Scalable Query Understanding for E-commerce: An Ensemble Architecture with Graph-based Optimization

Giuseppe Di Fabbrizio[†], Evgeny Stepanov[†], Ludovico Frizziero[†] and Filippo Tessaro[†]

Abstract

Query understanding is a critical component in e-commerce platforms, facilitating accurate interpretation of user intent and efficient retrieval of relevant products. This study investigates scalable query understanding techniques applied to a real-world use case in the e-commerce grocery domain. We propose a novel architecture that integrates deep learning models with traditional machine learning approaches to capture query nuances and deliver robust performance across diverse query types and categories. Experimental evaluations conducted on real-life datasets demonstrate the efficacy of our proposed solution in terms of both accuracy and scalability. The implementation of an optimized graph-based architecture utilizing the Ray framework enables efficient processing of high-volume traffic. Our ensemble approach achieves an absolute 2% improvement in accuracy over the best individual model. The findings underscore the advantages of combining diverse models in addressing the complexities of e-commerce query understanding.

Keywords

Query classification, Query understanding, Distributed and scalable machine learning.

1. Introduction

Accurately understanding and classifying user queries is crucial for providing a seamless shopping experience by boosting the product search results relevance in e-commerce [1]. Query understanding enables e-commerce platforms to interpret users' intents, retrieve relevant products, and personalize the user's journey through the shopping experience. However, the task of query understanding in e-commerce presents several challenges due to the diverse nature of queries, the large-scale product catalogs, and the need for efficient processing of high-volume traffic with noisy behavioral signals [2, 3].

Query understanding in e-commerce involves multiple sub-tasks, such as query classification, entity recognition, and intent detection. Query classification aims to categorize user queries into predefined product categories, facilitating improved product retrieval and ranking [4, 5]. Entity recognition identifies key information within the query, such as brand names, product attributes, and numerical values, which can be used to refine the search results [6]. Intent detection focuses on understanding the user's underlying goal, such as product discovery, comparison, or purchase [7].

One of the primary challenges in query understanding is the inherent ambiguity and diversity of user queries. E-commerce queries are often short, lacking context, and can have multiple interpretations [8]. Moreover, the large-scale product catalogs in e-commerce platforms, spanning thousands of categories and millions of products, pose a significant challenge in accurately mapping queries to relevant categories and products.

Various approaches have been proposed to address these challenges, leveraging traditional machine learning techniques and deep learning models. Rule-based systems and keyword matching have been widely used for query classification and entity recognition [9]. However, these approaches often struggle with the variability and complexity of natural language queries. Different query intents require different algorithms to yield optimum results [10]. Queries can be classified into *navigational* (e.g., product category, brand, title) and *informational* (e.g., product-related questions). While navigational queries require exact matching to catalog products, informational queries necessitate applying more complex understanding techniques.

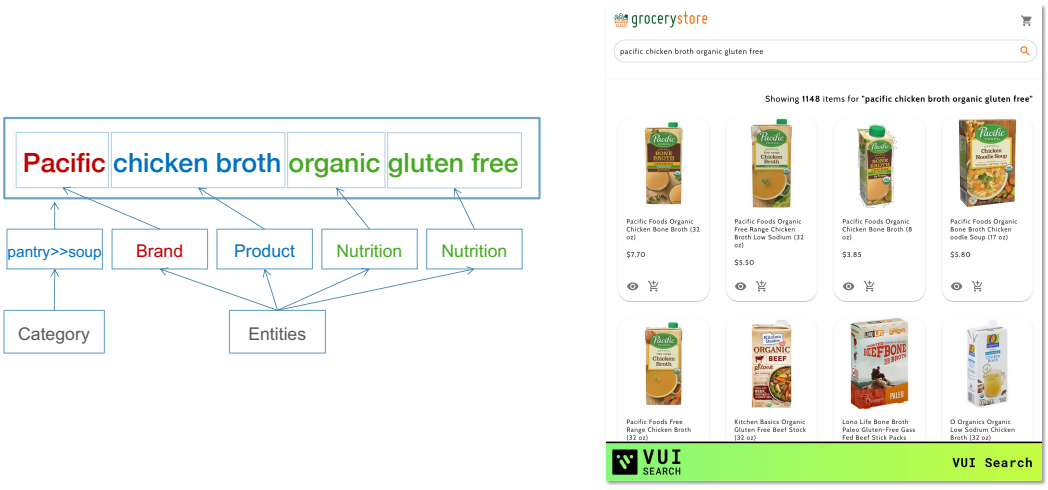
Another critical aspect of query understanding in e-commerce is efficiently processing high-volume traffic. E-commerce platforms receive millions of queries daily, requiring scalable and real-time query understanding systems. Distributed computing frameworks, such as Apache Spark and Ray, have been employed to parallelize query processing and handle the massive scale of e-commerce data [11, 12].

