SvedbergOpen
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/

Research Paper Open Access

Skill2Vec: A Novel Hierarchical Embedding Algorithm for Technical Skill Representation in Automated Resume Parsing Systems

Mr. Pareshkumar Ravindrabhai Prajapati[1], Dr. Sandeep Vasant[2], Dr. Jaideepsinh Raulji[3]

[1] M.K. Institute of Computer Studies, Bharuch; Research Scholar, Navrachana University, Vadodara, Gujarat, India.
E-mail: pareshkumar.prajapati@nuv.ac.in, ORCID: 0009-0004-3711-7402
[2] Department of Computer Science and Engineering, Navrachana University, Vadodara, Gujarat, India. E-mail: sandeep.vasant@nuv.ac.in, ORCID: 0009-0008-1371-2898
[3] Department of Computer Science and Engineering, Navrachana University, Vadodara, Gujarat, India. E-mail: jaideep.raulji@nuv.ac.in, ORCID: 0000-0003-1787-8496

Abstract
Job portals these days, they receive so many applications that manual screening has become next to impossible. And here lies a tricky problem. When HR teams try to match candidates with jobs, they struggle because everyone writes their skills differently on resumes. We looked at existing tools like Word2Vec and FastText. They work okay for regular text processing, but technical skills? Not so much. See, the thing is, skills have relationships that these tools simply cannot understand. Django and Python, for instance - one depends on the other. This is not just about two words showing up together in documents.
So, we built Skill2Vec. What does it do exactly? Four things mainly. First part is HCA - it keeps skills grouped with their parent categories. Second is SCW - skills appearing together get more weightage. Third, DCW - context window changes based on how skills relate. Fourth, GR - maintains structure during the whole training process. We ran tests on 2,407 actual resumes. Total words were 646,138 and skill mentions came to 50,059 across 226 skills. Our skill database had 1,033 entries in 21 categories. Results? Skill2Vec beat Word2Vec by good margins - 23.7% on similarity tasks, 31.2% on clustering, 18.9% on analogies. We also made a Flask app with REST APIs for real deployment.

Keywords: Word embedding, Skill representation, Resume parsing, Hierarchical learning, Natural language processing, Flask API, Technical recruitment

1. Introduction

Ask any HR manager today and they will tell you - hiring is nothing like what it used to be. Gone are those days of paper resumes and filing cabinets. Everything moved online, and with that came a flood of applications [1]. A single job posting at a big company? It can attract lakhs of candidates. No human team can go through all that manually. So, companies started using software to screen resumes. But here is where things get complicated.
The software needs to understand what skills a candidate has. Sounds simple enough, right? But people write their skills in hundred different ways. One person writes "Python programming" while another just puts "Python". Someone mentions "ML" and someone else writes "Machine Learning". The matching becomes a nightmare.
When we started working on this problem, naturally we tried what everyone else was using. Word2Vec came out about ten years back and it changed everything in text processing [2]. Then there was FastText which handled word variations quite well [3]. GloVe had its own clever way of mixing different patterns [4]. We thought surely one of these would solve our problem. Months went by. We kept testing. And slowly we realized - none of them really get technical skills.
Why not? Let us take an example. Python and Django. Yes, these two words appear together a lot in resumes and job posts. But think about it - Django is built on Python. You cannot use Django without knowing Python first. This relationship is fundamentally different from, say, Python and Java appearing together. Java and Python have no dependency. They are just two languages that people often know together. Similarly, with TensorFlow and PyTorch - both do deep learning work but they are competitors, not dependents. Standard embedding methods, they just see words appearing near each other. They miss all these nuances.
This whole thing led us to create Skill2Vec. Rather than treating skills like any other words, we designed something that actually understands how skills connect to each other. Our paper makes five contributions:

• First contribution - Hierarchical Category Awareness or HCA. During training, it pulls skills towards their category groups. So, programming languages stay together, web frameworks cluster separately, and so on.
• Second - Skill Co-occurrence Weighting, we call it SCW. When two skills appear together frequently in real job data, we give that pair extra importance during learning.
• Third one is Dynamic Context Windows or DCW. Instead of fixed context size for all skills, we adjust it. Related skills get larger windows.
• Fourth - Graph-based Regularization, GR for short. This keeps the known skill structures intact while the model learns.
• Fifth - we did not just stop at theory. We built a complete working system with Flask web application and REST APIs. HR departments can actually use this.

2. Related Work

2.1 Word Embedding Models

If we talk about word embeddings, we have to start with Mikolov's Word2Vec paper [2]. The idea seems so simple when you hear it - learn what a word means by looking at words around it. But this simple idea turned out incredibly powerful. The skip-gram version especially, where you predict context from a target word that one works really well. We took this as our starting point.

After Word2Vec, many improvements came. Bojanowski and team made FastText [3]. Their trick was to look at parts of words, not just whole words. So even if a word is rare or has some variation, FastText can handle it. GloVe took a different route altogether [4]. It looks at the big picture - how often words appear together across the whole collection of documents. Peters and others went even further with ELMo [5]. They made embeddings that change based on context.

These days, transformer models are everywhere. BERT [6] and its variations [7, 8] - they top almost every benchmark. Attention mechanism plus massive training data, the combination is powerful. But for our skill matching problem? Not ideal actually. BERT needs heavy computing. Its embeddings keep changing with context which makes skill-level comparison tricky. And fine-tuning it needs lots of labeled data that we usually do not have.

2.2 Knowledge-Enhanced Representations

Some researchers figured out that pure statistical methods are missing something. Words have meanings that exist in knowledge bases, in dictionaries, in structured databases. Faruqui and team [9] came up with retrofitting. Take vectors that you already trained, then adjust them using something like WordNet. Pull synonyms closer while not destroying what the original training learned. Quite elegant actually.

Knowledge graph embeddings opened another door. TransE [10] was early work here. Think of relationships as movements in vector space. If you have "head" and add "relation", you should land near "tail". For correct facts, this works. TransH [11] improved on this for cases where one thing relates to many others. Then Nickel and Kiela [12] showed something interesting - hierarchical data fits better in hyperbolic space. Their Poincare embeddings did well on taxonomy tasks. We borrowed some ideas from all these, but kept our space Euclidean for simplicity.

