Lessons Learned from Developing and Deploying a Large-Scale Employer Name Normalization System for Online Recruitment

Qiaoling Liu, Josh Chao, Thomas Mahoney,
Alan Chern, Chris Min, Faizan Javed
CareerBuilder LLC
Norcross, GA
{qiaoling.liu,josh.chao,thomas.mahoney,alan.chern,chris.min,faizan.javed}@careerbuilder.com

Valentin Jijkoun
Textkernel BV
Amsterdam, The Netherlands
jijkoun@textkernel.nl

ABSTRACT

Employer name normalization, or linking employer names in job postings or resumes to entities in an employer knowledge base (KB), is important for many downstream applications in the online recruitment domain. Key challenges for employer name normalization include handling employer names from both job postings and resumes, leveraging the corresponding location and URL context, and handling name variations and duplicates in the KB. In this paper, we describe the CompanyDepot system developed at CareerBuilder, which uses machine learning techniques to address these challenges. We discuss the main challenges and share our lessons learned in deployment, maintenance, and utilization of the system over the past two years. We also share several examples of how the system has been used in applications at CareerBuilder to deliver value to end customers.

ACM Reference Format:
Qiaoling Liu, Josh Chao, Thomas Mahoney, Alan Chern, Chris Min, Faizan Javed and Valentin Jijkoun. 2018. Lessons Learned from Developing and Deploying a Large-Scale Employer Name Normalization System for Online Recruitment. In *KDD '18: The 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, August 19–23, 2018, London, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3219819.3219842

1 INTRODUCTION

Entity linking [29] links entity mentions in text to the corresponding entities in a knowledge base (KB) and has many applications such as information extraction and content analysis, in both open domain and specific domains. For example, in the online recruitment domain, linking employer names in job postings or resumes to entities in an employer KB, which is defined as the employer name normalization task in [20, 21], is important to many business applications such as labor market analytics, semantic search for talent sourcing, and job recommendations (among others).

An effective employer name normalization system should be able to:

(1) Handle employer names from both job postings and resumes, which are in semi-structured format and could come from different sources. Therefore, the employer name fields could contain varying degrees of noises.
(2) Leverage the location and URL context if available. Most job postings have an associated location, and some include a company URL. Meanwhile, many resumes have a location listed for each employment history entry. Such context can be helpful for normalizing employer names.
(3) Handle name variations. An employer entity can have legacy names, nicknames, and acronyms, which may be used in job postings and resumes, and thus different from the entity name in the KB.
(4) Handle duplicates in the KB. The employer KB may be derived from data sources with duplicate records referring to the same employer, e.g., different branches or divisions of a company.

In this paper, we describe the CompanyDepot [20, 21] system developed at CareerBuilder[1], which uses machine learning techniques to address the above challenges. The system takes an employer name and its associated location and URL from job postings or resumes as input and normalizes it into entities or clusters in an employer KB. We apply query expansion based on external mappings to address the name variation challenge, and use clustering to address the duplicate entity challenge.

Next, we discuss the main challenges in deploying and using the system at CareerBuilder such as how to quickly scale up additional servers in response to spiky request volume, and how to solve scalability issues when using it in data-intensive applications. We share our lessons learned in deployment, maintenance, and utilization of the system in the past two years. In particular, we provide details on the optimizations made to the algorithms as well as the deployment and operations pipeline that lead to significant improvements in performance and cost savings. We also discuss the types of errors found in results, how we correct them, and how we use machine learning methods to automatically detect the errors on a large scale.

Compared to the previous work in [20, 21], the main contributions of this paper are as follows:

- Using machine learning to compute cluster representatives instead of using heuristics (Section 4.1.5 and Section 4.3);
- Service deployment and maintenance (Section 5);
- Applications at CareerBuilder (Section 6);
- Error handling and automatic error detection (Section 7).

[1] https://www.careerbuilder.com/

KDD '18, August 19–23, 2018, London, United Kingdom

ACM ISBN 978-1-4503-5552-0/18/08…$15.00
https://doi.org/10.1145/3219819.3219842

The rest of the paper is organized as follows. Section 2 discusses the related work and Section 3 defines the problem. Next, we describe the CompanyDepot system in Section 4, including the improvement we made in this paper and corresponding evaluation. We then discuss the deployment challenges and share our lessons learned in Section 5, and in Section 6 we show how the system has been used in CareerBuilder. Section 7 describes how we detect errors and handle them. Finally, we conclude the paper in Section 8.

2 RELATED WORK

As we solve the employer name normalization task, our work is related to the work on entity linking, domain-specific name normalization, as well as deduplication and clustering methods. We discuss each of them in the following.

2.1 Entity Linking with a Knowledge Base

Entity linking [29], also called named entity disambiguation (NED) or named entity normalization (NEN), has attracted much research effort since the availability of large KBs such as Wikipedia. It is a key step to understand and annotate the raw and noisy data in many applications. A comprehensive survey of the issues and methods about entity linking is provided in [29]. As summarized by the survey paper, a typical entity linking system has three modules: candidate entity generation, candidate entity ranking, and unlinkable mention prediction. Recently, methods based on deep neural networks and entity embedding learning have been proposed and show promising improvement for the entity linking task [10, 27].

The employer name normalization task discussed in this paper can be viewed as a general entity linking problem, yet it differs from the traditional entity linking task in three aspects [20, 21]: (1) different input data sources; (2) different contexts; (3) different KBs. Therefore, the employer name normalization task has unique challenges such as handling the location and the URL context associated with the employer names in jobs and resumes, as well as handling duplicate entities in the KB. The system presented in this paper adapts the three-module framework used in the typical entity linking systems. We use cluster-level normalization to handle duplicate results, which is not considered in entity linking systems.

2.2 Domain-Specific Name Normalization

Our work is also related to a set of domain-specific name normalization applications. For example, within the same recruitment domain, Yan et al. described how to normalize the company name on a LinkedIn member's profile position using social graphs based on binary classification [32]. The problem of academic institution name normalization discussed by Jacob et al. [11] is very similar to our task. Instead of an employer KB used in our work, they used a KB of academic institutions. Experiments in [21] showed that CompanyDepot performed better than the sCooL system [11] on five academic institution datasets. NEMO [14] addressed a related task of extracting and normalizing organization names from PubMed articles. The method first uses multi-layered rule matching to extract entity mentions, and then leverages unsupervised clustering to identify entity mentions referring to the same entity, which has a time complexity of $O(k \times N)$, where k is the number of clusters, and N is the number of all entity mentions.

There are also many other domain-specific name normalization applications, e.g., product item normalization [3], skill normalization [13], gene name normalization [30], disease name normalization [19], and person name normalization [22]. The main differences of these applications with ours lie in different input data sources, contexts, and KBs, which often bring some different challenges.

