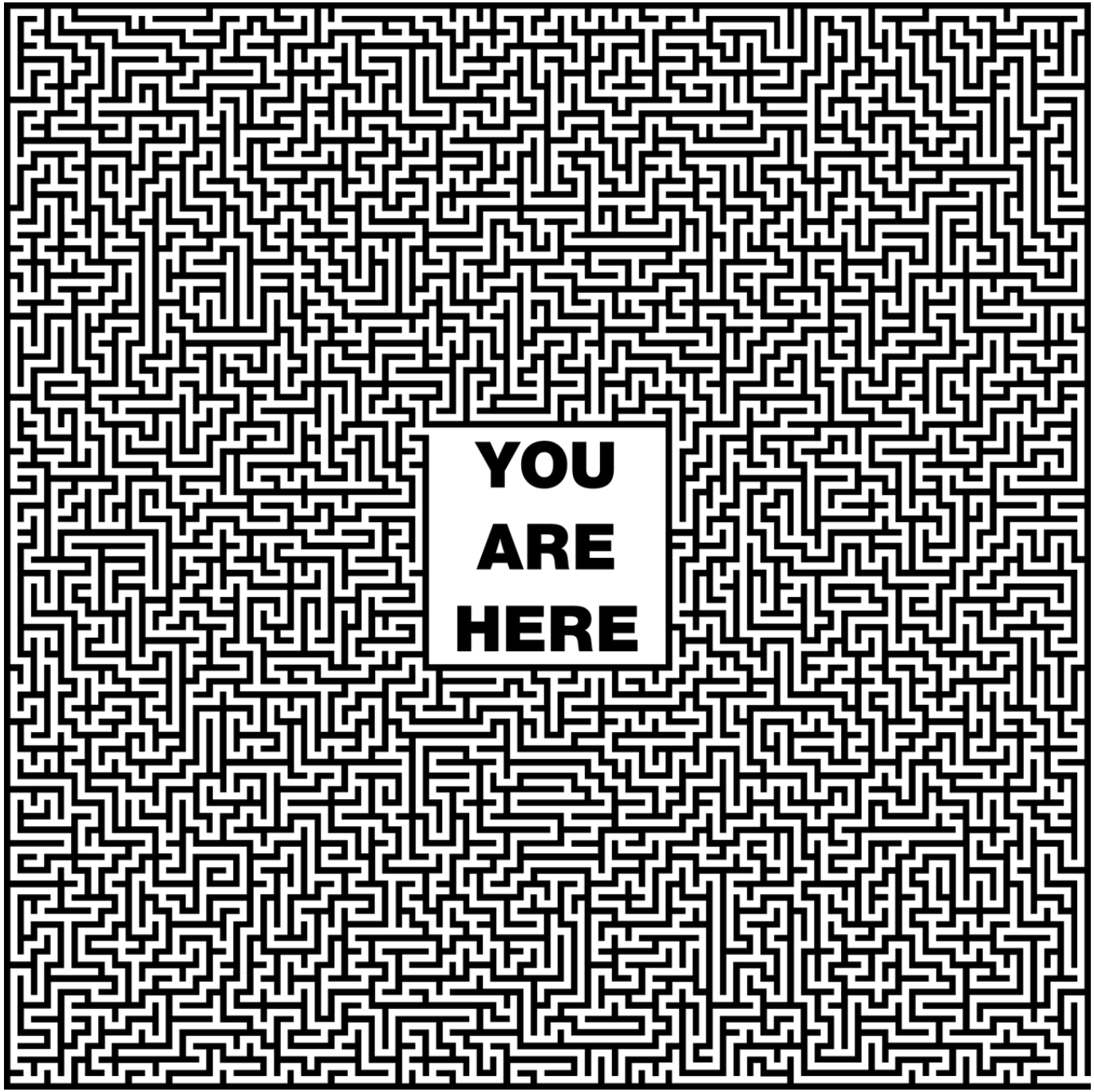
✦ Member-only story

Deep Walk: The overlooked lovechild of NLP and graph data structures

Random Walks and Word2Vec outperform matrix factorization, producing amazingly powerful node embeddings

Jake · Follow

Published in Towards Data Science · 11 min read · Jun 7, 2020



credit: Pixabay

The theory

In recent years, natural language processing (NLP) has experienced a renaissance of innovation; it seems every year, some new algorithm has outperformed its predecessor. For example, the seminal paper, "Attention is all you need" really shook up the NLP snowglobe, putting the previous state of the art neural network architectures, like the LSTM, to shame. However, you may be surprised to hear that graph data structures are just now reaching peak level academic interest with the application of neural networks.

