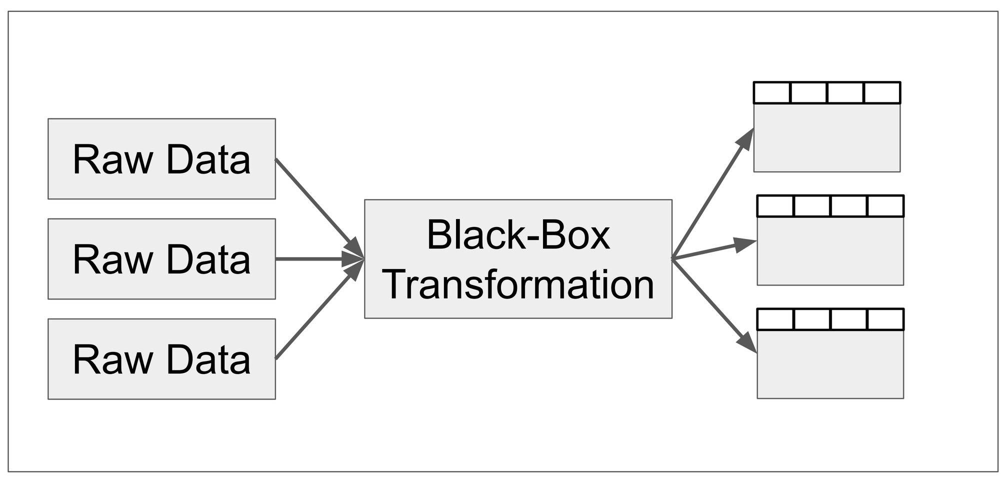
Automating the Generation of a Functional Semantic Types Ontology with Foundational Models

TWO SIGMA

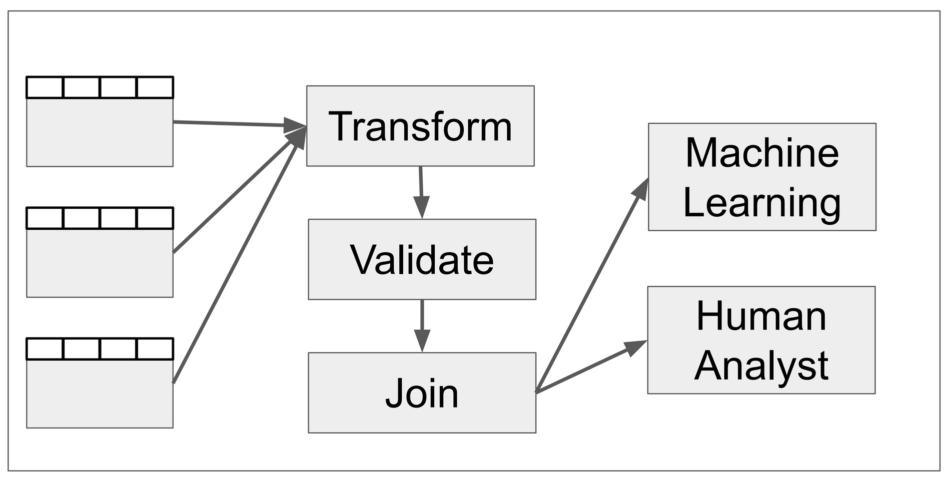
Sachin Konan, Scott Affens, Larry Rudolph

