Master's Thesis

Natural Entity Typing in Wikidata

Johannes Mannhardt

Examiner:	Prof. Dr. Hannah Bast
2nd Examiner:	Prof. Dr. Abhinav Valada
Adviser:	Natalie Prange

University of Freiburg

Faculty of Engineering

Department of Computer Science

Chair for Algorithms and Data Structures

February 10thth, 2025

Writing Period

08. 08. 2024 – 10. 02. 2025

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Prof. Dr. Abhinav Valada

Adviser

Natalie Prange

Declaration

I hereby declare that I am the sole author and composer of this thesis and that no sources or aids other than those listed have been used. Furthermore, I confirm that I have acknowledged the work of others by providing detailed references to their contributions.

I also declare that this thesis has not been submitted, either wholly or in part, for any other examination or assignment.

Place, Date:

Freiburg im Breisgau,
09.02.2025

Signature:

Abstract

Wikidata is a massive knowledge graph with millions of entities, each associated with multiple types through the properties instance of (P31) and subclass of (P279). However, it lacks a single, intuitive type for each entity (e.g., "Einstein is a person", "Germany is a country", "Mona Lisa is a painting"). To address this gap, we propose a supervised machine learning approach to automate the process of assigning types. Our method first pre-selects valid type candidates from existing properties, then leverages large language models (LLMs) to generate large-scale training data. Various machine learning models, including a Feedforward Neural Network, GraphSAGE, and R-GCN, are used for type prediction. A masking strategy ensures predictions remain within pre-selected types. Results on two human-annotated benchmarks demonstrate the effectiveness of our method in determining an entity's most natural type, achieving up to 86% top-1 accuracy.

Contents

List of Figures

List of Tables

1. Introduction

In the digital age, the ability to organize and structure knowledge efficiently has become increasingly important. With the advent of the World Wide Web, an innovation that sparked exponential growth of digital data, the need for structured information has become even more evident.

The Semantic Web [1], a concept introduced by Tim Berners-Lee in 2001, envisioned the creation of a web of structured, machine-readable information. This vision has played a key role in numerous Natural Language Processing (NLP) applications, including Named Entity Recognition, text summarization, search engines, and question answering. Consequently, the demand for structured information has driven the development of large-scale collaborative knowledge bases, such as DBpedia (2007), Freebase (2007; now inactive), and most prominently, Wikidata (2012).

Launched in 2012, Wikidata has evolved into one of the largest open knowledge bases, now containing over 109 million entities. It represents knowledge using the *Resource Description Framework* (RDF) [2] as *Subject-Predicate-Object* triples, which are referred to as *statements* in Wikidata. To construct these statements, Wikidata employs over 12,000 properties, such as *date of birth* (`P569`), *gender* (`P21`), *given name* (`P735`), and many more. Among these, two properties are particularly important for Wikidata's knowledge organisation: *instance of* (`P31`) and *subclass of* (`P279`). These properties establish a hierarchical classification where entities are either instances of other entities (`P31`), subclasses of other entities (`P279`), or both. This hierarchical structure enables

efficient navigation and querying of the knowledge base, for example through SPARQL [3] queries, and facilitates the inference of relationships between entities.

Despite its rich coverage of all kinds of topics, Wikidata faces a persistent challenge: it lacks a straightforward, single *natural type* for each entity. In this thesis, we define a natural type as the most intuitive and broadly applicable category a typical person would use to classify an entity in everyday language. For example, *Einstein* is a *person*, *Germany* is a *country*, and the *Mona Lisa* is a *painting*.

Existing properties do not sufficiently address this issue. `P31` and `P279` are intentionally flexible. While this simplifies the task of adding new entities to Wikidata in a collaborative way, it leads to inconsistent and overly broad/specific types. Some classes are so broad and abstract (*body of water*, *living thing*) that they provide little practical utility, while others are highly specific (*small municipality in Germany*, *civil parish in Ireland*), which fragments similar entities into countless narrow subcategories. Defining and assigning a natural type - a type most intuitively recognizable by humans - is an open problem this thesis aims to address.

This task is not only difficult due to the sheer size of Wikidata, but also because of the inherent complexity of categorizing real-world entities. Philosophers like Wittgenstein and Rosch have pointed out that human categorization is flexible, and often based on overlapping similarities and typical examples (prototypes) rather than strict rules. Wittgenstein's concept of *family resemblance* [4] and Rosch's *prototype theory* [5] show that there is not always a single, universally "correct" category - or, in our case, type - since categorical boundaries can't be strictly defined in the real world. Instead, human intuition allows for some uncertainty, which makes it difficult to formalize the notion of what constitutes a natural type.

Rather than searching for a single, context-free "correct" type, this thesis aims to find the classification that best matches human intuition. Having such types would greatly enhance the practical usefulness of Wikidata.

1.1. Motivation

Many NLP tasks depend on natural, intuitive labeling of entities. This would allow for better information extraction/summarization, text categorization, and question answering. For instance, consider a text summarization system processing news articles. If it recognizes *Berlin* merely as a *location* or a *place*, and not as a *city*, it might miss the core context of an article. Similarly, a question like "What *city* is the capital of *Germany*?" can be handled more effectively if the underlying knowledge base clearly labels *Berlin* as a *city*, and *Germany* as a *country* (instead of a *geographical location*).

Likewise, search engines can improve their results with precise entity types. Querying Google often results in a "knowledge panel" on the right side of the search results, including, among other things, the type of the entity that was searched for.

Now consider a scenario where you need all Wikidata entities that qualify as *churches*. You'd want to include not only *cathedrals*, but also *chapels*, *basilicas*, and other types of *churches*. The current state of Wikidata doesn't allow for this. Overly specific types fragment the category, and overly general ones offer little differentiation, so there is a gap between user expectations ("all *churches*") and the actual structure of Wikidata. The lack of natural types significantly reduces Wikidata's potential for many applications.

1.2. Problem Statement

Wikidata's flexible class structure is both a strength and a weakness. In Wikidata, every entity can act as a class for any other entity. While this allows to add new entities in a flexible way, it also leads to incredibly diverse and nested class hierarchies. Conventions exist for common entities, for example, all humans are instances of *human* (`Q5`) and all scientific articles are instances of *scholarly article* (`Q13442814`). However, these conventions are not enforced, and they do not exist for all types of entities.

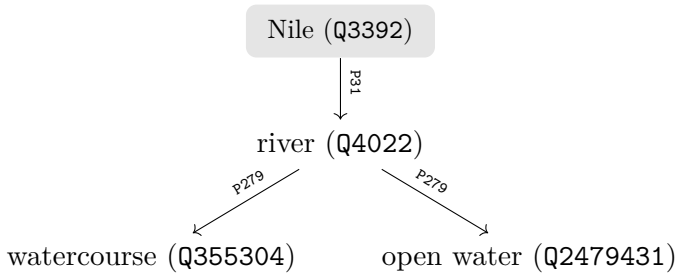

Figure 1.: Ontological Structure of Nile (Q3392). Nile is directly classified as *river* via P31.

Contributors have different perspectives and different priorities regarding how to connect new entities, so inconsistencies are bound to happen. The task of categorizing real-world entities is inherently complex, and discrepancies naturally arise when adding new entities, even with expert contributors. This makes the task of assigning consistent natural types challenging.

The concept of a natural type is central to this thesis. It should align with human intuition and reflect how humans naturally group the world around them. But types should also be useful for practical applications. Overly broad types such as *entity* offer little information. On the other end, low-level types such as *church in Berlin*, *tall building*, or *waterfall in Africa* are too narrow for most tasks.

Consider *Nile* (Q3392). Figure 1 shows its ontological structure. The nodes represent entities, and the edges indicate relationships. In particular, an edge from *Nile* to *river* indicates that there is a statement (triple) where the starting node is the subject (*Nile*), the edge label is the predicate (P31), and the target node is the object (*river*).

As shown in the figure, *Nile* is directly classified as *river* (Q4022) through P31, which represents its natural type. Here, there is no need to traverse a chain of subclasses to find this fundamental classification.

Now, consider *Saturn* (Q193): Many applications simply need to know that *Saturn* (Q193) is a *planet* (Q634). However, in Wikidata, this seemingly basic fact is not directly stated. Instead, more specific types are directly assigned via P31, and the natural type *planet*

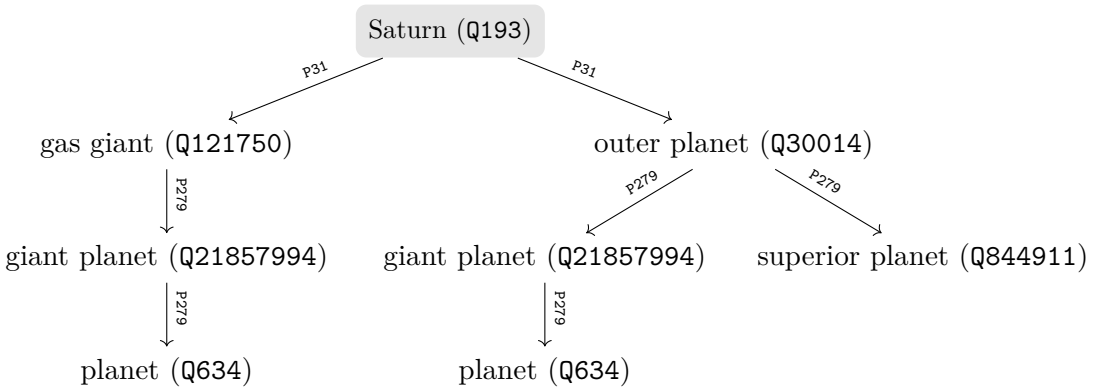

Figure 2.: Ontological Structure of Saturn (Q193). The natural type *planet* can only be inferred indirectly via P279 relations.

can only be inferred by traversing several connections. Specifically, *Saturn* is linked to *planet* (Q634) indirectly through P279 connections via *gas giant* (Q121750) and *outer planet* (Q30014), as shown in Figure 2. Programmatically identifying *planet* as the most fitting natural type presents a significant challenge.

The complexity increases even further when entities have multiple *instance of* relationships, like *London* (Q84), shown in Figure 3. *London* (Q84) is an instance of several types, including *metropolis* (Q200250), *financial centre* (Q1066984), *city* (Q515), and *megacity* (Q174844) among others. While *city* (Q515) may be the most appropriate natural type, the presence of other, more or less specific types highlights the challenge in automatically determining the most natural type.

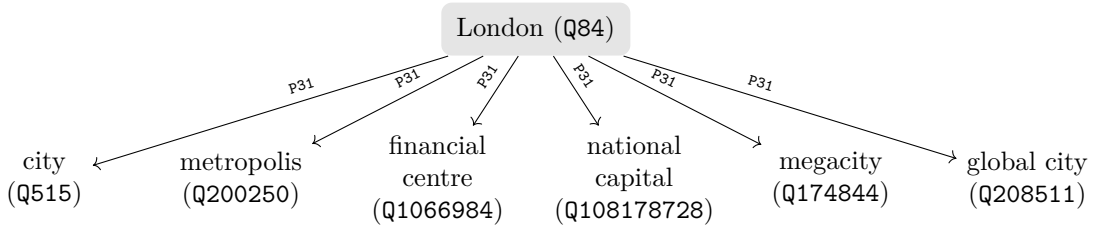

Figure 3.: Ontological Structure of London (Q84). London is associated with multiple instance-of relationships, which complicates determining its natural type.

Furthermore, the natural type of an entity can be context-dependent. *Mount St. Helens* (`Q4675`) is an instance of both *mountain* (`Q8502`) and *stratovolcano* (`Q169358`). Its natural type could be *mountain* (`Q8502`) in certain contexts, and *volcano* (`Q8072`) (superclass of *stratovolcano* (`Q169358`)) in others. This thesis does not consider context-dependent type assignment, so the result must be the most generally applicable type given the information available. In this example, *volcano* would probably be what most people immediately think of when hearing *Mount St. Helens*.

The examples show something important: there is no single, universally applicable rule by which the natural type of a Wikidata entity can be determined. The structure of relationships and the distribution of properties vary drastically across the knowledge graph. One might initially consider simple heuristics such as "the class with the highest in-degree across all entities is the natural type" or "the most common class across all entities is the natural type". However, counterexamples can always be found. The sheer diversity in Wikidata prevents a simple, rule-based approach. Similarly, manually assigning a natural type to each Wikidata entity is infeasible due to the immense amount of data. Therefore, an automated approach capable of inferring the most natural type based on existing properties and relationships is needed.

