Fast and Accurate Decision Trees for Natural Language Processing Tasks

Tiberiu Boros[*,**], Stefan Daniel Dumitrescu[**] and Sonia Pipa[**]

[*]Adobe Experience Manager, Machine Learning, Adobe Systems
[**]Research Institute for Artificial Intelligence, Romanian Academy
{boros}@adobe.com, {sdumitrescu,sonia}@racai.ro

Abstract

Decision trees have long been used in many machine-learning tasks; they have a clear structure that provides insight into the training data and are simple to conceptually understand and implement. We present an optimized tree-computation algorithm based on the original ID3 algorithm. We introduce a tree-pruning method that uses the development set to delete nodes from overfitted models, as well as a result-caching method for speed-up. Our algorithm is 1 to 3 orders of magnitude faster than a naive implementation and yields accurate results on our test datasets.

1 Introduction

Decision trees (DTs) are a well-established classification/prediction methodology in machine learning, in which the model is (as the name suggests) a tree where each node is a decision and each leaf represents an output class (label/distribution). A node can have any number of children, but commonly, most algorithms implement only binary trees with binary questions. Decision-tree classifiers are a very popular choice, mainly because they are easy to train and, by analyzing the tree structure one can easily validate certain assumptions or gain a better understanding of the corpora/task as opposed to, for example, neural networks in which the model is a matrix of numbers offering no insight.

DTs have been widely used in natural and spoken language-processing tasks such as tagging (Schmid, 2013), named entity recognition (NER) (Szarvas et al., 2006), letter-to-sound (LTS) conversion (Pagel et al., 1998), text categorization (Lewis and Ringuette, 1994), parameter estimation for statistical parametric speech synthesis (Zen et al., 2007).

One of the drawbacks of decision trees is their relatively modest performance in classification tasks. Current state-of-the-art approaches in natural language processing and other research fields employ more powerful methodologies, such as Support Vector Machines (SVMs), Conditional Random Fields (CRFs), and complex neural network architectures.

In what follows, we propose an optimized decision tree computation algorithm which follows the guidelines of the Iterative Dichotomiser 3 (ID3) algorithm (Quinlan, 1986), but computes entropy and information gain using a single pass over the training data. We also address the issue of overfitting the training set by introducing a tree-pruning algorithm that tunes an already existing tree using a development set. For significant speed increase we implement a result-caching method. We show that tree-pruning achieves up-to-par accuracy on our datasets by comparing results obtained using (a) the unpruned version of the tree (b) the pruned version of the tree and (c) various state-of-the-art methods in identical training and testing conditions.

We argue that the training speed-boost obtained by using our computational enhancements, as well as the accuracy-boost obtained by tree-pruning make DTs a desirable choice for feature-engineering and for real-life applications, whenever speed of implementation/training with slightly lower results are acceptable.

2 Related Work

The simple and robust principle behind decision trees made them an ideal choice for both academic and industrial (applied) research. There are several papers that address construction and optimiza-

Proceedings of Recent Advances in Natural Language Processing, pages 103–110,
Varna, Bulgaria, Sep 4–6 2017.
https://doi.org/10.26615/978-954-452-049-6_016

tion principles applied, from which we selected those relevant to our approach. Su and Zhang (2006) use independent information gain (IIG) to speed-up the process of tree construction and reduce the complexity of the calculus, Dai and Ji (2014) introduce an algorithm designed for distributed computation of trees, based on mapreduce. Quinlan (1987) introduces one of the first tree-pruning strategies, while Mehta et al. (1995) and Rastogi and Shim (1998) use the a more principled approach, based on the Minimum Description Length (MDL). Though MDL strategies work very well and are currently the main ingredient of statistical parametric speech synthesis (Zen et al., 2009), for simple NLP tasks they add "overhead" and increase computational complexity at training time. Other approaches to decision tree optimization are aimed straight at tree-construction strategies and refer to randomization, bagging and boosting (Dietterich, 2000).

Our proposed optimization methods (a) are based on the original ID3 algorithm with the **"missing-attribute" extension** and (b) focus on discrete features (not continuous values).

