140509_20.md

README

20. Knowledge Graph Enhanced Q&A System

Summary: Create a question-answering system that combines knowledge graphs with generative AI to provide accurate, structured responses with reasoning chains.

Problem Statement: Traditional Q&A systems often lack structured reasoning and relationship understanding. Your task is to build a system that combines knowledge graphs with generative AI to answer complex questions requiring multi-hop reasoning. The system should construct and query knowledge graphs, generate explanations for answers, and provide confidence scores based on knowledge graph completeness.

Steps:

â€¢ Design knowledge graph construction from unstructured text using NER and relation extraction

â€¢ Implement graph-based query processing for multi-hop reasoning

â€¢ Create integration between graph queries and generative AI responses

â€¢ Build explanation generation showing reasoning paths through the knowledge graph

â€¢ Develop confidence scoring based on graph connectivity and source reliability

â€¢ Include graph visualization and interactive exploration capabilities

Suggested Data Requirements:

â€¢ Structured and unstructured text data for knowledge extraction

â€¢ Curated question-answer pairs requiring multi-hop reasoning

â€¢ Entity and relationship ontologies for domain-specific knowledge

â€¢ Source credibility and reliability metadata

Themes: GenAI & its techniques, Knowledge Graph, Graph RAG

The steps and data requirements outlined above are intended solely as reference points to assist you in conceptualising your solution.

PRD (Product Requirements Document)

Product Vision and Goals

The Knowledge Graph Enhanced Q&A System aims to revolutionize information retrieval by integrating structured knowledge graphs (KGs) with generative AI, enabling precise answers to complex, multi-hop queries. Goals include improving answer accuracy by 30% over traditional systems, providing transparent reasoning to build user trust, and supporting domain adaptability for sectors like healthcare, finance, and research.

Target Audience and Stakeholders

- Primary Users: Researchers, analysts, students, and professionals in knowledge-intensive fields.
- Stakeholders: Data scientists for KG maintenance, end-users for querying, administrators for system oversight.
- Personas: E.g., a biomedical researcher querying drug interactions needing multi-hop paths (drug -> protein -> disease).

Key Features and Functionality

- Automated KG ingestion and construction from diverse sources.
- Natural language query parsing to graph traversals.

- Generative AI for response synthesis with embedded reasoning.
- Confidence scoring and explanations for accountability.
- Interactive visualizations for KG exploration.
- API endpoints for integration with external apps.

Business Requirements

- Support for 100+ concurrent users with low latency.
- Compliance with data privacy standards (e.g., GDPR for entity handling).
- Monetization: Open-source core with premium features like custom ontologies.

Success Metrics

- User satisfaction: NPS >80.
- Accuracy: F1-score >0.85 on multi-hop QA benchmarks like HotpotQA.
- Adoption: 50% reduction in manual research time.

Assumptions, Risks, and Dependencies

- Assumptions: Access to open LLMs (e.g., Llama) and graph DBs (e.g., Neo4j Community).
- Risks: Incomplete KG leading to low confidence; mitigate with fallback to pure generative AI.
- Dependencies: Public datasets like WikiData for initial KG seeding.

Out of Scope

- Real-time KG updates from live streams.
- Multilingual support beyond English initially.

FRD (Functional Requirements Document)

Building upon the PRDâ€™s vision, this FRD specifies detailed functional behaviors, ensuring alignment with user needs and technical feasibility.

