Automating the Generation of a Functional Semantic Types Ontology with Foundational Models

Sachin Konan
Two Sigma Investments, LP
sachin@twosigma.com

Scott Affens
Two Sigma Investments, LP
scott.affens@twosigma.com

Larry Rudolph
Two Sigma Investments, LP
rudolph@csail.mit.edu

Abstract

The rise of data science, the inherent dirtiness of data, and the proliferation of vast data providers have increased the value proposition of Semantic Types. Semantic Types encode contextual information onto a data schema, informing the user about the definitional meaning of data, its broader context, and relationships to other types. Previous work focuses on the recognition of statically-defined types, but what happens when a type-set isn't known apriori and how do we connect Semantic Types to downstream use-cases? `FSTO`-*Gen* addresses these questions by leveraging LLMs to generate Functional Semantic Types that include normalization, validation, and casting code to efficiently automate human-intensive data tasks.

1 Introduction

Onboarding datasets at scale is human-intensive because recognizing semantics about how data was generated, its bounds or other idiosyncrasies is critical to how it is processed and eventually used. Normalization requires understanding the form/representation as it is ingested, as well as the target form used by other datasets or people in an organization. Much of the effort involved in semantic understanding can be automated when values are typed with something richer than the basic primitives (e.g. integer, float, or strings types). In fields like finance, medicine, or broadly-spanning AI systems, new data is constantly being added, and automation can defray some of the ingestion cost.

The underlying pain-point in data onboarding stems from humans inconsistently naming tables and columns without knowing who may use them. Even when using data from a high-quality source [25], the consumer often invests significant human capital to develop to normalize and map data to a canonical representation as there are many challenges associated with normalizing columnar tables at scale. In addition to handling null/not-a-number values, inconsistent data formats, or out-of-distribution values, it is critical to understand the units of the values, which maybe implicit (e.g. dollars when giving the price of housing in the US), written in prose (e.g. revenue may be in millions of dollars as noted in the text of a 10k filing), or easily inferred (e.g. a summer temperature of 100 degrees is Fahrenheit and not Kelvin or Celsius). Additionally, to compare, join, or group values from columns in different tables, it may be necessary to recast values (e.g. convert a 12-hour am/pm time value to a 24-hour clock value or a country code to a country name). Identifying these semantics is behind the reliance on humans during onboarding.

We introduce *Functional Semantic Types* (`FST`) to automatically annotate columnar data with Semantic Types and provide a library of functional attributes to normalize, validate, and cast real data values. `FST`s are placed in a synthetically generated ontology, which directly maps columnar datasets to their hierarchically organized `FST`s (this process is `FSTO`-*Gen*). The generation of `FST`s relies upon Large Language Models (LLMs)[5] to assign and generate a Python class definition that contains semantic details and functional characteristics about the data. By their training breadth, LLMs offer a general solution for entity recognition [8], because they can leverage the distributional properties/real values of data, tabular metadata, and data dictionaries to construct more accurate type annotations. Furthermore, LLMs have shown the ability to generate code, and therefore are capable of transforming semantic context into functional attributes.

Our work takes advantage of the natural hierarchical organization of tabular data. We refer to a collection of tables as a product. The information within a product is assumed to be initially constructed, maintained, and labeled by the same community of interest. As such, columns and tables, as well as context, are correlated. Our system is likely to generate the same `FST` for multiple columns belonging to the tables within a product. The automatic generation of semantic types saves human labor, and even more so when multiple columns are assigned the same type. The system verifies this by merging a subset of values from these columns and checking that they all pass the same validation test.

At the final stage, we identify common FSTs across different products. First semantic types with the same name are agglomerated. Then all the uniquely named FSTs are turned into a graph by finding semantically similar FSTs in representation space and generating *cross-type-cast* functions to transfer data values between FSTs. Human verification shows that this first attempt identifies communities of semantically-*identical* entities.

There have been many other efforts to automatically assign semantic types to columnar data, but with some major differences (see Section 2). They assume the existence of an ontology or knowledge graph, do not build a bespoke ontology that is used for cross-type casting, and most of all they do not generate the functions that define the Semantic Type. Hence, the contributions of this work are:

- Automatically generating *Functional Semantic Type* Python class definitions with fields and functional methods that characterize, transform, and validate columnar data values.
- Aggregating commonly named FSTs across products, and generating conversion code.
- Demonstrating success in real-world collections of data and evaluating the functionality and correctness of each of these ontologies.
- Showing that generated ontologies have utility in downstream data discovery, joining, validation, and normalization applications.

2 Related Works

The association of columnar tables with entities has been previously treated as a multi-class prediction problem over some user-defined distribution of entity types/properties. Methods such as Sherlock[12], SATO[29], DoDuo[26], and TableGPT[8] use 78 semantic types described by the T2Dv2 Gold Standard[4] which matches properties from the DBpedia ontology with column headers from the WebTables corpus. AutoType[28] made predictions over 112 manually procured types that spanned different industries. However, the types used in these works are often too broad for industry-specific datasets (e.g. in finance, EBITA is a commonplace term, but missing from DBPedia[1] and WordNet[6] knowledge graphs).

We summarize the related works along several dimensions (although TableGPT[8] and Foundational Models[21] cover nearly all of these). *Table Question and Answer:* Table Cell Identification[27], Semantic Parsing[22], TabFact[3]. *Row-to-Row Transformation:* TDE[9]. *Entity Matching Between Rows:* Ditto[17], Deep Entity Matching[20], Auto-EM[31]. *Schema Matching Between Tables:* Valentine[14], SMAT[30]. *Data Imputation:* DataWig[2], Eracer[18], IMP[19], HoloClean[24].

A trend that spans most of these works is the success of LLMs as natural language processing engines for directly operating on real data values. Archetype[7] showed that LLMs are performant zero-shot annotators of semantic types across various domains. Additionally, LLMs have shown success in code generation tasks [16], specifically, for data processing and ingestion coding [11, 15]. Our work builds on these efforts but is unique in that it directly attaches any functional normalization during the entity recognition process (encapsulated in a FST), and hierarchically groups FSTs to build an ontology that identifies semantically identical entities across products. Additionally, unlike previous work in semantic type annotation, we do not specify the set of types beforehand; the aggregation of generated types across a universe of data becomes the type set (App. A.7).

3 Problem Formulation

Our FST system is applied to a *universe* of data, hierarchically composed of *tables* and *products*. A product consists of at least one columnar table. Columns, tables, and products all have labels. The tables within a product are assumed to have some informational similarity. In addition, our set of FSTs can either be a subclass of DatasetSemanticType (Fig. 1), or of GenericSemanticType (Fig. 2). DatasetSemanticType FSTs correspond to the standard types of data: numeric (with/without units), boolean, strings, and categorical. Examining the values in a column, as well as its relation to other columns, is all that is needed to make this type of assignment. In addition to a human-readable name, its definition includes descriptive characteristics about the type's semantics, domain, and example values, as well as a func-

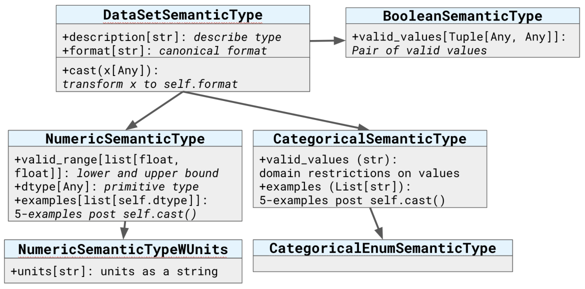

DataSetSemanticType
+description[str]: describe type
+format[str]: canonical format
+cast(x[Any]):
transform x to self.format
BooleanSemanticType
+valid_values[Tuple[Any, Any]]:
Pair of valid values
NumericSemanticType
+valid_range[list[float, float]]: lower and upper bound
+dtype[Any]: primitive type
+examples[list[self.dtype]]:
5-examples post self.cast()
CategoricalSemanticType
+valid_values (str):
domain restrictions on values
+examples (List[str]):
5-examples post self.cast()
NumericSemanticTypeWUnits
+units[str]: units as a string
CategoricalEnumSemanticType

Figure 1: The subclasses of `DatasetSemanticType` . Each class has specific and inherited instance fields, as well as an implementation of the `cast()`.

tional transformation from raw data to normalized, types. `GenericSemanticTypes` collate identically-named `ColumnSemanticTypes` across products by applying a standardized normalization procedure that undergoes self-validation (see Fig. 2). The `FSTs` generated at the table and product levels are called `T-FSTs` and `P-FSTs` (each a subclass of `DatasetSemanticType`), while those generated at the universe level are called `G-FSTs` (subclass of `GenericSemanticType`).

