

## APPENDIX B “Évaluation Code SKG-LLM-RAG”

```
from pathlib import Path

import json, math

from collections import defaultdict, Counter

from typing import Dict, List, Tuple, Iterable


import numpy as np

import pandas as pd


_HAS_ST = False

try:

    from sentence_transformers import SentenceTransformer

    _HAS_ST = True

except Exception:

    _HAS_ST = False


from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer

from sklearn.metrics.pairwise import cosine_similarity

from sklearn.decomposition import LatentDirichletAllocation


def read_jsonl(path: str) -> List[dict]:

    items = []

    with open(path, "r", encoding="utf-8") as f:

        for line in f:

            line = line.strip()
```

```

    if not line:
        continue
    items.append(json.loads(line))
return items

```

```

def write_jsonl(path: str, records: Iterable[dict]) -> None:

```

```

    p = Path(path)
    with p.open("w", encoding="utf-8") as f:
        for r in records:
            f.write(json.dumps(r, ensure_ascii=False) + "\n")

```

```

def _ndcg_at_k(ranked_ids: List[str], gold_ids: List[str], k: int) -> float:

```

```

    rel = [1 if x in set(gold_ids) else 0 for x in ranked_ids[:k]]
    dcg = 0.0
    for i, r in enumerate(rel, start=1):
        dcg += (2**r - 1) / math.log2(i + 1)
    idcg = (2**1 - 1) / math.log2(1 + 1) * min(len(gold_ids), 1)
    return dcg / idcg if idcg > 0 else 0.0

```

```

def _recall_at_k(ranked_ids: List[str], gold_ids: List[str], k: int) -> float:

```

```

    return 1.0 if set(ranked_ids[:k]) & set(gold_ids) else 0.0

```

```

class _SentenceEncoder:

```

```

    def __init__(self, docs: Dict[str, str], model_name: str = "all-MiniLM-L6-v2"):
        self.keys = list(docs.keys())
        self.docs = [docs[k] for k in self.keys]

```

```

self.backend = "st" if _HAS_ST else "tfidf"

if self.backend == "st":
    try:
        self.model = SentenceTransformer(model_name)

        self.doc_mat = self.model.encode(self.docs, normalize_embeddings=True)

    except Exception:
        self.backend = "tfidf"

if self.backend == "tfidf":
    self.vec = TfidfVectorizer(ngram_range=(1, 2), min_df=1)

    self.doc_mat = self.vec.fit_transform(self.docs)

def rank(self, query: str):
    if self.backend == "st":
        qv = self.model.encode([query], normalize_embeddings=True)

        sims = (qv @ self.doc_mat.T).ravel()

    else:
        qv = self.vec.transform([query])

        sims = cosine_similarity(qv, self.doc_mat).ravel()

    order = np.argsort(-sims)

    return [self.keys[i] for i in order], sims[order].tolist()

class _BM25:
    def __init__(self, documents: Dict[str, str], k1=1.5, b=0.75):
        self.keys = list(documents.keys())

        self.docs = [documents[k] for k in self.keys]

        self.k1, self.b = k1, b

        self.N = len(self.docs)

```

```

self.toks = [self._tok(d) for d in self.docs]

self.doc_len = [len(ts) for ts in self.toks]

self.avgdl = np.mean(self.doc_len) if self.doc_len else 0.0

self.df = Counter()

self.tf = []

for ts in self.toks:

    c = Counter(ts); self.tf.append(c)

    for t in c: self.df[t] += 1

self.idf = {t: math.log(1 + (self.N - df + 0.5) / (df + 0.5)) for t, df in self.df.items()}

def _tok(self, s): return s.lower().split()

def rank(self, query: str):

    q = self._tok(query)

    scores = np.zeros(self.N, dtype=float)

    for i, tf in enumerate(self.tf):

        dl = self.doc_len[i]

        denom = self.k1 * (1 - self.b + self.b * dl / (self.avgdl if self.avgdl > 0 else 1.0))

        s = 0.0

        for term in q:

            if term not in self.df: continue

            idf = self.idf.get(term, 0.0)

            f = tf.get(term, 0)

            if f == 0: continue

            s += idf * (f * (self.k1 + 1)) / (f + denom)

        scores[i] = s

    order = np.argsort(-scores)

    return [self.keys[i] for i in order], scores[order].tolist()

```

```

class SemanticRetrievalEvaluatorPlus:

    def __init__(self, queries_csv: str):

        qdf = pd.read_csv(queries_csv)

        self.queries = []

        for _, r in qdf.iterrows():

            gold = str(r["gold_var"])

            golds = gold.split("|") if "|" in gold else [gold]

            self.queries.append({"query_id": int(r["query_id"]), "query_text": r["query_text"],
"gold_ids": golds})

        @staticmethod

        def load_flat_corpus(jsonl_path: str) -> Dict[str, str]:

            corpus = {}

            for it in read_jsonl(jsonl_path):

                vid = it.get("var_code") or it.get("id", "").replace("var:", "")

                corpus[vid] = it.get("text", "")

            return corpus

        @staticmethod

        def load_skg_corpus(nodes_csv: str, kg_jsonl: str) -> Dict[str, str]:

            nodes = pd.read_csv(nodes_csv)

            glosses = defaultdict(list)

            for obj in read_jsonl(kg_jsonl):

                if obj.get("type") == "edge" and str(obj.get("h", "")).startswith("var:"):

                    var_code = obj["h"].split("var:")[-1]

                    gl = obj.get("statement_gloss", "")

                    if gl:

