Scaling Feature Type Inference

A. ZHOU and E. MANIRASITH*, University of California San Diego, United States

The widespread adoption of machine learning in industry has lead to the rise of Automated Machine Learning (AutoML) to enable scaling across a growing number of industrial tasks. However, current AutoML systems use simplistic and inaccurate heuristics for feature type inference, the first step in the AutoML pipeline, leading to significant decreases in final model performance. In this paper, we develop a scalable Apache Spark-based proof of concept for effectively scaling feature type inference to terabyte-scale datasets, and present experimental validation of up to 500GB. As a secondary contribution, we also present a toolkit for quickly configuring single-user Spark Clusters.

Additional Key Words and Phrases: AutoML, Apache Spark, Feature Type Inference, Big Data

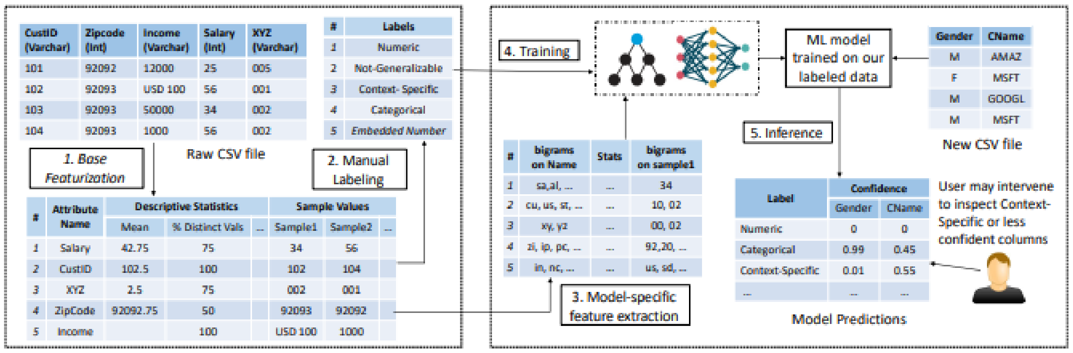
1 INTRODUCTION

The field of data science has seen intense growth over the past decade, especially in the area of machine learning. As a result of this growth, and the ongoing development of sophisticated tools, modern industrial machine learning often involves some level of automation in the machine learning lifecycle. Automated Machine Learning, or AutoML, now exists in the form of open source packages and distinct commercial offerings, like Google's Cloud AutoML, Amazon's SageMaker, and Microsoft's Azure Machine Learning services. Due to this investment of development effort, many tasks can be reduced down to plugging a dataset into a cloud platform service. Right now, the current state of the art in available AutoML offerings successfully handles feature engineering, model selection, and hyper-parameter tuning, in part due to extensive research in these areas over the past decades.

One commonly cited anecdote in introductory data science courses is "95% of Data Scientist time is spent cleaning data". The authors' experiences in academia and industry concur. However, despite these commonly cited anecdotes, the step of data preparation does not see much research focus, perhaps due to a perception of lower prestige. Although expert data preparation may produce high quality datasets and better performance, this comes at the cost of requiring expert time, which in turn reduces the scalability across tasks and datasets alike. As a result, automated data preparation is a *mandatory* complement to expert work that allows for application of Machine Learning to otherwise cost-ineffective tasks.

As a result, the ADA lab at UCSD have started Project SortingHat which aims to research new ways to automate data preparation with consistent accuracy. Project SortingHat uses an ML-based approach to automate ML data preparation by deconstructing ML data preparation tasks into prediction tasks. From Project SortingHat led to the creation of the ML Data Prep Zoo. The ML Data Prep Zoo is a repository of benchmark labeled datasets and pre-trained ML models for AutoML data preparation. The first AutoML data preparation task implemented was feature type inference. To automate feature type inference the task was formalized into a multi-class ML classification problem. Using the pre-trained models in ML Data Prep Zoo the ADA lab have been able to conclude that using the classical ML model Random Forest results in the greatest accuracy with each feature type. Therefore, with renew interest in new AutoML data preparation techniques there is a push to create scalable implementations for industry use.

Feature Type Inference is the task of identifying feature types from a given feature. As an example, the string `92092` can represent everything from the University of California San Diego's ZIP code to a product's price. Without further context, predictions will likely have low accuracy. As the first step in the data preparation process, poor results here will affect all downstream processing, with an performance lift of 7% across 30 datasets compared to existing open source systems when using a specifically trained model [5].