System Modules and Requirements

1. **KG Construction Module (FR-001)**:
    - Input: Unstructured text (e.g., PDFs, web articles), structured data (CSVs).
    - Functionality: Extract entities using NER (e.g., spaCy or BERT-based), relations via RE models (e.g., REBEL). Merge with ontologies (e.g., WordNet).
    - Output: Populated KG with nodes, edges, and metadata.
    - Validation: Ensure no duplicate entities; use entity resolution algorithms.
2. **Query Processing Module (FR-002)**:
    - Input: Natural language question.
    - Functionality: Parse intent with LLM (e.g., prompt: â€œTranslate to graph queryâ€), execute multi-hop traversals (e.g., shortest path algorithms in graph DB).
    - Output: Relevant subgraphs or fact triples.
    - Edge Cases: Handle ambiguous queries with clarification prompts.
3. **Generative AI Integration Module (FR-003)**:
    - Input: Query results from KG.
    - Functionality: Feed into LLM prompt template (e.g., â€œUsing facts: {facts}, answer {question} with step-by-step reasoningâ€).
    - Output: Natural language response with structured JSON for reasoning chains.
4. **Explanation and Confidence Module (FR-004)**:
    - Input: Query paths and sources.
    - Functionality: Generate human-readable paths (e.g., â€œEntity A relates to B via Câ€); compute confidence as weighted average (graph density * source score, where source score from metadata 0-1).
    - Output: Annotated response; threshold alerts if <0.6.
5. **Visualization Module (FR-005)**:
    - Input: Subgraph.
    - Functionality: Render interactive graphs (nodes clickable for details) using libraries like vis.js.
    - Output: Embeddable HTML/JS for web UI.

Interfaces and Integrations

- UI: Web-based with query input, response display, and viz panel.
- API: RESTful endpoints (e.g., POST /query with JSON body).
- Data Flow: User query -> Parse -> KG Retrieve -> LLM Generate -> Score & Viz -> Response.

Error Handling and Validation

- Invalid Query: Return suggestions via LLM.
- KG Gaps: Flag in confidence; suggest data augmentation.
- Functional Tests: Unit tests for each module (e.g., 90% coverage).

NFRD (Non-Functional Requirements Document)

Leveraging PRD goals and FRD specs, NFRD defines quality attributes for robustness.

Performance Requirements

- Latency: Query response <3s for graphs <50k nodes; scale with sharding.
- Throughput: 200 queries/min on standard hardware (16GB RAM, GPU optional).

Scalability and Availability

- Horizontal scaling: Containerized (Docker) for KG DB clusters.
- Uptime: 99.5%; use redundant DB instances.

Security and Privacy

- Authentication: OAuth for user access.
- Data Handling: Anonymize PII in entities; encrypt graph data at rest.
- Compliance: Audit logs for queries.

Reliability and Maintainability

- Error Rate: <1% failure; auto-retry on transient DB errors.
- Code Quality: Modular design, CI/CD pipeline, 85% test coverage.
- Monitoring: Integrate Prometheus for KG size, query times.

Usability and Accessibility

- UI/UX: Responsive design, keyboard navigation (WCAG 2.1 AA).
- Documentation: API docs with Swagger.

Environmental Constraints

- Deployment: Cloud-agnostic (AWS, GCP); support on-prem.
- Cost: Optimize for <0.01 USD per query.

AD (Architecture Diagram)

```
+--------------------+
| User Interface     |  (React.js: Query Input, Response Display, Interactive Viz)
+--------------------+
          |
          v
+--------------------+
| API Gateway        |  (FastAPI: Endpoints for Query, KG Upload)
+--------------------+
 /         |         \
v          v          v
+--------------------+ +--------------------+ +--------------------+
| KG Builder         | | Query Processor    | | LLM Integrator     |
| (spaCy, Transformers| | (Neo4j Cypher)     | | (HuggingFace API)  |
| for NER/RE)        | +--------------------+ +--------------------+
+--------------------+         |
          |                   v
          v          +--------------------+
+--------------------+ | Explainer & Scorer |
| Knowledge Graph DB | | (Path Gen, Conf Calc)
```

```
| (Neo4j: Nodes/Edges)| +--------------------+
+--------------------+          |
                               v
                      +--------------------+
                      | Visualization Engine|
                      | (vis.js/D3.js)     |
                      +--------------------+
```

This layered architecture separates concerns for modularity.