This thesis introduces a novel approach to determining the natural type of each Wikidata entity.

1.3. Outline

Chapter 2 provides the basic knowledge necessary to understand this thesis. In chapter 3, existing approaches to the problem are shown and discussed in their relevance to our task. Then, in chapter 4, the data and methods used in this work are presented in detail. The results of our approach are evaluated in chapter 5. Finally, chapter 6 highlights the limitations of our approach as well as possible directions for future research.

2. Background

This section provides a high-level overview of some of the concepts and technologies that are relevant to understanding this thesis.

2.1. Semantic Data and Ontologies

The Semantic Web, introduced by Tim Berners-Lee in 2001 [1], extends the traditional web by structuring data so that computers can interpret and process it. At its core, the Semantic Web relies on standards such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL) to represent data in a machine-understandable way.

RDF, a model for representing information on the web, was recommended by the World Wide Web Consortium (W3C) in 1999 [2]. It represents knowledge as a collection of triples, consisting of a subject, a predicate, and an object. These triples can be visualized as graphs, where nodes represent entities and edges correspond to properties that define relationships between them. By using standardized vocabularies, RDF provides a framework that allows data to be defined and linked in a consistent way.

Key RDF elements include `rdf:type`, which assigns a class to a resource (an entity), and `rdfs:subClassOf`, which defines hierarchical relationships between classes.

OWL is a more expressive language that enables the creation of new classes and properties, allowing for complex reasoning and inference tasks.

Several ontologies have been developed to provide a standardized vocabulary for various domains. Early efforts like WordNet organized words into synsets (sets of synonyms) and established lexical relationships, while later projects such as DBpedia, Freebase, and Schema.org aimed to represent real-world entities with varying degrees of strictness and flexibility. These projects represent different approaches to ontology design. While DBpedia has a fixed set of 800 classes and Schema.org follows a structured vocabulary, Wikidata, using `P31` and `P279`, offers a dynamic and continuously evolving classification system.

This enables users to create new classes with minimal restrictions, as there are no enforced constraints on how entities are classified. While this flexibility makes Wikidata highly adaptable and suitable for representing evolving knowledge across various domains, it also introduces challenges such as overlapping classes and inconsistent granularity in classifications.

2.2. Wikidata

Wikidata began in 2012 as a Wikimedia Germany project, aiming to create a publicly available, collaboratively built knowledge base. Unlike Wikipedia, which stores information in articles, Wikidata stores information in structured form matching the RDF model [6]. This structured form allows for easy querying and linking of data, for example through SPARQL queries [3].

We will mostly follow the officially outlined terminology [7], with one exception: In Wikidata, everything is considered an entity. There are six different kinds of entities, the most common ones are items (QIDs), properties (PIDs), and lexemes (LIDs). For typing, our work focuses exclusively on items, i.e. entities that have a QID and are connected through the `P31` and/or `P279` properties. Since *Entity Typing* is the standard term, we will use it throughout this work. For consistency, we will also refer to Wikidata items as entities, except in this section.

Items represent all kinds of things in the world, and they are clearly defined by a `QID`. Some examples are *Universe* (`Q1`), *Human* (`Q5`), *Chair* (`Q146`), *Eiffel Tower* (`Q243`), *Moon landing* (`Q495307`), *Attention Is All You Need* (`Q30249683`), etc. Wikidata also defines properties to capture relationships among items. Properties, uniquely defined by a `PID`, describe how one item relates to another [8]. Examples include properties like *instance of* (`P31`), *subclass of* (`P279`), *sex or gender* (`P21`), *place of birth* (`P19`), and *part of* (`P361`).

Together, items and properties form statements, represented as triples of the form subject, predicate, object (as in RDF).

Some examples of statements are:

- *Douglas Adams* (`Q42`) – *instance of* (`P31`) – *human* (`Q5`)
- *Reinhold Messner* (`Q189307`) – *place of birth* (`P19`) – *Brixen* (`Q185541`)
- *The Lord of the Rings* (`Q15228`) – *genre* (`P136`) – *fantasy* (`Q132311`)
- *iPhone* (`Q2766`) – *developer* (`P178`) – *Apple* (`Q312`)
- *Europe* (`Q46`) – *part of* (`P361`) – *Eurasia* (`Q5401`)

Wikidata imposes no strict ontology. While constraints exist, violating them results in only a warning [9]. For example, it would be possible to add e.g. *the Universe* (`Q1`) as the *head of government* (`P6`) of another entity. Collaborators are highly encouraged, however, to follow the given constraints and best practices.

2.2.1. Basic Membership Properties (P31, P279)

For our work, two properties are especially relevant:

- `P31` **(instance of)** roughly corresponds to `rdf:type`, indicating that an entity is a member of a certain class.

- `P279` **(subclass of)** corresponds to `rdfs:subClassOf`, indicating hierarchical relationships between classes. This property is transitive: If `A` is a subclass of `B` and `B` is a subclass of `C`, then `A` is a subclass of `C`.

Because of the transitivity of `P279`, every instance of a subclass is also an instance of all its superclasses 4.1.1. If `A` is an instance of `B` and `B` is a subclass of `C`, then `A` is also an instance of `C`. This transitivity is key when identifying the *natural type* of an entity. Multiple layers of subclass relationships may stand between an entity and its natural type.

This transitivity means entities like *Moon* (`Q405`) can have many valid types, like *regular moon* (`Q1086783`) and *natural satellite* (`Q2537`). This allows entities to have dozens of valid types through inheritance. Combined with the collaborative nature of Wikidata, this creates a difficult environment for choosing a single, most natural type for an entity.

2.3. NLP and Entity Typing

Natural Language Processing (NLP) is a subfield of Computer Science, artificial intelligence, and linguistics that focuses on enabling computers to understand, interpret and generate human language. It involves a wide range of tasks, such as Part-Of-Speech tagging, Sentiment Analysis, Machine Translation, and Named Entity Recognition (NER). More recently, NLP has become an area of intense research due to the rise of large language models.

A specialized task within NLP is Entity Typing, which focuses on classifying mentions of entities in text into types. While it is closely related to NER, which involves first identifying entities in text and, in some cases, assigning types to them, Entity Typing focuses solely on type assignment. It often involves working with a predefined ontology such as DBpedia or Wikidata, and existing systems vary significantly in terms of methodology and type definitions.

3. Related Work

This chapter reviews related work in the field of entity typing, and where it differs from our approach. We conclude by identifying the current research gaps that our work aims to address.

3.1. Types and Wikidata

The idea of assigning a type to each Wikidata entity is not entirely new. Discussions about adding a dedicated property for this purpose, `P107`, date back to 2013 [10]. Various suggestions for the property name were proposed, such as “item type”, “main item type”, or “entity type”.

During these heated discussions, one proposal was to classify entities using the integrated authority file (GND—Gemeinsame Normdatei), an ontology maintained by the German National Library. It provides six main types, which are divided into around 50 subtypes. The proposed implementation of `P107`, however, planned to only use the six main types of GND.

Ultimately, `P107` was deprecated in favor of `P31`, which offers much greater flexibility. Accommodating the diverse and evolving range of entities in Wikidata is a difficult task, and `P31` allows for a more nuanced and adaptable approach. This flexibility however comes with the cost of consistency, and it doesn’t allow for obtaining a uniform view of an entity’s type.

3.2. Entity Typing

The broader field of entity typing has seen considerable research during the last decade. This section reviews key contributions to the field.

In 2012, [11] introduced Tipalo, a system that automatically assigns types to DBpedia entities by leveraging their natural language (NL) descriptions from Wikipedia. First, a short description of an entity was extracted from its corresponding Wikipedia page. Then, they used their tool FRED, which produces a graph representation from NL text using computational semantics. This produces an OWL representation including a taxonomy of types, a process which is referred to as taxonomy induction. Relevant types were then selected based on graph patterns. The last step involved transferring this type hierarchy to classes (e.g. chess piece) as well as instances (e.g. pawn). The result was a complete Wikipedia ontology, which, however, is no longer accessible online. Evaluations on a human-annotated gold standard dataset showed good precision and recall. While Tipalo focused on creating a new ontology from Wikipedia descriptions to assign types to DBpedia entities, our goal is to assign the most natural type to each Wikidata entity based on existing classes. Tipalo focused on typing DBpedia entities, while our work focused on Wikidata entities.

In 2013, [12] introduced TRank, a system that ranks entity types based on context. The goal of their work was to improve search engine result pages, faceted browsing, and document summarization by providing the most contextually relevant entity types. For this purpose, they created a large-scale type hierarchy, combining multiple sources such as Freebase, DBpedia, and Wikipedia. This was done by combining subclass relationships, resolving cycles and gaps with equivalence mappings, as well as manual adjustments. Their work focused on assigning types to entities extracted from web pages and mapped to a knowledge base. It employs entity-centric, context-aware, and hierarchy-based methods to rank types. The system was evaluated using crowd-sourced relevance judgments across four distinct datasets: Entity Only (EO), Entity Collection

(EC), Paragraph (PG), and 3-Paragraphs (3PG). Building on the work of TRank[12], [13] introduced TRank++, an extension and refinement of the original TRank system for journal publication. They built on the previous work by adding new ranking approaches and features. While they also created a context-independent dataset (EO), TRank's primary goal was context-dependent type ranking, whereas our approach aims to identify a single, context-independent, natural type for each Wikidata entity. Their approach relied on a manually adjusted type hierarchy derived from several sources, while our work focuses solely on Wikidata and its existing entities.

[14] addressed the entity ranking problem by leveraging knowledge-graph (KG) embeddings, treating the task as a special case of KG completion. They introduced a new relevance property connecting entities and their types, creating triples in the form `<entity, relevance, entity_type>`. Relevance judgments for entity type pairs, which serve as the training data, were created via anonymous crowdsourcing. They considered two scenarios: (1) Using only the entity and its structural context in the KG, and (2) adding textual context (from New York Times articles) for the entity. The relevance of entity-type pairs was determined by crowd-sourced votes. The evaluation was done on the four datasets introduced in [12]. While their approach is a notable contribution, it differs from ours in several aspects. First, they used a different dataset. Second, while their approach focuses on ranking types based on context, our method identifies the single most natural type for each entity. Finally, they crowdsourced training data, while we leveraged modern language models to select relevant types.

[15] introduced ManyEnt, a benchmark for few-shot entity typing, and evaluated a BERT-based model on it. Based on the FewRel dataset, their goal was to predict the type of an entity positioned in a sentence, given only a limited number of examples. ManyEnt uses a manually created set of 256 specific and 56 general types from Wikidata. Training data and benchmarks were created via a breadth-first search (BFS) starting from the entity, assigning the first matching type from the set as the entity's type. For classification, they used BERT, a pre-trained language model, both with and without

fine-tuning. Prediction was based on each entity's embeddings created by BERT, and their distances to prototype embeddings of each type. The type with the smallest distance was chosen as the predicted type. Prototype embeddings were computed by averaging the embeddings of the support set of entities. They evaluated the model on two benchmarks, ManyEnt-256 and ManyEnt-56, using 1-shot and 5-shot settings. They achieved accuracies of up to 90% on ManyEnt-256 and 92% on ManyEnt-56 in a 5-shot, 5-way, fine-tuned setting. While their approach shares the use of Wikidata with ours, it differs in several aspects: The inclusion of context, i.e. the sentences in which the entities appear, the manual creation of a pre-defined set of types, and the use of BFS-based ground-truth labels. In contrast, our method employs a large language model (LLM) to generate training data, and we predict types based solely on the entity itself, without incorporating textual context.

Another interesting work is [16], a recent (2024) approach to refine the Wikidata Taxonomy using LLMs, introducing a method to clean up and simplify its structure. They propose a new, automatically refined taxonomy called WiKC, produced by methodically merging classes, cutting inaccurate or irrelevant links, and reducing inconsistencies through zero-shot LLM prompts. Their evaluation also involves an entity typing task, where they leverage their refined taxonomy to classify entities more reliably. Those type assignments are then judged by another LLM, based on context about the entity, and compared to the original Wikidata types. They reported an accuracy of 37.5% on the original Wikidata types and 75% on the WiKC types. Our work differs in two key ways. First, rather than modifying Wikidata's structure at the class level, we focus on selecting the single most natural type for each entity based on all existing classes. Second, while their LLM usage primarily involves predicting the correct relation (e.g., subclass-of vs. irrelevant) between classes, we generate large-scale training data via an LLM to directly train models. Even though our ultimate goal is very different from theirs, both approaches address challenges imposed by Wikidata's complexity. They prune and refine the taxonomy, while we aim to assign each entity its most intuitive and natural type.