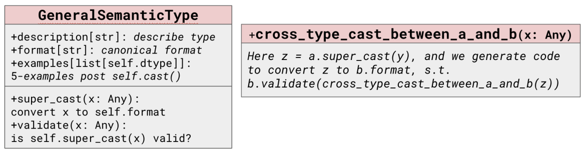

GeneralSemanticType
+description[str]: describe type
+format[str]: canonical format
+examples[list[self.dtype]]:
5-examples post self.cast()
+super_cast(x: Any):
convert x to self.format
+validate(x: Any):
is self.super_cast(x) valid?
+cross_type_cast_between_a_and_b(x: Any)
Here z = a.super_cast(y), and we generate code to convert z to b.format, s.t. b.validate(cross_type_cast_between_a_and_b(z))

Figure 2: Class definition of `GenericSemanticType` and the `cross_type_cast()` method.

Goal: Given a table, extract the underlying semantic entities (if any) per column and generate a `DatasetSemanticType` with a descriptive class name, instance fields, and a `cast()` that transforms a single columnar value to the format dictated by the class. Commonly named `FSTs` across products are used to generate a subclass of a `GenericSemanticType` that merges the semantics of all the input `DatasetSemanticType` definitions, consolidates their transform logic into a single `cast()`, and evaluates the correctness with `validate()`. Some `G-FSTs` may be convertible, so `cross_type_cast()`'s transform their values. Unlike traditional ontologies, the result is a synthetic ontology where *the hierarchy between* `T-FST`*,* `P-FST`*, and* `G-FST` *represent increasing layers of transformational generality*. Edges between `G-FSTs` represent a knowledge graph, but whose edges are not just semantic relationships, but functional transformations.

4 FST Ontology Generation (`FSTO`-*Gen*)

Step 1: `table` → `T-FST` - For each table in our universe, an LLM is provided with a serialized format of the table (App. A.1.1) and identifies the *subset* of columns that correspond to semantic entities that might be more broadly applicable to other datasets. For each identified column, the LLM generates the corresponding `T-FST` subclasses and a mapping from column name to subclass. Abstract syntax trees are used to parse the output string and store any class definition and this mapping dictionary. In our experiments, the LLM tends to create identical subclass names (but not always identical fields) for columns in the same `product` (App. A.1.2).

Step 2: `T-FST` → `P-FST` (`product`) - There exists many identical `T-FSTs` within a `product`, so we agglomerate identically-named ones into a single `P-FST` (App. A.2). For a given `T-FST` group, the unique columnar values spanned by the `T-FSTs` are aggregated and tested via the `cast()` to assess the number of values that pass and the number that changed. The `T-FST` that achieves the max criteria is selected as the `P-FST` for the group (App. A.2).

Step 3: `P-FST` → `G-FST` (`general`) - Across `products` there may exist identically-named, but functionally/semantically different `P-FSTs`. The LLM understand the differences in each `P-FST` based in all available information to generate a `G-FST` whose `super_cast()` handles the output of each `P-FST`'s `cast()`. Validity or sanity checks of the values is achieved by performing two consecutive transformations at the `product` and `general` levels. The `validate()` is responsible for performing type, bound, or value-based checks on the output, and is where the LLM use external lookups to establish a ground truth (App. A.3).

Step 4: `G-FST` → `G-FST` (`cross`) - There exist many `G-FSTs` that may represent identical (differently named) or distinct entities, that may be castable. For a given source `G-FST`, the k-nearest `G-FSTs` neighbors is identified by vectorizing each `G-FST` using an embedding model (App.A.4.1). An LLM determines the subset of the k neighbors that are convertible and for each, it generates a `cross_type_cast()` that transforms any output of the source `G-FST` to a value that would be accepted by the neighbors `validate()` (App. A.4.2).

5 Experimental Evaluation

We evaluated the `P-FSTs`, `G-FSTs`, and `cross_type_cast()`'s on three data universes, two of which are freely available to the public (as well as all our code and prompts) with results shown here and code definitions in the Appendix.

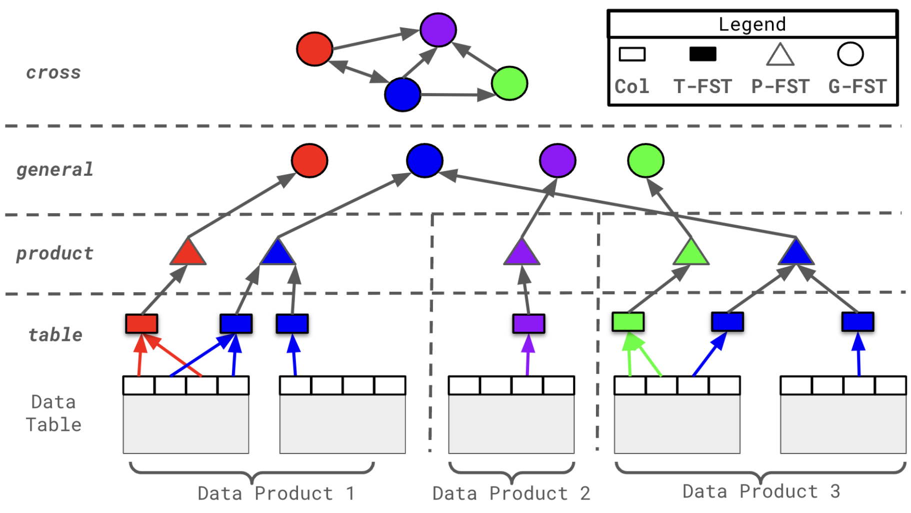


Figure 3: Unshaded rectangles represent columns, shaded rectangles represent T-FSTs, triangles represent P-FSTs, and shaded circles represent G-FSTs. Each color represents a specific semantic entity (i.e. ZipCode, City, State, etc). Circles at the general level are the generalization of the P-FSTs. At the cross level is a graph representing the relationship between G-FSTs.

Universe	Universe Properties			Ontology Properties			
	# Cols.	# Tables	# Data Prod.	# Cols.	# Data Set Types	# Data Prod. Types	# Gen Types
Kaggle	8649	707	237	**7051**	4730	3196	**2043**
Harvard	7007	484	12	**5898**	2998	2325	**2057**
FData	3535	428	13	**3203**	1681	853	**664**

Table 1: Properties of each universe and their ontologies.

Universe Curation - Our universes source from two open-source data providers, Kaggle and Harvard Dataverse, as well as a commercial financial dataset, we refer to by a fictional name FData. For Kaggle, we selected the 707 most commonly used Kaggle-*datasets* and extracted their associated tabular data files (files and datasets are represented as tables and products in our nomenclature). For Harvard, we selected the top 484 most downloaded datasets and organized them similarly to Kaggle, but Harvard required additional preparation due to the large number of fully null columns (App. A.6). FData consists of 428 tables containing a wide breadth of financial terminology. Table 1 shows the aggregated details of each universe. Notice that Kaggle contains the largest number of tables, products, and columns while Harvard and FData contain differing levels of product granularity and table widths, which are both factors that affected the performance of the product and general stages.

LLM Choice - We used OpenAI's gpt-4-0613 LLM with 8k context for the table, general, and cross stages of FSTO-*Gen*.

Evaluation Criteria - To evaluate the functionality of FSTO-*Gen*, we considered the throughput performance of each pipeline stage. Our analysis of the functional characteristics of each stage provides intuition about the behavior of each transformation; however, to demonstrate correctness, a human evaluation of the P-FSTs and cross_type_cast()'s was needed to assess the LLM's ability to select subsets of information (relevant columns that refer to semantic entities in table or relevant semantically similar general FSTs that are castable), as well generate accurate code.