HLD (High Level Design)

- **System Components**:
  - Frontend: React with Redux for state, integrated viz libraries.
  - Backend: Python FastAPI for APIs, Celery for async KG builds.
  - Data Layer: Neo4j for KG storage; vector embeddings for hybrid search.
  - AI Layer: Hugging Face Transformers for NER/RE/LLM; fine-tune on domain data.
- **Design Patterns**:
  - Microservices for scalability.
  - Observer for real-time viz updates.
  - Pipeline for data flow (ingest -> query -> respond).
- **Data Management**:
  - Sources: Public like Freebase, HotpotQA for QA pairs, schema.org ontologies.
  - Storage: Indexed nodes for fast traversal.
- **Security Design**:
  - JWT tokens for API auth.
- **High-Level Flow**:
  1. Ingest text -> Build KG.
  2. Query -> Parse to Cypher -> Retrieve -> LLM enhance -> Score & Viz.

LLD (Low Level Design)

- **KG Construction LLD**:
  - NER: Use pipeline = spacy.load(â€œen_core_web_trfâ€); entities = [ent.text for ent in doc.ents].
  - RE: Fine-tuned model like â€œBabelscape/rebel-largeâ€; extract triples from model output.
  - Merge: Use graph.merge(Node(â€œEntityâ€, name=ent, source_meta=reliability)).
- **Query Processing LLD**:
  - Parse: LLM prompt: â€œGenerate Cypher for: {question}. Entities: {extracted}â€.
  - Execute: driver.session().run(query, params); handle paths with BFS if needed.
- **Generative Integration LLD**:
  - Prompt Engineering: Chain-of-thought template with facts injected.
  - Model: tokenizer.encode(prompt); model.generate(max_length=200).
- **Confidence LLD**:
  - Formula: confidence = (1 / path_length) * avg_source_reliab * (connected_components / total_nodes).
  - Threshold: If <0.5, append â€œLow confidence due to sparse dataâ€.
- **Visualization LLD**:
  - Data Prep: Convert Neo4j results to JSON {nodes: [], links: []}.
  - Render: Use force-directed layout in vis.js; add tooltips for metadata.

Pseudocode

```
class KGQASystem:
    def __init__(self):
        self.graph = Neo4jDriver(uri, auth)
        self.ner_model = spacy.load("en_core_web_trf")
        self.re_model = load_rebel()
        self.llm = HuggingFaceModel("meta-llama/Llama-2-7b")

    def build_kg(self, text):
        doc = self.ner_model(text)
        entities = extract_entities(doc)
        relations = self.re_model(entities, text)
        for sub, pred, obj, rel_meta in relations:
            self.graph.add_node(sub, props)
            self.graph.add_node(obj, props)
```

```
            self.graph.add_edge(sub, pred, obj, rel_meta)

    def process_query(self, question):
        extracted_ents = extract_from_question(question)
        cypher = self.llm.generate_prompt("To Cypher: ", question, extracted_ents)
        results = self.graph.execute(cypher)
        if not results:
            return fallback_llm(question)
        reasoning_paths = build_paths(results)  # List of string paths
        prompt = f"Facts: {results}\nPaths: {reasoning_paths}\nAnswer: {question}"
        response = self.llm.generate(prompt)
        confidence = compute_conf(results, reasoning_paths)
        viz_data = subgraph_to_json(results)
        return {"answer": response, "reasoning": reasoning_paths, "confidence": confidence, "viz": viz_data}
```

This pseudocode emphasizes modularity and error handling.

140509_21.md

README

21. Model Quantization and Fine-tuning Platform

Summary: Develop a platform that enables efficient model quantization and fine-tuning for deploying large language models on resource-constrained environments.

Problem Statement: Large language models require significant computational resources, limiting their deployment in edge environments. Your task is to create a platform that automates model quantization, fine-tuning, and optimization for specific use cases while maintaining performance quality. The system should support various quantization techniques, provide performance benchmarking, and enable easy deployment to different hardware configurations.