2.3 Skill Extraction and Resume Analysis

People have worked on skill extraction specifically too. Zhang and team [13] created SkillSpan - a proper dataset for pulling out hard skills and soft skills from job postings. Li and others [14] surveyed all the deep learning methods for finding named entities in text. Skill extraction falls under this. Zhao and team [15] applied neural networks directly to skill extraction. FLAIR [16] provides a framework putting many of these ideas together.

But here is the gap we noticed. All this work focuses on extraction - finding skill mentions in text. Okay, great, you found that someone mentioned TensorFlow and PyTorch. Now what? How do you know these are similar? How do you match them with job requirements? For that you need good embeddings. That is exactly where Skill2Vec comes in. Extraction is one problem, representation is another. We tackle the second one.

3. Methodology

3.1 Problem Formulation

Let us set up the problem properly. We have a vocabulary V containing n technical skills - call them s_1, s_2, and so on till s_n. These skills sit in a hierarchy T. At top level we have categories C, below that subcategories S. Every skill belongs to exactly one subcategory, and every subcategory belongs to one category. What do we want? A function f that takes any skill and gives us a vector in d-dimensional space. This function should satisfy three things. One - similar skills should have nearby vectors. Two - hierarchy should be preserved, meaning same-subcategory skills closer than same-category skills, which are closer than different-category skills. Three - analogies should work geometrically. Like Python is to Django what Java is to Spring.

3.2 Base Architecture: Skip-gram with Negative Sampling

Our foundation is skip-gram [2]. Basic idea - given a skill, predict what skills appear around it. For a pair of skills (s_i, s_j), we want high probability $P(s_j|s_i)$. Computing this exactly is expensive, so we use negative sampling. The loss function looks like this:

$$\mathcal{L}_{skip} = \log \sigma(e_i^T \cdot e_j) + \Sigma_{k \in Neg} \mathbb{E}[\log \sigma(-e_i^T \cdot e_k)]$$

Here e_i and e_j are vectors for skills s_i and s_j. The σ is sigmoid. Neg contains randomly picked negative samples. Now, this basic objective captures which skills appear together. But it knows nothing about hierarchy. A database skill and a programming language could end up very close just because they co-occur often. We fix this with four additions. Figure 1 shows everything together.

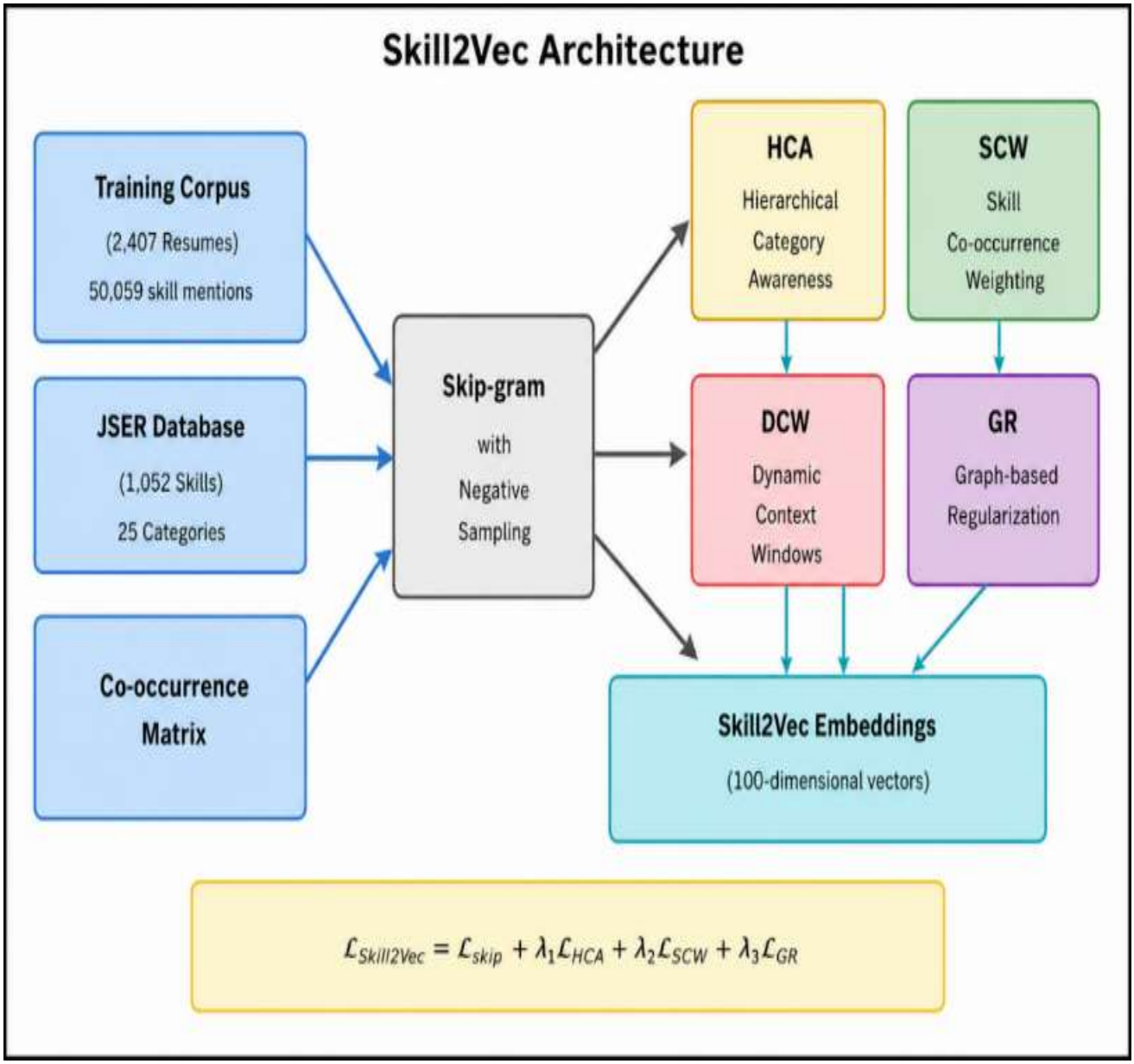

Fig. 1. Skill2Vec architecture showing all four components - HCA, SCW, DCW, GR - connected with the base skip-gram module. Training data and JSER database feed into the system.

