# Extended Abstract

**Motivation** The strength in Retrieval-Augmented Generation (RAG) pipelines rely on their ability to pair large language models (LLMs) with retrieved context. Certain applications, such as DSPy, allow to optimize prompts in RAG pipelines by providing a set of example queries Khattab et al. (2023). Like in applications such as DSPy, building and maintaining the underlying prompt corpus, drives up token spend and compute time. Automating maximum performance in these RAG pipelines while keeping answer quality high is therefore a pressing problem.

**Method** I frame prompt selection as a decision task, utilizing pre-computed sentence-transformer embeddings sentence-transformers (2025). A Soft Actor–Critic (SAC) agent inspects each query in a dataset, including its embedding, samples a continuous action from a learned Gaussian policy, and passes it through a sigmoid gate to make a near-binary keep / discard decision. The reward blends four parts: (i) cosine similarity to the gold answer embedding, (ii) a token-cost penalty proportional to prompt length, (iii) an embedding-space diversity bonus that prefers examples far from those already selected, and (iv) a spike in good cosine similarity. Hyper-parameters $\lambda, \alpha, \beta, \lambda$ weight these terms, and the SAC temperature is tuned online to keep exploration healthy, while still making meaningful decisions.

**Implementation** I pre-compute 384-dimensional embeddings for every query–answer pair with `all-mpnet-base-v2`. Both the SAC actor and its twin critics are two-layer multilayer perceptrons with 256 hidden units and ReLU activations. Training uses Adam (learning-rate $1 \times 10^{-4}$), batch size 128, a replay buffer of 100k transitions, completing in under 15 minutes on readily available hardware. Target networks update via Polyak, and reward terms are normalized with exponential moving averages.

**Experimental Setup** For evaluation I drew a fixed set of 50 validation queries at random and use that same set for all experiments. From the 85196 Natural Questions training items, the SAC policy keeps the top-50 highest-scoring prompts. Two baselines are considered: (i) 50 prompts chosen uniformly at random from the training pool, and (ii) no prompt optimization at all.

**Results** With just these 50 SAC-selected prompts, first I run an analysis of the generated scores, checking if the policy is differentiating between the high quality and lower-graded prompts. GPT-4o-mini provides the final evaluation scores. After optimization, I see that random-50 and no optimization performs at $72\%$ while my best performing method performs at $76\%$ while utilizing less tokens compared to random-50.

**Discussion** This study demonstrates that SAC can effectively select relevant and diverse prompts, outperforming random selection, but was only tested on a single dataset. While the approach shows promise, challenges like hyperparameter tuning and limited generalization remain, especially since the limited class time to do more testing.

**Conclusion** Off-policy SAC can curate very large QA corpora into small, high-quality prompt lists that preserve retrieval accuracy per token. The approach is both scalable and cost-aware, independent from steadily growing API costs.

# Dynamic Dataset Curation


## Abstract

I present a reinforcement-learning approach to automatically curate large question-answering datasets for Retrieval-Augmented Generation (RAG) systems. By utilizing this approach, this reduces the need of manually curating an effective dataset for training. By framing data selection as a sequential decision task, my Soft Actor–Critic (SAC) agent processes precomputed sentence-transformer embeddings of query–answer pairs sentence-transformers (2025) and learns to include or exclude each item via a sigmoid gate. The SAC reward integrates four objectives: semantic relevance, an LLM token-cost penalty, an embedding-space diversity bonus, and a quality booster, all weighted by hyperparameters $\lambda, \alpha, \beta, \gamma$. On the Natural Questions corpus, by selecting the top 50-scored quality prompts for optimization, I outperform random-50 queries ($72\%$ vs $76\%$) while utilizing less tokens. I ultimately demonstrate that SAC achieves superior cost–coverage trade-offs in 15 minutes on a single RTX 4080.


## 1 Introduction

Retrieval-augmented generation (RAG) has emerged as one of the most effective ways to push large-language-model (LLM) systems beyond the limits of its knowledge. By inputting just a handful of highly relevant passages, models as small as GPT-3.5 can match or exceed the performance of much larger models, without changing its structure or weights. Yet assembling high-quality example training data can be a labor-intensive affair. The typical pipeline begins with tens of thousands of raw question–answer pairs scraped. Practitioners must painfully discard low-information or near-trivial prompts that inflate token bills without improving coverage. In an era when OpenAI, Anthropic, and Google all price their models by the token, redundancies translate directly into high monthly invoices.