3 Enhanced ID3 Computation

The Iterative Dichotomizer 3 (ID3) is an algorithm created by Ross Quinlan to generate a decision tree from a dataset. The computation starts with the entire dataset S at the root node. For each iteration, the feature having the largest information gain $IG(S)$ is located among the unused features. The set S is then split and the two children of the current node are created: one for the examples that contained the selected feature and second for those that did not. The algorithm continues recursively for each newly created node, until one of the stop conditions is encountered:

- A subset contains only examples belonging to the same class. In this case, the node is turned into leaf labeled with the name of the class.
- There is no unused feature to select, but the examples still do not belong to the same class. The node is also turned into a leaf, this time labeled with the most common class in the subset.
- We do not obtain any information gain when splitting by any unused feature. In this situation, the created leaf is also labeled with the most common class in current subset.

The output of this algorithm will be a decision tree, with each non-terminal node representing the selected feature on which the data was split, and the terminal nodes (leafs) representing the class label of the final subset of the branch.

3.1 Speed-Optimized Computation of Entropy and Information Gain

Before describing our proposed optimizations, we have to clarify our view of the algorithm's data representation style. The standard way to view a dataset is that of a examples (instances) having a number of attributes and an output class. Each attribute (e.g. Temperature) can have several values (multinomial attribute, e.g. Hot, Average, Cold), an instance being a collection of a particular value for each of the available attributes. The ID3 implementation we propose handles missing features and treats attributes as a bag-of-words, meaning that an instance might have a feature like Temperature_Hot, or Temperature_Cold. This allows greater flexibility, especially for NLP tasks, and is essentially similar to the classic data representation style.

Returning to the ID3 algorithm training procedure, the decision to select a feature over another is given by the information gain. Given a dataset S with M features and N classes, information gain $IG_i(S)$ is the measure of the difference in entropy in S after it was split on feature i, meaning we measure how much the uncertainty in S has decreased by choosing to split by i. This implies running over S for each of the M features to compute $IG_i(S)$ and selecting the maximum value. Choosing to split by feature i will yield, two complementary subsets of S, one having all examples (instances) that have feature i, named the Yes_i subset, the other having the remaining examples that do not have feature i (the No_i subset). Iteratively, for each subset we have to find the next feature to split on, again running over all the examples in the subsets. We propose a way around these multiple passes over the dataset that brings a significant speed increase.

Starting from the definition, information gain is the difference in entropy of the initial set S and after the set was split on feature i (in all T subsets).

$$IG_i(S) = H(S) - \sum_{t \in T} (P(t) \cdot H(t)) \quad (1)$$

where $H(S)$ is the entropy of the initial set S, $H(t)$ the entropy of subset t, and $P(t)$ is the fraction of elements in t over the entire $|S|$. We note $|S|$ as the number of instances in S.

$$H(S) = -\sum_{x=1}^{N} P_x \cdot \log_2 P_x \tag{2}$$

where P_x is the number of instances of class x divided by $|S|$.

The optimization we propose in this paper is based on creating a contingency matrix. By having a matrix that stores partial results and additional information, we are able to reduce the number of operations that are repeatedly executed in the classical implementation of the ID3 algorithm.

The proposed matrix is presented in Table 1, where $O_{1,N}$ notation is used for the N output classes, and $F_{1,M}$ marks the M unique features. Cells at index $[i, j]$ (from position $[1, 1]$ to $[M, N]$) keep the number of instances in the dataset that contain feature F_i of class O_j. To compute $H(S)$ and $IG_i(S)$ for every feature i, an iteration through the dataset is required. To skip this step we add an extra row and column to the matrix: the $M+1^{th}$ row stores the total number of instances of class O_j and the $N+1^{th}$ column stores the number of instances that have label F_i.

Starting from the Entropy and Information Gain formulas, we developed the following set of equations to compute these measures using the partial results stored in matrix.