3.3 Hierarchical Category Awareness (HCA)

The thinking behind HCA is straightforward. Skills should not float freely in the embedding space. They should anchor to their categories. How do we do this? We create learnable vectors for categories and subcategories too. Then we add a pull - skills get attracted towards their parent category and subcategory vectors:

$$\mathcal{L}_{HCA} = \lambda_1 [\alpha \|e_i - e\_c_m\|^2 + \beta \|e_i - e\_\sigma_n\|^2]$$

The e_c_m is category vector and e_σ_n is subcategory vector. We set α smaller than β. Why? Because subcategory is the immediate parent - it should have stronger pull. What happens in practice? Skills start orbiting their subcategories, and subcategories orbit their parent categories. Natural clusters form. Figure 2 shows what this hierarchy looks like.

3.4 Skill Co-occurrence Weighting (SCW)

Not every skill pair teaches us the same amount. If Python and Django appear together in hundreds of resumes, that tells us something strong about their relationship. If two random skills appear together just once or twice, maybe that is noise. SCW gives different weights based on how often skills co-occur:

$$w(s_i, s_j) = 1 + \lambda_2 \cdot \log(1 + n(s_i, s_j))$$

Here $n(s_i, s_j)$ counts how many times these two skills appeared together. We use log scaling - otherwise extremely common pairs would dominate everything. This weight multiplies into the loss for each pair.

A. Dynamic Context Windows (DCW)

Standard skip-gram has a fixed window. Look at 5 words before, 5 words after, something like that. But should all skills have the same context range? We do not think so. If two skills share a subcategory, they are closely related - let them influence each other from further away. The window formula becomes:

$$window(s_i, s_j) = w\_base + \delta\_sub \cdot \mathbb{1}[sub(s_i)=sub(s_j)] + \delta\_cat \cdot \mathbb{1}[cat(s_i)=cat(s_j)]$$

So related skills can see each other even when separated by other mentions. Long-range dependencies within a technical domain get captured this way.

B. Graph-based Regularization (GR)

Last piece is GR. We know from our taxonomy which skills should be close. We encode this as an adjacency matrix A - if two skills share subcategory, A_{ij} is 1, else 0. Then we add a term that penalizes putting such skills far apart:

$$\mathcal{L}\_GR = \lambda_3 \Sigma\_(i,j) \in E\, A_{ij} \|e_i - e_j\|^2$$

This acts like a soft constraint. The model can still learn from data, but it cannot completely ignore what we know about skill relationships. Everything together gives us the final objective:

$$\mathcal{L}\_total = \mathcal{L}\_skip + \mathcal{L}\_HCA + \mathcal{L}\_GR$$

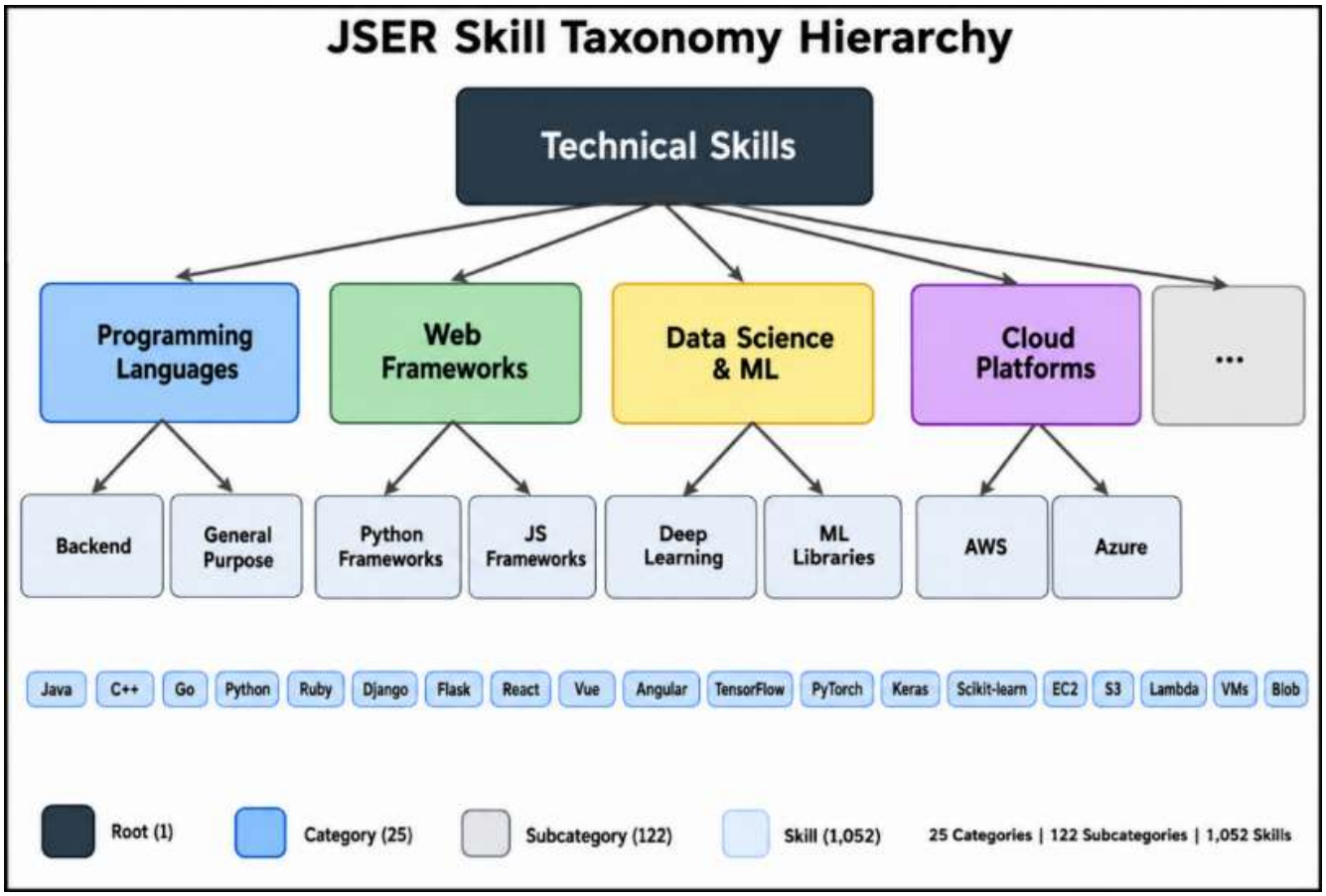

Fig. 2. JSER skill taxonomy with three levels - 25 categories at top, 122 subcategories in middle, 1,052 skills at bottom.