2.3 Deduplicating Domain-Specific KBs and Clustering Methods

The task of employer name normalization depends on an employer KB, and a key step in building such domain-specific KBs is deduplication [17]. Kardes et al. proposed graph-based blocking and clustering strategies for organization entity resolution [16]. McNeill et al. proposed a dynamic blocking method to efficiently deduplicate around 5 billion people records [23]. The book by Christen summarizes more methods for general duplicate detection [7].

Since our customers want to group not only duplicate employer entities but also sometimes entities that have corporate linkages (e.g., branches, divisions, and subsidiaries), we resort to clustering approaches to handle duplicate entities in the employer KB.

There are many types of clustering methods available in literature [12, 31], such as clustering based on partition [25, 26], hierarchy [33], density [8, 15], and graph theory [1, 16]. Different methods have different constraints and are suitable for different applications. Some methods depend on designing a feature vector for each data item to calculate the distance between two items [8, 25, 26], or a feature vector for each cluster of items to find the distance between two clusters [33]. Some methods require to prefix the number of clusters [25, 26], or select a suitable threshold for clustering [8, 15].

Considering the large size of our KB (about 19 million entities) and the number of clusters (k) is unknown, we chose an efficient graph-based clustering method for our task of cluster-level normalization. This also allows us to easily use the external mapping sources to create edges between entities for clustering and avoid manually prefixing k as required by many other clustering methods.

3 PROBLEM SETTING

Before defining the problem, we introduce some notations. Let E denote the entities in the employer KB. The employer names and the associated location contexts and URL contexts extracted from job postings and resumes are denoted by $Q = \{q_1, q_2, ..., q_t\}$, where $q_i = (n_i, l_i, u_i)$ is a triple of employer name, the associated location, and the associated URL. We call n_i the query name, l_i the query location and u_i the query URL. Note that l_i and u_i could be empty.

Next, we introduce a clustering function $C(e) \Rightarrow r$, where $e \in E$, $r \in R \subset E$, R is the set of all cluster representative entities (each cluster representative entity corresponds to a cluster), and r is the representative entity of the cluster that e belongs to.

We now define the employer name normalization task with two levels of complexity:

- Normalization at entity level is to infer a mapping function $f_E(q) \Rightarrow e$, where $q \in Q$ and $e \in E \cup \{NIL\}$, which maps a query to an entity in the KB that best matches the query or Not-In-Lexicon.
- Normalization at cluster level is to infer a mapping function $f_C(q) \Rightarrow r$, where $q \in Q$ and $r \in R \cup \{NIL\}$, which maps a

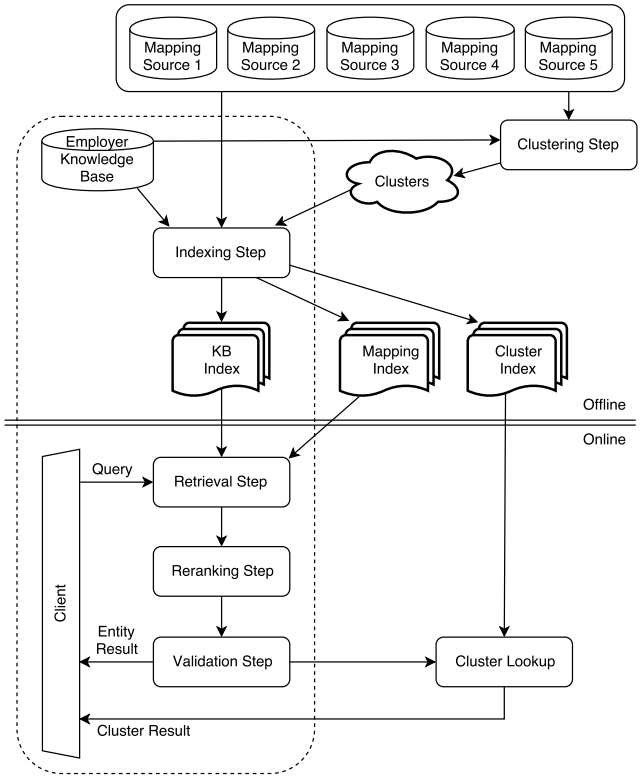

Figure 1: Architecture of the CompanyDepot system.

query to a cluster in the KB that best matches the query or Not-In-Lexicon.

On top of the above task, the applications at CareerBuilder posed a few extra requests to the normalization system: (1) high accuracy of entity attributes returned, e.g., industry and company size; (2) short query response time; (3) errors reported by users can be quickly fixed.

4 SYSTEM OVERVIEW

In this section, we briefly describe the CompanyDepot system for addressing the above normalization task (originally proposed in [20, 21]), the improvement we made in this paper (Section 4.1.5), and corresponding evaluation (Section 4.3).

The architecture of CompanyDepot is shown in Figure 1. Before taking any queries, the system needs to build several indexes offline. Once the indexes are ready, the system can take normalization requests. Each request consists of an employer name, its location context and URL context (both contexts could be empty). The system then performs normalization at both entity and cluster level, and returns both an entity result and a cluster result.

4.1 Offline Processing

4.1.1 Employer Knowledge Base. The employer KB used in our system is built using a third-party employer database, which contains about 26 million employer records [20]. Each record represents an employer with a set of attributes like business name, location, URL, industry code, company size, and parent company. Branches and subsidiaries of a company are represented as different records, often with the same business name. The database has a good coverage of employers in US, yet it is noisy and may have duplicate records for the same employer. For example, it has different records named "Enterprise Rent A Car", "Enterprise Rentacar", and "Enterprise Rent-A-Car Company" respectively. Based on the above employer database, we merged all the records with the same business name into a single entity (by assuming that two records with the same business name refer to the same employer) and kept a list of all the original locations. This resulted in a set of around 19 million employer entities that serves as our final employer KB. Note that the noises still exist in the KB, which needs to be handled by the employer name normalization system.

Table 1: Statistics and examples for mapping sources.

Source	Size	Example
Wikipedia	135K	IBM Corp. → International Business Machines Corporation
Stock	6K	MSFT → Microsoft Corporation
Hierarchy	272K	Amazon Web Services, Inc. → Amazon.com, Inc.
Legacy	26M	bankofamerica → Bank of America Corporation
Provider	10M	pricewaterhouse coopers → PwC

4.1.2 Mapping Sources. Five data sources are used in both our entity-level normalization (to do query expansion) and cluster-level normalization (to do graph-based clustering). Each source contains a set of mappings from surface forms to normalized forms. Table 1 shows the statistics and examples for each source. Details about how each mapping source is obtained can be found in [20]. Note that the mapping sources are not required to have any dependencies on the KB described in Section 4.1.1, as long as they are about employer entities and have a reasonable quality. Therefore, more mapping sources can be added in future.

