Shortest-Paths with Text Embeddings on Simple English Wikipedia

Ammar Ratnani
CS 229: Machine Learning
Stanford University
aratnani@stanford.edu

1 Introduction

Wikiracer is a folk game similar to *Six Degrees to Kevin Bacon*, but played on Wikipedia articles. In it, players are given a source and target article, and they must traverse links to get from the source to the target in as few hops as possible. Standard shortest-path algorithms can find an optimal solution for a given source-target pair in just a few seconds (Wenger, 2024; Hernandez, 2024), but this approach explores many unnecessary nodes. Ideally, the search would be guided by a heuristic, and the text of each article could be a firm base on which to build it; intuitively, the content of an article should predict the kinds of articles it is close to. Furthermore, recent advances in natural language processing have constructed vector embeddings that can encode the content of a large piece of text for downstream tasks. Hence, this project aims to use the text embeddings of the articles on Simple English Wikipedia to learn a shortest-path heuristic on its link graph.

2 Related Work

In the field of machine learning on graphs, the most similar prior work builds on the idea of node embeddings, such *node2vec* (Grover and Leskovec, 2016) or *DeepWalk* (Perozzi et al., 2014). They use *word2vec* (Mikolov et al., 2013) to embed each node in a way that encodes its structural characteristics. One task Grover and Leskovec (2016) evaluate these embeddings on is link prediction, which is a special case of what this project aims to do. Rizi et al. (2018) extend to the general case of distance estimation, training a regression model to directly output the distance between two nodes given their embeddings. Qi et al. (2020) also train an embedding-based model for distance estimation, but they train the embeddings in conjunction with the model instead of using *node2vec*. Outside of embedding-based approaches, Park et al. (2024) use graph convolutional networks to construct a shortest-path tree. Many classical methods to approximate the shortest-path length also exist, such as the works of Zhao and Zheng (2010), Gubichev et al. (2010), and Potamias et al. (2009).

Swafford and Barron (2015) also built a shortest-path heuristic for Simple English Wikipedia as their final project for CS 229 in 2015. The author was not aware of this when they decided on their project. However, they still believe this work is unique enough to merit its own report. For example, the previous project fixes the target at "Stanford University" and tries to predict the distance to it, while this project aims to develop a heuristic between all pairs of nodes. More importantly, the previous project came before the advent of Large Language Models. It models each document as a bag of words, while this project can take advantage of deep text embeddings. Nonetheless, the author thanks Swafford and Barron (2015) for providing a starting-point for this work.

3 Data Collection and Analysis

The main data source for this project is The Wikimedia Foundation (2024), which provides periodic database dumps for all of its wikis. Simple English Wikipedia is used instead of English Wikipedia primarily because it is 1.4 orders of magnitude smaller, with $260\,\mathrm{k}$ articles and $12\,\mathrm{M}$ links. Its link graph is filtered to only include the Main Namespace, as those articles directly contribute to the "content" of the wiki (Meta, 2024). Importantly, paths that start and end in the Main Namespace, but which temporarily go outside of it — even for a redirect, are not considered.

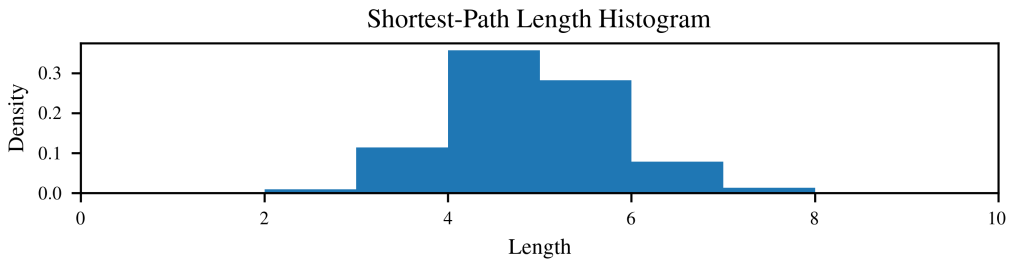

Figure 1: Distribution of shortest-path lengths in the link graph of Simple English Wikipedia, generated as in Sec. 5.1 by BFS. The $14.04\,\%$ of unconnected node pairs are not shown.

To construct the embeddings, the XML file containing the text and metadata for all articles is iterated over. Each non-redirect article in the Main Namespace is parsed with `mwparserfromhell` (Kurtovic, 2024) to strip away the templates. The results are fed into `text-embedding-small-3` (OpenAI, Inc., 2024) after being truncated to the maximum length. The output of the model is a 1536-dimensional unit vector, but it uses Matryoshka Representation Learning (Kusupati et al., 2022). For this project, that means the embeddings can be truncated and renormalized to reduce compute time while retaining predictive power. In fact, the full embeddings are almost never used here. As for the link graph, the `page`, `pagelinks`, `linktargets`, and `redirect` (MediaWiki, 2024) table dumps are downloaded and imported into MySQL. Pages outside of the Main Namespace are deleted, along with any now-broken links and redirects. After that, the data is imported into Memgraph, all redirects are collapsed into direct links, and the now-redundant redirect pages are deleted.

At this point, Exploratory Data Analysis (EDA) is conducted. One significant output is the distribution of shortest-path lengths. Samples are taken using BFS as described in Sec. 5.1, and the results are shown in Fig. 1. The graph's diameter is short, with most shortest-paths having six or fewer hops. Additionally, most nodes reach close to $86.4\,\%$ of the graph, suggesting it has one large strongly-connected component that most articles immediately link into. More importantly for training, this sampling technique produces wildly imbalanced "distance classes" by default, which must be compensated for. Another significant artifact of EDA is the distribution that nodes are selected according to. Swafford and Barron (2015) select nodes uniformly at random, but this approach tends to select low-quality articles. Instead, the distribution produced by *PageRank* (Page, 1999) is used, with this choice justified since at least one implementation of *Wikiracer* uses it too (Qian et al., 2024). A damping factor of $d = 80\,\%$ is used for no reason other than the fact the median of this distribution is around $10\,000$ articles. It is also close to the standard damping factor of $85\,\%$.