Today's Data

Vendor



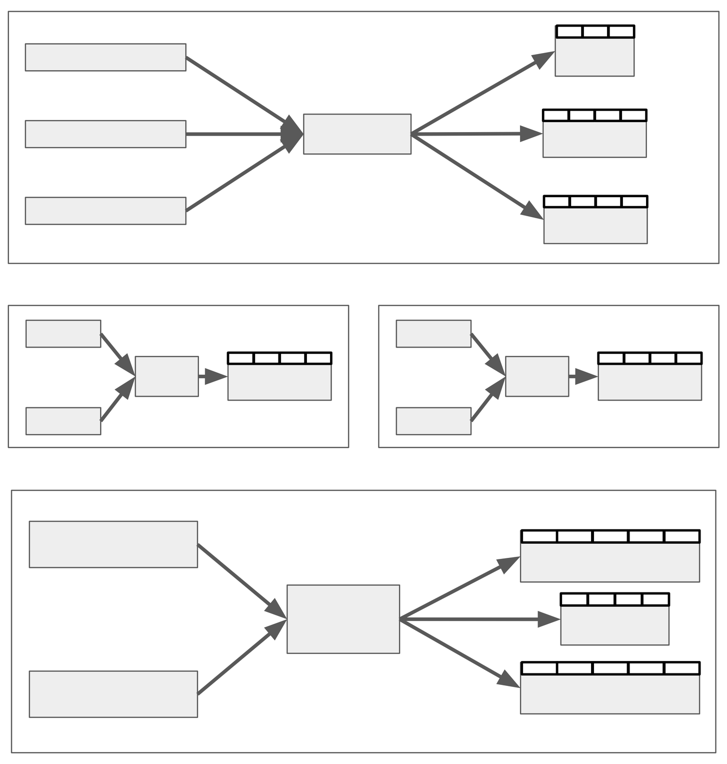
Consumer



- Data goes from vendors to consumers, but context is lost
- Vendors use specific terminology that isn't global
- Consumer has to build custom data pipelines to satisfy downstream needs
  - Fine for individuals, but it doesn't scale

Tomorrow's Data at Scale

Vendors



Consumers

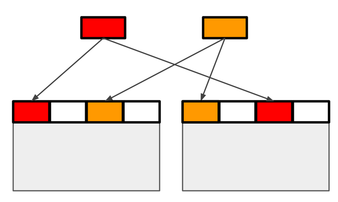
- We are moving towards a world where there are many vendors per consumer
  - Loss of context and inconsistent naming makes ingestion hard

Functional Semantic Types (FST)

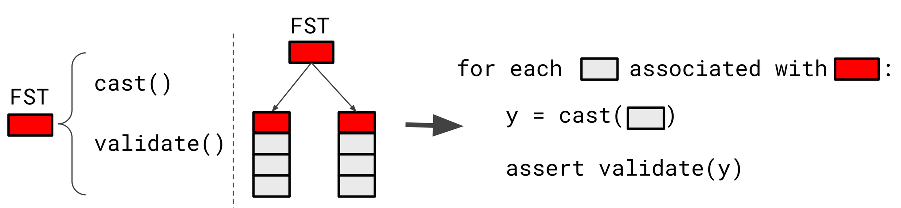
- Semantic Types are "entity tags" that relate a column to its real-world entity
  + Useful for discovery/search
  - Lacking context of data's source, units, and validations
- We introduce **Functional Semantic Types** (i.e. **FST**)
  - Types are represented as Python Classes and have relevant functions
  - Necessary for automating discovery/search/normalization/validation/joining

```
class FST:
    def __init__(self):
        self.description: str = ''  # short sentence describing its characteristics
        self.valid_values: str = '' # short sentence describing the finite domain of values
        self.format: str = ''       # short sentence that describes the canonical format
        self.examples: list = []    # 5-length list with 5 examples

    def cast(self, val):             # Normalize real data value to the canonical format

    def validate(self, val):         # Validate value according to type definition
```

