# CS294 (SysML) Project: Automated Dataset Assembly for IoT with Knowledge Graphs

Gabe Fierro
UC Berkeley
gtfierro@cs.berkeley.edu

John Yang
UC Berkeley
john.yang20@berkeley.edu

## 1 INTRODUCTION

The "built environment" — the buildings, cities and human-made aspects of our environment — produces an incredible amount of data. This data can be incorporated into a wide array of applications providing monitoring, analysis and control for buildings such as automated fault detection and diagnosis [**?** **?** ] and optimal control policies for subsystems in buildings [**?** **?** ]. Regression models are a common element in many of these applications: predicting when equipment will fail, estimating power usage, building occupancy, indoor or outdoor temperature, and so on.

Constructing regression models for these tasks is a difficult and largely manual and time-consuming process. The process is difficult primarily because of the extreme amount of heterogeneity between buildings. Every building is a "one-off" with its own custom-designed and custom-built architecture, control systems, processes and uses. This is related to the second issue: data discovery. There are hundreds or potentially thousands of data streams in a building. Determining which ones are relevant to a particular regression task requires not just domain expertise but also familiarity with the way a particular building is put together. Finally, we would like the models we develop to be explainable—that is, an analyst should be able to determine the nature of the relationship between the input data features and the predicted output. This can assist in determining causal relationships between data sources in the building.

To address these difficulties, we propose an automated dataset assembly process that can be used as the "frontend" for existing machine learning and statistics platforms that can produce regression models. At a high level, the dataset assembly process takes as input a description of the quantity to be predicted (such as a building's power consumption) and produces a dataset that can be used to train a regression model. The dataset is composed of timeseries data and other features that are discovered through a knowledge graph representation of the building and its data sources.

## 2 BACKGROUND AND RELATED WORK

This work builds upon contemporary research for knowledge representation and data science in the built environment. We provide brief background on knowledge graphs and the semantic web to provide the relevant context for the Brick ontology [**?** ] and the Mortar data platform [**?** ]. We then discuss how the proposed approach builds upon existing dataset discovery mechanisms and automated machine learning platforms.

### 2.1 Knowledge Graphs

Knowledge graphs—or *linked data*—are a form of structured data that represents the content and meaning of data by encoding the relationships between data elements. This is a form of "semantic metadata": it allows programs to reason about the context of data stored in the graph.

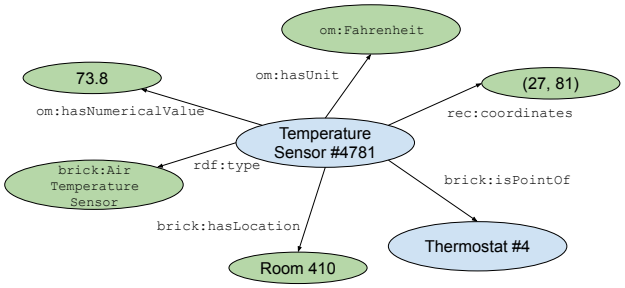


**Figure 1: A linked data representation of a temperature sensor. The name of the property is given by the link name; these adhere to standardized vocabularies, identified by the prefix (e.g. `om:` or `brick:`).**

Figure 1 contains a simplified example of what this context might look like for a temperature sensor. The knowledge graph encodes the current value of the sensor and its engineering units and two types of location for the sensor (room name and building coordinates). Note that the graph explicitly encoedes the *type* of the sensor. This means that the behavior of an entity can be understood independently of what it is called, but more importantly, it means that the definition of `brick:Air_Temperature_Sensor` can be used to infer additional properties of "Temperature Sensor #4781" that are not explicitly provided. The graph also contains a link to another entity ("Thermostat #4") and describes the nature of that link.

This context can be explicitly provided by a data source, or it can be *inferred* by an external process. The process of inference – the generation of new facts from existing knowledge – leverages formal descriptions of knowledge called "ontologies" Ontologies establish the vocabulary, meanings and restrictions for a set of concepts that make up a domain, and the relationships between them. For consistency, most knowledge graphs use property names drawn from several well-known ontologies such as RDF, RDFS and OWL.