$\boldsymbol{a}$	$\boldsymbol{O_1}$	$\boldsymbol{O_2}$	$\boldsymbol{O_3}$	$\boldsymbol{O_4}$	...	$\boldsymbol{O_n}$	
$\boldsymbol{F_1}$	2	0	5	0	...	0	X_1
$\boldsymbol{F_2}$	0	0	1	0	...	0	X_2
$\boldsymbol{F_3}$	1	3	1	7	...	0	X_3
$\boldsymbol{F_4}$	20	0	1	0	...	1	X_4
...	...	...	...	...	...	...	...
$\boldsymbol{F_m}$	1	0	4	1	...	10	X_m
	Y_1	Y_2	Y_3	Y_4	...	Y_n	

Table 1: Computation matrix

Now, the entropy $H(S)$ can be written as:

$$H(S) = -\sum_{i=1}^{N} \frac{a[M+1, i]}{|S|} \log_2 \frac{a[M+1, i]}{|S|} \tag{3}$$

where a is our proposed contingency matrix and $a[M + 1, i]$ refers to the last row that counts the number of instances having output class O_i.

To calculate $IG_i(S)$, considering i as the attribute to split on, we have to subtract from $H(S)$ the entropy of the subsets multiplied by the probability of splitting by feature i. The subsets, in our implementation, are always two: the Yes subset containing all instances that have feature i and the No subset, with instances not containing feature i:

$$IG_i(S) = H(S) - [P_{feat_i} \cdot H(Yes) + \overline{P_{feat_i}} \cdot H(No)] \tag{4}$$

Where P_{feat_i} is:

$$P_{feat_i} = \frac{a[i, N+1]}{|S|} \tag{5}$$

with $a[i][N + 1]$ being the number of instances that contain feature i, meaning the number of instances in the Yes subset. The $\overline{P_{feat_i}}$ is the complement of P_{feat_i}, meaning the number of instances in the No subset divided by the total number of instances in $|S|$.

Moving on, the entropies of $H(Yes)$ and $H(No)$ are:

$$H(Yes) = -\sum_{x \in Yes} P_{Yes_x} \cdot \log_2 P_{Yes_x} \tag{6}$$

$$H(No) = -\sum_{x \in No} P_{No_x} \cdot \log_2 P_{No_x} \tag{7}$$

so $IG_i(S)$ becomes:

$$IG_i(S) = H(S) - \sum_{x \in X} (-P_{feat_i} \cdot P_{Yes_x} \log 2P_{Yes_x} - (\overline{P_{feat_i}} \cdot P_{No_x} \log 2P_{No_x}) \tag{8}$$

$$P_{Yes_x} = P_{Yes_{i,j}} = \frac{a[i, j]}{a[i, N+1]} \tag{9}$$

$$P_{No_x} = P_{No_{i,j}} = \frac{a[M+1, j] - a[i, j]}{|S| - a[i, N+1]} \tag{10}$$

where $a[i, j]$ means the number of instances having feature i and output class O_j; $a[i, N+1]$ is the number of instances that have feature i; $a[M+1, j]-a[i, j]$ is the number of instances of class O_j minus those in the Yes subset; $|S| - a[i, N+1]$ is the number of instances that do not have feature i (the number of instances in the No subset).

Finally,parametrized by i and j, $IG_i(S)$ becomes:

$$IG_i(S) = H(S) - \sum_{j=1}^{N}(-P_{feat_i} \cdot P_{Yes_{i,j}} \log 2P_{Yes_{i,j}} - (\overline{P_{feat_i}} \cdot P_{No_{i,j}} \log 2P_{No_{i,j}}) \quad (11)$$

It can be seen that the calculation is now performed in a single operation, using the extra row and column of the contingency matrix.

The purpose of creating this contingency matrix is to avoid iterating at each step through all of the training examples, an amount that can vary from hundreds to hundreds of millions of instances, each with any number of features. The size of the contingency matrix itself is small as it does not very by the number of instances but by the number of features and classes, usually orders of magnitude smaller than the training corpus itself.

3.2 Decision-Tree Pruning

As with machine learning algorithms, data sparseness combined with noise will likely yield overfitted models, which means that the constructed tree will model a features/label combination that will never exists in real data. There are, of course, several techniques that can be used to prevent this from happening such as increasing the train-set size, decreasing the number of features and performing frequency cut-off over features and labels, but these are general guidelines applicable to any classifier. In what follows we propose a simple method to prevent overfitting the training data by introducing the possibility to use a development set in the training process.