3.3. Research Gaps

While entity typing is a well-researched topic in NLP, several gaps remain. Existing approaches primarily focus on predicting an entity's type in a specific context, often ranking types based on context in a few-shot setting. However, our work introduces a new perspective by aiming to identify the most natural and intuitive type for each entity, rather than a context-dependent one. As a result, there are no established baselines or benchmarks for this task.

Most existing research has focused on knowledge bases such as DBpedia, with only a few approaches considering Wikidata. However, even those that do focus on Wikidata typically do not address natural typing.

Furthermore, while many approaches rely on small-scale crowdsourced training data or few-shot learning, we leverage modern LLMs to generate large-scale training data which we use to train traditional machine-learning models.

In the following chapter, we describe our approach in detail and how it addresses these gaps.

4. Methodology

This work focuses on supervised entity typing in Wikidata. While unsupervised, rule-based approaches exist, such as selecting the superclass of an entity with the highest in-degree, they were not suitable for this task. Such methods, while potentially accurate and not uncommon, require more consistency than what Wikidata provides, and early experiments regarding those approaches were quickly abandoned.

Therefore, this work adopts a supervised approach, using a large set of labeled data to develop machine-learning models. While this approach offers great potential, it requires a careful methodology for data selection and model training, which is described in the following sections.

4.1. Data

4.1.1. Candidate Types

To determine which types are eligible for each entity as natural type candidates, existing connections through the properties `P31` and `P279` were used. This was done for two reasons:

1. **Entities as Types:** Assigning types that are not Wikidata entities (e.g., "Person", "Location") could introduce unnecessary complexity. It is standard practice to use an existing ontology for this purpose.

2. **Consistency:** Using Wikidata entities as types enables a selection of a well-defined set of candidate types through existing connections. This ensures consistency and allows for easier evaluation, as the search space will be limited to a set of pre-defined types.

However, as noted by [16], the rules for usage of `P31` and `P279` are not always followed in Wikidata. Sometimes, `P31` is used in place of `P279`, and vice versa. Our approach assumes the proper application of these properties, and we used the following criteria to determine which types are allowed for each entity:

- **Layer 1**: For a given entity, types directly reachable through `P31` are allowed.
- **Layer 2 and beyond**: From the second layer onward, types reachable through `P279` are allowed.
- **Exception 1**: If an entity has no outgoing `P31` edge, `P279` is allowed in the first layer.
- **Exception 2**: If no types are reachable through `P279` at the second layer, and there is only one valid connection in the first layer, `P31` is used again at the second layer.

Consider the entity *excavator* (`Q182661`) as an example, shown in Figure 4. In the first layer, *heavy equipment* (`Q874311`) is discarded as a potential type, as it is only reachable through `P279`. In the second layer, *first-order class* (`Q104086571`) is discarded, as it is only reachable through `P31`. In this example, potential type candidates are *physical tool* (`Q39546`), *appliance* (`Q1183543`), *machine* (`Q11019`), *converter* (`Q35825432`), and *mechanical device* (`Q1882685`).

Our approach is designed to be intuitive. If an entity is an instance of another entity, then due to the transitivity of `P279`, it is also an instance of all superclasses of that entity.

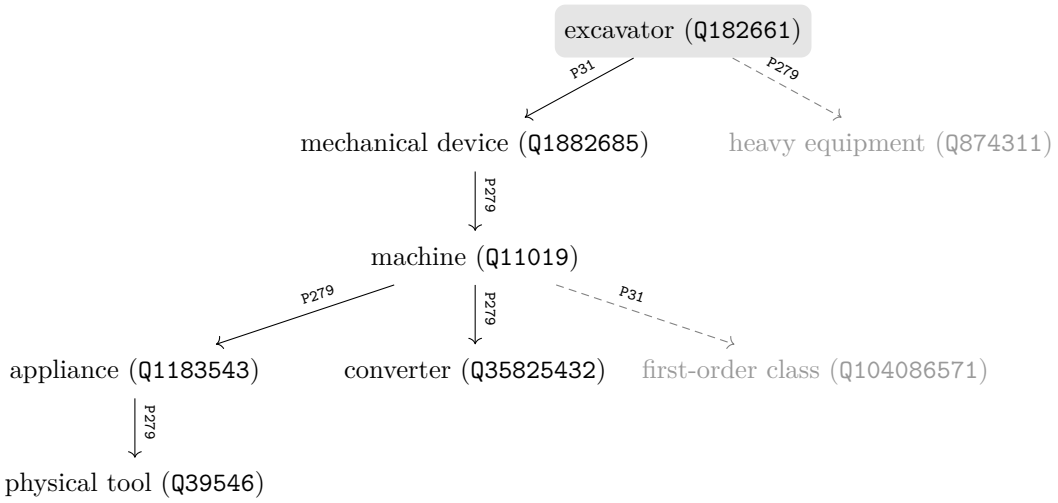

Figure 4.: Ontological Structure of Excavator (Q182661). Candidate types are indicated with solid lines, while discarded types are indicated with dashed lines.

Of course, there exist many entities that have no outgoing P31 connections, but only P279. Whether or not to assign a type to an entity that is solely a subclass of other entities is a difficult question; Exception 1 allows for this possibility. Instances and classes are not always clearly separated in Wikidata. While there exist some general guidelines about when an entity is a class or an instance, they are not strictly enforced, and there are exceptions. For example, P31 is typically used to connect instances to classes. However, some entities are meta-classes, meaning that each of their instances is a class themselves. Also, some entities are both an instance and a class, as described in [17]. To maintain a balance between accuracy and simplicity, we decided to allow traversing P279 in the first layer if no P31 edge is present.

The second exception is designed to handle a rare case where an entity is an instance or subclass of only one entity in the first layer, which itself is not connected to any other entities through P279. For example, *1993 Asian Athletics Championships* (Q596452) is an instance of *Asian Athletics Championships* (Q1138826). Q1138826 has no outgoing P279 connections but is connected through P31 to, for instance, *sports competition* (Q13406554), which would be a more appropriate type for Q596452. Similarly, *CSE-CSPG5 [lysosomal*

lumen] (Q50686512) is an instance of *Chondroitin Sulfate Proteoglycan 5* (Q21172743), which also doesn't have an outgoing P279 connection. However, Q21172743 is connected via P31 to *protein* (Q8054), which is a much better type for Q50686512 than *Chondroitin Sulfate Proteoglycan 5* (Q21172743), which would be the only type candidate without this exception. This scenario is infrequent and usually triggers a warning in Wikidata, such as a "value-requires-statement constraint" which indicates that, for instance, *Asian Athletics Championships* (Q1138826) should have a subclass of statement, since it is used as a class. In practice, this exception is rarely needed, but it is included to handle such edge cases.

4.1.2. Training Data

Now that we have established candidate-type selection criteria, the next step is to obtain training data. Getting high-quality, labeled training data is notoriously difficult. To automate this process, which is traditionally a manual, labor-intensive task, we used a large language model (LLM) to label entities.

Small versions of frontier LLMs (e.g., GPT-4o-Mini or Gemini Flash) are both capable and inexpensive, costing around $0.075 per million input tokens and $0.3 per million output tokens at the time of writing [18]. The drastic reduction in cost and improved capabilities over the last two years enabled us to generate thousands of labeled entities quickly and for only a few dollars.

However, when using a LLM, type assignments cannot be controlled directly. Instead, the LLM needs to be guided to select the correct type. The only way to do this is through prompt engineering. In practice, this involves a so-called "system-string" which is provided to the LLM with each API call. The system-string we used is included in the Appendix A. It consists of a series of instructions for the LLM, such as detailed criteria for type selection, preferred output format, and examples of valid type assignments.

LABEL: Andromeda Galaxy
DESCRIPTION: barred spiral galaxy within the Local Group

POSSIBLE TYPES:
deep-sky object (Q249389)
independent continuant (Q53617489)
material entity (Q53617407)
natural physical object (Q16686022)
galaxy (Q318)
concrete object (Q4406616)
disc galaxy (Q1371025)
entity (Q35120)
spiral galaxy (Q2488)
astronomical object (Q6999)
continuant (Q103940464)

Figure 5.: Example of LLM Input for an Entity. The input provides the LLM with an entity's label, description, and list of candidate types (including Label and QID), which were randomly shuffled and separated by line breaks.

For each entity, the model was then provided with both its label and description, as well as with the selection of candidate types (see Section 4.1.1). As instructed in the system-string, the model had to select a single type from the list of candidates. Types were provided in the format "Label (`Q1234567`)", shuffled randomly, and separated by line breaks. One such example is shown in Figure 5.

Deciding which entities to include in the training data is not trivial. Selecting entities randomly without filtering would yield a narrow coverage of structural positions within Wikidata, as entities with certain structural positions (e.g., humans or scientific articles) would be overrepresented. Moreover, entities that share the same natural type can be connected very differently in the graph.

To ensure broad coverage of structural positions and variety in the training data, entities were selected as illustrated in Figure 6.

For each entity, a string was created which reflects its structural position in the Wikidata graph. All entities directly connected to the entity through `P31` and `P279` were prefixed

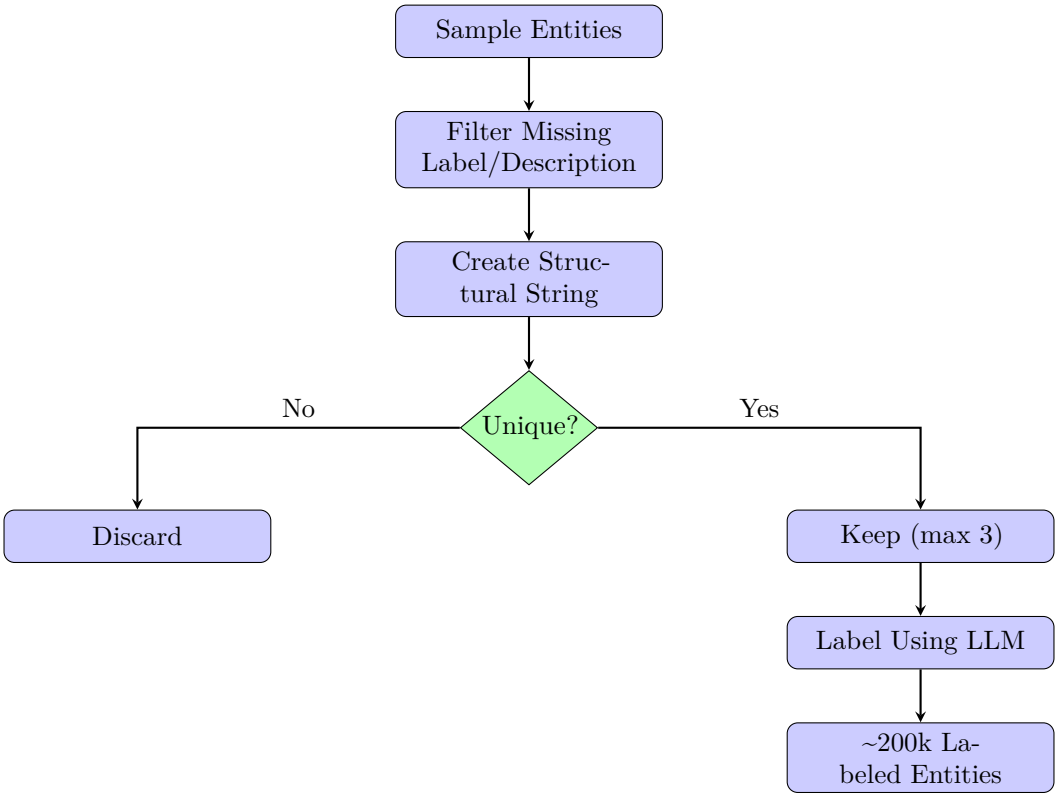

Figure 6.: Flowchart of the Training Data Generation Process. Entities are sampled from Wikidata, filtered for missing labels/descriptions, and assigned a unique structural string based on their `P31` and `P279` relations. Up to three entities per unique string are kept and subsequently labeled using an LLM, resulting in approximately 200,000 labeled entities.

with either "i" or "s" ("i" for `P31`, "s" for `P279`), sorted alphabetically and concatenated. This way, each structural position in the ontology is represented by a unique string.

