Distributed Reverse DNS Geolocation

Ovidiu Dan*
Lehigh University
Bethlehem, PA, USA, 18015
ovd209@cse.lehigh.edu

Vaibhav Parikh
Microsoft Bing
Redmond, WA, USA, 98052
vparikh@microsoft.com

Brian D. Davison
Lehigh University
Bethlehem, PA, USA, 18015
davison@cse.lehigh.edu

***Abstract*—IP geolocation databases map IP addresses to their geographical locations. These databases are used in a variety of online services to serve local content to users. Here we present methods for extracting locations from the reverse DNS hostnames assigned to IP addresses. We summarize a machine learning based approach which, given a hostname, aims to extract and rank potential location candidates, which can then potentially be fused with other geolocation signals. We show that this approach significantly outperforms a state-of-the-art academic baseline, and it is competitive and complementary to commercial baselines. Since extracting locations from more than a billion reverse DNS hostnames at once poses a significant computational challenge, we develop a distributed version of our algorithm. We perform experiments on a cluster of 2,000 machines to demonstrate that our distributed implementation can scale. We show that compared to the single machine version, our distributed approach can achieve a speedup of more than 150X.**

***Index Terms*—IP geolocation; distributed algorithms; MapReduce; geographic targeting; geotargeting; geographic personalization; reverse DNS; hostname geolocation**

I. Introduction

IP geolocation databases are used by online services to determine the geographical location of users based solely on their IP address. They map IP ranges, typically of 256 IP addresses in size, to locations at the city-level granularity. Companies such as MaxMind, Neustar IP Intelligence , and IP2Location provide commercial geolocation databases that are considered state of the art. They combine multiple data sources such as IP WHOIS information, network delay measurements triangulation through ping and traceroute, network topology analysis from BGP routing table dumps, reverse DNS hostnames, and others [1]. However, the exact methods they use to compile these databases are proprietary. Prior research has shown that these databases can sometimes be inaccurate or incomplete [2]. Therefore, it is vital for these types of services to use complete and accurate geolocation databases.

In this work we use reverse DNS hostnames assigned to IP addresses as a source of geolocation information. Figure 1 shows the type of location information that can be derived from a hostname. In this example we can determine that the IP address behind this reverse DNS hostname may be located in *Yokohama*, a city in Japan. Our task is to determine the location of IP addresses based on their reverse DNS hostnames, at city-level granularity. This task poses several challenges. First, naming schemes used by ISPs are heterogeneous and often contain abbreviations or ambiguous names. If a hostname contains the substring `mant`, does that refer to ***Mant**eo, North Carolina*, or to ***Mant**orville, Minnesota*? Or if it contains the term `snbr`, does that mean the location is ***San Br**uno, CA*, or ***San Ber**nardino, CA*, or ***San Bar**tolomeo, Italy*, or any of the other tens of locations around the world that could match this abbreviation? These types of ambiguous names are often difficult for humans as well. Second, reverse DNS hostnames sometimes contain strings that are not directly derived from the names of locations. For example, Verizon uses `nyc`**`m`**`ny` to refer to locations in *New York*, and GTE uses `mia`**`s`**`fl` to refer to *Miami, Florida*. Third, extracting locations from reverse DNS hostnames requires a scalable solution. There are more than 1.24 billion valid reverse DNS hostnames for IPv4 addresses alone. For each of these hostnames we have to evaluate and disambiguate potentially tens of candidate locations. Our proposed solution generates on average 48 potential location candidates for each hostname, which yields **close to 60 billion classifier decisions** in total.