4.1.3 Name Calibration. Considering that a query name (or a surface form) is often short while the official name (or normalized form) of an entity could be verbose, we calibrate them to enable more accurate matching. The calibration of an employer name works as follows: 1) Convert the name to lowercase, and replace "'s" with "s" (e.g., "Macy's" → "macys"); 2) Convert all the non-alphanumeric characters to space; 3) Remove stop-phrases (e.g., "pvt ltd" and "l l c") and stop-words (e.g., "inc", "corporation", and "the"); 4) Expand commonly used abbreviations, e.g., "ctr" → "center", "svc" → "services", "dept" → "department"; 5) remove all spaces in the name.

4.1.4 Clustering Step - Creating Clusters. This step computes the clustering function $C(e)$. Considering that the number of entities N in the KB is large (about 19M) and the number of output clusters k is unknown which can be also at million scale, we need a scalable and fast clustering algorithm in terms of both N and k. So we use an efficient graph-based clustering method as follows: (1) Create an undirected graph based on the mapping sources described in Section 4.1.2. For each mapping from a surface form to a normalized form, we add their calibrated forms as nodes in the graph and create an edge between them. We use calibration to avoid creating a sparse graph. (2) Remove low-quality edges that only appear in a

single mapping source. So each remaining edge has been verified by at least two mapping sources. We can increase this threshold to increase the quality of edges when more mapping sources are added. (3) Apply transitive closure to find all the connected components in the graph [18, 28]. Each connected component corresponds to a cluster. (4) Add each entity from the KB that is not included in the graph as a separate cluster. As the mapping sources we used to create the graph include both corporate linkages (from the hierarchy mapping source) and surface forms (from the other mapping sources), each cluster is a group of duplicate entities and linked entities by corporate linkage.

4.1.5 Clustering Step - Selecting Cluster Representative Entity. After creating the clusters, we need to select the most appropriate entity to represent each cluster. Ideally the cluster representative entity should have correct employer name and other attributes in the KB (e.g., industry, company size, website, headquarter location), which are important to our applications as described in Section 6.

In this paper we develop a machine learning method for this task, which is an improvement of the heuristic method used in [20]. We derive the features from each entity's name and attributes in the KB, e.g., number of branches, number of surface forms available from mapping sources, length of name, whether name has a legal word such as "inc" or "llc", company size, whether a valid naics code is available, whether website is available, whether headquarter location is available. Based on these features, we use a listwise learning-to-rank method, coordinate ascent [24], to rank the entities in the cluster so that the top ranked entity is chosen as the cluster representative. Coordinate ascent can directly optimize any user specified ranking measure, by updating one parameter at a time while holding other parameters fixed. It has been shown to provide superior performance compared to some other learning-to-rank models [4]. Specifically, we use the coordinate ascent implementation provided in the RankLib[2] library for our experiments. Since only the top-ranked entity matters in our task, we choose P@1 to optimize on the training data. Our preliminary experiments show that it performs the best among other available rankers in RankLib for this task. This learning-to-rank algorithm is also used in the reranking step described in Section 4.2.2 for the same reason.

4.1.6 Indexing Step. In this step, we build three Lucene[3] indexes for the KB, mappings, and clusters, respectively, so that online normalizations can be done efficiently. Table 2 shows the index structure for the three indexes. The KB index is used to support efficient entity search based on the name or URL in input query for normalization. Considering that URLs are often too sparse and noisy, we extract the domains of URLs to do domain match instead of direct URL match between query and entity. The mapping index is used to support query expansion based on the mapping sources. It enables efficient retrieval of the alternative names in the normalized form field for a query name using the surface form field, which can then be used for query expansion. The cluster index is used to support efficient look up of the cluster of an entity. As described in Section 4.1.4, entities in a cluster are identified by their calibrated names, so we use the calibrated name as the key of an entity to locate its cluster representative in the cluster index.

[2]http://sourceforge.net/p/lemur/wiki/RankLib/
[3]https://lucene.apache.org/

Table 2: Index structure.

(a) A document in the KB index.

Field	Value
id	15
normalized_form	International Business Machines Corporation
calibrated_name	internationalbusinessmachines
domain	ibm.com
json	{"id": "15", "normalized_form": "International Business Machines Corporation", ...}

(b) A document in the mapping index.

Field	Value
surface_form	IBM
normalized_form	International Business Machines Corporation
mapping_source	wikipedia

(c) A document in the cluster index.

Field	Value
cluster_member_key	internationalbusinessmachines
cluster_representative	International Business Machines Corporation

4.2 Online Normalization

4.2.1 Retrieval Step. In this step, we first use Lucene's powerful querying capabilities to retrieve a large set of entities possibly relevant to the query, and then apply several filters to keep only the most likely correct results as the final pool of candidate entities.

Specifically, we first search for the query name (and query URL if available) in the KB index via an aggregated search in Lucene which combines (1) keyword searches in the normalized form field; (2) fuzzy searches in the normalized form and calibrated name fields; (3) phrase searches in the normalized form field; (4) keyword searches in the calibrated name field based on the calibrated version of each of the query expansion names; and (5) keyword search in the domain field based on the domain of the query URL.

After obtaining the top N_0 (e.g., N_0=1000) entities from the Lucene searcher, we then generate the pool of candidate entities as follows: (1) From the N_0 results, add to the pool the top N_1 (e.g., N_1=10) entities that have the highest Lucene score. (2) From the N_0 results, use Levenshtein Distance to compute the top N_2 (or N_3) results whose original (or calibrated) normalized form has the minimum distance from the original (or calibrated) query name, and add them to the pool. (3) From the N_0 results, add to the pool the N_4 (or N_5) results for which a mapping from original (or calibrated) query name to original (or calibrated) entity normalized form exists in any mapping source. (4) From the N_0 results, add to the pool the N_6 results for which the domain of the query URL matches that of the entity URL.

The resulting pool contains $N = \sum_{i=1}^{6} N_i$ candidate entities. We observe that N ranges between 10 and 20 empirically.

4.2.2 Reranking Step. In this step, we use the same listwise learning-to-rank method, coordinate ascent [24], as used in Section 4.1.5, to rerank the candidate employer entities obtained from the retrieval step. Since only the top-ranked entity matters in our task, we choose P@1 to optimize on the training data.

Table 3: Evaluation of cluster representative computation.