For example, for *Earth* (`Q2`), which is an instance of *terrestrial planet* (`Q128207`) and an instance of *inner planet of the Solar System* (`Q3504248`), the resulting string would be "`iQ128207iQ3504248`". *Non-coding RNA* (`Q427087`), which is only a subclass of *RNA* (`Q11053`), has "`sQ11053`" as its string.

Next, we sampled a few million entities randomly from Wikidata, filtered those out that had no label or description, and created a structural string for each entity. A maximum of 3 entities per unique string were kept, while all others were discarded. From the original

millions of entities, only ~180,000 remained, which were then labeled using the LLM, one by one.

The large number of entities is beneficial for two main reasons:

1. **Rarity of certain types**: Some types (e.g., *ocean* (`Q9430`) for *Atlantic Ocean* (`Q97`)) apply to very few entities. Similarly, obtaining a diverse set of entities for *volcano* (`Q8072`) requires scanning through millions of entities.

2. **Structural variations even for common types**: For example, both *Apple* (`Q312`) and *BMW* (`Q26678`) are of the type *enterprise/company* (`Q6881511`), but their connections differ significantly. Similarly, *Saturn* (`Q193`) and *Uranus* (`Q324`) share the type *Planet* (`Q634`), but have very different structural positions in the graph. Covering a wide range of structural positions in the graph is crucial for training a well-generalizing model.

4.1.3. Evaluation Data

Following standard practice in natural language processing, we evaluated our system's performance using benchmark datasets. Two evaluation sets were created, each addressing different aspects of representation.

- **First Dataset (500 entities):** These entities were selected in the same way as the training data but restricted to one entity per unique structural representation string (as described in Section 4.1.2).

- **Second Dataset (300 entities):** These entities were deliberately chosen by hand to cover a diverse and well-known set of entities.

All 800 entities were carefully labeled by a human annotator, in accordance with the criteria described in Section 4.1.1. In most cases, a single type was designated as correct.

However, for certain entities, two or (in rare instances) three types were accepted, resulting in an average of 1.1 valid types per entity across both datasets.

This strategy was used to avoid penalizing the model for minor variations in specificity or phrasing. For example, *family name* (`Q101352`) and *surname* (`Q121493679`) were considered to be equally valid for the entity *Simpson* (`Q2800825`). This ensures that only clearly incorrect predictions are penalized, rather than minor variations that even humans might disagree on.

4.2. Feature Extraction

Machine learning models require numerical input. Therefore, to predict entity types, we must convert attributes of each entity into numerical feature vectors.

Generally speaking, a feature is a measurable property that can be used for a prediction task. For instance, the weight of a car is a feature that can be used to predict its fuel consumption, while the location and size of a flat can be used to estimate its price. In the context of Wikidata, however, feature extraction is more complex since entities have multiple textual and structural attributes, rather than numerical properties.

Wikidata entities are commonly characterized by a variety of attributes, such as labels (e.g. `Q42` corresponds to "Douglas Adams"), brief descriptions (like "English author and humorist"), statements, sitelinks to pages on other Wikimedia projects, and alternative names (aliases). Each of these information sources can help to identify an entity's type.

In this work, we combined three types of features to form a comprehensive feature vector for each entity.

1. **Node Degrees (Section 4.2.1):** in-degree, out-degree, and their ratio with respect to different properties.

2. **RDF2Vec Node Embeddings (Section 4.2.2):** embeddings capturing an entity's structural position in the Wikidata graph, generated using RDF2Vec.

3. **Universal Sentence Encoder (Section 4.2.3):** sentence embeddings that capture semantic information from entity descriptions.

4.2.1. Node Degrees

A simple feature for each node - requiring no additional precomputation - is its in-degree and out-degree, meaning the number of incoming and outgoing edges, respectively. Because a small number of nodes have extremely high in- or out-degrees, while the vast majority have relatively low degrees, we applied a logarithmic transformation. Additionally, the ratio of in-degree to out-degree was calculated. Using only `P31` and `P279`, these steps yield a total of 5 features for each node (in-degree `P31`, in-degree `P279`, out-degree `P31`, out-degree `P279`, and the ratio, each transformed logarithmically). We also incorporated additional edge types, namely *part of* (`P361`), *has use* (`P366`), and *day of week* (`P2894`), in the degree calculations.

4.2.2. Node Embeddings

Since `P31` and `P279` make a concrete statement about the type of an entity, they should provide highly valuable information for determining the type of an entity.

The simplest possible embedding is one-hot encoding, where each entity (node) is represented as a binary vector of length equal to the total number of entities, with a 1 at the position corresponding to that entity and 0 everywhere else. However, this would result in count(items) $\times$ count(items) parameters, which is infeasible. Also, such embeddings are very sparse and do not capture information about the similarity of entities.

Instead, more sophisticated approaches, such as DeepWalk [19], Node2Vec [20] and RDF2Vec [21], have been developed to generate node embeddings. These methods perform random walks on the graph, treating the resulting sequences as sentences, with nodes acting like words. Then, Word2Vec [22] is used to learn embeddings for each entity.

To incorporate edge information, the random walks can integrate edge labels, as proposed by [21] for RDF graphs. Although pre-trained RDF2Vec embeddings are available for Wikidata [21], we generated them ourselves to ensure they are up-to-date and to have full control over the training process.

For each entity, 10 random walks of length 8 were generated. Figure 7 shows examples of such walks.

`Q100000004` → i → `Q3914` → s → `Q5341295` → s → `Q43229` → s → `Q106668099`

`Q104044591` → i → `Q13433827` → i → `Q223393` → i → `Q19478619` → i → `Q19868531`

`Q102329267` → i → `Q13442814` → s → `Q30070590` → s → `Q191067` → s → `Q4263830`

`Q100801691` → i → `Q5` → i → `Q55983715` → i → `Q19478619` → i → `Q23958852`

Figure 7.: Examples of Wikidata Walks for RDF2Vec. Random walks of length 8 are generated using membership properties `P31` and `P279`, represented by "i" and "s", respectively.

Only `P31` and `P279` edges were added to the walks ("i" and "s" for `P31` and `P279`, respectively, to save space), since these are the type-defining properties. Among all possible connections through `P31` and `P279`, the next step in the walk was selected randomly.

To create embeddings from these walks, we used Skip-Gram [22]. Skip-Gram employs a shallow neural network with one hidden layer to predict surrounding words (nodes) given a target word. As illustrated in Figure 8, the dimensionality of the input and output layer is equal to the vocabulary size (number of nodes/entities), while the dimensionality

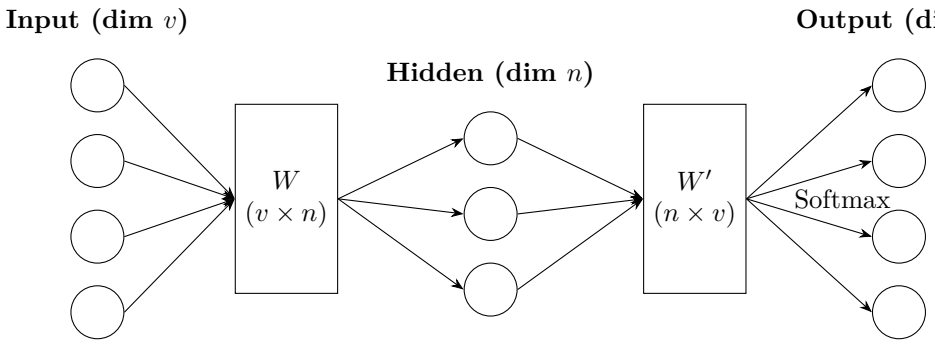

Figure 8.: Skip-Gram Model Architecture. A one-hot encoded input (dim v) is projected to a hidden layer (dim n) via weight matrix W $(v \times n)$, then mapped to the output (dim v) through W' $(n \times v)$ with a softmax function. After training, the rows of W serve as word embeddings.

of the hidden layer is equal to the embedding size, 32 in our case. The weight matrix W has dimensionality $v(vocabularysize) \times n(embeddingsize)$.

Training Skip-Gram involves learning pairs of words (nodes) that occur together within a specified window size. For example, for a given sequence of nodes and properties

$$\texttt{Q3914} \rightarrow \text{i} \rightarrow \texttt{Q5341295} \rightarrow \text{s} \rightarrow \texttt{Q43229}$$

and a window size of 2, the pairs for the target node `Q5341295` are:

- (`Q5341295`, `Q3914`) *(two steps back)*
- (`Q5341295`, i) *(one step back)*
- (`Q5341295`, s) *(one step forward)*
- (`Q5341295`, `Q43229`) *(two steps forward)*

This means that the window size directly influences the context distance the model considers. During training, the model learns to predict the context node given the target node, each one-hot encoded. The input vector is multiplied with W, zeroing out all rows of W except for the row corresponding to the input word. Then, the hidden layer activations

are multiplied with the output weight matrix (size $n \times v$) and passed through a softmax function, which results in a probability distribution over the vocabulary. Cross-entropy loss is then minimized between predicted probabilities and the one-hot encoded target word, using backpropagation to update W and W'. After training, the rows of W serve as word embeddings.

Intuitively, W is optimized to map target words to embeddings that are useful for predicting their context words. At the end of training, each node has a 32-dimensional vector, where semantically or structurally similar nodes end up close to each other in the embedding space.

We used `Gensim` to train the embeddings with a window size of 4, a vector size of 32, and a minimum count of 1, meaning that all nodes were included in the vocabulary. While a dedicated Node2Vec python package exists, we generated the random walks separately since `Gensim`, through the `PathLineSentences` functionality, allows training from sentences in a streaming fashion rather than having to load them into memory.

4.2.3. Description Embeddings

Another important feature for type prediction is the entity description. According to Wikidata guidelines, it should consist of two to twelve words [23]. Descriptions are not full sentences but rather small bits of information that provide additional context about the entity. The entity description exists to disambiguate entities with the same or similar labels (although, two entities can have the same description).

Some descriptions offer limited discriminative power for type prediction. For example, the description of the entity `Q107294393`, a Russian princess from the 17th century, is "died 1721", which isn't informative at all, and many entities lack descriptions altogether. Still, combined with other features, they should provide valuable information.

To create rich feature vectors from entity descriptions, we used Universal Sentence Encoder (USE) [24]. While traditional sentence embedding methods such as Bag of Words, Term

Frequency-Inverse Document Frequency [25] or averaging Word2Vec embeddings capture limited semantic information and ignore word order, USE overcomes these limitations.

USE is available in two variants: one based on a transformer architecture [26] and another using a deep averaging network (DAN) [27]. In our work, we used the transformer-based version accessible via `TensorFlow Hub` [28], which produces a 512-dimensional embedding that captures both semantics and word order.

A simplified illustration of how USE works is shown in Figure 9.

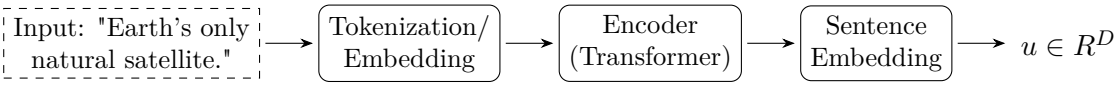

Figure 9.: Universal Sentence Encoder (USE). The input sentence is tokenized and embedded, then passed through the encoder to obtain a sentence embedding.

In this process, an input sentence is first tokenized, meaning the sentence is split into individual tokens (words or subwords). Each token is then converted into a numeric vector using a pre-trained lookup table. These token embeddings are then passed through an encoder, which in our case uses a transformer architecture, as described in [26], to capture contextual relationships among tokens. The output of the encoder is then used to generate a sentence embedding, which typically involves pooling operations, such as averaging the token embeddings or using a weighted combination. The final output, $u \in R^D$, represents the sentence embedding, where D (512 in our case) is its dimensionality.

Unlike the pre-trained RDF2Vec embeddings, USE embeddings were generated in real-time during training and inference. Storing 512-dimensional vectors for ~90 million descriptions would be infeasible (requiring approximately 400 GB of storage), and attempts to precompute and store them using dimensionality reduction techniques, such as incremental PCA, did lead to a slight reduction in performance.

