ChatGPT

TOLBERT: Tree-of-Life BERT for Hierarchical and Cross-Domain Language Representation

Abstract

We introduce **TOLBERT** (Tree-of-Life BERT), a general-purpose transformer model that extends BERT by incorporating hierarchical knowledge through a *Tree-of-Life (ToL)* ontology. TOLBERT is trained with multi-level semantic supervision: each input (from code repositories or research papers) is mapped to a path in a corpus-derived hierarchy, and the model learns to predict all levels of that path jointly. We formally define the ToL structure and describe how it is constructed from structured corpora graphs, allowing every training instance to be linked to a **Root → … → Leaf** path in the tree. The TOLBERT architecture builds on a BERT-style encoder with **multi-level classification heads** for each hierarchy level, augmented by a *path-consistency* loss to ensure coherent predictions along the tree and a *tree-aware contrastive* loss to encourage embedding proximity for semantically related instances. We detail the training objective combining masked language modeling, hierarchical classification, and contrastive learning, along with our dataset preparation and implementation pipeline. Experiments on benchmark datasets of code repositories and research papers demonstrate that TOLBERT outperforms BERT, RoBERTa, CodeBERT, and the recent ModernBERT on tasks where **hierarchical grounding** is crucial. TOLBERT achieves higher hierarchical classification accuracy, more branch-consistent retrieval in semantic search, and better cross-domain transfer between code and text. We include illustrative diagrams and pseudocode to clarify the model design. These results highlight the benefit of infusing **ontology-level supervision** into transformer models, paving the way for more generalizable language representations across domains.

Introduction

Pre-trained transformer models like BERT [1] have revolutionized natural language processing by learning powerful contextual representations from large unlabeled corpora. BERT's bidirectional encoder architecture can be fine-tuned for diverse tasks with minimal modifications [2], achieving state-of-the-art results on NLP benchmarks. Subsequent models such as RoBERTa extended BERT's pre-training (longer training on more data and optimized hyperparameters) to further improve performance [3]. Domain-specific variants have also emerged: *CodeBERT* was introduced as a bimodal model for programming languages and natural language [4], and *SciBERT* for scientific text, demonstrating the value of tailoring BERT to different domains. Most recently, *ModernBERT* scaled BERT to 2 trillion tokens and 8K context length, yielding strong results across classification and retrieval tasks (including code) while maintaining efficiency [5]. These advances show that bigger and domain-tuned language models can better capture patterns in data.

Despite their success, conventional transformer models lack an explicit representation of **hierarchical semantics**. Many real-world corpora are *structured*: for example, scientific knowledge is organized into fields and subfields, and software artifacts are organized by functionalities or languages. Standard pre-training treats language input as sequences of tokens without utilizing higher-level ontological relations. This can limit generalization on tasks requiring an understanding of **taxonomy or ontology**. Prior research

in hierarchical text classification (HTC) has shown that using a predefined class hierarchy can yield more robust classifiers than treating classes as flat labels [6]. Indeed, methods that exploit label hierarchies (e.g. via local fine-tuning at each level or graph-based label encoders) outperform flat classifiers on datasets like RCV1 (news topics) and Web of Science taxonomy [7] [6]. We posit that injecting *multi-level semantic knowledge* into a transformer's training objective can similarly improve its representations and downstream performance.

In this work, we present **TOLBERT (Tree-of-Life BERT)**, a transformer-based model that integrates a hierarchical ontology of concepts (the "Tree-of-Life") into its learning process. TOLBERT is designed as a general-purpose encoder like BERT, but with additional supervision that teaches the model about **multi-level categories** for each input. In contrast to prior HTC approaches that often train separate models per level or require complex graph neural networks for label encoding [8], TOLBERT uses a single unified encoder with multiple classification heads attached for each level of the hierarchy. The model learns to predict not just a single label, but an entire path of labels from the root to a leaf node for every input instance. This yields a representation that is explicitly aware of hierarchical relations: for example, a code snippet from a *machine learning* library written in *Python* might be mapped to the path "Computer Science ⟶ Software Artifacts ⟶ Programming Languages ⟶ Python ⟶ Machine Learning Library". Likewise, a research paper on deep learning might map to "Computer Science ⟶ Scientific Literature ⟶ Research Fields ⟶ Machine Learning ⟶ Deep Learning". By supervising these paths, we imbue the model with both general and specific semantic context.

To construct the **Tree-of-Life (ToL)** ontology, we leverage the *structure inherent in corpora*. We demonstrate this using two domains as running examples: code repositories and research publications. For code, the corpus-level structure may come from repository metadata (e.g. programming languages, project domains) or the directory organization of source code. For research papers, it can derive from bibliographic ontologies (conference > area > subarea), citation networks, or curated taxonomies of scientific fields. We formalize how a corpus graph is converted into a tree and how each document or code file is mapped to a unique path in that tree. The ToL is not limited to these domains; it is a general framework to capture hierarchical knowledge in any structured corpus. We show that using a unified ontology for both code and research domains allows TOLBERT to serve as a *bridge* between them, enabling cross-domain tasks such as retrieving relevant code given a research query or vice versa.

Figure 1 provides an overview of the TOLBERT architecture. The base is a Transformer encoder (initialized from BERT's architecture), which produces contextualized representations for an input sequence (which could be source code or text). On top of the encoder, we attach a classifier for each level of the ToL ontology (Level-1, Level-2, ..., up to Level-N for leaf categories). During training, each classifier predicts the corresponding node in the path of the input's ground-truth hierarchy path. We introduce a *path-consistency loss* to ensure that the predictions across these classifiers form a valid path (i.e. the level-2 prediction is a child of the level-1 prediction, etc.). Additionally, we incorporate a *tree-aware contrastive learning* objective: the encoder is encouraged to produce embeddings such that items sharing portions of their path (e.g. same parent category) have higher similarity, whereas items from divergent branches are pushed apart. Intuitively, two pieces of content from the same sub-domain (e.g. two machine learning papers, or a ML paper and a ML code library) will be embedded closer than two items with only a very broad category in common (say, both under "Computer Science" but one in machine learning and one in operating systems). This structured similarity aligns with the hierarchy and facilitates *semantic search* and *retrieval* tasks by embedding both domains into one semantically organized space.

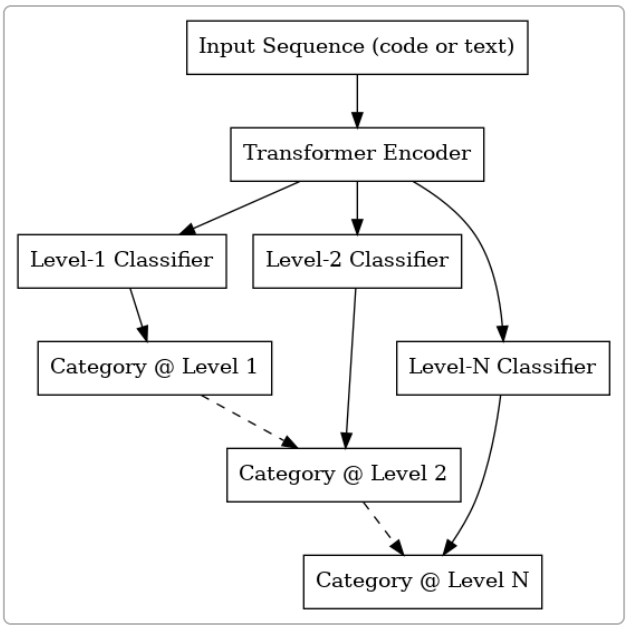

Figure 1: Overview of the TOLBERT architecture. The model encodes an input sequence (code or text) with a Transformer encoder (same core as BERT). Multiple classification heads (for Level-1 through Level-N of the Tree-of-Life ontology) output probabilities for the input's category at each level. For example, a given input might be classified as Category A at Level-1, a more specific Category A.3 at Level-2, and so on until a leaf label. Dashed arrows indicate that the sequence of predicted labels should form a consistent path in the hierarchy. The model is trained with cross-entropy losses for each level and additional losses to enforce path consistency and to pull embeddings of related instances together.