Raw CSV file
1. Base Featurization
2. Manual Labeling
4. Training
ML model trained on our labeled data
5. Inference
New CSV file
3. Model-specific feature extraction
User may intervene to inspect Context-Specific or less confident columns
Model Predictions

CustID (Varchar)	Zipcode (Int)	Income (Varchar)	Salary (Int)	XYZ (Varchar)
101	92092	12000	25	005
102	92093	USD 100	56	001
103	92093	50000	34	002
104	92093	1000	56	002

#	Labels
1	Numeric
2	Not-Generalizable
3	Context- Specific
4	Categorical
5	*Embedded Number*

#	Attribute Name	Descriptive Statistics			Sample Values		
		Mean	% Distinct Vals	...	Sample1	Sample2	...
1	Salary	42.75	75		34	56	
2	CustID	102.5	100		102	104	
3	XYZ	2.5	75		002	001	
4	ZipCode	92092.75	50		92093	92092	
5	Income		100		USD 100	1000	

#	bigrams on Name	Stats	bigrams on sample1
1	sa,al, ...	...	34
2	cu, us, st, ...	...	10, 02
3	xy, yz	...	00, 02
4	zi, ip, pc, ...	...	92,20, ...
5	in, nc, ...	...	us, sd, ...

Gender	CName
M	AMAZ
F	MSFT
M	GOOGL
M	MSFT

Label	Confidence	
	Gender	CName
Numeric	0	0
Categorical	0.99	0.45
Context-Specific	0.01	0.55
...	...	...

Fig. 1. The ML-based feature type inference workflow. (Reproduction of Figure 4 from Shah et al.)

Despite this performance gain, productionizing this model for use in industry faces several challenges. In this paper, we tackle the problem of *scalability*. The original base featurization routine presented by Shah et al. uses Pandas for computation [6]. Unfortunately, many high-quality features useful for classification require full-dataset aggregation. As industrial datasets often reach Terabyte scale, and Pandas operates on a single machine's main memory, AutoML with such tools will likely require a different tool [2]. As it is the industrial big data processing tool of choice, we re-implement these routines on Apache Spark [8].

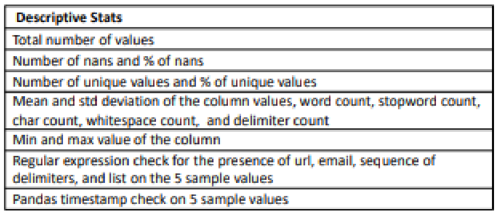
2 TASK

ML Data Prep Zoo has proven that automated feature type inference can be accurately done. This can be seen with the Random Forest model achieving a high accuracy of 91.2% when inferring feature types (Shah et al.). Therefore, we intend to scale this automation, so that it may be industry ready.

Figure 1 shows the entire workflow of AutoML feature type inference. The first step is to take the raw datasets and extract certain base features. Next, during training, these base features are labelled with a ground truth and used to create a model. Finally, at runtime, the trained ML model is used to infer feature types on a fresh dataset using the same base featurization.

The first step, base featurization takes the raw dataset and extracts specified base features needed from the dataset. These features include column name, descriptive statistics, and 5 sample values. The descriptive statistics include 25 descriptive stats (see Figure 2) ranging from total number of values to mean and standard deviation. For the sample values, 5 randomly distinct sampled values are chosen from each column.

Base featurization takes on average the longest amount of time during AutoML feature type inference and the whole workflow is bottle-necked at this step. This is because base featurization is the only step in the workflow that iterates through the whole dataframe including every row and column. Therefore, base featurization has a time complexity of $O(NM)$, while the actual inference (step 5) only has a time complexity of $O(M)$ where N and M are respectively the number of rows and the number of columns. Therefore, to scale AutoML feature type inference we intend to re-implement *base featurization* in PySpark



Descriptive Stats
Total number of values
Number of nans and % of nans
Number of unique values and % of unique values
Mean and std deviation of the column values, word count, stopword count, char count, whitespace count, and delimiter count
Min and max value of the column
Regular expression check for the presence of url, email, sequence of delimiters, and list on the 5 sample values
Pandas timestamp check on 5 sample values

Fig. 2. The descriptive stats used for base featurization. (Reproduction of Table 6 from Shah et al.)

3 SCALING

3.1 PySpark

To scale *base featurization*, we reimplemented the SortingHat base featurization algorithm in PySpark. Like the Python code, we assume that the input PySpark DataFrame is read with initial datatype inference. In Pandas, this is done by default; in PySpark, the `inferSchema` argument of `DataFrameReader.csv()` must be set to `true`.