Steps:

â€¢ Design automated quantization pipeline supporting multiple techniques (INT8, INT4, dynamic)

â€¢ Implement fine-tuning workflows with parameter-efficient methods (LoRA, QLoRA)

â€¢ Create performance benchmarking suite measuring accuracy, speed, and memory usage

â€¢ Build deployment optimization for different hardware targets (CPU, GPU, mobile)

â€¢ Develop model comparison and selection tools based on constraints

â€¢ Include monitoring and quality assessment for quantized models

Suggested Data Requirements:

â€¢ Pre-trained model checkpoints and configuration files

â€¢ Domain-specific fine-tuning datasets

â€¢ Hardware performance benchmarks and constraints

â€¢ Quality evaluation datasets for model comparison

Themes: GenAI & its techniques, Quantization, Fine-tuning

The steps and data requirements outlined above are intended solely as reference points to assist you in conceptualising your solution.

PRD (Product Requirements Document)

Product Vision and Goals

To democratize LLM deployment on edge devices by automating optimization, reducing model size by 4x-8x while retaining >95% accuracy. Goals: Support 10+ quantization methods, integrate with 5 hardware types, and provide one-click deployment.

Target Audience and Stakeholders

- Primary Users: ML engineers, mobile app developers, IoT specialists.
- Stakeholders: Hardware vendors for benchmarks, end-users for inference.
- Personas: An edge AI developer optimizing GPT-J for Raspberry Pi.

Key Features and Functionality

- Auto-quantization with technique selection.
- PEFT (Parameter-Efficient Fine-Tuning) workflows.
- Multi-metric benchmarking dashboard.
- Hardware-specific exporters (e.g., TFLite for mobile).
- Model selector with constraint-based ranking.
- Post-deployment monitoring for drift.

Business Requirements

- Open-source with enterprise edition for cloud integration.
- Integration with Hugging Face Hub for model loading.

Success Metrics

- Efficiency: >2x speed-up on target hardware.
- User Adoption: 1000+ downloads in first year.
- Quality: Perplexity <5% increase post-quantization.

Assumptions, Risks, and Dependencies

- Assumptions: Users have basic PyTorch knowledge.
- Risks: Accuracy loss in quantization; mitigate with calibration datasets.
- Dependencies: Libraries like bitsandbytes for QLoRA, public models from HF.

Out of Scope

- Custom hardware acceleration (e.g., FPGA design).
- Online learning during inference.

FRD (Functional Requirements Document)

System Modules and Requirements

1. **Quantization Pipeline (FR-001)**:
    - Input: Model checkpoint, calibration data.
    - Functionality: Support PTQ (Post-Training Quant), QAT; techniques: static INT8, dynamic, FP16.
    - Output: Quantized model with config.
2. **Fine-Tuning Workflow (FR-002)**:
    - Input: Quantized model, dataset.
    - Functionality: Apply LoRA/QLoRA; trainers with PEFT library.
    - Output: Adapted model adapters.
3. **Benchmarking Suite (FR-003)**:
    - Input: Models, eval dataset, hardware spec.
    - Functionality: Measure accuracy (e.g., BLEU), latency (ms), memory (MB), power (if sim).
    - Output: Comparative reports, graphs.
4. **Deployment Optimization (FR-004)**:
    - Input: Model, target (CPU/GPU/Android).
    - Functionality: Convert to ONNX/TFLite/CoreML; optimize ops.
    - Output: Deployable binary.
5. **Model Comparison (FR-005)**:
    - Input: Multiple models, constraints (e.g., max 1GB RAM).
    - Functionality: Rank by Pareto front (accuracy vs size).
    - Output: Recommended model.

Interfaces and Integrations

- UI: Web app for uploading, visualizing benchmarks.
- API: CLI commands like `quantize --model gpt2 --tech int8`.
- Data Flow: Load model -> Quantize -> Fine-tune -> Benchmark -> Deploy -> Monitor.

Error Handling and Validation

- Validation: Auto-check accuracy drop; rollback if >threshold.
- Errors: Handle incompatible hardware with warnings.

NFRD (Non-Functional Requirements Document)

Performance Requirements

- Process Time: Quantize 7B model <30min on V100 GPU.
- Inference: <50ms/token on mobile.