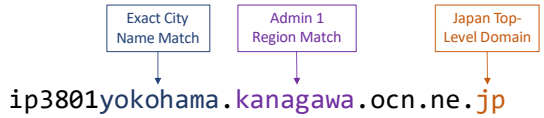

Fig. 1: Example information extracted from reverse DNS

To solve these problems, here we propose a distributed algorithm for extracting locations from reverse DNS hostnames. More specifically, our contributions are:

1) **We present a distributed approach to reverse DNS geolocation**. We build on our ongoing single-node reverse DNS geolocation work [3] and present a scalable distributed version. Our approach efficiently distributes computation steps among a cluster of machines.
2) **We evaluate both versions of the algorithm using the largest ground truth set reported in geolocation literature**. First, we show that our method significantly outperforms a state of the art academic baseline, and it is complementary and competitive with two commercial database baselines. Second, we perform **large-scale experiments on a cluster of 2,000 machines** to test the distributed version of our algorithm.

*Author is also affiliated with Microsoft Bing.

II. Related Work

The majority of IP geolocation research relies on active **network delay measurements** to locate addresses. These approaches issue pings or traceroutes from multiple geographically distributed landmark servers to a target IP, then triangulate the location [4], [5], [6], [7]. Network delay and topology methods have significant limitations. First, they have scalability issues as they require access to landmark servers distributed around the world and each target IP needs separate measurements. Second, not all networks allow ping and traceroute. Third, routes on the Internet do not necessarily map to geographic distances. These problems often lead to lackluster results, with error distance in the order of hundreds of kilometers. Some of the earlier research is also plagued by extremely small ground truth datasets, often focusing on a handful of IP addresses in a few US universities [4], [5].

Our work addresses several of these limitations. First, using reverse DNS hostnames for geolocation does not require any network delay measurements. Second, our ground truth dataset is several orders of magnitude larger than the ones used in previous work and it spans the entire planet. Third, our approach can be performed offline.

In this work we propose extracting IP locations from their reverse DNS hostnames. *Undns* is the most well-known and widely used reverse DNS geolocation approach [8]. It consists of manual rules which are expressed as regular expressions at the domain level. The disadvantage of this approach is that each domain requires manually generated and potentially error prone rules. Our approach is more scalable since it does not require human input. It also handles unique situations better, since it considers the terms of each hostname individually, without requiring domain-specific training.

DRoP, another state of the art reverse DNS based approach, aims to geolocate hostnames using automatically generated rules generated by finding patterns across all the hostname terms of a domain [9]. For example, it may find that for the domain `cogentco.com`, the second term from the right often contains airport codes. These rules are then validated using network delay information. *DRoP* places 99% of IPs in 5 test domains within 40 km of their actual location. However, it uses network delay measurements and its method of splitting hostnames is rudimentary.

Finally *DDec* [10] combines *undns* and *DRoP* rules by giving precedence to *undns* and using *DRoP* as fallback.

III. Approach

We present two versions of our approach. First, **we summarize our single machine version** where given a hostname, we output a ranked list of potential location candidates [3]. Second, we propose a distributed version where the computation steps are performed in parallel on a cluster of machines. This allows our algorithm to efficiently cover the entire IP space.

A. Datasets

Our ground truth set contains 67 million IP addresses with known geographic location. We compiled the ground truth set in March 2018 by randomly sampling the query logs of a large-scale commercial search engine.

In addition to the ground truth set, we use multiple publicly available datasets for feature generation, some of which we summarize here. **GeoNames** is a free database with geographical information [11]. The **CLLI** dataset contains codes used by the North American telecommunications industry to designate names of locations and functions of telecommunications equipment. **UN/LOCODE** is a worldwide geographic coding scheme developed and maintained by the UN. It assigns codes to locations used in trade and transport, such as rail yards, sea ports, and airports. **Rapid7 Reverse DNS** consists of reverse DNS hostnames across the entire IPv4 address space.

B. Single node classification

We begin by splitting hostnames into their constituent terms. Next, we iterate over each extracted term to find potential location candidates, and we also compute the primary and secondary features discussed below. We define the list of location candidates as the union of all locations which match any of the primary features. For example, since the term `hrbg` matches the location *Harrisburg, PA* using our *Abbreviations* feature, we select this city among the initial location candidates.