4 Link Prediction Model

This project proceeds in three phases. The first phase is building a model for link prediction: given the embedding vectors of the source and target articles, predict whether there exists a hyperlink from the source to the target. This task is a special case of distance estimation — just predict whether the distance is one or whether it is two or more. Additionally, Grover and Leskovec (2016) provide a baseline against which to compare. For those reasons, this is selected as the first phase.

4.1 Methods

An equal number of positive and negative samples are collected. Positive samples are taken uniformly without replacement from the graph's edge set, while negative samples are generated by selecting pairs of nodes uniformly[1] and filtering out those connected by an edge. In total $200\,\mathrm{k}$ samples are collected, and an 80-10-10 split is used. A logistic regression model under binary cross-entropy (described in Appendix F) is built in Tensorflow (Abadi et al., 2015) and Keras (Chollet et al., 2015). It is optimized with Adam (described in Appendix I) with default hyperparameters and learning rate. The model is trained for 2000 epochs, and the parameters with the best validation loss are kept. Finally, early stopping is used with a minimum change of 1.0×10^{-6} and a patience of 10 epochs.

[1]This is done with replacement, but that is an oversight. They should be sampled without replacement.

Table 1: Link-prediction performance by combination operator

Operator	Test AUC (%)	Loss		Accuracy (%)	
		Train	*Test*	*Train*	*Test*
Concatenation	85.34	0.4746	0.4831	77.45	77.35
Element-wise Product	97.78	0.1834	0.1835	92.60	92.92
Outer Product	98.96	0.0554	0.1298	98.40	95.23

The model is evaluated with different amounts of training data and with different text embedding lengths. Additionally, there is a choice of the combination operator K used to combine the nodes' embeddings before feeding them into the model. Concatenation $K(s,t) = s \parallel t$, the element-wise product $K(s,t) = s \odot t$, and the outer product $K(s,t) = s \cdot t^\intercal$ are evaluated.

4.2 Results

The results for the different combination operators are given in Table 1. Those metrics were collected with 256-dimensional embeddings over the entire dataset. It seems that somehow multiplying the elements of the input vectors is needed to achieve good performance, which makes sense given this is a shallow model and that this multiplication is similar to cosine similarity (Salahi, 2024) (described in Appendix D). A linear function of the concatenated vectors is not sufficient to express this multiplication, and the operator's performance suffers. The outer product performs best — see Appendix C for an explanation, but the quadratic number of features causes it to overfit. It gets worse on smaller datasets, with the outer product getting a $99.99\,\%$ training accuracy with $20\,\mathrm{k}$ samples.

Fixing the combination operator to element-wise multiplication, the model's performance increases slightly as either the number of training examples or the embedding dimension increase. Reducing the number of samples down to $2\,\mathrm{k}$ causes an Area-Under-the-Curve (AUC) degredation on the test set of just two percentage points, though test accuracy drops by eight. For embedding length, larger gains are observed on larger datasets. On the largest dataset, increasing the dimension from 256 to 1536 gives a one point improvement in test AUC. On smaller datasets, a plateau is observed at 512.

Finally, these results are comparable to the best ones given by Grover and Leskovec (2016). This performance demonstrates that this project's approach could work, which is the aim of this phase.

5 Distance Estimation Model

The second phase of this project is constructing a model for distance estimation: given the embedding vectors of the source and target nodes, predict the length of the shortest path between them. The source and target nodes are assumed to be distinct, since it is trivial to add a check for that case.

5.1 Methods

A set of $3\,\mathrm{k}$ landmark nodes are selected according to PageRank$(d = 80\,\%)$ without replacement. Breadth-First Search (BFS) is run from each landmark to every other node on the directed link graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. All the distances obtained are added to the dataset, meaning each landmark contributes $|\mathcal{V}| - 1$ pairs. An 85-10-5 split is used on the landmarks — i.e. a single landmark's data is never split between sets. Additionally, $2\,\mathrm{M}$ random edges are selected uniformly without replacement and added to the dataset as node pairs with distance one. They are also split 85-10-5. Finally, a maximum distance M is imposed on the samples. Pairs whose distance equals or exceeds M are treated as unconnected. A value of $M := 4$ is used in the default configuration, informed by Fig. 1.

The model is built in PyTorch (Paszke et al., 2019), and its detailed architecture is described in Appendix J.1. Very briefly, it is a multi-layer perceptron with residual connections. It is trained under a variant of mean squared-error loss, described in Appendix J.2. The loss is adjusted to not punish the model for predicting too low a distance for nodes that are as close as possible, and likewise to not punish it for too high a distance for nodes that have no path between them. Additionally, the learning rate is kept relatively high until the last five epochs, at which point it is reduced by a factor of ten. This is observed to allow training on higher learning rates while not sacrificing validation loss.