### 2.2 Brick

Brick [**?** ][1] is a recently-developed ontology for developing cyber-physical, data-driven applications for buildings. It contains over 700 classes for describing the physical, virtual and logical "things" in buildings (equipment, rooms, sensors, digital control points, building subsystems, and so on) and a set of properties for describing the

[1] https://brickschema.org

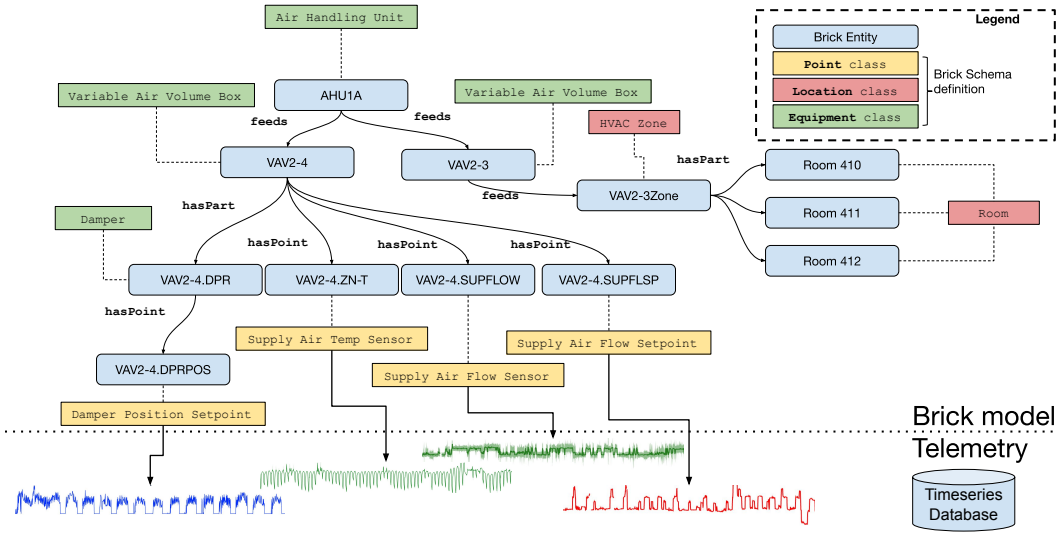


**Figure 2: Small Brick model with linked data. Blue rounded-corner boxes are "entities" in the building; monospace-text boxes are Brick classes with established definitions given by the Brick ontology. Labels on directed edges are properties defined in the Brick ontology.**

relationships between these things. Our approach for using knowledge graphs to establish context for data sources in buildings uses Brick because Brick's vocabulary sufficiently covers the necessary context required by real-world building applications [**?** ].

The top part of Figure 2 contains a small example of a Brick model. A *Brick model* is an instance of a knowledge graph that refers to terms in the Brick ontology. This model represents:

- **Equipment**: `AHU1A`, `VAV2-4`, `VAV2-3` and `VAV2-4.DPR` are all major pieces of equipment in the HVAC (heating, ventilation and air-conditioning) system for a building
- **Points**: the model represents digital data sources (called "points") that are associated with equipment entities. These include sensors and setpoints.
- **Locations**: the model represents three rooms that are members of a logical collection called an "HVAC zone". An HVAC zone is the set of rooms that are all conditioned by the same equipment and schedule.
- **Relationships**: the directed edges in the graph show how equipment is connected (`feeds`), how data is associated with equipment (`isPointOf`) and how locations relate (`isPartOf`)

Recent work on Brick has augmented the ontology with a set of properties about classes that permit inference of how those classes interact with quantities and substances found in buildings (e.g. substances like "Air" and "Water", and quantities like "Temperature" and "Flow") [**?** ]. Our approach takes advantage of these new inferences to provide more expressive graph traversal policies (§3).

## 2.3 Mortar

Mortar is a platform for developing and executing data-driven applications in buildings [**?** ]. It provides an API for applications to interact with and download timeseries data and Brick models for over 100 real buildings. Mortar stores timeseries data and Brick models in separate, dedicated databases (Figure 6) and embeds references to the data in the Brick model. Figure 2 represents resulting abstract data structure.

