**Project: Semantic Loss of Tabular Data Serialization**

**Databases:**

A database is a structured system for storing, managing, and retrieving data efficiently. It is designed to hold organized information in a way that enables easy access, management, and update.

* **Relational databases**

In a relational database data is organized into tables that hold information about each entity and represent pre-defined categories through rows and columns. This structured data is both efficient and flexible to access. Example:  SQL Server, Azure SQL, MySQL, PostgreSQL, and MariaDB.

* **Non-relational databases**

Non-relational databases, store unstructured or semi-structured data. They use a storage model that's optimized for the specific requirements of the type of data being stored. Non-relational databases allow for larger sets of distributed data to be accessed, updated, and analyzed quickly. Example: MongoDB, Azure Cosmos DB, DocumentDB, Cassandra, Couchbase, HBase, Redis, and Neo4j.

* **In-memory databases and caches**

In-memory databases are commonly used to store copies of frequently accessed information like pricing or inventory data. This is known as caching. When you cache data, you store a copy of it in a temporary location so that it loads faster the next time it's requested

**Serialization:**

Data serialization is the process of converting data into a format that can be easily saved to a storage medium or transmitted across a network.

A blue arrow pointing to a blue text

Description automatically generated

**Schema linking:**

Schema linking in a database refers to the process of connecting different parts of the database schema to show how they relate to each other. It helps to ensure that data in one table can be correctly linked to data in another table.

**A diagram of a number of data

Description automatically generated with medium confidence**

Transform the original data into question & answer prompts. For example:

A screenshot of a computer

Description automatically generated

Feed the prompts into the LLM and use output answer accuracy as the metric to compare the semantic loss of different data types against the original data. The ground truth is original data.

**Progress 1 (Completed):**

* Collected datasets.
* Designed prompts for different data types.
* A table with numbers and text

  Description automatically generatedDeveloped data processing code.

A table of data with numbers and letters

Description automatically generated

**Solution:** Utilize various LLM embeddings to directly compare the semantic differences across different data types, using a stringed data frame as the ground truth.

**Tabular Data:**

* Collected tabular test data.
* Developed data processing code.
* Deployed different transformer structures (encoder, decoder, encoder & decoder) for testing.

**Ground truth:**

A computer screen with white text

Description automatically generated

**Different data types:**

A screenshot of a computer

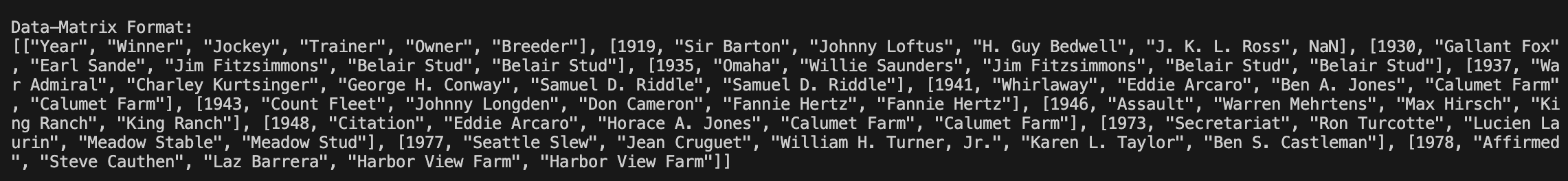
Description automatically generated

A black screen with white text

Description automatically generated

A screenshot of a computer

Description automatically generated



A screen shot of a computer

Description automatically generated

**Results:**

A screen shot of a black and white screen

Description automatically generated

**Conclusion:**

Compared to other models, GPT embeddings better preserve the original semantic meaning than BERT and T5. The results indicate that the GPT embedding is less sensitive to different data types, demonstrating its robustness.

**PDFs:**

* Tested additional raw data types such as PDFs.
* Developed corresponding data processing code.

**Ground Truth：**



**Different data type：**

A screen shot of a computer screen

Description automatically generated

A screen shot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer screen

Description automatically generated

**Results:**

A screenshot of a computer

Description automatically generated

**Conclusion:**

In the experiments involving table conversion and PDF conversion, it was found that GPT embeddings consistently preserve the original semantic meaning better. Additionally, data converted into DFLoader and Linearized data formats experiences less semantic loss compared to other formats.

**ER model：**

* Tested additional raw data types such as ER model.
* Developed corresponding data processing code.

**Ground truth：**

A computer screen with white text

Description automatically generated

**Different data type：**

**A computer screen shot of a program

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**Results:**

**A screenshot of a computer screen

Description automatically generated**

**PDF-ER Mixed:**

**Ground Truth:**

**A screenshot of a computer

Description automatically generated**

**Different data type：**

**A screen shot of a computer screen

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**A screen shot of a computer screen

Description automatically generated**

**Results:**

**A screen shot of a black screen

Description automatically generated**

**Tokenization:**

**Pretokenization:**

Pretokenization can be as simple as space tokenization, e.g. GPT-2, RoBERTa. More advanced pre-tokenization include rule-based tokenization, e.g. XLM, FlauBERT which uses Moses for most languages, or GPT which uses spaCy and ftfy, to count the frequency of each word in the training corpus.