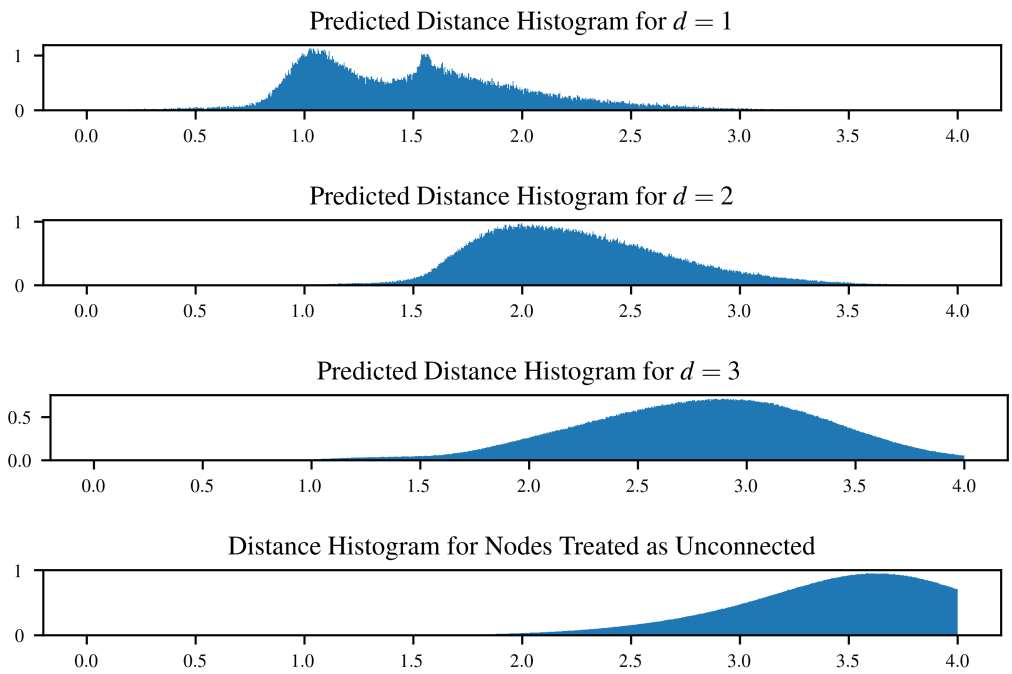

Figure 2: Histograms of the predicted distance $\hat{d}$ of node pairs for different true distances d over the test set, with the model in its default configuration.

5.2 Results

On the test set, the default model gets a Mean Absolute Error (MAE) of 0.472 hops, with a Mean Relative Error (MRE) of $30.2\,\%$. Both are weighted due to class imbalance, and both are taken over nodes which are treated as connected. The Mean Unconnected Prediction (MUP) is 3.54 hops out of the best possible 4. Histograms of the predicted distances are given in Fig. 2. In general, the model separates the classes passably, though its performance degrades as the nodes get farther apart. Lastly with regards to overfitting, the training, validation, and test losses are all within $5\,\%$ of each other[2].

A linear regression model (described in Appendix E) with concatenation as the combination operator is trained as a baseline[3]. It performs poorly compared to the default model. Its test loss is 2.91x higher, and increases of similar magnitude are observed in its MAE and MRE, at 1.65x and 1.94x respectively. Its MUP is 2.86 hops. Looking at the predicted distance histograms, given in Fig. 3 of Appendix A, it is apparent that this baseline has a hard time using the input features to discriminate between the distance classes, and it instead focuses on the average case.

Finally, this model is compared to prior work by Rizi et al. (2018) and Qi et al. (2020). It performs poorly, getting an unweighted MRE of $17.02\,\%$ — triple the $5\,\%$ and $6\,\%$ figures attained in the respective papers. It is not clear what is causing this, particularly since this project's methodology is similar to that of Rizi et al. (2018) in a similar setting. One possibility is that prior work operates on undirected connected graphs, while the link graph of Simple English Wikipedia $\mathcal{G}$ is neither.

6 A* Evaluation

The third phase of this project evaluates the models made in Sec. 5 on the task they were built for: running A*. Specifically, this phase runs A* from a source to a target, using the model under test to estimate the distance from every intermediate node to the target given their embeddings.

[2]This measurement is taken with each dataset truncated to the length of the smallest one. That is because it is observed that the number of samples affects the calculated loss by affecting the class weights.

[3]Surprisingly, element-wise multiplication performs worse on all metrics, despite its showing in Sec. 4.

Table 2: A* performance of different heuristics

Model		Time (s)	Nodes Expanded	Path Length	Test Loss
Hidden Length	*Max. Distance*				
	Null Heuristic	0.0249	17 718	3.5208	—
	Linear Baseline	0.6315	15 500	3.5313	1.6922
192	4	0.3648	3668	3.5396	0.5820
	5	0.3540	3494	3.5521	0.7793
	6	0.3788	3865	3.5604	1.0322
128	4	0.4164	4222	3.5313	0.6691
	5	0.3836	4124	3.5458	0.8734
	6	0.4402	4444	3.5604	1.2099

6.1 Methods

Each model is evaluated on the same "validation set" of 500 trials. For each trial, the source and target nodes are selected independently distributed as PageRank($d = 80\,\%$). The A* algorithm is run for each pair of nodes. The primary figure of merit is the number of nodes expanded in the main loop of the algorithm, though the path length expansion compared to optimal is also measured. The baseline is taken to be the "Null" heuristic, which outputs zero if the queried node is equal to the target and one otherwise. Finally, the check mentioned in Sec. 5 is implemented.

6.2 Results

All values in Table 2 are taken as a mean over the validation set, and node pairs with no path between them are ignored, since no heuristic can provide a benefit in that case. Table 3 in Appendix B includes those pairs. The default configuration achieves an $79.3\,\%$ reduction in the number of nodes expanded, while experiencing a negligible $0.53\,\%$ path length expansion. Unfortunately, that reduction does not translate into a quicker runtime, and in fact it runs 15x slower.

Since it cannot be tuned, different values of the maximum distance are evaluated directly on A*, with the different M chosen to cover Fig. 1. In general, increasing the maximum distance causes an increase in path length expansion, though the effect is very small — at most $1.12\,\%$. A different effect is observed on the number of nodes expanded, which indirectly affects the runtime. It seems a maximum distance of 5 works best. Presumably, $M = 4$ does not look far enough ahead to cull bad routes since it is too far left of the peak in Fig. 1, while setting M to large causes the model to underperform since it just has to do more to discriminate between the different distance classes.

