Department of Information Engineering

Master thesis in Artificial Intelligence and Data Engineering

Learning What to Crawl: Metadata-Enriched LLMs for Crawling Frontier Prioritisation

Supervisors:
Prof: Nicola Tonellotto
Dott.ssa: Francesca Pezzuti

Candidate **:**
Niccolò Settimelli

Academic Year 2024/2025

Abstract

The exponential growth of the Web requires crawlers to efficiently discover valuable content. Classical strategies such as PageRank rely on global link analysis, which, while effective, is computationally expensive and less practical for real-time frontier management.

This thesis proposes a shift in perspective: from global graph-based scoring to semantic-aware frontier prioritization. Leveraging transformer-based neural models, we estimate the potential of a page to lead toward relevant content through its outlinks, based solely on its textual and metadata features. This reframing from page-centric quality estimation to outlink utility prediction aligns crawling with the broader transition from keyword-based retrieval to semantic search.

The proposed approach integrates semantic modelling and metadata fusion to generate real-time priority scores that guide the crawler toward semantically rich content. By embedding neural estimation directly into the crawling process, our methodology improves efficiency, surfaces higher-quality documents earlier, and avoids the prohibitive costs of repeated graph-wide computations.

This work highlights how modern neural architectures can redefine the principles of web crawling, paving the way for next-generation crawlers that are lighter, adaptive, and better aligned with semantic search paradigms.

Chapter 1

Code and Implementation

This section documents the practical setup used to reproduce our finetuning experiments, the repository layout, and the exact launch commands. The goal is to make every step auditable, from dataset construction to model training and score extraction for crawler integration.

1.1 Repository and Environment

The codebase is publicly available on GitHub as `nikisetti01/crawler-quality-scoring-model` Àll experiments were run in a Python $\geq$ 3.10 environment to ensure compatibility with the `pyterrier-quality` stack used for QualScore conversion.

- **Virtual environment.**

```
python3 -m venv .venv
source .venv/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
```

- **Accelerate configuration (one-time).**

```
accelerate config
# select: multi-GPU (if available), mixed precision (fp16 or bf16),
# and gradient accumulation as required by GPU memory.
```

Repository Structure

The repository is organized by experimental task; each top-level folder may contain dedicated subfolders/files for *text-only* (prefix `only_text`) and *metadata* (prefix `metadata_`).

- **Dataset_generation**: scripts to build the two main training corpora (sampling, cleaning, joins).

- **Finetuning**: training scripts for all model families:
  - `finetune_only_text.py`: BERT text-only baseline (binary classification).
  - `finetune_metadata.py`: BERT with metadata fusion (additive Uni-Attention).
- **Test**: evaluation scripts on held-out sets to verify that fine-tuning improves intrinsic metrics.
- **Results**: notebooks for plots/tables reporting learning curves, ROC/AUC, calibration, and ablations.
- **Inference**: batch inference utilities to score large collections and convert scores into *QualScore* via `pyterrier-quality` (local `QualCache` hub).

Pipeline Orchestration (Dataset Creation)

Dataset construction is automated by a lightweight pipeline driver that sequentially executes preprocessing stages and logs outputs to disk. The following example mirrors the style of our internal scripts:[1]

```
#!/usr/bin/env bash
set -euo pipefail

# Select which models/datasets to prepare (arrays can be expanded)
models=("bert-base-uncased")
datasets=("train-split-a")

for MODEL_PATH in "${models[@]}"; do
  for DATASET_NAME in "${datasets[@]}"; do

    source ./scripts/models_config.sh   # load per-model hyperparameters

    python process_links.py \
      --dataset ${DATASET_NAME} \
      --out tmp/links_${DATASET_NAME}.csv

    python division_fast.py \
      --in  tmp/links_${DATASET_NAME}.csv \
      --test_out data/test_set.csv \
      --links_out data/link_labels2_cleaned.csv

    python add_text.py \
      --links data/link_labels2_cleaned.csv \
      --text_corpus /data/text \
      --train_out data/dataset_train.csv \
```

[1]The actual filenames may differ depending on your local dataset sources; the orchestration pattern remains identical.

```
      --val_out   data/dataset_val.csv

    python balancer.py \
      --in  data/dataset_train.csv \
      --out data/dataset_balanced.csv \
      --strategy label_ratio --pos_ratio 0.5

  done
done
```

Scripts and roles. *(i)* `process_links.py`: extracts/normalizes link-level labels; *(ii)* `division_fast.py`: builds a balanced test split and cleans link labels; *(iii)* `add_text.py`: joins labels with cleaned page text to produce `dataset_train.csv` and `dataset_val.csv`; *(iv)* `balancer.py`: optionally re-balances the training set according to a target positive ratio.

