DeezyMatch: A Flexible Deep Learning Approach to Fuzzy String Matching

Kasra Hosseini The Alan Turing Institute
Federico Nanni The Alan Turing Institute
Mariona Coll Ardanuy The Alan Turing Institute
{khosseini,fnanni,mcollardanuy}@turing.ac.uk

Abstract

We present DeezyMatch, a free, open-source software library written in Python for fuzzy string matching and candidate ranking. Its *pair classifier* supports various deep neural network architectures for training new classifiers and for fine-tuning a pretrained model, which paves the way for transfer learning in fuzzy string matching. This approach is especially useful where only limited training examples are available. The learned DeezyMatch models can be used to generate rich vector representations from string inputs. The *candidate ranker* component in DeezyMatch uses these vector representations to find, for a given query, the best matching candidates in a knowledge base. It uses an adaptive searching algorithm applicable to large knowledge bases and query sets. We describe DeezyMatch's functionality, design and implementation, accompanied by a use case in toponym matching and candidate ranking in realistic noisy datasets.

1 Introduction

String matching is an integral component of many natural language processing (NLP) pipelines. One such application is in entity linking (EL), the task of mapping a mention (i.e., a string) to its corresponding entry in a knowledge base (KB). Most EL systems currently rely on a lookup table (Ferragina and Scaiella, 2010; Mendes et al., 2011; Raiman and Raiman, 2018; Sil et al., 2018)[1] or shallow string similarity approaches (e.g., based on n-gram overlaps as in McNamee et al. (2011b); Plu et al. (2016), or super-string matching, as in Moro et al. (2014)) to narrow the entries of a KB down to a set of potential candidates the mention may refer to (i.e., aliases). While these choices allow fast run-time, they generally rely on the assumption that all surface forms of each entity are present as aliases in the KB. The performances of these systems degrade when dealing with domain-specific vocabulary (Munnelly and Lawless, 2018), local variations (Rovera et al., 2017), historical materials (Olieman et al., 2017; McDonough et al., 2019) and, in general, challenges that emerge when performing EL on non-standard documents.[2] This subtask of EL, often referred to as candidate ranking (and selection), is mostly ignored when designing downstream systems, even though its significant impact on downstream NLP pipelines has been shown previously (Quercini et al., 2010; Hachey et al., 2013).

In this paper, we present DeezyMatch, a new deep learning approach that strives to address advanced string matching and candidate ranking in a more comprehensive and integrated manner than existing tools. DeezyMatch is a free, open-source community software written in Python. It uses PyTorch (Paszke et al., 2019) to implement various state-of-the-art neural network architectures, and it has been tested on both CPU and GPU. One of the main features of DeezyMatch is its modular design and flexibility. We describe DeezyMatch's functionalities, design choices and technical implementation. We compare its performance in relation to other approaches on several realistic string matching scenarios, covering different languages, alphabets, and domains, and we evaluate the quality of the candidate ranker in a real-case setting. Thanks to its easy-to-use interface, DeezyMatch can be seamlessly integrated into existing EL systems. This allows DeezyMatch to be adopted out-

[1] See, for instance, DBpedia Lexicalization dataset used as a lookup table by DBpedia Spotlight: https://wiki.dbpedia.org/lexicalizations, or how Spacy currently retrieves candidates from a given KB: https://spacy.io/api/kb/#get_candidates.

[2] As opposed to Wikipedia or contemporary newspaper text, which are employed in widespread EL benchmarks, such as WikiDisamb30, CoNLL (YAGO), and AC KBP 2010.

Proceedings of the 2020 EMNLP (Systems Demonstrations), pages 62–69
November 16-20, 2020.