Graphs are tricky; they can be represented as (adjacency) matrices but they can also be represented as their own unique data structure, where each node references a variable number of other nodes in the same graph. This variability is really what makes graphs so fascinating — they're not quite tabular data, easily represented by a matrix, but they're not quite unstructured, either. In fact, some might argue that images are far more structured than graphs. The contents of an image might be highly variable, but every pixel has the same number of adjacent pixels (neighbors) given the specified dimensions. Graphs nodes, conversely, have variable numbers of neighbors. Despite their tricky nature, graphs occur *everywhere* in society; social networks, power grids, and supply chains are just a few distinct examples of graph versatility.

A good deal of thought has been put into questions such as *"How best to search a graph — depth or breadth first?"* But increasingly, finding similarity (or dissimilarity) between nodes is gaining interest. This could be of use in humanitarian goals (such as identifying potential cities at high risk of viral outbreaks given previous outbreak locations and their structure) or simple business goals (such as recommending friends and/or products to application users.)

Consider recommender systems — Not all users like the same number of products and not all products are liked by the same number of users; it all but begs for graph data structures. *Yet…it's extremely common to use matrices and decomposition techniques (like SVD) to render recommendations.*

To learn node embeddings, the graph structure must be *sampled* in some way. One naive (but surprisingly effective) means is through random walks, which is extremely simple: Start at any given node, identify all neighbors, randomly select one…and *walk.* Rinse and repeat until you've walked *enough* (this takes some judgment and introduces some bias.) In the context of recommender systems, our graph is ***bipartite*** — meaning there are two classes of nodes (products and users.) As such, a given random walk will hop from instances of one class to another. For example:

```
John -> Titanic -> Allie -> Pearl Harbor -> Adam -> The Pianist
```

Sampling a graph is a bit like wandering through a maze; you walk randomly from a starting point (potentially returning to the starting point multiple times before completing the walk) and repeat this process several times. By the time you've completed every walk, you'll have a pretty decent idea of what the maze ***looks like***. In their present form, however, these random walks aren't intuitively usable. No worries — NLP to the rescue!

Word2Vec, an algorithm proposed by Google in 2013, can learn the similarity of words through a sliding window scheme. X words are seen at a time (in the window), one target word and (typically) 5 context words from the immediately surrounding context (duh.) A neural network simply learns to answer the question, "*is a word in (in the current window) a context word* ***of the target word?"*** So a word like "dog" might have frequent context words such as "walk" and "pet." Likewise, the word "cat" might have semi-similar context words (perhaps not including walk — I can't name anyone who *walks* their cat.) *We're able to quantify the similarity between any two words by how frequently they share (or don't share) the same contexts.* And these contexts are captured in embeddings (a fully trained neural network layer of neurons when "activated" by the target word.) The embedding's observed numeric

values can be compared with others by means of distance metrics (such as cosine similarity) to determine how similar to words are.

So we know two things: (A) Our random walks have produced language samples and (B) Word2Vec can learn word embeddings (or in our case, node embeddings) given language. And this is effectively all there is to the Deep Walk algorithm.

If you're interested in the white paper, look no further!

The application

While working at a prominent technology company, I led a team of data science interns to build a product that can match job seekers with job posters. Competitors saturate job posters with job seekers and vice versa. The goal isn't high quality matches, it's attention overload. *If you can't find a job candidate, you're not trying hard enough.* My supervisor wanted a product that could help job seekers and job posters find each other — not by matching job titles but by matching skills. The textbook case I was given:

X-Corp (a fictitious name to conceal identity and interests) routinely needs C++ experts; they upload job-postings for "software engineer(s)" and can't seem to find qualified candidates in a timely enough fashion. What can you do?