Dataset	Heuristics	Machine Learning
Jobfeed_rep	better for 24.0% of cases	**better for 44.0% of cases**
Edge_rep	better for 22.5% of cases	**better for 41.5% of cases**

For each (query, entity) pair, we generate a vector of features, which can be grouped into three categories: (1) **Query features** indicate the complexity of finding a correct result for this query, which include the length of the query name, whether it indicates irrelevant input, and whether the query location is specified. (2) **Query-entity features** indicate the likelihood that the entity is a correct result for the query, which include the score from the Lucene searcher, string comparison of the query name and the normalized form (or the surface forms) of the entity, matching between the query location and the location list of the entity, matching between the query URL and the entity URL, whether a mapping from original (or calibrated) query name to original (or calibrated) entity normalized form exists in each (or any) mapping source. (3) **Entity features** indicate the prior knowledge about the entity being a correct result for some query, which include the entity popularity, the number of locations of the entity, whether the normalized form of the entity contains a legal word such as "inc" or "llc". More detailed description of the features can be found in [20, 21].

4.2.3 Validation Step. We use a binary classifier to validate the top-ranked result from the reranking step so that either a correct result or NIL is sent to the user. The features used for in this step include all the features used in the reranking step as well as the score output of the learning-to-rank method. We use LibSVM [5] as our binary classifier.

4.2.4 Cluster Lookup. We compute the cluster-level normalization $f_C(q)$ based on the entity-level normalization $f_E(q)$, by $f_C(q) = C(f_E(q))$. The key to computing this is to apply the clustering function $C(e) \Rightarrow r$. Based on the cluster index, we can efficiently look up the cluster representative of an entity.

4.3 Evaluation

As described in Section 4.1.5, in this paper we use the machine learning (ML) approach to learn cluster representatives instead of using heuristics as in [20]. Therefore, we would like to compare the performance of the two approaches. We create two datasets from two applications at CareerBuilder: (1) Jobfeed: which collects online job postings for labor market analysis, and uses CompanyDepot to enrich job postings with additional information about the advertising company, including normalized company name, company industry, company size. (2) Recruitment Edge: which collects and deduplicates publicly available candidate profiles for candidate search, and uses CompanyDepot to normalize the company name attributes in each candidate profile and indexes the corresponding company industry and company size for faceted search. For more details about these two applications, please refer to Section 6. We first collect the top 10,000 most popular queries in each application during a month period, and then random sample 200 queries from the queries that result in different cluster representatives using the two methods. For each selected query, we compare the two cluster representatives by looking at the employer name and other attributes (including industry, company size, URL, and headquarter location) and label which representative is better. Table 3 shows the results. We can see that on both datasets the ML approach to computing cluster representatives outperforms the heuristic approach in returning more accurate attributes.

For overall normalization quality, the experiments in [20] showed CompanyDepot's entity-level normalization outperforms several systems previously used at CareerBuilder, and the cluster-level normalization further outperforms the entity-level normalization by providing similar success rate but much fewer diverse results.

In this paper, we add two more datasets by taking a random set of the linked queries from the top 10,000 most popular queries in our applications during a month period. Two queries are linked if a normalizer returns the same result for them. Table 5 shows the normalization quality of different systems. WService is a third-party web service that supports employer name normalization. CD-E (CD-C) means CompanyDepot's entity-level (cluster-level) normalization.

We use the same entity-level metrics (precision, coverage, f-score) and cluster-level metrics (success rate, diversity reduction ratio, f-score) as used in [20] to measure normalization quality. Table 4 lists the definition of these metrics. From the definition, we can see that the entity-level metrics (precision, coverage, f-score) do not check the diversity of the results and therefore they could not measure how well the system handles duplicates in the KB. On the other hand, the cluster-level metrics check the diversity of the results and thus they should be used preferably, than the entity-level metrics.

We can see that on both datasets CompanyDepot's cluster-level normalization achieved the best cluster-level f-score. It outperforms WService in all metrics, while providing a little worse success rate but much fewer diverse results than entity-level normalization. Therefore, the applications at CareerBuilder are advised to use the cluster-level normalization result.

5 DEPLOYMENT AND MAINTENANCE

In this section, we describe how we deploy and maintain the Company Normalization Service, the production web service at CareerBuilder which makes the CompanyDepot system available for use by other teams. The service is basically a thin REST API layer around the normalizer, deployed in the Amazon cloud. It runs in Docker containers[4] using Auto Scaling[5]. This means the number of server nodes handling requests grows and shrinks in response to demand. The index files are loaded from Amazon S3 on each server node at startup.

5.1 Deployment Challenges and Lessons Learned

The service auto scaling helps us to dynamically maintain an appropriate number of server nodes at all times, and yields a high level of stability for most traffic patterns. However, some users of the service exhibit particularly *jerky* or *spiky* traffic patterns, which is most often linked to a large Hadoop or Spark job making thousands or even millions of highly concurrent requests to the

[4]https://www.docker.com/
[5]https://aws.amazon.com/autoscaling/

Table 4: Definition of entity-level and cluster-level metrics as used in [20]. For a set of test queries, suppose a normalization system returns I_c correct results, I_w wrong results, and I_n null results (i.e., NIL). Let G be the unique items in the I_c correct results and $I_{c|g}$ be the number of correct results for each $g \in G$.

Level	Metric	Definition
Entity-level	Precision	The percentage of correct results out of all non-null results from the system. $Precision = I_c/(I_c + I_w)$
	Coverage	The percentage of queries that the system returns a non-null result. $Coverage = (I_c + I_w)/(I_c + I_w + I_n)$
	F-score	The harmonic average of precision and coverage.
Cluster-level	Success Rate (SR)	The percentage of correct results from the system for all test queries. $SR = I_c/(I_c + I_w + I_n)$
	Diversity Reduction Ratio (DRR)	How much result diversity the system reduces correctly via clustering. $\mathbf{DRR} = 1 - \frac{exp(\mathbf{H}) - 1}{I_c - 1}$, where $\mathbf{H} = -\sum_{g \in G} \frac{I_{c\|g}}{I_c} \cdot ln\left(\frac{I_{c\|g}}{I_c}\right)$
	F-score	The harmonic average of success rate and diversity reduction ratio.

Table 5: Evaluation of normalization quality.

(a) Entity-level normalization quality.

Dataset	Metric	WService	CD-E	CD-C
Jobfeed_norm (395 queries)	Precision	79.89%	88.43%	83.75%
	Coverage	100.00%	100.00%	100.00%
	F-score	88.82%	**93.86%**	91.15%
Edge_norm (418 queries)	Precision	86.65%	98.74%	92.19%
	Coverage	100.00%	100.00%	100.00%
	F-score	92.85%	**99.37%**	95.94%

(b) Cluster-level normalization quality.