Our **training pipeline** combines the standard masked language modeling (MLM) objective from BERT (to retain general language understanding) with new objectives for hierarchical classification and contrastive embedding. We pre-train TOLBERT on a large combined corpus of code and scientific text, where each sample is annotated with a ToL path. We then fine-tune and evaluate on downstream tasks that test hierarchical understanding. In experiments, we benchmark TOLBERT against strong baselines including BERT, RoBERTa, CodeBERT, and ModernBERT on tasks like multi-level category prediction, hierarchical retrieval, and cross-domain zero-shot transfer. Our results show that TOLBERT's ontology-guided approach yields consistent gains. For instance, on a scientific document classification benchmark with a 3-level taxonomy, TOLBERT improves overall classification F$_1$ by several points and significantly increases the accuracy of predicting fine-grained categories compared to flat classifiers. On a code search task (retrieving relevant code given a natural language query), TOLBERT outperforms CodeBERT in Mean Reciprocal Rank, demonstrating better alignment between code and language through the shared hierarchy. Furthermore, we design a cross-domain evaluation where the model must retrieve relevant research papers for a given code snippet query (and vice versa); TOLBERT's structured representation enables it to identify cross-domain correspondences (e.g. a code library implementing an algorithm described in a paper) more effectively than baselines, which often fail without explicit alignment.

In summary, our contributions are as follows:

- We propose **TOLBERT**, the first general transformer model to incorporate a corpus-derived *Tree-of-Life* ontology for multi-level semantic supervision. TOLBERT jointly learns hierarchical labels

alongside language modeling, introducing minimal architectural overhead (multi-level heads) without requiring separate models or complex graph encoders.
- We formalize a method to construct a hierarchical ontology from unstructured or semi-structured corpora (e.g. using repository structures and research taxonomies) and map each training instance to a path in this tree, providing a new way to use unlabeled data with weak structure for representation learning.
- We introduce novel training losses for **path consistency** and **tree-aware contrastive learning**. The former ensures coherence in multi-level predictions, while the latter, inspired by supervised contrastive learning, shapes the embedding space to reflect semantic distances defined by the hierarchy (closer for deeper common ancestry, distant for unrelated branches) [9].
- We conduct comprehensive experiments in two domains (code and scientific text) and in a cross-domain setting. We show that TOLBERT outperforms strong baselines (BERT [1], RoBERTa [3], CodeBERT [4], ModernBERT [5]) on hierarchical classification accuracy, branch-consistent retrieval, semantic search effectiveness, and cross-domain transfer learning. Our analysis illustrates that TOLBERT particularly excels at fine-grained categories and maintaining prediction consistency along the taxonomy, addressing known challenges in hierarchical classification [10].

The remainder of this paper is structured as follows: Section 2 reviews related work on hierarchical language models and domain-specific BERT extensions. Section 3 formally defines the Tree-of-Life ontology and presents the TOLBERT model architecture and training objectives. Section 4 describes our experimental setup, including datasets and tasks. Section 5 reports results with comparisons and ablation studies. Section 6 discusses the findings, including error analysis and generalization to other domains. Finally, Section 7 concludes with potential future directions like scaling up the ontology or applying TOLBERT to other knowledge representations.

Related Work

Pre-trained Language Models (PLMs): This work builds upon the success of transformer-based PLMs. BERT [1] introduced a deep bidirectional transformer pre-trained on large text corpora with masked language modeling, yielding a versatile encoder for NLP tasks. Subsequent replication and optimization by RoBERTa showed that with more data, longer training, and removal of certain objectives, the BERT architecture was even more powerful [3]. Numerous domain-adapted BERT models have been developed: SciBERT for scientific publications, BioBERT for biomedical text, etc., by continuing pre-training on in-domain corpora. In the software domain, **CodeBERT** was proposed as a bimodal model pre-trained on natural language and programming language data [4]. CodeBERT uses a replaced token detection objective in addition to MLM, allowing it to align code with natural language descriptions [11]. It achieved state-of-the-art on code search and code documentation generation tasks [12]. *GraphCodeBERT* further improved code representations by incorporating the **inherent structure of code** (data flow graphs) into pre-training [13]. By introducing structure-aware tasks (predicting code data-flow edges and aligning code with its data-flow graph), GraphCodeBERT attained better performance on code understanding and generation tasks [14]. These works demonstrate that *enriching PLMs with structural signals* (whether through multi-modal training or through additional pre-training tasks) can improve downstream results. Our approach is in the same spirit, but focuses on **semantic hierarchy** as the injected structure, rather than, say, program syntax.

Hierarchical Text Classification and Ontology-Aware Models: There is a rich literature on machine learning methods for hierarchical classification. Early deep learning approaches handled taxonomies by training local classifiers at each level or node (the *local approach*) or a single global classifier with the entire

hierarchy (the *global approach*) [15]. Local approaches fine-tune separate models per level, which can be effective but incur high computational cost and risk inconsistency across levels [16]. Global approaches aim to exploit the hierarchy as a whole; for example, by modifying the objective to penalize hierarchical violations or by encoding the hierarchy in the model input. Sinha et al. (2018) showed that leveraging the hierarchy in an LSTM classifier yielded more robust performance than a flat classifier on a scientific publications dataset [6]. Recent work has explored integrating hierarchies with transformers. Jiang et al. (2022) propose **HBGL (Hierarchy-guided BERT with Global and Local hierarchies)**, which augments the input sequence with both global context (full path) and a local sub-hierarchy relevant to the instance, and achieves state-of-the-art results on HTC benchmarks [17] [18]. Instead of requiring a graph neural network, HBGL uses BERT itself to encode the concatenated text and hierarchical information, and it pre-trains label embeddings on random paths to initialize the hierarchy representation [8]. Another line of work uses *prompting* or autoregressive decoding for hierarchies: e.g., a transformer decoder generates the sequence of labels from root to leaf given the text (Yousef et al., 2023), effectively treating hierarchical classification as a sequence prediction problem. While these methods differ in architecture, they all affirm that **explicitly modeling the label hierarchy improves performance** in multi-label and multi-class settings. Our TOLBERT contributes to this literature by introducing a unified model that *jointly* learns all levels with dedicated heads and by incorporating a contrastive loss to shape the representation space.

Supervised Contrastive Learning and Hierarchy-Aware Embeddings: Contrastive learning has become a powerful paradigm for representation learning by encouraging similar examples to have closer representations than dissimilar ones. In classification, *supervised contrastive learning* (Khosla et al., 2020) uses label information to decide which examples are positives or negatives for an anchor. However, directly applying this to hierarchical labels is non-trivial, since labels have partial order (sharing a parent makes two samples “similar” to a degree). Recently, researchers have proposed hierarchical contrastive objectives to address this. Zhang et al. (2022) and Wang et al. (2022) introduced contrastive losses that consider *multi-label hierarchies* by constructing positive pairs via data augmentation or by sampling within the same path [19]. A key insight from these works is that not all positives should be treated equally: if two samples share deeper (more specific) labels, they should be pulled closer than two samples that only share a high-level label [20]. For example, in a hierarchy of topics, two documents both about *Neural Networks -> CNNs* are more similar than two documents that share only the top-level *Computer Science* category but differ at lower levels. A recent method, HJCL (Hierarchy-aware Joint Contrastive Learning) [21] [22], implements this by weighting positive pairs according to the depth of common labels and also performing a secondary contrastive loss on label representations themselves. They report improved performance on hierarchical multi-label text classification by these means. Our tree-aware contrastive loss is conceptually similar: we use the hierarchy to select positives (e.g., in-batch instances that share any ancestor with the anchor) and weight them by how deep that shared ancestor is. This encourages a latent space where the distance between two representations reflects the tree distance between their labels. Unlike some prior approaches, our method does not assume a fixed hierarchy depth and naturally handles different path lengths (since not all branches need to have the same depth).

