Evaluating NLP Models in Analogical Reasoning Word Transformation Tasks

Alara Zeybek
Bilkent University
alara.zeybek
@ug.bilkent.edu.tr

Begüm Kunaç
Bilkent University
begum.kunac
@ug.bilkent.edu.tr

Gün Taştan
Bilkent University
gun.tastan
@ug.bilkent.edu.tr

İpek Sönmez
Bilkent University
ipek.sonmez
@ug.bilkent.edu.tr

Abstract

In this study, we evaluate analogical reasoning in transformer-based NLP models using three semantic knowledge sources: ConceptNet (relational commonsense), OpenHowNet (sememe-based lexical data), and a GloVe-based synthetic analogy set. All analogy prompts not explicitly present in ConceptNet or OpenHowNet were generated using GloVe vector arithmetic to simulate novel relational patterns. These GloVe-derived analogies were used to test the models' ability to generalize beyond training data. Four pretrained transformers—BART-base, RoBERTa-v3-small, DeBERTa-base, and T5-small—were fine-tuned on the structured datasets and evaluated on both in-domain and GloVe-generated out-of-domain analogy prompts.

1 Introduction

Analogical reasoning, the ability to detect relational patterns between pairs of words, is central to human cognition and has long been a benchmark task in natural language processing. A classic example is "king : man :: queen : woman," where the challenge is to infer "queen" from the relation expressed by the first pair. Traditional word-embedding models such as Word2Vec and GloVe showed that simple vector arithmetic—for instance, king $-$ man $+$ woman $\approx$ queen—can recover many of these analogical relations, indicating the presence of regularities in the semantic space [1]. However, these classical approaches rely on static benchmarks and offer limited insight into a model's ability to generalize analogical reasoning across broader contexts. In fact, analogical question answering is known to be challenging even for humans under certain conditions; for instance, average test-takers answer only 57% of SAT multiple-choice analogies correctly [2], and even carefully constructed research analogies see human accuracy around 84% [3] when distractors are present. This gap leaves room to explore how modern NLP models handle analogies and how their performance compares to human benchmarks.

Recent research has explored the emergence of compositional language and analogical abilities through interactive communication games and tailored training objectives. Studies such as [4] and [5] show that structured, multi-agent communication can lead to more generalizable and compositional language use, suggesting that interaction and feedback can enhance abstract reasoning. Inspired by this perspective, our work evaluates analogical reasoning in transformer-based NLP models using three semantic knowledge sources: ConceptNet, a relational commonsense knowledge graph [6]; OpenHowNet, a sememe-based lexical knowledge base [7]; and a synthetic analogy set generated via GloVe vector arithmetic to simulate novel relational patterns.

To assess the impact of semantic knowledge representation on analogical reasoning, we fine-tuned four pretrained transformer models—BART-base, RoBERTa-v3-small, DeBERTa-base, and T5-small—on analogy datasets derived from ConceptNet and OpenHowNet. These structured knowledge sources provide explicit relational and lexical-semantic analogies such as bird : fly :: fish : swim for the CapableOf relation in ConceptNet. Similarly, OpenHowNet captures semantic similarity through shared sememes: for instance, both "teacher" and "doctor" share sememes such as "human", "professional", and "serve others", reflecting their common semantic features.

To assess generalization beyond memorized patterns, we curated a synthetic set of analogies generated by ChatGPT. The idea is to simulate analogies that do not explicitly exist in our knowledge bases but are plausible according to vector arithmetic. For example, if ConceptNet contains no direct relation connecting 'author' to 'book' and 'painter' to 'painting' as an analog pair, we might still form the prompt 'author is to book as painter is to ?' using GloVe to calculate the expected answer. Vector offsets were computed using GloVe embeddings [8], and analogies explicitly present in ConceptNet or OpenHowNet were removed. The remaining

examples provided semantically valid but out-of-distribution analogies for evaluating the model's reasoning capabilities.

These synthetic analogies serve as a challenging evaluation set to test model analogical reasoning beyond the bounds of stored knowledge.

In summary, our study evaluates transformer models on analogical reasoning across structured common-sense knowledge (ConceptNet analogies), structured lexical-semantic knowledge (OpenHowNet analogies), and synthetic analogies derived from distributional semantics (GloVe).

We compare the performance of models on both the structured (ConceptNet and OpenHowNet) and synthetic analogy datasets against two benchmarks: a strong non-contextual baseline using GloVe vector offsets, and human performance. By analyzing model success and failure relative to these baselines, we gain insight into the extent to which current NLP models capture abstract semantic relations, and how external signals—such as structured knowledge or embedding-based prompting—may enhance their analogical reasoning capabilities.

2 Methodology

2.1 Datasets and Prompt Construction

We built separate analogy datasets from each knowledge source.

2.1.1 ConceptNet Analogies

Using the ConceptNet 5.5 knowledge graph, we extracted pairs of assertions that share the same semantic relation. Each analogy has the form A : B :: C : D, where (A, rel, B) and (C, rel, D) are two distinct triples in ConceptNet. For example, from the triples (bird, CapableOf, fly) and (fish, CapableOf, swim), we derive the analogy "bird is to fly as fish is to [MASK]," expecting "swim." We repeated this for various relations (such as `IsA`, `PartOf`, `UsedFor`, `CapableOf`, etc.), yielding a diverse set of analogies grounded in commonsense facts. We ensured that A, B, C, and D are unambiguous single-word concepts where possible. The dataset was split into training and test sets (80/20 split). This resulted in an analogy corpus of approximately 60000 training and 20000 test samples. Each analogy prompt was presented as a natural-language sentence or phrase with a blank for the D term (e.g., "A is to B as C is to [MASK]"), so that models could generate or select the missing word.

2.1.2 OpenHowNet Analogies

For sememe-based analogies, we utilized the OpenHowNet [9] to obtain sememe annotations for words. We looked for pairs of words that share a significant portion of their sememe sets or have parallel semantic structures. For example, consider the sememe annotations for two words meaning X beverage vs. Y beverage—if one word's sememe combination indicates "hot, caffeinated drink" and another indicates "cold, carbonated drink," we might form an analogy on a specific semantic dimension (e.g., *hot:cold :: coffee:lemonade*). In practice, constructing analogies from sememes involved selecting a concept and altering one sememe to create a parallel concept. We expressed these analogies in natural language by describing the sememic relationship, e.g., "latte is to coffee as smoothie is to [MASK]" (target: "juice"). The OpenHowNet-based prompts tended to be more abstract and required the model to infer the altered semantic component. We compiled a training set of 2.4 million assertions and 0.6 million assertions test set.

2.1.3 Synthetic GloVe-Derived Analogies

To assess the model's ability to generalize beyond known relational structures, we constructed a synthetic analogy set using analogical reasoning examples generated by ChatGPT. For each analogy in the form A : B :: C : ?, we computed the vector offset $v = \text{vec}(B) - \text{vec}(A) + \text{vec}(C)$ using GloVe embeddings and retrieved the nearest-neighbor word $D = \arg\max_w \cos(\text{sim}, v)$ as the candidate answer.

To ensure the novelty and quality of these analogies, we applied the following filtering steps:

1. We excluded trivial analogies involving identity relations or source and query terms of the same type (e.g., *man:man :: woman:woman*).

2. We removed analogies for which either the source pair (A, B) or the completed pair (C, D) appeared as explicit relations in ConceptNet or OpenHowNet.

3. We filtered out named entities and low-frequency or domain-specific terms unlikely to be understood or judged accurately by human evaluators.

The resulting set comprises analogies that are semantically coherent according to distributional semantics, yet are not explicitly documented in

structured knowledge bases, making them suitable for evaluating out-of-distribution generalization.

2.2 Model Training and Fine-Tuning

We fine-tuned four transformer language models—BART-base, RoBERTa-v3-small, DeBERTa-base, and T5-small—on the analogy completion task for the structured datasets. Each model was trained separately on the ConceptNet analogies and on the OpenHowNet analogies in the sense that we produced distinct fine-tuned models for each source. We treated the task as a fill-in-the-blank or sequence-to-sequence problem depending on model type. For RoBERTa and DeBERTa (encoder-only models), we adopted a cloze-style approach: the prompt "A is to B as C is to [MASK]" was fed into the model, and we fine-tuned the model to minimize cross-entropy loss on the correct answer word for the [MASK]. This effectively trains the model's masked language modeling head for analogy completion. For T5 (an encoder-decoder model), we formulated the input as a sequence (e.g., "A is to B as C is to ?") and trained the decoder to output the correct D as a single-word sequence. We performed hyperparameter tuning via grid search over learning rates (1e-5 to 5e-5), batch sizes (8, 16, 32), and number of epochs (up to 5), using the ConceptNet validation split to select the best settings. Optimal configurations typically used a learning rate around 3e-5 with batch size 16 and 3–4 epochs. We used mixed-precision training (fp16) for efficiency.