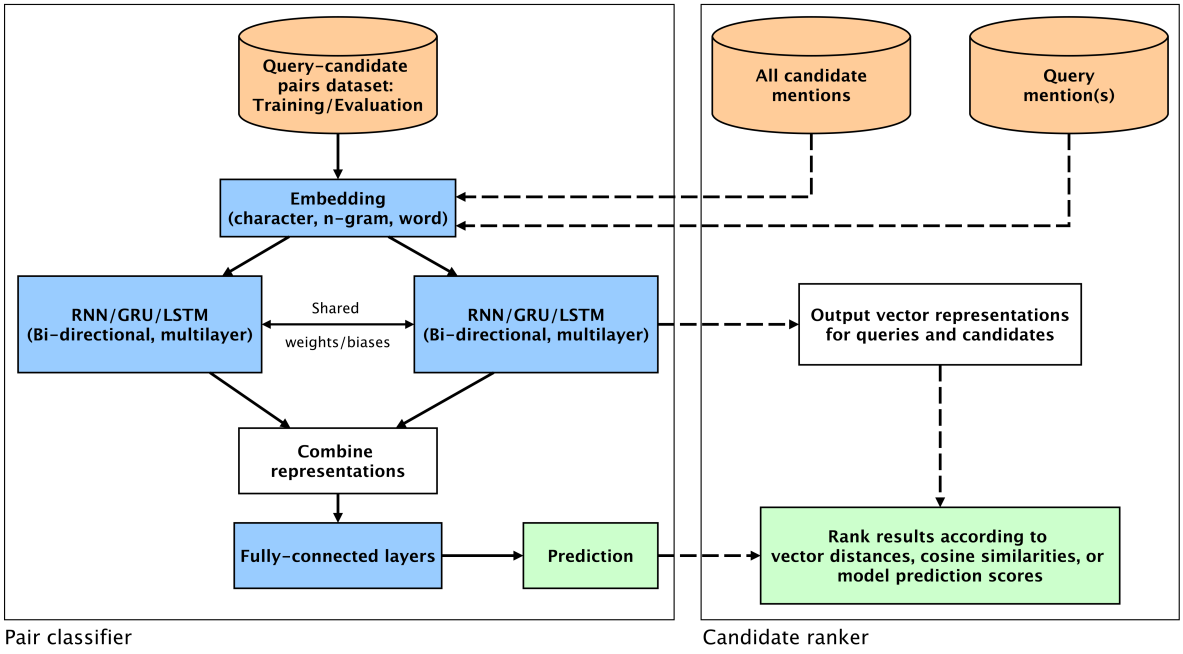

Figure 1: DeezyMatch architecture consists of two main components: *pair classifier* (left box) and *candidate ranker* (right box). The learnable parameters in *pair classifier* are highlighted in blue. During fine-tuning, any of these parameters can be frozen, that is, they will not be changed during fine-tuning. Various hyperparameters including the architecture of the neural network and tokenization can be changed by the user (see text). In *candidate ranker*, for each query and candidate pair, learned vector representations are first generated using a DeezyMatch model. These vectors are then used to rank candidates according to different metrics (e.g., L_2-norm distance, cosine similarity and prediction scores). The steps of *candidate ranker* are depicted by dashed lines in the figure.

side the NLP community, especially in Digital Humanities, where it could play a major role in addressing known issues concerning the EL systems and their adaptability to the non-standard nature of the datasets typically used in this field (Olieman et al., 2017).

DeezyMatch is released under MIT License. It is available via PyPI,[3] and its source codes are on GitHub.[4] We provide extensive documentation, including examples in Jupyter Notebooks, to enable the smooth adoption of all its components.

2 Description of the system

Fig. 1 shows the two main components of DeezyMatch: *pair classifier* and *candidate ranker*. Together they allow the training or fine-tuning of a query-candidate classifier and finding best matching candidates to a query from a KB.