It is also not clear *a priori* how well a lower loss translates to performance on A*, so smaller models with 128-dimensional hidden layers are evaluated. A more detailed summary is Table 5 in Appendix B. In general, the smaller heuristics attain around a $15\,\%$ higher test loss, and see a similar increase in the number of nodes expanded. The time taken also increases by between $8\,\%$ and $16\,\%$, despite the heuristic being $40\,\%$ cheaper to evaluate in theory.

Finally, a "test set" of 500 trials is collected just like the validation set, and the best configuration — hidden length of 192 and maximum distance of 5 — is run on it. It averages $0.342\,\mathrm{s}$ per sample, expanding 3172 nodes and getting a path length of 3.576, again ignoring unconnected pairs.

7 Conclusion

This project shows that text embeddings can encode information about article relationships. That information can be used to make a heuristic that explores far fewer nodes than a naïve search. Further gains should be possible, but evaluating the proposed heuristic is very expensive. A next step could be to take the text embeddings over article summaries, which would isolate whether the model performs better on adjacent articles because of memorization. Another step could be to experiment with different learning rate schedules, instead of the ad-hoc one given here. Finally, performance gains could be achieved by running both the search and the heuristic on the GPU.

8 Contributions

This project was done individually, so Ammar Ratnani contributed all the work.

Code for this project is available at `https://github.com/ammrat13/wikiracer-embeddings`. Additionally, `https://github.com/LaurentMazare/tch-rs/pull/913` came out of this work.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Neil Band. 2024. Personal communication. Is a CS 229 course assistant.

François Chollet et al. 2015. Keras. `https://keras.io`.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159.

Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks.

Andrey Gubichev, Srikanta Bedathur, Stephan Seufert, and Gerhard Weikum. 2010. Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, page 499–508, New York, NY, USA. Association for Computing Machinery.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition.

Raul Hernandez. 2024. WikiGameSolver.

Geoff Hinton. 2014. RMSProp. From Coursera.

Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, Xinrong Zhang, Zheng Leng Thai, Kaihuo Zhang, Chongyi Wang, Yuan Yao, Chenyang Zhao, Jie Zhou, Jie Cai, Zhongwu Zhai, Ning Ding, Chao Jia, Guoyang Zeng, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Ben Kurtovic. 2024. Mediawiki parser from hell.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. 2022. Matryoshka representation learning. In *Advances in Neural Information Processing Systems*.

MediaWiki. 2024. Manual:Contents — MediaWiki. [Online; accessed 7-November-2024].

Meta. 2024. Help: Namespaces. [Online; accessed 6-November-2024].

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space.

OpenAI, Inc. 2024. Vector embeddings: Learn how to turn text into numbers, unlocking use cases like search. [Online; accessed 7-November-2024].

Lawrence Page. 1999. The PageRank citation ranking: Bringing order to the web. Technical report, Technical Report.

Jisang Park, Sukmin Kang, Van-Vi Vo, and Hyunseung Choo. 2024. Efficient shortest-path tree construction based on graph convolutional networks. In *2024 18th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–4.

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*, volume 32. Curran Associates, Inc.

Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '14, page 701–710. ACM.

Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM '09, page 867–876, New York, NY, USA. Association for Computing Machinery.

Jianzhong Qi, Wei Wang, and Rui Zhangand Zhuowei Zhao. 2020. A learning based approach to predict shortest-path distances. In *EDBT/ICDT 2020 Joint Conference*, pages 367–370. EDBT.

Daniel Qian, Brice Halder, and Miles Liu. 2024. Competitive Wikipedia speedrunning.

Fatemeh Salehi Rizi, Joerg Schloetterer, and Michael Granitzer. 2018. Shortest path distance approximation using deep learning techniques. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, page 1007–1014. IEEE.

Kamyar Salahi. 2024. Personal communication. Is a CS 229 course assistant.

Zack Swafford and Alex Barron. 2015. Building an AI for the Wikipedia game. In *CS 229 Final Projects*. CS 229: Machine Learning.

The Wikimedia Foundation. 2024. Wikimedia downloads. [Online; accessed 7-November-2024].

Jacob Wenger. 2024. Six degrees of Wikipedia.

Xiaohan Zhao and Haitao Zheng. 2010. Orion: Shortest path estimation for large social graphs. In *3rd Workshop on Online Social Networks (WOSN 2010)*, Boston, MA. USENIX Association.

A Auxiliary Data for the Distance Estimation Model

This appendix contains data for Sec. 5 that was interesting but not deemed important enough to include with the main text. Fig. 3 shows the distributions of predicted distances for the different distance classes by the linear regression baseline mentioned in Sec. 5.2.

B Auxiliary Data for the A* Evaluation

This appendix contains data for Sec. 6 that was interesting but not deemed important enough to include with the main text. As mentioned in Sec. 6.2, Table 2 does not include data for samples with no path between them. Table 3 contains the same metrics as Table 2 but with those samples included. Sec. 6.2 also analyzes the performance of models with 128-dimensional hidden layers compared to those with 192-dimensions. While the data can be calculated from Tables 2 and 3, precalculated results in Tables 4 and 5 are provided for convenience.