In this paper, we propose an ensemble approach for query understanding in e-commerce, combining deep learning models and traditional techniques. Our approach leverages the strengths of both deep learning, such as DistilBERT [13], and traditional models, including logistic regression and rule-based systems. By integrating these diverse models, we aim to capture the

CLiC-it 2024: Tenth Italian Conference on Computational Linguistics, Dec 04 — 06, 2024, Pisa, Italy

[†] Work done when at VUI, Inc.

difabbrizio@gmail.com (G. Di Fabbrizio);
stepanov.evgeny.a@gmail.com (E. Stepanov);
ludovico.frizziero@gmail.com (L. Frizziero);
filippotessaro96@gmail.com (F. Tessaro)
https://difabbrizio.com/ (G. Di Fabbrizio)



(a) Query understanding parsing

(b) Search results

Figure 1: Query understanding parsing example with search results leveraging the query understanding signals

nuances of user queries and provide robust performance across various query types and categories.

We introduce an optimized graph-based architecture based on the Ray framework [12], enabling efficient processing of high-volume traffic and ensuring scalability.

2. Query understanding ensemble architecture

In this paper, we focus on navigational queries and classify them into product taxonomy categories while applying named entity recognition (NER) to capture relevant product attributes, such as *Brand*, *Nutrition*, *Flavor*, and numeric attributes like *quantities* and *measurements*. Figure 1 shows a typical example of a navigational search query in an e-commerce grocery domain where the query *"Pacific chicken broth organic gluten free"* is parsed into its attributes and categorized into its taxonomy label.

Classifying user queries into product taxonomy categories is a typical document classification problem that is complex and actively researched. The problem is complicated by the nature of available data, which can be either product descriptions with user-provided categories or user queries associated with catalog categories from user click-stream data. Products in the catalog are described in terms of attributes with associated values, and a subset of this mapping constitutes a set of entities that should be identified to build a search query and provide better search results.

Due to the rate of change in e-commerce, the classical approach of query annotation and model training is prohibitive. Consequently, the query understanding problem is cast as a document classification problem for matching user queries to the product taxonomy tree (categories) and a sequence labeling problem for entities of interest. For each problem, we propose using an ensemble approach with multiple models having different label sets and relations. Specifically, we predict two levels of the product taxonomy tree (L1 and L2) and extract the corresponding entities mentioned in the queries. Each level is predicted by an ensemble of models composed of business rules and machine learning models. Similarly, different machine learning and rule-based models are used to extract entities of interest.

2.1. Query understanding pipeline and ensemble components

The query understanding pipeline's classification and entity extraction components are trained and tested on pre-processed user queries. Common text pre-processing steps are applied, including spaCy's tokenization, lowercasing, and number normalization [14].

The classification ensemble consists of business rules, implemented as a lookup table, and two machine learning models: logistic regression and DistilBERT. DistilBERT is a compressed version of BERT [15] that retains 97% of the original model's performance while being 40% smaller and 60% faster at inference time. The key idea is to leverage knowledge distillation during the pre-training phase to learn a compact model that can be fine-tuned for downstream tasks. Integrating DistilBERT into a query understanding pipeline, alongside business rules and lo-

gistic regression, enhances the system's accuracy and robustness.