For each given hostname, we generate primary feature categories, which are based on city names, abbreviations, concatenations of city names and administrative regions, CLLI codes, UN/LOCODE codes, etc. Secondary feature categories are then generated in the context of a hostname and location candidate pair and their aim is to support the current candidate. For example, if a hostname contains the term `allentown`, the candidates found in the first phase include *Allentown, PA* and *Allentown, WA*. Secondary features are then computed to help in disambiguating locations. If the hostname also contains the term `wa` then during the evaluation of the *Allentown, WA* candidate, the secondary feature *Admin1* matches, which increases the confidence of this candidate. **For a detailed list of features, please consult our paper draft which describes the single-node approach in detail [3].**

We train a binary classifier where the input is a set of features describing a hostname and location candidate pair, and the output indicates whether the hostname is likely to be located in the candidate location. For this paper we have chosen to train the classifier using C4.5 decision trees. While this classifier might not yield the best overall results, it is extremely efficient, which is suitable for our goal of parsing all 1.24 billion reverse DNS hostnames in the *Rapid7* dataset.

C. Distributed classification

Not all IP addresses have a corresponding reverse DNS hostname and not all valid reverse DNS hostnames contain location information. The *Rapid7 Reverse DNS* dataset contains 1.24 billion valid hostnames, which covers 33% of the IPv4 space. While this only accounts for a third of IP addresses, this is still a very large number of hostnames. Furthermore, each hostname can generate tens of location candidates, and each of those candidates need to be classified and ranked to

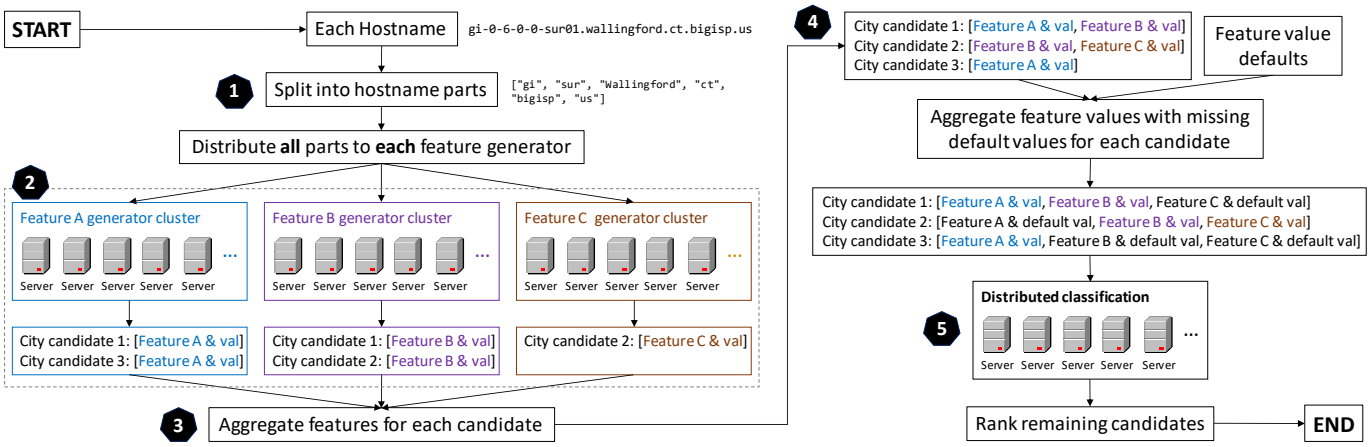

Fig. 2: Feature generation, feature aggregation, and classification each run distributed on a cluster of machines

find the most likely location. As we will show in Section IV, this results in run times on the order of tens of days to cover all reverse DNS hostnames on a single machine. We therefore propose a distributed version of our approach used for testing. We still perform the training phase on a single machine, since it requires a relatively small amount of training data.

