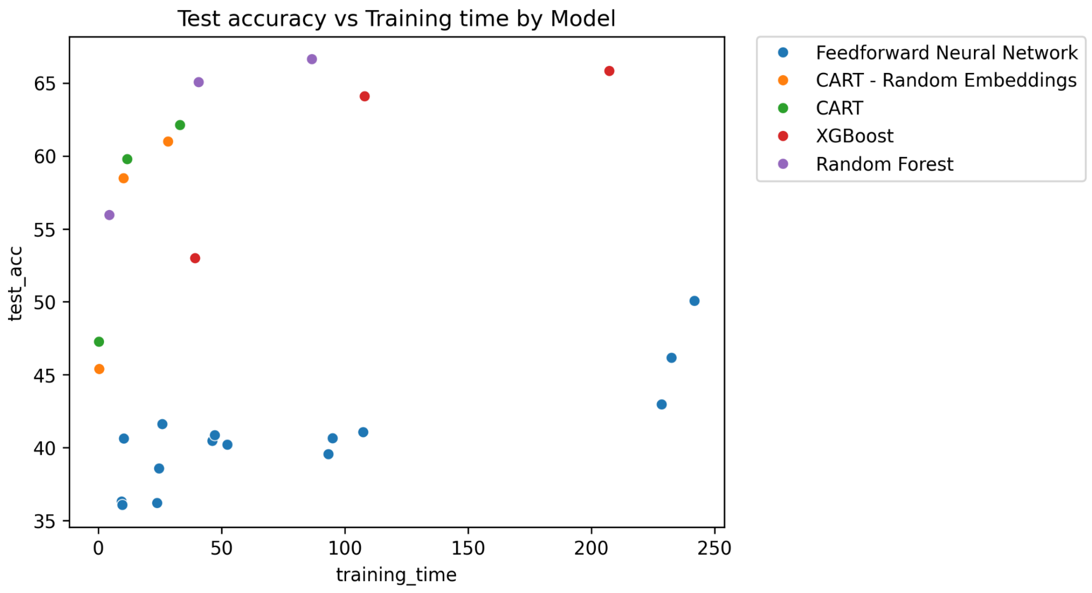
Evaluation

We evaluated our models in several different ways, including accuracy on a held out test set, training time, interpretability, and the quality of generated text. We also evaluated the efficiency of model training by looking at how long it takes certain methods to achieve a given accuracy.

Our key finding is that tree-based methods can sometimes outperform neural networks in next character prediction. As seen in the figure below, tree-based methods have 10-20% higher accuracy than neural networks on average, given similar training times and the small dataset sizes we used.



Breaking down performance of tree-based methods by model, we find that Random Forest performs the best, closely followed by XGBoost, CART and CART with random embeddings.



These are the summarized results of the set of parameters which gave the best test accuracy for each model:

Model	**Embed ding size**	**Dataset size**	**Epochs**	**Train Accuracy (%)**	**Test Accuracy (%)**	**Training Time (s)**	**Other parameters**
FFN	32	large	10	49.98	50.06	801.5	Hidden size = 128
CART with random embeddings	8	large	-	83.56	60.99	**28.4**	depth=56
CART	-	large	-	80.39	62.12	33.2	depth=129
XGBoost	-	large	-	90.22	65.83	207.4	max_depth=50 boost _rounds = 40
Random Forest	-	large	-	**90.36**	**66.64**	86.7	estimators=40

See full results for tree-based methods in Appendix 1 and full results for neural network methods in Appendix 2.