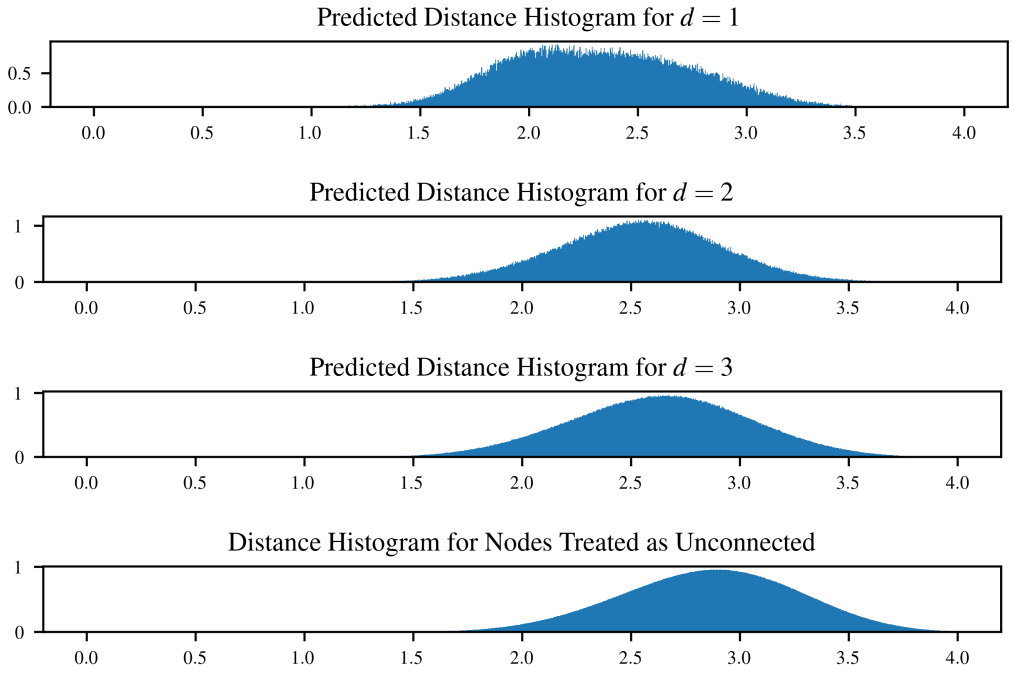

Figure 3: Same as Fig. 2, but for the linear regression baseline.

Table 3: A* performance of different heuristics, including unconnected node pairs

Model		**Time (s)**	**Nodes Expanded**	**Path Length**	**Test Loss**
Hidden Length	*Max. Distance*				
	Null Heuristic	0.0322	25 955	3.5208	—
	Linear Baseline	0.8094	23 827	3.5313	1.6922
192	4	0.7125	12 468	3.5396	0.5820
	5	0.7094	12 301	3.5521	0.7793
	6	0.7337	12 658	3.5604	1.0322
128	4	0.7618	12 999	3.5313	0.6691
	5	0.7190	12 906	3.5458	0.8734
	6	0.7969	13 214	3.5604	1.2099

Table 4: A* performance of heuristics with 128-dimensional hidden layers relative to models with 192-dimensional hidden layers, derived from Table 2

Max. Distance	**Test Loss (%)**	**Nodes Expanded (%)**	**Time (%)**	**Path Length (%)**
4	+14.97	+15.10	+14.14	−0.02
5	+12.06	+18.03	+8.36	−0.02
6	+17.22	+14.98	+16.21	0.00

Table 5: Same as Table 4, but derived from Table 3

Max. Distance	**Test Loss (%)**	**Nodes Expanded (%)**	**Time (%)**	**Path Length (%)**
4	+14.97	+4.26	+6.92	−0.02
5	+12.06	+4.92	+1.35	−0.02
6	+17.22	+4.39	+8.61	0.00

C Outer Product Performance

Band (2024) remarks that outer product features are a strange choice, but they may not be as strange as they seem. First, they generalize the element-wise product in a non-commutative way. Specifically for $u, v \in \mathbb{R}^n$, each scalar $u_i v_i$ is a diagonal element of the matrix $uv^\intercal$, and $uv^\intercal = (vu^\intercal)^\intercal$ which is generally not equal to $vu^\intercal$. Furthermore, consider the logit in the logistic regression, which can be written as

$$\begin{aligned} z &= \sum_{ij} W_{ij} u_i v_j \\ &= v^\intercal \cdot W \cdot u. \end{aligned}$$

Suppose now that W can be written as $V^\intercal U$ for some V, U. One way to do this, for example, would be to set $V := I$. Then,

$$z = (Vv)^\intercal \cdot (Uu).$$

In other words, it is as if the two input vectors are being transformed into the same space before a dot product is taken between them. Band (2024) is skeptical of this reasoning, pointing out that text embeddings should already reside in the same space. Nonetheless, this suggests a way to reduce the overfitting observed in Sec. 4.2. Namely, constrain W to be low rank by having both U and V map to a lower dimension $\mathbb{R}^m \subsetneq \mathbb{R}^n$. This follow-up is not explored.

D Cosine Similarity

A text embedding is just a function f that maps arbitrary pieces of text to vectors in $\mathbb{R}^n$. That function has the property that, if two inputs a and b are semantically "close", then the vectors they map to $u := f(a)$ and $v := f(b)$ are also "close". While the model learns what it means for two pieces of text to be similar, the definition of "closeness" in $\mathbb{R}^n$ must be fixed ahead of time — the model would not be of much use otherwise. Denote the "similarity" between u and v as $\Delta(u, v)$. One could define $\Delta(u, v) := \langle u, v \rangle$, which is called dot-product similarity. Unfortunately, this definition introduces a dependence on the magnitudes of u and v. To correct for this, instead define

$$\Delta(u, v) := \frac{\langle u, v \rangle}{\|u\|_2 \cdot \|v\|_2}.$$

This is called cosine similarity, because the expression above computes the cosine of the angle between u and v:

$$\cos \vartheta = \Delta(u, v).$$

As a result, $-1 \leq \Delta(u, v) \leq 1$. A similarity of 1 indicates that the two vectors point in the same direction, -1 indicates they point in opposite directions, and 0 means they are orthogonal. Finally, note that Δ is scale invariant, since both u and v are normalized before being fed into the dot product. In other words, for any positive constants λ and μ, it is true that

$$\Delta(\lambda u, \mu v) = \Delta(u, v).$$

Hence, the input vectors u and v can be normalized ahead of time. At that point, $\|u\|_2 = \|v\|_2 = 1$, so the cosine similarity can be computed with just dot product $\langle u, v \rangle$.