4.3. Machine Learning Models

4.3.1. Candidate Masking

The final training dataset contains over 4,000 types. However, as stated in Section 4.1.1, only a subset of these types is actually allowed for each entity. To enforce this constraint on the machine learning models, we used a masking mechanism, both during training and inference, on the output layer. This mask consists of a binary vector with a length equal to the number of types. It contains a 1 for allowed types and 0 otherwise. During the forward pass, this mask is applied to the model's Logits, as shown in Figure 10.

```
class MaskedFNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        # shortened, exemplary implementation
        self.layers = nn.Sequential(
            nn.Linear(input_dim, output_dim)
        )

    def forward(self, x, mask=None):
        logits = self.layers(x)
        if mask is not None:
            logits = logits + torch.log(mask + 1e-9)
        return logits
```

Figure 10.: Exemplary Implementation of a Masked Neural Network. The mask is applied on the logits in the forward pass, prior to the softmax activation.

Here, "Logits" refers to the raw output of a neural network prior to any activation function. Adding `torch.log(mask + 1e-9)` to the logits leaves allowed types (where the mask is 1) unchanged, since $\log(1) = 0$. The small value $1e-9$ is deliberately chosen to avoid affecting the probabilities of allowed types while preventing $\log(0)$ for disallowed types. For disallowed types (where the mask is 0), this operation pushes them to large negative values, because $\log(1e-9) \approx -20.72$.

The whole process is illustrated in Figure 11. As can be seen, after the mask is applied, the model's output is passed through a softmax function to obtain a probability distribution.

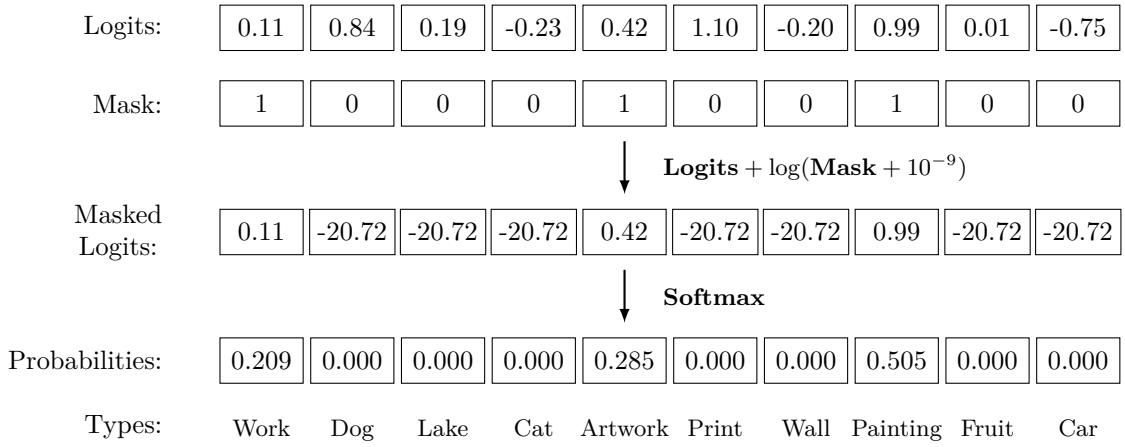

Figure 11.: Visualization of the Logit Masking Process. Logits corresponding to disallowed types (i.e., where the mask is 0) receive a large negative offset. After applying the softmax function, these logits yield probabilities close to zero.

In the example, Painting has a probability of 0.505, Artwork 0.285, and Work 0.209, while all other types have a probability of 0.

When the masked logits are processed by the softmax function, the exponentiation causes large negatives (assigned to disallowed types) to become probabilities near zero. The function is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad \text{for } i = 1, \ldots, K$$

Each Logit z_i is exponentiated using e as the base. When z_i is a large negative number (like -20.72 in our case), e^{z_i} becomes very small, as $e^{-20.72}$ is equivalent to $\frac{1}{e^{20.72}}$, and $e^{20.72}$ is very large.

This constrains the model to predict from only the pre-selected set of allowed types, rather than from all types. Using the mask during inference is straightforward, as it ensures that only allowed types achieve a high probability. The effect during training is more subtle. The mask ensures that no meaningful gradient is backpropagated for disallowed types, so the model learns to focus solely on the allowed ones. Without the

mask, the model has to learn to suppress wrong types, but due to the mask, disallowed types automatically receive a probability close to zero. Therefore, the model can solely focus on distributing probabilities among allowed types.

Comparable masks have been applied in various contexts, such as hierarchical image classification [29] and other applications requiring constrained output spaces. More broadly speaking, masking is a common technique in machine learning for a variety of tasks. Our approach extends this concept to entity typing in Wikidata.

4.3.2. Class Weights

Another important consideration in training is class imbalance. Although the training dataset was created in a way that ensures broad coverage of structural positions in Wikidata, some types are naturally overrepresented. For example, *type of chemical entity* (`Q113145171`) appears over 7,000 times in the finished training set, while *volcano* (`Q8072`) appears only 23 times. In order to allow for balanced training, we applied class weighting via Smoothed Inverse Frequency, as seen in the following formula:

$$w_j = \left(\frac{\max(c_k)}{c_j + \epsilon} \right)^{\alpha}$$

This is a commonly used technique for imbalanced datasets. w_j is the weight assigned to class j, c_j is the number of occurrences of class j, $\max(c_k)$ is the highest occurrence count among all classes, and ϵ is a small constant to prevent division by zero. α controls the difference between high and low weights.

A value for α of 1.0 results in pure inverse frequency, while a value like 0.5 (square root) makes the weighting less aggressive. In our case, 0.07 was chosen as the exponent, as it led to the highest validation accuracy.

4.3.3. Feedforward Neural Network

The simplest neural network architecture is the Feedforward Neural Network (FNN), also known as a Multi-Layer Perceptron (MLP). It consists of an input layer with the same dimensionality as the feature space, one or more hidden layers, and an output layer. The dimensionality of the output layer corresponds to the number of classes in the training data. In our case, we implemented a typical design for classification tasks, featuring fully connected layers, batch normalization, non-linear activation functions (ReLU), and dropout. Batch normalization normalizes the output of each layer, ensuring a mean of 0 and a standard deviation of 1. Dropout, on the other hand, randomly sets a certain percentage of neurons to zero during training to prevent overfitting.

Training was done using `cross-entropy loss`, which is a common choice for multi-class classification tasks, along with the `Adam` optimizer.

4.3.4. GraphSAGE

To incorporate information not only from individual nodes but also from their neighbors, we experimented with various graph neural networks (GNNs). While RDF2Vec already creates embeddings based on graph structure, it is trained in an unsupervised manner and does not account for the target task. In contrast, GNNs generate new embeddings for each node by aggregating information from its neighbors in a way that is fine-tuned for the specific task (i.e., supervised learning).

The first GNN we tested was GraphSAGE [30], a method for inductive representation learning on homogeneous graphs.

In contrast to transductive methods, which require the entire graph (and all node features) to be present during training, inductive methods can generalize to nodes not seen during training. To achieve the highest accuracy, we integrated RDF2Vec embeddings as node

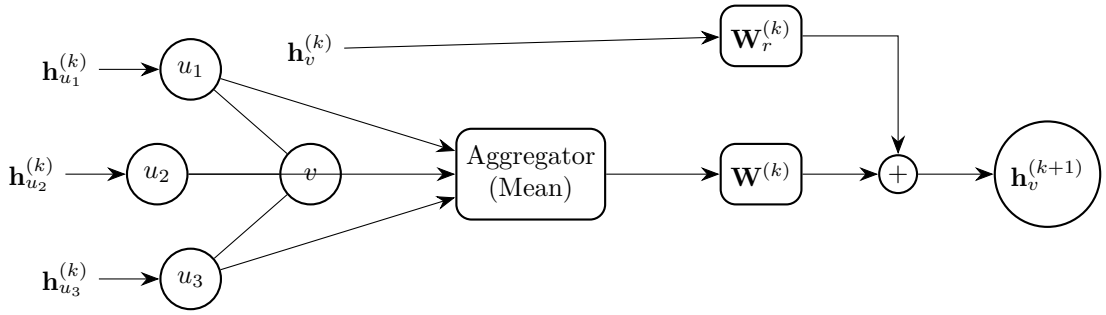

Figure 12.: GraphSAGE Convolution at layer k. The neighbor embeddings $h_{u_1}^{(k)}, h_{u_2}^{(k)}, h_{u_3}^{(k)}$ are aggregated, transformed by $W^{(k)}$ and combined with the central node's feature transformed by $W_r^{(k)}$ to produce $h_v^{(k+1)}$.

features within GraphSAGE (and other GNNs), rather than relying solely on standard inductive features.

GraphSAGE creates a task-specific embedding for each node by aggregating information from its neighbors. This process is usually referred to as graph convolution. Although the term "convolution" might indicate some similarity to convolutional neural networks (CNNs), GraphSAGE's approach is quite different. In a CNN, a convolution aggregates information from spatially adjacent pixels, while in GraphSAGE, the convolution aggregates information from a node's neighbors in the graph. Several aggregation functions can be used, such as mean, max, or LSTM-based aggregation.

Figure 12 shows the architecture of a simple GraphSAGE convolution. The learnable part of each convolutional layer is the weight matrix $W^{(k)}$, which is applied after the aggregation step to transform the combined features into a new embedding, as well as the root weight $W_r^{(k)}$. Each GraphSAGE layer uses its own weight matrix, which is shared among all nodes. The root weight $W_r^{(k)}$ is used to directly transform the central node's features before aggregation, allowing the model to preserve the original node information better.

For our implementation, we used the GraphSAGE module from `PyTorch Geometric`, a library designed for graph-based learning in `PyTorch`. While the original GraphSAGE described in [30] does not inherently support multiple edge types, the `PyTorch Geometric`

implementation provides this functionality through `HeteroConv` [31]. This wrapper allows for heterogeneous graphs (i.e., graphs with multiple edge types) by first aggregating node embeddings per edge type (using mean in our case) and then performing a cross-type aggregation step (also using mean).

Another essential factor in GraphSAGE, and GNNs in general, is the number of layers. Each additional layer increases the receptive field, meaning that information can propagate across more hops. With a single layer, only immediate neighbors are considered. With two layers, neighbors of neighbors are accounted for, since a neighbor's embedding already integrates information from its own neighbors. This pattern continues with each additional layer. In our experiments, we used two GraphSAGE convolutional layers and, following common practice, added a final linear layer on top for classification.

4.3.5. Graph Attention Network

One limitation of GraphSAGE and many other GNNs is that by using pooling functions (such as mean or max), all neighbors receive the same importance [32]. A Graph Attention Network (GAT) [33] addresses this limitation by computing attention coefficients for each neighbor of a node, effectively assigning different importance to different neighbors.

Like GraphSAGE, GAT is an inductive method for learning node embeddings. We used the implementation from `PyTorch Geometric` [34].

Conceptually, GAT's approach can be broken down into three main steps [33]:

1. **Scoring Function**

$$e(h_i, h_j) = \text{LeakyReLU}\left(\mathbf{a}^\top \cdot [\mathbf{W}h_i \,\|\, \mathbf{W}h_j]\right)$$

   $e(h_i, h_j)$ is the attention score between nodes i and j, and h_i and h_j are the current node embeddings. The node features are transformed by a weight matrix W, concatenated, and then passed through a learnable weight vector a. This

vector a determines which features in the concatenated vector indicate a stronger relationship between nodes.

2. **Attention Coefficients**

$$\alpha_{ij} = \text{softmax}_j\left(e(h_i, h_j)\right) = \frac{\exp\left(e(h_i, h_j)\right)}{\sum_{j' \in \mathcal{N}_i} \exp\left(e(h_i, h_{j'})\right)}$$

The softmax function normalizes the attention scores so that they can be used as weights for aggregation. After this step, α_{ij} is a value between 0 and 1 and indicates the relative importance of neighbor j for node i.

3. **Aggregation**

$$h_i' = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \cdot \mathbf{W} h_j\right)$$

This equation describes how a new embedding h_i' is computed for node i by aggregating information from its neighbors. For each neighbor j, its feature vector is multiplied by the matrix W and weighted by the attention coefficient α_{ij}. As described in equation 2, α_{ij} represents the importance of neighbor j for node i. These weighted features are then summed up across all neighbors. This results in a single vector that represents the aggregated, attention-weighted information from the neighborhood. Finally, the vector is passed through a non-linear activation function σ, which results in the new node embedding h_i'.