Though our proposed methodology is simple and intuitive and is inspired by the idea to reduce the tree-size be replacing a node with one of its subtrees presented in (Quinlan, 1993). However, in our approach we rely on the development set to perform this step. The idea is simple: use the standard ID3 to build a decision tree and then iteratively run a tree-pruning procedure until there are no more improvements on the development set. The algorithm can be outlined as:

1. Construct an initial tree using the available training data;
2. Take each node and measure the accuracy on the development set as if the node were a terminal leaf with the most probable output label[1];
3. If there are no improvements on the development set, stop the algorithm and return the current tree structure; Otherwise update the tree structure by removing the node with the highest accuracy gain and return to step 2.

Though trivial, our experiments showed that this is an effective approach to prevent the tree from over-fitting the training data and it provides significantly better accuracy rates on the test set, actually bringing the results very close to those obtained using state-of-the-art classifiers (see subsection 4.3).

The naive way to implement this algorithm is to compute the best performing tree-structure at every iteration (t) by pruning part of the tree and measuring the accuracy on the development data. This means that, at every step i_t, for a tree structure with k_t nodes (k_t is used to denote the remaining number of nodes at step t) the algorithm has to go through the entire development set, compute the predicted label using the new tree structure and measure the new accuracy. While this approach works for small datasets and trees, larger number of nodes and development examples render this algorithm unusable. For instance, in our initial experiments we let this algorithm run for 24 hours on a part-of-speech tagging corpus for Romanian (see section 4.3 for details) and it only pruned 26 nodes, while the actual convergence number was 245 nodes.

The prerequisites for computing the accuracy gain by pruning a node are the following: (a) the algorithm has to know which is the most probable label that the unmodified tree structure would predict if the runtime prediction algorithm would pass through the current node; (b) the algorithm has to know what would be the overall accuracy of the unmodified tree structure; (c) the algorithm has to compute the new accuracy figures if the node would be transformed into a leaf assigned with the most probable label. Because we already know the ground-truth for all the examples (training and development) and we can easily compute the prediction values for both the training and the development set before each iteration, we can speed-up

[1]When we compute the most probable label we use the data in the training set, because we don't want to completely bias the tree towards the development data.

accuracy computation at the expense of memory by caching the results.

As such, for every node (n) inside the tree, we compute 3 vectors (g_n, s_n and f_n) which have a length equal to the number of unique labels (l).

- g_n is used to cache the counts of each unique label generated by **training examples** that pass through this node at runtime;
- s_n is used to cache the counts of **successful predictions** for each unique label, generated by **development set examples** that pass through this node at runtime;
- s_n is used to cache the counts of **unsuccessful predictions** for each unique label, generated by **development set examples** that pass through this node at runtime;

By pruning a node we actually wind up generating correct prediction for all example instances of the most probable label an incorrectly classify all other examples. As such, for each node (n) we can compute the most probable label (l) as

$$l = argmax(g_n) \tag{12}$$

and the accuracy gain (A_n) as:

$$A_n = \frac{(s_{n,l} + f_{n,l}) - \sum s_n}{E} \tag{13}$$

where E is the total number of examples in the development set.

As such, we compute g_n in the initialization step of our algorithm and we update s_n and f_n by performing a single pass on the development set before each tree-pruning iteration. This means that for a tree with 15K nodes (which is not rare), we only require a single pass over the development set, instead of 15K passes, which makes this approach practically 15K faster than the naive implementation.

4 Experimental Validation

To provide a thorough evaluation of our proposed methodologies, we are (a) providing a clear view over the computation-time enhancements by comparing a naive ID3 implementation with our own version of the algorithm (subsection 4.2) and (b) demonstrate how we mitigate model over-fitting issues using tree-pruning by comparing our results to state-of-the-art methods in identical testing conditions (subsection 4.3)

4.1 Corpora Description

For our algorithm validation process we have selected a number of training datasets which we can use in our evaluation. The choice of these datasets is driven by reproducibility, in the sense that we have access to both training and datasets on which state-of-the-art methods were tuned and tested. Before we proceed with the actual evaluation of our system we will shortly review these datasets to familiarize the reader with the tasks themselves. Arguably, there are many other resources that can be used in the validation process, but we feel that the selected datasets provide a fair coverage on most of the typical NLP tasks. The chosen datasets refer to: (a) letter-to-sound (LTS); (b) syllabification; (c) part-of-speech tagging; (d) text classification and (e) tokenization