**Byte-Pair Encoding (BPE):** GPT2，RoBERTa （100M-200M）

After pre-tokenization, a set of unique words has been created and the frequency with which each word occurred in the training data has been determined. Next, BPE creates a base vocabulary consisting of all symbols that occur in the set of unique words and learns merge rules to form a new symbol from two symbols of the base vocabulary. It does so until the vocabulary has attained the desired vocabulary size. Note that the desired vocabulary size is a hyperparameter to define before training the tokenizer. For example:

("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

the base vocabulary is ["b", "g", "h", "n", "p", "s", "u"]. Splitting all words into symbols of the base vocabulary, we obtain:

("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)

BPE then counts the frequency of each possible symbol pair and picks the symbol pair that occurs most frequently. In the example above "h" followed by "u" is present *10 + 5 = 15* times (10 times in the 10 occurrences of "hug", 5 times in the 5 occurrences of "hugs"). However, the most frequent symbol pair is "u" followed by "g", occurring *10 + 5 + 5 = 20* times in total. Thus, the first merge rule the tokenizer learns is to group all "u" symbols followed by a "g" symbol together. Next, "ug" is added to the vocabulary. The set of words then becomes

("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)

BPE then identifies the next most common symbol pair. It’s "u" followed by "n", which occurs 16 times. "u", "n" is merged to "un" and added to the vocabulary. The next most frequent symbol pair is "h" followed by "ug", occurring 15 times. Again the pair is merged and "hug" can be added to the vocabulary.

At this stage, the vocabulary is ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"] and our set of unique words is represented as

("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)

Assuming, that the Byte-Pair Encoding training would stop at this point, the learned merge rules would then be applied to new words (as long as those new words do not include symbols that were not in the base vocabulary). For instance, the word "bug" would be tokenized to ["b", "ug"] but "mug" would be tokenized as ["<unk>", "ug"] since the symbol "m" is not in the base vocabulary. In general, single letters such as "m" are not replaced by the "<unk>" symbol because the training data usually includes at least one occurrence of each letter, but it is likely to happen for very special characters like emojis.

**WordPiece：**BERT，Electra（100M）

WordPiece is very similar to BPE. It irst initializes the vocabulary to include every character present in the training data and progressively learns a given number of merge rules. In contrast to BPE, WordPiece does not **choose the most frequent symbol pair, but the one that maximizes the likelihood** of the training data once added to the vocabulary.

Referring to the previous example, maximizing the likelihood of the training data is equivalent to finding the symbol pair, whose probability divided by the probabilities of its first symbol followed by its second symbol is the greatest among all symbol pairs. *E.g.* "u", followed by "g" would have only been merged if the probability of "ug" divided by "u", "g" would have been **greater** than for any other symbol pair. Intuitively, WordPiece is slightly different to BPE in that it evaluates what it *loses* by merging two symbols to ensure it’s *worth it*.

**SentencePiece:** T5, XLNet （100-200M）

SentencePiece treats the input as a raw input stream, thus including the space in the set of characters to use. It then uses the BPE or unigram algorithm to construct the appropriate vocabulary. Suppose we have the sentence "This is a test sentence." SentencePiece treats the entire sentence as a continuous character sequence: "T h i s \_ i s \_ a \_ t e s t \_ s e n t e n c e ." All characters, including spaces, are initially part of the vocabulary. SentencePiece then iteratively merges the most frequent character pairs. For example, it might first merge "s" and " " (space) because they occur frequently together. Through several iterations of merging, the final vocabulary might include tokens like: ["T", "h", "i", "s\_", "a", "t", "e", "n", "c", ".", "en", "t\_", "se", "nt", "ce"]. Using the final vocabulary, the sentence would be tokenized as: "T h i s\_ i s\_ a\_ t e s t\_ se nt en ce .".

**Result:**

**Table  
A screenshot of a computer

Description automatically generated**

**ER  
A black and white screen with numbers

Description automatically generated**

**PDF**

**A black and white screen with numbers

Description automatically generated**

**Different Model Parameter:**

**A black and white screen with numbers

Description automatically generated**

**Observation Results：**

1. When the model sizes are similar, GPT embedding is less sensitive to data format conversion.

2. When using GPT embeddings, the similarity between DFLoader and Linearized data is generally the highest.

3. As the number of model parameters changes, the sensitivity of the model to data conversion becomes uncertain.

4. Among different tokenization methods, WordPiece performs the worst.

5. After integrating different types of data, the model's sensitivity to data conversion decreases.

**Potential Questions：**

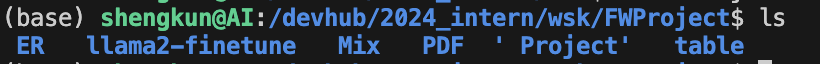
1. What model parameter size and tokenization method are the least sensitive to data format conversion?

2. How does the similarity change after integrating data from multiple different sources? How does the proportion affect the similarity?

3. How does the embedding similarity change after adding natural language instruction?

4. How does embedding similarity affect the model's output? Will higher similarity lead to more accurate final results?

**List:**



1.ER: Including test data and test code for ER model.

A screenshot of a computer

Description automatically generated

A blue rectangular sign with white text

Description automatically generated2.Llama2-fintune showcase.

3.Including Mixed data test and large parameter model test.

A screen shot of a computer

Description automatically generated

4.Including PDF test.

A black screen with green text

Description automatically generated

5. Including table test.

A computer screen shot of a computer program

Description automatically generated