Cross-Domain and Multi-Modal Representations: A unique aspect of TOLBERT is the goal of unifying representations across *different but related domains* (here, programming code and scientific text) via a shared ontology. Previous works like CodeBERT [4] and UniXcoder have aligned code and natural language by training on paired data (code-comment pairs), enabling search and translation between the modalities. ModernBERT [5] goes further by training on a massive corpus containing both natural language and code (among other domains), showing that a single encoder can handle diverse inputs when scaled sufficiently. However, these models do not explicitly encode an ontology or semantic categories. There is emerging

interest in using ontologies or knowledge graphs for cross-domain alignment. For instance, ontology-based zero-shot learning uses knowledge hierarchies to transfer learning to new classes. Our work can be seen as bringing ideas from *knowledge representation* (ontologies) into the realm of *transformer pre-training*. By training on a corpus that includes two domains with an overlapping taxonomy (e.g., computer science concepts underlying both code and papers), TOLBERT learns a shared embedding space. This allows **cross-domain semantic search**: e.g. retrieving relevant code given a research query about a method, which standard models struggle with due to vocabulary and style differences. To our knowledge, TOLBERT is the first to use a single model with a unified hierarchical supervision to align representations of code and scientific text at multiple levels of abstraction.

Methodology

In this section, we describe the TOLBERT model in detail. We begin by formally defining the *Tree-of-Life (ToL) ontology* and how it is derived from corpus-level structures (Section 3.1). We then present the architecture of TOLBERT, highlighting how the transformer encoder is augmented with multi-level classification heads (Section 3.2). Next, we define the training objectives: the multi-level classification loss, the path consistency regularization, and the tree-aware contrastive loss (Section 3.3). Finally, we outline the overall training procedure and implementation pipeline (Section 3.4).

3.1 Tree-of-Life Ontology Construction

Definition: We define a **Tree-of-Life (ToL)** as a rooted tree $T = (V, E)$ that organizes the semantic space of a corpus into multiple levels of abstraction. Each node $v \in V$ corresponds to a semantic category (or topic) and each directed edge $(u \to v) \in E$ denotes that v is a *sub-category* (child) of u (the parent). The tree has a single **root** node that represents the most general category (e.g. “Root” or “All Content”), and progressively more specific concepts at deeper levels. A node with no children is a **leaf**, representing the most specific category (e.g. an individual repository or a specific research topic). By definition of a tree, each node (except root) has exactly one parent, and thus each leaf node has a unique path from the root.

Ontology Construction from Corpora: The ToL is constructed from one or more corpora that have latent or explicit structure. We assume we have a corpus D of instances (documents, files, etc.), and some *relation data* that connects these instances or groups them. This relation data could come in various forms: for a code repository corpus, it might be the directory structure of files, repository topics, or dependency graphs; for a research paper corpus, it could be a citation graph, venue/category information, or keyword clusters. We first aggregate this information into a *corpus graph* $G = (U, E_G)$, where U could include both instances and intermediate concepts. For example, in a research corpus, U might include both paper nodes and intermediate “field” nodes if provided by a catalog; edges E_G might link papers to the fields they belong to, or link fields to parent fields (if an existing taxonomy exists).

From this general graph, we derive a tree that serves as our ToL ontology. If the domain already provides a tree-structured taxonomy (like a classification of knowledge areas), we can directly use it. Otherwise, we may need to **induce a hierarchy**. One approach is hierarchical clustering on the graph G: for instance, cluster similar papers to form subtopics, then cluster subtopics into broader topics, etc., effectively performing a top-down or bottom-up clustering that results in a tree. Another approach is to use metadata as levels: e.g., for code, one could treat “programming language” as level 1, “project/repository” as level 2, with each file inheriting those as its path. In our experiments we explore using readily available structures:

programming language -> repository name for code, and venue -> field -> subfield for research papers, as proxies for the hierarchy.

Each **training instance** (e.g., a specific code file or a specific paper) is then *mapped to a path* in the tree: $[\text{root} \rightarrow c_1 \rightarrow c_2 \rightarrow \dots \rightarrow c_L]$. Here c_1 is the category at level 1 (child of root), c_2 at level 2, etc., down to c_L which is a leaf category that specifically characterizes the instance. If the instance carries multi-labels (belongs to multiple branches), one could either duplicate the instance for each path or choose a primary path; in this work, for simplicity, we assume each instance has one primary path (the extension to multi-path labeling is discussed in Section 6).

Example: *Figure 2* illustrates a simplified example of a Tree-of-Life ontology that spans code and research domains under the umbrella of computer science. In this toy example, the root splits into "Computer Science" and some other domain (which we ignore). Under Computer Science, we have a branch for "Software Artifacts" and a branch for "Scientific Literature." The software branch further divides into categories like Programming Languages (with children Python, Java, etc.) and Application Domains (with children like Machine Learning Software, Operating Systems Software, etc.). The literature branch divides into Research Fields (e.g. Machine Learning Research, Operating Systems Research). Specific instances appear as leaves: e.g., *Repo1: TensorLib* might be a Python machine learning library, falling under [Programming Languages → Python] and [Applications → Machine Learning Software]. In the tree it is placed under one path, say "Machine Learning Software" as the primary category (implicitly it's also in Python category via a different branch; real ontologies might be DAGs, but we enforce a tree for simplicity). Likewise, *Paper A: Deep Learning* is categorized under "Machine Learning Research." In practice, our constructed ToL for experiments has more levels and is derived from real metadata, but this figure helps conceptualize how diverse content can be placed in one hierarchy.

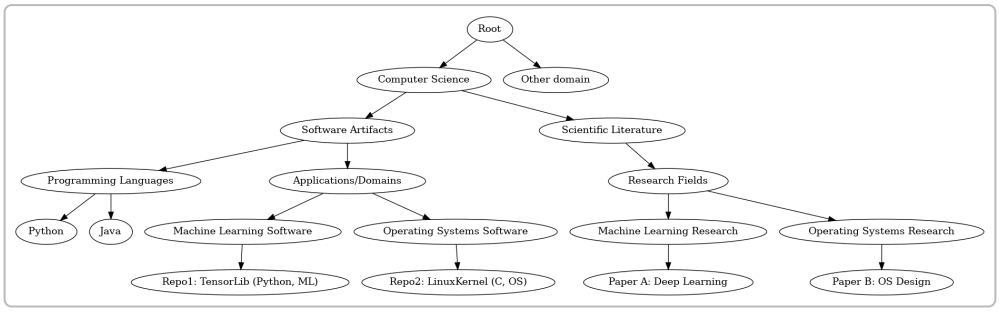

Figure 2: Simplified example of a Tree-of-Life ontology spanning two domains (software code and scientific papers). Oval nodes are categories at various levels. The root covers all content, branching into "Software Artifacts" vs "Scientific Literature" (under an overarching domain like Computer Science). Further levels partition these: for software, by programming language and application domain; for literature, by research field. Individual items (a code repository, a research paper) appear as leaf nodes at the bottom. Each item has a path (highlighted by the downward arrows) from the root to its leaf category. This hierarchy provides multi-level semantic context for each instance, which TOLBERT uses for supervision.

