Strongly Typed Data for Machine Learning

London, March 2022

James Fletcher, Vaticle
james@vaticle.com

Abstract

Heterogenous data holds significant inherent context. We would like our machine learning models to understand this context, and utilise this ancillary but critical information to improve the accuracy and versatility of our models.

In this article we explore how we can systematically make use of context in Machine Learning. We investigate the knowledge modelling techniques which, applied with the right Machine Learning strategies, give us a promising approach for robustly leveraging value from heterogeneous data in large knowledge models. We aim to create an abstract process that encodes contextual knowledge that can benefit any Machine Learning method we choose to use over a knowledge base. Including, for example, graph learning methodologies like our Knowledge Graph Convolutional Networks (KGCNs).

1 What is Strong Typing?

As an example of weak typing vs. strong typing we can consider JavaScript and Java. JavaScript allows any variable to accept any data type, whereas in Java you must be very specific and not violate the type that a variable holds on to (this is static typing). At first glance Java's strong typing sounds like an inconvenience, and you'd probably be right. However, it also saves you from a wide array of hard-to-catch bugs that are impossible to sneak past the Java compiler, but could bite you in JavaScript very easily. TypeDB is more like Java than JavaScript, but for your data.

We're talking about machine learning, so why is strong typing relevant to us? No-one talks about this in relation to machine learning; With this article we want to start that conversation. Why should we care? Well, we care about having the best features for our models to consume. Adding type information ensures that every piece of data we encounter is guaranteed to have a predefined type. We can rely on these types as additional and contextual features for all of the data our model consumes.

2 ML Problems Solvable with Strong Typing

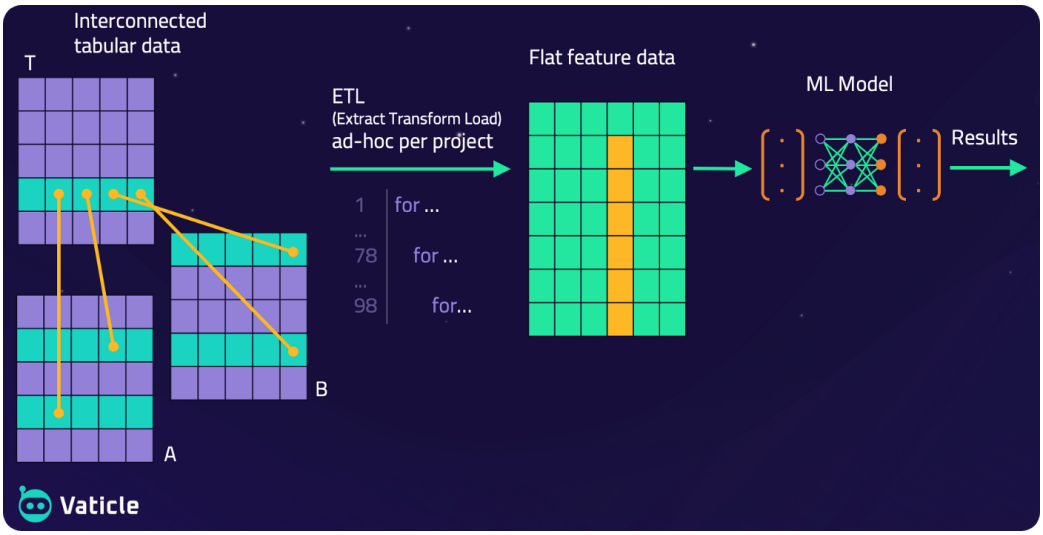

Figure 1 The flow of data from tables to model. We'd like to avoid flat feature data.

2.1 Only ND-arrays Fit Into Learning Pipelines

Your data is often tabulated, but that tabular data is interrelated, so in reality it represents a graph. Representing that graph data natively would be great, as our ML models could benefit hugely from the extra context of knowing which rows are related to which other rows and how — but here's the catch: you need N-dimensional arrays for ML model inputs, not graphs. That's true of all of the major ML frameworks, including TensorFlow and PyTorch.