The entity extraction ensemble comprises: (1) a conditional random fields model; (2) a catalog-based lookup table to extract *Brand*, *Flavor*, and *Nutrition*; and (3) a rule-based Duckling library[1] to extract numerical entities such as *Price* and *Quantity*.

2.2. Classification decision fusion

In our ensemble learning scenario, the models are trained on different data and have different, potentially overlapping label spaces, unlike typical ensemble learning, where the same data is used to train all models. Due to the label space differences, decision fusion is performed on the predictor-by-label prediction matrix of confidence scores rather than using a simple majority voting strategy. Rule-triggered hypotheses are assigned to a confidence score of 1.0 taking priority on model-based predictions.

The decision fusion process takes a matrix of confidence scores as input and outputs a vector of aggregated confidence scores. The label space difference is addressed by applying a max operation on the column of prediction scores per label, ignoring the values with respect to the label space membership. Taking the maximum score per prediction approximates the product rule [16]. The final label is decided as the $argmax$ of this confidence score vector. Unlike voting-based decision fusion, such an approach allows aggregation of decisions from rules and any number of predictors.

2.3. Entity span consolidation

Span consolidation aggregates entity extraction hypotheses from one or several entity extractors into a shallow parse containing only non-overlapping spans. By default, this process is performed for spans from the same model, but it can also be enabled for an ensemble of extractors.

Inspired by [17], the span consolidation is performed in three steps: (1) *Identity consolidation*: Resolves identical spans by keeping the span with higher confidence, or randomly if confidences are equal; (2) *Containment consolidation*: Resolves spans contained within each other by keeping the longer span, i.e., the one that contains the other; (3) *Overlap consolidation*: Resolves overlapping spans by keeping the longer span, or alternatively merging them and assigning the label of the longest span. *Priority consolidation* can be used to give higher weights to predictions from extractors with higher confidence.

The decision fusion and span consolidation are generally applied as the final step of the query understanding pipeline to yield hypotheses containing only a non-overlapping set of entities and a single classification prediction per level, as described in Section 4.

[1] https://github.com/facebook/duckling

3. Models and ensemble evaluation

The engine's configuration represents the ensemble as a sequence of operations, called nodes, organized into a graph. The edges of this graph represent the interdependencies between nodes. The engine organizes and dispatches computations to maximize parallelism. Machine learning models for query classification are trained on product catalog data and tested on user queries, ensuring equal representation of head, torso, and tail queries in terms of frequency. Table 1 shows the sizes of the training and testing data, and the output categories. We predict two levels of product taxonomy: L1 with 17 categories and L2 with 169 categories. However, not all L1 categories have L2 labels, making the L2 sets subsets of the L1 data. The NER test set is a subset of the manually annotated test data for non-numerical entities.

The performance evaluation of the component models and the ensemble utilizes precision, recall, and F1-score metrics. For multi-class classification tasks, we report accuracy along with macro-averaged precision, recall, and F1-score to account for dataset imbalance. Entity extraction performance is assessed using micro-averaged metrics and token-level accuracy, adhering to CoNLL-style evaluation protocols.

To quantify the efficacy of the model ensemble, we conducted a comparative analysis against logistic regression and DistilBERT for level one predictions, with results presented in Table 2. DistilBERT demonstrates superior performance compared to logistic regression across all metrics. The ensemble model, however, consistently outperforms both individual models.

Consequently, the query understanding system adopts the ensemble approach in lieu of individual models. Rule-based components are excluded from this evaluation due to their limited data coverage and restricted label subsets.

Level two models show similar performance patterns to level one, though with lower performance due to the larger label space and fewer training documents per label. Entity ensemble performance aligns with other ensembles, favoring precision.

While the ensemble approach demonstrates improved performance, it faces challenges with certain query types. Extremely short queries (e.g., "chips" can refer to potato, tortilla, or chocolate) can be ambiguous without context. Highly ambiguous queries (e.g., "greens") may span multiple categories within the grocery domain. Novel products or brands not present in the training data pose difficulties. Complex, multi-intent queries (e.g., "organic gluten-free pasta sauce and whole grain spaghetti") can lead to misclassifications or incomplete entity extraction.