Finetuning Launch Commands

We use `accelerate launch` as a drop-in replacement for `python`, enabling multi-GPU and mixed precision where available.

Text-only BERT (binary classification).

```
accelerate launch --mixed_precision=fp16 finetune_only_text.py \
  --model_name bert-base-uncased \
  --train_csv data/dataset_train.csv \
  --val_csv   data/dataset_val.csv \
  --out       ckpts/best_bert.pt \
  --batch_size 32 \
  --lr 2e-5 \
  --epochs 3 \
  --max_len 512 \
  --sep ","
```

BERT + Metadata (additive Uni-Attention).

```
accelerate launch --mixed_precision=fp16 finetune_metadata.py \
  --model_name bert-base-uncased \
  --train_csv data/dataset_train_metadata.csv \
  --val_csv   data/dataset_val_metadata.csv \
  --out       ckpts/best_meta.pt \
  --batch_size 24 \
  --lr 2e-5 \
  --epochs 3 \
  --max_len_text 512 \
  --max_len_meta 64 \
  --H 128 --A 64 --V_dom 50000 \
  --num_cols num_inlinks,num_outlinks,outlinks_domains_count,inlinks_domains_cou
  --sep ","
```

User-Selectable Parameters

The following arguments are surfaced to the user at launch time; they control data access, model choice, and training budget:

- `-model_name`: HF encoder backbone (e.g., `bert-base-uncased`).
- `-train_csv`, `-val_csv`: input files with columns `text,label` and, for metadata runs, anchor/domain/numeric fields as specified in the repository readme.
- `-out`: destination path for the best `state_dict`.
- `-batch_size`, `-lr`, `-epochs`: standard optimization knobs.
- `-max_len` / `-max_len_text`, `-max_len_meta`: token limits for text and metadata encoders.
- `-H`, `-A`, `-V_dom`: metadata latent dim, additive-attention hidden size, and hashing space for domains.
- `-num_cols`: comma-separated list of numeric metadata columns to include (automatically projected and fused).
- `-sep`: CSV separator (`,` or `\t`).
- `accelerate` options: device visibility (`CUDA_VISIBLE_DEVICES`), mixed precision (`fp16`/`bf16`), gradient accumulation.

1.2 Finetuning Experiment Setup and Optimization

The finetuning stage constitutes the core of our experimental framework and is organized into two main subdirectories: `only_text_finetuning` and `metadata_finetuning`. In the text-only case (`binary_classification` and `score_estimation`), the training pipeline differs structurally from the metadata setting. For text-only runs, we leverage existing modules from Hugging Face `transformers` and `datasets` to manage textual inputs, tokenize them, and feed them into either encoder-based models with a classification head (e.g., BERT, DeBERTa) or encoder–decoder models (QualT5, derived from T5).

The only customization added in this setting was a `WeightedTrainer` class, extending Hugging Face's standard `Trainer` to apply a class-weighted cross-entropy loss, thereby accounting for the imbalance in the label distribution:

```
class WeightedTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False):
        labels = inputs.pop("labels")
        outputs = model(**inputs)
        logits = outputs.logits
        loss_fct =
    torch.nn.CrossEntropyLoss(weight=class_weights_tensor)
        loss = loss_fct(
            logits.view(-1, model.config.num_labels),
```

```
            labels.view(-1)
        )
        return (loss, outputs) if return_outputs else loss
```

Listing 1.1: Weighted cross-entropy trainer for binary classification.