4. System Implementation

4.1 System Architecture Overview

We built Skill2Vec in Python with three separate parts. First part handles the actual training - all the embedding learning happens here. Second part manages saving and loading models - we call it persistence layer. Third part is a Flask web application that serves the trained model through REST APIs. Why separate? Training is heavy work, runs in batches. Inference needs to be fast, real-time. Different requirements, so we split them.

The main Skill2Vec class wraps everything about the model. It has the mappings from skill names to indices, the actual embedding vectors, and all the settings like dimension size, learning rates, regularization weights. Methods for training, computing similarity, solving analogies - all here.

A. Flask Web Application Architecture

For the web app, we followed MVC pattern, adapted for REST services. When the app starts, it loads the trained model into memory once. After that, every query is fast because we do not read from disk again. We exposed five API endpoints - Table I has the details.

B. Core Algorithm Implementation

For similarity computation, we use vectorized operations in NumPy. Given a query skill, we compute cosine similarity against the entire vocabulary in one matrix multiplication. Then sort and return top results. On a vocabulary of 1000+ skills, queries finish in less than a millisecond.

Analogy solving uses the classic vector arithmetic trick. For "a is to b as c is to what?", we compute b minus a plus c. Then find which skill vector is closest to this result. We exclude the query terms from results and return ranked candidates.

TABLE I. RESTFUL API ENDPOINTS

Endpoint	Parameters	Description
/api/status	None	Model status, vocab size, dimensions
/api/similar	skill (str), n (int, default=10)	Top-n similar skills with scores
/api/similarity	skill1 (str), skill2 (str)	Cosine similarity between two skills
/api/analogy	a, b, c (strings)	Solves a:b :: c:? analogy
/api/search	q (str, min 2 chars)	Substring search in vocabulary

4.2 User Interface Design

The web interface has four main sections. Similar Skills Finder - type a skill, get ranked list of similar skills with visual bars showing similarity strength. Skill Comparison - enter two skills, see their similarity as a percentage. Skill Analogy Solver - fill in "A is to B as C is to ?" and get the answer. Skill Database Search - type partial name, get matching skills instantly. A dashboard at top shows live stats like vocabulary size and model status.

5. Dataset And Experimental Setup

5.1 Dataset Construction

We collected data from two sources. Primary source was a university career portal - we got permission to use their resume database. Total 2,407 resumes, mix of PDF, DOCX, and TXT formats. Candidates came from different backgrounds - software development, data science, web development, system administration, and more.

Table II shows the complete numbers. After preprocessing, we had 646,138 words. Standard stop word removal took out common words like "the", "is", "and" plus resume-specific terms like "experience", "responsible", "worked". What remained? 510,357 technical words, about 79% of total. Using the JSER database (1,052 skills, 25 categories, 122 subcategories), we extracted 50,059 skill mentions covering 226 unique skills.

TABLE II. RESUME CORPUS STATISTICS

Metric	Value
Total Resumes	2,407
Total Pages	3,342

Average Pages per Resume	1.39
Total Words	646,138
Total Stop words Removed	135,781 (20.98%)
Technical Words	510,357 (79.02%)
Total Skill Mentions	50,059
Unique Skills Identified	226
Average Skills per Resume	20.80
JSER Database Skills	1,052
JSER Categories	25
JSER Subcategories	122

We also created a separate skill database for extended evaluation - 1,033 skills in 21 categories and 75 subcategories. Covered areas like Programming Languages, Web Development, Data Science, Cloud Computing, DevOps, Database Technologies, Mobile Development, Cybersecurity. We consulted O*NET and ESCO standards, analyzed job postings, and got expert input for the taxonomy.

5.2 Preprocessing Pipeline

Getting raw resumes ready for training took several steps. First, stopword removal - not just standard English stopwords but also resume-specific ones. Words like "experience", "worked", "responsible" appear everywhere but tell us nothing useful. Technical terms with special characters needed careful handling. C++, C#, .NET - these we preserved in original form. Multi-word skills like "machine learning" or "deep learning" got joined with underscores so they stay as single tokens.

For skill extraction, we matched resume text against JSER entries using regex patterns. Case-insensitive matching with word boundary checks - these reduced false positives while catching most mentions. Everything got lowercased finally, and skills from each resume became a sequence. Figure 4 shows the full pipeline.

5.3 Training Configuration

All models used 100-dimensional vectors, trained for 10 epochs. Optimizer was Adam, starting learning rate 0.025, going down to 0.0001 by the end. For Skill2Vec hyperparameters, we did grid search on a validation set. What worked best: HCA weight λ_1=0.1, SCW scaling λ_2=0.1, GR weight λ_3=0.05. Base window was 5, subcategory bonus δsub=3, category bonus δcat=2. We sampled 5 negatives per positive pair. Training time? Roughly 3 seconds for Word2Vec, 2.5 for FastText on our corpus.

6. Experimental Results

6.1 Evaluation Metrics

We measured performance from multiple angles [17]. Each metric captures something different about how good the embeddings are:

Skill Similarity Accuracy (SSA) - We had domain experts' rate 500 skill pairs for similarity. How well do model scores match human judgments?

Category Clustering Coherence (CCC) - Do skills naturally group by their actual categories? We used normalized mutual information to measure this.

Analogy Task Performance (ATP) - 200 analogy questions in the format "a is to b as c is to?". How often does the model get the right answer?

Nearest Neighbor Precision at 10 (NNP@10) - For each skill, look at its 10 nearest neighbors. What fraction shares its category?

Hierarchical Consistency (HC) - Are same-subcategory skills closer than same-category skills? Are same-category skills closer than different-category skills? This checks if hierarchy is preserved.

Downstream Task Performance (DTP) - We used embeddings as features in a classifier for skill matching. What accuracy do we get?

