MPI Solutions to Feature Extraction and Ensemble Learning

Kei Imada
Swarthmore College
kimada@swarthmore.edu

Matthew Parker
Swarthmore College
mparker3@swarthmore.edu

Abstract

Analysing articles takes time. Given a training set with articles, data scientists need to parse the text and then continue on to an iterative process of finding features, extracting the features, and testing the features on various models. Fortunately, most of these processes are embarrassingly parallelizable over different article instances, as they are independent from one another. In order to ease the text analysis model development process we provide an interface `FeatureExtractor` to easily add or remove features from training sets, distributed MPI solutions to parse a set of articles and extracting features from the set of articles. We also provide two distributed classifiers, a Naive Bayes and a voting model. We found that our interface successfully eased the development process, as modifying the set of features only required at most 20 lines of Python code. Our distributed solutions generally showed a linear speedup, as expected of the embarassingly parallel nature of these processes.

1 Introduction

In preparation for the SemEval 2019's Hyperpartisan News Detection task, we found that statistical machine learning approaches were effective in detecting hyperpartisian news. In particular, ensemble approaches, including voting classifiers, showed promise for high performance.

However, we found that natural language processing (NLP) takes time. Adding and removing features from training data entails rerunning the learning algorithm for the new features, which takes a nontrivial amount of time depending on the features and the architecture of the models. What is more, extracting multitudes of features from massive amounts of training data takes too long on a single machine with one process. Furthermore, ensemble models train multiple models, subtracting from the limited time we had for our tasks. All in all, the massive amount of time it takes to train and test our models were an impediment to our progress.

As a result, we looked into ways to speedup training, and realized that we would be able to achieve decreased training times with distributed methodologies, both in feature extraction (for all classifiers) and in training (for ensemble classifiers). While the ensemble classifier speedup is of interest to those using ensemble methods to accomplish NLP tasks, distributed feature extraction is applicable to most approaches to the SemEval task. Increased speed for feature extraction and ensemble learning will decrease the time needed for hyperparameter tuning and feature development for this task. As such, we believe it a worthwhile task, especially as institutional access to distributed computer systems increases.

In this paper, we intend to provide three contributions, two being parallel and distributed solutions and one being an interface to ease data science development:

- The `FeatureExtractor` class
- Distributed Feature Extraction
- Distributed Classifiers / Ensemble Methods

1.1 The `FeatureExtractor` Class

When creating models for NLP tasks, one of the many bottlenecks we found is finding the significant features. If we are lucky, we may find our intuition to be correct and are able to swiftly complete the task, but often times we do not know what features to train our models on. This usually entails an iterative approach where we repeatedly add or subtract features from our training data and test our models. The time complexity of such an approach is highly dependent on how much time is required to add or subtract features from the training data.

Hence we provide `FeatureExtractor`, a class that allows easy addition or removal of features from training data. This framework will allow for rapid testing with different combinations of features to find the model best suited for classification. In essence, our end goal on this front is to create an interface both general and simple enough that would make the feature extraction experience simpler and easier for data scientists.

From our research on hyperpartisan news, we created modules that each extracted specific features from training articles for rapid testing. The modules we created include bag of words, punctuations, and all capital words, all from the content of the article and from the title of the article. Combinations of these features were tested very swiftly using `FeatureExtractor`, exhibiting the simpleness of our framework.

1.2 Distributed Feature Extraction

Extracting features from text takes a substantial amount of time if there is a significant amount of required parsing. Generally speaking, feature extraction from an instance is entirely independent from one another, which means the process is embarrassingly parallel if we partition the training data over instances. This means that we could anticipate a near linear speedup if we distribute the feature extraction process over a cluster.

Our end goal for this front is to achieve at least a linear speedup for our feature extraction phase. We calculate speedup by comparing the runtime of feature extraction on one process to multiple processes.

