

Full Paper for ACM SenSys Proceedings

Abstract

More than 70% of commercial buildings have giant sensor networks to monitor different aspects of building performance. However, most of these deployed sensor networks have little metadata describing context, which precludes writing analytics applications without familiarity of the particular building’s sensor metadata structure.

In this paper, we propose a technique which learns how to transform a building’s metadata to a common namespace by using a small number of examples from an expert. Once the transformation rules are learnt for one building, it can be applied across buildings with a similar metadata structure. We illustrate this on a testbed consisting of 60 buildings comprising more than 20,000 sensor points. We also illustrate how this common namespace can help a user write analytics applications that do not require building-specific knowledge, and also scales across different buildings.

1 Introduction

Buildings are sites of very large sensor deployments, typically containing up to several thousand sensors reporting physical measurement, continuously. Moreover, with the recent interest in reducing building energy consumption and increasing their efficiency, it is important to consider ways to quickly bootstrap a set of building data streams into an analytical pipeline. These analyses consist of jobs that help give deeper insight into the overall performance of the building, determine where there are opportunities for energy savings, and discover broken sensors. However, current ‘point’ naming conventions form a bottleneck in the scalability of the data integration process. A ‘point’ refers to a physical location where a sensor is taking measurements. Each build-

ing vendor uses their own naming scheme and unique variants of each scheme are implemented from building to building; variations exist even across buildings that have contracted the same vendor. This makes the integration process laborious for building experts and a non-starter for non experts. Moreover, the process is fundamentally unscalable. If we want to quickly run the job across many sites we need to explore methods automatically normalizing the data and boosting the existing set of metadata per point. This will allow wide searchability of points across many buildings at once.

Consider a simple analysis program, which has the ability to identify anomalous readings from a specific kind of sensor. To execute this job, the process organizes each sensor by type and location, organizes the distribution of readings across types and locations, and identifies broken sensors where some fraction of their readings are above some threshold value on the distribution. In order to run this application the job needs to know the names of each sensor, its type, and its location. This information is typically encoded in the name of the sensor or ‘point’, as is common in the building-system nomenclature. For example, the point `BLDA1R435__ART` is constructed as a concatenation of common codes. The name of the building (first 4 characters), the air handling unit identifier (the fifth character), the room number (R435), and the type ART (area room temperature) – which indicates that this measurement is produced by a temperature sensor. In addition, to point names there may be some descriptive metadata. This extra information is also useful.

However, point names do not always follow the exact same structure within and across buildings and certainly do not follow the same convention across vendors. Because each point is named by a human, the names can vary. This makes it difficult to construct a general set of rules for name construction. In this paper we aim to boost the existing metadata by learning the rules of construction through a programming-by-example approach where the user provides some input-output examples to boost the existing metadata with extra terms. We can then search across the boosted metadata to increase our search harvest over time.

In order to meaningfully deal with disparate building streams in a scalable fashion the streams should be

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SenSys’13, November 11–15, 2013, Rome, Italy.
Copyright © 2013 ACM 978-1-4503-1169-4 ...\$10.00

searchable across various properties, such as building name, room location, and type. Moreover, we assert that wide searchability is necessary for achieving scalability. By providing a tool for searching across building streams, we minimize the deployment time for applications that allowing them to be used in *all* buildings, not just a single one.

In this paper, we propose a set of techniques which learns how to transform a building’s metadata to a common namespace by using a small number of examples from an expert. Once the transformation rules are learnt for one building, it can be applied across buildings with a similar metadata structure. We also show how processes that extract statistical features and generate descriptive metadata can help unify data streams with respect to much deeper attributes. We show how our approach makes it easier to write applications across buildings by demonstrating its use by three different applications: 1) a rogue zone detector, 2) a broken sensor finder and 3) an application that identifies and ranks the most comfortable rooms. We illustrate these on a testbed consisting of nearly 60 buildings comprising more than 20,000 sense points. We also illustrate how this common namespace can help a user write analytics applications that do not require building-specific knowledge and scales across different buildings. We believe this is an important study given the recent trends in the penetration of the internet of things in our home and our technique can be used to unify that data for broad search and exploration of new applications.