For each P-FST, we iterated over each child FST, sampled 1000 values from the columns covered by the FST, and aggregated a unique set. For each x in this set, P-FST's cast(x) gives a value w. For the G-FST parent of P-FST, G-FST's super_cast(w) gives a value y, which is validated using G-FST's validate(). Then for a neighboring G-FST, cross_type_cast(y) yields a value z, which we validated in the neighbor's validate(). For each cast function (cast(), super_cast(), cross_type_cast()), we record if it passes (indicated by ✓) or errors out (χ). The result of a cast can either be Complex (differs from the original), Identity (identical to the original), or an Exception (erroneous input or cast logic).

5.1 Functional Throughput Results

Column → P-FST: The results are summarized in Table 2. Non-null values mostly pass through unchanged, i.e. the data is already well-formatted and fits the canonical form of the P-FST. Some values change after casting, indicating normalization was necessary. Complex transformations range from rounding floats (App. A.8.1, A.8.2) to mapping

Col Entry	Cast Func	Return Val	Universe		
			Kaggle	**Harvard**	**FData**
Non-Null	✓	Complex	17.14%	19.97%	15.21%
		Identity	68.26%	54.25%	48.81%
	χ	Exception	0.86%	1.30%	0.71%
Null	✓	Complex	12.12%	14.72%	34.22
	χ	Exception	1.62%	9.76%	1.05%

Table 2: The distribution of `cast()` outcomes for product `FST`s across all universes. Trends indicate that data is already in the correct form, or it performs a Complex transformation.

country abbreviations to full names with lookup tables (App. A.8.3). A few times, a `RunTimeError` in the `cast()` is thrown.

Col Entry	Cast Func	Return Val	Validate Func	Universe		
				Kaggle	**Harvard**	**FData**
Non-Null	✓	Complex	Pass	11.59%	7.02%	9.83%
			Fail	2.14%	2.08%	0.37%
		Identity	Pass	79.86%	84.96%	83.68%
			Fail	2.61%	4.12%	2.59%
	χ	Exception	Fail	2.87%	1.38%	1.29%
Null	✓	Complex	Pass	0.04%	0.28%	2.00%
			Fail	0.88%	0.15%	0.24%
		Identity	Pass	0.00%	0.00%	0.00%
			Fail	0.00%	0.00%	0.00%
	χ	Exception	Fail	0.00%	0.00%	0.00%

Table 3: The distribution of outcomes after the application of a `G-FST`'s `super_cast()` and `validate()`. The most common outcome is when Non-null data undergoes an Identity transformation which generally passes the `validate()`. Data undergoing a Complex transformation indicates a different product-level normalization for the same semantic entity.

`P-FST` → `G-FST`: The results are summarized in Table 3. In general, values are left unchanged, indicating that data already achieved normalization in the `P-FST`, which is expected considering that there are many 1-1 correspondences between `P-FST`s and `G-FST`s (per Table 1). When the `super_cast()` is a Complex transformation, this indicates that different communities of interest (in this case products) have differing, yet locally standardized ways of representing the same data (App. A.8.4). In some of these cases, the `validate()` fails, indicating insufficient normalization in the `P-FST` or `G-FST`, incorrect `validate()`, or deeper insights are needed into the domain restriction of the `G-FST` (App. A.8.5). However, more frequently, Complex transforms pass `validate()` (App. A.8.6 A.8.7), indicating that LLMs can derive a common standard for a large variety of entity types at scale.

`G-FST` → `G-FST`: We found (Table 4) many semantically *identical* entities that differed only by class name (App: A.8.8). The generation of `T-FST`s is a generative process with no source of truth, and since LLMs are stochastic, it is likely to name identical semantic entities with slightly differing naming conventions. An artifact of generating ontologies from the bottom-up is that entities at the

Col Entry	Cast Func	Return Val	Validate Func	Universe		
				Kaggle	**Harvard**	**FData**
Non-Null	✓	Complex	Pass	14.26%	18.34%	11.10%
			Fail	0.95%	2.30%	0.97%
		Identity	Pass	77.30%	63.55%	78.75%
			Fail	7.16%	14.69%	5.78%
	χ	Exception	Fail	0.29%	0.45%	0.20%
Null	✓	Complex	Pass	0.00%	0.22%	1.52%
			Fail	0.00%	0.40%	1.67%
		Identity	Pass	0.00%	0.00%	0.00%
			Fail	0.00%	0.05%	0.00%
	χ	Exception	Fail	0.04%	0.00%	0.00%

Table 4: The distribution of outcomes after the application of `cross_type_cast()` between general `FST`s and the target's `validate()`. Values tend to undergo Identity transformation, indicating the existence of semantically-duplicative `FST`s.

most general level may be too specific or too broad; however, this is the fundamental purpose of the `cross_type_cast()`. While these `FST`s differ by name, they shouldn't differ in their semantics (and therefore are close in the vectorized representation space of their corresponding classes). The next most common outcomes were Complex transforms, where we witnessed nontrivial behavior involving lookups, mappings, etc.(App. A.8.9, A.8.10), as well as Identity mappings that failed the neighbor's `validate()` (App. A.8.11). We attribute these cases to hallucinations, whereby the LLM will justify a `cross_type_cast()` with faulty logic.

5.2 Human Evaluation Results

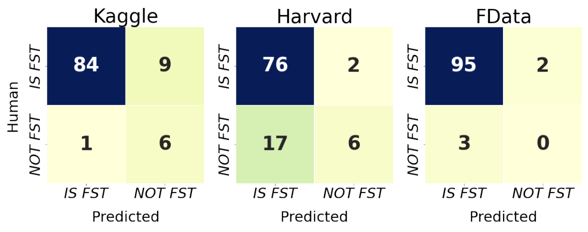


Figure 4: Confusion Matrices of the LLM's ability to recognize and generate `P-FST`s. We witness high True Positive, low False Positive, and low False Negative rates, indicating both high precision and recall.

As there is no ground truth, we performed a human evaluation to assess *quality* and *correctness*. We evaluated `P-FST`s using 50 tables from Kaggle, and 20 tables each fromHarvard and FData for a total of 1̃000 columns. Humans were tasked with labeling whether a column should have an FST associated with it. Our results show (Figure 4) that LLMs have high precision and recall in *recognizing semantic entities* and were able to *generate correctly-scoped*, and functionally correct `FST`s.

We counted when `P-FST`s were either too broad or too specific (Table 5). For those with correct scope, we labeled cases of when the generated class completely differed from the semantics of

	Incorrect Scope		Correct Scope		
Universe	Too Broad	Too Specific	Totally Incorrect	Slightly Wrong	Just Right
Kaggle	15.44%	1.52%	2.78%	5.06%	75.19%
Harvard	17.54%	0.88%	1.32%	0.44%	79.82
FData	49.07%	0%	3.27%	1.87%	45.79%

Table 5: Quality Distribution of True Positive, product-level `FSTs`. The LLM generates types that are too broad, or perfect.

the entity (Totally Incorrect), contained slightly erroneous fields or functional attributes (Slightly Wrong), or was (Just Right). In Kaggle and Harvard, these types were generally right, while in FData they were either too broad or perfect. We hypothesize LLMs tend to generalize unfamiliar concepts (e.g. it labeled a finance-specific growth rate with "Growth Rate"), making any cast or validation ineffectual. Finally, even when `P-FSTs` were scoped properly, some contained errors such as mismatches between its name or description or made a false assertion (App. A.8.11).

.

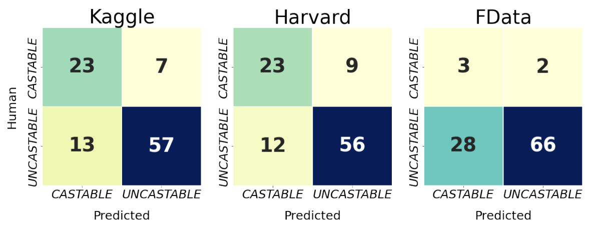


Figure 5: Confusion Matrices of LLM's ability to recognize true, castable relationships between `G-FSTs`. We witness lower levels of recall and precision, due to the LLMs hallucination.