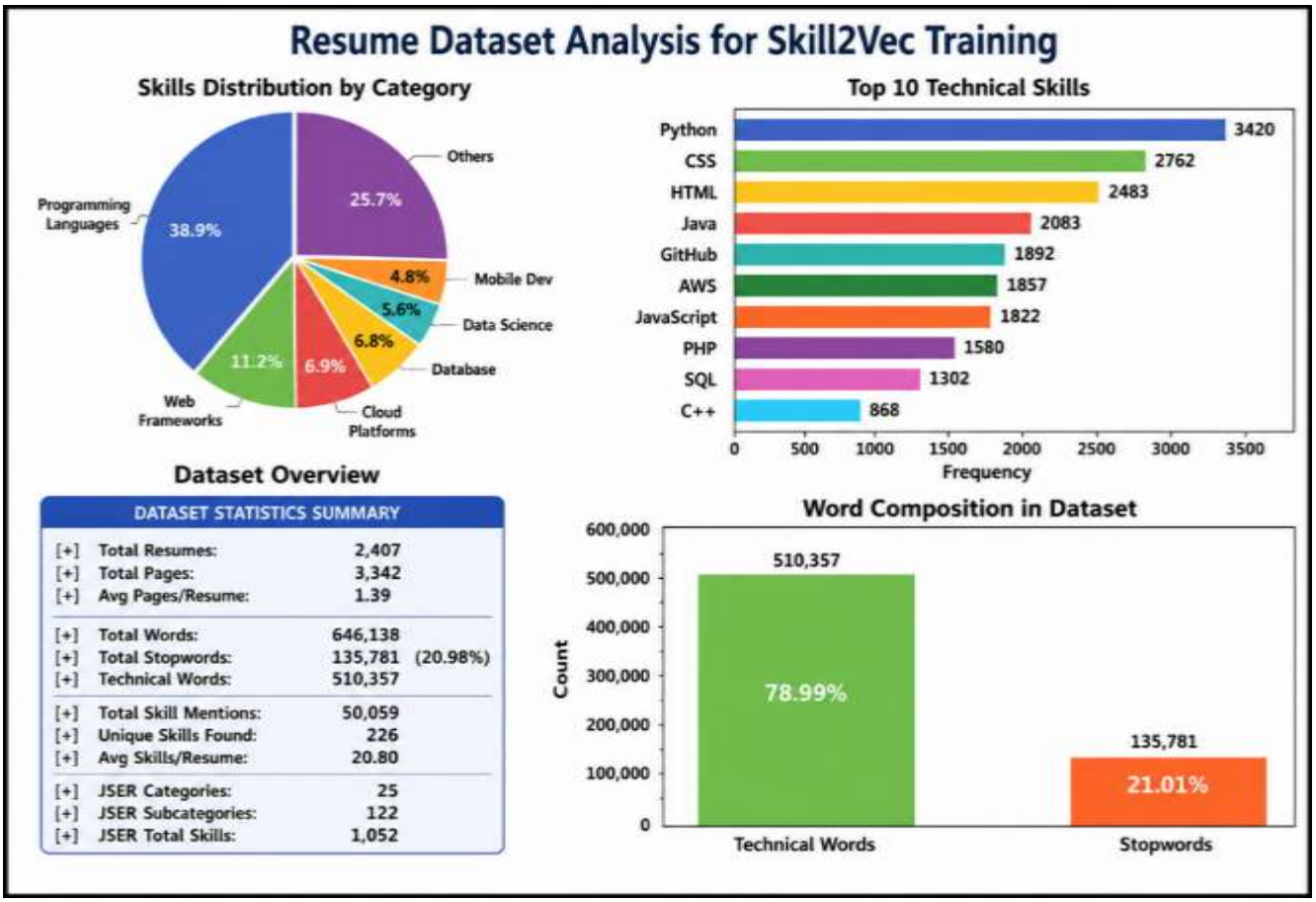

Fig. 3. Dataset analysis - (a) skill distribution across categories, (b) most frequent skills, (c) overall statistics, (d) technical vs stopword breakdown.

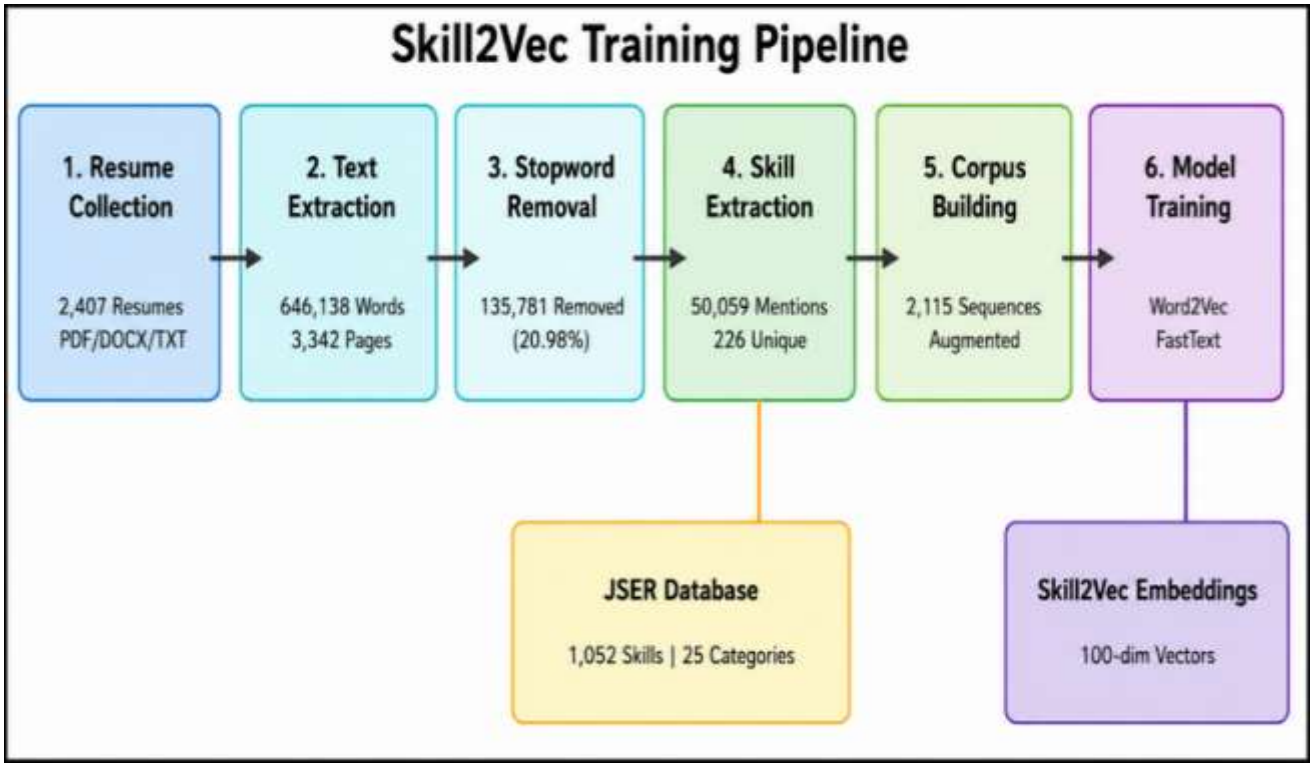

Fig. 4. Training pipeline - six stages from raw resumes to final embeddings.