With our implementation we perform two full scans of the dataframe. The first scan acquires all the descriptive statistics (figure 2). For the descriptive statistics we use a dictionary to loop through specified pyspark.sql.functions and extract the statistics needed all while iterating through the entire dataset. The second scan obtains the 5 sample values. This scan will iterate through the dataset and obtain 5 distinct values non-deterministically. If the column has less than 5 distinct values then the first n(n<5) values are chosen.

3.2 Scalability Experiment

After confirming our implementation of PySpark for Base Featurization is indeed accurate, we then intend to study the preprocessing time of our implementation. To do this we intend to time (seconds) our implementation for a series of runs. To test for scalability we will use a range of sizes for the input csv starting around 1 GB then increasing till we reached at least >100 GB of data.

The csv we chose for our experiment is the members.csv. This data was collected from meetup.com API in December 2017 and includes a list of members from the site at the time. The inital size of the csv is about 1.25 gbs and it has about 6 million rows and 14 columns. For our experiment we doubled (2x) the rows of the input csv after each timed run until we reached cluster capacity (Table 1). Based on our cluster configuration we were able to scale the csv till it reached around 600 gbs.

To make sure we only time the function itself all csvs used in the experiment were saved to memory first before running the experiment and the same SparkSession was used throughout.

For our experiment cluster, we use an OpenStack Ussuri cluster provisioned on CloudLab Wisconsin [3] [1]. Specifically, we set up an OpenStack cluster on five c220g5 worker nodes, each with 20 CPU cores, 192 GB of memory, and a 10 Gb/s network interface. The OpenStack controller node is similarly provisioned.

On this cluster, we use Spark-Up to provision ten worker nodes for Spark and HDFS. Each node has 8 virtual CPU cores, 80 GBs of memory, and 500 GBs of disk. On each node, we dedicate all CPU cores and 60 GiB of memory to Spark, and approximately 375 GBs of disk space for HDFS. We also provision a cluster manager node for the Spark master and the HDFS Namenode with 4 virtual CPU cores, 24 GBs of memory, and 160 GBs of disk. All machines at all layers of this cluster use Ubuntu 20.04 Focal, and thus version 5.4.0 of the Linux kernel. The experiment itself uses Python 3.10.2, Java 11 (Temurin 11.0.14.1+1), Spark and PySpark 3.2.1, and Hadoop 3.3.1 [4] [8] [7].

Trial	Size (GB)
0	0.649380
1	1.298761
2	2.597522
3	4.741887
4	9.483774
5	18.967548
6	37.935097
7	75.870194
8	151.740388
9	303.480775

Table 1. Filesize of compressed input data, stored in Parquet format.

Fig. 3

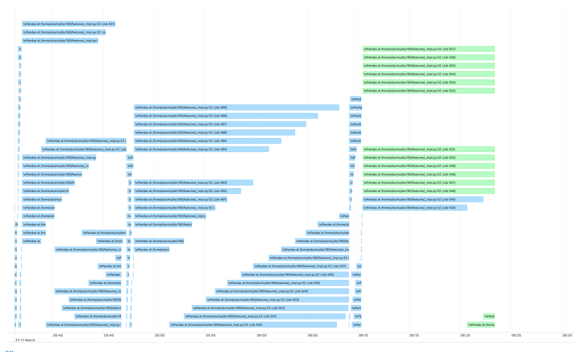


Fig. 4

3.3 Results

We present our experiment results in Figures 5 and 6.

First, we note that the experiment graphs appear relatively linear. Indeed, fitting a linear model between relative size and relative runtime returns a slope of 0.0815. That is, for every unit increase in input size, the relative runtime increases by approximately eight percentage points. We also find a Pearson correlation coefficient of 0.998972 and $r^2 = 0.997945$. Thus, we can safely conclude that runtime scales linearly with input size: only a negligible amount of variation in runtime is unexplained by variation in input size.