2.1 Pair classifier

Inspired by the work of Santos et al. (2018a), DeezyMatch's pair-classifier component has at its core a siamese deep neural network classifier. The network takes query-candidate pairs as inputs which can be further processed (e.g., lower-cased and normalized) and tokenized at different levels (character, n-gram and word). Such pairs are either possible referents of the same entity or not, which form the positive and negative examples for training and testing. The neural network architecture and its hyperparameters can be configured in the input file without requiring the user to modify the code. Currently, DeezyMatch supports Elman Recurrent neural network (RNN) (Elman, 1990), Long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) and Gated Recurrent Unit (GRU) (Cho et al., 2014) architectures. The number of layers and directions (mono or bi-directional) in the recurrent units as well as the dimensions of hidden states and embedding layers can be changed in the input file. The two parallel recurrent layers in Fig. 1 share their weights and biases which helps the model to learn transformations regardless of the order of strings in an input pair.

During training, a dataset of string pairs is first read, preprocessed, tokenized, and they are converted into dense vectors (i.e., two embeddings per

[3] https://pypi.org/project/DeezyMatch
[4] https://github.com/Living-with-machines/DeezyMatch

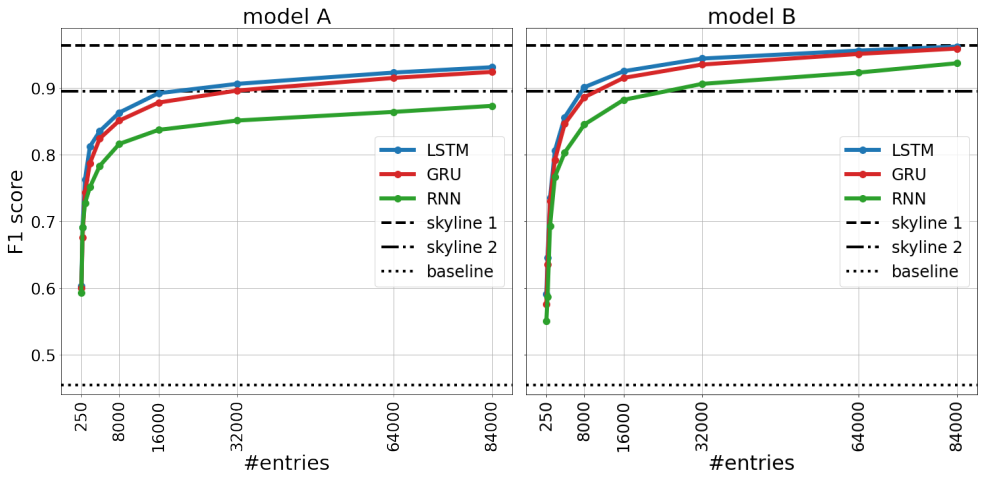

Figure 2: Impact of fine-tuning and freezing neural network layers on the performance of *pair classifier* as measured by F1-score. Three neural network architectures (LSTM, GRU and RNN) are fine-tuned and compared as a function of data instances (x-axis) used in fine-tuning. In model A, only the last layer (fully-connected layers in Fig. 1) is fine-tuned while in model B, both the recurrent units and the fully-connected layers are used. By adding more data instances in fine-tuning, the performance of all models improve logarithmically.

pair); the dimensionality of which can be specified in the input. The two embedding vectors of a string pair are then fed into two parallel recurrent units to generate vector representations (i.e., hidden states of the last units in each direction and layer). Next, the two vectors can be combined in different ways specified in the input, e.g., via concatenation, element-wise product, difference, or a combination of these. This aggregated representation is then given as input to a feed-forward network with one hidden layer and with Rectified Linear Unit (ReLU) as the activation function. The output layer has one unit with a sigmoid activation function for producing the final prediction. During training, the target and the predicted outputs are compared by the Binary Cross Entropy criterion. The dimensionality of the hidden layer and other hyperparameters (e.g., learning rate, number of epochs, batch size, early stopping and dropout probability) can all be tuned in the input file. DeezyMatch logs and outputs all standard evaluation metrics for binary classification (accuracy, precision, recall and F1) during training, evaluation and testing. Similar to Tam et al. (2019), it also calculates mean average precision (MAP), which evaluates the quality of candidate ranks per query. After a training is finished, DeezyMatch can be used to plot loss and evaluation metrics at each epoch for model selection. The outputs of each epoch can be also visualized during training via TensorBoard (Abadi et al., 2016).