We implemented our algorithm using the *SCOPE* distributed programming language [12]. A newer version called *U-SQL* is available to researchers using the *Azure* cloud. It combines declarative SQL-like statements with modules written in C#. The language is essentially a superset of the *MapReduce* paradigm and it has primitives that expand on the concept of *Map* and *Reduce* stages. For the evaluation section we used a version of *SCOPE* which allows precise allocation of resources. For storage we used the *COSMOS* distributed filesystem [12]. The public equivalent is called *Azure Data Lake*. The filesystem allows storing extremely large files, in the order of tens of terabytes or more. During processing, *SCOPE* reads and writes data to the *COSMOS* file system. In this work we treat the files as tables, where each data row corresponds to a row in a table.

Figure 2 describes the steps taken by the distributed algorithm. The first step is to split each hostname into its constituent terms. We achieve this using a *PROCESSOR*, which combines the concepts of *Map* and *Reduce* into a single statement. This primitive partitions data into multiple smaller batches. The batches are each processed on the available machines, until all pending batches are depleted. In the final stage the output from all machines is combined into a single output file, which is equivalent to a *reduce* phase that in this case just aggregates and concatenates the outputs from all machines. Each batch is processed by a C# module which receives input rows in sequence. The module processes each input row and can emit zero, one, or more outputs rows. We packaged the hostname splitter into such a module, where the input is a hostname, and the output is a data structure which contains the list of subdomain terms, as well as the public suffix and the domain top-level domain. If the row input does not contain a valid hostname, we return zero results. If however the input is valid, we return one row containing the original input and in addition we add a new column with the serialized hostname splitter output.

In the second step we find location candidates and we compute both primary and secondary features. Here we again we use a *PROCESSOR* to partition the dataset into smaller chunks which get distributed to available machines. For each primary feature and hostname input, we find the location candidates and we generate the value for the feature. We then also compute secondary features in the context of each hostname and location candidate pair. To make more efficient use of memory, we generate each primary feature category independently of the others. This allows us to use the entire cluster to compute each primary feature one by one separately. It also allows using multiple clusters in parallel, each generating a different feature. In the Evaluation section we used the same cluster to compute all features separately and in sequence. For each hostname input, this step outputs one feature value for each location candidate. For example, when processing the *First Letters* feature for the hostname `97-88-57-240.dhcp.roch.mn.charter.com`, this module outputs one row for each potential location candidate. So for our example it outputs one row for the ***Roch****ester, Minnesota* candidate, another one for the ***Roch****ester, New York* candidate, and so on. In this step for a given feature we read the datasets summarized in Section III-A into memory. We perform this pre-initialization step per data batch, before we read any input rows. During the Evaluation section we speculate that this extra step can contribute to the startup overhead. In a single-node implementation we need to read this data once, while in a distributed scenario we read the data multiple times per machine, once for each data batch.

After the partial results for each feature are generated, in the third step we *UNION* the results from each feature together. This operator is similar to the *UNION* operator in SQL in that it allows concatenating results from multiple ”tables”, in our case one for each primary feature category.

The fourth step aggregates individual features currently stored in individual rows back into a combined feature vector for each hostname and location candidate pair. To achieve this we use a *Reducer*, which aggregates rows on certain dimensions. In our case we reduce on the combination of hostname and location candidate. For each distinct hostname and candidate pair, the output contains all the aggregated features. The example in Figure 2 shows that in the partial results for *City Candidate 1* of a particular hostname both features *A* and *B* matched. After aggregation, these two partial feature vectors are combined into a single feature vector containing both features *A* and *B*. This *reducer* also has the role of filling in missing feature defaults. In Figure 2, *City Candidate 1* matches only two feature categories, while our entire feature set is larger. To obtain a complete feature vector array we fill in the values for the missing features with defaults. The result is a full feature vector array which is ready to be fed to the pre-trained classifier.