Discovery/Search

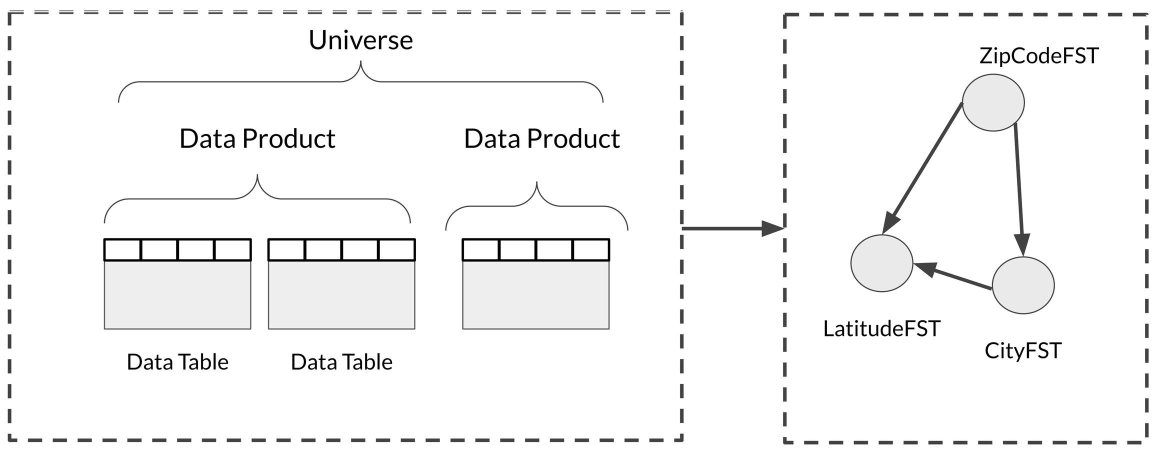


Normalization/Validation



Problem Formulation

- Given a universe of data, extract and generate the FSTs that span all the columns