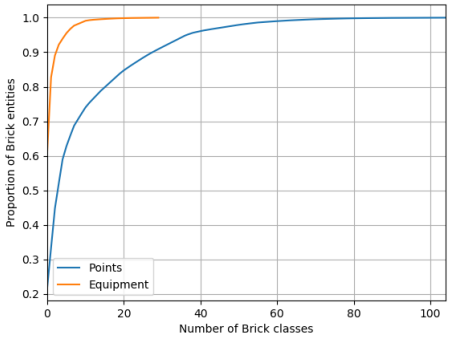


**Figure 3: CDF of the classes of points and equipment found in the Mortar testbed. The most common points are `Zone Air Temperature Sensor` (≈ 4500), `Reheat Valve Command` (≈ 2350) and `Supply Air Temperature Sensor` (≈ 2350).**

The Mortar dataset has over 100 different classes of points (data sources) and over 20 different classes of equipment across the buildings. Figure 3 shows the distribution of how common these points and equipment are. One of the challenges of working with real building data is the heterogeneity of the settings: every building is a custom "one-off" with its own architecture, subsystems and data sources. Contextualizing data sourcees with a Brick model makes it possible to reasona about these differences, and allows applications to discover relevant data.

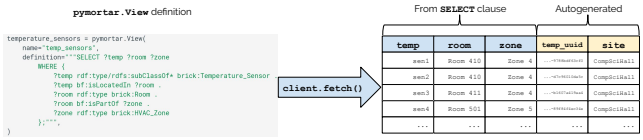


**Figure 4: Example of a Mortar View retrieving temperature sensors and some context.**

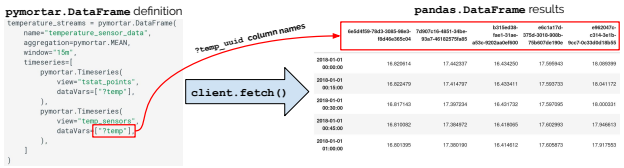


**Figure 5: Example of a Mortar dataframe retrieving data for the temperature sensors identified in the view defined in Figure 4.**

The Mortar API permits queries against the metadata (Brick models) and allows applications to retrieve timeseries data in terms of the metadata. Queries against metadata (called "views") let applications identify data sources which have the required context. For example, the view in Figure 4 is for an application which needs to know the locations of temperature sensors. Downloading data from Mortar is most easily done by referring to the contents of a view: see Figure 5. Our proposed approach will make use of this expressive API to perform progressive data discovery and retrieval.

## 2.4 Related Work

This work touches on several bodies of existing research. Ontology-based data access, typified by systems like ELITE [**?** ] and DALI [**?** ] use knowledge graphs to simplify the process of discovering relevant data sources, but do not address how to separate the metadata (in the knowledge graph) from the actual data that is discovered. As a result, the handling of the data is ad-hoc and underspecified.

This project also builds upon existing work on the featurization of knowledge graphs models. RDF2Vec [**?** ] adapts language modeling techniques to performing unsupervised feature extraction in knowledge graphs. The technique generates graph embeddings using the results of several graph traversal techniques. [**?** ] uses SPARQL queries to generate subgraphs that form the features for a particular node. Our feature extraction technique borrows ideas from both of these approaches, but takes the extra step of including data not present within the knowledge graph.

This project is also related to contemporary research on automated machine learning (autoML), which enables non-experts to take advantage of machine learning by automating the pipeline of cleaning and preparing data through model development and hyperparameter optimization [**? ?** ]. Rather than focus on the model development aspects of autoML, which are already well-addressed in the literature, we chose to focus on the discovery and preparation of data. In many settings, including the built environment, there are few well-established datasets and data preparation techniques that

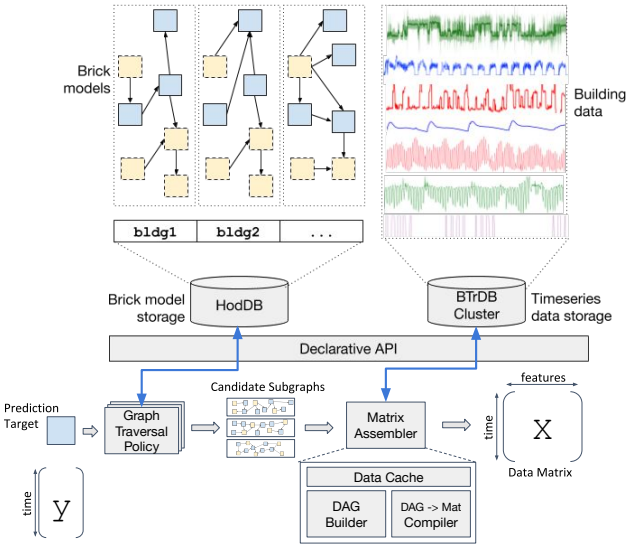


**Figure 6: Architecture of the Mortar platform and the dataset assembly platform**

are known to perform well. This is due in part to the heterogeneity of settings and a lack of data availability. By using knowledge graphs, we aim to make the data that is available easier to use and understand.