In practice, GAT uses multiple attention mechanisms ("heads") in parallel, each with its own learnable parameters (W and a). The outputs from these heads are then concatenated or averaged to produce the final node representation, which allows the model to capture different aspects of node relationships simultaneously.

Similar to GraphSAGE, each layer in a GAT updates node features by aggregating information from neighbors, meaning the number of layers determines the receptive field (i.e., how many hops in the graph the model can observe). We tried different numbers

of layers and eventually settled on two GAT layers and a third linear layer for the final classification.

4.3.6. R-GCN

The Graph Neural Networks described above are primarily designed for inductive learning on homogeneous graphs. While they can be adapted for heterogeneous graphs, they are not specifically designed for them. GAT, for example, takes an incredibly large amount of memory when used on a graph with multiple edge types, since it creates a separate attention mechanism (including parameters and adjacency-like structures) for each edge type, which causes memory usage to scale quickly with the number of relation types.

Relational Graph Convolutional Networks (R-GCN) [35] are a type of graph convolutional network specifically designed to handle multi-relational graphs. While originally designed to be transductive, meaning that the whole graph should be present during training, it can be used inductively, as we do in this work. In this case, it's important to ensure consistency for edges used during training and inference.

R-GCNs work by aggregating information from neighboring nodes, separated by edge types. Therefore, similarly to the mentioned GraphSAGE and GAT models, the number of layers directly affects the receptive field of the model. Node embeddings are updated according to the following formula, as described in [35]:

$$h_i^{(l+1)} = \sigma\left(\sum_{r\in\mathcal{R}}\sum_{j\in\mathcal{N}_i^r}\frac{1}{c_{i,r}}W_r^{(l)}\,h_j^{(l)} + W_0^{(l)}\,h_i^{(l)}\right)$$

Here, $h_i^{(l)}$ is the embedding of node i at layer l, and $h_i^{(l+1)}$ is the updated embedding at layer $l+1$. $\mathcal{R}$ is the set of edge types, such as `P31`, `P279`, `P361`, etc. $\mathcal{N}_i^r$ is the set of neighbors of node i connected by edge type r, and $W_r^{(l)}$ is the weight matrix for edge type r at layer l. To weight self-loops differently, $W_0^{(l)}$ is the weight matrix for self-loops at layer l. Additionally, R-GCNs use normalization by the number of neighbors $c_{i,r}$ for

node i under relation (edge type) r. The σ (sigma) in the formula represents a non-linear activation function, such as ReLU, applied element-wise to the aggregated neighborhood information.

Intuitively, at each layer, neighbors are aggregated, transformed by a weight matrix, and then summed up, separated by edge type. While this might seem similar to GraphSAGE, the explicit handling of relation-specific normalization and transformation makes R-GCNs particularly well-suited for graphs with multiple edge types. Because of this, we trained R-GCN using the same features as GraphSAGE and GAT, but with additional, common Wikidata edge types (properties), such as *part of* (`P361`), *has use* (`P366`), and *day of week* (`P2894`), and we used two layers as well as a third linear layer for the final classification.

4.3.7. Hyperparameter Tuning

Hyperparameters of all machine learning models were tuned using a standard procedure. We split the training dataset 80/20 into training and validation sets and then performed a grid search over a range of hyperparameters (such as dropout, weight decay, hidden layer size, and class weights). The combination of hyperparameters yielding the best performance on the validation set was then selected for the final model.

4.3.8. Baseline Neural Network (Natalie Prange)

In addition to the models presented above, a neural network developed by my supervisor, Natalie Prange, was evaluated on the benchmark datasets.

This neural network employs a feature set that is notably distinct from our other approaches. In particular, it integrates type name embeddings and entity description embeddings with multiple scalar features, including inverse type frequency, predicate variance, and boolean indicators that check for the presence of the type name within the entity description or label.

The architecture is based on a feed-forward neural network, and it was trained on the same dataset generated for this work (although a smaller subset than the one used for the other models).

This common training ground enables a direct comparison of performance across different feature sets and modeling approaches. This architecture is part of Natalie's forthcoming doctoral thesis.

4.3.9. Large Language Models

Just like LLMs were used to generate training data, we also evaluated their performance on the benchmark datasets. Each LLM was provided with the instruction string described in Section 4.1.2, the label and description of the entity, and the pre-selected types. We evaluated a selection of LLMs for this task: GPT-4o-Mini, GPT-4o, Gemini Flash, and Gemini Pro.

5. Evaluation

5.1. Performance

As noted in Chapter 3, there exists no prior work regarding entity typing in Wikidata that can be used for benchmarking. Therefore, we created two custom datasets to evaluate our approach:

- **Random 500 Dataset:** 500 randomly sampled entities, each manually labeled. These entities were selected similarly to the training set: one entity per unique ontology string (see Section 4.1.2) was chosen and then labeled.

- **Handpicked 300 Dataset:** 300 manually labeled entities chosen to cover a diverse set of types (e.g., continents, volcanoes, deities, etc.).

Figure 13 shows the results on both benchmarks comparing Accuracy@1 and Mean Reciprocal Rank (MRR). Accuracy@1 is the percentage of correctly classified entities, while MRR is the mean reciprocal rank of the first correct prediction. While Accuracy@1 only takes the top-1 prediction into account, MRR considers the rank of the correct prediction. For example, if the correct answer is ranked first, the RR is 1; if it is ranked second, the RR is 0.5; if it is ranked third, the RR is 0.33, and so on. In our case, the MRR is then calculated as the average of all individual reciprocal ranks across all entities in the benchmark.

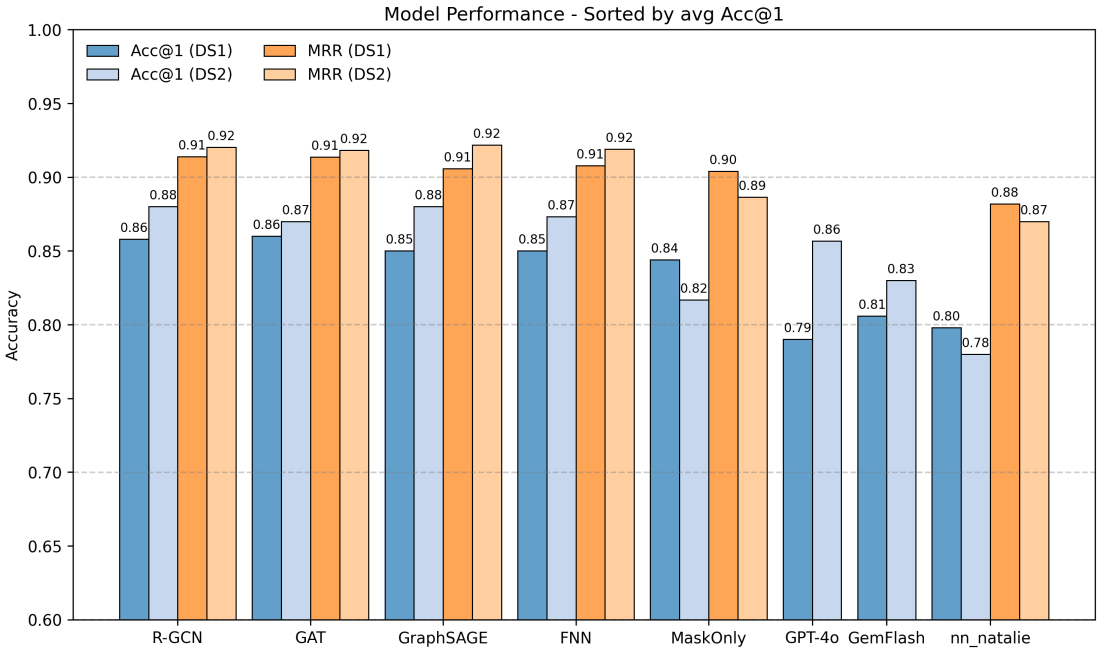

Figure 13.: Performance Comparison Across Models. Included are Accuracy@1 (percentage of correct top predictions) and MRR (mean reciprocal rank) for two datasets. DS1 - Random 500. DS2 - Hand-Picked 300. Higher values indicate better performance.

The R-GCN model leads with an average Accuracy@1 of 87% and an average MRR of 0.917, followed closely by the GAT, GraphSAGE, and FNN models, which show only marginal differences among them. Notably, the MaskOnly model—despite not incorporating node embeddings—delivers respectable accuracies of 84% and 82% on the Random 500 and Hand-Picked 300 benchmarks, respectively.

In contrast, LLMs show weaker overall performance. In particular, Gemini Flash, which was used to generate the training data, has an Accuracy@1 5% lower than the R-GCN for both benchmarks. GPT-4o, while performing better than Gemini Flash on the Random 500 dataset, falls short on the Hand-Picked 300 dataset. Additionally, the baseline model by Natalie Prange, which relies on traditional NLP features such as type name embeddings and entity description embeddings, yields the lowest accuracy among the tested methods. It is important to note that this model was trained on a smaller dataset

than the other approaches. Consequently, its lower accuracy is not solely a reflection of the feature set but also likely stems from the reduced volume of training data.

A likely reason the machine-learning models outperform Gemini Flash—the same system used to generate the training data—is that they consolidate multiple Gemini Flash outputs and thus smooth out individual errors. While Gemini Flash might correctly predict the type 83% of the time in single runs, the aggregated knowledge captured by the models leads to more consistent and accurate predictions.

The FNN was trained without using the RDF2Vec embeddings, only relying on the USE embeddings, degrees, and the mask. Adding the RDF2Vec embeddings did not improve the performance. Furthermore, more complex models like GraphSAGE and R-GCN did barely outperform the FNN. This suggests that the most critical information for type prediction may already be captured by the description embeddings and the mask. The additional information contained in graph embeddings such as RDF2Vec or GraphSAGE did not seem to provide substantial value here, and the performance ceiling might be limited by the quality of training data rather than the model's architecture itself. Discrepancies in performance between the benchmarks might be coincidental. However, they could also stem from the absence of descriptions (and thus USE embeddings) for 16 entities in the Random 500 dataset, while all entities in the Hand-Picked 300 dataset have descriptions.

Two plausible reasons for why the mask alone performs so well are: (1) Entities inheriting the same candidate types likely share similar positions in the hierarchy, which means they often also share the same natural type. (2) The candidate-type masks are not always unique for each entity because of transitive subclass relationships. For instance, *Mount St. Helens* (`Q4675`) is explicitly an instance of both *mountain* (`Q8502`) and *stratovolcano* (`Q113947`). *Stratovolcano* (`Q113947`) is a subclass of volcano (`Q8072`), which is a subclass of *mountain* (`Q8502`). Meanwhile, *Mount Egon* (`Q385596`) is only explicitly stated to be an instance of *stratovolcano* (`Q113947`). Despite these differences, both entities end up with the same set of type candidates via transitive closure—and thus share the same

mask. In this case, *volcano* (Q8072) is arguably the most natural type for both entities, so having the same mask is beneficial. Performance without the mask is shown in Figure 14. It can be seen that performance drops significantly without the mask.

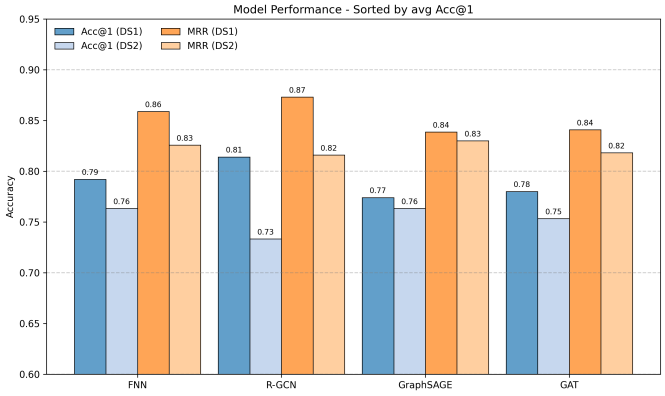

Figure 14.: Performance Comparison Without Mask. Included are FNN, GraphSAGE, GAT, and R-GCN.