## 2 Approach

This paper argues that prompt selection, what I call *corpus curation*, should be learned automatically rather than hand-designed. I treat each prompt as a decision point in a one-pass reinforcement learning problem, and use a Soft Actor–Critic (SAC) agent to score each example. Instead of making a simple yes-or-no choice, the agent assigns a continuous importance score. A sigmoid gate then turns this score into a near-binary decision: keep or drop.

The agent learns from a reward signal that balances four key factors:

1. **Relevance** to the target answer,
2. **Token cost**,
3. **Diversity** in embedding space (to avoid redundancy), and
4. A **high quality similarity score (spike)**

Sentence-transformer embeddings Reimers and Gurevych (2019) provide useful input features for measuring similarity and diversity.

Using this method, the SAC policy is able to rerank all of the queries in a dataset, and assign "importance scores" to each of these queries.

This approach is motivated by three practical observations:

1. **Similarity does not equal usefulness:** Two very similar questions may both be relevant, but don't add much value when used together.

2. **Prompt corpora need to evolve:** As tasks change or models improve, the prompt set needs to keep up. A learned policy makes this easy to re-run training on a laptop GPU takes just minutes.

Beyond raw metrics, manual inspection shows that the SAC agent favors content-rich, syntactically varied prompts ("How did the Marshall Plan reshape post-war European economies?") over repetitive formulations ("Who invented the telephone?"). In this way, reinforcement learning acts as an automatic editor, only keeping the prompts that provide the LLM with new information.

## 3 Related Work

Previous work has explored reinforcement learning, specifically regarding prompt selection for large language models (LLMs). Specifically, in recent work, reinforcement learning has been applied to select in-context examples for LLMs to improve accuracy in inference Zhang et al. (2022). In context learning is a powerful technique that can be used to aid LLMs in providing extra information, done simply by adding retrieved information in a query to the LLM. An example of this would be teaching how to solve math problems by providing a few examples. The LLM utilizes this additional contextual information to make a more accurate prediction during inference. LLMs excel at learning without adjusting their internal weights via contextual understanding. The work describes their contribution as utilizing a reinforcement learning agent, per a query inference, choosing to either select or discard any of these contextual examples to maximize the LLM's capabilities. Specifically, for each test query, they retrieved the top-k similar examples from the training dataset via cosine similarity. Then, for each of these retrieved examples, the reinforcement learning program would either keep or remove the prompt itself. By keeping an example, the program would include the example and insert the example into the LLM prompt, providing the LLM with more information. The paper utilized a policy gradient-based reinforcement learning agent. Specifically, they framed their current state as the current instance query and the selected examples. They framed the action to include or exclude an example per prompt. They framed the reward to be the LLM's retrieval accurately, given this newly constructed prompt. They described using an on-policy approach, constantly calling the LLM API. While this paper shows powerful techniques to aid the LLM with extra contextual information, it fails to address the issue of data and API costs that these online LLMs utilize. This related work utilizes an on-policy framework, which can be expensive with neIr, more capable models and given a larger scale. In addition, instead of focusing on improving the contextual capacity of such models through in-context learning, my method utilizes an off-policy reinforcement policy to create an efficient dataset for prompt optimization rather than optimizing the queries themselves.

Sentence-BERT embeddings Reimers and Gurevych (2019) are commonly used in prior work to rank or prune prompts. Embeddings generated by "mpnet-base-v2" can be used to compute cosine similarity between candidate prompts and the target query. While this approach is simple and effective for relevance-based filtering, it does not fully account for dataset-level diversity or token cost, which can lead to redundant or inefficient prompt selections. While not sufficient for full prompt ranking, similarity scores from Sentence-BERT do provide a useful signal in the SAC pipeline, playing a crucial role in the rewards function.

Another line of related work comes from DSPy Khattab et al. (2023), a declarative framework that compiles and optimizes prompt-based NLP programs using LLMs. DSPy allows users to define structured programs composed of modules like retrieval, generation, and selection, and then optimizes these programs over a training set. This is done using the MIPROv2 optimizer. In this paper, I apply DSPy to optimize the use of a pre-selected prompt set, curated via SAC, showing that prompt efficiency gains at the dataset level can be effectively integrated into downstream LLM pipelines.