In addition, an auxiliary function `compute_metrics` was employed during validation and testing. This function evaluates not only the loss trajectory but also key classification metrics such as accuracy and F1-score. Once defined, the parameters for Hugging Face's `TrainingArguments` and the custom `WeightedTrainer` were set as summarized in Table 1.1, using models `bert-uncased` and `deberta-v3`.

As an additional optimization, we parallelized training across two GPUs and enabled **mixed precision training (FP16) ** [**micikevicius2017mixed**]. By performing most operations in 16-bit floating point, this approach significantly reduces GPU memory footprint and computational overhead, while retaining numerical stability through automatic casting. All metrics during both training and validation were continuously monitored through direct integration with the W&B platform, providing reliable tracking for large-scale finetuning tasks.

Table 1.1: Training setup for text-only binary classification experiments.

Tokenizer	`AutoTokenizer` (RoBERTa / DeBERTa), `max_length=512`, truncation + padding
Data split	Train/Validation 90/10 (stratified, seed=42); Test from CSV
Preprocessing	Remove invalid rows (empty / non-string text)
Batch size	Train = 64 /device; Eval = 256 /device
Learning rate	2×10^{-5}
Epochs	3
Weight decay	0.01
Precision	FP16 (mixed precision enabled)
Loss function	Weighted Cross-Entropy (balanced class weights)
Freezing policy	All frozen except `encoder.layer.10`, `encoder.layer.11`, and `classifier`
Evaluation	Every 20,000 steps
Checkpointing	Save every 20,000 steps (keep last 3); best model loaded at end
Best model metric	Accuracy
Data collator	`DataCollatorWithPadding(tokenizer)`
Optimizer / Scheduler	Hugging Face defaults (AdamW + scheduler)
Logging	Every 500 steps, reported to W&B
Training status	∼200k steps completed

1.2.1 Score Estimation (Seq2Seq, QualT5)

For score estimation we adopt a generative reformulation: given the document text, the model produces the token `"true"` or `"false"`. The main deviation from the binary classification setup lies in the input preprocessing, where we apply minimal prompt engineering to create natural language instructions and very short targets.

```
def format_prompt(example):
    # Prompt: provide the document, ask for relevance (true/false)
    prompt = f"Document: {example['text']} Relevant:"
    # Target: "true" if label = 1, else "false"
    target = "true" if int(example["label"]) == 1 else "false"
    return {"input_text": prompt, "target_text": target}
```

Listing 1.2: Prompt formatting for QualT5 score estimation.

At evaluation time, predictions are decoded and mapped to {0,1} to compute accuracy (and optional F1). Training is managed via `Seq2SeqTrainer` with `predict_with_generate=True`, ensuring that validation mirrors the inference pathway.

Table 1.2: Training setup for text-only score estimation (QualT5).

Tokenizer	`AutoTokenizer` (QualT5)
Task format	Input: `Document: <text> Relevant:` → Target: `"true"` / `"false"`
Max lengths	Input `max_length = 512` (truncation + padding); Target `max_length = 5`
Data split	Train/Validation 90/10 (stratified by label, seed = 42); Test from CSV
Preprocessing	Drop rows with empty / non-string text
Batch size	Train = 64 /device; Eval = 256 /device
Learning rate	2×10^{-5}
Epochs	3
Weight decay	0.01
Precision	FP32 (no mixed precision set)
Loss / Objective	Seq2Seq Cross-Entropy via decoder (generate `"true"`/`"false"`)
Predict with generate	`True` (evaluation uses decoding)
Freezing policy	Encoder frozen except last 2 layers; decoder trainable
Evaluation	Every 20,000 steps
Checkpointing	Save every 20,000 steps (keep last 3); load best at end
Best model metric	Accuracy
Metrics function	Decode outputs → map to 0/1 → compute Accuracy
Data collator	`DataCollatorForSeq2Seq(tokenizer, model)`
Optimizer / Scheduler	Hugging Face defaults (AdamW + scheduler)
Training status	∼180k steps completed