All models without the mask were trained using RDF2Vec embeddings, USE embeddings, and degrees. Without the mask, GraphSAGE and GAT performed worse than the FNN and R-GCN. This likely stems from the fact that GraphSAGE and GAT were originally designed for homogeneous graphs, while R-GCN was specifically designed for heterogeneous graphs. Surprisingly, the FNN achieved the highest performance among all models, even surpassing R-GCN in average accuracy. The exact reasons for this are difficult to pinpoint. The FNN received structural information from the RDF2Vec embeddings, and perhaps the unprocessed RDF2Vec embeddings were more beneficial for the task than the neighborhood aggregation performed by the GNNs. During training, R-GCN achieved a validation accuracy around 3% higher than the FNN and it outperformed FNN on the Random 500 dataset by over 2%. However, FNN outperformed R-GCN on the handpicked 300 dataset. Since the main method presented in this work involves using the mask, these results were not further investigated. They show, however, that refining embeddings or model architecture alone could not compensate for the large search space of over 4000 types.

Figure 15 shows training and validation accuracy (train-test split 80/20) of the FNN under various configurations (with and without the mask, with and without RDF2Vec). While hyperparameters were carefully tuned to maximize validation accuracy, there still existed a clear gap between training and validation accuracy. This discrepancy was likely

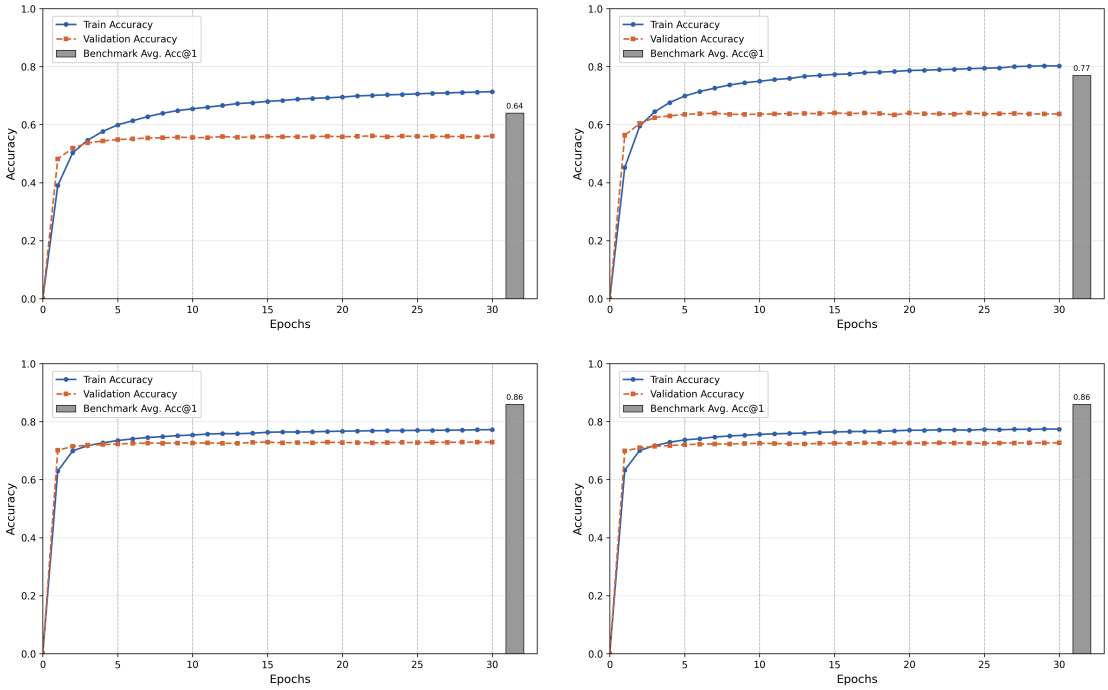

Figure 15.: Training and Validation Accuracy for FNN. Top left: Only USE embeddings. Top right: USE and RDF2Vec embeddings. Bottom left: USE embeddings with masking. Bottom right: USE and RDF2Vec embeddings with masking.

due to noise in the training data. Notably, the mask on its own yields higher validation accuracy than using USE + RDF2Vec without the mask, further reinforcing that the entity-specific candidate type constraint drives the majority of the predictive power.

Interestingly, the performance of the models on the benchmark datasets is significantly higher than the validation accuracy. Noise in the training data is likely the largest contributor to this discrepancy. For instance, the model might still be able to capture the broader pattern even if 10% of the training data is mislabeled. However, the presence of incorrect labels in the training set will naturally lead to a lower validation accuracy. Additionally, some entities in the benchmark were allowed to have multiple equally valid types (see Section 4.1.3), making it easier to predict their type. On average, each entity in the benchmark has 1.1 valid types, which is a small number, but it surely contributes to the discrepancy. While the impact of this multi-type allowance on this discrepancy is

likely minor compared to the noise in the training data, it is worth noting.

Results strongly suggest that the combination of a straightforward FNN, Universal Sentence Encoder embeddings, and the masking mechanism can achieve accuracy surpassing the results presented here. While 87% accuracy is already a strong result, it still means that 13% of entities are misclassified. Achieving higher levels of performance would likely involve refining the training data to better align with user preferences, as outlined in Chapter 6.

5.1.1. Natural Type Analysis

Table 1 shows a few predictions from the masked FNN. These unfiltered examples illustrate the model's typical performance. As shown, the model is highly accurate in predicting each entity's most appropriate type. Although the top-1 probability can be modest—for example, for *baseball cap* (`Q639686`) —the most fitting type remains the top prediction, often by a wide margin.

Despite the model's strong performance, incorrect predictions do occasionally occur, as shown in Table 2.

The exact reasons for these errors varied. In some instances, errors arise because the training data underrepresents certain entities. In other cases, the entity's description may be an outlier, or the entity may be uniquely complex in its connections.

Kreuzberg (`Q308928`) is predicted as a *locality of Berlin* (`Q35034452`) rather than a *neighborhood* (`Q123705`), which is too specific for our purposes. This error is easy to pinpoint: All neighborhoods of Berlin do have the description "locality of Berlin". Therefore, the LLM, during training data generation, always chose *locality of Berlin* (`Q35034452`) for any Berlin neighborhood, instead of following the criteria for selecting a fitting natural type outlined in the system string. As a result, this classification appears several times in the training data.

Entity (Wikidata ID)	Predicted Type (Wikidata ID)	Probability (%)
Mona Lisa (Q12418)	painting (Q3305213)	98.4
	work of art (Q838948)	0.6
	drawing (Q93184)	0.3
	visual artwork (Q4502142)	0.2
	sculpture (Q860861)	0.1
Craig (Q2671794)	given name (Q202444)	98.4
	male given name (Q12308941)	0.7
	name (Q82799)	0.7
	surname (Q121493679)	0.1
	family name (Q101352)	0.0
baseball cap (Q639686)	clothing (Q11460)	24.6
	headgear (Q14952)	6.5
	hat (Q80151)	6.0
	helmet (Q173603)	4.3
	accessory (Q362200)	3.1
Costa Concordia (Q190542)	shipwreck (Q852190)	67.8
	ship (Q11446)	5.9
	boat (Q35872)	4.8
	accident (Q171558)	3.2
	yacht (Q170173)	3.2
Mount St. Helens (Q4675)	volcano (Q8072)	66.9
	mountain (Q8502)	23.4
	landform (Q271669)	1.3
	cliff (Q107679)	1.0
	valley (Q39816)	0.8
American football (Q41323)	type of sport (Q31629)	74.1
	sport (Q349)	11.1
	game (Q11410)	5.6
	sports discipline (Q2312410)	1.9
	team sport (Q216048)	0.8

Table 1.: Examples of Correct Predictions. Top-5 predictions for various entities using the FNN.

This behavior highlights a critical issue: the LLM over-relies on description-matching entity labels rather than adhering to the instruction to choose more general, intuitive types. Whenever the description contains a specific label of one of the candidate types, the model tends to select it, even if this leads to an overly specific assignment.

The case of *cinnamon* (Q28165) illustrates another type of challenge: sometimes, what would be considered the most fitting natural type is not part of the candidate types. While *spice* (Q42527) would arguably be the most fitting type, it is not included among the candidate types. *Cinnamon* is connected via P31 to *crude drug* (Q735160), *food ingredient* (Q25403900), and *herbal medicinal product* (Q95997873), and from there, *spice* (Q42527) is not reachable via P279. In the benchmark, *food ingredient* (Q25403900) was determined to be the most fitting type, but the model failed to predict it, likely due to

Entity (Wikidata QID)	Predicted Type (Wikidata QID)	Probability (%)
Kreuzberg (Q308928)	locality of berlin (Q35034452)	72.6
	populated place (Q123964505)	12.8
	neighborhood (Q123705)	9.7
	human settlement (Q486972)	2.5
	ortsteil (Q253019)	2.2
Wilhelma (Q679067)	garden (Q1107656)	80.2
	botanical garden (Q167346)	17.1
	park (Q22698)	0.4
	museum (Q33506)	0.3
	arboretum (Q272231)	0.2
cinnamon (Q28165)	substance (Q378078)	12.3
	material (Q214609)	11.6
	fiber (Q161)	11.3
	dye (Q189720)	8.9
	type of wood (Q1493054)	6.0
quadrate bone (Q589072)	class of anatomical entity (Q112826905)	98.1
	class (Q5127848)	1.5
	entity (Q35120)	0.1

Table 2.: Examples of Bad Predictions. Top-5 predictions for various entities using the FNN. Selected errors illustrate common issues.

cinnamon's unique structural position within the knowledge graph.

Another example is *Quadrate bone* (Q589072), which, due to candidate type selection, can't be classified as a *bone* (Q3968). This is because its only P31 connection is to *anatomical entity* (Q112826905), which is a metaclass, making *Quadrate bone* (Q589072) a class itself (although it's never used as a class). Still, *bone* (Q3968) would be a more fitting natural type than *class of anatomical entity* (Q112826905), but it is not part of the candidate types.

These unusual cases require a solid familiarity with Wikidata to effectively leverage our natural types in downstream tasks. While the model's predictions may be optimal within the available candidate types, they are not always the most intuitive or natural choice.

Overall, the models show strong predictive performance. Unfitting classifications often reflect Wikidata's unique structure rather than the models' limitations. However, there are cases where the models fail to predict the most fitting type, despite it being part of the candidate types, indicating areas for further improvement.

5.2. Efficiency

In this section, we describe the most time-consuming components of the pipeline, discuss their computational complexities, and explain the practical trade-offs made.

5.2.1. Data I/O

For many of the components, all existing `P31` and `P279` edges (and sometimes, more) needed to be available. For this purpose, all corresponding edges were stored in a sparse row (CSR) matrix format. Since internal strings and integers are quite large in Python, this was done using `csr_matrix` from the `SciPy` library, which utilizes a `C++` backend. This way, memory usage was reduced by a factor of ~ 10 compared to using a dictionary with strings as keys and sets of strings as values. Using Python integers instead of strings didn't make much of a difference, as Python dictionaries still have a lot of memory overhead per entry. Using `csr_matrix` on the other hand with `dtype=np.int8` only uses 1 byte per edge, and the final matrix only takes about 2 GB of memory.

5.2.2. Candidate Masking

A relevant step of our approach is the creation of the candidate-type mask. This step requires iterating over all entities N where a mask must be generated. Within this loop, the most time-consuming operation is the pre-selection of allowed type candidates, which involves a slightly modified Breadth-First Search (BFS), as explained in Section 4.1.1.

The worst-case time complexity of BFS is $O(V + E)$, where V is the number of nodes and E is the number of edges. Although typically only a small subset of the graph is traversed for any given entity, the worst-case time complexity remains $O(V + E)$. Therefore, the overall time complexity of mask creation is $O(N \cdot (V + E))$.

In practice, however, this step is much faster than the theoretical worst-case. Because Wikidata is a sparse, directed graph, most nodes have shallow branching, and, as mentioned earlier, the average number of candidate types per entity is only about 32.

5.2.3. Embedding Generation

Since storing millions of 512-dimensional feature vectors is infeasible, and dimensionality reduction methods slightly decreased performance, we generate the full 512-dimensional USE embeddings in real time during both training and inference. This requires loading all necessary descriptions from disk and passing them through USE. Since USE is fairly lightweight, this step is quite fast and doesn't take significantly longer than loading the RDF2Vec embeddings from disk.

Generating RDF2Vec embeddings to capture the structural position of entities involves two main steps: (1) random-walk generation and (2) training Skip-Gram (see Section 4.2.2). These steps took approximately three days on a 2021 M1 MacBook Pro with 16 GB RAM. The generated random walks were stored in a 100 GB file. The final embeddings occupied 26 GB of storage. In retrospect, the added computational cost was not justified, as our experiments showed no performance improvement over using only USE embeddings plus the mask.