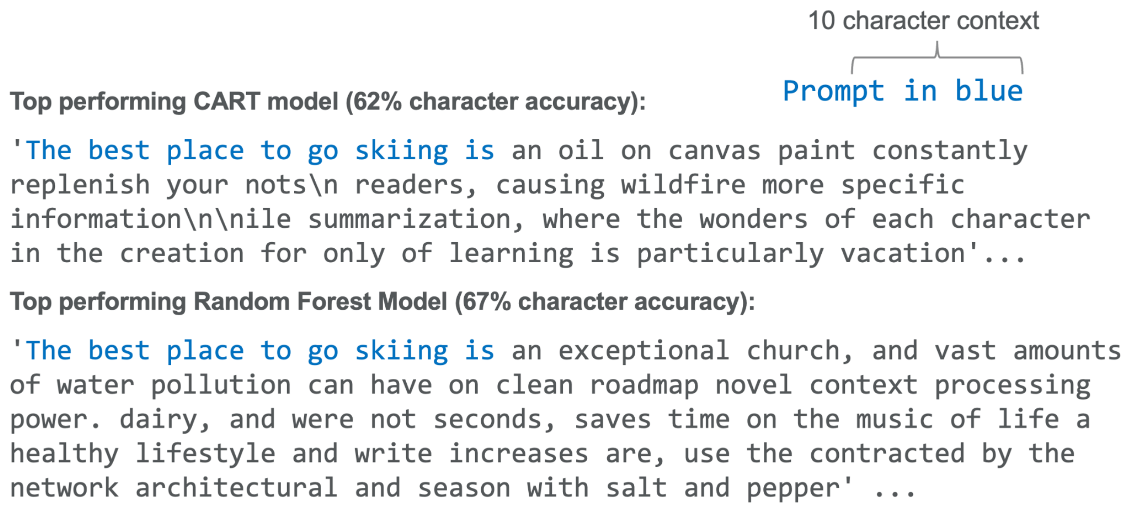
One hypothesis we have for why Random Forest performs best is because Random Forest creates fully developed trees. This means that there is only one training example per leaf node in each estimator, overfitting to the subset of data that each estimator is trained on. XGBoost on the other hand uses “weak learners”, which try to limit the growth of the tree based on a complexity parameter, leading to shallower trees. We can see that the best CART model is very large at a depth of 129, probably to capture as many “rules” as possible. Using this insight, it makes sense that a fully developed RF tree outperforms XGBoost, because each individual tree is much more powerful at capturing individual “rules” about the order of how tokens are placed.

One result we found surprising is that CART with random embeddings performs nearly as well as CART with one-hot encoded vectors. By random embedding we mean a random vector is generated to represent each token, and those vectors are used anywhere the respective token appears. What might be happening here is that CART can pick up on the useful association in the vectors that appear by chance, since there is a high chance that at least one of the features in the random embedding is related to or correlated with a vector element in another. Also, since most of the vector values of the random embedding are probably unique, a CART model could do something like use two splits in a row to bound the value of a specific letter from above and below, using two splits to do the same thing as one split in the one-hot encoding scheme. Additionally, since these random embeddings are about 4 times shorter than one-hot encoded features, the model has to consider fewer features when training the model, which speeds up training. Obviously however, random embeddings mean nothing and are not interpretable at all. These were just meant to be a baseline to compare against the different embedding adjustment methods.

None of the methods we used to try to update tree-based model word embeddings worked better than simply using random embeddings. Part of this probably has to do with the fact that the set of all possible embedding vectors is so huge, and the set of embedding vectors that are useful and semantically meaningful is much smaller. If our methods for finding that small set of semantically meaningful vectors is not good, then it is unlikely that we would be able to find a set of semantically meaningful vectors. See Appendix 1 for results of our embedding update methods. We only tested on the smallest dataset, because if we could not find an embedding update algorithm that worked on that small dataset, then it would most likely not work on a larger dataset.

The interpretability of individual trees also lends intuition to how predictions are made. For example, in the best performing CART model, the root node asks if the previous character is a comma, and if it is then the model predicts that the next character will be a space. Further, if the previous character is not a comma but instead a space, and the character before that is not a space, the model predicts 't'. We can see this clearly if we look at a visualization of the tree (Appendix 3).

Below are examples of text generated by the top performing CART and Random Forest model, given a 10-character prompt.


10 character context
Prompt in blue

Top performing CART model (62% character accuracy):

```
'The best place to go skiing is an oil on canvas paint constantly
replenish your nots\n readers, causing wildfire more specific
information\n\nile summarization, where the wonders of each character
in the creation for only of learning is particularly vacation'...
```

Top performing Random Forest Model (67% character accuracy):

```
'The best place to go skiing is an exceptional church, and vast amounts
of water pollution can have on clean roadmap novel context processing
power. dairy, and were not seconds, saves time on the music of life a
healthy lifestyle and write increases are, use the contracted by the
network architectural and season with salt and pepper' ...
```

These text completions form complete words, and make a small amount of grammatical sense from word to word. The sentences as a whole do not make grammatical sense however, and the topics are not consistent over the course of a sentence. Additional context length would probably fix this to some degree.