```

```

        glosses[var_code].append(gl)

corpus = {}

for _, row in nodes.iterrows():

    v = row["var_code"]

    text = f"{row['label']} {row['canonical_description']} " + " ".join(glosses.get(v, []))

    corpus[v] = text

return corpus

def evaluate_corpus(self, model_name: str, corpus: Dict[str, str],

                    sentence_embeddings: bool = True, topks=(1, 5, 10),

                    return_rankings: bool = False):

    if model_name == "BM25":

        ranker = _BM25(corpus)

    else:

        ranker = _SentenceEncoder(corpus) if sentence_embeddings else
_SentenceEncoder(corpus)

    rows = []

    rankings = []

    for q in self.queries:

        ranked_ids, scores = ranker.rank(q["query_text"])

        row = {"model": model_name, "query_id": q["query_id"]}

        for k in topks:

            row[f"recall@{k}"] = _recall_at_k(ranked_ids, q["gold_ids"], k)

            row[f"ndcg@{k}"] = _ndcg_at_k(ranked_ids, q["gold_ids"], k)

        rows.append(row)

    if return_rankings:

        rankings.append({

```

```

        "model": model_name,
        "query_id": q["query_id"],
        "ranked_ids": ranked_ids[:50],
        "scores": scores[:50]
    })

per_query = pd.DataFrame(rows)

return per_query, rankings, getattr(ranker, "backend", "bm25")

@staticmethod
def bootstrap_summary(per_query_df: pd.DataFrame, topks=(1, 5, 10), B: int = 300, seed:
int = 0) -> pd.DataFrame:

    rng = np.random.default_rng(seed)

    metrics = [f"recall@{k}" for k in topks] + [f"ndcg@{k}" for k in topks]

    out = []

    for model, g in per_query_df.groupby("model"):

        vals = {m: g[m].values for m in metrics}

        for m in metrics:

            boots = []

            for _ in range(B):

                idx = rng.integers(0, len(vals[m]), size=len(vals[m]))

                boots.append(np.mean(vals[m][idx]))

            mu = float(np.mean(boots))

            lo = float(np.percentile(boots, 2.5))

            hi = float(np.percentile(boots, 97.5))

            out.append({"model": model, "metric": m, "mean": mu, "ci_low": lo, "ci_high": hi})

    return pd.DataFrame(out)

```

```

class LinkPredictionEvaluatorPlus:

    def __init__(self, kg_jsonl: str):

        self.nodes = {}

        self.edges = []

        for obj in read_jsonl(kg_jsonl):

            if obj.get("type") == "node":

                self.nodes[obj["id"]] = obj

            elif obj.get("type") == "edge":

                self.edges.append(obj)

        self.tail_candidates_by_r = defaultdict(set)

        for e in self.edges:

            self.tail_candidates_by_r[e["r"]].add(e["t"])

    @staticmethod
    def _node_text(node: dict) -> str:

        return " ".join([

            node.get("label", ""),

            node.get("description", ""),

            node.get("canonical_description", ""),

        ])

    def _rank_tails(self, h_id: str, r: str, tails: List[str]):

        h_text = self._node_text(self.nodes.get(h_id, {}))

        rel_text = f"[REL {r}]"

        q_text = f"{h_text} {rel_text}"

        cand_texts = []

        cand_ids = []

        for t in tails:

```



```

    cand_ids.append(t)

    cand_texts.append(self._node_text(self.nodes.get(t, {})) + f" {rel_text}")

    encoder = _SentenceEncoder({cid: txt for cid, txt in zip(cand_ids, cand_texts)})

    ranked_ids, scores = encoder.rank(q_text)

    return ranked_ids, scores

def evaluate(self, num_samples: int = 200, hits_ks=(1, 5, 10)) -> pd.DataFrame:

    rng = np.random.default_rng(0)

    pool = [e for e in self.edges if e["r"] in ("described_by", "applies_to", "has_measure",
"tabulated_at", "contains")]

    if not pool:

        return pd.DataFrame([{"metric": "MRR", "value": 0.0}])

    samples = rng.choice(pool, size=min(num_samples, len(pool)), replace=False)

    ranks = []

    for e in samples:

        tails = list(self.tail_candidates_by_r[e["r"]])

        if e["t"] not in tails:

            tails.append(e["t"])

        ranked, _ = self._rank_tails(e["h"], e["r"], tails)

        rnk = ranked.index(e["t"]) + 1

        ranks.append(rnk)

    mrr = float(np.mean([1.0 / r for r in ranks]))

    rows = [{"metric": "MRR", "value": mrr}]

    for k in hits_ks:

        hits = float(np.mean([1.0 if r <= k else 0.0 for r in ranks]))

        rows.append({"metric": f"Hits@{k}", "value": hits})

    return pd.DataFrame(rows)