For walk generation, 10 walks of length 8 (counting nodes and edges) were generated per entity. Each step in a walk involves retrieving the entity's neighbors by iterating over each edge type (`P31` and `P279`). If a node has an average out-degree of d and there are K edge types, this step is $O(K + d)$. Hence, a single walk of length w takes $O(w \cdot (K + d))$. Let total entities be N and total walks per entity be T, then the total time complexity is $O(N \cdot T \cdot w \cdot (K + d))$. In practice, T, w, and K are small constants, so the practical time complexity is closer to $O(N \cdot d)$.

5.2.4. Runtime Performance

Training time across the models on a M1 MacBook Pro (2021) with 16 GB of RAM varied significantly. While the FNN took a few minutes to train, the GNNs all took >1 hour. As is the case with the RDF2Vec embeddings, the added complexity of these models did not lead to a significant performance improvement, so training them was not worth the additional time cost.

Once trained, inference is very efficient for all models, as they have relatively few parameters and can be run quickly on a CPU. If the goal is to label the entirety of Wikidata, the speed bottleneck isn't model inference, but rather data loading. Labeling all of Wikidata requires batch processing, as loading all embeddings into memory at once is infeasible on most machines. The adjacency matrix can be kept in memory during this process, requiring only a single load. However, embeddings need to be loaded for each batch, which is the most time-consuming part of the pipeline, as it involves iterating through a file with more than 90 million lines. This also applies to the USE embeddings, which are generated in real-time but still require reading descriptions from disk. Using more efficient data storage could significantly reduce this time. However, since the processing time is still manageable, optimizing this was not a priority. This way, processing all of Wikidata only takes a few hours.

6. Future Work

The task of assigning a natural type to each Wikidata entity is not merely a technical challenge but also a conceptual one. There is an inherent subjectivity in type assignment. For instance, is Mt. St. Helens a mountain or a volcano? Is an excavator a machine or a tool? What one person considers a natural type may differ significantly from another's perspective, which makes evaluating model performance extremely challenging. Acknowledging this subjectivity was essential to approach the task with realistic expectations.

These challenges were reinforced by inconsistencies in Wikidata itself. Not all entities which should have the same natural type are connected uniformly, which required the generation of a large and diverse training set. Large language models allowed us to tackle the task in a brute-force manner by generating large, though noisy, training data. Despite the noise, the achieved results are promising.

Simple models, such as an FNN, achieved surprisingly high accuracy, while models intended to capture more complex relationships, like GraphSAGE, GAT, and R-GCN, underperformed. Notably, a surprisingly high accuracy was achieved even without using any node features, only relying on the probability distribution across candidate types. Both RDF2Vec and the USE embeddings proved highly effective at capturing relevant information for entity typing, but only the USE embeddings were able to significantly improve the performance beyond masking.

Nevertheless, the quality of the training data remains a limitation. The only way to refine

this data was to craft a carefully tailored system string to guide the LLM in assigning the most suitable type to each entity. However, while modern LLMs perform extremely well on a wide range of tasks, they currently fail to follow instructions consistently. Smaller LLMs, like Gemini Flash, struggle particularly with long contexts and frequently fail to follow all instructions. Even when instructions are followed, inconsistencies are prone to occur, resulting in a noisy training set.

However, due to the sheer scale of the training data, the models were able to generalize effectively, surpassing even the performance of the LLMs on the evaluation benchmarks.

Looking ahead, there are several promising directions for future work.

1. **Training Data Curation:** Improving the overall quality of the training data would likely yield the most significant improvements. Human annotators could, for example, remove overly general labels and map overly specific labels to more general ones. This human-in-the-loop curation process would yield a more refined dataset, likely leading to enhanced model performance, more in line with human expectations.

2. **Using a More Advanced LLM:** Generating the training data with a more advanced (but potentially more expensive) LLM might reduce noise and improve the overall quality of training data. As LLM capabilities continue to evolve, future LLMs could potentially produce significantly better training data with fewer inconsistencies. Eventually, LLMs could become cost-effective enough to label the entirety of Wikidata.

3. **Human Feedback on Benchmarks:** A straightforward approach to improve the evaluation benchmarks would be public feedback. A web interface could be used to gather feedback by asking individuals to select the most suitable type from a list of candidates. This would improve the quality of the evaluation dataset and make it less subjective.

4. **Proposing a Dedicated Wikidata Property:** Ultimately, a new Wikidata property could be introduced for an entity's natural type. While Wikidata allows for property proposals [36], past efforts to create such a property faced challenges [10], as achieving consensus for such a topic is notoriously difficult. Previous proposals eventually failed due to disagreements over whether types should be pre-selected (from a given ontology) or not, and about the appropriate level of granularity. For a proposal to succeed, careful consideration of these issues would be necessary, as well as a strong dedication to the proposal process.

In conclusion, this work successfully demonstrated the feasibility of automated prediction of natural types for Wikidata using large LLM-generated training sets, carefully selected models, and a simple but effective masking approach. These results provide a solid foundation for further research on entity typing in Wikidata, offering both benchmarks and baseline findings to guide and inspire future work.

7. Acknowledgments

I would like to thank...

- Professor Hannah Bast for being my first examiner
- Professor Abhinav Valada for being my second examiner
- Natalie Prange for being my supervisor

Bibliography

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities," *Scientific American*, May 2001.

[2] W3C, "Resource description framework (RDF)," Jan. 2025. `https://www.w3.org/RDF/`.

[3] W3C, "SPARQL 1.1 query language," 2025. `https://www.w3.org/TR/sparql11-query/`.

[4] L. Wittgenstein, *Philosophical Investigations*. Oxford: Basil Blackwell, 1953.

[5] E. H. Rosch, "Natural categories," *Cognitive Psychology*, vol. 4, no. 3, pp. 328–350, 1973.

[6] Wikidata, "Wikidata: Relation between properties in RDF and in Wikidata," Jan. 2025. `https://www.wikidata.org/wiki/Wikidata:Relation_between_properties_in_RDF_and_in_Wikidata`.

[7] Wikidata, "Wikidata help: Data types," Jan. 2025. `https://www.wikidata.org/wiki/Help:Terminology`.

[8] Wikidata, "Wikidata help: Properties," Jan. 2025. `https://www.wikidata.org/wiki/Help:Properties`.

[9] Wikidata, "Wikidata: Property constraints portal," Jan. 2025. https://www.wikidata.org/wiki/Help:Property_constraints_portal.

[10] Wikidata, "Wikidata: Property talk: P107," Jan. 2025. https://www.wikidata.org/wiki/Property_talk:P107.

[11] A. Gangemi, A. Nuzzolese, V. Presutti, F. Draicchio, A. Musetti, and P. Ciancarini, "Automatic typing of DBpedia entities," in *The Semantic Web – ISWC 2012*, (Berlin, Heidelberg), pp. 65–80, Springer, Nov. 2012.

[12] A. Tonon, M. Catasta, G. Demartini, P. Cudré-Mauroux, and K. Aberer, "TRank: Ranking entity types using the web of data," in *The Semantic Web – ISWC 2013*, (Berlin, Heidelberg), pp. 640–656, Springer, Oct. 2013.

[13] A. Tonon, M. Catasta, R. Prokofyev, G. Demartini, K. Aberer, and P. Cudré-Mauroux, "Contextualized ranking of entity types based on knowledge graphs," *Journal of Web Semantics*, vol. 37-38, pp. 170–183, 2016.

[14] M. M. Rahman, A. Takasu, and G. Demartini, "Representation learning for entity type ranking," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, (New York, NY, USA), pp. 2049–2056, Association for Computing Machinery, 2020.

[15] M. Eberts, K. Pech, and A. Ulges, "ManyEnt: A dataset for few-shot entity typing," in *Proceedings of the 28th International Conference on Computational Linguistics* (D. Scott, N. Bel, and C. Zong, eds.), (Barcelona, Spain (Online)), pp. 5553–5557, International Committee on Computational Linguistics, Dec. 2020.

[16] Y. Peng, T. Bonald, and M. Alam, "Refining Wikidata taxonomy using large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, pp. 5395–5399, ACM, Oct. 2024.

[17] Wikidata, "Wikidata help: Basic membership properties," Jan. 2025. https://www.wikidata.org/wiki/Help:Basic_membership_properties.

[18] Google Developers, "Gemini 1.5: Flash updates for Google AI Studio and Gemini API," Jan. 2025. https://developers.googleblog.com/en/gemini-15-flash-updates-google-ai-studio-gemini-api/.

[19] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," *CoRR*, vol. abs/1403.6652, 2014.

[20] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," *CoRR*, vol. abs/1607.00653, 2016.

[21] P. Ristoski and H. Paulheim, "RDF2Vec: RDF graph embeddings for data mining," in *The Semantic Web – ISWC 2016: 15th International Semantic Web Conference, Proceedings, Part I*, (Berlin, Heidelberg), pp. 498–514, Springer, 2016.

[22] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *International Conference on Learning Representations*, 2013.

[23] Wikidata, "Wikidata help: Description," 2024. https://www.wikidata.org/wiki/Help:Description.

[24] D. Cer, Y. Yang, S.-Y. Kong, and et al., "Universal sentence encoder for english," in *Proc. EMNLP 2018: Sys. Demos*, (Brussels, Belgium), pp. 169–174, ACL, 2018.

[25] C. Sammut and G. I. Webb, eds., *TF-IDF*, pp. 986–987. Boston, MA: Springer, 2010.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017.

[27] M. Iyyer, V. Manjunatha, J. Boyd-Graber, and H. Daumé III, "Deep unordered composition rivals syntactic methods for text classification," in *Proc. 53rd ACL & 7th*

IJCNLP (Long Papers) (C. Zong and M. Strube, eds.), pp. 1681–1691, Association for Computational Linguistics, Jul 2015.

[28] TensorFlow Hub, "Universal sentence encoder," Jan. 2025. https://tfhub.dev/google/universal-sentence-encoder/4.

[29] T. Boone-Sifuentes, M. R. Bouadjenek, I. Razzak, H. Hacid, and A. Nazari, "A mask-based output layer for multi-level hierarchical classification," in *Proc. 31st ACM Int. Conf. on Information and Knowledge Management (CIKM '22)*, pp. 3833–3837, 2022.

[30] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *CoRR*, vol. abs/1706.02216, 2017.

[31] PyTorch Geometric, "HeteroConv," Jan. 2025. https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.HeteroConv.html.

[32] S. Brody, U. Alon, and E. Yahav, "How attentive are graph attention networks?," *CoRR*, vol. abs/2105.14491, 2022.

[33] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[34] PyTorch Geometric, "Graph attention networks," Jan. 2025. https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.models.GAT.

[35] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017.

[36] Wikidata, "Wikidata: Property proposal," Jan. 2025. https://www.wikidata.org/wiki/Wikidata:Property_proposal.

A. LLM System String

```
SYSTEM_STRING = (
    "Objective:
    From a pre-selected list, choose the most natural,
    everyday-language type for a Wikidata item based on its label
    and description.

    Rules:
    - Your choice **must** be one of the provided
    pre-selected types.
    - Generally, choose the broadest category that still represents a
    natural and commonly understood everyday term (e.g. choose
    'Disease' over 'Infectious Disease', 'RNA' over
    'Non-coding RNA', 'Star' over 'Variable Star', etc.).
    - However, if a more specific category is a **very commonly
    recognized and understood** everyday category, choose it.
    Think about what a typical person would call it (e.g., 'Lake'
    rather than 'Body of Water', 'Village' rather
    than 'Human Settlement', etc.).
    - Again, avoid too much specificity (e.g. choose 'Surname'
    over 'Japanese Surname', 'Monument' over 'Heritage
    Monument', etc.).
    - Generally speaking, a good type is short and intuitive, while
```

```
    a bad type is long and overly specific.
    - Return only the type (with label and QID).
    - Do not output JSON.

    Examples:
    - Berlin -> City
    - Albert Einstein -> Person
    - T-Shirt -> Clothing
    - Germany -> Country
    - Carbon Dioxide -> type of chemical entity
    - Breaking Bad -> Television Series
    - Jazz -> Musical Genre
    - Sagrada Família -> Church
    - Green Tea -> Drink
    - FC Bayern Munich -> Sports Club (Football Club would be too
    specific)

    Important: A type as long and specific as e.g. 'civil parish in
    Ireland' will **almost never** be a good choice (just 'civil
    parish' would be much better). Remember, a type should be
    short, intuitive, and represent a commonly understood category."
)
```