Using Deep Learning to Perform Fuzzy Joins

Abraham Judah Gale
Thesis Submitted to
Yeshiva University
yehuda.gale@gmail.com

Kavitha Srinivas
Advisor
Rivet Labs
Kavitha@rivetlabs.io

Judah Diament
Advisor
Yeshiva University
diament@yu.edu

1. INTRODUCTION

Joining two datasets is a key step in preparing the data for subsequent operations such as performing business analytics, building predictive models etc. Data management systems have largely focussed solely on equi-joins, which is based on exact equality of strings or numeric values. However, in many circumstances, exact equality is inadequate because the same entity may in many cases be expressed with slight variations of the same name, as shown in Figure 1 (e.g., *Douglas Adams* and *Douglas Noel Adams*). Although Figure 1 illustrates the problem for people's names, the same problem occurs for other entity types, such as company names, addresses, states in countries, and countries.

One common technique to automating joins of the sort described in Figure 1 is string matching. String matching approaches use variants of string similarity measures such as edit-distance, Jaro-Winkler and TF-IDF (e.g., [2]) to perform matching. Typically to scale the problem to a large number of rows in each table, a filtering or blocking strategy is used to reduce the number of pairs to be considered for string matching. For instance, prefix, string length, or suffix based filtering are often applied to the strings in the two tables to identify potentially joinable pairs that need to be matched. For the examples shown in Figure 1, the prefix filtering step will ensure that *Douglas Adams* will be compared only with *Douglas Noel Adams* and *Doug C. Engelbart* to determine their string similarity, assuming the prefix used for filtering is *Doug*.

More recently, data driven approaches have emerged as a powerful alternative to string matching techniques for the join problem described in Figure 1. Data driven approaches mine patterns in the data to determine the 'rules' for joining a given entity type. One example of such an approach is illustrated in [5], which determines which cell values should be joined based on whether those cell values co-occur on the same row across disparate tables in a very large corpus of data. Such a system can perform 'semantic joins' such as mapping country names to country codes in new unseen tables. Another example of a data driven approach is work by [10] that uses program synthesis techniques to learn the right set of transformations needed to perform the entity matching operation, based on a numerous examples. Unlike string matching algorithms, which apply the same generic operator to a cell value regardless of entity type, the approach outlined in [10] can learn different programs to transform cell values for different entity types.

In this paper, we propose a novel data driven approach to the problem of joining semantically different representations of data. Our approach relies on building supervised deep learning models to automatically learn the correct set of transformations needed to compute the equality of two cell values. Because a deep learning model can pick up the appropriate features of what should be used for matching from correlations in the data, it should in theory be able to generalize better across join problems than the data driven approaches outlined in [5]. Conceptually, our approach is similar in spirit to the approach described in [10], but we use deep neural networks to learn the right function for the join operation instead of synthesizing different programs for different entity types. The advantage of building such a function over the approach described in [10] is that we can use this function in novel ways to completely eliminate the filtering that is typically needed to identify the set of potentially joinable rows, as we describe below.

Our specific solution to the join problem involves building a deep neural network that learns to produce a small distance estimate for elements of a name pair that represent the same entity (e.g., *Douglas Adams-Douglas Noel Adams* should produce a distance estimate d that is closest to 0), and a much larger distance estimate for elements of the name pair that do not represent the same entity (e.g., *Douglas Adams-John Adams* should produce a distance estimate d that is greater than some margin m). The function learnt by such a network (often called a 'siamese network') is conceptually one that maps input vectors for the same entity closer together in vector space, while mapping input vectors for a different entity to a distance that is at least m distance away from the vectors for the same entity, as shown in Figure 1. This sort of function can be used to determine join equality on a subset of string pairs that are considered joinable using some sort of filtering step, as is often done in approaches to the semantic join type problem (e.g. [10]).