2.1.1 Transfer learning

In addition to training a model from scratch, DeezyMatch supports fine-tuning a pretrained model; this way, an already trained model on a large dataset can be fine-tuned to a new domain. This transfer learning approach helps especially where only limited training examples are available. Any learnable parameters (as highlighted by blue boxes in Fig. 1) can be "frozen" during fine-tuning, and the fine-tuning can be done on a specified number of training instances by the user.

Fig. 2 shows the results of two sets of models fine-tuned progressively on more training instances. In model A, both embedding and recurrent units are frozen (i.e., their parameters are not updated during fine-tuning), and in model B, only the embedding layer is frozen. The baseline, skyline 1 and 2 are trained on *WG:en*, *OCR* and *WG:en+OCR*, respectively. Refer to Section 3.1.1 for details on these datasets. The performance of these models is then assessed on the *OCR* test set. To show the impact of fine-tuning and choice of architecture on the model performance, we trained various models starting with the baseline model and included more training instances from the training set of *OCR*. In this experiment, only ≈8K data points were needed to improve the performance of all models from ≈0.45 (baseline) to ≈0.82. In model B, by using around 20% of the data points (≈16K), the performance of GRU and LSTM architectures improve to ≈0.92 which highlights the importance of fine-tuning in scenarios with limited training datasets. When in-

cluding all the data points, all models, except RNN in model A, pass skyline-2, and two of them reach the performance of skyline-1 (≈0.964). It is worth noting that model B shows better performance compared to model A in fine-tuning. The improved performance can be attributed to the more unfrozen parameters during fine-tuning, which increases the learning capacity.

2.2 Candidate Ranker

The trained pair-classifiers in Section 2.1 predict if an input string pair is a good match or not by providing not only the label (True/False) but also the confidence of the model on each label. The same models can then be used for the task of candidate ranking. First, a trained DeezyMatch model is used to generate vector representations for all known variations of entity names in a KB (i.e. "all candidate mentions" in Fig. 1). These vector representations are extracted from the recurrent units for each direction and layer. This step is done only once for a given model and KB. The vectors (e.g. forward/backward vectors in a bi-directional recurrent network) are then assembled to form one file containing all the vector representations for unique candidate mentions. Next, given a query (i.e. a mention of an entity as a string), the same DeezyMatch model generates its vector representation similar to the previous step. At the final stage, the query vectors are compared with candidate vectors using a metric specified by the user. The choices of this metric are the DeezyMatch prediction scores, L_2-norm distances (as implemented in the *faiss* library of Johnson et al. (2019)) or cosine similarities between the query and candidate vectors. Based on the selected method and for a given query, DeezyMatch ranks the results and outputs the best matching candidates (the number of which can be specified by the user).

An advantage of the proposed method is that vector representations for the KB are computed only once (for a given trained model). For all subsequent queries, only the query vectors are generated and compared to the KB vectors. This significantly reduces the computation time compared to more traditional methods (e.g. Levenshtein distance) in which one query is compared to n possible variations of all potential candidates in each run.

When the selected ranking metric is L_2-norm distance or cosine similarity, the above procedure can be done efficiently using generated matrices (i.e., assembled vector representations) and available linear algebra packages. However, model inference on large datasets can be prohibitively expensive. In DeezyMatch, we developed an adaptive method to avoid the search of whole KB for a given query. We start with the query vector and find a set of "close" candidate vectors as measured by the L_2-norm distance (i.e., two vectors are similar when the distance is low). We then perform model inference only on these candidates. If the number of desired candidates (specified by the user) is reached, DeezyMatch goes to the next query mention. Otherwise, it expands the search space by a user-specified search size and repeats the model inference on new instances. This procedure continues until the number of desired candidates is reached or all candidates in the KB are tested. In our experiments in Section 3.1.2, this adaptive procedure significantly reduces the computation time of similarity search in large datasets.