E Linear Regression

In linear regression, a set of training pairs $\{x^{(t)}, y^{(t)}\}_t$ is given, with each feature vector $x^{(t)} \in \mathbb{R}^n$ and each response variable $y^{(t)}$ a scalar. The aim is to learn an affine hypothesis function $\theta^\intercal x + \phi$ to predict the response variable given the feature vector. For notational convenience, a one is prepended to x so that the hypothesis function can be written as $\theta^\intercal x$. The loss function is squared error:

$$J(\theta) = \frac{1}{2} \sum_t \left(\theta^\intercal x^{(t)} - y^{(t)} \right)^2.$$

While this can be solved analytically, Adam is used here instead. For it, partial derivatives are needed:

$$\frac{\partial J}{\partial \theta_\mu} = \sum_t \left(\theta^\intercal x^{(t)} - y^{(t)} \right) \cdot x_\mu^{(t)}.$$

With some work, this section can be merged with Appendix F, as both linear regression and logistic regression are Generalized Linear Models. Specifically, linear regression assumes that y is normally distributed to maximize the likelihood.

F Logistic Regression

In logistic regression, again a set of training pairs $\{x^{(t)}, y^{(t)}\}_t$ is given, but each $y^{(t)}$ is a boolean instead of a scalar. The probability of $y^{(t)}$ being true is assumed to be $\sigma(\theta^\intercal x^{(t)} + \phi)$. Here, a one is again prepended to x to drop the $+\phi$, and

$$\sigma(x) := \frac{1}{1 + e^{-x}}.$$

To choose θ according to binary cross-entropy loss, the probability of observing the data given the model parameters — the likelihood — is maximized. This is equivalent to minimizing the negative-log of the likelihood, which is

$$\begin{aligned} J(\theta) &= -\sum_t \begin{cases} \ln \sigma(\theta^\intercal x^{(t)}) & \text{if } y^{(t)} = 1 \\ \ln\left(1 - \sigma(\theta^\intercal x^{(t)})\right) & \text{if } y^{(t)} = 0 \end{cases} \\ &= -\sum_t \begin{cases} \ln \sigma(\theta^\intercal x^{(t)}) & \text{if } y^{(t)} = 1 \\ \ln \sigma(-\theta^\intercal x^{(t)}) & \text{if } y^{(t)} = 0 \end{cases}. \end{aligned}$$

For Adam,

$$\begin{aligned} \frac{\partial J}{\partial \theta_\mu} &= -\sum_t \begin{cases} \sigma(-\theta^\intercal x^{(t)}) \cdot x_\mu^{(t)} & \text{if } y^{(t)} = 1 \\ \sigma(\theta^\intercal x^{(t)}) \cdot -x_\mu^{(t)} & \text{if } y^{(t)} = 0 \end{cases} \\ &= \sum_t \begin{cases} \sigma(-\theta^\intercal x^{(t)}) \cdot -x_\mu^{(t)} & \text{if } y^{(t)} = 1 \\ \sigma(\theta^\intercal x^{(t)}) \cdot x_\mu^{(t)} & \text{if } y^{(t)} = 0 \end{cases}. \end{aligned}$$

Again, with some work, this section can be merged with Appendix E, as both linear regression and logistic regression are Generalized Linear Models. Specifically, logistic regression assumes that y is Bernoulli to maximize the likelihood.

G Backpropagation

In the context of this project, backpropagation is an efficient way to the derivatives of a loss function with respect to every weight and bias in a multi-layer perceptron. Notation is taken from Appendix J.1. First, the gradient of a single training example's loss ℓ_t is taken with respect to the model output on that training example $\hat{d}_t = a_{t\xi}^{[N_H+1]}$. Next, for each layer starting at $i = N_H + 1$, derivatives are computed for each of that layer's parameters and with respect to that layer's inputs. Specifically, assuming $a_\xi^{[i]} = f(z_\xi^{[i]})$ for some function f — either the identity or ReLU — write

$$\begin{aligned} \frac{\partial \ell_t}{\partial z_\mu^{[i]}} &= \frac{\partial \ell_t}{\partial a_\mu^{[i]}} \cdot f'(z_\mu^{[i]}) \\ \frac{\partial \ell_t}{\partial W_{\mu\nu}^{[i]}} &= \frac{\partial \ell_t}{\partial z_\mu^{[i]}} \cdot a_\nu^{[i-1]} \\ \frac{\partial \ell_t}{\partial b_\mu^{[i]}} &= \frac{\partial \ell_t}{\partial z_\mu^{[i]}} \\ \frac{\partial \ell_t}{\partial a_\mu^{[i-1]}} &= \sum_\xi \frac{\partial \ell_t}{\partial z_\xi^{[i]}} \cdot W_{\xi\mu}^{[i]}. \end{aligned}$$

This process continues for $i = N_H + 1, \cdots, 1$. Note that each step uses the results of the previous one via $\partial \ell_t / \partial a_\mu^{[i]}$. This is where backpropagation gets its efficiency from.

H Residual Connections

The idea of residual connections is introduced by He et al. (2015). They save the activations of an earlier point p and add them to a later point in the model. So, if a later layer q would normally be $F(p)$, adding a residual connection to that point would give $q = F(p) + p$. For backpropagation (described in Appendix G), the only change is that

$$\frac{\partial q_\nu}{\partial p_\mu} := \frac{\partial q_\nu}{\partial p_\mu} + \mathbf{1}[\mu = \nu], \text{so}$$
$$\frac{\partial \ell_t}{\partial p_\mu} := \frac{\partial \ell_t}{\partial p_\mu} + \frac{\partial \ell_t}{\partial q_\mu},$$

and all the partial derivatives that depend on that are affected. Ultimately, this has the effect of stabilizing training by preventing vanishing gradients.

I Adam

Recall that Stochastic Gradient Descent (SGD) picks a random training example, say the t-th one, and updates the k-th model parameter according to

$$\theta_k := \theta_k - \alpha \cdot \frac{\partial \ell_t}{\partial \theta_k}.$$

Here, the learning rate α is a hyperparameter, and ℓ_t denotes the loss of the model on the t-th example.