I pondered the question and realized that job descriptions and resumes were not being adequately analyzed for skills. **A job title alone was enough to trigger a match on Glassdoor, Indeed, etc.** We needed to learn relationships between skills-to-skills, jobs-to-jobs, and (most importantly) skills-to-jobs. The implications of successfully learning these relationships are highly profitable.

Due to non-disclosure agreement status, I cannot provide the actual data, nor the specific architecture used to solve this problem. However, I'll illustrate the concepts through a synthetic problem using publicly available data. The Bureau of Labor Statistics maintains a dataset, called O*NET (*which I believe is short for Occupational Network.)* This resource is maintained by statisticians, economists, and labor researchers. Several datasets are freely available online. Of interest, the technology skills dataset.

Data Example - Technology Skills:

O*NET-SOC Code	Title	Example	Commodity Code	Commodity Title	Hot Technology
11-2011.00	Advertising and Promotions Managers	Actuate BIRT	43232314	Business intelligence and data analysis software	N
11-2011.00	Advertising and Promotions Managers	Adobe Systems Adobe Acrobat	43232202	Document management software	Y
11-2011.00	Advertising and Promotions Managers	Adobe Systems Adobe AfterEffects	43232103	Video creation and editing software	Y
11-2011.00	Advertising and Promotions Managers	Adobe Systems Adobe Creative Suite	43232402	Development environment software	N
11-2011.00	Advertising and Promotions Managers	Adobe Systems Adobe Dreamweaver	43232107	Web page creation and editing software	N

Technology skills data preview

We're only interested in the second and third columns, job title (Title) and skill title (Example.) This data forms a bipartite graph where a given job has relationships with multiple skills and a given skill has multiple relationships with given jobs. With a little data preprocessing, we can borrow a page from recommender systems (though I will demonstrate that this is less effective than Deep Walk approach.) We simply create a "one-hot" encoding for each skill-job association, described by a 1 if a skill is required by a job (or 0 if it is not) in a job-skill matrix. *Not at all dissimilar to recommender systems!*

For the sake of brevity, I'll just leave the repository link here.

Pandas has some cool built-in functions that allow us to transform data in the format discussed above into one hot encodings relatively painlessly! (But

```
[50] x = pd.get_dummies(skill_data.set_index('Title')['Example'])

     x = x.groupby(lambda var:var, axis=0).sum()

     cols = x.columns.to_list()
     rows = x.transpose().columns.to_list()
```

```
[98] y = x.to_numpy()
     y

     array([[0, 0, 0, ..., 0, 0, 0],
            [0, 0, 0, ..., 0, 0, 0],
            [0, 0, 0, ..., 0, 0, 0],
            ...,
            [0, 0, 0, ..., 0, 0, 0],
            [0, 0, 0, ..., 0, 0, 0],
            [0, 0, 0, ..., 0, 0, 0]], dtype=uint8)
```

Data cleaning

Now, we've got a few options for matrix decomposition. Personally, I prefer using PyTorch's gradient-based methods using a mean-squared-error cost function. However, SVD has enjoyed widespread usage in recommender systems as well. Code specifics have been detailed in the repo!

We can qualitatively evaluate our performance by "spot checking" node embeddings. A simple function that I wrote finds nodes of the correct category, (jobs or skills) given a cosine similarity minimum threshold and maximum count threshold. Some of the results are intuitive, such as Database administrators and perhaps Biostatisticians. However, roles like

Supply Chain Managers and Technical Writers beg the question — "*is everything arbitrarily similar?*" And there's evidence to support this conclusion, both the intuitive matches and unintuitive matches fall in the cosine similarity range (0.8->0.9). Perhaps the meaningful matches are just a matter of chance?

```
[97] get_similar('C++',category='jobs',sim_threshold=0.8,count_threshold=10)

 [('Business Intelligence Analysts', 0.88223124),
  ('Database Administrators', 0.8667243),
  ('Sales Engineers', 0.8611303),
  ('Supply Chain Managers', 0.8567646),
  ('Compliance Managers', 0.8285511),
  ('Biostatisticians', 0.8275735),
  ('Computer User Support Specialists', 0.8239314),
  ('Software Developers, Systems Software', 0.8194614),
  ('Technical Writers', 0.81792337),
  ('General and Operations Managers', 0.8146519)]
```