2.3 DeezyMatch interface

DeezyMatch is available as a Python library and can be used as a stand-alone command-line tool or as a module in existing Python NLP pipelines. As an example, the training and inference steps described in Section 2.1 can be executed by:

```
from DeezyMatch import train
from DeezyMatch import inference

# train a new model
train(input_file_path,
      dataset_train_path,
      model_name)

# model inference
inference(input_file_path,
          dataset_inference_path,
          pretrained_model_path)
```

Other functionalities, such as fine-tuning and candidate ranking, have similar easy-to-use interfaces. Consult DeezyMatch's GitHub page for additional information and examples.

3 Comparison with existing systems

The majority of readily available EL tools rely on a lookup table or on shallow string similarity approaches to select an initial set of candidates, followed by a disambiguation step. TagMe! (Ferragina and Scaiella, 2010), for instance, a well established EL baseline, performs candidate selection through perfect matches between mentions and a list of alias surface forms derived from Wikipedia, as also discussed by Hasibi et al. (2016).

Alternatives to perfect matches involve the adoption of edit-distance techniques, such as Levenshtein distance (see, for example, its adoption in McNamee et al. (2011a); Moreno et al. (2017)). While there are many implementations of such approaches readily available, these methods suffer from poor scalability (i.e., time complexity, as we discuss in our experiments in Section 3.1.1). Due to this, some EL pipelines (e.g., Greenfield et al. (2016)) have incorporated such techniques only when no exact matching entry can be retrieved.

More recently, researchers have developed deep learning solutions for candidate selection. Le and Titov (2019) framed it as a distance learning task with a noise detector in their EL system, in which the linkage between mentions that are not necessarily in the KB is learned from lists of positive candidates (the top matching candidates) and negative candidates (randomly sampled from the KB). Tam et al. (2019) have recently presented STANCE, a model for computing the similarity of two strings by encoding the characters of each of them, aligning the encodings using Sinkhorn Iteration, and scoring the alignment with a convolutional neural network. The associated repository[5] offers codes for reproducing the experiments in the paper. Unfortunately, their implementation is not directly comparable with DeezyMatch, as it was not designed to be integrated directly into an EL pipeline.

The work closest to ours, which has directly inspired our initial development, is by Santos et al. (2018a). The authors presented a recurrent neural network architecture to encode pairs of toponyms followed by a multi-layer perceptron to determine if they are matching. The authors accompanied their work with a repository to reproduce the results presented in the paper.[6] However, the user has little control over the model architecture, including its hyperparameters and processing steps. Moreover, the authors do not offer a method for either loading a trained model and applying it to new data or for candidate ranking.

Building upon this previous work, we present an easy to use library that *(a)* relies on deep neural networks for fuzzy string matching and candidate ranking beyond surface similarities; *(b)* is significantly faster than edit-distance approaches; and *(c)* can be seamlessly integrated into existing EL pipelines with a single Python command.

[5] https://github.com/iesl/stance

[6] https://github.com/ruipds/Toponym-Matching

3.1 Performance

We test DeezyMatch in the context of geographical candidate selection, the task of identifying potential entities that can be referred to by a toponym (i.e., a place name). This can be understood as the middle step between named entity recognition (in this case, toponym recognition) and the downstream task of EL (in this case, toponym resolution). See Coll Ardanuy et al. (2020) for a detailed description of the datasets and KBs, experimental settings, and analysis of the results reported in Sections 3.1.1 and 3.1.2. Evaluation of the impact of transfer learning and domain adaptation (as described in Section 2.1.1) on candidate ranking will be the subject of future work.