2 Motivation

Buildings are notoriously complex from a management perspective. They consume a large fraction of the energy produced in the United States and much of is wasted [?]. There has been much work in the building science community to reduce their energy consumption and make them more efficient, but the route to broader impact is typically carried out through regulations guided by the findings of studies in those communities [?]. We aim to let solutions reach buildings *directly* by making sense of the data they produce as quickly and accurately as possible. In order to achieve this at scale, we must explore ways to deal with the data produced from sensors within them and to enable broad analysis across several buildings at a time. Our study focuses on any building equipped with a network of sensors. Nearly three-quarters of commercial buildings contain a rich sensing fabric, installed as part of the building management system [?]. It is the data from these system and variants of it, that we wish to unify and make sense of in a more systematic and automated fashion.

The data parallels the complexity of buildings. Ad-hoc data management practices make it difficult for any analytical solution to be widely ported or run across building systems. For example, consider the following stream names: BLDA1R435__ART, BLDA1R435__ARS, BLDA1R545__ART. Each name encodes contextual infor-

mation in the form of concatenated character sequences. In these, the first 4 characters refer to the name of the building, the next one encodes the air handling unit association, the next four encode the room, and the last three encode the acronym for the type. Although the examples given are well structured, many variants within the same data set exist. For example, BLD_1R435_ARG_ is the encoding for a different sensor in the same room as the others, but with a name that is *like* although not exactly the same structure as the others.

When dealing with a small number of points such differences are usually not a problem. Upon visual inspection, the two encodings are similar enough that the engineer can decode the meaning. However, for automatic processing or processing a large number of points across buildings, these kinds of variations makes it difficult to generalize the character-construction rule set. *Fundamentally, full coverage is attainable if we could learn all the codes and map them to more descriptive search terms.* Since many of the codes change from across vendors and buildings, Analytical applications need to make building-specific adjustments in order to run the same analysis job across each building. For example, if we are trying to determine the fraction of uncomfortable rooms in each building on our campus, the two collections of sense points we need are the temperature sensors and the corresponding set points in each building. Because the code for temperature sensor and the code for set point is slightly different in each building, the query must be adjusted each time the job is set to run on the data from a new building.

At a high level we want to normalize the metadata across all buildings so that one query can run across all of buildings, and we want the normalization process to be automatic, so that the overhead in adding a new building to the set is small – the latter of which is important for achieving scale. Fundamentally, a tradeoff exists between the degree to which we can automate the normalization task and the level of coverage you get in the general rule construction. You can get full coverage with no automation. This is essentially the approach that is common today. Every solution that incorporates a building’s data is manually adjusted, specifically for that particular building. You can get some degree of coverage if you use a general set of rules, increasing the automation factor and decrease the manual one. Tuning is quite arduous and in most cases, non-programmers are tuning the data ingestion script.

The problem for a large number of buildings is particularly pernicious to the goal. Although there are some similarities for certain kinds of sensor labels and metadata, searching across them yields results of varying success. Consider the simplest kind of a search, a grep scan across the metadata associated with the points across a set of buildings. If we wish to attain all the temperature sensors or set points for a particular building, without knowing these codes, we should be able to attain it by search for all points in a particular building that measure “temperature” or that have “setpoint” in

their description.

We demonstrate this by collecting all the metadata across our building testbed. The testbed consists of almost 60 buildings and over 20,000 sense points. We collected the names for each of the set points and any associated metadata that describes the meaning of the name. Then, we run a number of grep searches on the data and calculate the relative success of the query. Figure 1 shows the results for two queries. The ‘temperature’ grep string is “buildingA room temp” and the ‘setpoint’ grep string is “buildingA room setpoint”. For the temperature query we attain 87.5% of the actual sense points we are searching for, while for the setpoint query we attain only 56% of the setpoints in the building. We could try different search queries that yield different results, however, because there’s only *some* overlapping metadata terms across buildings, coverage tends to be quite poor.