1.2.2 Multimodal Experimentation Setup

The multimodal experiments extend the text-only pipeline by introducing a dedicated metadata fusion layer on top of a pre-trained encoder (BERT). The architecture is implemented in three PyTorch modules: (i) a *MetadataEncoder* that projects heterogeneous metadata (anchor texts, domain lists, and numeric link statistics) into a shared hidden space; (ii) an *Additive Unified Attention* that aligns metadata tokens over the contextual textual sequence; and (iii) a *MultiModalWebClassifier* that fuses the enriched metadata representation with the textual [CLS] vector for binary prediction. The implementation follows the codebase in `model_bert.py`. :contentReference[oaicite:0]index=0

Model Components

MetadataEncoder. Each metadata source is mapped through a small MLP into a common dimensionality and stacked as a short "meta sequence". This yields a compact representation amenable to cross-modal attention.

```
class MetadataEncoder(nn.Module):
    """
    Encode heterogeneous metadata sources (anchors, domains, numerics)
    into a unified hidden representation. Each feature type is projected
    through a small feed-forward layer into the same hidden space.
    """
    def __init__(self, hidden_dim):
        super().__init__()
        self.anchor_encoder = nn.Sequential(nn.Linear(32, hidden_dim),
    nn.ReLU())
        self.domain_proj    = nn.Sequential(nn.Linear(64, hidden_dim),
    nn.ReLU())
        self.numeric_proj   = nn.Sequential(nn.Linear(8,  hidden_dim),
    nn.ReLU())

    def forward(self, anchor_out, anchor_in, domain_out, domain_in,
    numerics):
        # Stack different metadata encodings along a pseudo-sequence
    axis
        return torch.stack([
            self.anchor_encoder(anchor_out.float()),
            self.anchor_encoder(anchor_in.float()),
            self.domain_proj(domain_out.float()),
            self.domain_proj(domain_in.float()),
            self.numeric_proj(numerics)
        ], dim=1)  # shape [B, 5, H]
```

Listing 1.3: `MetadataEncoder`: project and pack heterogeneous metadata.

Additive Unified Attention. A Bahdanau-style additive attention lets each metadata token attend over the full textual sequence. Residual connections and LayerNorm stabilize optimization; dropout regularizes the cross-modal match.

```
class AdditiveUniAttention(nn.Module):
    """
    Unified additive attention (Bahdanau-style) that aligns metadata
    tokens
    with the textual representation. Each metadata token queries the
    entire
    text sequence to extract relevant contextual information.
    """
    def __init__(self, meta_h=HIDDEN_DIM, txt_h=BERT_H, attn_h=64):
        super().__init__()
        self.Wq = nn.Linear(meta_h, attn_h, bias=True)   # project
    metadata tokens
```

```
        self.Wk = nn.Linear(txt_h, attn_h, bias=True)    # project textual tokens
        self.v  = nn.Linear(attn_h, 1, bias=True)        # scoring vector
        self.Vv = nn.Linear(txt_h, meta_h, bias=True)    # project values into meta space
        self.norm = nn.LayerNorm(meta_h)
        self.dropout = nn.Dropout(0.1)

    def forward(self, meta_tokens, bert_hidden, bert_attn_mask):
        B, M, H = meta_tokens.size()
        L = bert_hidden.size(1)

        # Linear projections
        Qe = self.Wq(meta_tokens)    # queries from metadata
        Ke = self.Wk(bert_hidden)    # keys from text

        # Compute additive attention scores: v^T tanh(Q + K)
        Qe_exp = Qe.unsqueeze(2).expand(-1, -1, L, -1)  # [B, M, L, A]
        Ke_exp = Ke.unsqueeze(1).expand(-1, M, -1, -1)  # [B, M, L, A]
        e_ij = torch.tanh(Qe_exp + Ke_exp)
        scores = self.v(e_ij).squeeze(-1)               # [B, M, L]

        # Apply mask to ignore padding tokens
        if bert_attn_mask is not None:
            mask = (bert_attn_mask == 1).unsqueeze(1).expand(-1, M, -1)
            scores = scores.masked_fill(~mask, float('-inf'))

        # Normalize into attention weights
        attn = torch.softmax(scores, dim=-1)            # [B, M, L]

        # Weighted sum of textual values projected into metadata space
        Vproj = self.Vv(bert_hidden)                    # [B, L, H]
        context = torch.bmm(attn, Vproj)                # [B, M, H]

        # Residual connection + layer normalization
        out = self.norm(meta_tokens + self.dropout(context))
        return out
```

Listing 1.4: `AdditiveUniAttention`: Bahdanau-style cross-modal alignment.