# 3 KNOWLEDGE GRAPH-BASED DATASET ASSEMBLY

To facilitate the training of regression models in the setting of heterogeneous buildings, we want to automate the assembly of relevant datasets. Figure 6 provides an overview of the proposed process.

We begin by restricting posible subjects of a regression model (the entity whose future is to be predicted) to be entities in a Brick model. Relevant data sources can thus be discovered by traversing the knowledge graph. Through the use of a *traversal policy*, the set of candidate data sources can be restricted or broadened. The system then retrieves the actual timeseries data for the candidate data sources, and performs cleaning, processing and other featurization of the data according to the operations expressed in a dataflow DAG. Compiling this DAG produces a dataset $D$ with $t$ timestamps and $m$ features.

## 3.1 Input Parameters

The initial input to the system is a description of the regression target. The regression target can be given as the name of a node in a Brick model corresponding to a specific sensor or meter that we want to model, or it may be a set of nodes that are returned as the result of a query against the Brick model. The latter allows the user of the system to be somewhat agnostic to the actual structure of the Brick model, as this may differ from building to building.

The system also incorporates other parameters into the dataset assembly process.

**Figure 7: A selection of graph traversal policies for finding causally related data sources, written as SPARQL queries. `<start>` represents the node in the graph from which the search begins (recursive step). The dot (`.`) is the conjunction operator. Each three-term clause represents part of a graph pattern. Variables start with ?.**

- *Desired resolution of timeseries data.* Data collection in the built environment is often conducted at different sampling intervals; resampling these samples to a common timebase is an important element of data preparation. However, the method of resampling may differ based on the type of data. For example, while samples from a temperature sensor may be averaged or interpolated, discretely-valued data sources (enumerations of device statuses, controller setpoints) may require a different method.
- *Desired prediction horizon.* The length of the desired prediction horizon (i.e. how far ahead to predict) effects the set of data sources and features that are useful to a regression model. Many physical dynamics in buildings—such as temperature—change over longer time periods; for example, predicting the temperature 15 minutes in the future has less to do with the current state of the air conditioning unit and more to do with the state of the air conditioning unit 15 minutes ago.

## 3.2 Knowledge Graph Traversal

After determining the set of prediction targets, the system performs a traversal through the knowledge graph to identify nodes that map to relevant data sources. A traversal policy is a query that defines the subgraph to be traversed. A restrictive policy may miss some possible causally related data sources, but will be faster to execute, and the resulting data sources can offer insight into how well certain classes of data sources perform in regression models. On the other hand, while more permissive policies take longer to execute, they can find relevant data sources without prior knowledge of the structure of the knowledge graph.

Figure 7 contains a number of graph traversal policies, expressed as the predicate of a SPARQL query[2]. Policies are executed recursively, so the data sources returned by a graph traversal for a policy are found in the transitive closure of the policy's predicate. In each policy, the `?thing` variable resolves to nodes that are traversed in recursive executions of the policy and the `?newdata` variable resolves to relevant data sources.

**Equipment-only**. This policy uses Brick's `isFedBy` property, which captures the upstream flow of media in a building, to return data sources that are *upstream* of the target data sources. Brick's properties are agnostic to the actual media, so this policy will find relevant data sources for data sources related to air, water, light, electricity or other media.

**Cross-system**. This policy uses a larger set of Brick's properties to capture data sources related by media flow, mechanical composition or a control relationship—potentially across subsystems of a building (e.g. from HVAC to lighting). This widens the set of potential data sources at the cost of exploring a larger set of the graph.

**Substance-related**. This policy uses the Brick ontology to infer the substances each entity along a traversal are associated with. This is more complex because it has to ask the knowledge graph if each "thing" measures or regulates a given substance.