Future work could explore incorporating user session data or personalization techniques to provide additional

Table 1
Dataset sizes used to train and test components of the ensemble

	Training	Testing	Labels
Level 1	230,463	5,445	17
Level 2	212,087	4,486	169
NER	17,862	544	3
Brands Lookup	9,924	–	1

Table 2
Models and ensemble performance

Model	precision	recall	f1-score	accuracy
L1 DistilBERT	0.77	0.77	0.77	0.81
L1 Logistic Regression	0.76	0.70	0.73	0.75
L1 Model Ensemble	0.79	0.79	0.79	0.82
L2 Model Ensemble	0.68	0.67	0.66	0.70
Entity Ensemble	0.83	0.59	0.69	0.74

context for ambiguous queries and improve handling of out-of-vocabulary terms and multi-intent queries.

4. Graph-based architecture for scalable processing

Query understanding systems in e-commerce search engines must generate real-time responses within strict service level agreements (SLA). They execute complex logic involving different models interacting both in series and parallel.

Our engine is constructed as a sequence of operations (nodes) arranged in a graph showing their interdependencies (edges). Like neural networks, the graph-based engine organizes and dispatches each computation to maximize parallelism.

Parallelization occurs at multiple levels, including inter-operation parallelism and entire graph replicas, depending on deployment requirements. Each operation within the graph is a complex model component, requiring specific optimization strategies, such as data vectorization and memory sharing, to optimize the overall graph structure.

We represent the graph using the notation `node: [arg1, arg2, ..., argN]`, where `node` requires incoming edges from `arg1` through `argN`. The full configuration of the graph can be seen in Appendix A

The engine processes the notation by following these steps: First, it optimizes the graph by joining (inlining) nodes based on certain criteria, which increases parallel operations as much as possible. Next, it decides how many replicas of the graph to run on a single physical server. Each node is then mapped to a separate system process using the *Actor* model [18] for inter-process communication, with message passing between processes handled using Ray [12].

Each node is initialized by loading the models into memory, leveraging shared memory and copy-on-write primitives provided by the server's operating system. Each node is loaded only once, and subsequent processes assigned the same node reference the original memory. Since the models are used for inference, not training, there are no write operations, reducing memory footprint and improving loading times. Finally, the batching service handles the backpressure control system and the REST API for listening to incoming requests.

At startup, the engine performs several optimizations on the graph topology. The simplest is graph *culling*, removing nodes that do not interact with others. Each node's expected computational burden can be specified. Simple nodes (e.g., string regex preprocessors) are less resource-intensive than full neural network nodes. The engine modifies the graph by combining nodes or *inlining* to facilitate parallel operations and minimize costly inter-process communications. This results in lighter nodes being replicated multiple times and fused into heavier nodes, each mapped to a single system process.

After inlining, the engine performs graph linearization, converting the graph into a linear sequence, where each node depends only on preceding nodes, not subsequent ones. The engine dispatches nodes in order, synchronizing results only when necessary. This strategy minimizes pauses and maximizes parallelism. Nodes with a higher computational burden are prioritized, reducing the need for the backpressure control system, leveraging the fact