MultiModalWebClassifier. The final head concatenates the textual [CLS] vector and the metadata vector (mean-pooled over meta tokens) and predicts a binary logit. A masked-mean over domain embeddings avoids bias from padding.

```
class MultiModalWebClassifier(nn.Module):
    """
    Integrates:
      1) BERT text encoding,
      2) metadata tokens (anchors, domains, numerics),
      3) additive cross-modal attention.
```

```
    Outputs a binary relevance logit.
    """
    def __init__(self, encoder: BertModel, hidden_dim=HIDDEN_DIM,
use_lora=False):
        super().__init__()
        self.encoder = encoder
        self.domains_embedding = nn.Embedding(30522, 64)
        self.meta_encoder = MetadataEncoder(hidden_dim)
        self.uni_attn     = AdditiveUniAttention(meta_h=hidden_dim,
txt_h=BERT_H, attn_h=64)
        self.head = nn.Sequential(
            nn.Linear(hidden_dim + BERT_H, 128),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(128, 1)  # binary logit
        )

    def forward(self, input_ids, attention_mask,
                anchor_out_ids, anchor_in_ids,
                anchor_out_mask, anchor_in_mask,
                domains_out_ids, domains_in_ids,
                numerics):

        enc = self.encoder(input_ids=input_ids,
attention_mask=attention_mask)
        last_hidden = enc.last_hidden_state           # [B, L, 768]
        cls = last_hidden[:, 0]                       # [CLS] [B, 768]

        # Masked mean pooling over domain IDs (skip padding)
        def masked_mean(emb, ids):
            mask = (ids != 0).unsqueeze(-1)
            emb = emb * mask
            return emb.sum(1) / mask.sum(1).clamp(min=1e-6)

        dom_out = masked_mean(self.domains_embedding(domains_out_ids),
domains_out_ids)
        dom_in  = masked_mean(self.domains_embedding(domains_in_ids),
domains_in_ids)

        meta_tokens = self.meta_encoder(anchor_out_ids, anchor_in_ids,
dom_out, dom_in, numerics)  # [B, 5, H]
        attn_out = self.uni_attn(meta_tokens, last_hidden,
attention_mask)                          # [B, 5, H]
        meta_vec = attn_out.mean(dim=1)

        logits = self.head(torch.cat([cls, meta_vec],
dim=-1)).squeeze(-1)
        return logits
```

Listing 1.5: with metadata and classify.]`MultiModalWebClassifier`: fuse [CLS] with metadata and classify.

Design notes. The model includes residual connections and LayerNorm inside the cross-modal attention block, a masked-mean for domain embeddings, and a modest MLP head with dropout. These choices proved effective to stabilize gradients and mitigate overfitting under heterogeneous inputs. :contentReference[oaicite:1]index=1

Dataset Creation (Tokenization and Normalization)

The dataset creation script tokenizes clean text and textual metadata while normalizing numeric features. The core tokenization routine is reported below.

```
def tokenize_fn(batch):
    def to_str_list(x):
        return [str(e) if e is not None else "" for e in x]

    text_enc      = tokenizer(to_str_list(batch['text']),
    truncation=True, padding="max_length", max_length=512)
    anchor_out_en =
    tokenizer(to_str_list(batch['outlink_outlink_anchors']),
    truncation=True, padding="max_length", max_length=32)
    anchor_in_enc =
    tokenizer(to_str_list(batch['inlink_inlink_anchors']),
    truncation=True, padding="max_length", max_length=32)
    domain_out_en = tokenizer(to_str_list(batch['domains_out']),
        truncation=True, padding="max_length", max_length=64)
    domain_in_enc = tokenizer(to_str_list(batch['domains_in']),
        truncation=True, padding="max_length", max_length=64)

    return {
        'input_ids':          text_enc['input_ids'],
        'attention_mask':     text_enc['attention_mask'],
        'anchor_out_ids':     anchor_out_en['input_ids'],
        'anchor_out_mask':    anchor_out_en['attention_mask'],
        'anchor_in_ids':      anchor_in_enc['input_ids'],
        'anchor_in_mask':     anchor_in_enc['attention_mask'],
        'domains_out_ids':    domain_out_en['input_ids'],
        'domains_out_mask':   domain_out_en['attention_mask'],
        'domains_in_ids':     domain_in_enc['input_ids'],
        'domains_in_mask':    domain_in_enc['attention_mask'],
    }
```