Finally, in the last step we run the binary classifier on the feature vectors. We package the classifier into a C# module and we apply it to the input feature vectors using the *PROCESSOR* primitive. In the interest of fairness, in both the single machine version and the distributed version we used the same classifier pre-trained on a single machine. The location candidates of each hostname are evaluated individually, so the input is a hostname and location candidate pair, along with the feature vector, and the output is the binary classification result. In essence the classifier decides for each hostname which of its candidates are not suitable. We finally rank the remaining candidates by classifier confidence, and we break ties by using the population of cities as a proxy for popularity.

IV. Evaluation

We first evaluate our approach against a state of the art academic baseline and two commercial geolocation databases. We show that our method significantly outperforms the academic baseline and it is complementary and competitive to commercial databases. We then perform multiple scalability experiments on the distributed version of our algorithm. We compare the run times of the single node and multi node approaches. We also study how the run time of the distributed version changes as we increase the number of machines used.

A. Single-node evaluation

Our ground truth dataset contains 67 million IP addresses with known IP location, of which we used 40 million for training and 27 million for testing. We first evaluate our approach *RDNS-DT* against a state-of-the-art academic baseline called *DNS Decoded*, or *DDec* for short [10]. Similar to our method, this baseline receives a hostname as input and attempts to extract its location. *DDec* is based on two other approaches: *undns* [8] and *DRoP* [9].

The authors of *DDec* have not open sourced their code and this baseline is only available by querying a web endpoint. Out of politeness, we restricted the number of requests we made to a more manageable size. We selected eight domains that are covered by the baseline. From our testing set we selected all data points that intersected these eight domains. This resulted in a subset of 1.6 million hostnames. We then issued requests to the *DDec* endpoint for each of these hostnames. For each result, we selected the location from the top *DDec* rule that matched the hostname.

TABLE I: Comparison between *DDec*, a state of the art academic baseline, and *RDNS-DT*, which is our approach that uses C4.5 decision tree classification

Metric		**Median Error in km**		**Coverage**	
Domain	**#**	**DDec**	**RDNS-DT**	**DDec**	**RDNS-DT**
163data.com.cn	166K	1,517.5	**10.6**	**100%**	94.5%
bell.ca	200K	5,875.2	**5.8**	2.3%	**95.2%**
brasiltelecom.net.br	32K	808.7	**15.8**	**100%**	76.6%
charter.com	580K	**60.8**	88	78.0%	**89.0%**
frontiernet.net	67K	36.5	**27.7**	3.6%	**99.2%**
nttpc.ne.jp	0.9K	16.2	**9.1**	16.2%	**57.6%**
optusnet.com.au	100K	704.4	**15.7**	**100%**	98.9%
qwest.net	408K	8,038.7	**20.2**	4.1%	**97.6%**
Overall	1.6M	177.9	**42.4**	49.7%	**93.3%**

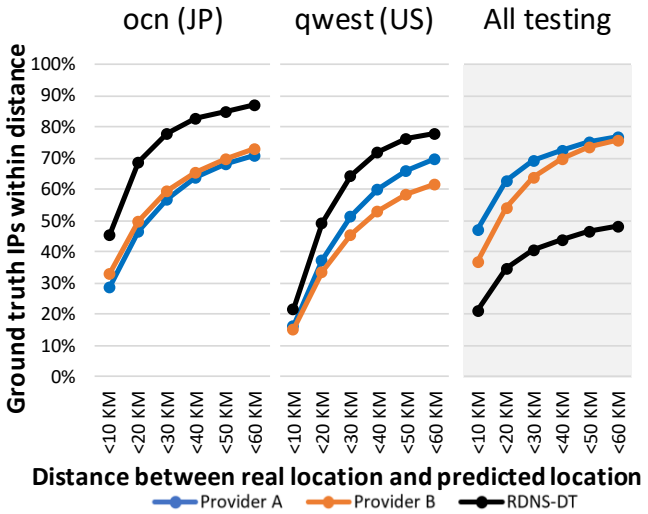

Fig. 3: Commercial Evaluation Error Distance

After running our single-node reverse DNS geolocation method over the same testing subset, we calculated the median error distance and coverage. We define error distance as the distance between an extracted location for a hostname, and the real geographical location of the underlying IP address for the hostname. We define coverage as the number of hostnames where a geolocation method made a decision, over the total number of hostnames in the testing subset.