3.1.1 Pair classifier

We compare our method to Santos et al. (2018a) and normalized Levenshtein Damerau edit distance (henceforth *LevDam*)[7] on three datasets of positive and negative string pairs. The datasets against which we compare the three methods are *Santos* (∼4.3M toponym pairs in different alphabets derived from GeoNames[8] and introduced in Santos et al. (2018b)); *WG:en* (∼670K toponym pairs derived from the English version of WikiGazetteer; see Coll Ardanuy et al. (2019)); and *OCR* (∼93K named entity pairs derived from OCR'd text aligned to its human correction).

All datasets are balanced and contain an equal number of positive and negative pairs per query. In all cases, negative examples have been constructed carefully to capture both trivial and challenging transformations. Table 1 reports the F-Score of the three methods on the three datasets. Both for *LevDam* and *DeezyMatch*, we have left out 10% of each dataset for testing, whereas for Santos et al. (2018a), we show an F-Score obtained through two-fold cross validation (the setting allowed by the implementation). The DeezyMatch models used in the experiments have similar architecture and hyperparameters.[9]

3.1.2 Candidate ranker

We evaluate the performance of DeezyMatch's *candidate ranker* in a real-case toponym resolution application by assessing the quality of ranked candidates and its computation time on three datasets:

[7] https://pypi.org/project/pyxDamerauLevenshtein

[8] https://www.geonames.org

[9] Refer to the *DeezyMatch Models* section in Coll Ardanuy et al. (2020) for the choice of hyperparameters.

	Santos	WG:en	OCR
LevDam	0.70	0.74	0.76
Santos et al. (2018a)	0.82	0.92	0.95
DeezyMatch	0.89	0.94	0.95

Table 1: DeezyMatch's *pair-classifier* performance as measured by F-score compared with other methods.

	P@1	MAP@10	MAP@20	T/q
ArgM:exact	0.69	-	-	-
ArgM:LD	0.78	0.72	0.70	9s
ArgM:DM	0.78	0.76	0.74	0.3s
WOTR:exact	0.86	-	-	-
WOTR:LD	0.92	0.84	0.80	31.6s
WOTR:DM	0.93	0.90	0.87	0.7s
FMP:exact	0.77	-	-	-
FMP:LD	0.92	0.82	0.76	14.1s
FMP:DM	0.85	0.82	0.78	0.7s

Table 2: DeezyMatch (*DM*) *candidate ranker* performance on three datasets compared to two other methods: LevDam (*LD*) and *exact*. *T/q* indicates 'Time per query' on CPU.

(1) *ArgManuscrita* (*ArgM*), a toponym-resolved dataset in Spanish created from a seventeenth-century travelogue and composed of 799 toponyms (of which 200 are unique after lower-casing); (2) *WOTR*, an OCR-corrected dataset of letters and reports in English from 1860s, of which we used its test set. It contains 1,479 toponyms manually annotated with their resolved coordinates (of which 584 unique toponyms, after lower-casing); and (3) *BNA-FMP*, a dataset of digitized nineteenth-century newspaper articles in English with 1,248 toponyms already recognized and resolved to their correct geographic coordinates (of which 509 unique toponyms, after lower-casing), containing several toponyms with OCR errors, such as *'DORSETSIIIRR'* for *'Dorsetshire'*.

As KBs, we used the English version of WikiGazetteer (with 2,455,966 candidate mentions) for *WOTR* and *BNA-FMP*; and the Spanish version of WikiGazetteer combined with the *HGIS de las Indias* gazetteer (Stangl, 2018) (with 556,985 candidate mentions) for *ArgManuscrita*. We considered that a retrieved candidate mention correctly matched a query if it could refer to an entity in our KB within 10 km of the coordinates in the gold standard.[10] In Table 2, we compare DeezyMatch with two baselines: *exact* selects the exact-matching candidate from the KB, which is the most common approach in EL systems, and *LevDam* ranks candidates according to the normalized Levenshtein-Damerau edit distance, traditionally considered a strong baseline. The advantage of using DeezyMatch is stressed both in terms of mean average precision (at 10 and 20 candidates) and especially by its reduced computation time in comparison with LevDam.[11]