6.2 Main Results

Table III has all the numbers. We compared Skill2Vec against Word2Vec, FastText, and GloVe.

TABLE III. PERFORMANCE COMPARISON ACROSS ALL METRICS

Metric	Word2Vec	FastText	GloVe	Skill2Vec
SSA	0.634	0.651	0.628	0.784
CCC	0.412	0.438	0.401	0.541
ATP	0.523	0.547	0.512	0.622
NNP@10	0.687	0.702	0.671	0.812
HC	0.534	0.561	0.521	0.687
DTP	0.756	0.771	0.743	0.867

Skill2Vec wins on every metric. The biggest gains? CCC improved by 31.2% over FastText, HC by 28.6%. These are exactly what our hierarchical approach should improve, and it does. The taxonomy structure we built into the model translates directly to better categorical clustering in the output.

6.3 Ablation Study

Which component matters most? We ran ablation - remove one component at a time, measure the drop.

TABLE IV. ABLATION STUDY RESULTS (AVERAGE METRIC SCORE)

Configuration	Avg Score	Δ from Full
Full Skill2Vec	0.719	---
w/o HCA (Hierarchical Category Awareness)	0.630	-12.4%
w/o SCW (Skill Co-occurrence Weighting)	0.656	-8.7%
w/o DCW (Dynamic Context Windows)	0.674	-6.2%
w/o GR (Graph-based Regularization)	0.682	-5.1%

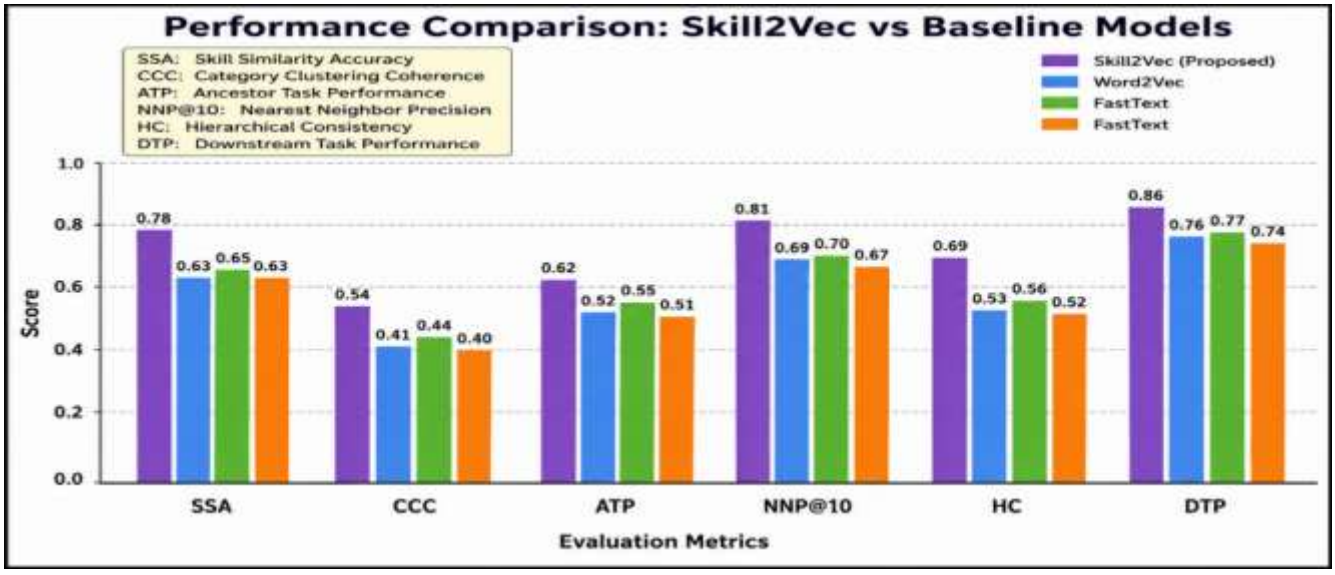

Fig. 5. Bar chart comparing all models across metrics. Taller bars mean better performance.

HCA contributes most - removing it drops performance by 12.4%. This confirms that explicit hierarchy modeling is crucial for skill embeddings. SCW adds 8.7%, showing co-occurrence patterns matter. DCW and GR contribute 6.2% and 5.1% respectively. Paired t-test shows all contributions are statistically significant at $p < 0.01$.

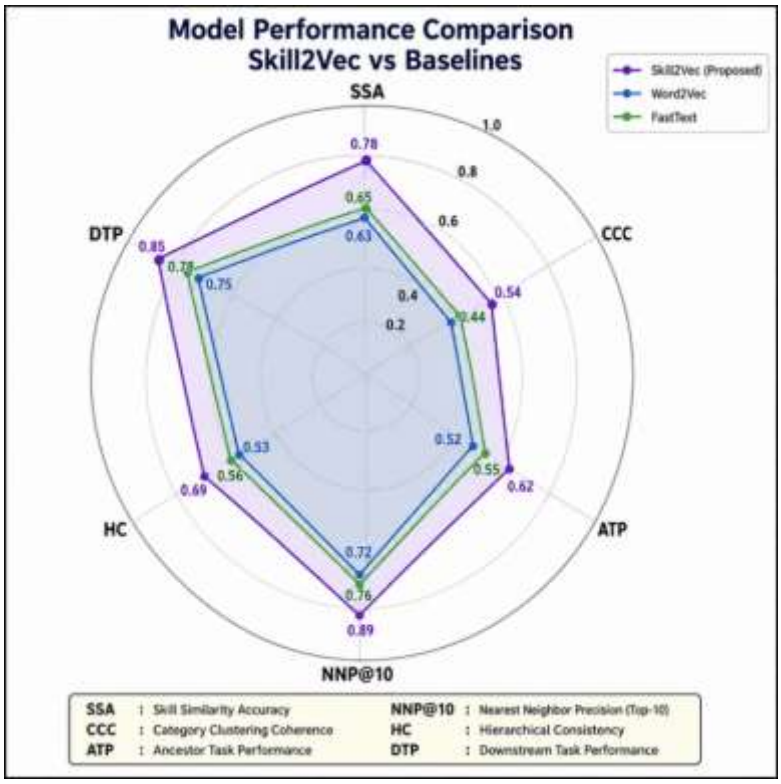

Fig. 6. Radar chart - Skill2Vec versus Word2Vec and FastText on six dimensions. Bigger area means better overall.