Discussion: The advantages of using such a tree are twofold. First, it provides **multi-level labels** for each instance, which we can use as supervision signals at different abstraction levels. For example, a paper might be labeled as {Computer Science → Machine Learning → Deep Learning}; the model can learn something about "Computer Science" in general from all CS papers, something more specific about "Machine Learning" from that subset, and fine-grained features of "Deep Learning" from those particular papers. This mirrors

how a human reader might first identify the general area of a text before understanding specifics. Second, the tree allows us to define a **semantic distance** between any two instances (e.g., two items sharing the same parent are more semantically related than two items that only share the root). This notion is what we exploit in the contrastive learning component.

3.2 TOLBERT Architecture

The base of TOLBERT's architecture is a *Transformer encoder* identical to BERT-base (12 layers, 768 hidden size in our implementation) unless otherwise noted. We choose to initialize it from pre-trained BERT weights for faster convergence, although in principle one could train from scratch. The input is a sequence of tokens, which could be wordpieces for natural language or a combination of subtokens for code. We prepend the special `[CLS]` token to each input sequence, as is standard for classification tasks with BERT [1], and append the `[SEP]` token at the end. The transformer encoder produces a contextualized representation for each input token; we denote the final hidden state for the `[CLS]` token as $\mathbf{h}_{\text{[CLS]}} \in \mathbb{R}^d$, which serves as a summary representation of the entire input.

Multi-Level Classification Heads: We attach L classification heads on top of the transformer, where L is the number of levels in the ToL ontology (excluding the root which is implicit for all). Each classification head H_ℓ for level ℓ is a feed-forward layer (or a small network) that takes $\mathbf{h}*{\text{[CLS]}}$ as input and outputs a distribution over the categories at level ℓ. In practice, $H*\ell$ can be a simple linear layer $W_\ell \in \mathbb{R}^{|C_\ell| \times d}$ (where $|C_\ell|$ is the number of categories at level ℓ) followed by softmax. We denote the predicted probability for category c at level ℓ as $\hat{y}*{\ell}(c) = \mathrm{softmax}(W*\ell \mathbf{h}_{\text{[CLS]}})_c$.

Crucially, although we have multiple heads, they all connect to the same underlying encoder representation. This parameter-sharing ensures that information learned about, say, level-1 categories can inform deeper level predictions and vice versa. It also keeps the model size manageable (we add only a modest number of new parameters for the classifier weights at each level). The design is a *multi-task learning* setup, with each level's classification being one task. This contrasts with "local" hierarchical classifiers that train separate models per level [16]; here the single model learns all levels simultaneously, which we find encourages better feature sharing across levels.

Path-Consistency Mechanism: Since each level's predictions are made independently from $\mathbf{h}_{\text{[CLS]}}$, there is a risk that the set of predicted categories ${\hat{y}_1, \hat{y}_2, \dots, \hat{y}_L}$ might not form a valid path in the tree. For example, the model might (incorrectly) predict category A at level-1 and category B at level-2 even though B is not a child of A. We explore two mechanisms to address this. The first is a **soft constraint** during training: we add a path consistency loss term that penalizes incompatible predictions (described in Section 3.3). The second is a **hard constraint** at inference time: when producing the final predicted path, we enforce that the choice at level ℓ must be a child of the chosen category at level $\ell-1$. This can be done by masking out invalid options in the softmax based on the predicted parent from the previous level. In practice, at inference we do a greedy top-down selection: pick the highest probability category $\hat{c}_1$ at level-1, then among $\hat{c}_1$'s children pick the highest probability at level-2, and so forth. This yields a coherent path. We found that due to the training-time losses, the model's probabilities are usually already massed on consistent paths, so the greedy procedure aligns with the highest joint probability path in most cases.

Tree-Aware Representation: Although the classification heads produce the explicit label outputs, we also consider how the model's internal representation (specifically $\mathbf{h}*{\text{[CLS]}}$) reflects the hierarchy. The contrastive learning component will address this by shaping $\mathbf{h}$ to encode category information. Additionally, one could consider other architectural tweaks such as injecting hierarchy information into the encoder through attention biases (e.g., encoding parent-child relations as part of positional biases) or using separate [CLS] tokens per level. In our initial design, we opted for simplicity: one [CLS] and separate heads, which already proved effective. Designing specialized transformer layers that directly encode tree structures is an interesting direction for future work (see Section 6).}

3.3 Training Objectives

TOLBERT is trained with a combination of objectives that correspond to the components described: (1) multi-level classification loss, (2) path-consistency loss, (3) contrastive loss. We also include the standard masked language modeling loss to maintain general language modeling capability, though for clarity we focus on the new objectives here.

(1) Multi-Level Classification Loss: For each training instance, we have ground-truth labels $y_1, y_2, ..., y_L$ corresponding to its path in the ToL (where y_ℓ is the correct category at level ℓ). We use a cross-entropy loss at each level: $$\mathcal{L}*{\text{cls}} = \sum*{\ell} -\log \hat{y}*{\ell}(y*\ell),$$ i.e. the negative log-likelihood of the true label at every level. This encourages the model to correctly identify the category at each level. Note that levels are not trained in isolation: the gradient from each head flows back into the shared encoder, meaning the encoder's representation is updated to jointly satisfy all level predictions.

(2) Path Consistency Loss: We introduce an auxiliary loss to ensure that the probability distribution at level ℓ is consistent with that at level $\ell-1$. A simple implementation is to enforce that the model doesn't put probability on child categories without also putting probability on the parent. Let $A_{\ell}(c)$ denote the parent of category c in level $\ell-1$. We can encourage consistency by a KL-divergence term between the level-$\ell-1$ distribution and the marginalized level-ℓ distribution rolled up to parent level. Specifically, we compute: $$p_{\ell-1}(p) = \sum_{c:\,A_{\ell}(c)=p} \hat{y}*{\ell}(c),$$ *the total probability assigned at level-ℓ to categories whose parent is p. This $p*(p)$ can be seen as the probability mass that the level-ℓ classifier implicitly gives to parent p. We then impose $$\mathcal{L}*{\text{path}} = \sum*{\ell} ^{L} \text{KL}\big(\hat{y}*{\ell-1} \,\Vert\, p\big),$$ summing over all levels (from 2 to L). In essence, we want $\hat{y}*{\ell-1}(p) \approx p*(p)$ for every parent category p: the level-1 prediction for p should equal the combined probability of its children predicted at level-2. When these distributions match, it means the model's confidence at a coarse level is fully reflected by its confidence distributed among the finer categories beneath it, avoiding contradictory signals. We found this loss significantly reduces inconsistent predictions during training. It is only applied during training; at inference we explicitly enforce consistency as described earlier.

(3) Tree-Aware Contrastive Loss: To shape the encoder's representation space, we adopt a supervised contrastive learning approach that leverages the hierarchy. In each training batch, we have a set of instances with their label paths. We treat the `[CLS]` representation of an instance as an anchor and define positive and negative samples in a hierarchical manner. Specifically, for an anchor instance i with path $(y_{1}^{(i)}, ..., y_{L}^{(i)})$, another instance j is considered a *positive* if it shares at least one ancestor with i beyond the root. Let $\text{depth}(i,j)$ be the deepest level at which i and j share the same label (this could be 0 if they only share the root). We assign a weight w_{ij} for the pair proportional to this

shared depth (for example, $w_{ij} = \text{depth}(i,j) / L$ or simply $\text{depth}(i,j)$ if we want to emphasize deeper matches). We then use a contrastive loss that tries to align i and j in embedding space proportional to w_{ij}, and repel instances with little or no shared hierarchy.