4 Conclusions

We presented DeezyMatch, a new user-friendly Python library for fuzzy string matching and candidate ranking, based on deep neural network architectures. DeezyMatch can be seamlessly integrated into existing EL pipelines. Its flexibility allows the user to easily fine-tune a pretrained model or to adapt the model architecture to the specificity of a real-case scenario. We compared its design, implementation and functionalities with other approaches. In the future, we plan to support self-attention and state-of-the-art pretrained character-based models, integrate learning to rank functionalities in the candidate selection process and to release a zoo of models trained on large datasets which can be fine-tuned further in other downstream NLP tasks.

DeezyMatch was designed with flexibility in mind, and we encourage the community to further extend its implementation for addressing other related tasks, such as record linkage, transliteration and data integration.

Acknowledgments

We thank Katherine McDonough and the anonymous reviewers for their careful and constructive reviews. This work was supported by Living with Machines (AHRC grant AH/S01179X/1) and The Alan Turing Institute (EPSRC grant EP/N510129/1).

[10] Due to a lack of a true gold standard of coordinates for locations, allowing an error distance is common practice in evaluating toponym resolution systems (DeLozier et al., 2015; Roller et al., 2012; Speriosu and Baldridge, 2013). Additionally, due to this, we skip toponyms if no correct match has been found by all the tested methods.

[11] Here, we do not take into account the time spent on training the model and generating the candidate vectors, as this is done offline only once and can be reused for all following candidate ranking tasks that use the same model and gazetteer. Training a model on a dataset with ≈670K pairs (including preprocessing) takes 28m on GPU and 54m on CPU, while generating and combining candidate vectors takes 39m on GPU and 204m on CPU.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Mariona Coll Ardanuy, Kasra Hosseini, Katherine McDonough, Amrey Krause, Daniel van Strien, and Federico Nanni. 2020. Deezymatch: A deep learning approach to geographical candidate selection through toponym matching. In *SIGSPATIAL: Poster Paper*.

Mariona Coll Ardanuy, Katherine McDonough, Amrey Krause, Daniel CS Wilson, Kasra Hosseini, and Daniel van Strien. 2019. Resolving places, past and present: toponym resolution in historical british newspapers using multiple resources. In *Proc. of GIR*.

Grant DeLozier, Jason Baldridge, and Loretta London. 2015. Gazetteer-independent toponym resolution using geographic word profiles. In *Proc. of AAAI*.

Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science*, 14(2):179–211.

Paolo Ferragina and Ugo Scaiella. 2010. Tagme: on-the-fly annotation of short text fragments (by wikipedia entities). In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 1625–1628.

Kara Greenfield, Rajmonda S Caceres, Michael Coury, Kelly Geyer, Youngjune Gwon, Jason Matterer, Alyssa C Mensch, Cem Safak Sahin, and Olga Simek. 2016. A reverse approach to named entity extraction and linking in microposts. In *# Microposts*, pages 67–69.

Ben Hachey, Will Radford, Joel Nothman, Matthew Honnibal, and James R Curran. 2013. Evaluating entity linking with wikipedia. *Artificial intelligence*.

Faegheh Hasibi, Krisztian Balog, and Svein Erik Bratsberg. 2016. On the reproducibility of the tagme entity linking system. In *European Conference on Information Retrieval*, pages 436–449. Springer.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Jeff Johnson, Matthijs Douze, and Herve Jegou. 2019. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*.

Phong Le and Ivan Titov. 2019. Distant learning for entity linking with automatic noise detection. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4081–4090, Florence, Italy. Association for Computational Linguistics.

Katherine McDonough, Ludovic Moncla, and Matje van de Camp. 2019. Named entity recognition goes to old regime france: geographic text analysis for early modern french corpora. *International Journal of Geographical Information Science*, 33(12):2498–2522.