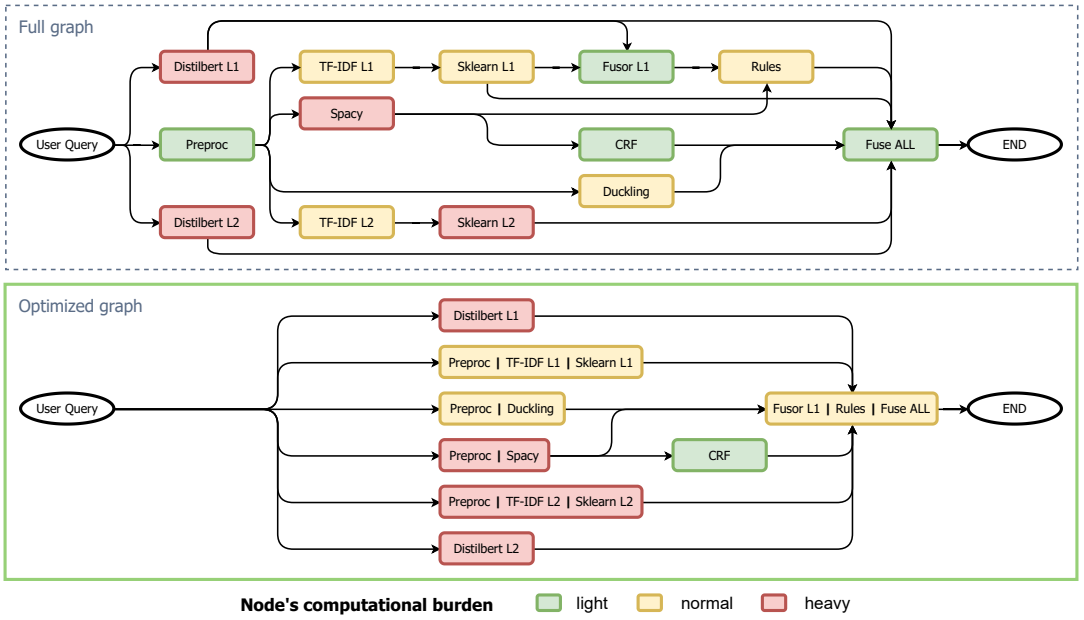


Figure 2: Visualization of the ensemble as a computational graph. (Top) The graph as defined in Appendix A. (Bottom) The graph after optimization by the engine.

that CPU and data transmission tasks are handled by separate CPU circuitry.

Query understanding systems receive hundreds of individual requests per second. Processing a single request is expensive due to inter-node communications. Batching multiple requests reduces overhead and enables vectorization, leveraging hardware primitives for efficient processing. The batching algorithm uses two thresholds: *batch size* and *waiting time* for further samples. This balances server resource utilization and processing time.

Lastly, the engine addresses CPU oversubscription [19], which occurs when parallel execution threads exceed available CPU cores, leading to overhead from context switching. The backpressure control system ensures no more than N nodes run in parallel, enhancing performance by reducing oversubscription. The number N depends on available CPU resources and the code executed within each node. A simple formula for determining N is:

$$N = \left\lfloor \frac{M_{cpu}}{\max_{i \in nodes}(threads_i)} \right\rfloor + 1 \qquad (1)$$

where $threads_i$ is the number of threads or processes that an individual node can utilize independently, and M_{cpu} denotes the available CPU cores on the server.

5. Performance analysis at scale

Multiple tests were conducted using different AWS[2] EC2 instances on the engine described in Section 4 and the ensemble configuration as in Appendix A. The optimal balance between cost, latency, and throughput was achieved with the `m6i.2xlarge` instance, which features 8-Cores Intel Xeon vCPU @ 3.5GHz, for which we report the results.

The test's target SLA stipulated that response times for 99% of requests should remain below 100ms.

All tests initiate a single instance of the engine with a graph replication factor of one[3]. Another server, which hosts the client simulator implemented using a Python package called Locust, is instantiated. Both servers share the same AWS network. The simulator issues multiple queries to the engine's server, each randomly sampled from a dataset of actual queries over a sustained duration of 30 seconds. The rate of each request follows an exponential distribution with a rate of T requests per second, mimicking a Poisson process, a common model for traffic patterns.

Table 3 reports the execution times of each node, along

[2] https://aws.amazon.com/

[3] Replication factors greater than one were also tested, but they caused immediate CPU oversubscription problems, as anticipated. The SLA targets were unattainable without resorting to costly GPUs.