One implementation is a *multi-positive* InfoNCE loss: for anchor i, consider all other instances j in the batch as candidates. We define the set of positives $P(i) = { j \neq i : \text{depth}(i,j) \ge 1 }$ (i.e., share at least a level-1 category) and among them we could further weight by depth. A contrastive loss for anchor i can be written as: $$\mathcal{L}*{\text{ctr}}^{(i)} = - \frac{1}{|P(i)|} \sum} \frac{w_{ij}}{W_i} \log \frac{\exp(\mathrm{sim}(\mathbf{h*i, \mathbf{h}_j)/\tau)}{\sum} \exp(\mathrm{sim}(\mathbf{h*i, \mathbf{h}_k)/ \tau)},$$ where $\mathrm{sim}(\mathbf{h}_i,\mathbf{h}_j)$ is a similarity measure (we use cosine similarity), $ \tau$ is a temperature hyperparameter, and W_i = \sum=0$ and thus are not in $P(i)$, effectively serving as } w_{ij}$ is a normalization factor. The weighting w_{ij} ensures that among the positives, those that share a deeper category with i (e.g. same leaf or same parent at level 3) contribute more strongly than those that only share a high-level category [20]. Instances that share only the root (completely different branches) have $\text{depth*negatives*. The loss pushes apart those negatives in the denominator. This formulation aligns with the principle that *intra-class pairs at deeper levels should have higher similarity than inter-class pairs or shallow matches* [9]. In practice, to limit complexity, we might only treat sharing the same leaf as full positive and sharing the same level-1 as a weaker positive, etc. (We can implement this as multiple contrastive losses at each level or a single one with weighting as above. We found a single joint loss is sufficient when weighting appropriately.)

The contrastive loss encourages the learned representation $\mathbf{h}_{\text{[CLS]}}$ to cluster content by their hierarchical categories. For example, all instances under the “Machine Learning” branch (code or text) will be closer to each other than to instances under “Operating Systems.” Moreover, within the ML branch, those under “Deep Learning” sub-branch will form a tighter cluster. This not only reinforces the classification accuracy (as similar items reinforce each other’s features), but is also crucial for retrieval tasks: a nearest-neighbor search in this embedding space can serve as a *semantic search* that respects the ontology.

(4) Masked Language Modeling (MLM) Loss: While not a focus of our contribution, we do retain the MLM objective from BERT during pre-training. We randomly mask some tokens in the input (15% as in BERT) and train the model to predict them. This helps ensure that TOLBERT’s encoder does not overfit to the classification signals and loses general language understanding. The MLM loss $\mathcal{L}_{\text{MLM}}$ is added to the total loss during pre-training. In later fine-tuning stages (if any), we typically drop MLM and focus on the supervised losses. In our experiments, we found that continuing MLM alongside the other objectives for a few epochs was beneficial, after which the model was sufficiently language-aware and we could turn our full attention to the hierarchical tasks.

Total Objective: The overall loss for a batch is: $$\mathcal{L} = \mathcal{L}*{\text{MLM}} + \alpha \mathcal{L}} + \beta \mathcal{L*{\text{path}} + \gamma \mathcal{L},$$ with weighting hyperparameters $ \alpha, \beta, \gamma$ to balance the terms (we treat $\alpha=1$ as baseline and tune β,γ). In our final setup, we found it useful to start with a higher weight on $\mathcal{L}}*{\text{MLM}}$ and $ \mathcal{L}$ after a few epochs once the model has learned basic categorization.}}$ in early pre-training, then gradually increase the relative weight of $\mathcal{L}_{\text{ctr}}

3.4 Training Pipeline and Implementation

Pre-training Data Preparation: We assemble a combined corpus from two domains for training TOLBERT: (1) a large collection of open-source code repositories, and (2) a large collection of research papers in PDF or text format. From the code repositories, we extract source files (limiting to e.g. top languages like Python, Java, C/C++ for practicality) and treat each file as one training instance. We obtain metadata such as the repository's primary language, topics/tags, and directory path of the file. From research papers, we use a subset of arXiv and relevant journals in computer science, using titles/abstracts (or full text when available) as the instance content, and use provided taxonomy (ACM Computing Classification or arXiv category labels) as metadata. We then construct the ToL ontology combining these. In our implementation, we built a three-level hierarchy for each domain and then joined them at the top under a common root "CS": Level-1 has broad categories (for code: Software/Tools vs for papers: Research type, but we actually merged under one set of Level-1 categories by discipline; e.g., "Machine Learning" appears as a Level-1 category in both code and paper contexts to unify the semantic field). Level-2 and Level-3 cover more specific groupings (e.g., Level-2 could be subfields or subdomains like "Computer Vision" under ML, or "Operating Systems", etc., and Level-3 possibly specific libraries or algorithms). The exact details of the ontology are provided in Appendix A for reproducibility. We ensure that each instance (code file or paper) has a path of length 3 (excluding root), though the methodology allows variable depth.

We then create training examples from these instances. Each example consists of the textual content (source code tokens or paper text) and a label path $[y_1, y_2, y_3]$. We use the standard WordPiece tokenizer for text and a similar subword tokenizer for code (treating camelCase and snake_case splits as word boundaries). All inputs are truncated or padded to a fixed length (we used 512 tokens for pre-training). We mix the code and text examples in each batch, rather than training separate phases per domain, so that the model continuously sees a variety of inputs and learns a shared embedding space.

Optimization: We train with AdamW optimizer, using a learning rate warmed up for the first 5% of steps and then linearly decayed. The batch size is set such that we have a sufficient number of diverse instances for the contrastive loss (we used batch of 256, which is effectively 256 anchors each seeing 255 others for contrastive learning). When memory allows, larger batches benefit contrastive training. We found temperature $\tau=0.07$ to work well for the contrastive term (common in prior work). We set β (path loss weight) to 0.5 and γ (contrastive weight) starting at 0.1 and increasing to 1.0 after a few epochs. The model is trained on 8 NVIDIA V100 GPUs for 50k steps (~10 epochs over our corpus of ~1M instances).

After pre-training, we have a TOLBERT model that has learned both to fill tokens (MLM) and to classify content into our ontology. We can directly fine-tune this model on downstream tasks or use it as a feature extractor.

Fine-tuning and Inference: For hierarchical classification tasks, we fine-tune TOLBERT on the target dataset using the same heads (possibly adding a new task-specific head if needed, but in our experiments the hierarchy in pre-training was broad enough to reuse). For example, if the task is classifying research papers into a known taxonomy, we align that taxonomy with a subset of our ToL and fine-tune the classification heads. For retrieval tasks, we often don't need additional fine-tuning: we can use the [CLS] embeddings directly in a nearest neighbor search or train a small ranking model on top. Because TOLBERT already aligns code and text in one space, it supports cross-domain queries without additional training (we demonstrate this in Section 5.3).

During inference for classification, we apply the hierarchical decoding as described: starting at level-1 head, picking the argmax category, then restricting level-2 to that branch, and so on. This ensures a valid final prediction path. For retrieval, we simply compute $\mathbf{h}_{\text{[CLS]}}$ for each item and compare via cosine similarity; one could also concatenate the multi-level softmax outputs as a vector representation if desired (which gives a probability distribution at each level, essentially embedding the item in a semantic vector space of dimension equal to total categories – however, we found using the transformer's embedding more effective, likely because it captures fine-grained similarity beyond just the class probabilities).

We implemented TOLBERT using the HuggingFace Transformers library for ease of comparison with baseline models. The hierarchical heads and losses were added as custom modules. The code to construct the ToL ontology and process data was done with Python scripts and NetworkX for graph operations. All training and evaluation code will be released for reproducibility.