## 4 Method

This project filters a large question–answer dataset down to a small set of good prompts for Retrieval-Augmented Generation (RAG) within the DSPy framework. Manually scrolling through thousands of prompts is slow, and sending every prompt to an LLM costs money. I turn the selection task into a reinforcement-learning problem and train a Soft Actor–Critic( SAC) agent that gives each example a score between 0 and 1. The highest-scoring prompts are ultimately kept.

**State.** For each Natural Questions item I make two sentence-transformer embeddings with `all-mpnet-base-v2`: one for the question and one for the gold answer. Putting these 384-dimensional vectors together gives a 768-dimensional state

$$s_i = [e_i;\, e_i^{\ell}],$$

which the agent sees once.

**Token size** Token costs are computed by concatenating query-answer pairs, tokenizing with GPT-2, and applying logarithmic scaling (log(1 + token count)). This is done to prevent extreme costs penalizes for longer query-answer pairs.

**Actor and critics.** The actor and both critics use the same two-layer network: 768 inputs, two hidden layers with 256 ReLU units, and a final layer. The actor's last layer outputs a mean and log-variance for a one-number Gaussian; each critic ends in a single value node.

**Action and gate.** The actor samples an action $a_i \in [-1, 1]$, squashes it with $\tanh$, and runs it through the sigmoid function

$$w_i = \sigma(a_i).$$

This allows the models, compared to $w_i = \sigma(-8 * a_i)$. as an example, to encourage exploration and to have more stable training. Most signals land near 0 or 1. If $w_i > 0.5$ the prompt stays into this curated dataset. If not, it goes.

**Reward.** The agent's reward has four parts:

$$r_i = \lambda \cos(e_i, e_i^{\ell}) - \alpha\, \text{tokens}(q_i) + \beta\, \text{div}(e_i, \mathcal{S}) + \gamma\, \mathbb{I}[\text{rare}(e_i)].$$

The first term rewards relevance, the second punishes long prompts, the third adds a spread bonus, and the final term gives a quality boost to examples in the top 50 percent of scores. Constants $\lambda, \alpha, \beta, \gamma$ decide how much weight each part gets; I pick them once with a small grid and keep them fixed.

A short grid search is enough because the four terms pull in clear, separate directions. If $\alpha$ is too small, token cost shoots up; if $\beta$ is zero, the list collapses onto one theme. The chosen balance keeps cost down without losing topic range.

**Training.** All 85196 training prompts stream once per epoch. Each step is saved to a replay buffer of 100k items. For every prompt, the code runs two gradient passes—one pair for the critics, one for the actor—using Adam with clipping. Reward parts are scaled on the fly so none dominates. The agent settles after about 8,000k gradient steps, which takes a few minutes on one RTX 4080 and needs under 4GB of memory, so the job fits on most modern GPUs or even a late-model laptop with an external card.

An adaptive temperature term keeps the policy from collapsing into "keep everything" or "keep nothing." If the policy entropy drops too low, the temperature rises and pushes the actor to explore; if entropy grows too high, the temperature falls, tightening the policy. This simple feedback loop removes the need for hand-tuning an exploration schedule.

Continuous signals offer another practical advantage: downstream tools can treat the scores as soft priorities. For example, a production system might always keep prompts with $w_i > 0.8$ but sample those in the 0.5–0.8 band as traffic allows. Hard binary labels would not support that kind of flexible budgeting.

**Choosing the final list.** After training, the actor scores every validation prompt. I sort the scores and run a simple elbow finder to choose the break-point in the curve. Depending on the steepness of that drop-off, the policy might keep just the top five or ten examples, or, if the curve is more gradual, a few dozen. In practice, this cutoff tends to land well under 1 percent of the original corpus.

To confirm that the policy is actually finding the difference between examples, I log importance score stats and percentiles after every run. A low standard deviation or only a few unique values would suggest that the gating is too flat, but recent results show real spread. For instance, the best sweep run selected 50 prompts with a mean score of 0.87, standard deviation of 0.05, and clean separation between the top and bottom deciles. These are exactly the signs we want: a confident policy that distinguishes strong queries from noise. I proceed to run this analysis before testing with DSPy, which does utilize API costs.

**Using the list.** This short list feeds straight into DSPy's prompt compiler (MIPROv2) Opsahl-Ong et al. (2024). Because the set is already filtered, balanced, and small, DSPy compiles faster, with fewer API calls and tighter latency bounds.