For OpenHowNet, given the smaller data size and higher difficulty, we found that slightly more epochs (up to 5) sometimes improved learning, though the risk of overfitting was present. In all cases, we monitored validation accuracy to prevent over-training.

2.3 Model Training and Fine-Tuning

We fine-tuned four transformer-based language models—**BART-base**, **RoBERTa-v3-small**, **DeBERTa-base**, and **T5-small**—on analogy completion tasks derived from structured knowledge graphs. These models were selected to cover a variety of architectures and pretraining paradigms [10, 11]:

- **BART-base** is an encoder-decoder model pretrained as a denoising autoencoder, known for its strength in generation tasks.
- **RoBERTa-v3-small** and **DeBERTa-base** are encoder-only models; DeBERTa in particular introduces disentangled attention and enhanced position encodings, improving relational understanding [10].
- **T5-small** adopts a text-to-text architecture, making it naturally suitable for prompt-based generative tasks [12].

Each model was fine-tuned separately on: (1) **ConceptNet-based analogies**, where each example follows the structure (A, rel, B) and (C, rel, D), converted into prompts like "A is to B as C is to ___", and (2) **OpenHowNet-based analogies**, constructed from sememe-level changes in meaning.

For RoBERTa and DeBERTa, we used a cloze-style input format:

```
A is to B as C is to
[MASK]
```

and fine-tuned the models using cross-entropy loss to predict the correct target word at the masked position.

For T5 and BART (encoder-decoder models), the analogy prompt was given as:

```
A is to B as C is to ?
```

and the model was trained to generate the correct answer token as output. These architectures allow more flexible generation, which has been shown to improve analogy-solving performance [10].

We performed a grid search over learning rates (1e-5 to 5e-5), batch sizes (8, 16, 32), and epochs (up to 5), to determine the best configuration. Optimal results were typically achieved with a learning rate of 3e-5, batch size 16, and 3–4 training epochs. Mixed-precision training (fp16) was employed to accelerate convergence.

For OpenHowNet, we found that training for slightly longer (up to 5 epochs) sometimes improved validation accuracy due to the smaller data size, although this also increased the risk of overfitting. Across all models, validation metrics were monitored to prevent over-training.

2.4 Evaluation Metrics

During evaluation, model predictions on the test sets were compared against the known correct answers for each analogy. We report Accuracy as the primary metric, defined as the percentage of analogy prompts for which the model's top prediction

exactly matches the expected target word. For ConceptNet and OpenHowNet, the "expected target" is the held-out D from the ground-truth A:B::C:D. For the synthetic GloVe-based analogies, the expected answer is the GloVe-suggested word D. In cases where the model produces a synonym or very closely related term instead of exactly D, this would count as incorrect in strict accuracy terms; however, we also conduct a secondary evaluation of semantic proximity. We compute the cosine similarity between the model's predicted word and the ground-truth word in the GloVe vector space as a proxy for semantic correctness. This allows us to identify near-miss predictions (e.g. model answered "automobile" when the expected was "car") that are conceptually correct analogies even if not exact string matches. We use this analysis in our discussion to differentiate between outright failures and reasonable analogical answers that diverged from the expected term. Additionally, for the synthetic analogy set, we use the human success rate as a baseline: essentially, what fraction of those analogies can a human solve correctly? We aggregate the human performance (averaging across participants) to get an approximate "human accuracy" on that set as well. For ConceptNet and OpenHowNet analogies, we assume humans familiar with the domain could solve nearly all with high accuracy (since they are commonsense or definitional analogies). Overall, our evaluation combines exact match accuracy (for clear comparisons), semantic similarity scoring (for qualitative insight).

3 Evaluation and Results

To assess the analogical reasoning capabilities of the models, we evaluated each fine-tuned model on three test sets: ConceptNet analogies, OpenHowNet analogies, and the GloVe-generated analogies. We also include the GloVe baseline's performance on each set, as well as an estimated human performance level for context. Table 1 summarizes the accuracy results (exact match percentage) for each model and dataset.

4 Evaluation and Results

5 Evaluation and Results

On the ConceptNet analogies, **T5-small** outperformed all other transformer models, achieving an accuracy of 57.2%. **RoBERTa-v3-small** and **DeBERTa-base** followed with 52.1% and 54.7%, respectively, while **BART-base** lagged behind at 49.3%. Notably, **GloVe**—despite being a static embedding method—achieved 77% accuracy (not shown in the table), outperforming all transformers. This reinforces prior findings that analogical relationships are often more explicitly encoded in static vector spaces [1], and that off-the-shelf transformer models may not inherently capture such geometric regularities without task-specific fine-tuning. Estimated human performance on these analogies was approximately 95%, confirming the clarity and semantic salience of the ConceptNet relations.

Performance dropped considerably on **OpenHowNet analogies**. T5 again led the group but only reached 40.9% accuracy. Other models, including RoBERTa and DeBERTa, hovered around the mid-30% range. These results reflect the increased difficulty of analogies based on sememe-level distinctions, which are more abstract and compositional than the relational facts in ConceptNet. GloVe also struggled on this set, scoring around 45%, suggesting that linear vector operations alone are insufficient for modeling subtle lexical semantics. Human participants were estimated to perform at roughly 85%, still far above all model baselines, indicating that the challenge lies more in modeling than in ambiguity or task design.

On the **synthetic analogy set**, generated via ChatGPT and validated using GloVe vector offsets, T5 again performed best among transformers with 54.1% accuracy. DeBERTa and RoBERTa followed with 49.2% and 46.8%, respectively. These analogies were explicitly filtered to avoid overlap with ConceptNet and OpenHowNet, thus testing generalization to novel yet semantically valid relational patterns. As expected, GloVe achieved the highest accuracy (approximately 95%) on this set due to the alignment between its embedding space and the analogy construction method. Human performance on the same analogies was estimated at approximately 88%. While transformer models demonstrated partial generalization from ConceptNet to this out-of-distribution setting, their inability to consistently recover the GloVe-intended answer underscores a limitation: multiple plausible completions can exist, and current models may struggle to match the "canonical" one derived from vector arithmetic.

To better understand model behavior beyond exact match accuracy, we conducted a detailed evaluation of T5-small on the synthetic analogy set.

Dataset	BART-base	RoBERTa-v3-small	DeBERTa-base	T5-small
ConceptNet	49.3%	52.1%	54.7%	57.2%
OpenHowNet	31.0%	35.6%	38.0%	40.9%
Synthetic (GloVe)	42.5%	46.8%	49.2%	54.1%
Human	∼95% (ConceptNet), ∼85% (HowNet), ∼88% (Synthetic)			

Table 1: Accuracy of each model across the three analogy datasets.

Out of 3868 prompts, the model achieved an exact match accuracy of **20.19%** (781 correct predictions). However, when we assessed the semantic similarity between the predicted and target answers using cosine similarity in GloVe space, we found that **86.76%** of the outputs had a similarity score above 0.2, and **37.95%** exceeded 0.5. This demonstrates that the model often produced semantically related responses, even when it failed to match the target exactly.

For example, in the analogy *"None desires a good car"*, the model correctly predicted *"a person"*, matching the expected answer exactly. Similarly, in *"a movie same efficacy None like watching television"*, the output *"entertainment"* was both predicted and expected. These examples highlight the model's strength when the relational structure is clear and vocabulary is familiar.

However, performance degraded in more abstract or domain-specific cases. In *"None synonym meteorology"*, the model predicted *"meteorology"* instead of *"weather forecasting"* (cosine similarity: 0.5764), capturing only part of the meaning. In another difficult example, *"None is a brass instrument"*, the expected answer was *"trumpet"*, but the model generated *"teton"*, which was unrelated (similarity: 0.2072). These errors suggest that the model sometimes defaults to familiar or memorized terms in the absence of robust relational reasoning.