Adam (Kingma and Ba, 2015) presents several innovations over SGD. First, it adapts the learning rate based on the previous derivatives observed. This adjustment is done for each parameter θ_k, meaning the parameters do not share a single learning rate. Second, it applies momentum to the gradient used for updates. This helps with escaping local minima, with moving through flat areas of the loss landscape, and with smoothing the gradient samples. Finally, it takes a moving average of the square of the partial derivative, and uses that to scale the update. This modification is inspired by RMSProp (Hinton, 2014) and AdaGrad (Duchi et al., 2011). It has a few ideas behind it. For example, if the samples of the squared derivative are large, but the averaged derivative is small, then the loss landscape is flat and the learning rate is reduced. Additionally, if most samples are zero but a few are large, those large samples will cause large updates, improving performance with sparse gradients.

The final Adam algorithm operates as follows. A random training example is picked, say the t-th one. For each parameter θ_k, the derivative is sampled

$$g_k := \frac{\partial \ell_t}{\partial \theta_k}.$$

Note $g_k \in \mathbb{R}$. This sample is used to update the moving averages of the gradient and squared gradient:

$$m_k := \beta_1 m_k + (1 - \beta_1) \cdot g_k$$
$$v_k := \beta_2 v_k + (1 - \beta_2) \cdot g_k^2.$$

Here, $\beta_1, \beta_2 \in [0, 1)$ are hyperparameters. With this update rule, it turns out that m_k and v_k are biased estimators of $\mathbb{E}_t[\partial \ell_t / \partial \theta_k]$ and $\mathbb{E}_t[(\partial \ell_t / \partial \theta_k)^2]$ respectively. This bias is corrected for by setting

$$\hat{m}_k := \frac{m_k}{1 - \beta_1^\tau}$$
$$\hat{v}_k := \frac{v_k}{1 - \beta_2^\tau},$$

where τ is the zero-indexed current timestep. Finally, the parameter is updated as

$$\theta_k := \theta_k - \alpha \cdot \frac{\hat{m}_k}{\sqrt{\hat{v}_k}}.$$

By default, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Additionally, a small ϵ is added to the denominator of the above expression to prevent division by zero. It is a hyperparameter, and by default $\epsilon = 1 \times 10^{-8}$.

J Neural Network Architecture

J.1 Architecture Description and Hyperparameters

A standard Multi-Layer Perceptron (MLP) is used. It has an input layer $a^{[0]}$ with either 256 or 512 nodes depending on the combination operator K (described in Sec. 4.1). To query the distance from u to v, with text embeddings s and t respectively, the input layer is set to $a^{[0]} := K(s,t)$. The output layer is one node with no activation function — i.e. $h(x) = W^{[N_H+1]}a^{[N_H]} + b^{[N_H+1]}$ where $h(x) \in \mathbb{R}$. Finally, N_H hidden layers are used each with L_H nodes and ReLU activation. That is,

$$a^{[i]} = \text{ReLU}\left(W^{[i]}a^{[i-1]} + b^{[i]}\right)$$

for $i = 1, \cdots, N_H$, where the function $\text{ReLU}(t) = \max\{0, t\}$ is applied independently to each element of its vector argument. Each hidden layer has the same length; Salahi (2024) says this is standard practice and doing this reduces the search space. Lastly, formally write the model parameters as $\theta = \{W^{[i]}, b^{[i]}\}_{i=1}^{N_H+1}$, and define the hyperparameters of the architecture as $\phi := \{K, N_H, L_H\}$.

The description above uses the *Baseline* architecture. Two other architectures are considered, both using residual connections (described in Appendix H). The first of the two alternatives uses *Single-Pass* hidden layers. A residual connection is added between the input of the i-th hidden layer and the output of its affine transformation, just before the ReLU. Formally,

$$a^{[i]} = \text{ReLU}\left(W^{[i]}a^{[i-1]} + b^{[i]} + a^{[i-1]}\right)$$

for $i = 2, \cdots N_H$. Another architecture uses *All-Pass* hidden layers. A residual connection is added between the input of the first hidden layer and the output of the last layer's affine transformation, just before the ReLU. Formally,

$$a^{[N_H]} = \text{ReLU}\left(W^{[N_H]}a^{[N_H-1]} + b^{[N_H]} + a^{[1]}\right).$$

In both of these cases, no residual connection is added from the inputs $a^{[0]}$ since the dimensions do not match. (In fact, using residual connections is one reason all hidden layers have the same length.)

A third architectural knob involves constraining the model's output to be at least one via a softplus on the final layer, similar to what Rizi et al. (2018) do. Formally, $\text{softplus} : \mathbb{R} \to \mathbb{R}$ is given by

$$\text{softplus}(t) = \ln(1 + e^t),$$

and the final layer's output is set to

$$a^{[N_H+1]} := \text{softplus}(a^{[N_H+1]}) + 1.$$

Lastly, the choice of residual connections or whether to use softplus are hyperparameters of the model. They go in ϕ and are tuned.