2.2 Data Lacks Context and Consistency

When we flatten our data to build features we are prone to omitting context — which table is the example coming from? What is that example related to? What is the nature of each of those relations (the difference between being a buyer and a seller in a transaction)? All of this is difficult to capture. Combine that with how error-prone data retrieval can be (e.g. mixing up foreign keys) and how dirty our data can be. We can see that we're missing out on something.

3 How Does Strong Typing Help?

3.1 Inter-related Tabular Data is Really a Graph

Assuming your tabular data has inter-related rows and therefore is actually a graph, why not represent it that way? Well, the first barrier is working with graph data for our machine learning. But, maybe, that's actually possible or even enormously beneficial. This potential can be realised by graph ML methods. We'll come back to that shortly...

You might be wondering why working with a graph has anything to do with strong typing. A strongly typed program in Java is a graph: a set of classes with arbitrarily directed dependencies upon one another. By analogy, strongly typed data is also a graph, and a rigorous one.

3.2 Strong Typing Adds Context Throughout

By enforcing strong typing we ensure we have a clear structure defined via a schema. TypeDB checks all inserted data against this schema for validity, re-

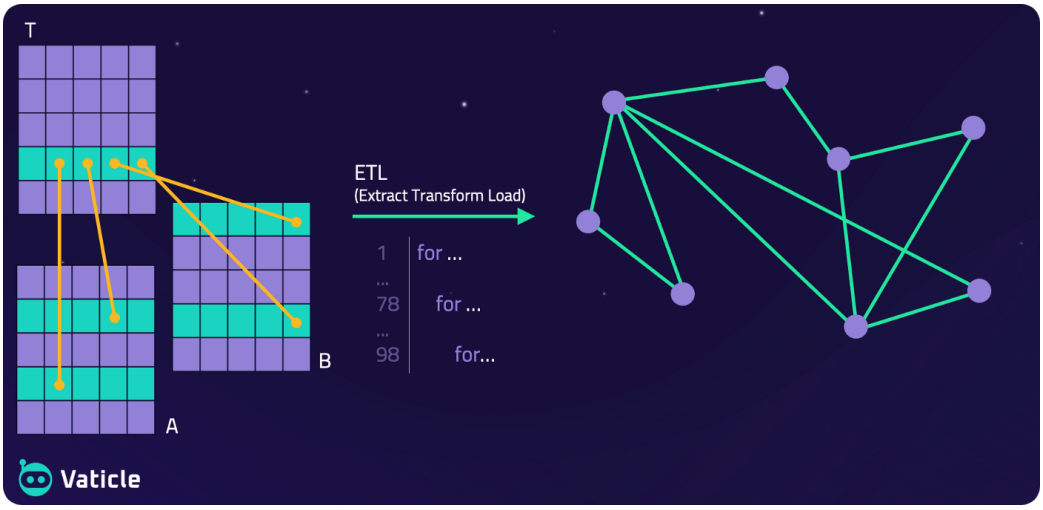

Figure 2 Transforming tabular data with inter-related columns into a graph with an ETL script.

jecting any logically invalid values or relations (and many more constraints). This gives us concrete assurance that all of our data is both typed (within a rich type hierarchy) and logically consistent; a great platform on which to develop context-aware learning models.

3.3 Graph ML

There's been a lot of attention on graph learning in the ML community, and there's growing support and libraries out there for learning over graphs. This is a big contrast to when we were first actively researching this space as it was breaking, in 2018, and received a strong reaction to our work on KGCNs in these posts:

- Knowledge Graph Convolutional Networks: Machine Learning over Reasoned Knowledge
- KGCNs: Machine Learning over Knowledge Graphs with TensorFlow

Our approach built on Graph Nets from DeepMind, and at the core of these methods is message passing. Message passing is our preferred flavour of graph learning here at Vaticle. Its most attractive quality is that it learns over native graphs, with the implications that:

- your graph structure stays intact during learning.
- the learner can make findings from the context of the graph as naturally as possible.