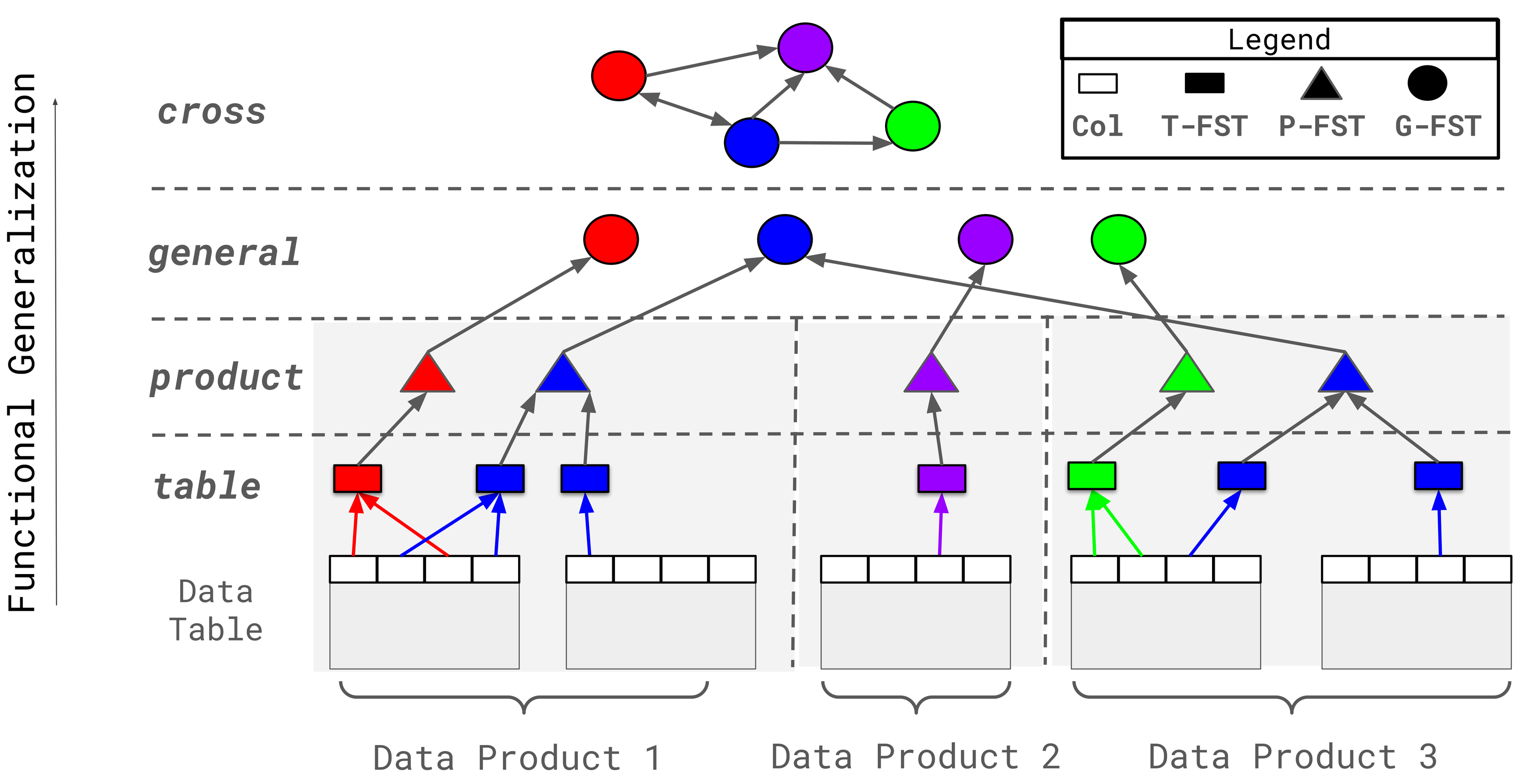


Testing Universes

1. Kaggle: 707 Tables
2. Harvard: 484 Tables
3. FData: 428 Tables

- To generate FSTs at scale, we use Large Language Models, specifically GPT4

FST and Graph Generation

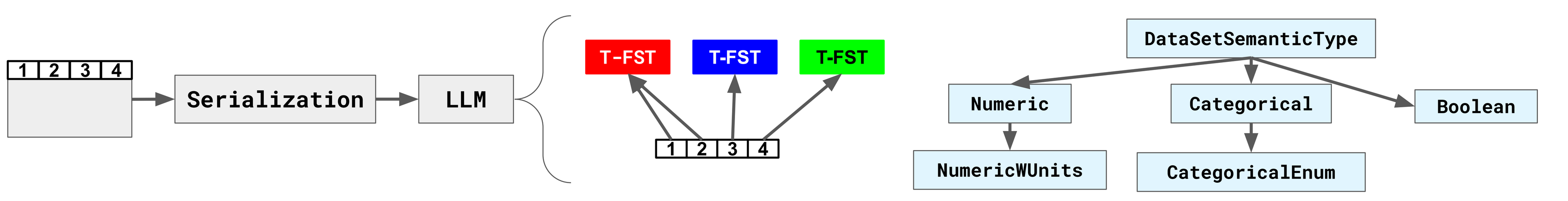


- Our graph represents a hierarchy of functional generalization, meaning:
  - FSTs at the ***general*** layer can normalize more representations of the same entity than those at the ***table/product*** layers
- We use LLMs to generate FSTs and edges from G-FST -> G-FST.

table

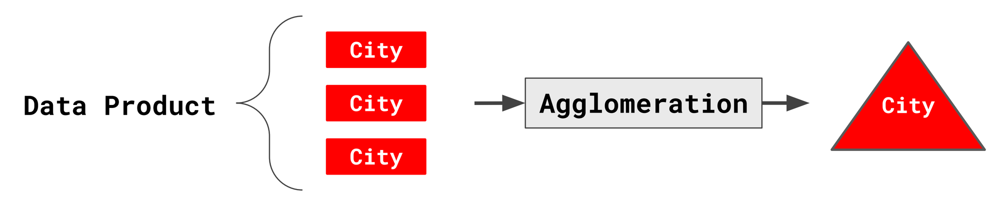
An LLM processes a string-serialized table and:
1. Finds the subset of columns corresponding to an entity
2. Generate T-FSTs for those columns

A T-FST is the most specific subclass of the following class hierarchy:



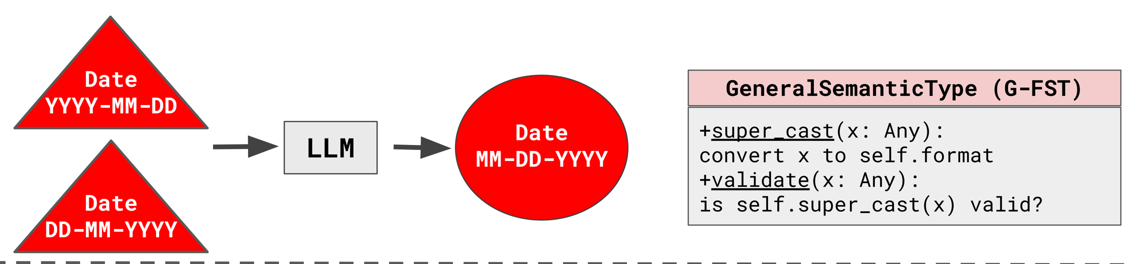
product

Many T-FSTs within a product are redundant, so we select a representative via a heuristic