To assess the quality of the `cross_type_cast()`, we sampled 20 `G-FSTs` from each universe, as well as its k=20 neighbors in representation space. For each neighbor, we recorded whether there should exist a `cross_type_cast()` between the source and destination. This served as a ground-truth comparison to the edges present in the generated ontology, with which we classified if the LLM was able to recognize when two `G-FSTs` were truly cross-type-castable. As seen in Fig 5, we witnessed acceptable recall on Kaggle and Harvard, low recall on FData, and low precision across all universes. The reasons for low recall on FData are related to the analysis in Table 5: when an LLM can't recognize semantic entities, it's difficult to assess whether a cross-type-cast is allowed. Additionally, because LLMs tend to hallucinate, the false positive rate was large across all universes, especially for FData, which is an artifact of the generality of the `FSTs`. LLMs found cross-type-casts between types like "Percent" and "Ratio", which might refer to entirely different entities, but when generalized, hallucinations become more likely.

6 Cost Analysis

Universe	Step	Avg Prompt Length	# LLM Calls	# Human Changes
Kaggle	Step 1	3685	707	20
	Step 3	1255	2043	8
	Step 4	5895	2043	23
Harvard	Step 1	3888	428	19
	Step 3	1254	664	6
	Step 4	5843	664	7
FData	Step 1	3723	484	1
	Step 3	1257	2057	2
	Step 4	5931	2057	20

Table 6: Breakdown of LLM-Dependent Parts of `FSTO`-*Gen*. We chose GPT-4 with 8k context, which was enough to fit the instructions, table serialization, and examples for the prompts in Step 1,3, and 4. Step 4 required the largest number of tokens because its prompt contained several examples to help reduce hallucination behavior. .

`FSTO`-*Gen* utilizes OpenAI's GPT-4 LLM to perform Steps 1, 3, and 4, which attaches external and human-in-the-loop costs. During Step 1, there is one LLM call per table in the Universe, and during Steps 3 and 4, another single LLM call per `G-FST`. The execution time for each call is a summation of the time to send the request + the inference time of LLM + the time to receive the response payload. However, since inferences operate on independent tables or `G-FSTs`, we batched our calls into groups of 24 (number of cores on our machine). This makes `FSTO`-*Gen* extensible to larger universes, pursuant to increased parallelism. Finally, since the response may sometimes contain errors in its construction or run-time dependencies that weren't explicitly detailed in the prompt, human correction was necessary. A human adjusted these outputs to ensure compilability. These were minimal relative to the number of LLM calls (Table 6).

7 Applications

Data Normalization - Data Normalization is baked into the construction of our ontology. The `cast()` in `T-FSTs`, `super_cast()` in `G-FSTs`, and `cross_type_cast()`'s represent normalization layers for typing a columnar value. Listing 1 shows the `super_cast()` generated for a `G-FST` for "Income-Level". Across tables, Income-Level can be represented as a range or a specific amount, and the aggregation process results in a single all-inclusive transformation covering both cases.

```python
def super_cast(self, val):
    if isinstance(val, str):
        if val == 'Less Than 5000': return 0.0
        elif val == '5000-10000': return 7500.0
        elif val == '10000-20000': return 15000.0
        elif val == 'More Than 20000': return
            20000.0
        else: raise Exception('Invalid income
            level')
    elif isinstance(val, (int, float)):
        if val >= 0: return float(val)
        else: raise Exception('Invalid income
            level')
    else: raise Exception('Invalid income level')
```

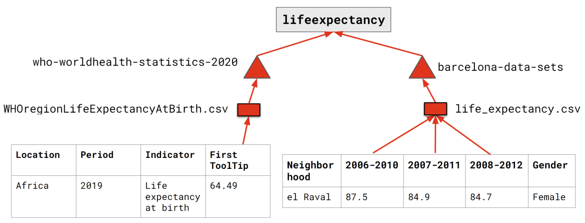
Listing 1: The `super_cast()` of `G-FST`, "IncomeLevel," which normalizes range or number inputs to a number.

Data Validation - Validation of a semantic entity generally requires knowledge about the distributional properties of a column or worldly knowledge about the domain set of the entity. For accelerometer data, the LLM asserted bounds between $[-1, 1]$ (App. A.8.1). Additionally, there is robustness in generating fault-tolerant `T-FSTs`, such as the "Precipitation" `T-FST` (App. A.8.2. Precipitation data is usually numeric, but there were several occurrences of the letter 'T'. Leveraging worldly knowledge resulted in assigning 'T' to a value of 0 as it generally refers to trace amounts of rainfall. Finally, when dealing with entities with large domain sets, LLMs use external lookup tables, (e.g. the set of US States configured in `pycountry`) to assert the correctness of state names to their two-letter abbreviation (Listing 2). An interesting avenue for future work is utilizing validation to perform better normalization. While we provide a summary report and a subset of values, these may not be enough to construct robust validation over an entire column, so an iterative approach that couples feedback from normalization may be used to improve validation.

```python
def validate(self, val):
    casted_val = self.super_cast(val)
    us_states = pycountry.subdivisions.get('US')
    for state in list(us_states):
        if state.name.title() == casted_val:
            return True
    return False
```

Listing 2: The `validate()` of `G-FST`, "USState," which uses pycountry to checks the validity of `super_cast()`.

Data Fusion - Data Fusion is automatically unlocked with the graph construction of the `product` and `general` stages of `FSTO`-*Gen* to join tables on the common columnar entity. In general, any two tables can be merged using a common `P-FST` or `G-FST` ancestor, and the explicitness of the ancestor-child relationships in `FSTO`-*Gen* reduces the complexity of finding a semantic match. Additionally, the construction of `cross_type_cast()`'s allow for non-trivial joins between tables. For example, the LLM generated a `cross_type_cast()` to convert education level from string to integer enums by mapping their domains (App. A.8.9).


lifeexpectancy
who-worldhealth-statistics-2020
barcelona-data-sets
WHOregionLifeExpectancyAtBirth.csv
life_expectancy.csv

Location	Period	Indicator	First ToolTip
Africa	2019	Life expectancy at birth	64.49

Neighbor hood	2006-2010	2007-2011	2008-2012	Gender
el Raval	87.5	84.9	84.7	Female

Figure 6: Two Data Tables, belonging to different products, contain columns associated with the semantic entity "Life-Expectancy", but neither are explicitly named. The LLM uses table context to perform this association and `FSTO`-*Gen* identifies the relation as a graph.

Data Discovery - Discovery is achieved by allowing practitioners to semantically search over the informational/functional properties of `FSTs` or traverse the relationships in the synthetic ontology. For example, `FSTO`-*Gen* generated a `FST` for `lifeexpectancy` that spanned four products, two of which came from the "WHO-world-health-statistics-2020" product where a column was named "First ToolTip", and one from "Barcelona-data-sets" product where columns associated with life-expectancy were named by time-ranges. It is doubtful that someone searching for data related to life-expectancy would have known that these products would be related, but automated contextualization with LLMs combined with the hierarchical composition of `FSTO`-*Gen* found these relations.

8 Conclusion and Limitations

`FSTO`-*Gen* is an LLM-powered framework for automating the generation of `G-FSTs` and their relations from columnar data. Our synthetic ontology is hierarchical and functional, such that successive layers represent the transformation of semantically identical columns into a singular representation. These ontologies are useful in automating various data onboarding tasks: normalization, validation, fusion, and discovery. `FSTO`-*Gen* is less successful with domain-specific datasets, because LLMs are less familiar with terminology, leading to incorrectly scoped `FSTs`. We are exploring fine-tuning [10] and retrieval-augmented-generation [13] to overcome this gap. However, even in its current state, `FSTO`-*Gen* shows promise in reducing the tedious job of data onboarding.

References

[1] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *international semantic web conference*, pages 722–735. Springer.

[2] Felix Biessmann, Tammo Rukat, Phillipp Schmidt, Prathik Naidu, Sebastian Schelter, Andrey Taptunov, Dustin Lange, and David Salinas. 2019. Datawig: Missing value imputation for tables. *Journal of Machine Learning Research*, 20(175):1–6.

[3] Wenhu Chen, Hongmin Wang, Jianshu Chen, Yunkai Zhang, Hong Wang, Shiyang Li, Xiyou Zhou, and William Yang Wang. 2019. Tabfact: A large-scale dataset for table-based fact verification. *arXiv preprint arXiv:1909.02164*.

[4] Marco Cremaschi, Flavio De Paoli, Anisa Rula, and Blerina Spahiu. 2020. A fully automated approach to a complete semantic table interpretation. *Future Generation Computer Systems*, 112:478–500.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[6] Christiane Fellbaum. 2010. Wordnet. In *Theory and applications of ontology: computer applications*, pages 231–243. Springer.