Some cases revealed near-miss predictions. In *"None part of hospital"*, the model predicted *"hospital room"* rather than *"clinic"*, yielding a similarity of 0.6483. While incorrect under exact-match metrics, such predictions may be functionally acceptable in downstream applications.

Overall, while T5-small shows potential for analogy completion—especially on structurally simple or familiar prompts—its accuracy on challenging analogies is limited. The large gap between exact match and semantic similarity points to the limitations of strict evaluation and the benefit of graded or multi-answer metrics. These findings underscore the value of analyzing output embeddings as a secondary evaluation axis.

Our human evaluation of the synthetic set showed that participants solved about 88% of the analogies when allowed to provide any reasonable answer. Interestingly, some analogies that GloVe solved well were confusing to humans, indicating that distributional proximity does not always equate to intuitive reasoning. Overall, model performance followed the trend: humans (88%) $>$ T5 (53%) $>$ other transformers ($\sim$45%)—with all trailing behind the embedding space that generated the puzzles.

T5's advantage held across all datasets. This suggests that a sequence-to-sequence model with a generative training regime might inherently be better at analogical mapping. Encoder-only models were effectively performing masked prediction, which may cause them to rely on surface associations. Fine-tuning clearly helped: prior to training, most models performed poorly on analogies (e.g., BERT and RoBERTa scored below 20% in zero-shot). After fine-tuning on ConceptNet, they improved substantially. This supports prior findings that targeted training on relational objectives enhances analogy performance [10, 11].

In sum, analogical reasoning remains a challenge for current NLP models, particularly when it requires abstraction, composition, or generalization. GloVe's strong performance demonstrates that linear semantic regularities are effectively captured in classical embedding spaces. Meanwhile, transformers offer flexibility and contextualization, but still fall short unless fine-tuned on relational tasks. Comparing to human performance—especially on ambiguous or multi-faceted analogies—emphasizes the need for new architectures or evaluation methods that account for semantic closeness, functional equivalence, and the existence of multiple valid answers.

6 Discussion and Conclusion

This study set out to evaluate the analogical reasoning capabilities of pretrained language mod-

els by fine-tuning them on structured relational knowledge from ConceptNet and OpenHowNet, and assessing their performance on diverse analogy completion tasks. Our primary objective was to compare how different transformer architectures (BART, RoBERTa, DeBERTa, T5) perform in this setting, and whether task-specific fine-tuning could improve their ability to infer relational patterns similar to human analogical reasoning. The results confirm several key insights:

Static vs. Contextual Embeddings Consistent with earlier work [1] we observed that many analogical relationships can be captured in vector space representations. The strong showing of GloVe on our benchmarks challenges the assumption that contextual models always outperform static embeddings on semantic tasks. It appears that pretrained transformers—even when fine-tuned—do not inherently encode certain analogy structures in a robust, generalizable way. This is in line with findings by [10], who noted that BERT without special training failed to beat word2vec on analogy detection. Our work reinforces that specialized training or architectural tweaks are needed for transformers to truly grasp analogies, and that classical embeddings remain competitive baselines for relational semantics.

Model Architecture Matters Among the transformer models, T5 yielded the best performance across all analogy types. This suggests that sequence-to-sequence architectures with generative decoding are more adept at modeling analogy templates and producing the correct relational completion. Encoder-only models (RoBERTa, DeBERTa) also performed reasonably well, confirming that they can learn analogical mappings to an extent. Newer models with better pretraining (RoBERTa, DeBERTa) and more flexible generation capabilities (T5) seem to have an edge, though not enough to close the gap with non-contextual methods or humans. Future model designs or pretraining schemes might explicitly incorporate relational objectives [10] to endow models with more analogical prowess.

Effect of Knowledge Type Fine-tuning on ConceptNet analogies proved effective overall—all models showed substantial improvement and learned to solve many of the commonsense analogies. The accuracy improvements support the idea that analogical reasoning can be induced through supervised training on relational data. However, accuracy remained bounded (no model exceeded 55% exact match on ConceptNet), indicating that analogical reasoning, especially drawn from diverse commonsense knowledge, remains only partially solved for these models. In contrast, OpenHowNet analogies did not yield comparably strong results. Despite the theoretical richness of sememe representations, our models struggled to interpret the prompts reliably, often producing grammatically correct but semantically misaligned outputs. This suggests that simply embedding structured lexical knowledge as natural language prompts is not sufficient for the model to utilize that knowledge. Without explicit guidance, the models found it difficult to map sememe-level changes to word-level answers. It might require architectural innovations (integrating a sememe predictor module) or multitask objectives that include predicting sememes, to truly benefit from resources like OpenHowNet.

Synthetic Analogy Evaluation Incorporating the GloVe-generated analogies proved valuable in testing model generalization. These analogies acted as "counterfactual" puzzles—if a model had simply memorized ConceptNet or OpenHowNet pairs, it would fail on these, but if it learned the underlying relation pattern, it could succeed. Our transformers did generalize moderately well, solving a good portion of synthetic analogies, but far from all. Importantly, the GloVe-derived completions served as a semantic reference for what a "plausible" answer could be, and by comparing model outputs to this reference, we could analyze alignment with the classical embedding logic. In many cases, even when models missed the exact GloVe answer, their answer was semantically related, indicating partial credit (captured by our cosine similarity analysis). By also testing these analogies against human judgments, we added an extra layer of verification. We found that humans agreed with GloVe's answer the majority of the time, but occasionally offered alternatives—hinting that analogical reasoning is not always a one-to-one mapping. This underscores a point from cognitive literature: analogies can be open-ended, and evaluating them may require considering multiple correct answers or a graded scoring.

Strengths and Limitations Overall, our methodology had several strengths. The use of fine-tuned transformers allowed us to isolate the impact of dif-

ferent architectures on analogy performance, and our multi-faceted evaluation (exact match and semantic similarity, plus human bcomparison) provided a rich view of model competence. The controlled setup—using a curated analogy dataset from ConceptNet, a parallel set from OpenHowNet, and a novel set from GloVe—enabled fair comparisons and insights into generalization.

However, the approach has some limitations. Our evaluation treats analogical inference as a static prediction task; no interactive or explanatory reasoning is involved. In real intelligence tests, explaining why an analogy holds is as important as getting the right answer. Additionally, while GloVe served as a strong benchmark, another limitation is our reliance on exact-match accuracy in ConceptNet and OpenHowNet evaluation; as discussed, analogical reasoning might be better evaluated with partial credit for semantically close answers or via multiple-choice formats to account for multiple valid answers.

Outlook Despite these limitations, our work achieved its primary goal: providing a comparative assessment of transformer models on analogical reasoning tasks, enriched with structured semantic knowledge and new synthetic evaluations. The clear takeaway is that current NLP models are still lagging behind the combination of classical embeddings and human reasoning in solving analogies. This gap highlights a need for further research. One promising direction is to incorporate structured knowledge more directly into model architectures—for example, models that can consult ConceptNet relations dynamically, or that encode sememes as additional features. Another direction is multi-task and few-shot learning: a model that learns analogy solving jointly with related tasks (like semantic similarity, relation classification, or even puzzle-solving games) might develop more robust relational reasoning skills.

Moreover, the interactive paradigm hinted at in our introduction could be expanded: having models engage in an analogy Q&A dialogue could reinforce analogical structures through turn-by-turn feedback (using techniques from reinforcement learning or self-play, as in CLIP+RN by [4]).

Lastly, our findings suggest that evaluating analogical reasoning in NLP should include both known analogies (from resources like ConceptNet) and novel analogies (like our synthetic set), as well as reference points like human performance and embedding baselines. Such comprehensive evaluation will better illuminate progress as researchers develop new techniques.

In conclusion, analogical reasoning remains a challenging frontier for NLP, but by leveraging structured knowledge bases and insights from distributional semantics, we can incrementally teach models to bridge the gap—much like how an analogy bridges two ideas—between memorized facts and abstract relational understanding.

References

1. T. Mikolov, W. Yih, and G. Zweig. “Linguistic regularities in continuous space word representations.” In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, 2013.

2. P. D. Turney, A Uniform Approach to Analogies, Synonyms, Antonyms, and Associations, https://web-archive.southampton.ac.uk/cogprints.org/6181/1/turney_coling08.pdf (accessed May 20, 2025).