1.3 Distributed Classifiers / Ensemble Methods

Finally, we propose distributed classifiers that fits on training data and classifies on testing data in parallel. We also propose distributed ensemble classifiers, which trains the models used in the ensemble in parallel on each machine. In particular, we investigate voting methods, as we deemed it was highly parallelizeable. Boosting methods (fitting multiple weak learners and weighting them based on their performace on hard-to-classify examples) and bagging methods (training multiple weak learners on random subsets of the training data set) are similar in that they both feature training of multiple individual models, and hence lend themselves well to parallelization. We ultimately decided against these approaches, as maintaining and updating distributed classifier weights in boosting proved to be time-consuming in implementation, and depending on the parameters (especially subset size), the amount of memory consumed by each core for bagging could collectively put a strain on the campus system.

As a result, implementing a distributed voting classifier makes sense for this task. As a form of ensemble learning, a distributed voting classifier can demonstrate speedup for general ensemble learning tasks, This method is a parallel solution to speed up ensemble learning while restraining memory usage, as one node only needs to keep one set of training data. Our end goal, similar to our second goal, is to achieve near linear speedup. (Polikar et al., 2012) (Zhou., 2012).

2 Previous Work

Due to advances in distributed computing technology, as well as the Swarthmore College distributed computing network at our service, we were able to select from multiple distributed computing methodologies in our implementation of distributed algorithms for this task. We examined Apache Spark, Apache Hadoop, and the Message Passing Interface (MPI) for this purpose.

Spark is known for its high levels of data parallelism across multiple cores, as well as its speed with iterative algorithms (Zaharia et al., 2010). However, we found that data resilience was less of a priority for our purposes, as our training data was small enough to the point where resilient storage was not an issue, and caching it on one machine could make sense for our purposes. Additionally, our implementation of feature extraction was able to perform in one pass on each node, and the results from one node did not affect the results on other nodes, so fault tolerance was not an issue.

Apache Hadoop was another distributed computing framework we considered using for this project. Hadoop would have been more appropriate for our usages – it is a Java implementation of the MapReduce programming paradigm specified by Jeff Dean and Sanjay Ghemawat in their seminal 2004 paper (Dean and Ghemawat, 2004).

The MapReduce paradigm consists of two steps a "map" step partitioning a dataset over a cluster and converting it into an optimal format, and a "reduce" step returning the desired conclusions from the data. The fundamental innovation made by Dean and Ghemawat wasn't the creation of the

paradigm, as similar functionality existed in the renounced MPI, but rather their distributed implementation with increased scalability and high fault tolerance. This has evident use in feature extraction, as well as in building ensemble models.

While Hadoop and other MapReduce implementations clearly held promise for our tasks, we decided against using them for a few reasons, but mainly, because of interoperability. Hadoop is written in Java, and our project is almost exclusively written in Python. Using Hadoop would require writing a lot of boilerplate IO, as we couldn't find satisfactory Python bindings for it. Moreover, we wouldn't be able to easily run it on the Swarthmore College's distributed computing system. For these reasons, we opted to roll our own distributed feature extractors and ensemble classifiers using the Python MPI implementation `mpi4py`.

The Message Passing Interface has support for parallel and distributed architectures with thread-safe interfaces and more, making it the best candidate for our use case (Dongarra et al., 1996). We considered a few ensemble learning algorithms for this task, namely AdaBoost classifiers and bagging classifiers. In looking at bagging, one paper (Chawla et al., 2003) revealed that better-than-single-classifier performance could be achieved with parallel bagging on a relatively low number of instances. Since the number of computers making up our distributed system has a relatively low order of magnitude, this is promising for any distributed ensemble implementation we choose to explore.

A distributed implementation of the AdaBoost algorithm was developed in 1999 (Fan et al., 1999), created through multiple distributed instances of weak classifiers learning on non-random samples from the total data set generated through picking a mix of easy-to-classify and hard-to-classify examples. In this approach, the sequential nature of AdaBoost was preserved. This resulted in speedup (linearity not specified) as well as equal or superior performance as compared to the non-distributed version.