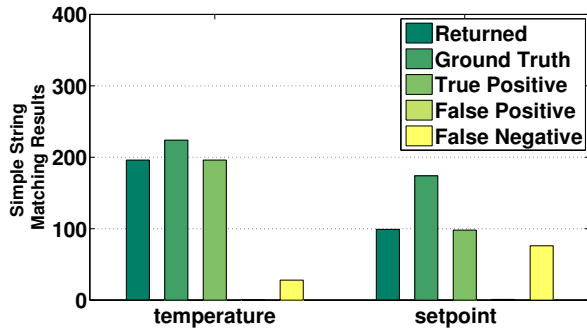


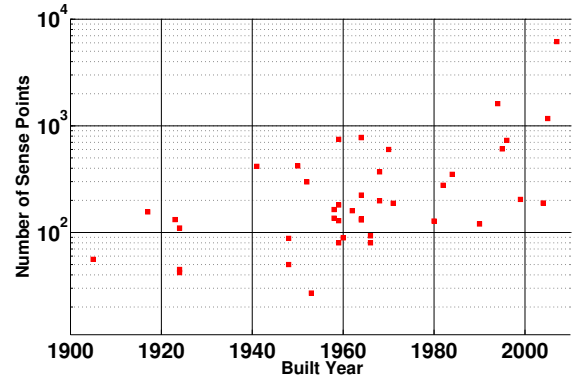
Figure 1: Results when running a grep search on the point names in a single building.

A key observation we have made while attempting to write ingestion scripts in that we create a program P_1 to that parses the data and generates a set of point names n_1 with coverage c_1 of the points we need to obtain. We notice there are points missing, so we write another program P_2 that generates another set of points n_2 which includes some of all of the points in n_1 and gives us coverage c_2 . We keep noticing there are points missing and keep either expanding or adjusting the program to get closer and closer to full coverage. In practice, the coverage is not easily attainable without an expert, familiar with the data set, inspecting the results. So the challenge is to attain the good coverage in one (or a few) buildings using input-out examples, using a technique similar to the work by Gulwani et al [1], and use this the resulting program and a few more example in each building to have the process learn the variants of all the codes and boost them with extra tags that are learned from other buildings.

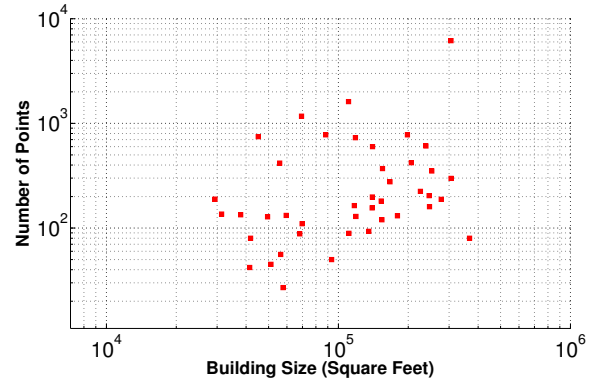
This process allows us to leverage the knowledge of the expert while minimizing the integration overhead per building. Moreover, as more examples the provided, the better that algorithm can learn how to cover a broader set of codes and boost them with common

tags. Also, the technique used in previous work is well suited for our problem, since users tend to be non-programmers. The input-output example interactive model is well suited for interacting with non-programmers and getting the kind of information we need from them to automate the program creation process to learn the various point code.

In our experiments we used an extensive building testbed that consists of 56 buildings containing over 22,600 sense points. These buildings represent a vast range in age, size, and density of deployment. It also represents deployments that were set up by more than one vendor. As expected, newer buildings have many more sense points than older ones – although some old buildings that have been retrofitted have over 1000 points within them. The general trend in the number of sense points versus the year the building is built is roughly linearly increasing in the log of the number of points, as shown in Figure 2a.



(a) Number of Points vs Year Built



(b) Number of Points vs Building Size

Figure 2: Relationship between the built year and the building size on the number of sense points. This data is summarize from a testbed that we used for experiments that consists of almost 60 buildings and over 20,000 sense points.

The maximum number of points in a single building is 6169 and the minimum is 27. The built years spans over 100 years – from 1905 to 2007. The size range

spans over an order of magnitude in square footage from about 30,000 square feet to over 360,000 square feet. There is no observable correlation between the size of the building and the number of sense points. Figure 2b shows a log-log plot of the number of points versus the size of the building. There is one large building with many sense points, but the size seems to have little to do with the density of the deployment. The access this kind of breadth of system and building types makes our study unique.

3 Automating Metadata Acquisition

In this section, we go into detail about how we apply program synthesis techniques and the input-output model of interaction to extract enough information from sensor names to enable sufficient coverage of sensor applications. We first introduce some basic terminology we will be using throughout this section, followed by an overview of the synthesis technique in Section 3.2. We then describe how we adapt the technique to the context of sensor name qualification in Section 3.3. We finally evaluate our adapted technique on our testbed in Section 4.

3.1 Terminology

The expert is expected to point out (*Tag Name*, *Tag Value*, *Value Type*) tuples in the sensor name. A *tag* is mapped on to a substring of the sensor name, which is called its *value*. A tag can have a constant or a variable value. A value should be regarded a constant if it is not specific to that particular sensor.