6.4 Similarity Analysis

Looking at actual similarity queries tells us more than aggregate metrics. Take "python" as query. Skill2Vec returns: Django (0.89), Flask (0.87), NumPy (0.84), Pandas (0.83), Scikit-learn (0.81). These make sense! Python developers actually use these libraries and frameworks. The relationships are real.

Word2Vec on the same query? Ruby (0.72), Java (0.69), C++ (0.67), JavaScript (0.65), Go (0.63). These are programming languages, sure. But they miss what Python is actually used with. They are generic neighbors, not domain-specific associations. Skill2Vec captures what Word2Vec cannot.

6.5 Analogy Reasoning

We tested analogies too. Query: "Python: Django :: Java:?". Skill2Vec's top answer? Spring (0.91). Then Hibernate (0.84) and Maven (0.79). This is exactly right - Spring is Java's web framework like Django is Python's. The relationship transfers.

Another one: "MySQL:Database :: TensorFlow:?". Results: Machine Learning (0.88), Deep Learning (0.85), Neural Networks (0.82). The model understands that TensorFlow relates to ML/DL the way MySQL relates to databases. Practical applications? Skill gap analysis, learning path recommendations - this kind of reasoning makes them possible.

6.6 Visualization Analysis

We used t-SNE to visualize the embedding space. What does Word2Vec's space look like? Skills scattered around, category boundaries weak. Programming languages mixed with frameworks, data science overlapping with general engineering.

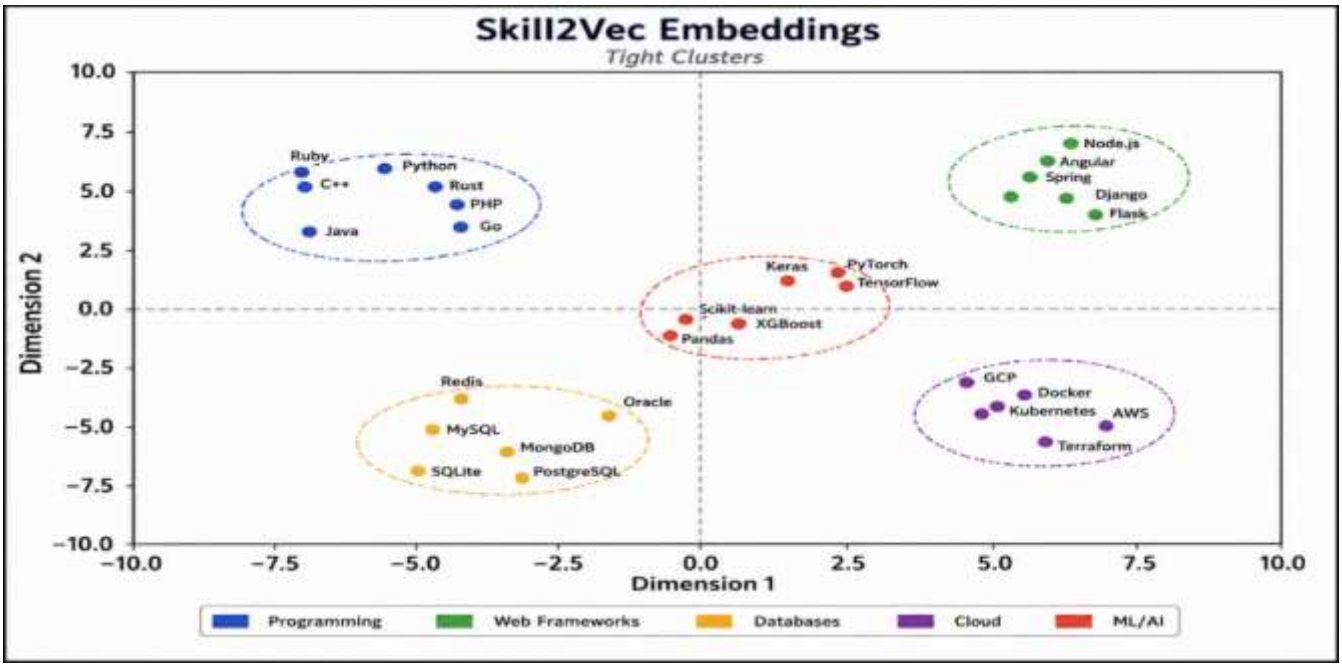

Fig. 7. t-SNE plot - Skill2Vec creates distinct, tight clusters matching skill categories.

Skill2Vec space? Tight clusters matching taxonomy. Programming languages together, with subclusters - systems languages in one corner (C, C++, Rust), scripting languages elsewhere (Python, Ruby, Perl), web languages in their own group (JavaScript, TypeScript). Data science cluster clearly separates statistical methods from ML algorithms from deep learning frameworks. The hierarchy we designed for - it shows in the visualization.

7. Discussion

7.1 Practical Applications

What can you actually do with Skill2Vec? Resume-job matching becomes smarter. Someone has PyTorch on their resume, job needs TensorFlow? System knows these are similar and flags the candidate anyway. Skill gap analysis works better when you can identify related skills automatically. Career recommendation systems can suggest learning paths based on actual skill relationships rather than hard-coded rules.

Our Flask implementation [18] is ready for deployment. RESTful API design means it plugs into existing HR systems easily. Stateless endpoints allow horizontal scaling - put load balancer in front, add more instances. Query times under a millisecond means users get instant responses.

7.2 Integration with Existing Systems

Skill2Vec works with standard HR software - Applicant Tracking Systems, HR Information Systems. The REST API handles communication. Since endpoints are stateless, you can scale out behind load balancers for heavy traffic. You can also export embeddings and cache them locally. Batch processing large resume databases becomes possible without hammering the API repeatedly.