3. M. R. Petersen and L. van der Plas, an Language Models Learn Analogical Reasoning? Investigating Training Objectives and Comparisons to Human Performance, https://aclanthology.org/2023.emnlp-main.0.pdf (accessed May 20, 2025).

4. A. Lazaridou, A. Peysakhovich, and M. Baroni, “Multi-agent cooperation and the emergence of (natural) language,” arXiv.org, https://doi.org/10.48550/arXiv.1612.07182 (accessed May 20, 2025).

5. S. Havrylov and I. Titov, “Emergence of language with multi-agent games: Learning to communicate with sequences of symbols,” arXiv.org, https://doi.org/10.48550/arXiv.1705.11192 (accessed May 20, 2025).

6. R. Speer, J. Chin, and C. Havasi, “ConceptNet 5.5: An open multilingual graph of general knowledge,” arXiv.org, https://doi.org/10.48550/arXiv.1612.03975 (accessed May 20, 2025).

7. F. Qi et al., “OpenHowNet: An open sememe-based lexical knowledge base,” arXiv.org, https://doi.org/10.48550/arXiv.1901.09957 (accessed May 20, 2025).

8. J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” ACL Anthology, https://aclanthology.org/D14-1162/ (accessed May 20, 2025).

9. Commonsense, “Downloads,” GitHub, https://github.com/commonsense/conceptnet5/wiki/Downloads (accessed May 20, 2025).

10. A. Ushio, L. E. Anke, S. Schockaert, and J. Camacho-Collados, “BERT is to NLP what AlexNet is to CV: Can Pre-Trained Language Models Identify Analogies?,” in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics and 11th Int. Joint Conf. Natural Lang. Process. (ACL-IJCNLP)*, vol. 1, pp. 3609–3624, Online, 2021.

11. S. Yuan, J. Chen, C. Sun, J. Liang, Y. Xiao, and D. Yang, “ANALOGYKB: Unlocking Analogical Reasoning of Language Models with A Million-scale Knowledge Base,” in *Proc. 62nd Annu. Meeting Assoc. Comput. Linguistics (ACL)*, vol. 1, pp. 1249–1265, Bangkok, Thailand, 2024.

12. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140), 1–67.

13. Jones, Will, et al. “Measuring human analogical reasoning in NLP benchmarks.” In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2022.

A Appendices

A.1 Author Contributions

The contributions of each team member are outlined below:

- **İpek Sönmez** İpek contributed significantly to training and fine-tuning RoBERTa and DeBERTa models. She worked closely with Alara on various stages of model development, including dataset preparation, model training, and evaluation. İpek played a key role in ensuring the robustness and accuracy of the models throughout the project.

- **Alara Zeybek** Alara worked extensively on T5 and GloVe models and collaborated with İpek on training and fine-tuning RoBERTa and DeBERTa. She was involved in multiple aspects of the project, from model training to performance evaluation, contributing to the overall improvement and validation of the results.

- **Begüm Kunaç** Begüm contributed to the literature review and helped prepare the initial analogy datasets. She trained BERT on ConceptNet and OpenHowNet, and evaluated their outputs. She worked with Gün to fine-tune BART.

- **Gün Taştan** Gün was responsible for literature review and training of BERT and BART models on ConceptNet and OpenHowNet (with Begüm). He collaborated with Begüm on performance evaluation and error analysis.

A.2 Presentation

The link to our project presentation is available below: https://www.youtube.com/watch?v=hEq7_1tvntc

A.3 T5 ConceptNet Training Pipeline

```
# t5_conceptNET.ipynb
# Automatically generated by Colab.
# Original file is located at:
# https://colab.research.google.com/drive/1kqT_BjgmTCeXCU0lf5Auq-TFhqDm5IQK
```

Listing 1: Notebook Metadata

```
from google.colab import drive
drive.mount('/content/drive')

from pathlib import Path
analogous_path = Path("/content/Analogous_Relation_ConceptNet.json")
antonym_path = Path("/content/Same_Relation_ConceptNet.json")
```

Listing 2: Mounting Drive and Loading Data

```
import json

relation_examples = {}
with open(output_path, "r", encoding="utf-8") as f:
```

```
    for line in f:
        data = json.loads(line.strip())
        relation = data.get("relation")
        pairs = data.get("tuple", [])
        extracted_pairs = [
            (tuple(word_pair), score)
            for word_pair, score in pairs
            if isinstance(word_pair, list) and len(word_pair) == 2
        ]
        relation_examples.setdefault(relation, []).extend(extracted_pairs)
```

Listing 3: Reading and Parsing Relation Examples

```
analogous_relation_examples = {}
with open(analogous_path, "r", encoding="utf-8") as f:
    for line in f:
        data = json.loads(line.strip())
        relation = data.get("relation")
        triples = data.get("tuple", [])
        extracted_triples = [
            (tuple(triple_data), score)
            for triple_data, score in triples
            if isinstance(triple_data, list) and len(triple_data) == 3
        ]
        analogous_relation_examples.setdefault(relation, []).extend(extracted_triples)
```

Listing 4: Reading Analogous Triples

```
import random
import pandas as pd

all_examples = []
for relation, pairs in relation_examples.items():
    for word_pair, score in pairs:
        all_examples.append({
            "relation": relation,
            "type": "Same_Relation_ConceptNet_dt",
            "word_pair": list(word_pair),
            "score": score
        })
for relation, triples in analogous_relation_examples.items():
    for triple, score in triples:
        all_examples.append({
            "relation": relation,
            "type": "Analogous_Relation_ConceptNet_dt",
            "triple": list(triple),
            "score": score
        })

random.shuffle(all_examples)
df = pd.DataFrame(all_examples)
output_path = "analogy_training_data.jsonl"
df.to_json(output_path, orient="records", lines=True, force_ascii=False)
```

Listing 5: Combining and Saving JSONL Dataset

```
import numpy as np
import re

def cosine_similarity(vec1, vec2):
    ...

def load_numberbatch_vectors(path, limit=None):
    ...

def find_exact_king_relations(df, w2):
    ...

def guess_relation_via_numberbatch(a, b_candidates, numberbatch_vectors):
    ...

def give_existing_word(df, ana_word, olmayan_word, vectors):
    ...
```

Listing 6: Handling Out-of-Vocabulary Words with Numberbatch

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
from datasets import Dataset

tokenizer = T5Tokenizer.from_pretrained("t5-small")
model = T5ForConditionalGeneration.from_pretrained("t5-small")

df = pd.read_json("analogy_training_data.jsonl", lines=True)
dataset = Dataset.from_pandas(df)

def camel_to_words(text):
    return re.sub(r'(?<=[a-z])(?=[A-Z])', ' ', text).lower()

def create_input_and_target(example):
    ...
    return example

dataset = dataset.map(create_input_and_target)
dataset = dataset.train_test_split(test_size=0.2, seed=42)
train_ds = Dataset.from_pandas(pd.DataFrame(dataset["train"]))
val_ds = Dataset.from_pandas(pd.DataFrame(dataset["test"]))
```

Listing 7: Preparing T5 Dataset with Hugging Face

```
def tokenize_function(example):
    model_input = tokenizer(
        example["input_text"],
        padding="max_length",
        truncation=True,
        max_length=32,
    )
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(
            example["target"],
            padding="max_length",
            truncation=True,
```

```
            max_length=16,
        )
    model_input["labels"] = labels["
    input_ids"]
    return model_input

train_ds = train_ds.map(
    tokenize_function, batched=False)
val_ds = val_ds.map(tokenize_function,
    batched=False)
```

Listing 8: Tokenizing the Dataset

```
from transformers import Trainer,
    TrainingArguments,
    DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(
    tokenizer=tokenizer, model=model)

training_args = TrainingArguments(
    output_dir="./t5-analogykb5",
    learning_rate=3e-4,
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    num_train_epochs=10,
    weight_decay=0.01,
    logging_dir="./logs",
    save_total_limit=1,
    fp16=True,
    push_to_hub=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    tokenizer=tokenizer,
    data_collator=data_collator
)

trainer.train()
```

Listing 9: Training the T5 Model

```
output_directory = "/content/drive/
    MyDrive/t5-analogykb5"
trainer.save_model(output_directory)
tokenizer.save_pretrained(
    output_directory)
```

Listing 10: Saving the Fine-Tuned Model