J.2 Loss Function Description

A modified version of Mean Squared-Error (MSE) loss is used. First, a maximum distance M is set, as described in Sec. 5.1. This prevents the model from trying too hard to fit datapoints with a low signal-to-noise ratio. The remaining possible distances — the remaining "distance classes" — are $\{1, \cdots, M-1, \infty\}$. Each class is weighted inversely proportional to how many times it occurs in the training set. Finally, the loss ℓ_i for a single training example (s_i, t_i, d_i) with weight w_i is

$$\ell_i = w_i \cdot \begin{cases} \begin{cases} (\hat{d}_i - 1)^2 & \text{if } \hat{d}_i \geq 1 \\ 0 & \text{else} \end{cases} & \text{if } d_i = 1 \\ \begin{cases} (\hat{d}_i - M)^2 & \text{if } \hat{d}_i \leq M \\ 0 & \text{else} \end{cases} & \text{if } d_i = \infty \\ (\hat{d}_i - d_i)^2 & \text{else} \end{cases}.$$

To justify this loss function, note that it is mostly equivalent to weighted MSE. In the edge case where $d_i = 1$, the loss is forced to its minimum possible value when the model predicts a distance of less

Table 6: Metrics for different hyperparameters on distance estimation

Configuration		Validation Loss	Training Loss	Training Time (s)
	baseline	0.4595	0.4087	3868
N_H	2	0.5394	0.4849	2993
	4	0.4555	0.3988	4830
L_H	128	0.5183	0.4701	3024
	160	0.5115	0.4598	3492
	256	0.4649	0.4151	5029
K	$\odot$	0.6368	0.5935	1500
Residuals	*Single-Pass*	0.4595	0.4098	3934
	All-Pass	0.4559	0.4055	3929
Final Layer Activation	Softplus	0.4805	0.4267	3779
Training Epochs	40	0.4434	0.3879	4471
	45	0.4408	0.3789	5029
	50	0.4216	0.3577	5587

than one. It is okay if the model does this. Two nodes cannot be closer than one apart, and the output of the model is clamped to be at least one anyway. Hence, it does not make sense to punish the model for that case. Similar reasoning applies to $d_i = \infty$. The nodes are being treated as unconnected, so as long as the model returns a sufficiently large value, the exact prediction does not matter.

Finally, the derivative of ℓ_i is computed. Define

$$\hat{k}_i = \begin{cases} \begin{cases} \hat{d}_i - 1 & \text{if } \hat{d}_i \geq 1 \\ 0 & \text{else} \end{cases} & \text{if } d_i = 1 \\ \begin{cases} \hat{d}_i - M & \text{if } \hat{d}_i \leq M \\ 0 & \text{else} \end{cases} & \text{if } d_i = \infty \\ \hat{d}_i - d_i & \text{else} \end{cases},$$

Then,

$$\ell_i = w_i \cdot k_i^2$$

and

$$\frac{\partial \ell_i}{\partial \hat{d}_i} = 2w_i \cdot \hat{k}_i \cdot \frac{\partial \hat{k}_i}{\partial \hat{d}_i}.$$

Finally, note that if $\partial \hat{k}_i / \partial \hat{d}_i$ is zero then so is $\hat{k}_i$, which means the expression above simplifies to

$$\frac{\partial \ell_i}{\partial \hat{d}_i} = 2w_i \cdot \hat{k}_i.$$

This would be the same expression as for weighted MSE loss if the function $\hat{k}_i(\hat{d}_i)$ were the identity.

J.3 Hyperparameter Tuning

After some hyperparameter tuning, a baseline configuration is found with: three 192-dimensional hidden layers, concatenation for combination, no residuals, and no softplus. Changes to each of these hyperparameters are made, and the resultings are shown in Table 6. Increasing N_H gives a very slight improvement, while reducing it causes a significant performance drop. The same can be said for L_H and for the number of epochs. So, these hyperparameters are not increased further both to save compute time and since doing so gives little benefit.

Using element-wise multiplication significantly degrades performance, which is surprising given the operator's performance on link prediction. However, its showing in that case could be an artifact of the shallow model used (Salahi, 2024). For more expressive models, concatenation generally

Table 7: Metrics for different residual connection strategies on multiple training runs

Residuals	**Validation Losses**		
Baseline	0.4618	0.4722	0.4724
Single-Pass	0.4500	0.4618	0.4649
All-Pass	0.4503	0.4693	0.4874

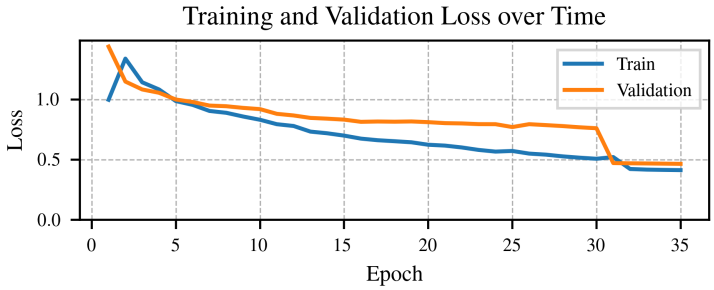

Figure 4: Training and validation losses of the default model over the course of its training. The learning rate was reduced just after epoch 30, causing a 38 % reduction in validation loss.

performs better (Band, 2024). Additionally, placing a softplus on the final layer moderately degrades performance. This is consistent with advice from Band (2024) that neural networks generally perform worse when a nonlinearity is placed on their output. For residuals, the validation losses are very close to each other. Furthermore, the final loss appears to vary randomly by up to 10 %. Hence, multiple runs are collected for the different residual choices, and the results are given in Table 7. It appears that *Single-Pass* performs best, which is strange. He et al. (2015) use a non-linearity between residual connections. Without it (as is here), the model theoretically does not gain any expressive power.

The hyperparameters of Adam are left at their default. The learning rate and learning rate schedule are tuned. The learning rate is set as large as possible without causing divergence, which results in $\alpha = 7.0 \times 10^{-5}$. As for the learning rate schedule, good approach would have been to use a warmup-stable-decay (Hu et al., 2024) or a cosine strategy. Unfortunately, an ad hoc step schedule is used instead. The learning rate is held constant until the last five epochs, at which point it is reduced by a factor of ten. As shown in Fig. 4, the reduction causes a significant reduction in the validation loss — much more than the jump in training loss.

Ultimately, this tuning produces a default configuration of: three 192-dimensional hidden layers, concatenation for combination, *Single-Pass* residuals, and no softplus. The maximum distance M is not tuned here, since losses from different Ms are not directly comparable. It is tuned in Sec. 6.2.