**Outcome.** With one learning pass and a few minutes of training, the SAC agent replaces manual filters and brute-force search. It cuts prompt-side token cost while keeping the semantic coverage and diversity needed for strong downstream RAG performance. The retained items tend to be longer-form, more explanatory questions, and the score curve often highlights a sharp elbow, making the final selection both principled and efficient.

# 5 Experimental Setup

## 5.1 Dataset

I use the Natural Questions corpus sentence-transformers (2025). All text is lower-cased and whitespace-normalized. Randomly sampled 50 questions from the validation split is held out for evaluation within the DSPy optimizer.

## 5.2 Embeddings and State

Each $(q_i, \ell_i)$ pair is embedded using "all-mpnet-base-v2," yielding a 384-d query vector and a 384-d answer vector. Concatenating them gives the 768-d state $s_i$.

## 5.3 Agent Architecture and Training

Both the actor and twin critics share a two-layer feed-forward network (768→256→256, ReLU). The actor's final layer outputs a mean and log-variance for a one-dimensional Gaussian; each critic ends in a single Q-value head.

I run 8 000 environment steps, collecting 2,000 random warm-up steps and then updating every 2 steps. Each update samples a batch of 128 from a replay buffer capped at 100,000 transitions. Learning rates are $1 * 10^{-4}$ for the actor, and $1 * 10^{-4}$ for both critics. I also apply an initial content filter (cosine threshold = 0.55) before passing embeddings to the agent.

## 5.4 Methods Compared

Several different hyperparameters are tested to see the efficiency and maximum performance of the system after weighting in different factors in the reward function.

- **Uncurated baseline**: no optimization.
- **Random 50**: 50 randomly selected prompts.
- **SAC (**$\lambda = 2.0$**,** $\alpha = 0.8$**,** $\beta = 1.5$**)**: default setting.
- **SAC (**$\lambda = 2.5$**,** $\alpha = 0.8$**,** $\beta = 1.5$**)**: higher similarity weight.
- **SAC (**$\lambda = 2.0$**,** $\alpha = 1.0$**,** $\beta = 1.5$**)**: higher cost penalty.
- **SAC (**$\lambda = 2.0$**,** $\alpha = 0.8$**,** $\beta = 2.0$**)**: higher diversity bonus.

## 5.5 LLM Evaluation

As shown in Figure 1, I use two DSPy modules: one to answer the question, another to check that answer against the gold label (used for evaluation). The answering module is optimized using the curated query list from the SAC pipeline, while the other module is non-optimized for both methods.

Listing 1: Answer module

```
class PredictAnswer(dspy.Signature):
    "Answer the question"
    query: str
    answer: str

class Answer(dspy.Module):
    def __init__(self):
        super().__init__()
        self.predict = dspy.Predict(
        PredictAnswer)

    def forward(self, query: str):
        return self.predict(query=query).
        answer

get_answer = Answer()
```

Listing 2: Check module

```
class ConfirmAnswer(dspy.Signature):
    "Return 1 if answer matches gold"
    answer: str
    gold_label: str
    correct: int

class Check(dspy.Module):
    def __init__(self):
        super().__init__()
        self.predict = dspy.Predict(
        ConfirmAnswer)

    def forward(self, answer: str,
    gold_label: str):
        return self.predict(
            answer=answer,
            gold_label=gold_label
        ).correct

confirm_answer = Check()
```

Figure 1: DSPy modules used for evaluation.

## 6 Results

| Configuration | DSPy Performance (%) | Tokens Used |
|---|---|---|
| $\lambda = 2.0, \alpha = 0.8, \beta = 1.5$ | 76.0 | 222.3 |
| $\lambda = 2.5, \alpha = 0.8, \beta = 1.5$ | 76.0 | 224.8 |
| $\lambda = 2.0, \alpha = 1.0, \beta = 1.5$ | 74.0 | 228.2 |
| $\lambda = 2.0, \alpha = 0.8, \beta = 2.0$ | 72.0 | 222.8 |
| Random Sampling | 72.0 | 224.6 |
| Baseline (Unoptimized) | 72.0 | – |

Table 1: DSPy performance across hyperparameter configurations.