Move to the present day and the most notable library in the space is PyTorch Geometric, which has implementations for a wide range of published graph learning algorithms, many of them based on message passing. We can see that dealing with natural graphs can definitely be to our benefit, and that there are well-established ways and pre-built pipelines to accept graph data into TensorFlow and PyTorch!

3.4 Creating a TypeDB Model

Now that we're sufficiently motivated to improve our machine learning using TypeDB, let's look at creating a database. To create a TypeDB database we first need to create a schema for it, written in TypeQL. Check out the schema docs to find out more. Here's a quick example to give you a flavour:

```
define
person sub entity,
  plays teaching:teacher,
  plays transaction:buyer,
  plays transaction:seller;

transaction sub relation,
  relates buyer,
  relates seller;

teaching sub relation,
  relates teacher;
```

Nodes and edges (that our tabular data described) should be modelled as entities and relations. This is not a simple mapping of nodes to entities and edges to relations. Users need to craft a well-thought-out knowledge model to harness the power of this system.

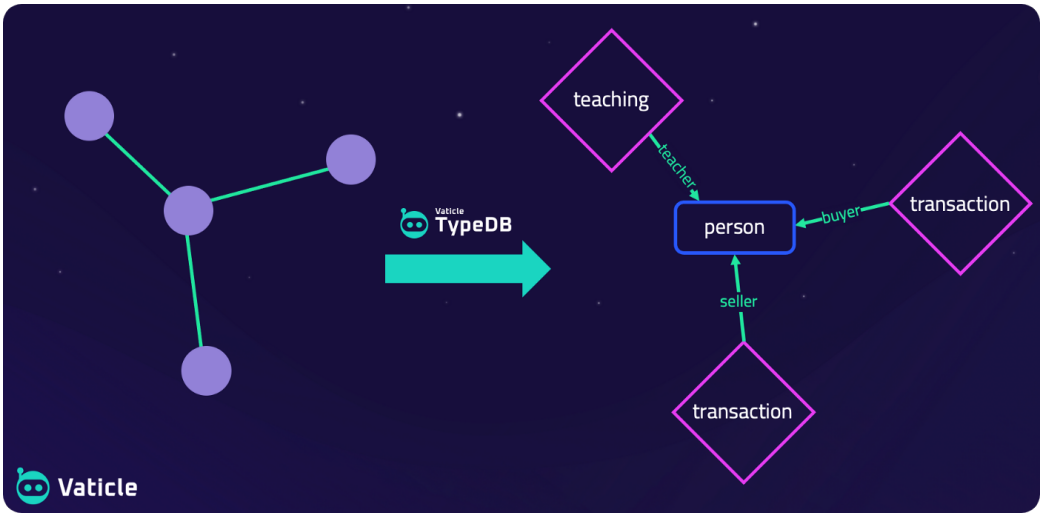

Figure 3 Properties of a node are modelled as attributes. One-to-many properties are trivial to model.

We'll represent properties of each node (properties of entities or relations) using attributes in TypeDB. This is typically straightforward. Make sure to observe how we can have multiple attributes of the same type, no more do we need to hack the DB with field names like:address1, address2 (nor create a whole new relation/node/join table to achieve this).

For the database aficionados out there these aspects, in particular the attribute model, make TypeDB first normal form! This makes it the most natural way to represent knowledge. Once you have a model you can load your data with your own ETL, or consider using this open-source tool that Bayer, the pharmaceutical giant, built to easily, quickly and fault-tolerantly ingest their tabular data into TypeDB ready for ML work. To read more on TypeDB Loader see this post.