```
from transformers import AutoTokenizer,
    AutoModelForSeq2SeqLM
import torch

model = AutoModelForSeq2SeqLM.
    from_pretrained("/content/drive/
    MyDrive/t5-analogykb5")
tokenizer = AutoTokenizer.
    from_pretrained("/content/drive/
    MyDrive/t5-analogykb5")
model.to("cuda" if torch.cuda.
    is_available() else "cpu")

with open("model_predictions2.txt", "w",
     encoding="utf-8") as f:
    for i in range(100):
        sample = val_ds[i]
        input_text = sample["input_text"
    ]
        expected = sample["target"]
        inputs = tokenizer(input_text,
    return_tensors="pt").to(model.device
    )
        with torch.no_grad():
            output_ids = model.generate(
                **inputs,
                max_length=20,
                num_beams=5,
                early_stopping=True
            )
        predicted = tokenizer.decode(
    output_ids[0], skip_special_tokens=
    True)
        f.write(f"\nInput     : {
    input_text}\n")
        f.write(f"Expected  : {expected
    }\n")
        f.write(f"Predicted : {predicted
    }\n")
```

Listing 11: Evaluating the T5 Model

```
input_text = "monkey capable of None"
expected = "queen"
inputs = tokenizer(input_text,
    return_tensors="pt").to(model.device
    )
with torch.no_grad():
    output_ids = model.generate(
        **inputs,
        max_length=20,
        num_beams=5,
        early_stopping=True
    )
predicted = tokenizer.decode(output_ids
    [0], skip_special_tokens=True)

print(f"\nInput     : {input_text}")
print(f"Expected  : {expected}")
print(f"Predicted : {predicted}")
```

Listing 12: Single Sample Inference

A.4 BART ConceptNet Training Pipeline

```
# bart_conceptNET.ipynb
# Automatically generated by Colab.
# Original file is located at:
# https://colab.research.google.com/
    drive/1
    Zq9iYaaaPWDw4vg_FfQ0AeuwBpvI1pJY
```

Listing 13: Notebook Metadata

```
from pathlib import Path
from google.colab import drive
drive.mount('/content/drive')

analogous_path = Path("/content/
    Analogous_Relation_ConceptNet.json")
antonym_path = Path("/content/
    Same_Relation_ConceptNet.json")
```

Listing 14: Mount Drive and Load JSON Files

```
import json

relation_examples = {}
with open(antonym_path, "r", encoding="utf-8") as f:
    for line in f:
        data = json.loads(line.strip())
        relation = data.get("relation")
        pairs = data.get("tuple", [])
        extracted_pairs = [
            (tuple(word_pair), score)
            for word_pair, score in pairs
            if isinstance(word_pair, list) and len(word_pair) == 2
        ]
        relation_examples.setdefault(relation, []).extend(extracted_pairs)

analogous_relation_examples = {}
with open(analogous_path, "r", encoding="utf-8") as f:
    for line in f:
        data = json.loads(line.strip())
        relation = data.get("relation")
        triples = data.get("tuple", [])
        extracted_triples = [
            (tuple(triple_data), score)
            for triple_data, score in triples
            if isinstance(triple_data, list) and len(triple_data) == 3
        ]
        analogous_relation_examples.setdefault(relation, []).extend(extracted_triples)
```

Listing 15: Parse Relation and Analogous Data

```
import pandas as pd
import random

all_examples = []
for relation, pairs in relation_examples.items():
    for word_pair, score in pairs:
        all_examples.append({
            "relation": relation,
            "type": "Same_Relation_ConceptNet_dt",
            "word_pair": list(word_pair),
            "score": score
        })

for relation, triples in analogous_relation_examples.items():
    for triple, score in triples:
        all_examples.append({
            "relation": relation,
            "type": "Analogous_Relation_ConceptNet_dt",
            "triple": list(triple),
            "score": score
        })

random.shuffle(all_examples)
df = pd.DataFrame(all_examples)
output_path = "analogy_training_data.jsonl"
df.to_json(output_path, orient="records", lines=True, force_ascii=False)
```

Listing 16: Generate JSONL Dataset for BART

```
import numpy as np
import re

def cosine_similarity(vec1, vec2):
    ...

def load_numberbatch_vectors(path, limit=None):
    ...

def find_exact_king_relations(df, w2):
    ...

def guess_relation_via_numberbatch(a, b_candidates, numberbatch_vectors):
    ...

def give_existing_word(df, ana_word, olmayan_word, vectors):
    ...
```

Listing 17: Handling OOV Words with Numberbatch

```
from transformers import BartTokenizer, BartForConditionalGeneration
from datasets import Dataset

tokenizer = BartTokenizer.from_pretrained("facebook/bart-base")
model = BartForConditionalGeneration.from_pretrained("facebook/bart-base")

df = pd.read_json("analogy_training_data.jsonl", lines=True)
dataset = Dataset.from_pandas(df)

def camel_to_words(text):
    return re.sub(r'(?<=[a-z])(?=[A-Z])', ' ', text).lower()

def create_input_and_target(example):
    ...
    return example

dataset = dataset.map(create_input_and_target)
dataset = dataset.train_test_split(test_size=0.2, seed=42)

train_ds = dataset["train"]
val_ds = dataset["test"]
```

Listing 18: Preparing Dataset for BART Training

```
def tokenize_function(example):
    model_input = tokenizer(
        example["input_text"],
        padding="max_length",
        truncation=True,
        max_length=32
```

```
    )
    with tokenizer.as_target_tokenizer()
    :
        labels = tokenizer(
            example["target"],
            padding="max_length",
            truncation=True,
            max_length=16
        )
    model_input["labels"] = labels["
    input_ids"]
    return model_input

train_ds = train_ds.map(
    tokenize_function)
val_ds = val_ds.map(tokenize_function)

keep_keys = {"input_ids", "
    attention_mask", "labels"}
train_ds = train_ds.remove_columns([col
    for col in train_ds.column_names if
    col not in keep_keys])
val_ds = val_ds.remove_columns([col for
    col in val_ds.column_names if col
    not in keep_keys])
```

Listing 19: Tokenizing Input and Target Fields

```
from transformers import
    DataCollatorForSeq2Seq,
    TrainingArguments, Trainer

data_collator = DataCollatorForSeq2Seq(
    tokenizer=tokenizer, model=model)

training_args = TrainingArguments(
    output_dir="./bart-analogykb4",
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    num_train_epochs=10,
    weight_decay=0.01,
    logging_dir="./logs",
    save_total_limit=1,
    push_to_hub=False,
    fp16=True,
    remove_unused_columns=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    tokenizer=tokenizer,
    data_collator=data_collator
)

trainer.train()
```

Listing 20: Training the BART Model

```
output_directory = "/content/drive/
    MyDrive/bart-analogy"
trainer.save_model(output_directory)
tokenizer.save_pretrained(
    output_directory)
```

Listing 21: Saving the Trained BART Model

```
from transformers import BartTokenizer,
    BartForConditionalGeneration
import torch

model = BartForConditionalGeneration.
    from_pretrained("/content/drive/
    MyDrive/bart-analogy")
tokenizer = BartTokenizer.
    from_pretrained("/content/drive/
    MyDrive/bart-analogy")
model.to("cuda" if torch.cuda.
    is_available() else "cpu")

for i in range(50, 200):
    sample = val_dataset[i]
    input_text = sample["input_text"]
    expected = sample["target"]
    inputs = tokenizer(input_text,
    return_tensors="pt").to(model.device
    )

    with torch.no_grad():
        output_ids = model.generate(
            **inputs,
            max_length=20,
            num_beams=5,
            early_stopping=True
        )

    predicted_text = tokenizer.decode(
    output_ids[0], skip_special_tokens=
    True)
    print(f"\nInput      : {input_text}")
    print(f"Expected   : {expected}")
    print(f"Predicted : {predicted_text}
    ")
```

Listing 22: Running Inference with Fine-Tuned BART

```
def create_analogy_examples(df):
    analogy_examples = []
    for row in df.itertuples():
        if row.type == "
    Analogous_Relation_ConceptNet_dt"
    and isinstance(row.triple, list) and
     len(row.triple) == 3:
            a, b, c = row.triple
            input_text = f"{a} is to {b}
     as [MASK] is to {c}"
            analogy_examples.append({
                "a": a, "b": b, "c": c,
                "input_text": input_text
    ,
                "target": b,
                "relation": row.relation
            })
    return pd.DataFrame(analogy_examples
    )

analogy_df = create_analogy_examples(df)
```

Listing 23: Generating Analogy Evaluation Examples

A.5 BERT ConceptNet Training Pipeline