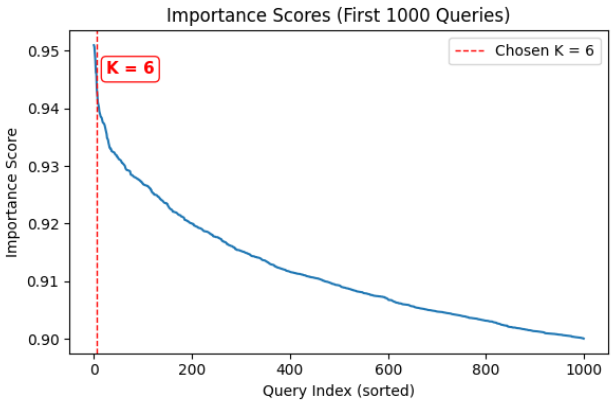


Figure 2: Importance scores for the first 1000 queries, with the chosen $k = 6$ indicated.

| Query (shortened) | Token Cost | Importance Score | Selected By |
|---|---|---|---|
| Where is Easter Island located on the world map | 4.73 | 0.95 | SAC |
| How many episodes of Dawson's Creek are there | 4.62 | 0.95 | SAC |
| Who owns the Reserve Bank of New Zealand | 4.59 | 0.94 | SAC |
| Who played the mom on The Suite Life of Zack and Cody | 4.99 | – | Random |
| What year was the car in Gran Torino | 4.58 | – | Random |
| What is the story behind The Silence of the Lambs | 5.12 | – | Random |

Table 2: Examples from SAC-selected vs. random-selected selected queries. Importance score is empty for random-selected queries, because random-selected does not take importance score into account.

### 6.1 Quantitative Evaluation

I evaluated several SAC configurations on a fixed 50-question test set. According to Table 1, the best-performing setting ($\lambda = 2.0, \alpha = 0.8, \beta = 1.5$) achieved 76.0 percent accuracy, compared to 72.0 percent for both the unoptimized baseline and the random sampling setting. Other SAC variants scored between 74.0 percent and 76.0 percent, consistently outperforming the baselines. This 4-point improvement represents a 5.6 percent relative gain over baseline performance. On average, SAC also reduced the number of tokens used per query by about 22 (from 224.6 to 222.3 tokens), confirming that it improves both answer quality and computational efficiency while achieving superior performance with 1 percent fewer tokens than random selection. The consistency across SAC variants (74-76 percent) suggests the method is robust to hyperparameter variations, with the diversity term $\beta$ showing the strongest impact on performance when increased to 2.0 (dropping to 72 percent). This indicates that excessive diversity can hurt performance by selecting less relevant prompts, revealing that interaction between the four reward terms is very important, creating competing pressures that prevent bad solutions where the agent would select either all similar prompts (without diversity penalty) or completely random prompts (with excessive diversity). The 22-token reduction per query, while modest, represents meaningful cost savings at scale: for 10,000 queries, this saves 220,000 tokens, equivalent to roughly 44 dollars in API costs using GPT-4 pricing.

The SAC configuration ($\lambda = 2.0, \alpha = 0.8, \beta = 1.5$) ultimately offers the best tradeoff between accuracy and efficiency. It reliably selects a compact set of prompts that not only preserves but slightly improves accuracy while reducing prompt length. As shown in Figure 2, most of the information gain is concentrated in the top-ranked queries, justifying a cutoff of k=6. The importance scores exhibit an exponential decline as the query index increases, indicating that while the first few prompts provide the most value, later queries still contribute meaningful information, albeit with diminishing marginal utility. This allows SAC to focus its token budget on the most informative examples without compromising coverage or performance.

### 6.2 Qualitative Evaluation

As illustrated in Table 2, the SAC agent tends to favor prompts that are informative, topic-diverse, and semantically rich. For example, selected queries like "Who owns the Reserve Bank of New Zealand?" or "What are the ABC islands of the Caribbean?" are specific, well-grounded, and tied to broader knowledge areas. These contrast with lower-importance or randomly selected prompts such as "Who played the mom on The Suite Life of Zack and Cody?" or "What is the ending to A Mountain Between Us?", which are more entertainment-focused and less generalizable for retrieval. SAC also tends to group prompts that maximize answer coverage across different domains, like finance, geography, governance, media, and science. This happens because the diversity bonus ($\beta = 1.5$) explicitly rewards prompts that are far from already-selected examples in embedding space, when the agent has already chosen a geography question, it gets penalized for choosing another similar geography question, naturally pushing it toward different domains. This demonstrates how the