[7] Benjamin Feuer, Yurong Liu, Chinmay Hegde, and Juliana Freire. 2023. Archetype: A novel framework for open-source column type annotation using large language models.

[8] Heng Gong, Yawei Sun, Xiaocheng Feng, Bing Qin, Wei Bi, Xiaojiang Liu, and Ting Liu. 2020. Tablegpt: Few-shot table-to-text generation with table structure reconstruction and content matching. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 1978–1988.

[9] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek Narasayya, and Surajit Chaudhuri. 2018. Transform-data-by-example (tde) an extensible search engine for data transformations. *Proceedings of the VLDB Endowment*, 11(10):1165–1177.

[10] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models.

[11] Junjie Huang, Chenglong Wang, Jipeng Zhang, Cong Yan, Haotian Cui, Jeevana Priya Inala, Colin Clement, Nan Duan, and Jianfeng Gao. 2022. Execution-based evaluation for data science code generation models. *arXiv preprint arXiv:2211.09374*.

[12] Madelon Hulsebos, Kevin Hu, Michiel Bakker, Emanuel Zgraggen, Arvind Satyanarayan, Tim Kraska, Çagatay Demiralp, and César Hidalgo. 2019. Sherlock: A deep learning approach to semantic data type detection. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1500–1508.

[13] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*.

[14] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. 2021. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 468–479. IEEE.

[15] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

[16] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

[17] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *arXiv preprint arXiv:2004.00584*.

[18] Chris Mayfield, Jennifer Neville, and Sunil Prabhakar. 2010. Eracer: a database approach for statistical inference and data cleaning. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 75–86.

[19] Yinan Mei, Shaoxu Song, Chenguang Fang, Haifeng Yang, Jingyun Fang, and Jiang Long. 2021. Capturing semantics for imputation with pre-trained language models. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 61–72. IEEE.

[20] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*, pages 19–34.

[21] Avanika Narayan, Ines Chami, Laurel Orr, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911*.

[22] Panupong Pasupat and Percy Liang. 2015. Compositional semantic parsing on semi-structured tables. *arXiv preprint arXiv:1508.00305*.

[23] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*.

[24] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. Holoclean: Holistic data repairs with probabilistic inference. *arXiv preprint arXiv:1702.00820*.

[25] Amazon Web Services. Data marketplace - aws data exchange. Https://aws.amazon.com/data-exchange.

[26] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çağatay Demiralp, Chen Chen, and Wang-Chiew Tan. 2022. Annotating columns with pre-trained language models. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1493–1503, New York, NY, USA. Association for Computing Machinery.

[27] Huan Sun, Hao Ma, Xiaodong He, Wen-tau Yih, Yu Su, and Xifeng Yan. 2016. Table cell search for question answering. In *Proceedings of the 25th International Conference on World Wide Web*, pages 771–782.

[28] Cong Yan and Yeye He. 2018. Synthesizing type-detection logic for rich semantic data types using open-source code. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 35–50, New York, NY, USA. Association for Computing Machinery.

[29] Dan Zhang, Madelon Hulsebos, Yoshihiko Suhara, Çağatay Demiralp, Jinfeng Li, and Wang-Chiew Tan. 2020. Sato: contextual semantic type detection in tables. *Proc. VLDB Endow.*, 13(12):1835–1848.

[30] Jing Zhang, Bonggun Shin, Jinho D Choi, and Joyce C Ho. 2021. Smat: An attention-based deep learning solution to the automation of schema matching. In *Advances in Databases and Information Systems: 25th European Conference, ADBIS 2021, Tartu, Estonia, August 24–26, 2021, Proceedings 25*, pages 260–274. Springer.

[31] Chen Zhao and Yeye He. 2019. Auto-em: End-to-end fuzzy entity-matching using pre-trained deep models and transfer learning. In *The World Wide Web Conference*, pages 2413–2424.

A Appendix

A.1 Step 1: (`table` → `T-FST`)

A.1.1 Table Serialization

In step 1 of FSTO-*Gen*, an LLM is tasked with converting a table into a mapping dictionary which maps some subset of the tables' columns to FSTs names and the definition of FSTs themselves. The motivation for this scheme sources from the serialization process in Sherlock[12] and DoDuo[26], which showed that semantic type annotation is enhanced when the annotation engine can understand table-level and column-level semantics. In fact, in our experiments, there were many cases where columns were badly named, but table/product names helped the LLM perform inference. The problem with table serialization is fitting it into the context window of the LLM, so given a table as a pandas dataframe, we convert it to a string using `serialize()` as defined in Listing 3.

```
import pandas as pd

def serialize(df: pd.DataFrame, data_dict:
     dict[str, str]) -> str:
    string = ''
    for col in df.columns:
        is_numeric = ... # check if col is
             numeric/categorical
        quartile_1, median, quartile_3 =
             df['col'].quantile([0.25,0.5,0.75])
        if is_numeric:
            string += f"""
            -col: {col}
            *description: {data_dict[col]}
            *mean: {df[col].mean()}
            *std: {df[col].std()}
            *min: {df[col].min()}
            *0.25: {quartile_1}
            *0.5: {median}
            *0.75: {quartile_3}
            *max: {df[col].max()}
            *first_five: {df[col].iloc[:5].values}
            *num_na: {df[col].isna().sum()}
            """
        else:
            string += f"""
            -col: {col}
            *description: {data_dict[col]}
            *num_na: {len(df[col].unique())}
            *top_5_most_frequent:
                 {df[col].value_counts().nlargest(5)}
            *num_na: {df[col].isna().sum()}
            """
    return string
```

Listing 3: `serialize()` function which converts a pandas dataframe table to a string.

A.1.2 LLM Prompt

Below we denote the prompt (denoted in the typewriter font) used to generate a set of T-FSTs and their mappings to columnar names for a particular table. The prompt is formatted using Jinja templating (https://jinja.palletsprojects.com/en/3.1.x/) which is materialized using the following inputs:

1. base_class_definitions - T-FST Base Classes as seen in Fig. 1
2. numeric_col_summary - App. A.1.1
3. categorical_col_summary - App. A.1.1
4. end_to_end_example - End to End Example with column summary and resultant T-FSTs and mapping.
5. dataset_name - Name of the table
6. dataset_description - Description of the table or empty string if it doesn't exist
7. column_dict - App. A.1.1
8. necessary_imports - App. A.5

```
<START_PROMPT>
You are SemanticGPT an assistant that identifies Semantic Types for tabular data. Semantic Types are data types that ingrain semantic context into an entity. Semantic Types are valuable because they restrict the domain with which a column can span, meaning that Semantic Types have a fixed domain of values and format. Here are Python base class Semantic Type definitions: {{base_class_definitions}}

I am going to provide you with the name of the dataset, along with a hyphenated list of the columns enclosed in “‘, as well as certain properties per each column that represent a summary of all columns. If the column is inferred to be numeric, the input will be in the form of: {{numeric_col_summary}}. If the column is inferred to be non-numeric, the input will be in the form of: {{categorical_col_summary}}.

Your goal is to read through the column summary and try to figure out 1) if there exists a Semantic Type for a given column 2) If you haven't already constructed a Semantic Types definition for the column, construct one that inherits the MOST SPECIFIC class definition from the provided base class
```

```
definitions.  3) Assign each column to
the constructed Semantic Type, using a
dictionary. Note, columns may be mapped
to the same Semantic Type or not at all
(I expect there to be a small set of
constructed Semantic Types).  To aid in
this process, I created a decision-tree
i want you to follow:

  - Does the column relate to a semantic
    type?
      - YES: should the column only take
        in two values?
          - YES: BooleanSemanticType
          - NO: does the column represent
            a numerical semantic type?
            - YES:
              NumericSemanticTypeWithUnits
              || NumericSemanticType
            - CategoricalEnumSemanticType
              || CategoricalSemanticType
      - NO: do nothing

{{end_to_end_example}}.

Now I want you to generate the
corresponding Python SemanticType
definitions for the given table.  It
is of UPMOST importance that your code
compiles and runs.  Do not add any
extra text, "‘, or "python" prefixes.
I just want the class definitions and
the Mapping dictionary.  Also the class
names should be real-world entities and
spelled correctly. For any string column,
think about how to extract a uniform
representation in the cast() function.
- SUPER IMPORTANT: I will provide you
with the list of libraries to start
with, don’t import anything else.  Just
start writing the classes definitions
and Mapping dictionary.  Make sure your
class definition names don’t conflict
with the imports.
COLUMNS="‘
dataset_name:{{dataset_name}}
{{dataset_description}}
{{column_dict}}
"‘
{{necessary_imports}}
<END_PROMPT>
```

A.2 Step 2: (T-FST → P-FST) Details

```python
import numpy as np

def agglomerate(tfsts: list[DataSetSemanticType],
     unique_set: list[Any]):
    mat = np.zeros((len(tfsts), 2))
    for ix in range(len(tfsts)):
        num_passes = 0
        num_changes = 0
        for x in unique_set:
            try:
                y = tfsts[ix].cast(x)
                num_passes += 1
                num_changes += y != x
            except Exception as e:
                pass
        mat[ix] = [num_passes, num_changes]
    max_ix = mat[mat[:, 0] == mat[:,
        0].max()].argmax()
    return tfsts[max_ix]
```

Listing 4: The agglomerate function is used to combine identically-named T-FSTs (belonging to the same data-product) into a single P-FST. Since our ontology increases in functional generalization, we pick the T-FST with the most general cast() as the P-FST.