- **The LTS lexicon** used in our validation process contains two sub-datasets: the CMU-Dictionary (Weide, 2005) and the Romanian Speech Synthesis Database (Stan et al., 2011);
- **The syllabification lexicon** is the automatically Onset-Nucleus-Coda (ONC) (Bartlett et al., 2008) labeled corpora also has two sub-corpora, for Romanian and English;
- **The part-of-speech lexicon** is based on the coarse part-of-speech datasets (Petrov et al., 2011) provided in the Universal Dependencies Database (Nivre et al., 2016);
- **Morphological attributes lexicon** is compiled from the Universal Dependencies Database based on the specifications of Zeman (2008);
- **The text classification datasets** contain the WebKB dataset (Craven et al., 1998), and 20 newsgroups (20ng);
- **The tokenization corpus** is automatically extracted from the multilingual training-files provided with Universal Dependencies.

Before we proceed with the actual validation we must clarify the testing conditions regarding morphological attribute resolution. As mentioned, the training data was compiled from the Universal Dependencies treebank. The morphological labels strictly refer to attributes such as gender, number, case etc. and not to the part-of-speech itself

Table 2: Training time on the selected corpora, reported using the naive and optimized implementations of the algorithm

Dataset	# examples	# features	# labels	Naive	Optimized
LTS EN (CMUDICT)	666771	295	159	212.7s	9.8s
LTS RO (RSS)	53491	206	48	14.6s	0.18s
SYL EN	1373012	240	22	720.2s	12.9s
SYL RO	4760735	291	25	885.4s	41.9s
TAG EN (UD)	204605	2133	17	5811.3s	19.8s
TAG RO (UD)	185113	2403	17	4956.1s	19.1s

(which is actually used as an input feature). The original files are in CONLL format[2] and the attributes are stored in a special column in the form of key/value pairs. In order to create our training data we used the following procedure:

1. We went through the entire training data and we grouped attributes by their key, thus obtaining attribute groups;
2. We created separate training, development and test files for each individual attribute group;
3. For every word/token, we used the following features, extracted from every word inside a 5-token window (centered on the current item): coarse part-of-speech, first 4 characters of the word (4 individual features), last 4 characters of the word (also 4 individual features), word style (lowercased, capitalized or uppercased)

4.2 Training Speed Optimization

To check our proposed optimization technique we will measure the training speed gain for each of the previously mentioned datasets. Table 2 shows the training time measured for both the naive and optimized implementations of the algorithm. To ensure comparable results, we verified that the naive and optimized tree structures are identical.

The table shows that our proposed implementation of the algorithm speeds up training time by a 1-3 orders of magnitude compared to the naive tree construction. This translates in the ability to perform a significantly larger number of tests during feature selection and tuning phases. To our knowledge, there is no other implementation of any tool or classifier that can build a **comparable model** in such a short period of time.

[2]http://universaldependencies.org/format.html - accessed 2017-05-03

4.3 Accuracy-Boost Using Tree-Pruning

To prove that our tree pruning strategy is effective on real life data, in what follows we are going to compare the accuracy of the un-pruned trees to that of pruned trees. For reference, we are also going to report the highest accuracy of any other state-of-the-art method, provided that the testing conditions are identical. Table 3 summarizes the results and, as can be seen, the results obtained using the pruned tree are significantly better that the un-pruned version and are very close to state-of-the-art results reported by other authors. The notable difference on the tagging set between the ID3 classifier and the CRF is mostly influenced by the fact that the CRF implementation generates label bi-grams and uses a Viterbi decoder to select an optimal state sequence. A similar approach can also be obtained using the ID3 tree, but the tree-pruning part would require a different methodology for computation and optimization and it was currently out-of-scope. However, when it comes to morphology extracted from local word features, the ID3 implementation is closer to the CRF results, mainly because morphologic attribute resolution is not influenced by bi-grams (we are not saying that there are no context dependencies between the attributes of words - we imply that these dependencies are not easily handled by the simple use of label bi-grams).