3.5 From DB Queries to ML Input

Having crafted a schema and loaded data, how do we interface with ML? The majority of graph learning libraries out there, at least PyTorch Geometric and Graph Nets, take in Python NetworkX graphs. So how can we integrate with these libraries and more besides? Trying to transform to NetworkX, we realise that the TypeDB graph can be represented as a node-edge graph where:

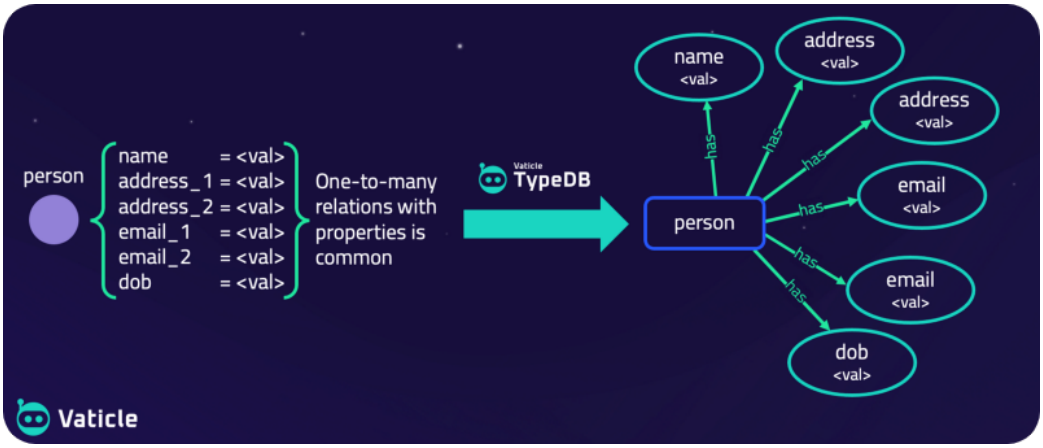

Figure 4

- Each node has a `type`
- Each edge has a `role type`, or is a `has` edge
- A node has a single value if it represents an `attribute`

Does this seem pointless? We did all that modelling effort to end up back at a simple node edge representation again? Not so, we gained these benefits:

- Enforced types throughout
- Enforced a maximum of one value per node of a known value type (string/boolean/long/double/datetime)
- Enforcement is via data validation as the data was inserted into TypeDB (catches any malformed data according to your schema)
- New facts are inferred by rules, which we haven't explored in this article

Rather than export the whole database to an in-memory NetworkX graph we make selective queries to TypeDB. We assemble the results into a graph, handled for us by KGLIB. This gives us highly scoped subgraphs to work with. You can think of each subgraph as equivalent to an example or sample in traditional non-graph ML.

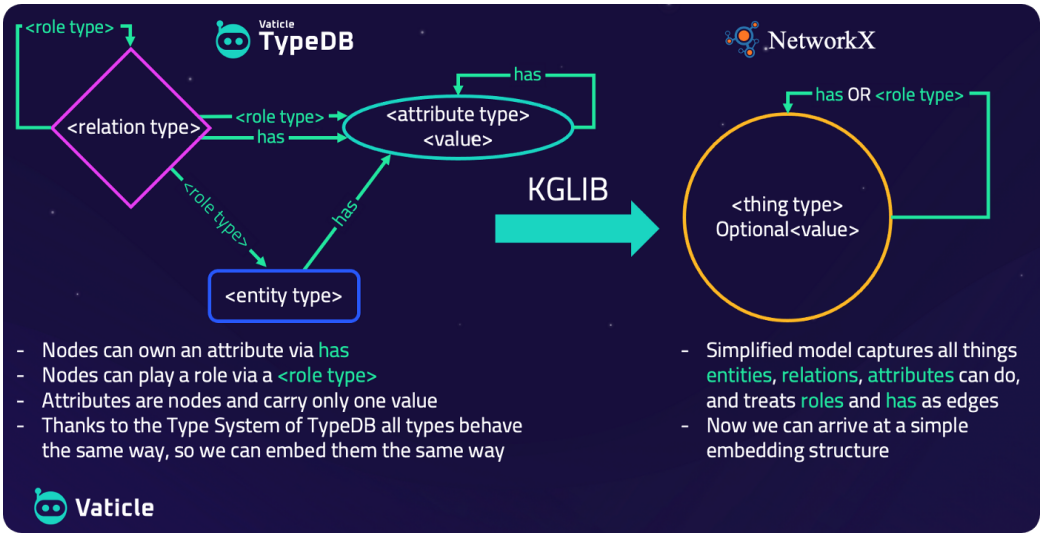

Figure 5

3.6 Graph Embedding

KGLIB is the ML library for TypeDB. It includes the means to embed our NetworkX graphs to build features for each node and edge. Presently we use these features as input to our KGCN implementation, and we hope to make this workflow generically useful for other graph ML algorithms. We have types to embed for things and roles, and we'll count `has` as a type. We also have optional values to embed. The value embedding is less interesting, so we'll omit that here.