However, our observation is that one can actually exploit what the siamese network produces to eliminate this filtering step altogether. Specifically, the last hidden layer of the siamese network is effectively a 'vector embedding' for the same versus different estimate. That is, the last layer is a

department	employee
101	Douglas Adams
101	Andreas Capellanus
101	John Adams Whipple
101	Douglas Engelbart

employee	salary
Douglas Noel Adams	10000
Andrea Cappellano	20000
John A. Whipple	30000
Doug C. Engelbart	30000

Figure 1: Example of a join problem

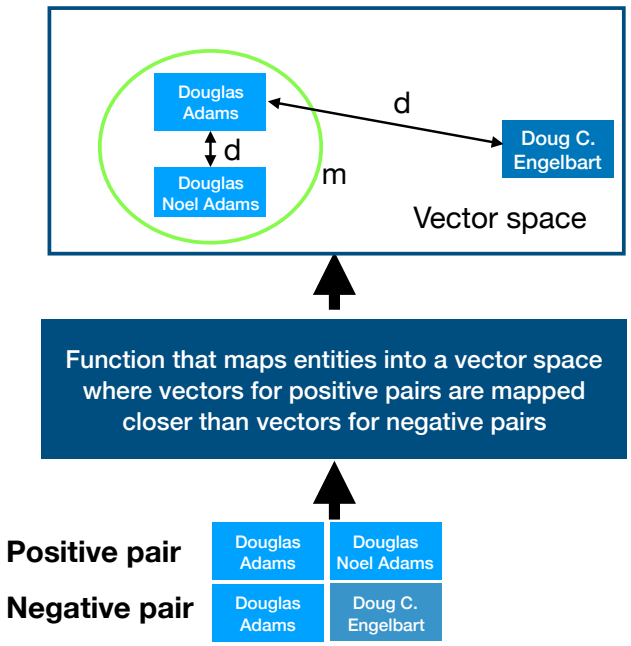

Figure 2: Conceptual overview of neural network approach to matching

vector in a lower dimensional space that contains the critical features needed for computing the distance estimate. We can in fact take these vector embeddings for all vectors in the two tables to be joined, and use approximate nearest neighbors algorithms to find the nearest neighbors. Ideally, the nearest neighbor that has a distance lower the margin m should be the correct match. Because approximate nearest neighbor algorithms have been applied successfully to millions of items, this approach should scale just as well as filtering/join algorithms, but have the added advantage that the filtering step is also data driven, and hence can adapt to different entity types very effectively.

Our key contributions to the join problem are as follows:

- We demonstrate that we can use a triplet loss network to learn a distance function that successfully discriminates same pairs from different pairs with an accuracy of 97%. Our results suggest that deep learning models can be used successfully as another mechanism for data driven joins, assuming some sort of filtering approach has been applied to the data.
- We try to extend this further; to see if the output of the function can in fact be used in an approximate nearest neighbor algorithm to identify neighbors to match with, without any filtering.

2. RELATED WORKS

There have been a number of attempts to solve the fuzzy join problem. Many of the attempts use various string matching algorithms for example [9]. The main issue with using string matching comparisons is that they do not work across a wide variety of entities. To our knowledge no serious machine learning approach has been used for this problem. The problem of blocking has also been dealt with using MapReduce [8]. This approach requires a lot of computing power and, since they use a string matching approach in the end, is hard to generalize across many types of entities. There has also been work done on a related yet different problem of fuzzy joining on whole rows [4]. We deal with the case where we only have one column of information.

3. ALGORITHM

3.1 Rule Based

We started our process with a simple rule based method for matching and another one for blocking.

In the rule-based method, we block based on items that have one word in common. This is the simplest reasonable blocking strategy. Table 1 illustrates what the blocking of the three names from Figure 1 would look like. To perform the actual matching, we used three simple rules see Table 3:

1. If a word is unique to two items, we match them. ('John Adams Whipple' matches 'John A. Whipple' since 'Whipple' only apears in those two names)
2. If, when all spaces are removed, one of the items is a substring of the other, we match them. ('the flower company' matches 'theflowercompany.com' since 'theflowercompany' is a subset of 'theflowercompany.com')
3. If we treat each name as a set of words, and one set is a subset of the other set, we match them. ('Douglas Adams' matches 'Douglas Noel Adams' since {'Douglas', 'Adams'} $\subseteq$ {'Douglas', 'Noel', 'Adams'})

Even using these three rules in combination resulted in too many false negatives. Therefore we decided to treat all matches resulting from any of the three rules as valid.