Traversal complete by reaching a configured maximum recursion depth or by running out of edges to traverse. The system retrieves a configurable range of timeseries data for each node in the discovered subgraph using the Mortar API. This set of timeseries data are the roots of the DAG that defines operations on the data to produce the final dataset.

Figure 8 shows the distribution of subgraph sizes for the *equipment-only* and *cross-system* policies across all buildings in the Mortar testbed. To establish the size of a subgraph, we executed the graph traversal algorithm (the first stage of the pipeline in Figure 6) for all temperature sensors in the buildings. Temperature sensors are found all throughout a building's HVAC and water subsystems, so they may have very few upstream data sources (if they are at the "root" of the subsystem) or very many upstream data sources (if they are at the "leaves" of the subsystem). The heterogeneity of knowledge graph structures is evident in the larger subgraph sizes (and the long tail on the distribution) of the *cross-system* policy.

## 3.3 Dataset Assembly

The dataset assembler produces a dataset $D$ from the data found by the graph traversal process. The dataset consists of a set of $T$ vectors (one for each timestamp) with $m$ features. The $T$ timestamps are aligned to a user-provided frequency which simplifies the model training process, but raises the issue of having to deal with missing data. The $m$ features are derived from a combination of manual and automated methods. These features, and the final dataset, are the result of applying a DAG of operators to the original data and its context.

The system defines a small library of operators that operate on relations and return relations; more formally, they are stateless functions $f(R^n) \to R^m$ where $R^n$ and $R^m$ are $n$- and $m$-dimensional relations. Operators may operate on timeseries data (represented by a time-indexed relation) or on data context (represented by a relation). We consider the following operators:

- Project (relational operator): returns a subset of the features of the input relation
- Select (relational operator): returns the items of the input relation that match a predicate
- One-hot encoder: produces the one-hot encoding of the values in the input relation
- Time features: produces indicator variables for each timestamp in the input relation: e.g. the weekday, the month, the hour, etc. These features are useful for extrating periodic or seasonal relationships from the data
- RDF2Vec (not implemented): Use techniques from [**?** ] and [**?** ] to generate embeddings of the data source position and context in the knowledge graph
- Interpolation (not implemented): offer a family of options for performing interpolation of data sources to estimate missing

[2]SPARQL is the W3C recommended query language for the semantic web.

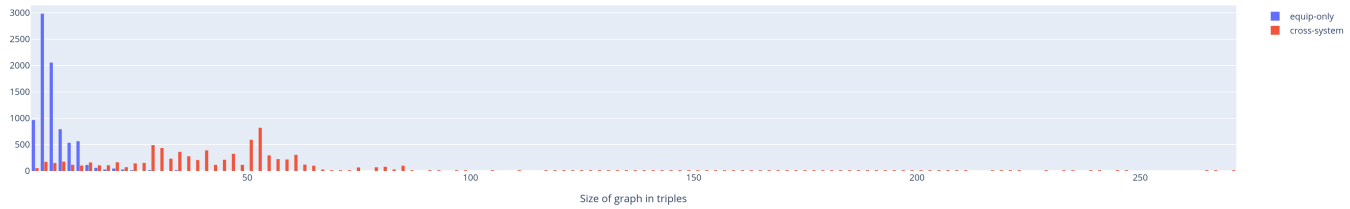


**Figure 8: Distribution of subgraph sizes (in RDF triples) for all temperature sensors in the Mortar testbed under the *equipment-only* and *cross-system* traversal policies.**

data. This may use naive methods (linear or polynomial interpolation) or use existing regression models if they already exist.

Operators are arranged into a DAG where the nodes in the DAG are instances of operators and the edges in the DAG capture the data dependencies. The DAG has one "root" node (with no data dependencies) for each data source found by the graph traversal. All other nodes have at least one output. The set of nodes with no data *dependents* (i.e. at the opposite end of the DAG from the data sources) are all joined on their time index in order to form the final dataset. All features at the same timestep are concatenated in a consistent order.

The library of operators is implemented in Python and uses the `pymortar` package for accessing the Mortar API. Creating the DAG for a particular collection of data sources is currently a manual process, though we have begun developing a set of techniques for automating this process.