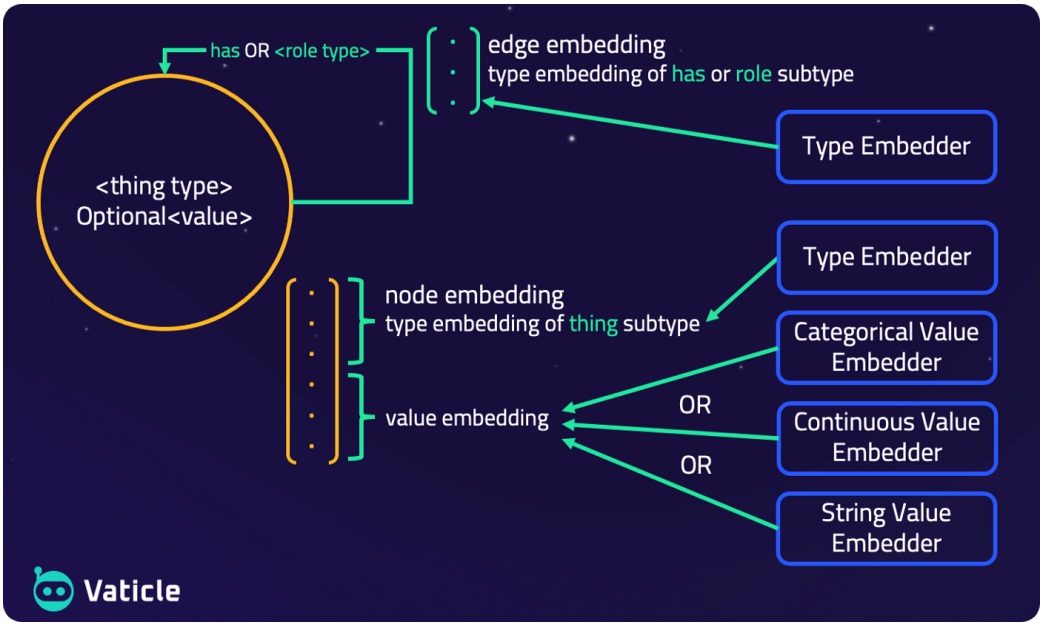

Figure 6 Embedding types and values for a NetworkX representation of TypeDB data.

3.7 Type Embedding

Most interesting is the Type Embedder. This indexes all of the types in the schema and uses this index with the `embed` module available in TensorFlow and PyTorch. This maps all of our types into an embedding space. This embedding step is part of the end-to-end model, therefore the embedding space is learned according to the objective function of your learner! This is implemented in KGLIB.