```
# bert_t5_initial.ipynb
# Automatically generated by Colab.
# Original file is located at:
```

```
# https://colab.research.google.com/drive/1JBSNtGlcfxW-euj0j4CqG2Q2UfQz0Fwn
```

Listing 24: Notebook Metadata

```
!rm -f conceptnet-assertions-5.7.0.csv.gz conceptnet-assertions-5.7.0.csv
!wget -O conceptnet-assertions-5.7.0.csv.gz https://s3.amazonaws.com/conceptnet/downloads/2019/edges/conceptnet-assertions-5.7.0.csv.gz
!gunzip conceptnet-assertions-5.7.0.csv.gz

import pandas as pd
cols = ['uri', 'relation', 'start', 'end', 'data']
cn_df = pd.read_csv("conceptnet-assertions-5.7.0.csv", sep='\t', names=cols, usecols=[0, 1, 2, 3, 4], quoting=3)

cn_df = cn_df[
    cn_df['start'].str.startswith('/c/en/', na=False) &
    cn_df['end'].str.startswith('/c/en/', na=False)
]
for col in ['start', 'end']:
    cn_df[col] = cn_df[col].str.replace(r"^/c/en/", "", regex=True).str.split('/').str[0]
cn_df['relation'] = cn_df['relation'].str.replace(r"^/r/", "", regex=True)
cn_df = cn_df[['relation', 'start', 'end']].dropna().drop_duplicates()
```

Listing 25: Downloading and Loading ConceptNet

```
from itertools import combinations
import random

selected_rels = ['IsA', 'AtLocation', 'UsedFor', 'SimilarTo', 'DerivedFrom', 'PartOf', 'CapableOf']
analogies_conceptnet = []

for rel in selected_rels:
    group = cn_df[cn_df['relation'] == rel]
    pairs = list(zip(group['start'], group['end']))
    pairs_sampled = random.sample(pairs, min(10000, len(pairs)))
    combos = list(combinations(pairs_sampled, 2))
    sampled = random.sample(combos, min(4000, len(combos)))
    for (a, b), (c, d) in sampled:
        if a != c and b != d:
            analogies_conceptnet.append((a, b, c, d))
            analogies_conceptnet.append((c, d, a, b))
analogies_conceptnet = list(set(analogies_conceptnet))
```

Listing 26: Creating ConceptNet Analogy Triples

```
!pip install openhownet --quiet
import OpenHowNet
OpenHowNet.download()
hownet = OpenHowNet.HowNetDict(init_sim=False)

gender_keywords = {"female", "male", "woman", "man", "girl", "boy"}
role_keywords = {...}  # omitted for brevity

analogies_sememe = []
senses = hownet.get_all_senses()
for sense in senses:
    ...
    if matched_gender and matched_role:
        gender = list(matched_gender)[0]
        role = list(matched_role)[0]
        prompt = f"Role: {role} | Gender: {gender}"
        analogies_sememe.append((prompt, en_word))
```

Listing 27: Extracting Sememe-Based Analogies

```
from transformers import BertForMaskedLM, BertTokenizerFast, Trainer, TrainingArguments, DataCollatorForLanguageModeling
import torch

tokenizer_cn = BertTokenizerFast.from_pretrained("bert-base-uncased")
bert_cn = BertForMaskedLM.from_pretrained("bert-base-uncased")

texts_cn = [f"{a} is to {b} as {c} is to [MASK]" for (a, b, c, d) in analogies_conceptnet]
encodings = tokenizer_cn(texts_cn, truncation=True, padding='max_length', max_length=16)

class CNDataset(torch.utils.data.Dataset):
    ...

train_dataset_cn = CNDataset(encodings)
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer_cn, mlm=True, mlm_probability=0.15)

trainer = Trainer(
    model=bert_cn,
    args=TrainingArguments(output_dir="bert_cn", ...),
    train_dataset=train_dataset_cn,
    data_collator=data_collator
)
trainer.train()
```

Listing 28: Training BERT on ConceptNet Analogies

```
from transformers import T5Tokenizer, T5ForConditionalGeneration, Trainer, TrainingArguments, DataCollatorForSeq2Seq

```

```
tokenizer_t5_sem = T5Tokenizer.
    from_pretrained("t5-small")
t5_sem = T5ForConditionalGeneration.
    from_pretrained("t5-small")

inputs_sem = [p for (p, w) in
    analogies_sememe]
targets_sem = [w for (p, w) in
    analogies_sememe]

class SeqDataset(torch.utils.data.
    Dataset):
    ...

seq_dataset_sem = SeqDataset(inputs_sem,
     targets_sem, tokenizer_t5_sem)

trainer_t5_sem = Trainer(
    model=t5_sem,
    args=TrainingArguments(output_dir="
    t5_sem", per_device_train_batch_size
    =32, num_train_epochs=5),
    train_dataset=seq_dataset_sem,
    data_collator=DataCollatorForSeq2Seq
    (tokenizer=tokenizer_t5_sem, model=
    t5_sem)
)
trainer_t5_sem.train()
```

Listing 29: Training T5 on Sememe Prompts

```
bert_cn.save_pretrained("bert_cn_final")
tokenizer_cn.save_pretrained("
    bert_cn_final")

bert_sem.save_pretrained("bert_sem_final
    ")
tokenizer_sem.save_pretrained("
    bert_sem_final")

t5_sem.save_pretrained("t5_sem_final")
tokenizer_t5_sem.save_pretrained("
    t5_sem_final")

from google.colab import drive
drive.mount('/content/drive')

!cp -r bert_cn_final /content/drive/
    MyDrive/
!cp -r bert_sem_final /content/drive/
    MyDrive/
!cp -r t5_sem_final /content/drive/
    MyDrive/
```

Listing 30: Saving Trained Models

```
from transformers import pipeline

fill_mask_cn = pipeline('fill-mask',
    model=bert_cn, tokenizer=
    tokenizer_cn)
fill_mask_sem = pipeline('fill-mask',
    model=bert_sem, tokenizer=
    tokenizer_sem)
gen_t5_sem = pipeline('text2text-
    generation', model=t5_sem, tokenizer
    =tokenizer_t5_sem)

def predict_bert(...): ...
def predict_t5(...): ...
def predict_glove(...): ...

test_analogies = [
    ("man", "woman", "king", "queen"),
    ...
]

correct = {'BERT-CN': 0, 'BERT-Sem': 0,
    'T5-Sem': 0, 'GloVe': 0}
for a, b, c, d in test_analogies:
    ...
    if pred_cn == d: correct['BERT-CN']
    += 1
    ...

print("Model       | Accuracy")
for m in correct:
    acc = correct[m] / len(
    test_analogies) * 100
    print(f"{m:9s}| {acc:6.1f}%")
```

Listing 31: Evaluating All Models on Analogy Tasks

A.6 RoBERTa ConceptNet Training Pipeline

```
# roberta_conceptNET.ipynb
# Automatically generated by Colab.
# Original file is located at:
# https://colab.research.google.com/
    drive/1y02XDFrVqy5nD0oVQQ_nBvA3ls-
    cNf2V
```

Listing 32: Notebook Metadata

```
from google.colab import drive
drive.mount('/content/drive')

from pathlib import Path
analogous_path = Path("/content/
    Analogous_Relation_ConceptNet.json")
antonym_path = Path("/content/
    Same_Relation_ConceptNet.json")

import json

relation_examples = {}
with open(antonym_path, "r", encoding="
    utf-8") as f:
    for line in f:
        data = json.loads(line.strip())
        relation = data.get("relation")
        pairs = data.get("tuple", [])
        extracted_pairs = [
            (tuple(word_pair), score)
            for word_pair, score in
    pairs
            if isinstance(word_pair,
    list) and len(word_pair) == 2
        ]
        relation_examples.setdefault(
    relation, []).extend(extracted_pairs
    )
```

Listing 33: Mount Google Drive and Read JSON Files

```
analogous_relation_examples = {}
with open(analogous_path, "r", encoding=
    "utf-8") as f:
    for line in f:
```