3.2 Siamese Network

A method for logically mapping a set of points onto a manifold using a set of previously known matches and non-matches is described by [3]. The goal of this technique is to minimize the distance between matched pairs and maximize the distance between unmatched pairs. This is accomplished using a network architecture know as a *Siamese* network. In this architecture there are two deep neural networks that share weights attached to a final layer that determines the distance between the output of the two hidden networks and minimizes the loss function. To train the network a set of matched and non-matched pairs of inputs is sent through the network and the weights are adjusted to maximize or minimize distance accordingly. This is accomplished using the following loss function

$$L(W, Y, \vec{X}_1, \vec{X}_2) = (1-Y)\frac{1}{2}(D_W)^2 + (Y)\frac{1}{2}\{max(0, m-(D_W))\}^2$$

Where D_W is the euclidean distance between the two outputs, m is a radius beyond which we ignore dissimilar points, and Y is the dissimilar label.

Table 1: Table 2

Bucket Name	Contents
A	{John A. Whipple}
Adams	{Douglas Adams, Douglas Noel Adams, John Adams Whipple}
Andreas	{Andreas Capellanus}
Andrea	{Andrea Cappellano}
Cappellano	{Andrea Cappellano}
Capellanus	{Andreas Capellanus}
Douglas	{Douglas Adams, Douglas Noel Adams}
John	{John Adams Whipple, John A. Whipple}
Noel	{Douglas Noel Adams}
Whipple	{John Adams Whipple, John A. Whipple}

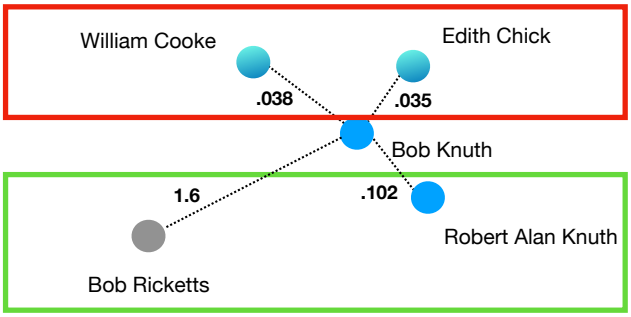

Figure 3: The model preformed well on somewhat similar names and badly on completely different names

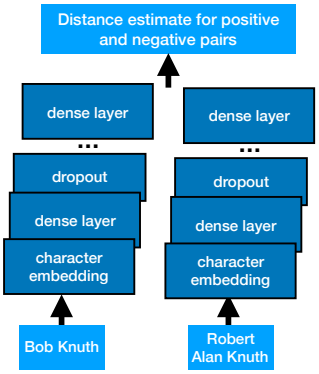

Figure 4: A siamese network has two networks which share weights which feed into a distance estimator

By mapping the inputs onto meaningful locations on a manifold, this method clusters similar inputs close to each other. Using this property we can use the distance layer to decide if two inputs are matched. We simply set a limit for euclidean distance and match if the outputs of the hidden layers are within that limit, as in Figure 4. This lets us use this method for our problem of matching, we treat items as similar to each other if they are aliases for the same entity and different otherwise. To begin with we created the non-matched pairs by choosing other entities with at least one word in common, to avoid the network learning a useless function. This successfully brought the matches close together in the input but failed to properly distance pairs that should be different, as shown in Figure 3. Since we never fed the network pairs that were completely different it did not learn to distinguish them from matches. To combat this we created two types of different pairs: ones that share at least one word in common and ones that are completely different.

3.3 Approximate Nearest Neighbors

We used the ANNOY package to find the nearest neighbors in output of the hidden layer computed in the previous section.[1] This package approximates the nearest neighbors by first calculating binary trees using the randomized k-d forest method.[6] These trees are calculated by splitting the vector space by drawing random hyperplanes. Each side of the hyperplane is one node of the binary tree. The algorithm then continues to split each subspace recursively until no more than a predetermined number of items are in each subsection. Some of the neighbors can be found by looking at the items in the same subspace, or subspaces near the target item in the tree. The processes of building a tree is repeated a number of times. By taking a union of the points from all the trees, a reasonable neighborhood is approximated. Once we have a neighborhood of points, we can calculate distance and sort the subset of points. Since much of the time goes into creating the trees, queries can be executed quickly, in logarithmic time.