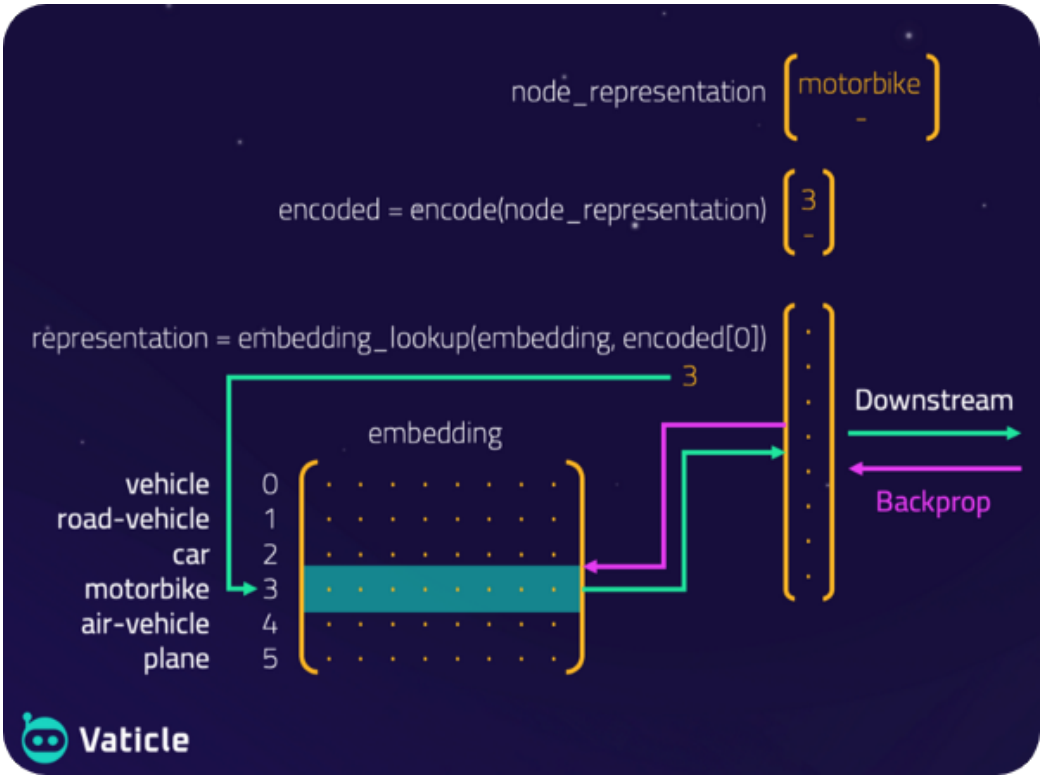

Figure 7 In this example we find motorbike has index 3, so that points us to a particular index of the embedding space built to yield the embedding for the motorbike type.

3.8 Future Work for Type Embedding

More exciting still, we would like to leverage type hierarchy in our embeddings. In this example we would like the learner to be able to benefit from the knowledge that both cars and motorbikes are of type `road-vehicle` and `vehicle`. So to embed the type `motorbike` this means taking the embeddings from all of its parent types and combining them into a single embedding. In the hierarchy we will always have a linear sequence of parent types as we walk the tree. In this case: `entity`; `vehicle`; `road-vehicle`; `motorbike`.

We see that this is an arbitrary length sequence, so models that are appropriate for NLP are also applicable here, think RNNs and the like. But perhaps more interesting (and perhaps more fashionable and effective) would