3 `FeatureExtractor`

To simplify the model development process, we created three abstract data types: `Vocabulary`, `Features`, and `Labels`. As shown in Figure 1, The `Features` type would take in a labeled article a and map it to its corresponding feature set f. The `Vocabulary` type contains references that will be used by the `Features` type to find the features. For example, for a `Features` type that finds the bag of words features of an article, the `Vocabulary` type would contain all possible words the `Features` class would anticipate. For a `Features` type that finds the punctuation features of an article, the `Vocabulary` type would contain all possible punctuation. Lastly, the `Labels` type would simply take in the labeled article and extract the label.

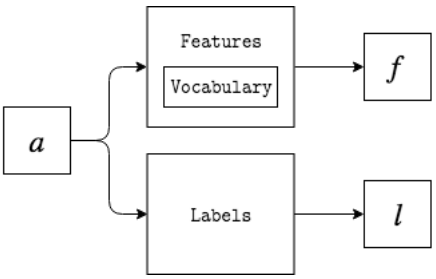

Figure 1: The architecture of how features and labels are extracted from an article with the `Vocabulary`, `Features`, and `Labels` types. a represents the source labeled article (already parsed), f represents the set of extracted features from a, and l represents the label the `Labels` type extracted from a.

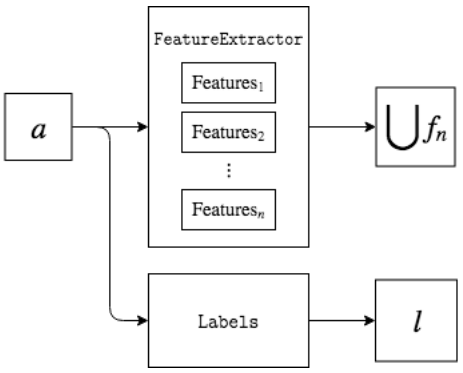

Figure 2: The feature extraction done with the `FeatureExtractor` type. The interface is the same with `Features`, and will output a union of the feature sets $\{f_n\}$ extracted from the n `Features` types.

Since it is common to frequently add or remove features from our data set, we created a `FeatureExtractor` subtype for the `Features` type. As shown in Figure 2, the `FeatureExtractor` type takes in an article, refers to its list of `Features` to extract features, and returns a union of the set of features extracted by every `Features` type.

Buzzfeed reports that many hyperpartisan news report fake news, motivating our approach to de-

tect fake news, which is widely researched (Silverman et al., 2016). Horne lists an index of features that correlate with fake news, such as title length, and uppercase words (Horne and Adali, 2017). We implemented these feature extractors as a test to measure the simplicity of our framework.

4 Distributed Methods

We identified three major steps in the process that we could distribute parsing articles, feature extraction, and models (both ensemble and their individual learners).

We leveraged the OpenMPI abilities of the Swarthmore Computer Science labs to use various connected lab machines to perform these large computations. We parse articles and learned our models on commodity computer cluster provided by Swarthmore College. Each of the machines we used had 8 core Intel(R) Xeon(R) CPU E3-1245 v5 @ 3.50GHz and 31 Gb of memory. These machines had `mpi4py==3.0.0` and a Network File System (NFS) installed on them. We use NFS as the central network location to cache processed data. Each machine had at most two MPI processes to reduce the memory load of on each machine.

For the rest of the paper, in order to streamline our descriptions, let us define these terminologies:

- $m :=$ the number of machines
- $n :=$ the number of training articles
- $q :=$ the number of validation articles
- $\{c_i\} :=$ the set of machines
- $\{a_i\} :=$ the set of training articles (not parsed)
- $\{a'_i\} :=$ the set of parsed training articles
- $l_i :=$ the classifications for a training article a_i
- $\{v_j\} :=$ the set of validation articles (not parsed)
- $\{v'_j\} :=$ the set of parsed validation articles
- $\{M_k\} :=$ the set of trained models where each $M_k : \{f'_j\} \to \{l'_j\}$ is a bijective function
- $l'_j :=$ the classifications a model M_k generated for f'_j
- $P : \{a_i\} \cup \{v_j\} \to \{a'_i\} \cup \{v'_j\}, a_i \mapsto a'_i, v_j \mapsto v'_j :=$ the bijective parsing function
- $E : \{a'_i\} \cup \{v_j\} \to \{f_i\} \cup \{f'_j\}, a_i \mapsto f_i, v_j \mapsto f'_j :=$ the bijective feature extraction function using our `Features` type (or more specifically, our `FeatureExtractor` type)
- $L : \{a'_i\} \to \{l_i\}$, the bijective labeling function running `Labels`
- $\{T_k : \{\{f_i\}\} \to \{M_k\}\} :=$ the set containing learning functions that map the training data feature matrix $\{f_i\}$ to a trained model M_k

4.1 Parsing

As one might suspect, the parsing of an article is independent from one another, thus parsing multiple articles is embarrassingly parallel. Hence we made a simple MPI scatter/gather program to scatter the articles $\{a_i\}$ with m equal sized partitions into each of the m machines running an instance of P to output parsed articles $\{a'_i\}$. The output of parsed articles is then cached to a central network location where we can access for all $c \in \{c_i\}$.

For our purposes, our parsing function E extracts tokens, lemmas, and parts of speech tags from both titles and contents using `spacy==2.0.12`.

4.2 Feature Extraction

Feature extraction was identified as a straightforward use case for distributed and parallel tasks because of the nature of the work. To be more specific, we ran the feature extraction function E on each c_i, and partitioned the n articles to m equal partitions. We then scattered the m partitions to the m machines, and then gathered the output feature set to create our sparse feature matrix. After a parsed article a'_i goes through the feature extraction function E to output f_i, its row in the training data feature matrix is not mutated.

4.3 Distributed Naive Bayes Classifier

If we let $\forall T \in \{T_k\}, T =$Naive Bayes learning function, every model $M \in \{M_k\}$ is an identical Naive Bayes classifier. We can use this cohort of Naive Bayes classifiers as a distributed Naive Bayes classifier, where we could partition the q validation feature sets to m equally partitioned sets, and distribute them to M_k in parallel in order to get q classifications.

4.3.1 Correctness

We tested our correctness of distributed feature extraction and our distributed Naive Bayes model by checking our model output with our previous sequential single process. We indeed got the same