3.4 Triplet Loss

While the previos aproach maps 2 entities at a time using 2 networks that share weights, this aproach maps 3 entities at a time with 3 networks: the anchor the positive and the negative. The loss function minimizes the difference between the anchor and the positive at the same time as it maximises the difference between the anchor and the negative. By doing this we can deal with an unmatched pair and a matched pair at the same time: the anchor is one name, the positive is a match for that name and the negative is another name which is not a match.[7] We accomplish this by minimizing the following loss function:

$$L(\vec{X}_a, \vec{X}_n, \vec{X}_p) = | \, ||\vec{X}_a - \vec{X}_p||^2 - ||\vec{X}_a - \vec{X}_n||^2 + \alpha|$$

where $\vec{X}_a$, $\vec{X}_n$, $\vec{X}_p$ are the embedded anchor, negative and positive vectors and α is a constant margin. The key choice here is which vectors to include in the training set. If too

many easy triplets are including it will slow down the learning since the constraint will be easily satisfied. If too many hard triplets are included the network will learn a strange function based on unusual cases

3.5 LSTM

A regular Deep Neural network would treat the name like a bag of words, without accounting for order. We experimented with using an LSTM architecture to account for order the words apear in. This makes sense for our use case since first name and last names are different from eachother. In the implimentation we used GRUs (Gated Recurrent Unit), which function like LSTMs but are computationally cheaper.

4. EXPERIMENTS

4.1 Rule Based

To test the rule based matcher we ran the blocker followed by the matcher and calculated F-scores based on how many correct matches were in the top 3. We got an F-score of 0.6. Since the issue is mostly false negatives, and therefore even some of the first three slots were often empty, so even if we accept any of the top 1000 as a success, we get an F-score of 0.63, not much better.

4.2 Siamese Network

Our first experiment was using a Siamese network to match already blocked entities. The architecture of the network was as shown in Figure 4. As described above this involves two networks that share weights mapping the inputs onto a vector space and computing distance. Before feeding the entities into the networks, we used Kazuma character embeddings to encode each entity as a vector. We used character level embeddings since many names would not have been represented at all if we had used word level embeddings. We then trained the network on our 182772 pairs of names pulled from DBpedia. These are the pairs we have after running the data through the cleanser. We used 95% of them for training and withheld 5% for testing. In addition to these pairs we created an equal number of negative pairs to train the model. Wherever possible, the negative pairs had at least one word in common with each other. This was done so that the model would not just learn the obvious function of reject unrelated pairs. We trained the model in 10 epochs. We found that the fscore on the training data was .89 on the test data. This showed that this architecture is an effective matcher. However we only successfully included the correct match in around 15% of pairs in the top five. This was barely better than the 5% we got from the embedding alone.

4.3 Triplet Loss

The triplet loss experiment was extremely similar to the Siamese one, only we used the Triplet loss network instead of the Siamese one. We used a margin of 1. We also used a GRU instead of a regular deep neural network to capture the order of the words. The last change we used was in selection of negative points, instead of picking them based on some rule, we picked the closest 40 points that were not matches from the character embedding. We did this since those names are the hardest ones for the model to learn, this also avoids the problem in Figure 3. When we did this we got 69% of the items in the top 40. This is a relatively effective blocker. It also correctly placed 97% of positives closer to the anchor than negatives, making it a very effective matcher.

5. CONCLUSIONS

While we have a very good machine learning based matcher, that does not appear to translate into very good blocking. We still need to explore why we only getting around 69% of the matches in the top 40 items, when we have the positive closer than the negative in 97% of cases.

6. REFERENCES

[1] E. Bernhardsson. Nearest neighbors and vector models part 2 algorithms and data structures, Oct 2015.

[2] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of IJCAI-03 Workshop on Information Integration*, pages 73–78, August 2003.

[3] R. Hadsell, S. Chopra, and Y. Lecun. Dimensionality reduction by learning an invariant mapping. *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2 (CVPR06)*.

[4] Y. He, K. Ganjam, and X. Chu. Sema-join. *Proceedings of the VLDB Endowment*, 8(12):13581369, Jan 2015.

[5] Y. He, K. Ganjam, and X. Chu. Sema-join: Joining semantically-related tables using big table corpora. *Proc. VLDB Endow.*, 8(12):1358–1369, Aug. 2015.

[6] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):22272240, Jan 2014.

[7] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[8] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. *Proceedings of the 2010 international conference on Management of data - SIGMOD 10*, 2010.

[9] J. Wang, G. Li, and J. Fe. Fast-join: An efficient method for fuzzy token matching based string similarity join. *2011 IEEE 27th International Conference on Data Engineering*, 2011.

[10] E. Zhu, Y. He, and S. Chaudhuri. Auto-join: Joining tables by leveraging transformations. International Conference on Very Large Databases (VLDB), May 2017.