Table 3
Quantiles for $T = 30tps$, times are in milliseconds. Nodes are sorted from fastest to slowest.

name	50%	95%	99%
preprocessor	0.05	0.06	0.07
fusor l1	0.38	0.53	0.58
fuse all	0.58	0.95	1.11
rules	0.87	1.29	1.56
crf	0.94	1.35	1.96
tfidf l2	1.36	2.39	5.22
tfidf l1	1.41	2.29	4.75
sklearn l1	1.62	2.68	4.86
duckling	2.80	11.95	35.71
spacy	12.87	24.70	33.68
distilbert l1	14.77	27.27	39.20
distilbert l2	14.90	27.75	37.58
sklearn l2	17.40	29.53	39.93
main loop	53.23	142.63	206.22
rest api	58.57	154.50	219.90

Batching disabled

name	50%	95%	99%
preprocessor	0.05	0.06	0.07
fusor l1	0.40	0.66	0.83
fuse all	0.59	1.16	1.65
rules	0.84	1.33	1.70
crf	0.96	1.62	2.01
tfidf l2	1.39	2.03	4.88
tfidf l1	1.43	2.05	3.88
sklearn l1	1.64	2.53	5.81
duckling	3.33	18.59	30.84
spacy	12.33	18.68	25.42
sklearn l2	15.71	18.99	22.87
distilbert l2	15.44	27.49	36.29
distilbert l1	15.60	27.34	37.23
main loop	30.51	41.18	51.74
rest api	55.85	84.35	**92.82**

Batching enabled

with the main engine loop responsible for scheduling them and the outer REST API handling incoming requests and facilitating the connection between the engine and the outside world. The runtime of each individual node must be strictly shorter than the main engine loop, representing the actual time taken for parallel graph execution. Node runtimes do not consider inter-process communication, which is accounted for in the main loop. On the other hand, the Rest API contributes to the main loop by including the time required to handle the HTTP connection with the requesting client. The outer Rest API time must stay below 100ms @ 99% percentile to comply with the target SLA.

When batching is disabled, at the given rate T, new requests arrive while the server is still processing previous ones. These requests are immediately dispatched, leading to CPU oversubscription, which slows down all requests. This effect tends to cascade, as the increased processing time makes it more likely that other requests will arrive, further slowing the system.

When batching is enabled, the engine pauses to accumulate requests into a batch until thresholds of 5 samples or $50ms$ are met. Given each request arrives every $1/T \approx 30ms$, the average batch size is around 1.5 samples. Therefore, vectorization alone cannot explain the server's ability to meet the target SLA. The process unfolds as follows: (1) the first batch is dispatched for processing, (2) for the next $50ms$, new requests are queued into a new batch while (3) the engine likely completes the first batch within $51.7ms$ (with 99% probability), (4) the second batch is then dispatched, utilizing just released resources. Thus, batching acts as backpressure control on cheaper hardware without a GPU and at low rates of T. In production, multiple instances would handle fluctuating traffic, making batching efficient for scaling while meeting the SLA. The optimal batching period should match the main loop time @ 99%, which is around $50ms$ in this case.

From a single request's perspective, with $T = 30tps$, batches are dispatched precisely every $50ms$, meaning requests encounter a uniform distribution over this interval with an average wait of $25ms$ in the batch queue. The entire batch is then processed, typically taking X time to complete before the response is extracted and forwarded through the HTTP channel, taking an additional Y. Empirically, X represents the main loop runtime, averaging around $30ms$ @ 50%. The Rest API, implemented using FastAPI[4], has been benchmarked to yield a duration of $Y \approx 2 - 5ms$, giving us

$$\text{REST API @50\%} = 25\text{ms} + 30\text{ms} + 2\text{ms} \approx 56.25\text{ms}$$

For REST API @ 99%, the wait time is always $25ms$ on average, but X and Y change accordingly, giving approx $90 - 95ms$.

6. Conclusion and future work

This paper proposed a novel ensemble approach for query understanding in e-commerce, combining deep learning models like DistilBERT with traditional techniques like

[4] https://fastapi.tiangolo.com/

logistic regression and rule-based systems. The ensemble architecture aimed to capture the nuances of user queries and provide robust performance across query types and categories. Data augmentation techniques were employed to improve the DistilBERT model's handling of brands, misspellings, and short queries. An optimized graph-based architecture using the Ray framework enabled efficient, scalable processing of high-volume traffic.