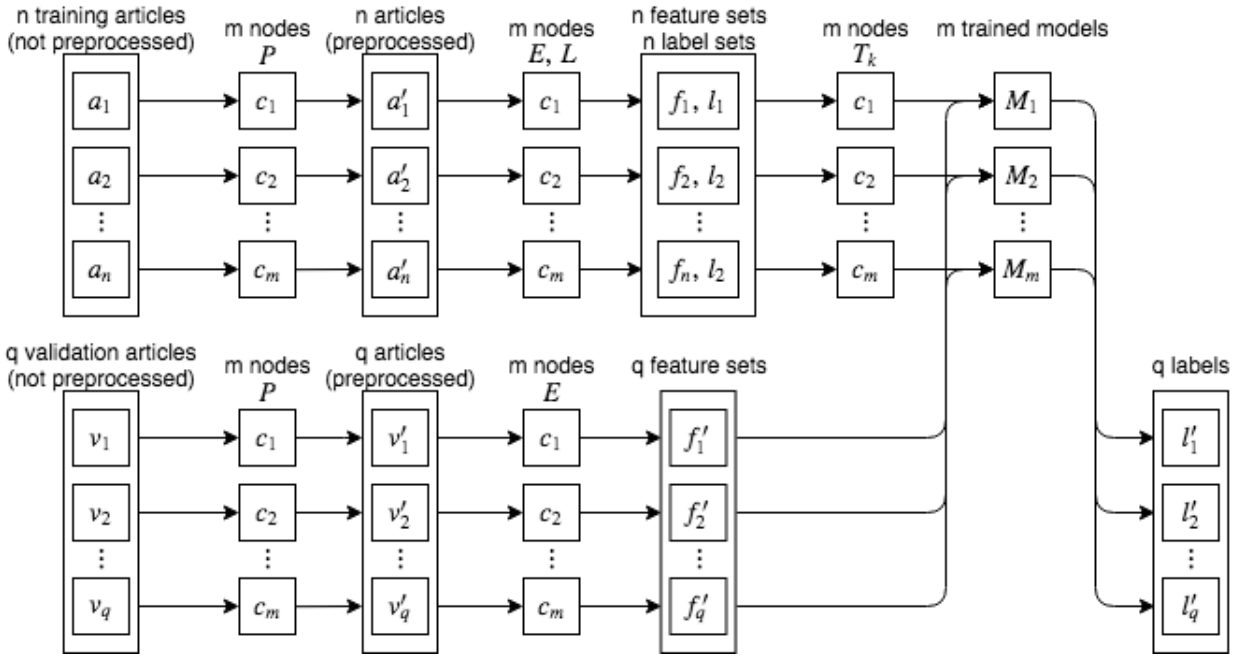

Figure 3: The pipeline that takes in n training articles and q validation articles, and outputs q classifications. The boxes around the articles, features, and classifications signify that the outputs of the distributed algorithm is AllGathered in MPI terms (gathered and then broadcasted) (Dongarra et al., 1996).

results, verifying the correctness of our implementations.

4.4 Distributed Voting Classifier

For the Distributed Voting Classifier, we created an extension to `scikit-learn==0.19.2`'s `VotingClassifier`. This classifier takes in a list of `classifier_name, classifier` tuples, and fits them (in parallel, if possible) to a given training data set. It then uses the weighted or non-weighted votes of those classifiers to output a final classification for test data. We identified the model fitting step as an opportunity to use distributed computing, and assigned each machine n classifiers, where n is equal to $|\{M_k\}|/m$ where $|\{M_k\}| = 32$ for testing purposes (although adjustable to any number of models). We then ran the training step in parallel, with each classifier training on its assigned machine.

4.5 Measuring Scalability

Because parsing takes so much more time than feature extraction and learning the Naive Bayes/Voting model, we measured how much time it would take to parse 645 articles in contrast to the 150000 articles we used to extract features and learn our models. In general, we extracted bag of words features with 30000 words in our `Vocabulary`, skipping the first 100 stop words.

5 Simplicity of `FeatureExtractor`

With the framework we developed, we found that it was quite simple to add new entries to our repertoire of features. For each `Features` class, we only had to add around 10 lines of Python code to our code base. Sometimes we had to add another `Vocabulary` class for our `Features` class (e.g. a punctuation `Vocabulary` class), and those were also around 10 lines long.

In order to add or remove features from our training data, all we had to do was to add or remove elements from the list of `Features` classes we passed into our `FeatureExtractor` type. We saw no limitations for this portion of our experiment.

6 Scalability of Distributed Parsing

As shown in Figure 4, we see a linear speedup of parse time, confirming our prediction that parsing articles is highly parallelizable. We see a small amount of sub-linearity at 32 nodes, which could be explained by the communication required to gather the parsed data together to one parsed file.

7 Scalability of Distributed Feature Extraction

We found that our hypothesis on the scalability of feature extraction to be aligned with the result. As