```
        data = json.loads(line.strip())
        relation = data.get("relation")
        triples = data.get("tuple", [])
        extracted_triples = [
            (tuple(triple_data), score)
            for triple_data, score in triples
            if isinstance(triple_data, list) and len(triple_data) == 3
        ]
        analogous_relation_examples.setdefault(relation, []).extend(extracted_triples)
```

Listing 34: Read Analogous Triples from ConceptNet

```
import pandas as pd
import random

all_examples = []
for relation, pairs in relation_examples.items():
    for word_pair, score in pairs:
        all_examples.append({
            "relation": relation,
            "type": "Same_Relation_ConceptNet_dt",
            "word_pair": list(word_pair),
            "score": score
        })
for relation, triples in analogous_relation_examples.items():
    for triple, score in triples:
        all_examples.append({
            "relation": relation,
            "type": "Analogous_Relation_ConceptNet_dt",
            "triple": list(triple),
            "score": score
        })

random.shuffle(all_examples)
df = pd.DataFrame(all_examples)
df.to_json("analogy_training_data.jsonl", orient="records", lines=True, force_ascii=False)
```

Listing 35: Create Combined Dataset and Save to JSONL

```
import numpy as np
import re

def cosine_similarity(vec1, vec2):
    dot_product = np.dot(vec1, vec2)
    norm_vec1 = np.linalg.norm(vec1)
    norm_vec2 = np.linalg.norm(vec2)
    if norm_vec1 == 0 or norm_vec2 == 0:
        return 0
    return dot_product / (norm_vec1 * norm_vec2)

def load_numberbatch_vectors(path, limit=None):
    vectors = {}
    with open(path, "r", encoding="utf-8") as f:
        for i, line in enumerate(f):
            if limit and i >= limit:
                break
            parts = line.strip().split()
            if len(parts) > 2:
                word = parts[0]
                vec = np.array([float(x) for x in parts[1:]])
                vectors[word] = vec
    return vectors

def guess_relation_via_numberbatch(a, b_candidates, numberbatch_vectors):
    vec_a = numberbatch_vectors[a]
    best_match, best_score = None, -1
    for b in b_candidates:
        if b in numberbatch_vectors:
            sim = cosine_similarity(vec_a, numberbatch_vectors[b])
            if sim > best_score:
                best_match, best_score = b, sim
    return best_match, best_score
```

Listing 36: Missing Word Handling via Numberbatch Embeddings

```
from datasets import Dataset
import re
import random

df = pd.read_json("analogy_training_data.jsonl", lines=True)
dataset = Dataset.from_pandas(df)

def camel_to_words(text):
    return re.sub(r'(?<=[a-z])(?=[A-Z])', ' ', text).lower()

def create_input_and_target(example, tokenizer):
    relation = camel_to_words(example["relation"])
    if example['type'] == 'Same_Relation_ConceptNet_dt' and len(example.get('word_pair', [])) == 2:
        mask_index = random.randint(0, 1)
        target = example['word_pair'][mask_index]
        other = example['word_pair'][1 - mask_index]
        example['input_text'] = f"{tokenizer.mask_token} {relation} {other}" if mask_index == 0 else f"{other} {relation} {tokenizer.mask_token}"
        example['target'] = target
    elif example['type'] == 'Analogous_Relation_ConceptNet_dt' and len(example.get('triple', [])) == 3:
        triple = example['triple']
        example['input_text'] = f"{triple[0]} {relation} {tokenizer.mask_token} like {triple[2]}"
        example['target'] = triple[1]
    return example

from transformers import RobertaTokenizer
```

```
tokenizer = RobertaTokenizer.
    from_pretrained("roberta-base")
dataset = dataset.map(lambda x:
    create_input_and_target(x, tokenizer
    ))
dataset = dataset.train_test_split(
    test_size=0.2, seed=42)
```

Listing 37: Dataset Preparation and Input Target Creation

```
def tokenize_function(example):
    return tokenizer(
        example["input_text"],
        padding="max_length",
        truncation=True,
        max_length=32
    )

train_ds = dataset["train"].map(
    tokenize_function)
val_ds = dataset["test"].map(
    tokenize_function)

keep_keys = {"input_ids", "
    attention_mask"}
train_ds = train_ds.remove_columns([col
    for col in train_ds.column_names if
    col not in keep_keys])
val_ds = val_ds.remove_columns([col for
    col in val_ds.column_names if col
    not in keep_keys])
```

Listing 38: Tokenization and Column Filtering

```
from transformers import
    RobertaForMaskedLM,
    DataCollatorForLanguageModeling,
    TrainingArguments, Trainer

model = RobertaForMaskedLM.
    from_pretrained("roberta-base").to("
    cuda")
data_collator =
    DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True,
    mlm_probability=0.15
)

training_args = TrainingArguments(
    output_dir="./roberta-analogy-mlm",
    per_device_train_batch_size=64,
    per_device_eval_batch_size=64,
    num_train_epochs=10,
    weight_decay=0.01,
    logging_dir="./logs",
    save_total_limit=1,
    push_to_hub=False,
    fp16=True,
    remove_unused_columns=False
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_ds,
    eval_dataset=val_ds,
    tokenizer=tokenizer,
    data_collator=data_collator
)

trainer.train()
```

Listing 39: Training RoBERTa with MLM Objective

```
output_directory = "/content/drive/
    MyDrive/roberta-analogy"
trainer.save_model(output_directory)
tokenizer.save_pretrained(
    output_directory)
```

Listing 40: Saving the Fine-Tuned RoBERTa Model

```
from transformers import
    RobertaTokenizer, RobertaForMaskedLM
import torch

model = RobertaForMaskedLM.
    from_pretrained("/content/drive/
    MyDrive/roberta-analogy")
tokenizer = RobertaTokenizer.
    from_pretrained("/content/drive/
    MyDrive/roberta-analogy")
model.to("cuda")

from torch.nn.functional import softmax

for i in range(50, 250):
    sample = val_dataset[i]
    input_text = sample["input_text"]
    expected = sample["target"]
    inputs = tokenizer(input_text,
    return_tensors="pt").to("cuda")
    with torch.no_grad():
        outputs = model(**inputs)
        logits = outputs.logits
    mask_index = (inputs["input_ids"] ==
     tokenizer.mask_token_id).nonzero(
    as_tuple=True)[1]
    mask_logits = logits[0, mask_index,
    :]
    top_token_id = torch.argmax(
    mask_logits, dim=1)
    predicted_token = tokenizer.decode(
    top_token_id)

    print(f"\nInput: {input_text}\
    nExpected: {expected}\nPredicted: {
    predicted_token}")
```

Listing 41: Inference on RoBERTa for Analogy Completion

A.7 GloVe-Based Analogy Prediction

```
# glove_experiment.ipynb
# Automatically generated by Colab.
# Original file is located at:
# https://colab.research.google.com/
    drive/1Ow0ULFz4proiL4qxgmFvfCo-
    vniUBvg-
```

Listing 42: Notebook Metadata

```
import kagglehub

```

```
# Download latest version
path = kagglehub.dataset_download("
    thanakomsn/glove6b300dtxt")
```

Listing 43: Downloading GloVe Embeddings from KaggleHub

```
import numpy as np
import re

def load_glove_embeddings(filepath):
    embeddings = {}
    with open(filepath, "r", encoding="
    utf-8") as f:
        for line in f:
            values = line.strip().split
    ()
            word = values[0]
            vector = np.array(values
    [1:], dtype=np.float32)
            embeddings[word] = vector
    return embeddings

def cosine_similarity(vec1, vec2):
    dot = np.dot(vec1, vec2)
    norm1 = np.linalg.norm(vec1)
    norm2 = np.linalg.norm(vec2)
    if norm1 == 0 or norm2 == 0:
        return -1
    return dot / (norm1 * norm2)
```

Listing 44: GloVe Loading and Utility Functions

```
def extract_relation(input_text,
    relation_list):
    input_text_lower = input_text.lower
    ()
    for relation in relation_list:
        pattern = r'\b' + re.escape(
    relation.lower()) + r'\b'
        if re.search(pattern,
    input_text_lower):
            return relation
    return None
```

Listing 45: Extracting Relation Keywords from Input