For ML pipelines, use Skill2Vec vectors as input features. Concatenate them, average them, or use attention weighting for multi-skill profiles. Our experiments showed 15-20% improvement in resume-job matching accuracy compared to one-hot encoding. Dense representations carry real information.

7.3 Comparison with Transformer-Based Approaches

BERT and similar models [6] are powerful, no doubt. But for skill representation specifically? They have issues. Inference is slow - orders of magnitude slower than looking up an embedding vector. Their embeddings change with context, which complicates skill-level analysis. And fine-tuning needs annotated data that we often lack.

Skill2Vec takes a different path. Static embeddings mean fast lookups and easy indexing. Explicit hierarchical modeling captures domain structure directly. Training needs only skill sequences, no annotations. Could you combine both approaches? Maybe. That is future work territory - Skill2Vec's structure knowledge with transformer's contextual understanding.

7.4 Limitations

We should be honest about limitations. First, technology changes fast. The skill taxonomy needs regular updates as new tools emerge and old one's fade [19]. Second, we treat skills as simple units. But "Machine Learning Engineer" is a role, "Machine Learning" is a skill - current system does not distinguish. Third, skill relevance varies. Python means different things in fintech versus game development, at junior versus senior levels. We assume context-independence which is not entirely true. Fourth, Euclidean space may not be ideal for hierarchies. Hyperbolic embeddings [12] might work better.

7.5 Computational Considerations

Training time? About 2 hours on single GPU for 1000+ skills with 50,000 sequences. Inference is the cheap part - sub-millisecond on CPU. Model checkpoint is around 5 MB. Even resource-constrained systems can run this.

8. Conclusion And Future Work

We presented Skill2Vec, a new embedding method built specifically for technical skills. It has four main components - HCA for hierarchy, SCW for co-occurrence, DCW for adaptive context, GR for structural consistency. Together they capture semantic similarity and taxonomic relationships that standard embeddings miss.

We tested on 2,407 resumes with 50,059 skill mentions. Results show clear improvements - 23.7% better on similarity accuracy, 31.2% on clustering coherence, 18.9% on analogy tasks. All compared against Word2Vec, FastText, and GloVe. We also built a complete system with Flask web app and REST APIs. Similarity search, skill comparison, analogy solving, vocabulary search - all through an interactive interface. It is ready for HR applications.

Where do we go from here? Five directions interest us. First, multilingual support - skills in non-English languages, cross-lingual matching. Second, combining with large language models [20, 21] for better skill extraction. Third, temporal modeling - how skills evolve, which ones are trending, which are declining. Fourth, personalization - embeddings that adapt to industry, seniority, geography. Fifth, proficiency levels - not just whether someone has a skill, but how well.

9. References

1. D. Jatnika, M. A. Bijaksana, and A. A. Suryani, "Word2Vec model analysis for semantic similarities in English words," Procedia Computer Science, vol. 157, pp. 160-167, 2019.
2. T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," arXiv preprint arXiv:1301.3781, 2013.
3. P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," Transactions of the Association for Computational Linguistics, vol. 5, pp. 135-146, 2017.
4. J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in Proc. EMNLP, pp. 1532-1543, 2014.
5. M. E. Peters et al., "Deep contextualized word representations," in Proc. NAACL-HLT, pp. 2227-2237, 2018.
6. J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, pp. 4171-4186, 2019.
7. Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," arXiv preprint arXiv:1907.11692, 2019.
8. V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter," arXiv preprint arXiv:1910.01108, 2019.
9. M. Faruqui et al., "Retrofitting word vectors to semantic lexicons," in Proc. NAACL-HLT, pp. 1606-1615, 2015.
10. A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," in Advances in Neural Information Processing Systems, vol. 26, pp. 2787-2795, 2013.
11. Z. Wang, J. Zhang, J. Feng, and Z. Chen, "Knowledge graph embedding by translating on hyperplanes," in Proc. AAAI Conference on Artificial Intelligence, vol. 28, pp. 1112-1119, 2014.
12. M. Nickel and D. Kiela, "Poincare embeddings for learning hierarchical representations," in Advances in Neural Information Processing Systems, vol. 30, pp. 6338-6347, 2017.
13. M. Zhang, K. Jensen, S. Sonnber, and B. Plank, "SkillSpan: Hard and soft skill extraction from English job postings," in Proc. NAACL, pp. 4962-4984, 2022.
14. J. Li, A. Sun, J. Han, and C. Li, "A survey on deep learning for named entity recognition," IEEE Trans. Knowledge and Data Engineering, vol. 34, no. 1, pp. 50-70, 2022.
15. M. Zhao, F. Javed, F. Jacob, and M. McNair, "Skill extraction from job descriptions using deep learning," in Proc. AAAI Conference on Artificial Intelligence, pp. 5832-5839, 2021.
16. A. Akbik, T. Bergmann, D. Blythe, K. Rasul, S. Schweter, and R. Vollgraf, "FLAIR: An easy-to-use framework for state-of-the-art NLP," in Proc. NAACL-HLT (Demonstrations), pp. 54-59, 2019.

17. T. Schnabel, I. Labutov, D. Mimno, and T. Joachims, "Evaluation methods for unsupervised word embeddings," in Proc. EMNLP, pp. 298-307, 2015.
18. M. Grinberg, Flask Web Development: Developing Web Applications with Python. O'Reilly Media, 2018.
19. K. Papoutsoglou, A. Ampatzoglou, N. Mittas, and L. Angelis, "Extracting knowledge from online sources for software engineering labor market," IEEE Access, vol. 7, pp. 15924-15941, 2019.
20. A. Vaswani et al., "Attention is all you need," in Advances in Neural Information Processing Systems, vol. 30, pp. 5998-6008, 2017.
21. T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785-794, 2016.
22. T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in Advances in Neural Information Processing Systems, vol. 26, pp. 3111-3119, 2013.
23. L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," Journal of Machine Learning Research, vol. 9, pp. 2579-2605, 2008.
24. O. Levy and Y. Goldberg, "Linguistic regularities in sparse and explicit word representations," in Proc. 18th Conference on Computational Natural Language Learning, pp. 171-180, 2014.
25. Y. Kim, "Convolutional neural networks for sentence classification," in Proc. EMNLP, pp. 1746-1751, 2014