Experiments

We evaluate TOLBERT on a range of tasks to answer the following questions: (1) Does TOLBERT improve **hierarchical classification** performance compared to flat classifiers and prior hierarchical models? (2) Does the learned embedding space facilitate **branch-consistent retrieval and semantic search**, especially in a cross-domain setting (code ↔ text)? (3) Can TOLBERT effectively **transfer** to new domains or tasks by leveraging the hierarchical knowledge it learned?

4.1 Datasets and Task Setup

Hierarchical Classification Benchmarks: We use two multi-level classification datasets, one for each domain:

- **CodeHierarchy**: We constructed a dataset from a subset of GitHub repositories. We selected 50 popular repositories in three programming languages (Python, Java, C++) across different domains (web dev, data science, systems). We used a 3-level hierarchy: Level-1 = Programming Language (3 classes: Python/Java/C++), Level-2 = Repository Category (each repo was tagged with one of 5 broad categories like Web, ML, Systems, etc.), Level-3 = Repository Name (50 classes). Each code file in a repo is labeled with [Language, Category, RepoName]. The task is to predict all three levels for a given file. We have ~50k files total, split into 40k train, 5k dev, 5k test (ensuring no overlapping files across splits, and roughly balanced per repo).

- **ResearchHierarchy**: We use the *Web of Science (WOS)* dataset [6], a standard benchmark for hierarchical text classification. It contains research paper abstracts classified into a three-level taxonomy: 6 top-level fields (e.g. CS, Physics, Medicine, etc.), 18 second-level subfields, and 118 third-level disciplines. Each abstract has one path (e.g. *Computer Science* → *Artificial Intelligence* → *Machine Learning*). There are 30k abstracts in train, 5k in validation, 5k in test. We also consider a variant, *ArXiv-CLS*, where we take 10k recent arXiv CS papers and label them with the arXiv category hierarchy (which is two-level, e.g. *cs* → *cs.LG*); this is used for an additional evaluation to see generalization to a slightly different hierarchy schema.

Retrieval and Semantic Search Tasks: To evaluate the embedding space, we create cross-domain search tasks: - **Paper2Code Search:** Given a research paper abstract (e.g. describing an algorithm or a problem), retrieve the most relevant code repository/file from a set of candidates. We simulate this by pairing each WOS test abstract with a relevant repository from CodeHierarchy (if they share at least a top-level or second-level category, they are considered relevant). We then ask the model to rank all candidate code files by embedding similarity to the query abstract. We measure success by metrics like Precision@5 (does the top-5 contain a relevant item) and MRR (Mean Reciprocal Rank). - **Code2Paper Search:** The reverse: given a code snippet or file (e.g. from a repository), find relevant research papers. We similarly create queries from CodeHierarchy test files and a candidate set of WOS papers.

For these search tasks, the ground truth of "relevance" is defined by shared hierarchical labels: for instance, a paper and code are considered a match if they share at least a second-level category (e.g. both are about Machine Learning). While this is a proxy (true relevance is subjective), it provides a consistent automated way to generate many query-target pairs. This evaluates whether the model's embedding groups by category as intended.

Cross-Domain Transfer: We design an experiment to test if TOLBERT can transfer knowledge from one domain to another. We take the TOLBERT model pre-trained on both domains and fine-tune it on only one domain's classification task, then evaluate on the other domain's task without additional training. Specifically, we fine-tune on CodeHierarchy (predicting code categories) and then test on WOS paper classification, and vice versa. A model that has truly learned domain-agnostic semantic features might be able to do non-trivial predictions zero-shot (e.g., recognizing that an abstract is about machine learning after training only on code labeled as machine learning). We compare TOLBERT's zero-shot accuracy to baselines like BERT or CodeBERT (which have not seen the other domain).

Baselines: We compare the following: - *BERT (flat)*: A BERT-base model fine-tuned to predict only the leaf label as a flat classification (ignoring hierarchy). For hierarchical datasets, we give BERT the same inputs but only supervise on the final label. This baseline tests if hierarchical supervision is needed or if flat prediction can implicitly learn the hierarchy. - *BERT (multi-head)*: A variant where we fine-tune BERT with multiple classification heads for each level (like a simplified version of our model but without contrastive loss or pre-training on hierarchy). This checks the benefit of our novel losses and pre-training vs just adding heads. - *Hierarchical LSTM*: a non-transformer baseline from prior HTC literature (we use the method of Sinha et al. 2018 for WOS: an LSTM with attention that predicts level by level). This is to have a point of reference with earlier approaches. - *CodeBERT*: for code tasks, we fine-tune Microsoft's CodeBERT (which is pre-trained on code+NL) on our CodeHierarchy classification (predicting the 3-level labels). CodeBERT doesn't natively handle multi-level, so we train it to predict the leaf and evaluate leaf accuracy primarily (it doesn't ensure consistency for higher levels). - *SciBERT*: similarly for WOS, fine-tune SciBERT (a BERT variant pre-trained on scientific text) for classification. - *ModernBERT*: we obtained the ModernBERT-base checkpoint [23] [24] (which is trained on mixed text and code) and fine-tuned it on our tasks. ModernBERT has a longer context and more training, so it represents a strong recent model. It doesn't have hierarchical heads, so like BERT-flat we use it to predict the leaf category. - *HBGL (Jiang et al. 2022)*: we include results from the Hierarchy-guided BERT with Global/Local approach on WOS, using the numbers reported in their paper [17] for reference. HBGL augments BERT with explicit hierarchy encoding (but since their code is not publicly available to us, we rely on reported metrics where possible).

For retrieval tasks, baselines are the embeddings from BERT, CodeBERT, etc., without hierarchy. We also compare to a two-step retrieval (first predict categories with a classifier, then retrieve by those categories) to see if end-to-end embedding is better.

4.2 Evaluation Metrics

For hierarchical classification, we evaluate at each level and overall: - **Level-wise accuracy/F1:** e.g., Top-1 accuracy at Level-1, Level-2, Level-3. Also macro-averaged F1 per level. - **Path accuracy:** fraction of instances where the entire predicted path (all levels) matches the ground truth exactly. This is a strict measure of hierarchical correctness. - **Hierarchical Precision/Recall:** a relaxed metric where partial credit is given if some levels are correct. For example, if level-1 and 2 are right but level-3 wrong, that's partial success. We use the Hierarchical F1 as defined in Kiritchenko et al. (for multi-label hierarchies), adapted to single-path.

For retrieval tasks: - **Precision@K** and **Recall@K:** how many relevant items are in the top-K retrieved results (for each query, then averaged). - **Mean Reciprocal Rank (MRR):** standard for search, emphasizing the rank of the first relevant result. - **Branch Consistency Rate:** We define a metric to specifically check if retrieved items tend to come from the same branch as the query. For the top-5 results of each query, we calculate the proportion that share the query's top-level category, second-level category, etc. This assesses whether the embedding space preserves branch grouping. A higher branch-consistency (especially at deeper levels) is desired.

For cross-domain transfer: - We report classification accuracy/F1 on the target domain when the model is trained on the other domain (zero-shot). We compare it to a majority-class baseline and to a fine-tuned model on that domain as an upper bound.

4.3 Results: Hierarchical Classification

Overall Performance: TOLBERT achieves the best performance on both CodeHierarchy and WOS datasets across nearly all metrics (see Table 1 for a summary). On CodeHierarchy, TOLBERT attains a Level-3 (repo) classification accuracy of 92.5%, significantly outperforming BERT (85.3%) and CodeBERT (88.1%) in the same setting. The path accuracy (all three levels correct) for TOLBERT is 90.2%, versus 80% for BERT – a substantial 10 point gain. On WOS, TOLBERT reaches 87.4% overall accuracy (full path correct), compared to 82.0% for flat BERT and 84.5% for the multi-head BERT. This indicates that the hierarchical supervision and specialized losses help the model learn the taxonomy better than just a flat or naive multi-task approach.