Sample Input: Suppose the expert is presented with an example BLDA1R465__ART, he should qualify it in order as

BLDA1R465__ART : (site, BLD, const), (ahu, A, const), (ahuRef,1, var), (zone,R, const), (zoneRef, 465, var), (zone air temp sensor, ART, const).

In this example the **site** tag's value is BLD, which is not specific to that particular sensor. Hence, the expert should mark it as a constant. On the other hand, the value of the **ZoneRef** tag is specific to that sensor, and hence should be marked as variable.

Sample Output: The synthesis technique should then be able to identify the tags in a new sensor name automatically. For example, given the sensor name BLDA5R234__ART, it should output the set of tuples shown below:

BLDA5R234__ART : (site, BLD), (ahu,A), (ahuRef,5), (zone,R), (zoneRef,465), (zone air temp sensor,ART).

We term each of these tuples as a *qualification*, because it qualifies a set of alphanumeric characters into normalized metadata tags. We term the output as a *full qualification*, if every alphanumeric character in the sensor name was *qualified* by the set of outputted *tags*.

3.2 Synthesis technique overview

In this section, we will first describe the high-level logic of the synthesis technique in [?] which synthesizes simple regular expressions to transform input columns of a desired spreadsheet to a desired output column. We

String expr P	$:= \text{Switch}((b_1, e_1), \dots, (b_n, e_n))$
Bool b	$:= d_1 \vee d_2 \vee \dots \vee d_n$
Conjunct d_i	$:= p_1 \wedge p_2 \wedge \dots \wedge p_n$
Predicate p_i	$:= \text{Match}(v_i, r, k)$
Trace expr e	$:= \text{SubStr}(v_i, p_1, p_2)$
Position p	$:= \text{Cpos}(k) \mid \text{Pos}(r_1, r_2, c)$
Integer expr c	$:= k \quad (\text{integer const.})$
Regular Expression r	$:= \text{TokenSeq}(T_1, \dots, T_n)$
Token T	$:= C+ \mid \text{specialToken}$

Figure 3: Language for learning substring extraction

will then describe the set of specific techniques to adapt this to our context.

The Algorithm:

The main aim of the technique is to learn two sets of information from the inputted examples — (a) whether a string transformation is applicable on a particular input, and (b) what is the set of regular expressions that transform the input string to the output string.

From each user-provided example, the set of all expressions from the language (shown in Figure 3), that could extract the required substring from that input is computed. If there are multiple input-output examples, the substring extraction rules of the multiple examples are intersected to obtain a more concise set of expressions. If the extraction expression sets cannot be intersected, they are maintained as two disjoint sets, which we shall hereby term as a *partitions*.

Finally, for each disjoint set of extraction expressions, a boolean classifier is built in the DNF form, to differentiate the examples in one partition to examples in all other partitions. When a new string is given to this tool, the classifier is applied on it to figure out which partition the new input falls into, and the corresponding set of transformation expressions are then applied on it.

The Language:

The top level expression of the language is the classifier — the **Switch**(b_i, e_i) function, which applies the substring expression e_i to the input only if it matches the boolean expression b_i . The boolean function is in DNF form and is composed of predicates of the form **Match**(v_i, r, k), which evaluates to true, iff the input v_i has k occurrences of the regular expression r .

The Substring expression **SubStr**(v_i, p_1, p_2), evaluates to the substring between positions p_1 and p_2 of the string v_i . **CPos**(k) denotes the position k in the substring. A position expression **pos**(r_1, r_2, c) when applied on a string s evaluates to a position t in the subject string s such that r_1 matches some suffix $s[0..t]$ and r_2 matches some prefix of $s[t..l]$ (where $l = \text{Length}(s)$). Also, t is the c th such match starting

from the left end of the string. The regular expressions are either just a single token τ , or a token sequence, **TokenSequence**($\tau_1.. \tau_n$), or ϵ (which matches the empty string). The tokens τ range over some token classes, e.g one token for alphabetic characters, one for numeric characters, and one for each special character.

We provide a couple of examples to elucidate how this technique works.

Example 1:

Consider, again, the sensor name **BLDA1R465__ART**. The substring **ART** can be obtained, among other expressions, by either of the following language transformations : **SubString**(s , **Cpos**(11), **Cpos**(14)), or **SubString**(s , **Pos**(**UnderscoreToken**, **AlphToken**, 1), **Pos**(**AlphToken**, ϵ , 1)).