Dataset	Metric	WService	CD-E	CD-C
Jobfeed_norm (395 queries)	SuccessRate	79.89%	88.43%	83.75%
	DiversityRR	51.59%	44.33%	61.79%
	F-score	62.69%	59.06%	**71.11%**
Edge_norm (418 queries)	SuccessRate	86.65%	98.74%	92.19%
	DiversityRR	51.91%	38.90%	80.83%
	F-score	64.92%	55.81%	**86.14%**

service. Being able to quickly scale up additional nodes in response to spiky request volume has been an ongoing challenge, since the service was first introduced two years ago.

The first version of the service was running directly on traditional servers, without using Docker containers. It took roughly 20 minutes for new server nodes to come online. If a node cluster was scaling up in response to spiky traffic, the service would typically throw a sizable percentage of HTTP server errors in the time window until it had scaled up to the desired cluster size. To minimize disruption for our users, we took the approach of over-provisioning at all times, i.e., always running a high number of servers instead of relying on auto scaling. However, this was not a good solution, as running large numbers of the heavyweight servers needed to hold the CompanyDepot indexes was quite expensive.

Over the past two years, the R&D team (that developed CompanyDepot) and the services team (that deployed the web service) have made several rounds of joint efforts to improve the service's startup time and reduce its large memory and CPU requirements. This work has delivered significant improvements and steadily improved scale-up time while driving down costs. We summarize the main optimizations we made in the following.

5.1.1 Operations Optimization. Firstly, to reduce costs and improve response times, we utilized Amazon's DynamoDB key-value store[6] as a cache for CompanyDepot results in the web service. This has paid off tremendously; even after multiple iterations of CompanyDepot optimization work, a request that can be served from the cache is still as much as 100 times faster than one that incurs a cache miss and must be normalized through CompanyDepot. Figure 2 shows median, average, 95th percentile, and 99th percentile timings for requests that yield cache hits vs. requests that yield cache misses. The 95th-percentile timing for cache hits is still mostly under 10 ms; for cache misses, it hovers around 1200 ms. Our cache hit rate typically exceeds 90%, with occasional dips to 60%-70%. In addition to the benefits in response time, DynamoDB record retrieval is also much less CPU-intensive than a CompanyDepot normalization. The combination of these two factors greatly reduces strain on our server cluster, allowing us to handle our traffic with fewer servers and significantly reduce operational cost. Our caching also helps us better absorb large spikes of traffic by serving most requests more quickly. To avoid serving out-of-date normalization results from the cache, we use the data version described in Section 5.2.2 as part of the composite cache key, which is basically a serialized form of the incoming service request along with the current data version. Including the data version in the key has the effect of invalidating all currently cached items whenever the data version is incremented. Immediately following the deployment of a new data version, our monitoring graphs of cache hit rate show a sudden drop to zero. However, we have observed that this number typically rebounds to normal 90% levels within only 5-10 minutes of

[6] https://aws.amazon.com/dynamodb/

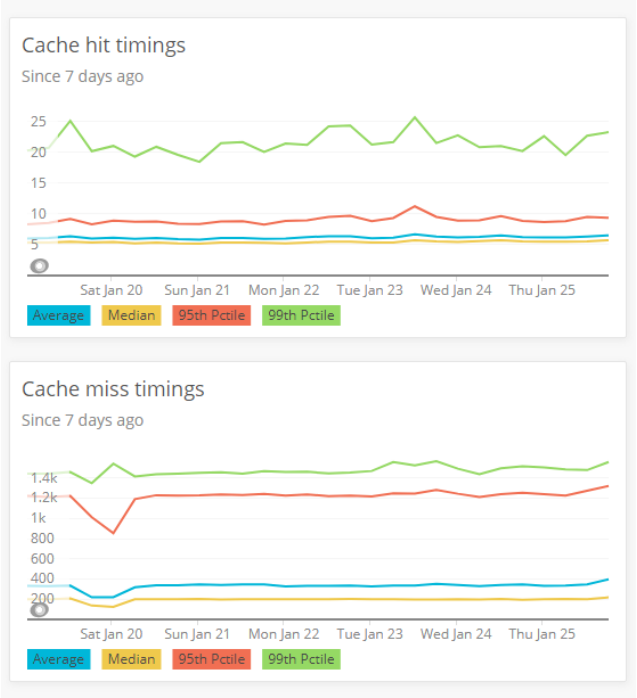

Figure 2: Service response time (in ms) monitored by NewRelic.

a deployment. This indicates that our customers' normal requests are heavily skewed toward a relatively small number of employers.

Secondly, we migrated the service from running on traditional servers to running in Docker containers. This reduced startup time by about three minutes, as Docker containers by nature are much more lightweight than traditional servers to start up. It also dramatically reduced operational costs. Since several Docker containers can reside together on a single server, we are now able to provide N running processes on far fewer than N physical servers, and scale up Company Normalization and other services at CareerBuilder significantly without needing to provision additional servers.

Besides providing the web service described above, we also provide a Java library which includes a jar file and corresponding data files (including index files and configuration files) so that clients could also directly use the library for Big Data job, to save time or cost. Actually, the largest user of CompanyDepot at CareerBuilder, called Recruitment Edge (described in Section 6.2), is using this approach. This has yielded multiple benefits across CareerBuilder's ecosystem. First, it improves reliability for Edge because, unlike the web service, the offline library never becomes temporarily unavailable or degraded. Second, it reduces operational costs for the services team because the removal of Edge as a web service customer drastically reduces the need to over-provision or plan for very spiky traffic. Finally, it improves stability for the remaining customers of the service, because the removal of Edge's spiky traffic lessens the likelihood of temporary service degradations.

5.1.2 CompanyDepot Optimization. Firstly, we eliminated the index compression after we found that downloading the uncompressed index files from Amazon S3 directly was faster than downloading plus unzipping the compressed index files. Changing our approach here reduced the startup time from 220 to 163 seconds (25.91% reduction).

Secondly, we removed redundant data in the KB index to reduce the overall file size. Most importantly, long enumerated string fields can be turned into a look up table. For example as the employer industry indicator, a NAICS[7] description such as "Employment Placement Agencies" can be looked up from NAICS code 561311. Storing only the NAICS code for each entity in the index, and looking up the NAICS description in real time from memory, offers a drastic index size reduction. We also removed some debugging information from the stored JSON field and optimized our JSON format to skip serializing empty collections. All together, we reduced the index size by 19.6%. A smaller index size means less download time. The startup time is further reduced from 163 to 115 seconds (29.45% reduction).