*The results reported in Jiampojamarn et al. (2008) are probably obtained using a different split or corpora preparation (filtering) procedure. In practice when we tried to reproduce the results with an identical feature template, using the same classifier, we only achieved a 65.19% accuracy on our test-set.

For letter-to-sound and syllabification we report word-accuracy (not label accuracy which is significantly higher) and for document classification we report the number of correctly classified examples.

Table 3: Accuracy figures for the selected datasets, reported for the pruned, un-pruned and reference state-of-the-art methods and algorithms

Dataset	Un-pruned			Pruned			State-of-the-art			
	Train	Dev	Test	Train	Dev	Test	Name	Train	Dev	Test
LTS EN	72.67	56.93	56.47	67.58	64.11	63.18	MIRA	N/A	N/A	**71.99***
LTS RO	98.36	93.24	93.44	97.07	95.31	95.05	MIRA	N/A	N/A	**96.29**
Syl EN	94.77	82.63	83.21	89.31	87.89	85.58	CRF	N/A	N/A	**86.22**
Syl RO	99.27	98.89	98.97	99.21	99.07	**99.16**	CRF	N/A	N/A	99.10
Tag EN	97.28	94.03	93.98	96.05	95.13	95.04	CRF	N/A	N/A	**97.63**
Tag RO	98.78	93.12	93.02	96.42	95.68	95.48	CRF	N/A	N/A	**96.72**
Morph En	–.–	–.–	97.22	–.–	–.–	**98.46**	Custom	N/A	N/A	93.82
Morph RO	–.–	–.–	94.81	–.–	–.–	**95.76**	Custom	N/A	N/A	95.56
Tok EN	–.–	–.–	–.–	–.–	–.–	**99.05**	GRU	N/A	N/A	98.69
Tok RO	–.–	–.–	–.–	–.–	–.–	**99.80**	GRU	N/A	N/A	99.55
WebKB	99.76	75.35	74.11	86.10	81.78	**78.15**	LSI	N/A	N/A	75.56
20NG	99.96	51.72	52.16	81.85	72.32	**62.91**	LSI	N/A	N/A	62.78

Because the current version of the UD corpus was newly released, there are no official papers that evaluate any state-of-the art methods on the latest release of the corpora. However, in this case we compare the pruned tree results with the results reported on the official UD page (tokenization - using Gated Recurrent Units (GRUs) (Straka et al., 2016)). Also, we were unable to evaluate tokenization results in a "static manner", because token boundaries for a given character index depend on the previously generated breaks. Thus, we only provide the final evaluation results over the test set. Also, we must mention that in order to keep the testing conditions similar to previously reported results we did not alter the tests sets in any way and we used 10% of the original training data for building our development sets.

The state-of-the-art results obtained for WebKB and 20NG are based Latent Semantic Indexing (LSI) (Zelikovitz and Hirsh, 2001).

5 Conclusions and Future Work

We introduced our optimized version of the ID3 tree-computation algorithm, as well as a method to fine-tune an existing tree using a development set to selectively prune it. The later mentioned algorithm is also optimized for speed using a result-caching approach.

We showed that by fine-tuning a tree structure one can achieve results comparable to state-of-the art classifiers. We argue that decision trees can be easily used in the feature selection process of any machine learning algorithm. Combined with the enhanced computation time and tree-pruning methodology make this a useful contribution, especially in the field of natural language processing where working with discrete features in a bag-of-words fashion is common.

Additionally, the results reported in section 4.3 are obtained by simply extracting features inside the context-window and not by introducing any predefined feature-sets. This is not the case for the state-of-the art results to which we compared our system to, where the features are actually the result of complex careful crafting of feature templates. The robust principles behind decision trees do not require an extensive feature engineering process. Instead, the constructed tree can offer a good overview of the task and dataset itself and can actually guide the process of constructing feature templates for other classifiers. Finally, one very important note about this implementation is that it was used during the preparation of a shared task which involved more that 50 datasets on which various tasks had to be performed. The enhanced speed of this algorithm allowed us to make over 1000 training runs and explore a rich set of features, in the short available time, which, without the optimization would otherwise had been impossible.