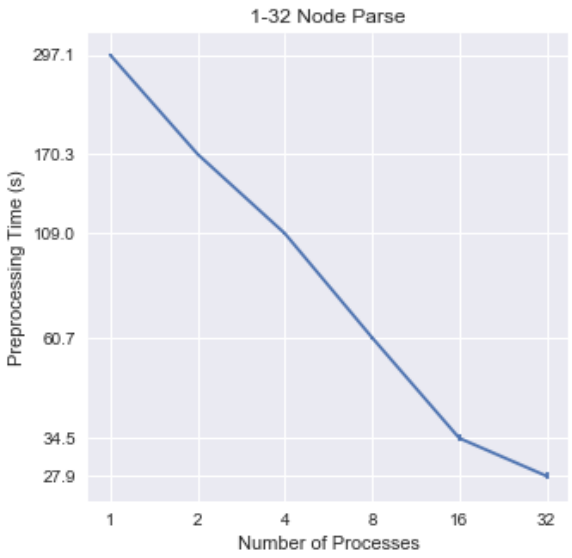

Figure 4: Parsing Time (s) vs Number of Processes. For this experiment we extracted tokens, lemmas, and parts of speech tags from both titles and contents using `spacy==2.0.12` from 645 articles with three trials. *The relationship fits our hypothesis of near linear speedup.*

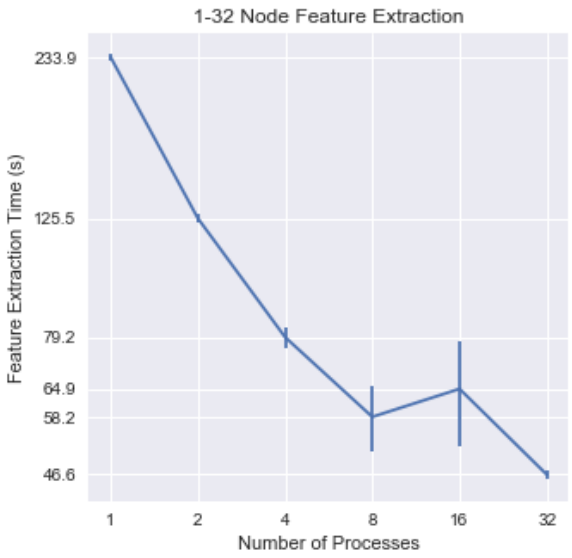

Figure 5: Feature Extraction Time (s) vs Number of Processes. For this experiment we extracted bag of words features from the content of 150000 articles with 30000 features. We timed our runs three times. *The relationship shown mostly fits our hypothesis of near linear speedup.* The anomaly at 16 processes may be caused by the fact that we used machines that we did not have exclusive access to, which also explains the high standard deviation.

shown in Figure 5, it is clear that we can achieve near linear speedup, but our speedup decreases as we increase the number of processes as shown by the positive concavity of our graph.

We see an anomaly around 8 and 16 processes where there is a high standard deviation and the askew data point at 16 processes. Since we tested our experiment on the same machines sequentially, in other words, we tested 8 process runs shortly after 16 process runs, it is possible that an unknown memory intensive software was run on one or more machines, which caused the machines to swap when we ran our feature extraction program. For our future work, we intend to experiment on machines that we have exclusive access to.

8 Scalability of Distributed Naive Bayes Classifiers

In comparison with the distributed feature extraction, we found that distributing the classification function of our Naive Bayes model is linear up to some point, after which we see a decrease in performance. As shown in Figure 6, we achieve linear speedup until 4 processes, whereafter we get an increase in classification time. This is explained by the communication time required to gather the results to report back to the user. Also note the short 12.0 seconds used to classify 150000 articles in contrast to the 233.9 seconds to extract bag of words features from the same amount of articles – we deem that it is more beneficial to speed up preprocessing and feature extraction rather than classification.

We found a generally high standard deviation for 4-32 processes and an askew 16 processes point. We will bring up the aforementioned possible cause of an anonymous memory intensive program being run on one or more machines while our experiment was run.