After Step 1 of FSTO-*Gen*, for each product, there exist many T-FSTs with the same name, which tend to also have the same semantics, so we perform an agglomeration of them into a single P-FST to reduce the number of LLM calls at the general stage. Given a grouping G of T-FSTs, we ideally would perform an agglomeration similar to the general stage, because while the T-FSTs have the same name, they may still have different fields or cross_type_cast()’s. However, this is computationally expensive across the many groupings in the product stage, so instead we pick the most functionally general T-FST that performs meaningful normalization on the columnar input. We seek to maximize throughput through the cast(), so we pick the T-FST that throws the least errors and performs the most Complex transformation on a set of columnar values. To assemble this set, we sample 1000 values from each column associated with a T-FST and create a unique set. We show this in the agglomerate() (Listing 4) which takes in a grouping of T-FSTs and a unique set of values, and generates a P-FST.

A.3 Step 3: (P-FST → G-FST) Prompt

Below we denote the prompt used to generate a single G-FST from a list of P-FSTs with a common name. We use the following variables:

1. general_sem_type_class_def - Fig. 2
2. end_to_end_example - Example where we provide a list of P-FSTs and an ideal G-FST.

3. class_defs - P-FSTs to aggregate

4. necessary_imports - App. A.5

```
<START_PROMPT>
You are GroupGPT, an assistant that will receive a list of Python class definitions and return a single class that spans them all. For context, each class represents a Semantic Type, which is a real-world entity that corresponds to some piece of columnar data I have collected. Each type has its own attributes, specified formatting, and a cast() function that takes as input a single value from the column and returns a formatted, casted value. You will populate the following class using the instructions from the comments: {{general_sem_type_class_def}}.
- For super_cast(), I want your code to be ROBUST, and handle ALL of the outputs generated by the cast() of the provided classes.
- For validate(), I want your code to sanity-check that the value is correct.
- MOST IMPORTANT: Make sure that your solution compiles and will execute when I instantiate the class. - A useful strategy for "picking" a canonical format, is to pick the format of ONE class, and convert any output from the cast() of the provided classes to that format.

{{end_to_end_example}}.

I want you to output a single class that INHERITS GenericSemanticType (but change the class name). Do not add any extra text, “‘, or "python" prefixes. The class names should be real-world entities, spelled correctly, and SHOULD BE LOWERCASE.
- SUPER IMPORTANT: I will provide you with the list of libraries to start with, don’t import anything else. Just start writing the classes definitions and Mapping dictionary. Make sure your class definition names don’t conflict with the imports.
CLASSES = “‘
{{class_defs}}
RETURN=
{{necessary_imports}}
<END_PROMPT>
```

A.4 Step 4: (G-FST $\rightarrow$ G-FST)

A.4.1 Algorithm Details

There exist many G-FSTs that are semantically similar, or even identical, so to identify joins across products, we generate `cross_type_cast()`’s between similar G-FSTs. First, we serialize each G-FST into a string by concatenating the class name with its `description` instance field. Then for each string we vectorize the class using an embeddings model (we use the `all-MiniLM-L6-v2` model[23]) to convert the model into a 384-length vector and find the nearest k neighbors using kNN (we use $k = 20$ to reduce the number of tokens in the `cross` LLM prompt). For each G-FST, we concatenate the `cross` prompt with the G-FST and each of its 20 neighbors to receive a maximum of 20 output `cross_type_cast()`’s.

A.4.2 LLM Prompt

Below we denote the prompt used to generate up to k `cross_type_cast()`’s. We use the following variables in the prompt:

1. len_targets - k G-FSTs to compare to
2. simple_example - Example where we provide a list of P-FSTs and an ideal G-FST.
3. class_defs - P-FSTs to aggregate
4. necessary_imports - App. A.5
5. simple__partial_example - Partial Example of two castable G-FSTs.
6. full_example - Full Example with input class definitions and generated `cross_type_cast()`’s.
7. partial_example_1 - Partial Example that shows a true positive
8. partial_example_2 - Partial Example that shows a false positive
9. partial_example_3 - Partial Example that shows a false positive
10. src_class_def - Source G-FST
11. target_class_defs - Target G-FSTs

12. necessary_imports - App. A.5

```
<START_PROMPT>
You are CastGPT, an agent that will help me convert between two Semantic Type Class Definitions. These class definitions have a super_cast() method, which converts a value to the class's canonical format, and a validate() method which sanity-checks the result of the validate() method. Given a root class definition and {{len_targets}} target class definitions, I want you to generate at MOST {{len_targets}} functions. For example, given root class a and target class b you will generate a method called cross_type_cast_between_a_and_b(val).
This is how it will be used:
“‘
casted_a_val = a().super_cast(val)
casted_b_val =
cross_type_cast_between_a_and_b(
casted_val
)
b().validate(casted_b_val)
“‘
There are two main challenges here. The first is that you need to figure out if class a and class b represent the same type of information, and whether the result of a().super_cast(val) can be casted to the form/function described by b().super_cast(val). If that is possible, then you need to generate the right python mapping code to perform the conversion of a().super_cast(val) to the format of b().

{{simple__partial_example}}

The full form of the function is defined as follows. For each (a,b) pairing that is valid (maximum {{len_targets}}), I want you to generate: {{cross_type_cast_def}}.

{{full_example}}
{{partial_example_1}}
{{partial_example_2}}
{{partial_example_3}}

Now I want you to try on the following examples. Like the example, generate the cross_type_cast() functions according to the template I gave you, don't give me anything else but code!
- DO NOT generate an empty cross_type_cast() function, just skip it.
- Also, if you need bizzare mapping code, DO NOT generate a cross_type_cast() function.
- I want you to be EXTREMELY conservative with your conversions. There shouldn't be a lot of conversions that work, because only small numbers of entities actually represent the same type of information.
- SUPER IMPORTANT: I will provide you with the list of libraries to start with, don't import anything else.
SOURCE=“‘
{{src_class_def}}
“‘
TARGETS=“‘
{{target_class_defs}}
“‘
FUNCTIONS = “‘
{{necessary_imports}}
<END_PROMPT>
```

A.5 FST allowed Imports

To perform data manipulation tasks and lookups, in the declaration of each FST, we allowed the following set of imports:

1. numpy - to perform array manipulation.
2. pandas - to handle na/null values.
3. datetime - to perform date string operations.
4. math - to perform rounding.
5. pycountry/countryinfo - to perform geographic lookups.