Last but not least, we focused on reducing the service's average response time. The less computation each request requires, the fewer servers needs to be allocated, which reduces the operating cost. We noticed that most of the computation is spent on retrieving and deserializing top 1000 entities from Lucene, and we can reduce the retrieval size to 100 without affecting the normalization quality on our labeled dataset. This yields 41.1% reduction in average response time, from 570 to 336 milliseconds, in production.

Through the combination of the above optimizations and some other smaller efforts, the R&D and services teams have successfully driven down startup time and operational cost of the Company Normalization Service by more than 90% since its initial release.

5.2 Maintenance

In this section, we describe how we maintain the Company Normalization service, specifically, how we monitor the service and how we version and update the service.

5.2.1 Service Monitoring. The Company Normalization service powers a variety of job seeker- and recruiter-facing applications in the CareerBuilder ecosystem. The success and perceived integrity of these applications depends on the service being fast, reliable, and highly available. To that end, the services team uses several technologies to effectively monitor the web service for uptime, response times, and error rates.

NewRelic[8] is the most heavily relied upon tool for monitoring the service. The services team uses NewRelic to monitor response times and general application health. It also allows for logging of custom parameters in request transactions. These custom parameters can be aggregated into charts and graphs in NewRelic's Insights dashboard, which enables both the services team and the R&D team to extract information about how the service is used. As an example, the two teams collaboratively used this feature to measure median, average, 95th percentile, and 99th percentile timings for every step of the CompanyDepot normalization process. This information enabled the R&D team to make informed decisions about where to

[7] https://www.census.gov/eos/www/naics/
[8] https://newrelic.com/

focus their efforts in order to optimize the normalization code. For example, we discovered that *Getting Top 1000 entities from Lucene* was a *hotspot* and a good candidate for further optimization.

While NewRelic is extremely effective at application- and server-level monitoring, it relies on requests being able to actually reach the application servers. Therefore, it would miss some scenarios when service users get errors. To catch such missed scenarios, we also monitor the service via a Splunk logging system[9] deployed within the routing layer. While Splunk lacks the intimate knowledge of the application that NewRelic has, it offers a more "zoomed-out" view, enabling us to track (and optionally alert on) changes in response times or error rates from a perspective closer to what end users see.

Finally, we also leverage a homegrown monitoring system for running constant integration tests against the service in its staging and production environments. This system was developed by the services team, and issues several test requests to the service every minute, verifying that the service responds successfully with the expected data. Any failures are reported immediately to the services team via email. This monitoring tool thus behaves both as an uptime monitor and as a runner platform for integration tests, and because it is developed and maintained at CareerBuilder, we are able to customize it as needed.

5.2.2 Versioning Strategy and Release Frequency. As the data files change more often than the jar file, we version them separately using the semantic versioning strategy as d-MAJOR.MINOR.PATCH and j-MAJOR.MINOR.PATCH, where the first letter refers to the data version and the jar version, respectively. We increase MAJOR version when we have a majority of data changed or make incompatible API changes. This happens every one or two years.

We increase MINOR version when we have an unknown minor portion of data changed or add functionality in a backwards-compatible manner. This happens every one or two quarters. Major and minor version updates usually require clients to reflow all their data or cache through the new version.

We increase PATCH version when we have a known portion of data changed or make backwards-compatible bug fixes. This often happens every two to four weeks, allowing us to support a two-week turnover time for fixing errors reported by clients. Patch version updates usually could be adopted by clients by only reflowing a given list of data items through the new version.

Generally, the Company Normalization service only runs the most recent version of the normalizer. Our internal users have adapted to a regular cadence of updates to CompanyDepot, so running only the latest version helps us reduce operational burden and infrastructure costs.

6 USAGE AND PAYOFF

In this section, we show how the CompanyDepot system has been used in CareerBuilder to deliver value to end customers, and discuss how to solve scalability issues when using it in data-intensive applications.

[9]https://www.splunk.com/

6.1 Jobfeed

6.1.1 Usage of the CompanyDepot data. Jobfeed[10] is a product of Textkernel that collects and aggregates online job postings from hundreds of thousands of websites for the purpose of labor market analysis and lead generation. The product is available for 10 countries, covering most of Europe and North America. In the U.S., Jobfeed collects and processes about 45 million jobs (170 million job postings) per year starting from 2016. Jobfeed uses this data to help users answer questions such as: *How has the demand for IT professionals changed in the Financial industry in the last few years?* or *Find medium-sized direct employers (no agencies!) that currently advertise for open positions related to Hadoop or Spark*. To this end, Jobfeed employs in-house semantic parsing, classification and deduplication technologies to collect, process and index online jobs in near real time, as they are found by the web crawlers. To enrich job postings with additional information about the advertising company (normalized company name, company industry, company size) Jobfeed uses the CompanyDepot system: an input is the raw company name extracted from the job posting, the location of the job (city and state), and the website where the posting was found (if found on a company site and not on a job board). The result from CompanyDepot is indexed and used in the Jobfeed system.

6.1.2 Implementation and Challenges. In 2017, Jobfeed U.S. processed on average 466,098 new job postings per day, with the maximum of 827,028 postings. Jobfeed uses CompanyDepot via web service, which greatly simplifies integration but does incur extra processing time: the round trip between Jobfeed and the service data centers. For a single job posting, a request takes 0.18 seconds on average, and Jobfeed generates 437 requests per minute (RPM), when excluding periods of inactivity. In addition to the real-time processing of new jobs, in Jobfeed we also reprocess entire historical job data when major updates to CompanyDepot are released: this is important to ensure that users can perform consistent analyses over longer time periods. With 170 million job postings collected per year, this presents a bigger scaling challenge: reprocessing two years of data within two weeks would require 16,865 RPM. To reduce the load on CompanyDepot during data reprocessing, we aggressively cache the service responses on the input (raw company name and website). The input job location is ignored for the purpose of caching during data reprocessing to trade off normalization quality for run time, as we found that there is a less than 5% difference in results comparing using input location or not, on a random dataset. We use Least Recently Used (LRU) cache policy with a memory limit of 5GB, which allows us to reduce the load by a factor of 10, while keeping reprocessing time within one week for a year of job data.

6.2 Recruitment Edge

6.2.1 Usages of the CompanyDepot data. Recruitment Edge[11] is a product at CareerBuilder that collects, normalizes, and deduplicates billions of publicly available candidate profiles [2]. Normalization is a crucial step in de-duplication and company name is one of the attributes Edge needs to normalize. Edge leverages

[10]https://www.jobfeed.com/
[11]https://edge.careerbuilder.com/

the CompanyDepot system to normalize approximately 116 million raw company names from candidate profiles. An input query from Edge is typically a raw company name and its country when geographic location of the company is available. From the CompanyDepot results, Edge indexes the normalized company names, the industry codes, and the company sizes to allow its users to facet on these fields for searching candidates. Meanwhile, Edge uses the normalized company names to de-duplicate employment histories of candidate profiles.