Matrix factorization approach

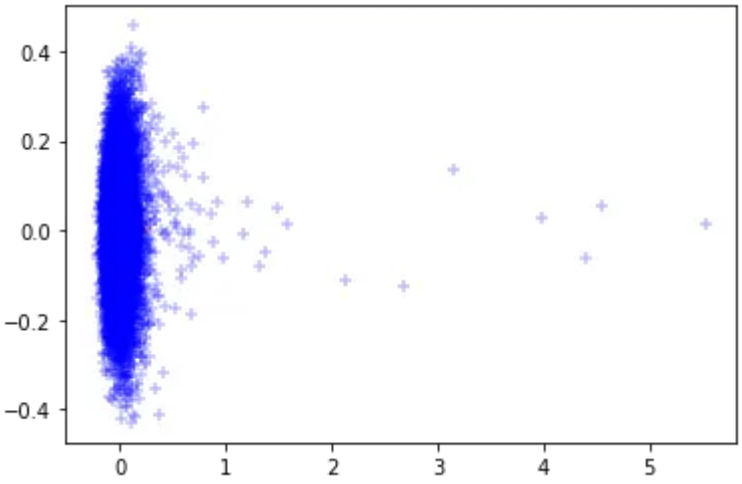
```
[99] get_similar('C++',category=None,sim_threshold=0.8,count_threshold=10)

 [('C++', 0.9999999),
  ('Adobe Systems Adobe Illustrator', 0.9666313),
  ('Microsoft SQL Server', 0.958392),
  ('Structured query language SQL', 0.948711),
  ('Oracle PeopleSoft', 0.9469066),
  ('Microsoft Visual Basic', 0.9451555),
  ('R', 0.94439894),
  ('Computer aided design CAD software', 0.93979764),
  ('Python', 0.93830895),
  ('Microsoft Project', 0.9356728)]
```

Category set to None

Note, in the second screenshot, I set the category to none. This retrieves both skills and jobs; however, we're not capturing any jobs given the threshold supplied. In fact, Adobe Systems Illustrator is the second most common node to C++; it's "more similar" than Python and far more similar than Software Developers (as shown above in the jobs only matches.)

One interesting way that can we can evaluate the results of node embeddings is to look at their 2-dimensional distribution. We can reduce the dimensionality from 10 to 2 via PCA, which preserves the bulk of variance in the model, captured in a 2D representation. PCA is a tricky concept and we won't be making a lengthy detour in the theory there today. For now, let's just examine the results!



Matrix Factorization PCA plot

You can't tell in this representation, however, occupations are encoded in red, while skills are encoded in blue. Virtually all nodes are mapped to the same near 0 x-axis value, while all meaningful variation occurs in the y-axis. There are a few outliers, but these are a handful of *skills* only. So…what went wrong?

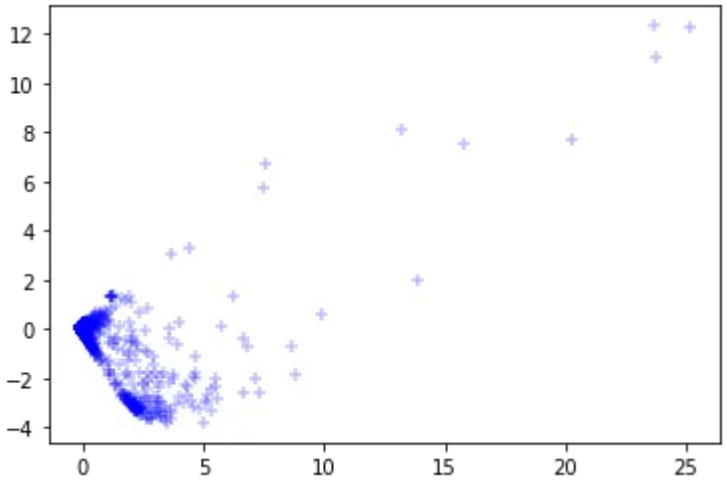
Well, we have a couple of important limitations to discuss. First, there are nearly 9,000 technology skills and only (almost) 1,000 jobs. Additionally, some jobs require far more skills than do others. Because of this, a given matrix row corresponding to a job, might sum to a relatively high number like 100 (meaning the job holds 100 skills) or a relatively low number, like 10. The latent variable at play here is node connectedness, betweenness, and/or centrality. Some nodes are well connected with skills and these skills may or