9 Scalability of Distributed Ensemble Training

When we trained our ensemble learner on multiple cores, we saw near-linear speedup up to 8 cores, at which point the speedup became sublinear. We also noted the same deviation as in feature extraction at 16 cores the training time increased, rather than decreased, before finally decreasing to the lowest observed data point at 32 cores. The spike at the 16-core mark, as seen in Figure 7, suggests some sort of anomaly during training, consistent with the similar occurrences

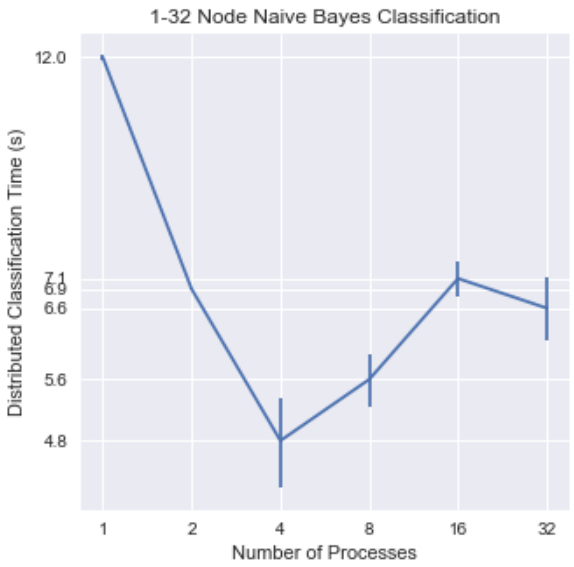

Figure 6: Classification time for our distributed Naive Bayes model in seconds vs Number of Processes. We classified 150000 articles with three trials. *We see that we achieve near linear speedup until we reach 4 nodes, from which point we get negative speedup because of communication costs.* Again, we see an anomaly at 16 nodes, which we think it is because of the limitations of the exclusivity of the machines we used for our experiment.

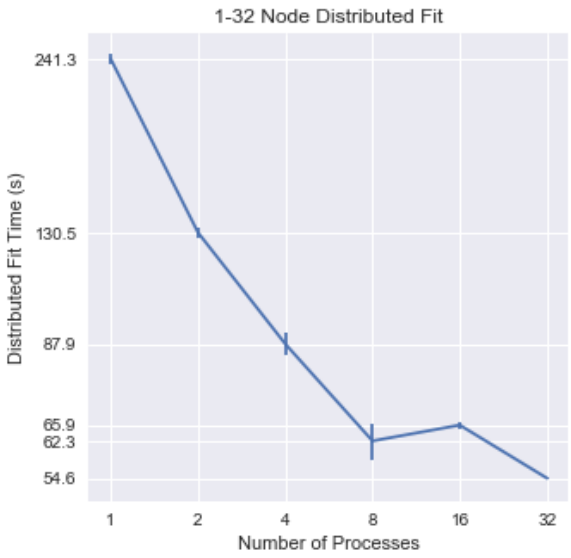

Figure 7: Training Time for our distributed Voting Classifier. Notable here is the relatively low standard deviations for 1, 2, 4, and 32 cores, and the relatively high standard standard deviations for 8 and 16 processes. This Voting Classifier used 32 Decision Tree Classifiers with depth 3 with 150000 training instances with 30000 features to train and make its decisions. These Decision Tree Classifiers proved to be time-consuming to train in a single-process model, but were more feasible to use in a distributed model.

above. We can achieve near-linear speedup due to the embarrassingly parallel nature of the work, illustrating promising benefits of using distributed systems with ensemble models.

10 Limitations

Since the machines we used were shared devices for the Swarthmore College computer science community, we did not have exclusive access to these machines. This means it is possible for other users to run memory intensive programs at any time during our experiment. If the programs has high memory usage, we could have had swapping behavior, which would affect our timings of our experiment, as seen on Figure 5, Figure 7 and Figure 6.

Since we had a limited number of machines with similar specifications, we ran two processes on one machine. This could mean that we could have unintentionally sped up our program by enabling access to shared memory. However, we believe that the speed up gained by shared memory is inconsequential to our data because we ran this configuration for every trial.

11 Discussion

In this paper we present three contributions: the `FeatureExtractor` class to make text feature extraction simpler, distributed solutions to the time consuming processes of parsing and feature extraction, and distributed ensemble methods.

We found that our `FeatureExtractor` class successfully simplified the model development process to around 10 lines of Python code to add a feature (around 20 if a `Vocabulary` class need to be added). In order to add or remove features from our training data, we only had to make the changes to the `Features` class list passed into the `FeatureExtractor` class.