Scalability and Availability

- Handle models up to 70B params.
- Cloud deployable with auto-scaling.

Security and Privacy

- Secure model uploads; no data retention.
- Compliance: MIT license for open components.

Reliability and Maintainability

- Fault Tolerance: Resume interrupted fine-tuning.
- Code: 90% coverage, modular plugins for new techs.

Usability and Accessibility

- Intuitive GUI with tutorials.
- Support dark mode, screen readers.

Environmental Constraints

- Run on CPU-only for low-end users.

AD (Architecture Diagram)

```
+--------------------+
| User Interface/CLI |  (Streamlit: Upload, Config, Viz Dashboard)
+--------------------+
          |
          v
+--------------------+
| Workflow Orchestrator |  (Airflow/Dagster: Pipeline Management)
+--------------------+
 /      |      |      \
v       v      v       v
+-----+ +-----+ +-----+ +-----+
| Quant| | Fine| | Bench| | Deploy|
| Pipe | | Tune | | Suite| | Opt  |
+-----+ +-----+ +-----+ +-----+
          |             |
          v             v
+--------------------+ +--------------------+
| Model Registry     | | Monitoring Agent   |
| (HF Hub)          | | (Prometheus)       |
+--------------------+ +--------------------+
```

HLD (High Level Design)

- **Components**:

- Orchestrator: Use Hugging Face Accelerate for distributed.
    - Quant: Torch.quantization, bitsandbytes.
    - Benchmark: Torch Profiler, hardware sims.
    - Deployment: ONNX Runtime.
- **Design Patterns**:
    - Factory for quantization types.
    - Observer for monitoring.
- **Data Management**:
    - Datasets: Alpaca for fine-tune, GLUE for eval.
- **High-Level Flow**:
    1. Config input -> Run pipeline stages sequentially or parallel.
    2. Store artifacts in registry.

LLD (Low Level Design)

- **Quantization LLD**:
    - Static INT8: model = torch.quantization.quantize(model, qconfig_spec, inplace=False)
    - Calibration: Run forward passes on 1000 samples.
- **Fine-Tuning LLD**:
    - LoRA Config: from peft import LoraConfig; config = LoraConfig(r=16, lora_alpha=32)
    - Trainer: from transformers import Trainer; trainer.train()
- **Benchmark LLD**:
    - Accuracy: from evaluate import load; acc = load(â€œaccuracyâ€).compute(preds, refs)
    - Latency: with torch.profiler.profile(): model(input); print(profile.key_averages())
- **Comparison LLD**:
    - Pareto: Use scipy.optimize for multi-objective ranking.

Pseudocode

```
class QuantFinePlatform:
    def __init__(self):
        self.hf_hub = HFHub()

    def quantize(self, model_name, tech='int8', calib_data):
        model = self.hf_hub.load_model(model_name)
        if tech == 'int8':
            q_model = torch.quantization.quantize_dynamic(model, {nn.Linear: torch.qint8})
        elif tech == 'int4':
            q_model = bitsandbytes.quantize(model, 4)
        q_model.calibrate(calib_data)
        return q_model

    def fine_tune(self, q_model, dataset, method='qlora'):
        config = LoraConfig(...) if method == 'lora' else QLoRAConfig(...)
        peft_model = get_peft_model(q_model, config)
        trainer = Trainer(peft_model, train_dataset=dataset, eval_dataset=val)
        trainer.train()
        return peft_model

    def benchmark(self, models, eval_data, hardware='cpu'):
        results = []
        for m in models:
            acc = evaluate_model(m, eval_data)
            lat, mem = profile_inference(m, hardware)
            results.append({'acc': acc, 'lat': lat, 'mem': mem})
        return results

    def deploy(self, model, target='mobile'):
        if target == 'mobile':
            converted = convert_to_tflite(model)
        return converted

    def compare(self, benchmarks, constraints):
        filtered = [b for b in benchmarks if b['mem'] < constraints['max_mem']]
        ranked = sort_by_pareto(filtered, keys=['acc', '-lat'])
        return ranked[0]
```