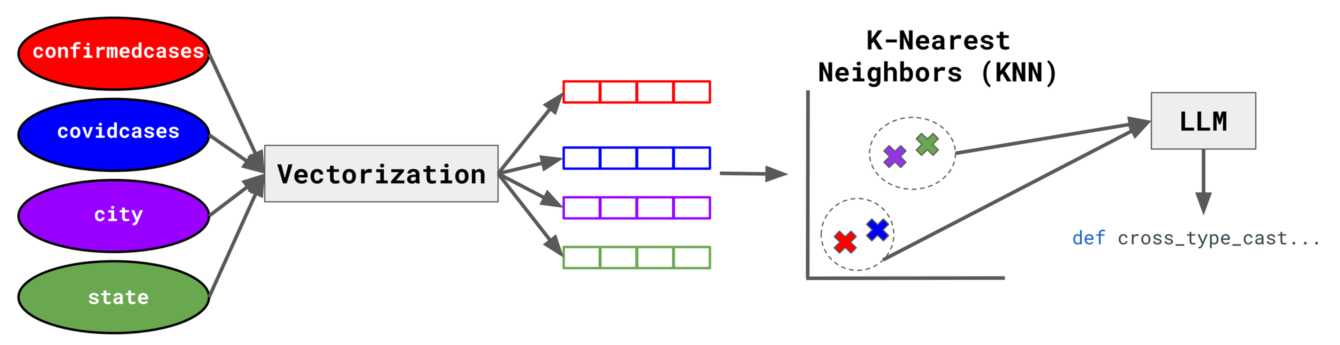
general

P-FSTs may represent same entity, but handle distinct representations
An LLM generates a G-FST

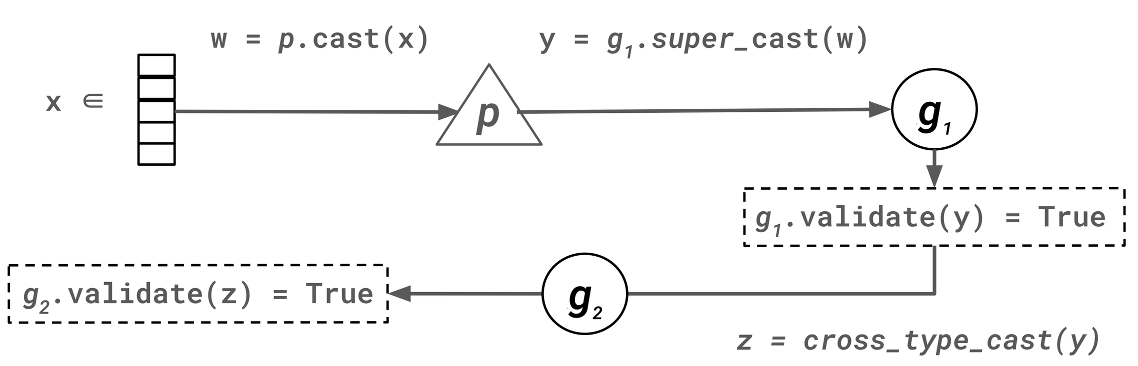


cross

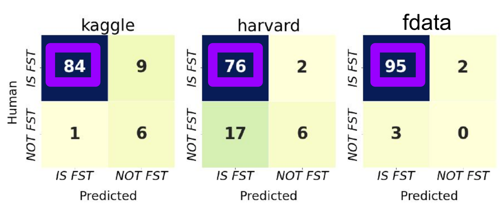
Synonymous G-FSTs are identified with KNN and linked using an LLM
An LLM generates a "cross-type-cast"