Paul McNamee, James Mayfield, Dawn Lawrie, Douglas Oard, and David Doermann. 2011a. Cross-language entity linking. In *Proceedings of 5th International Joint Conference on Natural Language Processing*, pages 255–263, Chiang Mai, Thailand. Asian Federation of Natural Language Processing.

Paul McNamee, James Mayfield, Dawn Lawrie, Douglas W Oard, and David Doermann. 2011b. Cross-language entity linking. In *Proceedings of 5th International Joint Conference on Natural Language Processing*, pages 255–263.

Pablo N Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer. 2011. Dbpedia spotlight: shedding light on the web of documents. In *Proceedings of the 7th international conference on semantic systems*, pages 1–8.

Jose G Moreno, Romaric Besançon, Romain Beaumont, Eva D'hondt, Anne-Laure Ligozat, Sophie Rosset, Xavier Tannier, and Brigitte Grau. 2017. Combining word and entity embeddings for entity linking. In *European Semantic Web Conference*, pages 337–352. Springer.

Andrea Moro, Alessandro Raganato, and Roberto Navigli. 2014. Entity linking meets word sense disambiguation: a unified approach. *Transactions of the Association for Computational Linguistics*, 2:231–244.

Gary Munnelly and Séamus Lawless. 2018. Investigating entity linking in early english legal documents. In *Proceedings of the 18th ACM/IEEE on Joint Conference on Digital Libraries*, pages 59–68.

Alex Olieman, Kaspar Beelen, Milan van Lange, Jaap Kamps, and Maarten Marx. 2017. Good applications for crummy entity linkers? the case of corpus selection in digital humanities. In *Proceedings of the 13th International Conference on Semantic Systems*, pages 81–88.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

Julien Plu, Giuseppe Rizzo, and Raphaël Troncy. 2016. Enhancing entity linking by combining ner models. In *Semantic Web Evaluation Challenge*, pages 17–32. Springer.

Gianluca Quercini, Hanan Samet, Jagan Sankaranarayanan, and Michael D Lieberman. 2010. Determining the spatial reader scopes of news sources using local lexicons. In *Proc. of SIGSPATIAL*.

Jonathan Raphael Raiman and Olivier Michel Raiman. 2018. Deeptype: multilingual entity linking by neural type system evolution. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Stephen Roller, Michael Speriosu, Sarat Rallapalli, Benjamin Wing, and Jason Baldridge. 2012. Supervised text-based geolocation using language models on an adaptive grid. In *Proc. of EMNLP*.

Marco Rovera, Federico Nanni, Simone Paolo Ponzetto, and Anna Goy. 2017. Domain-specific named entity disambiguation in historical memoirs. *CLiC-it 2017 11-12 December 2017, Rome*, page 287.

Rui Santos, Patricia Murrieta-Flores, Pável Calado, and Bruno Martins. 2018a. Toponym matching through deep neural networks. *International Journal of Geographical Information Science*.

Rui Santos, Patricia Murrieta-Flores, and Bruno Martins. 2018b. Learning to combine multiple string similarity metrics for effective toponym matching. *International journal of digital earth*.

Avirup Sil, Gourab Kundu, Radu Florian, and Wael Hamza. 2018. Neural cross-lingual entity linking. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Michael Speriosu and Jason Baldridge. 2013. Text-driven toponym resolution using indirect supervision. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1466–1476, Sofia, Bulgaria. Association for Computational Linguistics.

Werner Stangl. 2018. 'the empire strikes back'?: Hgis de las indias and the postcolonial death star. *IJHAC*.

Derek Tam, Nicholas Monath, Ari Kobren, Aaron Traylor, Rajarshi Das, and Andrew McCallum. 2019. Optimal transport-based alignment of learned character representations for string similarity. *arXiv preprint arXiv:1907.10165*.