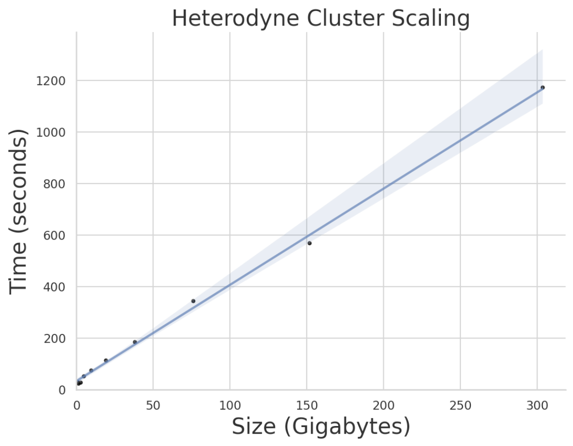


Fig. 5

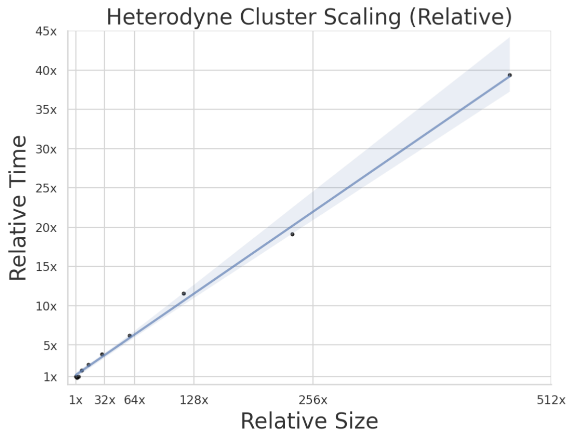


Fig. 6

Further inspection of Spark stage and job logs in Figure 3 and 4 reveal that the current implementation's bottleneck lies in `toPandas` calls; this is due to lazy evaluation. That said, the relative distribution of runtimes in 4 indicates that the bottleneck actually lies with the "Simple" features. However, we leave further optimization of this to future work.

Finally, it's worth noting that these results only apply for in-memory analytics, and likely will not scale beyond that data access regime. Furthermore, this dataset is relatively simple. Large numbers of high cardinality input features may also cause pathological worst-case behavior. However, caveats aside, we are confident that this proof of concept implementation will scale well into the terabytes – or petabytes, with enough high-memory servers. Given the general availability of cloud compute with terabytes of main memory, this should not pose an issue.

4 CONCLUSION

For theoretical improvements in data science to see realization in practical application, development of a scalable implementation is necessary. In this paper, we propose a proof-of-concept system for scalable feature type inference using Apache Spark. Validation tests of the scalability of this system using a medium-scale Spark cluster match expected results, with runtime scaling linearly with input data size. Finally, we describe limitations of this system for future work to resolve.

REFERENCES

[1] OpenStack contributors. 2020. *OpenStack*. OpenInfra Foundation. https://www.openstack.org/

[2] Pandas contributors. [n. d.]. *Pandas*. https://pandas.pydata.org

[3] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[4] Python Software Foundation and Python contributors. [n. d.]. *Python*. https://python.org

[5] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. Towards Benchmarking Feature Type Inference for AutoML Platforms. *Proceedings of the 2021 International Conference on Management of Data* (2021).

[6] Vraj Shah, Jonathan Lacanlale, Premanand Kumar, Kevin Yang, and Arun Kumar. 2021. *Towards Benchmarking Feature Type Inference for AutoML Platforms*. Technical Report. University of California San Diego.

[7] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[8] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664

ACKNOWLEDGMENTS

This work was supported by Arun Kumar, Vraj Shah, and the ADALab at the University of California San Diego. This material is based upon work supported by the National Science Foundation under Grant Nos. 1419199, 1743363, and 2027208. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

5 APPENDIX

5.1 Spark-Up

Spark-Up is a forthcoming toolkit for rapid deployment of varying Spark cluster topologies using platform-agnostic infrastructure as code technologies for single user

Specifically, the toolkit consists of three components:

- Packer VM image build configuration
- Terraform Infrastructure-as-Code definitions
- Ansible provisioning scripts

By splitting provisioning into a preparing static “baseline” virtual machine image and a runtime configuration step, the cost of preparing new Spark clusters is reduced to approximately ten minutes.

Despite being designed around the OpenStack environment, the toolkit is still of use to other environments. Although the Packer and Terraform configuration is tightly coupled to OpenStack out of necessity, porting to other cloud environments should take no more than one or two hours. Finally, the Ansible scripting is designed for easy reconfiguration – should it be necessary to use a nonstandard configuration, most relevant downstream parameters are exposed as easily-overridable variables. Indeed, this project's infrastructure uses this capability to create an asymmetrical Spark/HDFS manager/worker configuration.