Example 2:

Suppose the synthesis algorithm has seen two examples **BLDA1R465__ART**, whose desired output is **A** at index 3, and **BLD__R479__ART**, whose desired output is **479**. One of the possible expressions that the synthesis algorithm can come up with is : **Switch**((b_1, e_1), (b_2, e_2)), where $b_1 = \text{Match}(s, \text{AlphToken UnderscoreToken}, 1)$, $e_1 = \text{SubString}(s, \text{Cpos}(3), \text{Pos}(\text{AlphToken}, \text{NumToken}, 1))$, and $b_2 = \text{Match}(s, \text{UnderscoreToken AlphToken}, 2)$ and $e_2 = \text{SubString}(s, \text{Cpos}(6), \text{Cpos}(9))$.

In general, there can be many expressions in the defined language that can obtain the desired substring, and provide a classifier to specify which types of inputs each type of substring extraction should work on.

3.3 Adapting to our context

The intuition behind adapting this synthesis technique to our context is that we can consider each *tag* as a potential output for a given sensor name. From each given example and for each tag in that example, we can compute the set of all expressions from the language that could extract the required substring from that sensor name. Also, such a technique could learn a boolean classifier such that all tags that are present in the example evaluate their **Match** condition on this example to True, and all the tags that are not, evaluate theirs to False.

If the same tag is present in multiple examples, the tag's new extraction expression is the intersection of it's extraction expression sets for each of those examples. This may result in disjoint *partitions* for each tag. Finally, a boolean classifier is built to differentiate examples of each partition from all other provided examples.

However, directly adapting this technique to the sensor naming runs into problems. We conducted an experiment where we provided examples for sensor names in a building containing 1585 sense points, applying the technique with no adaptation. Sensor names were chosen at random from the corups of all sensor names, and a full qualification of it was provided as the next input. Figure 4 shows the result of our experiment.

We would expect the percentage of sensor names to increase with each added example. We find this trend up to around 25 examples, after which the synthesis technique started adding qualifying substrings of the sensor

name to erroneous tags. At closer inspection, it turns out that the tokens (one for alphabetic characters, one for numeric characters, and one for each special character) used by the regular expressions and the **Match** predicates were not expressive enough to capture the difference of applicability of different tags.

To illustrate the problem, consider the examples

Example 1 : **BLDA4S1831_STA** : [(site, **BLD**, const), (ahu, **A**, const), (ahuRef, 4, var), (supply fan, **S**, const), (supply fanRef, 1831, var), (status point, **STA**, const)] ; and

Example 2: **BLDA3R5871_VAV** : [(site, **BLD**, const), (ahu, **A**, const), (ahuRef, 3, var), (zone, **R**, const), (zoneRef, 5871, var), (vav, **VAV**, const)]

Both these sensor names have the exact same arrangement of numeric and alphabetic characters, and special symbols, and no regular expression used by **Match**(v_i, r, c) would be able to discern between the two. The result would either be both tags being mapped to the same substring in an effort to qualify it, or neither. Hence, we modify the existing language and set of tokens to stay general enough, yet be more expressive.

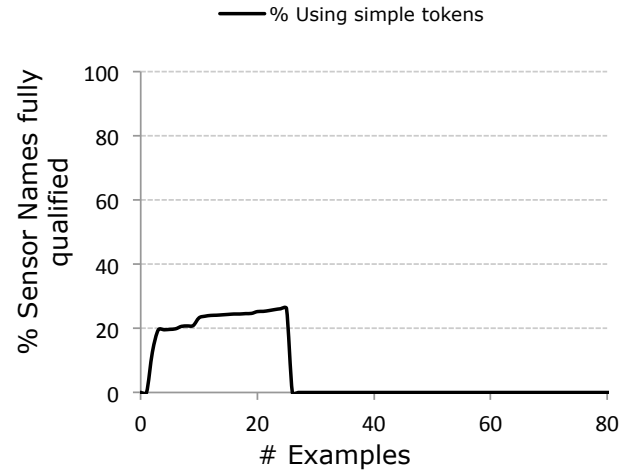


Figure 4: Number of sensor names fully qualified with a token set consisting simply of alphabetic characters [A-Z], numerals [0-9], and one token for each special character.