may not be well connected with other jobs. Additionally, the one-hot matrix is very sparse, due to the job-to-skill imbalance. *Matrix factorization needs to decompose a matrix into two, which when multiplied together, return the original matrix. As you can see, the presence of zeros far outweighs the influence of ones.*

In other words, jobs are similar in terms of the skills that *neither* possess! This same intuition can be applied to recommender systems when matrix sparsity is an issue.

You might have noted that using a one-hot encoded matrix might be suboptimal. And you would be correct! There are various encoding strategies that we can take from NLP to enhance our results...however, this will inevitably introduce user bias. or example, a pseudo-TF-IDF design might account for the total times a skill occurred over all jobs and the total skill count of a single job. *Microsoft Excel is extremely common in this dataset; should it have the same influence as C++, a relatively uncommon skill? Excellent question!*

(For those of you curious, I also used SVD to decompose the one-hot encoded matrix. Here is the PCA plot.)



SVD PCA plot

We could spend weeks trying to feature engineer the *best* design (regardless if the decomposition technique) but at this point, let's take a step back and consider a more intuitive, graph-friendly approach. Enter — Deep Walk.

As discussed, we need to assemble a "corpus" of language through graph sampling via random walks. Let's examine a random walk from the skill, *Python*.

```
random_walk(net,'Python')
```

```
['Python',
 'Statisticians',
 'SAS JMP',
 'Statisticians',
 'DataDescription DataDesk',
 'Statisticians',
 'R',
 'Geneticists',
 'R',
 'General and Operations Managers',
 'Oracle PeopleSoft',
 'Computer User Support Specialists',
 'Encryption software',
 'Intelligence Analysts',
 'Refugee, Asylum and Parole System RAPS',
 'Intelligence Analysts',
 'Microsoft Excel',
 'Freight Forwarders',
 'CargoWise ediEnterprise',
 'Freight Forwarders',
 'CEDAS Gateway']
```

Random walk starting from Python

Note that the occupation, *Statisticians*, occurs 3 times, as intermediate hops between the skills, *SAS JMP* and *DataDescription DataDesk*. What has the random walk has discovered? These skills are somewhat isolated, not very interconnected, and have few job associations. This is exactly what flummoxed the matrix factorization approach. Yet, random walks have eloquently captured the centrality (or lack thereof) in these skills. Note, this random walk was executed using NetworkX, a terrific python-based graph/network modeling package.

Now, instead of implementing Word2Vec from scratch, we'll simply use the Gensim implementation. To keep things consistent/comparable with the matrix factorization results, we'll be learning 10 latent features as well. In either case, this is a hyper-parameter you'll want to keep an eye on — it effectively controls overfit (which is a tricky topic in the context of recommender systems and node embeddings.)