Table I displays the comparison between our method and the academic baseline. A lower median error signifies better results. Overall our approach significantly outperforms the academic baseline. In the single case where the baseline had better median error distance, our approach had higher coverage. And whenever the baseline had higher coverage for a domain, our median error distance was significantly better.

Although reverse DNS geolocation on its own cannot match

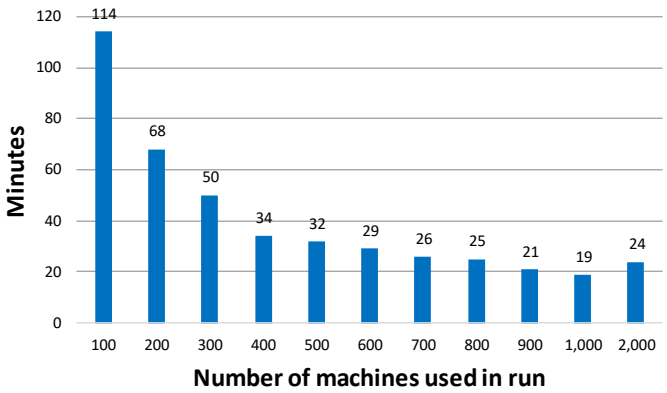

Fig. 4: Wall clock run time in minutes for the City Name feature when varying number of machines

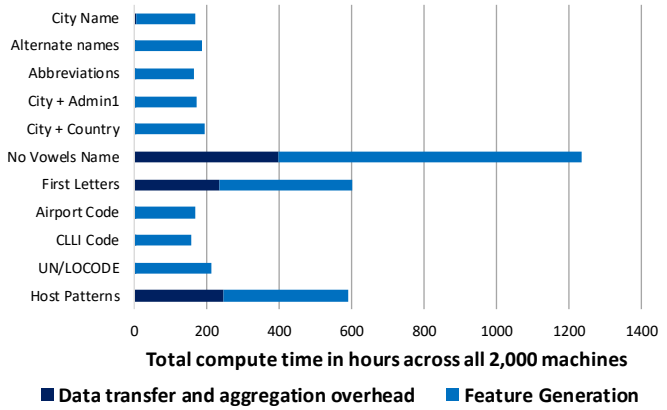

Fig. 5: Compute time per feature using distributed version

commercial databases, we evaluate our approach to show that it can complement and potentially improve existing databases. We tested our approach against two state of the art commercial databases using our entire test dataset of 27 million hostnames. Figure 3 displays cumulative error distance in kilometers. The X axis represents the maximum distance between the real location and the predicted location. The Y axis shows how many hostnames and their IP addresses fall within the error distance. For instance, the *<20 km* column in the first graph shows that our method, labeled *RDNS-DT*, places approximately 68.9% of hostnames in the ground truth set within 20 kilometers of their actual location. The first two graphs in the figure show that on certain domains our approach outperforms, and thus could be used to improve commercial databases. However, as expected, the third graph shows that overall the commercial databases still outperform our method. Results show that median error is 11.1, 16.7, and 75.8 kilometers for *Provider A*, and *Provider B*, and our approach *RDNS-DT*, respectively.

V. Distributed Evaluation

The purpose of this evaluation section is not to determine the accuracy of the algorithm, but to determine how well the distributed version of the algorithm scales. We use a version of *SCOPE* that can be customized to precisely allocate resources. Although the machines we use are multi-core, we ran the algorithm on a single core on each machine. The reason we made this decision was to obtain a more fair comparison to our single-node version, which is also single threaded. When eventually deployed in production, it is therefore likely our distributed algorithm will run faster than presented here, yielding an even larger difference to the single-node version. The machines are each connected to a 10 Gbps network. We used similar hardware configurations when running both the single-node and distributed versions.