Listing 1.6: Tokenization function for multimodal inputs.

Training proceeds in two phases. Phase-1 freezes the encoder and trains the fusion head at large batch size; Phase-2 unfreezes the encoder and injects LoRA adapters into self-attention projections for parameter-efficient adaptation. The salient training code is shown next, following `train.py`. :contentReference[oaicite:2]index=2

```
def run_eval_bert(model, val_loader, criterion):
    model.eval()
    val_loss, correct, total = 0.0, 0, 0
    with torch.no_grad():
```

```
        for batch in val_loader:
            outputs = model(**{k: v for k, v in batch.items() if k != "label"})
            loss = criterion(outputs, batch["label"])
            val_loss += loss.item()
            preds = (outputs > 0.0).long()
            correct += (preds == batch["label"].long()).sum().item()
            total   += batch["label"].size(0)
    return (val_loss / max(len(val_loader), 1)), (correct / max(total, 1))
```

Listing 1.7: BERT evaluation loop (loss + accuracy with 0.5 threshold).

```
def train_phase_bert(model, train_loader, val_loader, epochs, lr, max_steps,
                     freeze_encoder=True, use_lora=False):
    # Freeze/unfreeze backbone
    for p in model.encoder.parameters():
        p.requires_grad = not freeze_encoder

    # Optional: add LoRA adapters to the encoder (after freeing)
    if use_lora:
        accelerator.wait_for_everyone()
        model.encoder = apply_lora_to_bert_encoder(model.encoder)
        model = accelerator.prepare(model)

    optimizer = make_optimizer(model, lr=lr)
    optimizer = accelerator.prepare(optimizer)

    best_val = float("inf")
    for epoch in range(epochs):
        model.train()
        running, steps_this_epoch = 0.0, 0

        with tqdm(total=min(max_steps, len(train_loader)),
                  desc=f"Epoch {epoch+1}",
                  disable=not accelerator.is_local_main_process) as pbar:
            for batch in train_loader:
                if steps_this_epoch >= max_steps:
                    break
                optimizer.zero_grad(set_to_none=True)
                outputs = model(**{k: v for k, v in batch.items() if k != "label"})
                loss = criterion(outputs, batch["label"])
                accelerator.backward(loss)
                torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
                optimizer.step()

                running += loss.item()
                steps_this_epoch += 1
                pbar.update(1)
```

```
        val_loss, val_acc = run_eval_bert(model, val_loader, criterion)
        train_loss = running / max(1, min(max_steps, len(train_loader)))

        if accelerator.is_local_main_process:
            wandb.log({"epoch": epoch + 1, "train_loss": train_loss,
                       "val_loss": val_loss, "val_accuracy": val_acc})
            if val_loss < best_val:
                best_val = val_loss
                accelerator.save(model.state_dict(),
"best_model_bert.pt")
```

Listing 1.8: Two-phase BERT training (freeze → unfreeze + LoRA).

Table 1.3: Phase-1: Base Multimodal (frozen encoder).

Tokenizer	`BertTokenizerFast`
Inputs	Text (512), Out-anchors (32), In-anchors (32), Out-domains (64), In-domains (64), Numerics (8 standardized)
Metadata prep	Train-only `StandardScaler` (numerics); anchors/domains cleaned & normalized; domain lists joined with space
Split	Train/Validation = 90/10 (stratified by label, seed=42)
Batch size	256
Optimizer	AdamW (lr = 10^{-3}, weight_decay=0.01)
Loss	`BCEWithLogitsLoss(pos_weight)` (from label distribution)
Freezing policy	Encoder frozen, train metadata fusion head only
Evaluation metric	Validation accuracy (threshold 0.5 on sigmoid)
Max steps	25,000
Infrastructure	`accelerate` (device placement + AMP), W&B logging

Table 1.4: Phase-2: LoRA Fine-Tuning (unfrozen encoder).