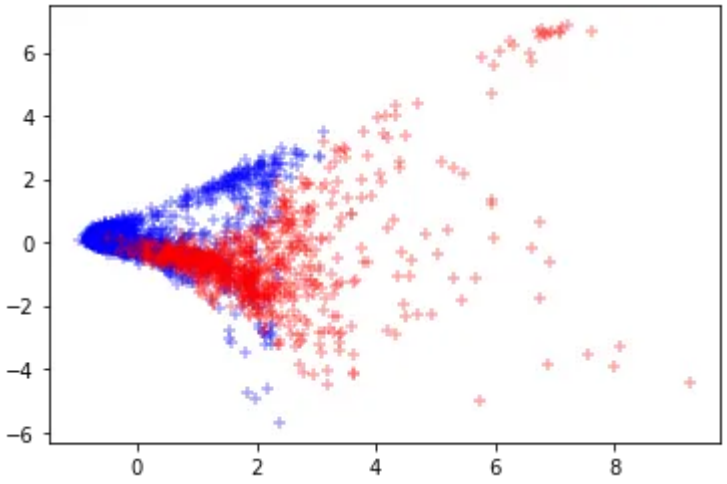
Let's see how Deep Walk performed. Note that both skills and jobs are being retrieved, illustrating that they're mapped to similar spaces but not are not arbitrarily similar to one another — node similarity differences are meaningful!

```
[ ]  embeddings.most_similar('C++')

/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:1: Dep
  """Entry point for launching an IPython kernel.
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: Futu
  if np.issubdtype(vec.dtype, np.int):
[('Python', 0.986066997051239),
 ('The MathWorks MATLAB', 0.9857805967330933),
 ('Oracle Java', 0.9749922156333923),
 ('Automotive Engineers', 0.9691824316978455),
 ('Physicists', 0.9678109884262085),
 ('Robotics Engineers', 0.9551489353179932),
 ('Wind Energy Engineers', 0.9533091187477112),
 ('Linux', 0.9500349760055542),
 ('C', 0.9420895576477051),
 ('Microsoft Visual C#', 0.93912
```

Technology Skills (1).xlsx

To further illustrate the quality of node embeddings produced, let's examine another PCA plot. (Again, occupations and red and skills are blue.)



Deep Walk PCA plot

Wow! This is not only beautiful, it shows so many things we ought to want to from our embeddings. (A) Embeddings are distributed over a far greater space. (B) There are distinct categorical regions that are primarily "skills" versus regions that are primarily "occupations." (C) However, these regions have a moderate amount of overlap, depicting the highly interconnected nodes (whether they be skills or occupations.) *One reason that this method might have performed so ridiculously well is because of the random walks themselves. There might have been more skills than occupations, but the entire*

"corpus" takes the form of `job -> skill -> job -> skill...` *meaning that the effect of sparsity will be mitigated in a clever way!*

Lastly, the source question we started with — How do we find C++ programmers *faster?* Naive intuition would tell us that software developers are the incumbent C++ experts. However, we can see that Physicists and Robotics Engineers are mapped much closer to C++. We can't know for certain *why* this is the case, but we can propose some narratives. C++ is a low level language. Increasingly, web developers are making use of frameworks so that they don't need to reinvent the wheel continuously. It's unrealistic to expect a web developer to be a security expert, networking expert, graphics rendering expert, database experts, etc etc and all the elements that might be required in a web application. Physicists, however, may very well need to know to collect data from sensors designed by engineers at the same lab and a low level language will help them maximize the utility of the computational resources allotted to them.

TL/DR — Recommender systems have made extensive use of matrix factorization. And this can be extremely useful when there is minimal class imbalance (whether that be users-to-products, jobs-to-skills, etc.) Encoding tricks like TF-IDF can (theoretically) enhance upon the results produced by one-hot encodings. However, sparsity can greatly limit the effectiveness of matrix factorization. Graph-friendly approaches are the future of

recommender systems, information retrieval, and beyond! I encourage you to experiment with the code in the linked repository and apply these techniques to novel applications of your own choosing.

Please subscribe if you think my content is alright! :)

Python Data Science Machine Learning Network Analytics

Written by Jake

1.8K Followers · Writer for Towards Data Science

Amazon Scientist, author/blogger, and continual learner

Follow

More from Jake and Towards Data Science

Jake

Wizard-Level SQL

Crush any Data Scientist Interview!