A.6 Data Curation Details

The Kaggle dataverse was sourced from the top 1000 most downloaded Kaggle Datasets, and we extracted the ".csv" files present into each dataset. Each Kaggle dataset was termed as a product, while the ".csv" files as a table. Across all universes, we selected tables that had at least 80% of columns with at least more than one-null value. From the set of 1000, we selected the top 707. In the end,

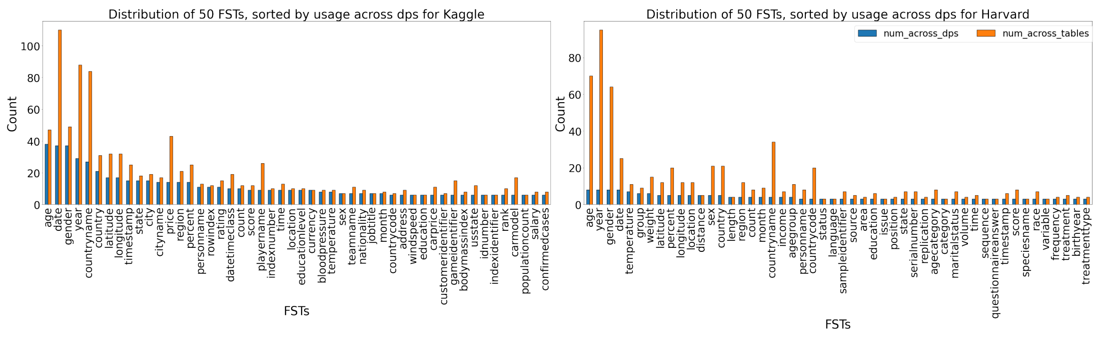


Figure 7: Distribution of types and their usage across data products (num_across_dps) and data tables (num_across_tables) in Kaggle and Harvard universes. We constructed a histogram of type usage and sorted the G-FSTs from high to low usage across data products. The top 50 are shown in the above histograms.

there were 237 products with an average of 3 tables per product. We performed a similar process for the Harvard dataverse, except we selected the top 500 by filtering on whether the datasets contained ".tab" files (tabular data files) and if they were released publicly. Additionally, Harvard datasets are generally mapped directly to a single tabular file, so to enhance the product stage we categorized datasets by their subject tag, which could be any one of: "agriculturalsciences, businessandmangement, earthandenvironmentalsciences, law, socialsciences, artsandhumanities, chemistry, engineering, mathematicalsciences, astronomyandastrophysics, computerandinformationscience, medicinehealthandlifesciences". With a small number of products, there is less opportunity for aggregation to occur at the `product` stage as columns are less likely to have semantically similar information in wider groupings. This explains the greater decrease in the number of types from `FSTs` to `P-FSTs` in Kaggle versus Harvard (Table 1). The FData universe is a commercial, proprietary universe and its details can't be revealed under confidentiality agreements.

A.7 Type Distribution

In Fig. 7 we show the distribution of `G-FSTs`, sorted by usage across data products, in Kaggle and Harvard. Notice how the most frequently occurring `G-FSTs` are common across both Kaggle and Harvard, for example, "age", "date", "year", "gender", "latitude", "longitude", "country", "education", "currency", etc. are repeated. The aforementioned `G-FSTs` are canonical examples of Semantic Types, however as you move closer to the tails of the histograms, there are more dataset-specific types like "confirmedcases", "salary", "treatmenttype", or "speciesname". The middle and right-tail of this distribution uncovers domain-specific datatypes, and the benefit of `FSTO`-*Gen* is that these types can be discovered through automation rather than a human exhaustively iterating through a universe.

A.8 Examples

A.8.1 BodyAcceleration

```
class
    bodyacceleration(NumericSemanticTypeWithUnits):

    def __init__(self, *args, **kwargs):
        self.description = 'The mean body
            acceleration in a certain direction'
        self.valid_range = [-1.0, 1.0]
        self.dtype = float
        self.format = 'Body acceleration should be
            a floating point number between -1 and
            1'
        self.units = 'The unit of body acceleration
            is 1g, where g is the acceleration due
            to gravity'
        self.examples = [-1.0, -0.5, 0.0, 0.5, 1.0]

    def cast(self, val):
        num = float(val)
        if num < -1.0 or num > 1.0:
            raise Exception('Invalid body
                acceleration')
        return round(num, 6)
```

Listing 5: Body Acceleration `P-FST`

This `P-FST` (Listing 5) was generated from the "human-activity-recognition-with-smartphones" product from Kaggle. This `FST` represents accelerometer values, and the generated cast() function will float-cast and round the number. However, these values are stored as strings and contain various rounding conventions. The `cast()` standardizes the number of decimal points to 6, and converts all strings to floats. Additionally, using its understanding of accelerometer data, the LLM it assigned a `unit` of "1g" for acceleration. It also used the min/max values from App. A.1.1 to create bounds. These may not be right, but these are rules enforced by the data and the LLM's knowledge about accelerometer values.

A.8.2 Precipitation

```python
class precipitation(NumericSemanticTypeWithUnits):
    def __init__(self, *args, **kwargs):
        self.description = 'Precipitation levels in
            inches'
        self.valid_range = [0, float('inf')]
        self.dtype = float
        self.format = 'Precipitation should be a
            floating point number indicating
            inches of precipitation.'
    self.units = 'Inches'
        self.examples = [0, 0.254, 0.508, 0.762,
            1.016]

    def cast(self, val):
        if val == 'T':
            return 0.0
        return round(float(val), 3)
```

Listing 6: Precipitation P-FST

In the construction of a *precipitation* FST (Listing 6) in the "weatherww2" product from Kaggle, the product contains weather information during World War 2, and each table contains data relative to specific geographic locations. A column, named "precip.", consists of a mixture of floating-point values and a single letter ’T’. Without context, it could be confusing to any user of the data, or even cause failures in any downstream pipelines that rely upon the column being floating-point. The table and product stages of FSTO-*Gen* identified a few important features: 1) the table corresponded to U.S. weather conditions 2) "precip." refers to "precipitation" 3) precipitation in the U.S. is measured in inches, so it subclassed a NumericSemanticTypeWithUnits and added "inches" as a unit 4) identified the existence of a value ’T’, which canonically refers to when *trace* amounts of rain occur and replaced ’T’ with 0. Encapsulated in an FST, this type allows any user to understand how it was processed.

A.8.3 NationalityName

Each table in the "fifa-22-complete-player-dataset" product contains information about individual players in the soccer videogame, FIFA 2022. One type that arose in this product was a P-FST (Listing 7) that represents the *nationality* of a soccer player, and the generated cast() function checks that the country is valid by using a lookup from the pycountry library. It produced an Exception for the value “China PR”, which is the name of the soccer team, not a nationality name.

```python
class nationalityname(CategoricalEnumSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Name of a nationality'
        self.valid_values = 'Name should be a
            string and a valid country name'
        self.format = 'Names should be capitalized'
        self.examples = ['United States', 'France',
            'Germany', 'Canada', 'Brazil']

    def cast(self, val):
        country = pycountry.countries.get(name=val)
        if country is None:
            raise Exception('Invalid country name')
        return country.name
```

Listing 7: NationalityName P-FST

A.8.4 Latitude

These two T-FSTs (Listing 8) were generated in

```python
class latitude(NumericSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Latitude where the loss
            event occurred'
        self.valid_range = [32.548, 41.867]
        self.dtype = float
        self.format = 'Latitude should be a
            floating point number'
        self.examples: list = [39.739, 39.747,
            39.763, 39.78, 39.133]

    def cast(self, val):
        return float(val)

class latitude(NumericSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Geographical latitude'
        self.valid_range = [-90.0, 90.0]
        self.dtype = float
        self.format = 'Latitude should be a
            floating point number between -90.0
            and 90.0'
        self.examples = [32.516372, 32.478485,
            32.442435, 32.408118, 32.373841]

    def cast(self, val):
        num = float(val)
        if num < -90.0 or num > 90.0:
            raise Exception('Invalid latitude')
        return num
```

Listing 8: Latitude P-FSTs

Harvard’s "earthandenvironmentalsciences" product, and merged during the product stage of FSTO-*Gen*. This example shows how table-level generation can produce a class with the same name, but slightly different semantics. The first class signifies that the *latitude* corresponds to a loss event, while the second class refers to the most general notion of *latitude* and contains a bound check within its cast().

A.8.5 Duration

The duration G-FST in Listing 9 represents the temporal *duration* between events and correctly throws an error when the value is less than 0, indicating that an event was in the past. Generally, the duration between events is seen as a scalar, regardless of whether the event was passed in or not,

```
class duration(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Duration in seconds'
        self.format = 'Should be a positive
             floating point number representing
             seconds'
        self.examples = [22.534, 22.745, 22.105,
             23.477, 22.684]

    def super_cast(self, val):
        if isinstance(val, str):
            val = float(val)
        return round(val, 3)

    def validate(self, val):
        casted_val = self.super_cast(val)
        if not isinstance(casted_val, float) or
             casted_val < 0:
            return False
        return True
```

Listing 9: Duration G-FST

so this function failed `validate()` on the column value of "-27.83". Whether or not this is the correct behavior, alerts any user of this G-FST about how negative numbers are being handled and allows them to alter the behavior.