6.2.2 Implementation and Challenges. To leverage the CompanyDepot system in a scalable manner, Edge uses it in a distributed fashion, specifically, from Hadoop with a Spark process, which stores the normalization results in a table. As mentioned in Section 5.1.1, there are two ways to interact with CompanyDepot: via web service or via library. Edge initially architected its normalization Spark process to call the web service. At approximately 4,500 requests per minute (RPM), it took about 18 days for Edge to run all of its requests through the service. This approach was not only slow but also cost-inefficient. Whenever there was a major update to CompanyDepot, all of the company names had to be re-normalized, which incurred a significant bump in the expenses. In addition, the rate at which Edge needs to call the web service occasionally stressed the service out, thereby affecting other products using it.

To alleviate these pain points that Edge was inflicting on the web service and other products as well as to save costs, Edge turned to interacting directly with the CompanyDepot library. On one hand, Edge was able to shave off the latency associated with calling a web service, but on the other hand, its throughput was constrained by the amount of resources allotted to Edge in CareerBuilder's Hadoop cluster as the CompanyDepot library was especially memory-intensive. When deciding the memory allocation, there was a trade-off between increasing parallelism by reducing memory footprint and decreasing garbage collection by increasing heap space. We found that sacrificing parallelism a bit with plenty of heap space yielded better run time for Edge. When Edge maximized its resource utilization, it was able to normalize at approximately 5,500 RPM, which yielded a run time of 14.6 days for a full run. This run time can be further reduced with the improvement of the memory-efficiency of CompanyDepot. In fact, the optimizations made in Section 5.1.2 enabled Edge to save three additional days in run time, reaching 7,000 RPM and a run time of 11.5 days.

6.3 School Normalization and Major Normalization

The CompanyDepot architecture can also be easily applied to other named entity normalization problems, with some customization of KB, mapping sources, feature set used for learning to rank, and whether to do clustering. Next, we show how we build our school normalization system SchoolDepot and major normalization system MajorDepot by adopting the CompanyDepot architecture.

The SchoolDepot system uses the same feature set as used in CompanyDepot, while using the school KB as well as the Wikipedia and legacy mappings collected in [11]. The clustering step is skipped here as the school KB is much cleaner than the employer KB. Compared to the heuristic method described in [21], the success rate of SchoolDepot using learning to rank improved from 90.1% to 95.5% on a UK dataset (2448 samples), and from 81.1% to 84.8% on an Edge Top 10k dataset (486 samples).

The MajorDepot system uses the Classification of Instructional Programs (CIP)[12] as the major KB. Mappings and clustering are not used. Its learning-to-rank process uses a simple set of four features: the score from the Lucene searcher, whether the major name is suffix of the query, number of common words between query and major name, popularity score for the major. The success rate of MajorDepot is about 80% on a resume dataset.

Table 6: Error types and error handling.

Type	Subtype	Handling
Wrong input	n/a	inform clients
Wrong normalization	missing entity	use KB config file
	wrong entity result	use mapping/calibration file
	wrong clustering	use cluster config file
Wrong attributes	n/a	use KB config file

7 FEEDBACK LOOPS

In this section, we describe how we correct errors found in results and use machine learning methods to automatically detect errors.

7.1 Error Handling

An important request we received from our clients is that errors reported by them can be quickly fixed, preferably with a two-week turnover time. The decoupling of the KB index, mapping index and cluster index enables our system to independently update each index, and to easily support ad-hoc correction of errors in the KB, mappings, clustering function, and calibration rules using configuration files. Table 6 summarizes the types of errors we encountered and how we handled them respectively. The configuration files that contain the correct data manually edited for error fixes are integrated with the indexes during system startup.

7.2 Automatic Error Detection

Since the employer KB is large, it is impossible to manually identify the errors shown in Table 6 in a large scale. Therefore, we use machine learning techniques to automatically detect the errors. With the observation that the main jobs posted by an employer often relate to the employer industry, e.g., truck driver jobs often correspond to employers in the transportation industry, we developed a system that classifies the industry of an employer using job posting data [6, 9]. We aggregate job postings from an employer and derive features from employer names, employer descriptions, job titles, and job descriptions to predict the industry of the employer, using SVM and Random Forest models. When any model prediction disagrees with the industry returned by CompanyDepot, a potential error is suggested. We found that Random Forest is more effective than SVM in identifying the errors, which achieves precision 0.69, recall 0.78, and f-score 0.73. It especially better handles mixed feature vectors when normalization errors occur. The potential errors suggested by the machine learning models can then be manually verified and corrected.

[12] https://nces.ed.gov/ipeds/cipcode/

8 CONCLUSION AND FUTURE WORK

In this paper, we described the CompanyDepot system developed at CareerBuilder, which uses machine learning techniques to address the employer name normalization task at two levels of complexity: entity level and cluster level. Specifically, we solved entity-level normalization in three steps: entity retrieval, reranking based on learning, and validation. We used external mappings to do query expansion and entity clustering, which helps address the name variation and duplicate entity challenges. For selecting cluster representative entities, we used a machine learning approach instead of heuristics, which led to more accurate attributes returned in results. Next, we discussed the main challenges in deploying the system such as how to quickly scale up additional servers in response to spiky request volume. We shared our lessons learned in deployment, maintenance, and utilization of the system in the past two years, especially the optimizations we made that led to significant performance improvements and cost savings. We showed how the system is being used in applications at CareerBuilder, how we addressed the scalability issues, and how we applied the same system architecture for normalizing schools or majors. Finally, we discussed the types of errors reported by users, and how we used machine learning methods to automatically detect more errors on a large scale.

In the future, we plan to work on further improving the accuracy of attributes such as industry, company size, and URL. We also plan to improve normalization quality for international employer names (e.g., UK and Canada) by considering country in the calibration and clustering steps. It will also be helpful to build a web interface that enables our data analysts who are not familiar with the technical details of CompanyDepot to be able to correct errors found in the results.

REFERENCES

[1] J. Gary Augustson and Jack Minker. 1970. An Analysis of Some Graph Theoretical Cluster Techniques. *J. ACM* 17, 4 (Oct. 1970), 571–588.

[2] Janani Balaji, Faizan Javed, Chris Min, and Sam Sander. 2017. An Ensemble Blocking Approach for Entity Resolution of Heterogeneous Datasets. In *Proceedings of the 30th International FLAIRS (Florida Artificial Intelligence Research Society) Conference.*

[3] A. Borkovsky. 2003. Item name normalization. (April 29 2003). US Patent 6,556,991.