Batch size	32
Optimizer	AdamW (lr = 10^{-4}, weight_decay=0.01)
Freezing policy	Encoder unfrozen
LoRA injection	Self-attention projections: Q, K, V in `encoder.layer.*.attention.self`
LoRA config	$r = 8$, $\alpha = 16$, dropout = 0.05, `bias="none"`
Loss / Metric	Same as Phase-1 (`BCEWithLogitsLoss` + validation accuracy)
Max steps	10,000
Checkpointing	Save `best_model_bert.pt` on best validation loss/accuracy

1.2.3 Testing and Inference Implementation

The testing and inference phase is implemented to ensure scalability and reproducibility across both the text-only and the multimodal branches. For binary classification with BERT/DeBERTa, evaluation is embedded in the training script itself, while for the multimodal BERT and text-only score estimation with QualT5, dedicated testing scripts are provided under the `test/` directory. In all cases, pretokenized shards are loaded directly from disk to avoid redundant preprocessing, checkpoints are restored with non-strict weight loading, and batch-level inference is executed with pinned-memory data loaders and large batch sizes to maximize throughput.

Model restoration follows the training setup. In the BERT multimodal branch, the encoder and metadata fusion head are re-instantiated; if LoRA adapters were used during fine-tuning, they are automatically attached to the self-attention projections before loading the state dictionary. In the QualT5 branch, the encoder-decoder is loaded together with its tokenizer to ensure consistency in decoding the target tokens “true” and “false”. All models are placed on the selected device (`cuda` or `cpu`) and run fully in `no_grad` mode.

Scoring logic differs across branches. In the BERT multimodal case, the classifier outputs a single logit per sample; applying the sigmoid yields a probability $\hat{p} = \sigma(\text{logit})$, with hard predictions obtained at a fixed threshold of 0.5. For QualT5 score estimation, only the first decoding step is considered, restricting logits to the tokens corresponding to `"true"` and `"false"`. A local softmax over these two logits yields the probability of `"true"`, ensuring consistency with the generative training objective. In both cases, hard predictions are obtained via a threshold, while soft probabilities are retained for computing ranking metrics and proper scoring rules.

Outputs are exported as CSV files. For text-only score estimation and multimodal inference, minimal tuples such as `clueweb_id, url, score` are produced, while binary classification additionally logs the predicted labels. These outputs can optionally be converted into a compact on-disk *QualCache*, a binary in-

dex mapping document identifiers to float quality scores, consisting of two files: `quality.f4` (dense float array) and `docno.npids` (identifier mapping). This conversion is performed through the `pyterrier_quality` library and is particularly useful for static pruning or crawler frontier prioritization.

The following commands illustrate reproducible runs for the testing phase and optional cache conversion:

```
python3 only_text_testing.py \
  --model pyterrier-quality/qt5-small \
  --test_csv dataset_generation/dataset/dataset_test.csv \
  --max_length 512 \
  --ckpt checkpoints/QualT5_finetuned/final_model \
  --batch_size 256 --device cuda
```

Listing 1.9: Only-text (QualT5) — testing on CSV.

```
python3 metadata_testing.py \
  --model bert-base-uncased \
  --test_dir /mnt/ssd_data/tokenized_bert_test \
  --chunks 126 \
  --ckpt /path/to/best_model_bert.pt \
  --batch_size 128 --num_workers 4 \
  --device cuda
```

Listing 1.10: Multimodal (BERT + metadata) — testing on pre-tokenized shards.

```
python3 qual_cache_build.py \
  --csv /path/to/scores.csv \
  --outdir /path/to/qual_cache_out \
  --chunksize 1000000
# Produces: qual_cache_out/quality.f4  and  qual_cache_out/docno.npids
```

Listing 1.11: Optional: convert scores CSV to an on-disk QualCache.