```

```
class TopicDiscoveryEvaluatorPlus:
```

```
    def __init__(self, texts: List[str]):
```

```
        self.texts = texts
```

```
    @staticmethod
```

```
    def _npmi(word_i: str, word_j: str, cooc: Counter, occ: Counter, N: int, eps: float = 1e-12)
```

```
-> float:
```

```
    pij = (cooc[(word_i, word_j)] + eps) / N
```

```
    pi = (occ[word_i] + eps) / N
```

```
    pj = (occ[word_j] + eps) / N
```

```
    pmi = math.log(pij / (pi * pj))
```

```
    return pmi / (-math.log(pij))
```

```
def _build_windows(self, tokens_list: List[List[str]], L: int = 15):
```

```
    occ = Counter()
```

```
    cooc = Counter()
```

```
    N = 0
```

```
    for toks in tokens_list:
```

```
        if not toks:
```

```
            continue
```

```
        for i in range(0, len(toks), L):
```

```
            window = toks[i:i + L]
```

```
            if not window:
```

```
                continue
```

```
            N += 1
```

```
            unique = list(dict.fromkeys(window))
```

```
            for w in unique:
```

```

        occ[w] += 1

    for a_i in range(len(unique)):
        for b_i in range(a_i + 1, len(unique)):
            a, b = sorted([unique[a_i], unique[b_i]])
            cooc[(a, b)] += 1

    return cooc, occ, N

def evaluate(self, K: int = 8, M: int = 20, L: int = 15, max_features: int = 5000) ->
pd.DataFrame:

    cv = CountVectorizer(max_features=max_features, stop_words="english")

    X = cv.fit_transform(self.texts)

    vocab = np.array(cv.get_feature_names_out())

    lda = LatentDirichletAllocation(n_components=K, random_state=0,
learning_method="batch")

    lda.fit(X)

    topic_words = []

    for comp in lda.components_:
        top_idx = np.argsort(-comp)[:M]
        topic_words.append(vocab[top_idx].tolist())

    tokens_list = [doc.lower().split() for doc in self.texts]

    cooc, occ, N = self._build_windows(tokens_list, L=L)

    npmis = []

    for words in topic_words:
        pair_vals = []

        for i in range(len(words)):
            for j in range(i + 1, len(words)):
                a, b = sorted([words[i], words[j]])

                pair_vals.append(self._npmi(a, b, cooc, occ, N))

```

```

    npmis.append(np.mean(pair_vals) if pair_vals else 0.0)
U = len(set(w for tw in topic_words for w in tw))
TD = U / (K * M) if K * M > 0 else 0.0
rows = [{"metric": "NPMI", "value": float(np.mean(npmis))},
        {"metric": "TopicDiversity", "value": float(TD)}]
return pd.DataFrame(rows)

```

```

class GroundedQAEvaluatorPlus:
    def __init__(self, outputs_jsonl: str):
        self.records = read_jsonl(outputs_jsonl)
    def evaluate(self, ks=(1, 3)) -> pd.DataFrame:
        rows = []
        hit_at_k = {k: [] for k in ks}
        precisions, recalls, f1s = [], [], []
        support_coverage = []
        for r in self.records:
            sys_cites = [c.strip() for c in r.get("citations", [])]
            gold_cites = [c.strip() for c in r.get("gold_citations", [])]
            support_coverage.append(1.0 if sys_cites else 0.0)
            for k in ks:
                hit_at_k[k].append(1.0 if set(sys_cites[:k]) & set(gold_cites) else 0.0)
            tp = len(set(sys_cites) & set(gold_cites))
            fp = len([c for c in sys_cites if c not in set(gold_cites)])
            fn = len([c for c in gold_cites if c not in set(sys_cites)])
            prec = tp / (tp + fp) if (tp + fp) > 0 else 0.0
            rec = tp / (tp + fn) if (tp + fn) > 0 else 0.0

```

```

    f1 = 2 * prec * rec / (prec + rec) if (prec + rec) > 0 else 0.0

    precisions.append(prec); recalls.append(rec); f1s.append(f1)

    rows.append({"metric": "SupportCoverage", "value":
float(np.mean(support_coverage))})

    for k in ks:

        rows.append({"metric": f"CitationHit@{k}", "value": float(np.mean(hit_at_k[k]))})

    rows.append({"metric": "CitationPrecision", "value": float(np.mean(precisions))})

    rows.append({"metric": "CitationRecall", "value": float(np.mean(recalls))})

    rows.append({"metric": "CitationF1", "value": float(np.mean(f1s))})

    return pd.DataFrame(rows)

```