Impact
Our findings suggest that tree-based methods may be viewed as a viable alternative to neural network methods for next character prediction in small to medium sized datasets (<1M training examples).

CART models can be trained extremely quickly and are interpretable, allowing us to inspect the reasons why the model predicted what it did. This suggests that the choice between neural networks and tree-based methods is context-dependent, and could motivate the formulation of alternative architectures of more interpretable mid-sized language models. For scenarios where datasets are small and computing power is limited, tree-based methods could be a better choice than neural networks for natural language processing tasks.

Our experiments indicate that using more data produces better results, and there is no reason why the process couldn't scale to larger datasets. However, increasing the context lengths of these models may require significant amounts of data.

Future Work

We can improve the results of our models in several ways.

- Using larger datasets would allow models to see more examples and produce better results, and perhaps follow some of the scaling laws we see with large language models today where feeding in an enormous amount of data leads to excellent performance. We could increase the context lengths of our models if we had more data too, which would give more contest to what token should be predicted next.
- Using a different tokenization scheme than individual characters could lead to better results as well. For example, we could split training examples by bigrams, trigrams, by spaces, or using byte pair encoding. Using a larger vocabulary would lead to very large input vectors however, so it would also require a lot of data so that tokens could be seen multiple times.
- We could try to create a mixed-integer optimization approach combining OCTs with jointly finding optimal embeddings, such that the misclassification loss with a given embedding representation is minimized. This would be very computationally expensive to train however, given the 100+ depth of CART trees we see.
- We could try better search methods for finding tree-based model embeddings. For example, if we use the above OCT approach, maybe we could create a cutting planes approach to solve for the vectors to jointly solve for the embeddings, combined with a local search for the OCT structure. An approach similar to our current methods with the current models might be to use a genetic algorithm to try to find better word embeddings.
- We could try to find binary embedding vectors for our tree-based methods instead of continuous vectors. This could potentially lead to more interpretable embeddings, because it indicates categories. For example, maybe one element of the embedding is “vowel or not vowel”, or “letters that come after c”, or something else that might be more meaningful than a continuous vector.

Team Contributions

- Seth: Implementation of tree-based methods and tree based embedding search
- Jason: Implementation of neural network methods
- Together: project abstract, presentation and report

References

[1] Ruebsamen, G. 2023. Cleaned Alpaca Dataset. https://github.com/gururise/AlpacaDataCleaned/tree/main
[2] Taori, R et al. 2023. Stanford Alpaca: An Instruction-following LLaMA model. Github Repository https://github.com/tatsu-lab/stanford_alpaca
[3] Touvron, H et al. 2023. LLaMA: Open and Efficient Foundation Language Model. arXiv preprint arXiv:2302.13971
[4] Mikolov, T et al. 2013. Efficient Estimation of Word Representations in Vector Space. Advances in neural information processing systems, 2013

Appendix

Appendix 1: Performance of tree-based methods

Model	**Embedding size**	**Dataset size**	**Epochs**	**Train Accuracy (%)**	**Test Accuracy (%)**	**Training Time (s)**	**Other params**
CART - random embeddings	8	small	-	85.16	45.39	0.43	depth=36
CART - random embeddings	8	medium	-	84.66	58.47	10.28	depth=45
CART - random embeddings	8	large	-	83.56	60.99	28.35	depth=56
CART - mean embedding collection	8	small	20	85.29	45.81	58.88	depth=38
CART - gradient estimation	8	small	5	85.28	44.23	207.95	depth=31
CART -	8	small	100	85.66	44.31	49.86	depth=32