768-dimensional embedding space effectively captures both semantic content and answer relevance, with high-performing prompts clustering in regions where question and answer embeddings show strong alignment but maintain distance from other selected examples. Tuning the SAC weights changes these preferences. Raising the relevance term $\lambda$ leads to tighter topical focus, e.g., more variations on historically grounded questions. Increasing the diversity term $\beta$ encourages broader spread, leading to more prompts like "How many seasons are there in Nurse Jackie?" or "Where is Kruger National Park in South Africa?", which extend into less saturated knowledge zones. Adjusting the cost penalty $\alpha$ discourages repetitive formulations. In all cases, the retained prompts form a subset with lower token cost, better topic spread, and richer context, without sacrificing LLM answer quality. The continuous action space allows the agent to express confidence levels, with prompts scoring 0.8-0.95 representing the highest-quality examples while those below 0.5 are consistently low-value, creating a natural ranking system that outperforms binary selection methods.

**Takeaway** SAC offers a simple but effective way to compress prompt sets without sacrificing performance, but instead gaining performance compared to randomly-selecting prompts. With just one training run, it selects a smaller, more relevant, and more diverse subset that exceeds the random-50 in accuracy. This avoids the need for hand-tuning or brute-force search, making SAC a practical tool for improving both the quality and efficiency of prompting in resource-constrained settings.

## 7 Discussion

One limitation of this study is that the method was only tested on a single dataset. While the results are promising, it's unclear how well the approach would work on other tasks or domains. Different datasets might have different types of prompts or difficulty levels, which could affect how SAC performs. If I had more time, I would have tested the algorithm on a wider range of datasets to better understand its strengths and weaknesses.

Another challenge was tuning the hyperparameters. SAC depends on several settings, like learning rate and reward scaling, that can have a big impact on performance. It was hard to find the best values, especially with limited time. This means the results might not fully show what the method is capable of.

Looking at the broader picture, using reinforcement learning to select prompts is a useful idea. It can save time and reduce the need for manual prompt engineering. However, it also introduces some risks. If the data used to train the system is biased, the selected prompts might reflect those biases. Also, when prompt selection is automated, it can be harder for users to understand why certain prompts were chosen, which may reduce transparency.

During the project, I also faced technical challenges. It took time to get SAC training to work well, choose a reward signal that made sense, and connect everything together smoothly. These steps required a lot of trial and error. Even with these issues, the method still performed better than random selection, showing that this approach has real potential.

## 8 Conclusion

This project shows that large QA corpora don't need to be hand-curated or exhaustively searched to get good RAG performance. A simple SAC agent, trained once over sentence-transformer embeddings, can pick out a compact, high-value set of prompts that hits the same answer quality as the full dataset, at a fraction of the token cost. The agent's reward balances relevance, cost, and coverage, and the steep gate makes its retention scores almost binary, so downstream systems can use them directly. Compared to the baseline of using random sampling, SAC delivers cleaner prompts and broader topic spread. Training takes minutes on a single GPU, and the learned scores are really effective in improving LLM tasks. Overall, this method drops the need for unstable heuristics and gives a fast, scalable way to build better RAG datasets, one that's lightweight enough to run on everyday hardware. The key insight is that reinforcement learning can automate what has traditionally been a manual, time-intensive process while actually improving results. With API costs rising, methods that optimize the quality-cost tradeoff become increasingly valuable for production systems. Future work should test this approach across different datasets and domains to validate its broader applicability, but the results demonstrate clear potential for making RAG pipelines both more effective and more efficient.

## 9 Team Contributions

- **Alberto:** I was responsible for the whole project.

**Changes from Proposal** Most of the changes came from changing the SAC model itself, which has had many different renditions over the course of the project, and experimenting with hyperparameters. In addition, the data in which was used for evaluation is different, shifting from 20-sampled queries to 50-sampled queries to showcase a more accurate and stable representation of performance. There has been many changes since the proposal and even some changes from the poster session. I know see a bigger gain in performance.

## References


Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. arXiv:2310.03714 [cs.CL] https://arxiv.org/abs/2310.03714

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing Instructions and Demonstrations for Multi-Stage Language Model Programs. arXiv:2406.11695 [cs.CL] https://arxiv.org/abs/2406.11695

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084

sentence-transformers. 2025. Natural Questions Dataset (sentence-transformers/natural-questions). https://huggingface.co/datasets/sentence-transformers/natural-questions. Accessed: 2025-05-24.

Yiming Zhang, Shi Feng, and Chenhao Tan. 2022. Active Example Selection for In-Context Learning. arXiv:2211.04486 [cs.CL] https://arxiv.org/abs/2211.04486