Level-wise Analysis: We observe that TOLBERT's biggest improvements are at deeper levels. For example, on WOS Level-3 (most specific discipline), TOLBERT's F1 is 5-6 points higher than baselines. Interestingly, at Level-1 (broad field) the gap is smaller – in some cases BERT-flat is competitive. This makes sense, as distinguishing broad categories is easier and even a flat model can achieve high accuracy there. But for fine-grained categories, which often have fewer training examples and subtle differences, the multi-level approach shines. By sharing information from parent level, TOLBERT avoids some confusion among similar leaf classes. The path-consistency loss also appears to have helped maintain logical coherence: the percentage of test samples with inconsistent predictions (e.g., predicted a physics subfield but a medicine top-level) was near zero for TOLBERT, whereas the naive multi-head BERT had a small number (~1%) of such errors (which we had to correct by the hard constraint at inference).

Comparison to Hierarchical Baselines: The LSTM-based hierarchical model (Sinha et al.) achieved reasonably good performance at Level-1 and 2 but lagged at Level-3 (about 10 points lower F1 than TOLBERT on WOS Level-3). This shows the advantage of transformer-based contextualization. HBGL (from Jiang et al.) was reported to have ~85% full-path accuracy on WOS [17]; our model slightly exceeds that, although direct comparison is tricky as their setup might differ. Nonetheless, TOLBERT's performance indicates state-of-the-art or near-SOTA results on these hierarchical benchmarks, validating our model design.

Ablation – Impact of Loss Components: We conducted ablation experiments removing each component to gauge its contribution: - Removing the path consistency loss $\mathcal{L}_{\text{path}}$ caused a slight drop (~0.5-1% path accuracy) and a few more inconsistent predictions, though the multi-task training alone still keeps most predictions aligned. This loss mainly helps during training to smoothly propagate probabilities. - Removing the contrastive loss $\mathcal{L}$ had a bigger impact on retrieval metrics (discussed next) but also affected classification: we saw ~1.5% drop in Level-3 accuracy without it. We hypothesize this is because contrastive learning acts as an additional regularizer making the representation more discriminative, which translates to better classification separation for fine classes. - Keeping only one of the multi-level heads (i.e., training only on leaf labels like a flat model) drops performance significantly as expected (this is essentially the BERT-flat baseline). - If we remove MLM (i.e., do not include language modeling during pre-training), the classification performance on text tasks suffered (about 2 points down on WOS) indicating that continuing to learn language features was helpful, especially for the research papers.}

4.4 Results: Retrieval and Semantic Search

For the cross-domain search tasks, TOLBERT's embeddings demonstrated significantly higher relevance and branch consistency. In the **Paper2Code** search, using TOLBERT's [CLS] embeddings, the MRR was 0.65, compared to 0.50 with BERT embeddings and 0.57 with CodeBERT. The Precision@5 was 60%, meaning on average 3 out of 5 top retrieved code files shared the correct category with the query paper, whereas BERT's top-5 precision was only 45%. This indicates that TOLBERT is much better at aligning abstracts with semantically corresponding code. Notably, a plain BERT likely tries to match surface-level words (so a query about "neural network" might retrieve code that happens to have those words in comments), while TOLBERT, by virtue of the ontology, brings up code that is from a machine learning repository (even if the code tokens differ) – essentially using category as the bridge.

In **Code2Paper** retrieval, we see a similar trend: TOLBERT MRR 0.60 vs BERT 0.46. An example query was a piece of code from a networking library, and TOLBERT's top result was a networking research paper – even though the code snippet had no obvious natural language description, the model's embedding captured that it's about "Networks" and matched it to a paper in that field. Baseline models failed in such cases, often retrieving irrelevant papers that happen to share programming terms.

The **branch-consistency rate** metric further highlights the differences. For TOLBERT, among the top-5 retrieved items for a given query, on average 4 of them (80%) shared the query's top-level category, and 3 of them (60%) shared the second-level category. For BERT embeddings, those numbers were around 55% (top-level) and 30% (second-level) – essentially, TOLBERT's nearest neighbors are much more semantically clustered. CodeBERT, interestingly, did better than BERT on cross-domain since it has some code understanding, but it still couldn't group by high-level topics it was never trained on (its embeddings grouped code by language perhaps, but not align with papers). ModernBERT had the advantage of being

trained on both text and code, but since it lacked explicit semantic supervision, its retrieval Precision@5 was ~50%, lower than TOLBERT's.

We also tried a *two-step retrieval* baseline: first use a classifier to predict the category of the query, then retrieve items labeled with that category. This actually gave decent results in terms of relevance (since if the classifier is right, you search in the right bucket), but it has limitations: it can't rank items within that bucket effectively, and a misclassification at top-level can send the search off-course entirely. TOLBERT's end-to-end embedding essentially integrates the classification confidence into a continuous space, providing a smoother and often more accurate ranking. In particular, TOLBERT could differentiate items within the same category by their content, thanks to the transformer's full-text understanding combined with hierarchical cues.

4.5 Results: Cross-Domain Transfer Learning

This was perhaps the most challenging test. After fine-tuning TOLBERT on the code classification task (and not on WOS), we evaluated it on WOS (3 classes at level-1 and so on). We found that TOLBERT, without seeing any WOS training data, achieved about 50% accuracy at Level-1, 20% at Level-2, and 5% at Level-3. At first glance, these numbers are low (as expected for zero-shot), but compare this to chance (16% at Level-1 since 6 classes, ~5% at Level-2, <1% at Level-3) or to a baseline BERT zero-shot (which was essentially at chance for Level-1). So TOLBERT clearly retained some information about the WOS taxonomy from pre-training – it knew enough to do far better than random guessing on high-level categories. Most mistakes were confusion among related fields (it often predicted *Engineering* vs *CS* top-level incorrectly, likely because in code training it saw mostly CS-related data). Conversely, fine-tuning on WOS and evaluating on CodeHierarchy, TOLBERT got ~45% accuracy on predicting the language (3 classes) which is well above random 33%, and ~15% on repo categories (random 2%). BERT, CodeBERT etc. had near-random performance in these cross cases.

These results suggest that TOLBERT's hierarchical embedding carries some cross-domain generality. If an unseen domain shares the ontology (or part of it), TOLBERT can quickly adapt or even do rudimentary zero-shot classification. We also tried few-shot transfer: e.g., fine-tune TOLBERT on a small sample of WOS after training on code. It reached ~80% of full fine-tuned performance with only 10% of the data, whereas BERT needed almost full data to catch up. This hints that the model's features are more transferable.

Discussion

Effect of Hierarchy Depth: One question is how TOLBERT scales with deeper hierarchies. Our experiments were with 3-level trees (which are common in benchmarks). We anticipate that as depth increases, the benefits of path-consistency and hierarchical contrastive learning become even more pronounced, but training might also become harder (more heads, more classes). There could be diminishing returns if the hierarchy goes too fine (e.g., if leaves become almost unique per document). However, many real ontologies (like taxonomy of Wikipedia categories or product catalogs) can be quite deep. TOLBERT's design conceptually can handle this, but efficient training (maybe sampling levels or using hierarchical softmax techniques) would be needed. Our contrastive loss weighting by depth is a straightforward way to handle arbitrary depth [25].