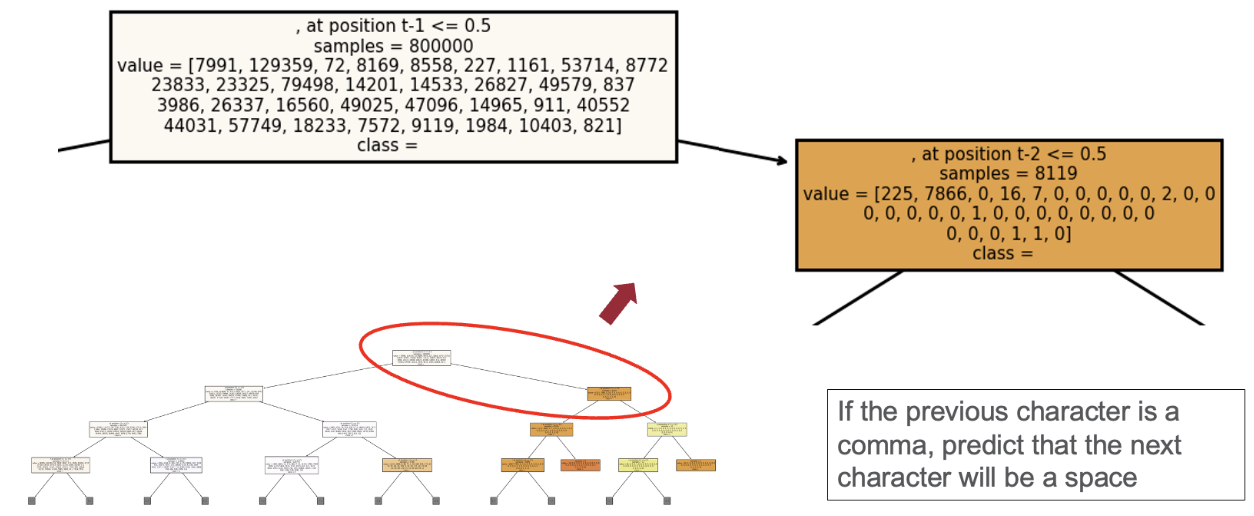
noise search							
CART	-	small	-	79.85	47.26	0.31	depth=71
CART	-	medium	-	80.83	59.78	11.78	depth=118
CART	-	large	-	80.39	62.12	33.19	depth=129
XGBoost	-	small	-	97.94	52.99	39.31	max_depth=50 boost_rounds = 200
XGBoost	-	medium	-	92.74	64.09	108.10	max_depth=50 boost_rounds = 50
XGBoost	-	large	-	90.22	65.83	207.365	max_depth=50 boost_rounds = 40
Random Forest	-	small	-	97.94	55.95	4.51	estimators =200
Random Forest	-	medium	-	92.73	65.06	40.75	estimators =50
Random Forest	-	large	-	90.36	66.64	86.70	estimators =40

Appendix 2: Performance of neural network methods (Benchmark)

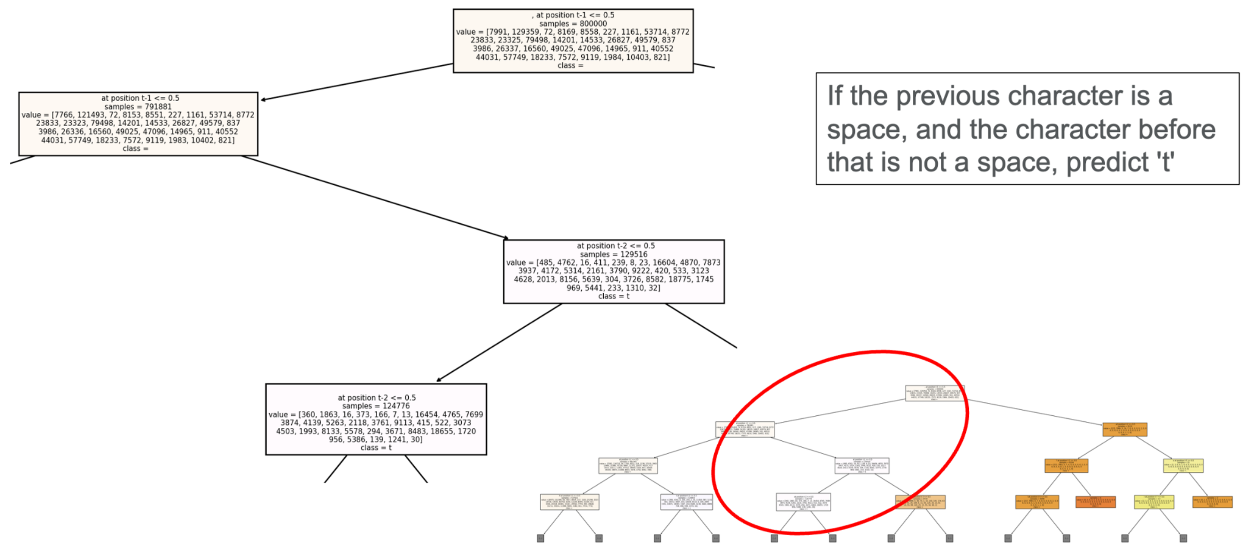
Model	**Embedding size**	**Dataset size**	**Epochs**	**Train Accuracy (%)**	**Test Accuracy (%)**	**Training Time (s)**
FFN	8	small	100	50.00	40.64	95.1
FFN	8	small	50	46.44	40.46	46.3
FFN	8	small	25	43.34	36.20	23.9
FFN	8	small	10	38.40	36.30	9.5