✨ · 8 min read · Sep 2, 2022

193 6

Serafim Batzoglou in Towards Data Science

Large Language Models in Molecular Biology

Deciphering the language of biology, from DNA to cells to human health

40 min read · Jun 2

758 7

Christian Koch in Towards Data Science

From Data Engineering to Prompt Engineering

Solving data preparation tasks with ChatGPT

8 min read · May 23

710 11

Jake in Towards Data Science

Bayesian Price Optimization with PyMC3

PyMC3, Killer Visualizations, and Probabilistic Decision Making

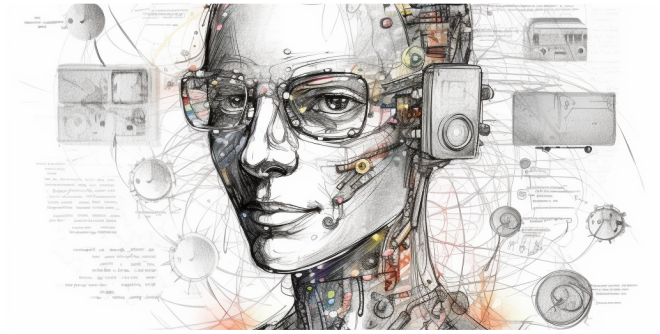
· 5 min read · Dec 9, 2021

211 4

See all from Jake

See all from Towards Data Science

Recommended from Medium



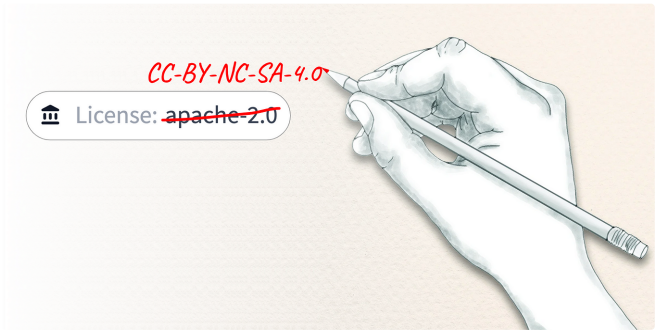
Tomaz Bratanic in Neo4j Developer Blog

Knowledge Graphs & LLMs: Fine-Tuning Vs. Retrieval-Augmented...

What are the limitations of LLMs, and how to overcome them

12 min read · Jun 6

326 6



Clemens Mewald in Towards Data Science

The Golden Age of Open Source in AI Is Coming to an End

NC, SA, GPL, and other acronyms you don't want to see in the open source license of the...

7 min read · Jun 7

624 8

Lists

What is ChatGPT?

9 stories · 104 saves

Stories to Help You Level-Up at Work

19 stories · 99 saves

Staff Picks

348 stories · 111 saves

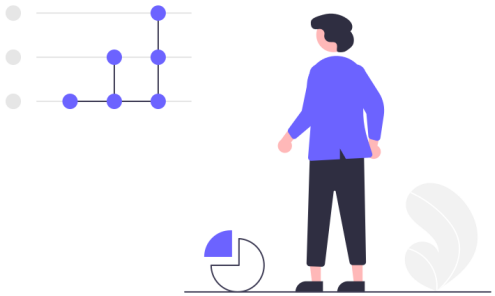
Galina Alperovich in GoPenAI

The Secret Sauce behind 100K context window in LLMs: all tricks...

tldr; techniques to speed up training and inference of LLMs to use large context...

16 min read · May 16

806 3



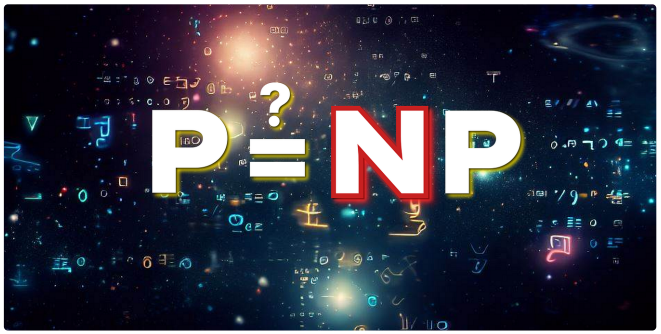
Ravenspike in Dev Genius

Analyzing graph networks Part 2: Utilizing advanced methods

Story overview

· 16 min read · Jun 5

28



Russell Lim in Nice Math Problems

Jason Huang

Do You Really Know What the 'N' Stands For?

Understanding the Infamous P vs NP Problem

· 6 min read · 5 days ago

288 5

Graph Encoder-Decoder Models for NLP

All of the contents in this article are excerpted from the tutorial video, slides, and website of...

5 min read · Dec 31, 2022

27 1

See more recommendations