One technique takes advantage of the semantic information available for each data source to infer possible featurizations of the data source. For example, because a temperature sensor has scalar values (implied by its `brick:Sensor` type), possible featurizations might extract the minimum or maximum value for the sensor within some time range. However, a status register reflecting the state of a rooftop unit (e.g. heating, cooling, fan-only, off) is an enumeration (implied by its `brick:Status` type), so a one-hot featurization is more appropriate. Another technique uses the parameterization of each operator's relation to determine what operators can be applied. This would allow the automatic generation of DAGs and would automate dataset assembly process end-to-end.

# 4 EVALUATION

To evaluate the current prototype of the system, we conduct a microbenchmark of the system, and then examine the extent to which the discovered datasets contribute to improved regression models.

## 4.1 Microbenchmark

The microbenchmark measures the elapsed time for performing the graph traversal for all 8,201 temperature sensors in the Mortar testbed using the *equipment-only* and *cross-system* traversal methods. Figure 9 is the CDF of response times for graph traversal. While the time to execute the *cross-system* traversal policy is on average twice as slow as the *equipment-only* policy, both policies have a $99^{th}$-percentile response time of under 500ms.

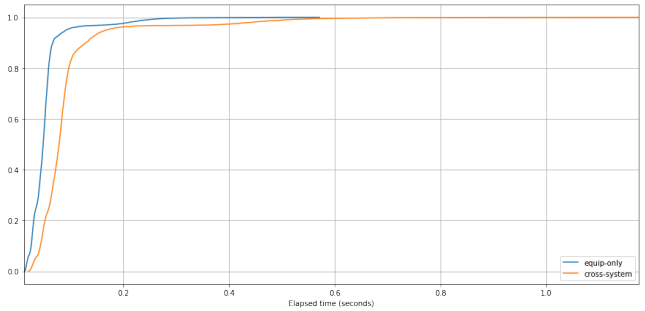


**Figure 9: CDF of elapsed time for performing graph traversal for temperature sensors in the Mortar testbed**

## 4.2 Regression Models

To evaluate the efficacy of the discovered datasets, we trained a set of regression models for predicting building energy usage for a small office building in downtown Berkeley. We hold the regression model pipeline constant and vary the input dataset: previous $n$ timestamps of the prediction target and a dataset derived from an *equip-only* policy. The regression model is a simple scikit-learn pipeline that applies a PCA transformation on the input dataset and passes the output to a ridge regression model. The pipeline uses a grid search algorithm to determine the best hyperparameters these two stages. The *equip-only*-based datasets produced regression models that had a 10-40% lower MSE than the baseline dataset (past $n$ values of the prediction target).

Figures 10 and 11 contain the results of applying a *cross-system*-derived regression model on three different quantities in buildings. Figure 10 is a supply air temperature sensor, which has a great deal of upstream equipment that can assist in developing a predictive model. Figure 11 is a building energy meter, which can be predicted more easily with the context of equipment states. Regression models trained on datasets using the data discovery mechanism have a 5-10% lower MSE than baseline models that only use the past $n$ values. Also, because the regression models incorporate state from other data sources in the building, these models can be used to estimate values in hypothetical scenarios (e.g. how hot does the building get if all thermostats are calling for heat).

# 5 CONCLUSION

This report has presented a system for automatically assembling datasets for use in training regression models in IoT settings. The

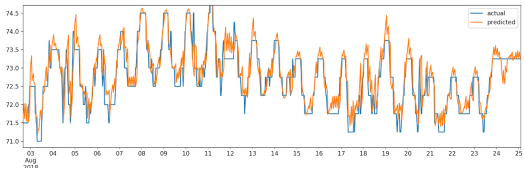


**Figure 10: Results of regression model for a supply air temperature sensor**

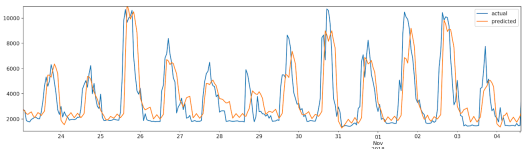


**Figure 11: Results of regression model for a building's energy consumption**

approach uses a knowledge graph to represent data sources and their context, which allows data sources to be discovered in terms of their relationship to the quantity that is being modeled.

This project focused mainly on creating the infrastructure that makes the dataset assembly possible. Future work will refine the design of the operators (possibly building upon existing Python-based dataflow methods) and use a wider array of regression models (LSTMs, random forests, and so on) to better evaluate how well the created datasets perform.