Evaluation



- Normalization code had complex behavior, including string normalization, type-casting, external library usage, and more
- The generated code raised *runtime exceptions* in less than 2% of cases across the universes
  - Implies that the code was well-constructed or there were few occurrences of invalid values
- Human evaluation showed that LLMs were **performant at entity detection**, even without a class distribution

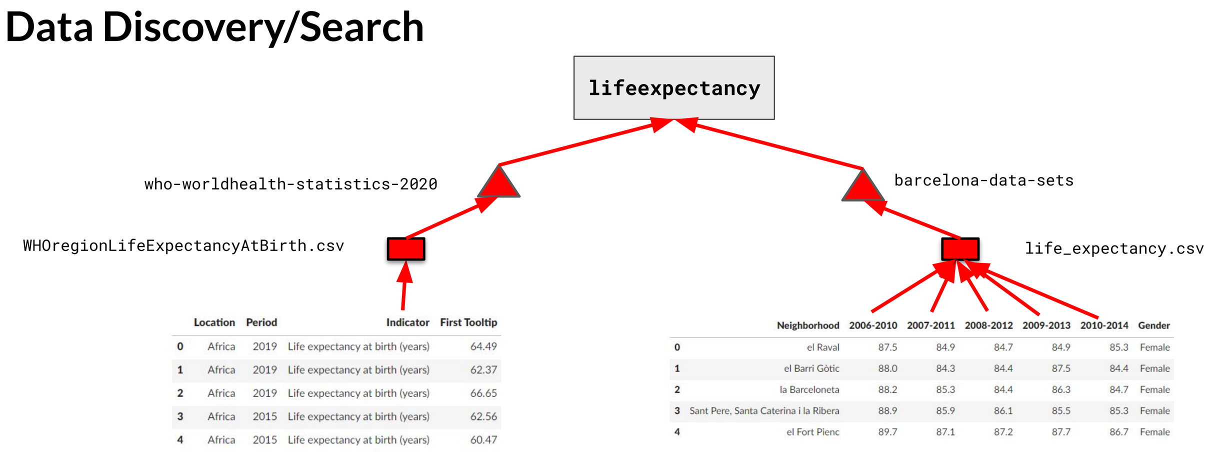


	Incorrect Scope		Correct Scope		
dataverse	Too Broad	Too Specific	Totally Incorrect	Slightly Wrong	Just Right
kaggle	15.44	1.52	2.78	5.06	75.19
harvard	17.54	0.88	1.32	0.44	79.82
fdata	49.07	0	3.27	1.87	45.79

- Additionally, the FSTs were generally well-scoped, but for domain-specific data, **they were too general**

Applications

Data Discovery/Search



LLMs identified a relationship between two tables referring to life-expectancy, but "life-expectancy" wasn't in either header

Data Fusion

```
class currencyvalue(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'A currency value'

class currencyinr(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Currency value in INR'


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
```

LLMs can use libraries to perform complex mappings, like currency conversions.

Data Normalization

```
class incomelevel(GeneralSemanticType):

    def __init__(self, *args, **kwargs):
        self.description = 'Income level'
        self.format = 'Income as a positive number representing
        self.examples = [217500.0, 194000.0]

    def super_cast(self, val):
        if isinstance(val, str):
            if val == 'Less Than 5000':
            elif val == '5000-10000':
                return 0.0
                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')

    def validate(self, val):
        casted_val = self.super_cast(val)
        if isinstance(casted_val, float) and casted_val >= 0:
            return True
        else:
            return False
```

Income is represented differently across tables: string range or float

- LLM reconciled the differences
- Validation check asserts correctness

With transformation logic in functions, it can be easily grepped

Data Validation

```
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 is a floating point.'
        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)


class bodyacceleration(NumericSemanticTypeWithUnits):

    def __init__(self, *args, **kwargs):
        self.description = 'The mean body acceleration
        self.valid_range = [-1.0, 1.0]
        self.dtype = float
        self.format = 'Body acceleration is floating point'
        self.units = 'The unit of body acceleration is 1g'
        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)
```

With wide background knowledge, LLMs can generate meaningful validations

Precipitation: "T" means trace amounts of water

Acceleration: Normalized values should be [-1,1]