FFN	16	small	100	61.54	39.55	93.4
FFN	16	small	50	57.61	40.85	47.3
FFN	16	small	25	51.35	38.57	24.7
FFN	16	small	10	44.03	36.07	9.8
FFN	32	small	100	80.27	41.06	107.5
FFN	32	small	50	71.84	40.20	52.4
FFN	32	small	25	63.54	41.61	26.0
FFN	32	small	10	51.45	40.62	10.4
FFN	8	medium	10	42.97	42.96	228.6
FFN	16	medium	10	46.88	46.16	232.6
FFN	32	medium	10	50.48	50.06	241.9
FFN	8	large	10	43.14	43.56	884.4
FFN	16	large	10	46.91	46.84	839.7
FFN	32	large	10	49.98	50.06	801.5

Appendix 3: Supporting figures demonstrating the interpretability of tree-based methods



If the previous character is a comma, predict that the next character will be a space


, at position t-1 <= 0.5
samples = 800000
value = [7991, 129359, 72, 8169, 8558, 227, 1161, 53714, 8772 23833, 23325, 79498, 14201, 14533, 26827, 49579, 837 3986, 26337, 16560, 49025, 47096, 14965, 911, 40552 44031, 57749, 18233, 7572, 9119, 1984, 10403, 821]
class =
at position t-1 <= 0.5
samples = 791881
value = [7766, 121493, 72, 8153, 8551, 227, 1161, 53714, 8772 23833, 23323, 79498, 14201, 14533, 26827, 49579, 837 3986, 26336, 16560, 49025, 47096, 14965, 911, 40552 44031, 57749, 18233, 7572, 9119, 1983, 10402, 821]
class =
at position t-2 <= 0.5
samples = 129516
value = [485, 4762, 16, 411, 239, 8, 23, 16604, 4870, 7873 3937, 4172, 5314, 2161, 3790, 9222, 420, 533, 3123 4628, 2013, 8156, 5639, 304, 3726, 8582, 18775, 1745 969, 5441, 233, 1310, 32]
class = t
at position t-2 <= 0.5
samples = 124776
value = [360, 1863, 16, 373, 166, 7, 13, 16454, 4765, 7699 3874, 4139, 5263, 2118, 3761, 9113, 415, 522, 3073 4503, 1993, 8133, 5578, 294, 3671, 8483, 18655, 1720 956, 5386, 139, 1241, 30]
class = t
If the previous character is a space, and the character before that is not a space, predict 't'

Appendix 4: Additional text completions

Top performing CART model:

```
'Where is the best place to go skiing?\n\nsurface area of the
triangle abc is simple language processes that release oxygen to
generate longterm strategies for liquids adequality.'

'abcdefghijk fears and is chairs has excerpts to support their
thoughts and extract if wealth, and prevent certain traits and
loyalty.\n\n. promote green energy consumption of new destination or
resoup platform.'

'The best place to go skiing is mostly driven by factoring outside
and easier months.'
```