Moreover, we confirmed that distributing parsing and feature extraction is a highly effective method to speed up the processes, as we found near linear speedup. We were able to cut down the 297.1 seconds to parse 645 articles on one process to 27.9 seconds with 32 processes, and 233.9 seconds to extract features from 150000 articles to 46.6 seconds with 32 processes.

Distributing the classification is linear up to some point, whereafter we found that performance

is decreased because of the required communication overhead. However, we deem that it is more beneficial to speed up preprocessing and feature extraction rather than classification, given the short 12.0 seconds to classify 150000 articles with one process.

For distributing ensemble models, we confirmed that distributed computing is a highly effective method to reduce training time. We saw near-linear speedup when training our voting classifier simply by using a small number of processes rather than a single-process sequential approach, and although speedup tapered off as the number of processes grew higher, we were still able to reduce the time needed to run the training step. This holds promise for future distributed ensemble learners distributed bagging techniques could be implemented and get the same speedup benefits as our voting classifier does. While this distributed paradigm isn't as easily translatable to boosting classifiers because of their inherently less parallelizable nature, it provides evidence that distributed boosting algorithms could also be used to reduce training times.

11.1 Future Work

Since we were more familiar with MPI than MapReduce, we chose MPI for this paper. However, we believe MapReduce is more suited for this task, as it fits their paradigm of processing and generating large data sets using map and reduce functions. If we continue on with this thread of work, we may port our implementations to MapReduce in order to further increase our target demographics.

We limited to two processes per machine because increasing any more would cause swapping, causing a major slowdown of the program. Since we had 8 cores on each machine, it is possible to multithread the processes while keeping the order of used memory constant to further speed up parsing and feature extraction. An alternative is to use network RAM implementations such as Dr. Newhall's NSwap to prevent swapping behavior on the cluster via shared network memory (Newhall et al., 2003).

We partitioned the set of articles based on the number of articles. However, some articles were one sentence long whereas longer articles were several paragraphs long, which would mean our partitioning method could have resulted in an uneven partition. A better approach, an approach we will attempt in the future, would be to weigh the articles based on their lengths when partitioning the articles.

Since we focused on distributing and speeding up our development and training process, we did not experiment as much as we wanted on the actual performance of these models. Future work could include testing with different classifiers in our distributed voting classifier and find which ones are more suited to classify hyperpartisan articles.

References

Nitesh V Chawla, Thomas E Moore, Lawrence O Hall, Kevin W Bowyer, W Philip Kegelmeyer, and Clayton Springer. 2003. Distributed learning with bagging-like performance. *Pattern recognition letters*, 24(1-3):455–471.

Jeffrey Dean and Sanjay Ghemawat. 2004. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA. USENIX Association.

Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. 1996. A message passing standard for mpp and workstations. *Commun. ACM*, 39(7):84–90.

Wei Fan, Salvatore J. Stolfo, and Junxin Zhang. 1999. The application of adaboost for distributed, scalable and on-line learning. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 362–366, New York, NY, USA. ACM.

Benjamin D. Horne and Sibel Adali. 2017. This just in: Fake news packs a lot in title, uses simpler, repetitive content in text body, more similar to satire than real news. *CoRR*, abs/1703.09398.

Tia Newhall, Sean Finney, Kuzman Ganchev, and Michael Spiegel. 2003. Nswap: A network swapping module for linux clusters.

Robi Polikar, Cha Zhang, and Yunqian Ma. 2012. *Ensemble Machine Learning: Methods and Applications*, 1 edition. Springer-Verlag New York.

Craig Silverman, Ellie Strapagiel, Hamza Shaban, Ellie Hall, and Jeremy Singer-Vine. 2016. Hyperpartisan facebook pages are publishing false and misleading information at an alarming rate.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA. USENIX Association.

Zhi-Hua Zhou. 2012. *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC Machine learning & pattern recognition series. Chapman & Hall / CRC Press.