While the ensemble performed well, there are limitations to address in future work. The system focused only on navigational queries for product categorization and entity extraction. Extending it to handle informational and other query types could further improve relevance. Exploring more advanced data augmentation, model compression, and hardware acceleration techniques could enhance accuracy and efficiency.

The query understanding ensemble demonstrated the value of combining diverse models and leveraging distributed computing frameworks for scalability in e-commerce search engines. E-commerce platforms can benefit from adopting similar, ensemble-based approaches customized to their query traffic and product data. The architecture enables efficient real-time query processing while meeting strict latency requirements, critical for delivering a seamless shopping experience.

7. Appendix

A. Graph configuration

In our query understanding system, the relationships between various models and preprocessing components are organized within a graph-based architecture. This architecture plays a crucial role in managing the interdependencies between different models, ensuring efficient computation and scalability.

The graph representation is designed to handle the integration of multiple machine learning and rule-based models while facilitating optimized parallel processing. Each key in the graph corresponds to a node, which indicates a component or model, and the associated value is a list of other nodes that provide input to it. This differs from traditional adjacency lists, where the focus is on child nodes. Instead, in our graph, the value lists contain ancestor nodes, indicating which components feed information into the current node.

A key aspect of this architecture is that certain elements, such as `user_query`, are considered implicit nodes representing external inputs to the system. These external inputs play a foundational role in initiating the data flow throughout the graph. The architecture is designed to handle multiple outputs, listed within the `outputs` key. This is not a graph node but serves as an indicator to the engine of what to select as the final result. The output key is also vital for the process of graph topology optimization and linearization described in Section 4. This representation not only makes it easier to track data flow but also helps optimize the query understanding ensemble for real-time processing in e-commerce environments.

Figure 3: Graph Representation of Query Understanding Ensemble

```
execution_graph:
  preprocessor: [ user_query ]
  distilbert_l1: [ user_query ]
  distilbert_l2: [ user_query ]
  tfidf_l1: [ preprocessor ]
  tfidf_l2: [ preprocessor ]
  vui_duckling: [ preprocessor ]
  spacy: [ preprocessor ]
  crf: [ spacy ]
  sklearn_l1: [ tfidf_l1 ]
  sklearn_l2: [ tfidf_l2 ]
  fusor_l1: [ distilbert_l1, sklearn_l1 ]
  rules: [ spacy, fusor_l1 ]
  fuse_all: [
    rules, crf, distilbert_l1, sklearn_l1,
    distilbert_l2, sklearn_l2, vui_duckling
  ]
outputs: [ user_query, preprocessor, parse ]
```

Figure 3 illustrates the graph structure that defines the Query Understanding Ensemble. The nodes represent components that work together to process user queries and extract meaningful insights. The graph starts with preprocessing steps that normalize and clean the user input. Subsequently, components such as `DistilBERT` and `TF-IDF` are leveraged to extract semantic features and contextual information. Additional models like the `CRF` (Conditional Random Fields) and `vui_duckling` focus on identifying specific entities such as brands, quantities, and attributes.

The outputs from these models are fused together through specific nodes such as `fusor_l1` and `fuse_all`, which combine signals from the intermediate models based on confidence scores and rule-based decisions. The final outputs represent the processed user query, refined and enriched through multiple layers of analysis, ready for downstream tasks such as categorization and search relevance adjustments.

This architecture's flexibility and efficiency enable it to handle the complexities of e-commerce queries in real time while supporting high-volume traffic and diverse query types. It also lays the groundwork for the performance optimizations and parallel processing strategies outlined in Section 4.

References

[1] H. Deng, Y. Zhang (Eds.), Query Understanding for Search Engines, 1st ed., Springer, 2020. doi:10.1007/978-3-030-58334-7.