Our implementation of the enhanced ID3 algorithm is written in C++ and we provide a JAVA library for the tree-pruning and prediction algorithms. The tool is freely available and can be downloaded[3].

[3] http://slp.racai.ro/

References

Susan Bartlett, Grzegorz Kondrak, and Colin Cherry. 2008. Automatic syllabification with structured svms for letter-to-phoneme conversion. In *ACL*. pages 568–576.

Mark Craven, Andrew McCallum, Dan PiPasquo, Tom Mitchell, and Dayne Freitag. 1998. Learning to extract symbolic knowledge from the world wide web. Technical report, DTIC Document.

Wei Dai and Wei Ji. 2014. A mapreduce implementation of c4. 5 decision tree algorithm. *International Journal of Database Theory and Application* 7(1):49–60.

Thomas G Dietterich. 2000. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning* 40(2):139–157.

Sittichai Jiampojamarn, Colin Cherry, and Grzegorz Kondrak. 2008. Joint processing and discriminative training for letter-to-phoneme conversion. In *ACL*. pages 905–913.

David D Lewis and Marc Ringuette. 1994. A comparison of two learning algorithms for text categorization. In *Third annual symposium on document analysis and information retrieval*. volume 33, pages 81–93.

Manish Mehta, Jorma Rissanen, Rakesh Agrawal, et al. 1995. Mdl-based decision tree pruning. In *KDD*. volume 21, pages 216–221.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, et al. 2016. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC 2016)*. pages 1659–1666.

Vincent Pagel, Kevin Lenzo, and Alan Black. 1998. Letter to sound rules for accented lexicon compression. *arXiv preprint cmp-lg/9808010* .

Slav Petrov, Dipanjan Das, and Ryan McDonald. 2011. A universal part-of-speech tagset. *arXiv preprint arXiv:1104.2086* .

J. Ross Quinlan. 1986. Induction of decision trees. *Machine learning* 1(1):81–106.

J. Ross Quinlan. 1987. Simplifying decision trees. *International journal of man-machine studies* 27(3):221–234.

J Ross Quinlan. 1993. *C4.5: programs for machine learning*. Elsevier.

Rajeev Rastogi and Kyuseok Shim. 1998. Public: A decision tree classifier that integrates building and pruning. In *VLDB*. volume 98, pages 24–27.

Helmut Schmid. 2013. Probabilistic part-ofispeech tagging using decision trees. In *New methods in language processing*. Routledge, page 154.

Adriana Stan, Junichi Yamagishi, Simon King, and Matthew Aylett. 2011. The romanian speech synthesis (rss) corpus: Building a high quality hmm-based speech synthesis system using a high sampling rate. *Speech Communication* 53(3):442–450.

Milan Straka, Jan Hajic, and Jana Straková. 2016. Ud-pipe: Trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*.

Jiang Su and Harry Zhang. 2006. A fast decision tree learning algorithm. In *AAAI*. volume 6, pages 500–505.

György Szarvas, Richárd Farkas, and András Kocsor. 2006. A multilingual named entity recognition system using boosting and c4. 5 decision tree learning algorithms. In *International Conference on Discovery Science*. Springer, pages 267–278.

Robert Weide. 2005. The carnegie mellon pronouncing dictionary [cmudict. 0.6].

Sarah Zelikovitz and Haym Hirsh. 2001. Using lsi for text classification in the presence of background text. In *Proceedings of the tenth international conference on Information and knowledge management*. ACM, pages 113–118.

Daniel Zeman. 2008. Reusable tagset conversion using tagset drivers. In *LREC*.

Heiga Zen, Takashi Nose, Junichi Yamagishi, Shinji Sako, Takashi Masuko, Alan W Black, and Keiichi Tokuda. 2007. The hmm-based speech synthesis system (hts) version 2.0. In *SSW*. Citeseer, pages 294–299.

Heiga Zen, Keiichi Tokuda, and Alan W Black. 2009. Statistical parametric speech synthesis. *Speech Communication* 51(11):1039–1064.