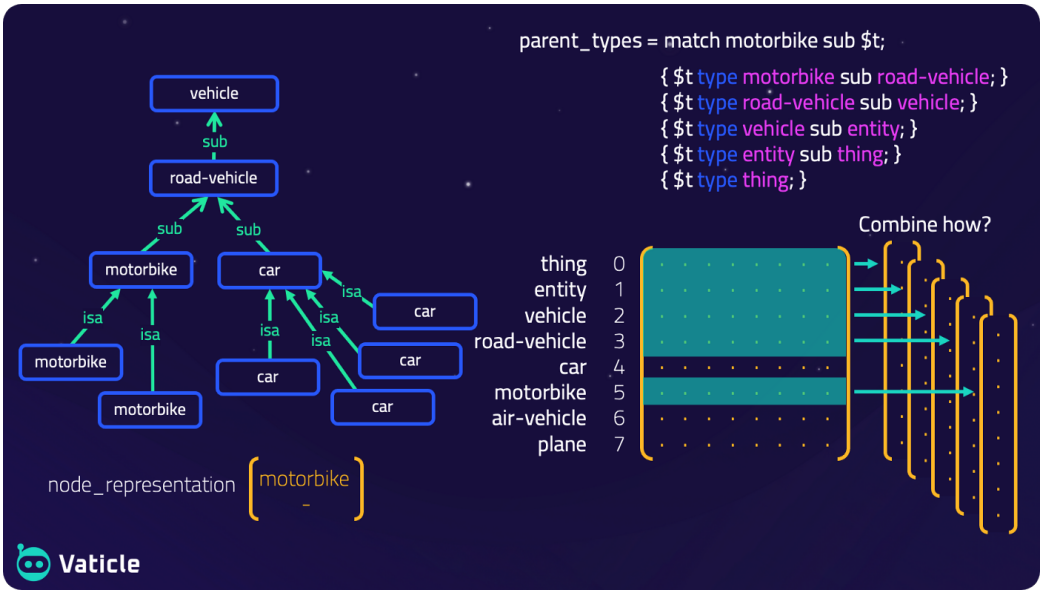

Figure 8

be to use an attention model to pick the contribution of each parent type to the final embedding. This isn't implemented in KGLIB yet, but is on the roadmap.

4 The Pipeline

What we have achieved here is to circumvent the issues that bad data representations cause for ML. In doing so, we are able to use very advanced graph learning methods. These methods can make use of the rich context we have added using a type system to make much more informed predictions.

Overall we see that this pipeline has some completely generic components from KGLIB which is achievable due to the abstraction we add by using TypeDB. It means that here at Vaticle we can push forward on the Encoding and Embedding tools that make it much easier to get a graph learning pipeline up and running for your TypeDB instance. Users need to:

1. Create a TypeDB schema and insert their data into TypeDB (take a look at TypeDB Loader for a quick solution).

2. Select queries of interest to be used as subgraphs for learning

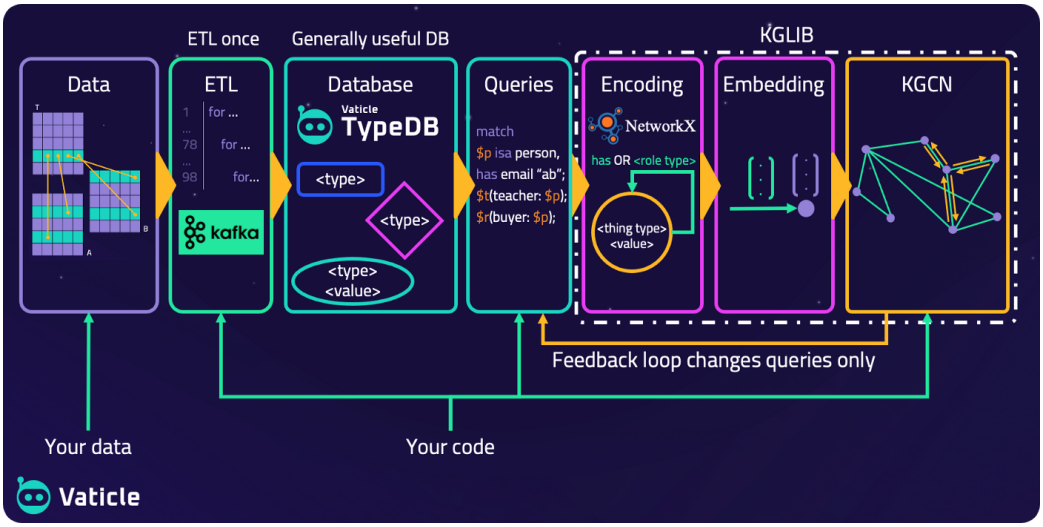

Figure 9

3. Pick a graph ML method to use. Currently only one is supported, our own KGCN implementation, however soon we will interface with PyTorch Geometric and Graph Nets so users can pick from the vast selection of pre-built models there.

5 About KGLIB

KGLIB started out as our ML research space for exploring graph learning. Now that the field is more established we are transitioning the repo into a dev tooling library to streamline working with TypeDB for ML. This includes the elements of the pipeline listed above. On the roadmap is to include integration with PyTorch Geometric and Graph Nets. Visit the repo!

6 Conclusion

We have demonstrated the potential that a strongly typed knowledge graph holds for machine learning: How it can provide context to allow a learner to really understand and capitalise upon contextual and connected data.

We have shown that a this approach is implementable using TypeDB as the strongly typed knowledge base and using any of the wide range of

graph learning approaches being developed by the machine learning community. This stack is a natural solution to tasks such as link prediction, entity classification and attribute prediction.