A.8.6 Gender

```
class gender(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'A gender'
        self.format = 'In lower-case and as a
             string'
        self.examples = ['male', 'female', 'male',
             'female', 'male']

    def super_cast(self, val):
        str_val = str(val).lower()
        if str_val in ['male', 'female', 'm', 'f',
             '1', '2']:
            if str_val == 'male' or str_val == 'm'
                 or str_val == '1':
                return 'male'
            elif str_val == 'female' or str_val ==
                 'f' or str_val == '2':
                return 'female'
        else:
            return 'other'

    def validate(self, val):
        casted_val = self.super_cast(val)
        if casted_val in ['male', 'female',
             'other']:
            return True
        else:
            return False
```

Listing 10: Gender G-FST

A.8.7 Timestamp

The timestamp G-FST in Listing 11 represents a timestamp in a specific format. The P-FSTs that it agglomerates each use a unique format, so as seen in the `super_cast()`, it performs an exhaustive normalization process of each type to that of `self.format`.

```
class timestamp(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'A timestamp'
        self.format = "A string in the format
             'YYYY-MM-DD HH:MM:SS'"
        self.examples = ['2020-01-01 00:00:00',
             '2019-12-31 23:59:59', '2020-02-29
             12:34:56', '2019-02-28 01:23:45',
             '2020-12-31 11:11:11']

    def super_cast(self, val):
        if isinstance(val, int) or isinstance(val,
             float):
            return datetime.utcfromtimestamp(
                val
            ).strftime('%Y-%m-%d %H:%M:%S')
        elif isinstance(val, str):
            try:
                return datetime.strptime(
                    val, '%Y-%m-%d %H:%M:%S'
            ).strftime('%Y-%m-%d %H:%M:%S')
            except ValueError:
                try:
                    return datetime.strptime(
                        val, '%m/%d/%Y %H:%M:%S'
                    ).strftime('%Y-%m-%d %H:%M:%S')
                except ValueError:
                    try:
                        return datetime.strptime(
                            val, '%d/%m/%Y %H:%M:%S'
                        ).strftime('%Y-%m-%d
                         %H:%M:%S')
                    except ValueError:
                        try:
                            return
                              datetime.strptime(
                             val,
                             '%Y-%m-%d
                                 %H:%M:%S%z'
                            ).isoformat()
                        except ValueError:
                            try:
                                return datetime \
                                .strptime(
                                    val,
                                    '%H:%M:%S'
                                ).strftime(
                                '%Y-%m-%d %H:%M:%S'
                                )
                            except ValueError:
                                raise Exception(
                                'Invalid timestamp'
                            )
        else:
            raise Exception('Invalid timestamp')

    def validate(self, val):
        casted_val = self.super_cast(val)
        try:
            datetime.strptime(casted_val, '%Y-%m-%d
                 %H:%M:%S')
            return True
        except ValueError:
            return False
```

Listing 11: Timestamp G-FST

A.8.8 Redundant G-FST Names

In Listing 12, we show the generated G-FSTs related to Covid-19 Case Counts. Each class contains differing levels of granularity in its name, but the descriptions are all relatively similar. In cases like these, the `super_cast()`'s between any pair is a Identity mapping.

```
class confirmedcases(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Number of confirmed
            COVID-19 cases'
        self.format = 'The number should be a
            non-negative integer'

class covidcases(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'COVID-19 Cases'
        self.format = 'COVID-19 cases should be a
            positive integer, representing the
            number of cases'

class casescount(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Count of COVID-19 cases'
        self.format = 'Count of cases should be an
            integer with no decimal places'

class numcases(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Number of COVID-19
            cases'
        self.format = 'Number of cases should be a
            positive integer'
```

Listing 12: Covid-19 Case G-FSTs (only the name and its description and format fields are shown for brevity)

```
class education(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Level of education'
        self.format = 'In capitalized string form'
        self.examples = ['Secondary / Secondary
            Special', 'Higher Education',
            'Incomplete Higher', 'Lower
            Secondary', 'Academic Degree']

class mothereducation(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = "Education level of the
            student's mother"
        self.format = 'Education level should be an
            integer'
        self.examples = [0, 1, 2, 3, 4]

def cross_type_cast_between_education_\
    and_mothereducation(val):
    reason = 'Both the Education and
         mothereducation classes represent the same
         real-world entity, which is the education
         level. However, they represent this
         information in different formats. The
         education class represents education
         levels as strings, while mothereducation
         represents them as integers. We can map
         the string representation to the integer
         representation by identifying keywords in
         the string that correspond to different
         integer values.'
    mapping = {'no education': 0, 'education level
         1': 1, 'education level 2': 2, 'education
         level 3': 3, 'education level 4': 4,
         'education level 5': 5, 'education level
         6': 6, 'higher education': 7}
    return mapping.get(val.lower(), 0)
```

Listing 13: Conversion Between Two Categorical Enum G-FSTs representing education (only the name and its description and format fields are shown for brevity).

A.8.9 Nontrivial Education `super_cast()`

In Listing 13, we show a Complex transformation between two G-FSTs, where the source represents education as a set of enum strings, while the latter represent it as numbers. Using the set of unique values in the former and the range bounds in the latter, the LLM generates a `cross_type_cast()` that works on real-data values. The reasoning string was generated by the LLM to justify its behavior.

A.8.10 Nontrivial Currency `super_cast()`

```
class currencyvalue(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'A USD currency value'
        self.format = 'Currency Value should be a
            floating point number'
        self.examples = [113524789243.0,
            63497164978.0, 49124317794.0,
            9572240391.0, 1918358283.0]

class currencyinr(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Currency value in INR'
        self.format = 'Currency should be
            represented as a floating point number'
        self.examples = [450000, 370000, 158000,
            225000, 130000]

def cross_type_cast_between_currencyvalue_\
    and_currencyinr(val):
    reason = 'Here, the real-world entity is the
         same, i.e., a currency amount. However, it
         is represented in a different unit. We are
         converting from an unspecified currency to
         INR. As a default, I am assuming the
         source currency is USD. If this assumption
         is incorrect, this mapping would not be
         valid and you would need to adjust the
         source currency accordingly.'
    from forex_python.converter import CurrencyRates
    cr = CurrencyRates()
    conversion_rate = cr.get_rate('USD', 'INR')
    return val * conversion_rate
```

Listing 14: Conversion Between USD and Indian Rupee G-FSTs (only the name and its description and format fields are shown for brevity).

In Listing 14, we show a Complex transformation from a United States Dollar G-FST to an Indian Rupee G-FST. While in App. A.5, we stated only a fixed set of libraries could be used – the LLM ignored this rule and used a completely valid currency conversion library, `forex_python`, to perform conversions using the most up-to-date exchange rate.

A.8.11 Incorrect Weight `super_cast()`

In Listing 15, we show an incorrect Identity transformation from a crop yield weight G-FST in kg/ha (ha=hectare, a unit of area) to a fish yield G-FST in (kg). The LLM incorrectly asserts that the two types are castable and hallucinate in its reasoning. We hypothesize that batching outputs from

```python
class yieldweight(GeneralSemanticType):
    def __init__(self, *args, **kwargs):
        self.description = 'The yield weight in
            kg/ha of an entity'
        self.format = 'Yield weight should be
            formatted as a floating point number
            (in kg/ha units)'

class fishweight(GeneralSemanticType):
    def __init__(self, *args, **kwargs):
        self.description = 'The weight of the fish
            in kg'
        self.format = 'Weight should be a
            non-negative number, representing the
            weight in kg'

def cross_type_cast_between_yieldweight_\
    and_fishweight(val):
    reason = 'Both yieldweight and fishweight
        represent the real-world entity, weight.
        No conversion is required as both are
        represented as float.'
    return val
```

Listing 15: Incorrect Conversion Between yieldweight and fishweight G-FSTs because of differing units (only the name and its description and format fields are shown for brevity).

the LLM and performing a consensus, or using more examples in the prompt, could help alleviate these issues.