[2] S. Jiang, Y. Hu, C. Kang, T. Daly, D. Yin, Y. Chang, C. Zhai, Learning query and document relevance from a web-scale click graph, in: Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 185–194. doi:10.1145/2911451.2911531.

[3] P. Nigam, Y. Song, V. Mohan, V. Lakshman, W. A. Ding, A. Shingavi, C. H. Teo, H. Gu, B. Yin, Semantic product search, in: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 2876–2885. doi:10.1145/3292500.3330759.

[4] Y.-C. Lin, A. Datta, G. Di Fabbrizio, E-commerce Product Query Classification Using Implicit User's Feedback from Clicks, in: 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 1955–1959. doi:10.1109/BigData.2018.8622008.

[5] G. Di Fabbrizio, E. Stepanov, F. Tessaro, Extreme Multi-label Query Classification for E-commerce, in: eCom'24: ACM SIGIR Workshop on eCommerce, July 18, 2024, USA, 2024.

[6] J.-W. Ha, H. Pyo, J. Kim, Large-scale item categorization in e-commerce using multiple recurrent neural networks, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 107–115. doi:10.1145/2939672.2939678.

[7] Y. Qiu, C. Zhao, H. Zhang, J. Zhuo, T. Li, X. Zhang, S. Wang, S. Xu, B. Long, W.-Y. Yang, Pre-training Tasks for User Intent Detection and Embedding Retrieval in E-commerce Search, in: Proceedings of the 31st ACM International Conference on Information & Knowledge Management, CIKM '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 4424–4428. doi:10.1145/3511808.3557670.

[8] D. Shen, Y. Li, X. Li, D. Zhou, Product query classification, in: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 741–750. URL: https://doi.org/10.1145/1645953.1646047. doi:10.1145/1645953.1646047.

[9] B. Ramesh, Bhange, X. Cheng, M. Bowden, P. Goyal, T. Packer, F. Javed, Named Entity Recognition for E-Commerce Search Queries, in: 2018 IEEE International Conference on Big Data (Big Data), 2020. URL: https://api.semanticscholar.org/CorpusID:219530417.

[10] M. Tsagkias, T. H. King, S. Kallumadi, V. Murdock, M. de Rijke, Challenges and research opportunities in ecommerce search and recommendations, SIGIR Forum (2020).

[11] E. Shaikh, I. Mohiuddin, Y. Alufaisan, I. Nahvi, Apache spark: A big data processing engine, 2019 2nd IEEE Middle East and North Africa COMMunications Conference (MENACOMM) (2019) 1–6. URL: https://api.semanticscholar.org/CorpusID:211120979.

[12] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, I. Stoica, Ray: a distributed framework for emerging ai applications, in: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, USENIX Association, USA, 2018, p. 561–577.

[13] V. Sanh, L. Debut, J. Chaumond, T. Wolf, DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter, ArXiv abs/1910.01108 (2019). URL: https://api.semanticscholar.org/CorpusID:203626972.

[14] M. Honnibal, I. Montani, S. Van Landeghem, A. Boyd, spaCy: Industrial-strength Natural Language Processing in Python (2020).

[15] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, in: J. Burstein, C. Doran, T. Solorio (Eds.), Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Association for Computational Linguistics, Minneapolis, Minnesota, 2019, pp. 4171–4186. doi:10.18653/v1/N19-1423.

[16] J. Kittler, M. Hatef, R. Duin, J. Matas, On combining classifiers, Pattern Analysis and Machine Intelligence, IEEE Transactions on 20 (2002) 226–239.

[17] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, S. Vaithyanathan, An algebraic approach to rule-based information extraction, in: 2008 IEEE 24th International Conference on Data Engineering, IEEE, 2008, pp. 933–942.

[18] C. Hewitt, Actor model of computation: Scalable robust information systems, 2015. arXiv:1008.1459.

[19] C. Iancu, S. Hofmeyr, F. Blagojević, Y. Zheng, Oversubscription on multicore processors, in: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010, pp. 1–11. doi:10.1109/IPDPS.2010.5470434.