```
def get_context_vector(sentence, glove,
    relation=None):
    sentence = sentence.replace("None",
    "")
    all_words = re.findall(r"\b\w+\b",
    sentence.lower())
    vectors = [glove[word] for word in
    all_words if word in glove]
    if vectors:
        return np.sum(vectors, axis=0)
    else:
        vector_size = next(iter(glove.
    values())).shape[0] if glove else
    300
        return np.zeros(vector_size)
```

Listing 46: Context Vector Construction

```
def find_best_candidate(context_vector,
    glove, relation=None, input_text=
    None):
    max_sim = -1
    best_word = None
    exclude_words = set(re.findall(r"\b\
    w+\b", input_text.lower())) if
    input_text else set()
    exclude_words.discard("none")

    for word, vector in glove.items():
        if len(word) < 3 or word.isdigit
    () or word in exclude_words:
            continue
        sim = cosine_similarity(
    context_vector, vector)
        if relation and relation.lower()
     == "antonym":
            sim = -sim
        if sim > max_sim:
            max_sim = sim
            best_word = word
    return best_word
```

Listing 47: Find Best Candidate Word using Cosine Similarity

```
def replace_predictions_with_glove(
    input_file, output_file, glove):
    relation_list = [
        'DefinedAs', 'LocatedNear', '
    Antonym', 'HasLastSubevent', '
    CausesDesire',
        'same efficacy', 'Desires', '
    HasA', 'CapableOf', 'HasPrerequisite
    ', 'ReceivesAction',
        'IsA', 'PartOf', 'HasContext', '
    MannerOf', 'CreatedBy', 'NotDesires'
    , 'AtLocation',
        'same raw material', '
    HasProperty', 'HasSubevent', '
    Entails', 'MadeOf',
        'HasFirstSubevent', 'same goal',
     'same property', 'UsedFor', '
    Synonym',
        'MotivatedByGoal', 'NotCapableOf
    ', 'same capability', 'Causes',
        'at location', 'used for', '
    manner of'
    ]

    blocks = []
    current_block = {}

    with open(input_file, "r", encoding=
    "utf-8") as f_in:
        for line in f_in:
            line = line.strip()
            if not line:
                if current_block and "
    input" in current_block:
                    blocks.append(
    current_block)
                    current_block = {}
                continue
            if line.startswith("Input"):
                if current_block and "
    input" in current_block:
                    blocks.append(
    current_block)
                current_block = {"input"
    : line.split(":", 1)[1].strip()}
            elif line.startswith("
    Expected"):
```

```
                current_block["expected"] = line.split(":", 1)[1].strip()
            elif line.startswith("Predicted"):
                current_block["predicted_line"] = line
                blocks.append(current_block)
                current_block = {}

    if current_block and "input" in current_block:
        blocks.append(current_block)

    with open(output_file, "w", encoding="utf-8") as f_out:
        for block in blocks:
            input_text = block["input"]
            f_out.write(f"Input     : {input_text}\n")
            if "expected" in block:
                f_out.write(f"Expected  : {block['expected']}\n")
            relation = extract_relation(input_text, relation_list)
            context_vec = get_context_vector(input_text, glove, relation)
            predicted = find_best_candidate(context_vec, glove, relation, input_text)
            f_out.write(f"Predicted : {predicted}\n\n")
```

Listing 48: Replacing Predictions with GloVe Outputs

```
import os

glove_path = os.path.join(path, "glove.6B.300d.txt")
glove = load_glove_embeddings(glove_path)

input = "model_predictions.txt"
replace_predictions_with_glove(input, "glove_predictions2.txt", glove)
```

Listing 49: Final Execution: GloVe Path and Prediction

A.8 Prediction Evaluation

```
# prediction_accuracy.ipynb
# Automatically generated by Colab.
# Original file is located at:
# https://colab.research.google.com/drive/1l2GgceNsfAgt593eEbpsUd2a7x9S6xfd
```

Listing 50: Notebook Metadata

```
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity

# Load SBERT model
sbert_model = SentenceTransformer('all-MiniLM-L6-v2')

def compute_semantic_similarity(text1, text2):
    embeddings = sbert_model.encode([text1, text2])
    sim = cosine_similarity([embeddings[0]], [embeddings[1]])[0][0]
    return sim
```

Listing 51: Loading SBERT and Similarity Function

```
input_file = "glove_predictions2.txt"
output_file = "glove_evaluation_output.txt"

parsed_data = []
current_triplet = {}

with open(input_file, "r", encoding="utf-8") as f:
    for line in f:
        line = line.strip()
        if not line:
            continue
        if line.startswith("Input"):
            current_triplet['input'] = line.split(":", 1)[1].strip()
        elif line.startswith("Expected"):
            current_triplet['expected'] = line.split(":", 1)[1].strip()
        elif line.startswith("Predicted"):
            current_triplet['predicted'] = line.split(":", 1)[1].strip()
            if all(k in current_triplet for k in ('input', 'expected', 'predicted')):
                parsed_data.append((current_triplet['input'], current_triplet['expected'], current_triplet['predicted']))
                current_triplet = {}
```

Listing 52: Parsing GloVe Prediction Output

```
with open(output_file, "w", encoding="utf-8") as out_f:
    for input_text, expected, predicted in parsed_data:
        similarity = compute_semantic_similarity(expected, predicted)
        out_f.write(f"Input     : {input_text}\n")
        out_f.write(f"Expected  : {expected}\n")
        out_f.write(f"Predicted : {predicted}\n")
        out_f.write(f"Similarity: {similarity:.4f}\n")
        out_f.write("-" * 40 + "\n")
```

Listing 53: Writing SBERT Similarity Scores to File

```
SIMILARITY_THRESHOLD = 0.2
exact_matches_file = "accuracy.txt"
correct = 0
total = 0
lines_buffer = []
```

```

with open(output_file, "r", encoding="utf-8") as f, open(exact_matches_file, "w", encoding="utf-8") as exact_f:
    for line in f:
        lines_buffer.append(line)
        if line.startswith("Similarity"):
            try:
                sim = float(line.split(":", 1)[1].strip())
                if sim >= SIMILARITY_THRESHOLD:
                    correct += 1
                    for buffered_line in lines_buffer:
                        exact_f.write(buffered_line)
                    exact_f.write("\n")
                total += 1
            except ValueError:
                pass
        elif line.startswith("-" * 10):
            lines_buffer = []
accuracy = correct / total if total > 0 else 0
print(f" Similarity score >= 0.2: {accuracy:.4f} ({correct}/{total})")
```

Listing 54: Evaluation at Threshold = 0.2

```
SIMILARITY_THRESHOLD = 1.0
exact_matches_file = "exact_matches.txt"
correct = 0
total = 0
lines_buffer = []

with open(output_file, "r", encoding="utf-8") as f, open(exact_matches_file, "w", encoding="utf-8") as exact_f:
    for line in f:
        lines_buffer.append(line)
        if line.startswith("Similarity"):
            try:
                sim = float(line.split(":", 1)[1].strip())
                if sim >= SIMILARITY_THRESHOLD:
                    correct += 1
                    for buffered_line in lines_buffer:
                        exact_f.write(buffered_line)
                    exact_f.write("\n")
                total += 1
            except ValueError:
                pass
        elif line.startswith("-" * 10):
            lines_buffer = []
accuracy = correct / total if total > 0 else 0
print(f" Similarity score = 1.0: {accuracy:.4f} ({correct}/{total})")
```

Listing 55: Evaluation at Threshold = 1.0 (Exact Match)

```
SIMILARITY_THRESHOLD = 0.5
exact_matches_file = "similarity_half.txt"
correct = 0
total = 0
lines_buffer = []

with open(output_file, "r", encoding="utf-8") as f, open(exact_matches_file, "w", encoding="utf-8") as exact_f:
    for line in f:
        lines_buffer.append(line)
        if line.startswith("Similarity"):
            try:
                sim = float(line.split(":", 1)[1].strip())
                if sim >= SIMILARITY_THRESHOLD:
                    correct += 1
                    for buffered_line in lines_buffer:
                        exact_f.write(buffered_line)
                    exact_f.write("\n")
                total += 1
            except ValueError:
                pass
        elif line.startswith("-" * 10):
            lines_buffer = []
accuracy = correct / total if total > 0 else 0
print(f" Similarity score > 0.5 : {accuracy:.4f} ({correct}/{total})")
```

Listing 56: Evaluation at Threshold = 0.5