We begin by evaluating how well our approach scales for generating individual features when we vary the number of machines in the cluster. This corresponds to step 2 in Figure 2. Due to space constraints, for this first experiment we only present the results for generating the *city name match* feature, but we have verified the the algorithm scales similarly for the other features as well. We compute the feature on the entire set of 1.24 billion hostnames multiple times, in each run varying the number of machines. We generated the feature in increments of 100 machines up to 1,000 machines. We also ran the feature on the full cluster of 2,000 machines.

Figure 4 shows how well our feature generation step scales when we increase the number of machines. We make the following observations. First, as we increase the number of machines gradually from 100 to 400, we observe a significant reduction in run time. For this feature the wall clock run time reduces from almost two hours to about half an hour. Second, as we continue increasing the number of machines to 500 and beyond, we see diminishing returns. The run time slowly tapers from 32 minutes down to about 19 minutes when we reach 1,000 machines. Third, we interestingly observe that when we run the algorithm on 2,000 machines, the run time actually increases to 24 minutes.

There are multiple potential reasons for obtaining diminishing returns when we use more than 500 machines. One possibility is that using a large number of machines forces data transfer when accessing the *COSMOS* distributed filesystem. *SCOPE* and *COSMOS* generally try to schedule work close to or on the machines that physically store the input data. It is possible that using these many machines forces the input data to be transferred across machines or across racks. Another potential explanation is that, as we mentioned in Section III-C, the pre-initialization stage where the nodes read each dataset into memory could have an impact on runtime. Finally, based on our preliminary investigation the most likely explanation for the changes in run time is that using a large number of machines runs into the risk of the task being slowed down accidentally by unhealthy nodes. We ran the experiment multiple times on all 2,000 machines and we noted the status of the machines during each run. We found that most machines completed each batch within minutes, but invariably some machines took longer or failed completely. In the first case the machine might be thermally throttled or it may have randomly received a batch that just naturally took longer to process. In the second case, *SCOPE* has a feature where it monitors each

batch and if the machine fails to process it in a certain amount of time, it reschedules the task on a different machine. This results in a few laggard machines that impede the progress of the overall task and increase run time.

Next, we measure the total compute time for each primary feature. While we could generate all 11 primary features at the same time on different clusters, in this paper we generate them in sequence using the same cluster, to make sure we are measuring using the same underlying hardware. We split compute time in two categories. The first category represents data transfer and data aggregation overhead. It measures how much compute time was used for transferring or aggregating data. The second category measures compute time used to actually generate the features. In both cases we obtained the numbers by summing compute time across all 2,000 machines in the cluster. Figure 5 shows total time in hours for each feature run. We observe that for the majority of the features compute time is mostly used on the feature generation task itself. However, three of the features report excessive overhead. Upon investigating we determined that these features generate the most location candidates per input hostname, and thus the most output data that needs to be aggregated, transferred, and stored.

Take for instance the *No Vowels Name* feature. The reason why this feature generates so many candidates is that it can match short combinations of letters based on the names of cities. This feature allows partial matches using the first 3 or more letters of the names. For example, it matches `gnvl` to *Greenville, SC* and `rvrs` to *Riverside, CA*. Furthermore, we extended this feature with more complex variations. We select the first and last letters of each word in the name, even if the letters are vowels. We then generate combinations of letters from this list, in order. Examples matched by this variation include `oxfr` for *Oxford, MA*, and `ftmy` for *Fort Meyers, FL*. The increased complexity of this feature is reflected in high compute hours both for data overhead and for feature generation. The other two features that exhibit this same pattern have similar characteristics.