[4] R. Busa-Fekete, Gy. Szarvas, T. ÃĽltetÅŚ, and B. KÃľgl. 2012. An apple-to-apple comparison of Learning-to-rank algorithms in terms of Normalized Discounted Cumulative Gain. In *B. Proceedings of ECAI-12 Workshop, Preference Learning: Problems and Applications in AI.* http://www.inf.u-szeged.hu/~busarobi/publ.html

[5] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Issue 3. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[6] Alan Chern, Qiaoling Liu, Josh Chao, Mahak Goindani, and Faizan Javed. 2018. Automatically Detecting Errors in Employer Industry Classification using Job Postings. *accepted by the Data Science and Engineering journal* (2018).

[7] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection.* Springer Publishing Company, Incorporated.

[8] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD'96).* 226–231.

[9] Mahak Goindani, Qiaoling Liu, Josh Chao, and Valentin Jijkoun. 2017. Employer Industry Classification Using Job Postings. In *2017 IEEE International Conference on Data Mining Workshops, ICDM Workshops 2017, New Orleans, LA, USA, November 18-21, 2017.* 183–188. DOI: http://dx.doi.org/10.1109/ICDMW.2017.30

[10] Nitish Gupta, Sameer Singh, and Dan Roth. 2017. Entity Linking via Joint Encoding of Types, Descriptions, and Context. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2681–2690. http://aclweb.org/anthology/D17-1284

[11] Ferosh Jacob, Faizan Javed, Meng Zhao, and Matt Mcnair. 2014. sCooL: A system for academic institution name normalization. In *Collaboration Technologies and Systems (CTS), 2014 International Conference on.* 86–93.

[12] A. K. Jain, M. N. Murty, and P. J. Flynn. 1999. Data Clustering: A Review. *ACM Comput. Surv.* 31, 3 (Sept. 1999), 264–323.

[13] Faizan Javed, Phuong Hoang, Thomas Mahoney, and Matt McNair. 2017. Large-Scale Occupational Skills Normalization for Online Recruitment. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.* 4627–4634.

[14] Siddhartha Jonnalagadda and Philip Topham. 2010. NEMO: Extraction and normalization of organization names from PubMed affiliation strings. *Journal of Biomedical Discovery and Collaboration* 5 (2010), 50.

[15] Amin Karami and Ronnie Johansson. 2014. Choosing dbscan parameters automatically using differential evolution. *International Journal of Computer Applications* 91, 7 (2014).

[16] Hakan Kardes, Deepak Konidena, Siddharth Agrawal, Micah Huff, and Ang Sun. 2013. Graph-based Approaches for Organization Entity Resolution in MapReduce. In *Proceedings of TextGraphs-8 Graph-based Methods for Natural Language Processing Workshop at EMNLP.*

[17] Mayank Kejriwal, Qiaoling Liu, Ferosh Jacob, and Faizan Javed. 2015. A Pipeline for Extracting and Deduplicating Domain-Specific Knowledge Bases. In *Proceedings of 2015 IEEE International Conference on Big Data.*

[18] Lars Kolb, Ziad Sehili, and Erhard Rahm. 2014. Iterative Computation of Connected Graph Components with MapReduce. *Datenbank-Spektrum* 14, 2 (2014), 107–117.

[19] Robert Leaman, Rezarta Islamaj Dogan, and Zhiyong Lu. 2013. DNorm: disease name normalization with pairwise learning to rank. *Bioinformatics* 29, 22 (2013), 2909–2917.

[20] Qiaoling Liu, Faizan Javed, Vachik S. Dave, and Ankita Joshi. 2017. Supporting Employer Name Normalization at Both Entity and Cluster Level. In *Proc. of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17).* 1883–1892.

[21] Qiaoling Liu, Faizan Javed, and Matt McNair. 2016. CompanyDepot: Employer Name Normalization in the Online Recruitment Industry. In *Proc. of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16).* 521–530.

[22] Walid Magdy, Kareem Darwish, Ossama Emam, and Hany Hassan. 2007. Arabic Cross-document Person Name Normalization. In *Proceedings of the 2007 Workshop on Computational Approaches to Semitic Languages: Common Issues and Resources (Semitic '07).* 25–32.

[23] Wiliam P. McNeill, Hakan Kardes, and Andrew Borthwick. 2012. Dynamic Record Blocking: Efficient Linking of Massive Databases in MapReduce. In *Proceedings of 10th International Workshop on Quality in Databases (QDB) at VLDB.*

[24] Donald Metzler and W. Bruce Croft. 2007. Linear Feature-based Models for Information Retrieval. *Inf. Retr.* 10, 3 (June 2007), 257–274. DOI: http://dx.doi.org/10.1007/s10791-006-9019-z

[25] Malay K Pakhira. 2014. A Linear Time-Complexity k-Means Algorithm Using Cluster Shifting. In *Computational Intelligence and Communication Networks (CICN), 2014 International Conference on.* IEEE, 1047–1051.

[26] Hae-Sang Park and Chi-Hyuck Jun. 2009. A Simple and Fast Algorithm for K-medoids Clustering. *Expert Syst. Appl.* 36, 2 (March 2009), 3336–3341.

[27] Minh C. Phan, Aixin Sun, Yi Tay, Jialong Han, and Chenliang Li. 2017. NeuPL: Attention-based Semantic Matching and Pair-Linking for Entity Disambiguation. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17).* 1667–1676.

[28] Thomas Seidl, Brigitte Boden, and Sergej Fries. 2012. CC-MR — Finding Connected Components in Huge Graphs with Mapreduce. In *Proceedings of the 2012 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part I (ECML PKDD'12).* 458–473.

[29] Wei Shen, Jianyong Wang, and Jiawei Han. 2015. Entity Linking with a Knowledge Base: Issues, Techniques, and Solutions. *IEEE Trans. Knowl. Data Eng.* 27, 2 (2015), 443–460.

[30] Joachim Wermter, Katrin Tomanek, and Udo Hahn. 2009. High-performance gene name normalization with GENO. *Bioinformatics* 25, 6 (2009), 815–821.

[31] Dongkuan Xu and Yingjie Tian. 2015. A Comprehensive Survey of Clustering Algorithms. *Annals of Data Science* 2, 2 (2015), 165–193.

[32] Baoshi Yan, Lokesh Bajaj, and Anmol Bhasin. 2011. Entity Resolution Using Social Graphs for Business Applications. In *International Conference on Advances in Social Networks Analysis and Mining, ASONAM 2011.* 220–227.

[33] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *SIGMOD Rec.* 25, 2 (June 1996), 103–114.