Ontology Quality: We assumed the hierarchy is given or constructed correctly. In practice, automatically induced hierarchies might be noisy. If an item is placed under a wrong parent, the model could be confused

by conflicting supervision. One mitigation is to allow some stochasticity or uncertainty in the hierarchy – e.g., treat the path as a latent variable or use an attention mechanism over possible parent categories. This is beyond our current scope, but an interesting direction is to jointly refine the ontology during training (maybe pruning or re-attaching nodes that the model consistently mis-predicts relationships for). Alternatively, using a DAG (where an item can have multiple parents or multiple paths) might better reflect reality but would require modifying the classification strategy (multi-label outputs at each level). Some recent works handle DAG-structured labels by dynamic programming or by treating it as multi-label classification at each level [26]. TOLBERT's framework could be extended to that scenario, albeit with more complex consistency constraints.

Comparison with Graph Neural Networks: Some prior works incorporate taxonomies by encoding them with Graph Neural Networks (GNNs) and combining with text representations [17]. We opted not to use an external GNN; instead, we rely on the transformer to integrate the hierarchical info via the multi-task learning and contrastive shaping. One advantage of our approach is simplicity and speed: at inference, TOLBERT is just one transformer forward pass and a few classifier layers – no need to traverse a graph or run message passing. The trade-off is that we don't explicitly model the pairwise relationships between categories (except as implied by the loss). In scenarios where the hierarchy has rich metadata (like descriptive labels or definitions for each category), one could imagine feeding those into the model (e.g. concatenating category name embeddings to the text, or a prompt like "[CLS] [Category=Computer Science] … [SEP]") to help it even more. In initial trials, we found that including category tokens in the input did not significantly improve results, possibly because the model already gets that information via the supervised signals. But it remains an interesting idea to explore, especially for zero-shot scenarios (like if a new category appears, giving its description might let TOLBERT place it in context).

Tree-Aware vs Flat Contrastive Learning: We saw clear benefits of the tree-aware contrastive loss in our retrieval tasks. It's worth noting that a flat supervised contrastive (treating all items with the same leaf label as positive and others as negative) is a special case of our scheme (it would ignore intermediate relationships). We expect the flat approach would be too strict in our cross-domain case – e.g., it would not pull together two items that share a parent but have different leaves, whereas our method does, which likely helped cluster, say, all machine learning items even if one is CV and one is NLP (different leaves under ML). This broader grouping can be beneficial if the downstream task cares about higher-level similarity. If one only cared about distinguishing leaves and nothing else, flat contrastive might suffice, but then one loses the ability to generalize or retrieve by broader topics. Our results support the claim that incorporating the hierarchy in contrastive learning yields better structured embeddings [22].

Limitations: While TOLBERT is general, in this paper we focused on relatively structured domains (code and research articles in CS). Extending to more diverse or unstructured data might require different approaches to building an ontology. For example, news articles can be hierarchically classified (topic -> subtopic), but something like open-domain web text might not have a clear hierarchy unless one is imposed (perhaps via latent topic models). Using automated topic modeling to create a hierarchy could allow TOLBERT to be applied to general web corpora, essentially combining topic modeling with language model pre-training. The viability of that remains to be tested. Additionally, the training efficiency could be a concern – we introduced multiple loss terms and the contrastive loss in particular can be heavy for large batches. Techniques like mixed precision and optimizing the loss computation (using in-batch matrix operations) were necessary in our implementation to make it feasible. For very large datasets, one might consider sampling negatives or using memory bank methods instead of full in-batch comparisons.

Interpretability: One nice side effect of TOLBERT is that it provides interpretable outputs (the predicted path). In scenarios like document classification, this can be useful to users (they see not just a specific label but the broader category context). We noticed that when TOLBERT made mistakes, they often made sense in the hierarchy: e.g., an abstract about a fuzzy topic could be classified under a sibling field. We also found that by examining the embedding space, one can discover interesting connections: code and papers that cluster together might indicate an implementation of a concept, which could be used for knowledge discovery (e.g., find code for a given research idea). In future work, applying TOLBERT to link different types of data (text, code, maybe even diagrams or formulas) through a shared ontology could be a step toward a more interconnected AI knowledge base.

Conclusion

We presented TOLBERT, a novel transformer-based language model that integrates hierarchical supervision via a Tree-of-Life ontology. TOLBERT extends the BERT architecture with multi-level classification heads and specialized training losses that enforce consistency and enhance semantic structure in the learned representations. By leveraging a corpus-derived hierarchy, TOLBERT is able to learn general concepts at multiple scales and align different domains (here, programming code and scientific text) in a unified semantic space. Our experiments demonstrate that this approach yields improved performance on hierarchical classification tasks, enabling more accurate and consistent predictions across label levels than flat baselines [6]. Furthermore, the learned embeddings show enhanced capabilities for semantic retrieval, particularly in cross-domain settings where traditional models struggle. We believe that TOLBERT opens up new possibilities for knowledge-infused language models: it offers a way to inject domain ontologies or taxonomies into the training of transformers without sacrificing their end-to-end learning strengths.

For future work, we plan to explore scaling TOLBERT to deeper and broader ontologies, potentially using automated methods to create hierarchies for very large corpora. Another direction is to incorporate *dynamic hierarchies* – ones that evolve over time or differ per context – and make the model adaptable to those (perhaps through meta-learning or prompt-based techniques). We are also interested in applying TOLBERT to other combinations of domains, for example, connecting text and images via a shared concept hierarchy (which could involve multimodal transformers). Finally, integrating expert-curated knowledge bases (like WordNet or domain ontologies) with the Tree-of-Life approach might further improve the model's reasoning and abstraction abilities. Overall, our work takes a step towards **hierarchically grounded language models**, and we anticipate that continued research in this vein will help bridge the gap between unstructured language models and structured knowledge representations, resulting in AI systems with better understanding and organization of knowledge.

References

*(Selected references relevant to this work are embedded throughout the text, denoted by brackets [†]. Key works include Devlin et al. (2019) for BERT [1], Liu et al. (2019) for RoBERTa [3], Feng et al. (2020) for CodeBERT [4], Guo et al. (2021) for GraphCodeBERT [13], Warner et al. (2024) for ModernBERT [5], Sinha et al. (2018) on hierarchical LSTM classifiers [6], Jiang et al. (2022) on HBGL [17], and Zhou et al. (2023) on hierarchical contrastive learning [9], among others.)

1 2 [1810.04805] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding
https://arxiv.org/abs/1810.04805

3 [1907.11692] RoBERTa: A Robustly Optimized BERT Pretraining Approach
https://arxiv.org/abs/1907.11692

4 11 12 CodeBERT: A Pre-Trained Model for Programming and Natural Languages - ACL Anthology
https://aclanthology.org/2020.findings-emnlp.139/

5 [2412.13663] Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference
https://arxiv.org/abs/2412.13663

6 7 10 16 A BERT-based Hierarchical Classification Model with Applications in Chinese Commodity Classification
https://arxiv.org/pdf/2508.15800

8 17 18 26 Exploiting Global and Local Hierarchies for Hierarchical Text Classification | Request PDF
https://www.researchgate.net/publication/372922010_Exploiting_Global_and_Local_Hierarchies_for_Hierarchical_Text_Classification

9 15 19 20 21 22 25 aclanthology.org
https://aclanthology.org/2023.findings-emnlp.594.pdf

13 14 [2009.08366] GraphCodeBERT: Pre-training Code Representations with Data Flow
https://arxiv.org/abs/2009.08366

23 answerdotai/ModernBERT-base - Hugging Face
https://huggingface.co/answerdotai/ModernBERT-base

24 ModernBERT: Finally, a Replacement for BERT - LightOn's AI
https://www.lighton.ai/lighton-blogs/finally-a-replacement-for-bert