On average our algorithm generates 48 location candidates per hostname. Since the *Rapid7* dataset contains 1.24 billion hostnames, it means the classifier needs to evaluate close to 60 billion data points. Our final experiment compares the run times of the single-node and distributed variants across all 1.24 billion hostnames. In both cases we measure the full run time of the algorithms. One of the differences between the variants is that the single machine version does not need to aggregate features because it fills the final feature vector in local memory. Since the computation is distributed over multiple machines, the distributed version requires transferring and combining the results into the final vector before the classification step. Table II shows the stark difference between total run times and it highlights the need to run this process in a distributed manner. The run time decreases from a whopping 75.5 days on a single machine, to only half a day on the distributed cluster. The result is a speedup of 156X for the distributed version. Looking at the detailed results we observe that, interestingly, the classification time between the algorithms is not so different. The reason is that a decision tree can be compiled into an `if` statement which runs very fast.

TABLE II: Wall clock run time for single machine compared to distributed version using 2,000 machines - **156X speedup** from 75.5 days down to 0.48 days

	Single	**Distributed**
Candidate and feature generation	1805.4 hours	6.4 hours
Feature aggregation	Not required	2.8 hours
C4.5 decision tree classification	6.8 hours	2.4 hours
Total	**1812.2 hours**	**11.6 hours**

VI. Conclusions and Future Work

We presented a distributed machine learning approach to geolocating reverse DNS hostnames. Our method significantly outperforms a state-of-the-art academic baseline and it is competitive and complementary with commercial baselines. We have also shown that the distributed version of our algorithm has a 156X speedup over our single-node version.

One direction for future work would be to focus on combining reverse DNS hostname information with WHOIS databases and network delay to form a geolocation database across the entire IP space.

References

[1] J. A. Muir and P. C. Van Oorschot, "Internet geolocation: Evasion and counterevasion," *ACM Computing Surveys (CSUR)*, vol. 42, no. 1, p. 4, 2009.

[2] M. Gharaibeh, A. Shah, B. Huffaker, H. Zhang, R. Ensafi, and C. Papadopoulos, "A look at router geolocation in public and commercial databases," in *Proceedings of the 2017 Internet Measurement Conference*. ACM, 2017, pp. 463–469.

[3] O. Dan, V. Parikh, and B. D. Davison, "IP Geolocation through Reverse DNS," 2019, preprint. [Online]. Available: https://www.ovidiudan.com/papers/2019/reverse-dns/

[4] V. N. Padmanabhan and L. Subramanian, "An Investigation of Geographic Mapping Techniques for Internet Hosts," in *SIGCOMM 2001*. San Diego, California, USA: ACM, 2001, pp. 173–185. [Online]. Available: http://doi.acm.org/10.1145/383059.383073

[5] B. Gueye, A. Ziviani, M. Crovella, and S. Fdida, "Constraint-Based Geolocation of Internet Hosts," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1219–1232, Dec 2006.

[6] S. Laki, P. Mátray, P. Hága, I. Csabai, and G. Vattay, "A model based approach for improving router geolocation," *Computer Networks*, vol. 54, no. 9, pp. 1490–1501, 2010.

[7] G. Ciavarrini, M. S. Greco, and A. Vecchio, "Geolocation of internet hosts: Accuracy limits through cramér–rao lower bound," *Computer Networks*, vol. 135, pp. 70–80, 2018.

[8] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with Rocketfuel," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 4, pp. 133–145, 2002.

[9] B. Huffaker, M. Fomenkov, and k. claffy, "DRoP:DNS-based Router Positioning," *ACM SIGCOMM Computer Communication Review (CCR)*, vol. 44, no. 3, pp. 6–13, Jul 2014.

[10] Center for Applied Internet Data Analysis, "DDec - DNS Decoded - CAIDA's public DNS Decoding database," 2018, accessed: 2018-07-31. [Online]. Available: http://ddec.caida.org/help.pl

[11] M. Wick, "Geonames," 2018, accessed: 2018-06-27. [Online]. Available: http://download.geonames.org/export/dump/

[12] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1265–1276, 2008.