3.3.1 Challenges

Adaptation of the synthesis technique to our context involves certain challenges. First, the number of tags required to fully qualify an entire building might be large, whereas certain tags may be applicable on a very limited number of sensor names. In such a case, the **Match**(v_i, r, c) expression for each tag should be expressive enough to differentiate a small group of sensor names from the remaining.

Second, care should be taken while merging the **SubString** rules for each tag.

Second, a building may have 1000s of sensor points, making visual inspection of correctness of sensor name

qualification very hard. Incorrectly qualification of sensor names can be mitigated by being more conservative with the boolean preciates in the top level **Switch** operator.

Below, we list five techniques that augment the basic language shown in Figure 3 to enable correct classification

3.3.2 Token Set

Different conventions of sensor naming from building to building precludes us from having an apriori set of special tokens to increase the effectiveness of our regular expressions. The new tokens need to be general enough. Hence, we utilize each constant tag value in the examples provided by the expert as different tokens, in addition to the normal tokens described above.

3.3.3 Splitting constant tags

It may happen that a tag has multiple constant values for different points in the same building. For instance, in our test building, the *zone* tag was applicable either as the constant character R or the constant character C. We treat the same tags which maybe represented by two different constant string to be two different tags altogether. This helps in building better classifiers for the tags.

3.3.4 Modifying Boolean Classifiers

Since manual inspection of the qualification of all the points is not feasible, we take steps to strengthen the boolean classifiers corresponding to each tag. We implement different strategies for tags that have constant and variable values. If a tag has a constant value, we ensure that each clause in the DNF expression used by the **Switch** operator has existence of the constant value in the sensor name as the first boolean **Match** operator. This ensures that if a tag is evaluated to be applicable on a string, it has some chance of extracting the correct substring.

Also, in cases where a tag which has a variable value, and is a reference to another tag with a constant value which is applicable on the same string, we mandate that the constant tag also have been deemed applicable on the same sensor name. For instance, in the example BLDA1465_ART, the tag **zoneRef** which maps to 465 has an indicator tag **zone** which has a constant value. Thus, we would only evaluate the **Match** expressions for **zoneRef** iff the the **Match** condition for the **zone** tag has also evaluated to true.

3.3.5 Modifying Matching expression

To make the classifiers more general, so that regular expressions learnt for one building may be applicable to sensor names of other buildings with a similar naming convention, we augment the language to include Match arguments where it evaluates to true iff the regular expression is satisfied at position equal to k. For instance, a building could be labelled ANV, and the expression learnt for air handler would not work because the token A will be matched more than once.

4 Evaluation of Learning By Example

In this section, we gauge the effectiveness of our learning by example technique by evaluating the number of examples required to qualify labels in two large commercial buildings.

4.1 Testbed

We manually generated ground truth data for all the points in two buildings whose building management system was installed by different vendors. Building 1 has 1586 sensor points and was built in the 1990s. Building 2 was built in the 2000s and has 2551 sense points. The label characteristics of the two buildings are shown in Figure 5.

Figure 5a shows that in these two buildings, a few labels (about 20 in each building) frequently appear in a lot of sensor names. This is pretty common in commercial buildings, where a majority of the points are related to zone or room information. For instance, Building 1, has a room setpoint sensor, an airflow sensor and temperature sensor for each of its more than 200 rooms. Each of these points have a label indicating that they are a room, and a room number. These most frequent labels also fully qualify a large number of the sensor names in both buildings. In other words, learning proper classifiers and qualifications for about 20 labels could yield a full qualification for 80% of the sensor names in both these buildings.

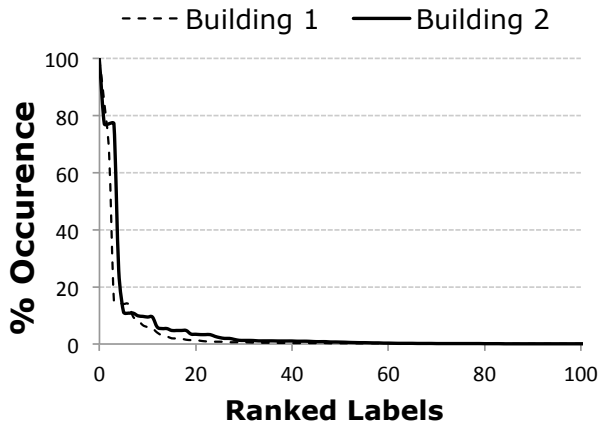
The distribution frequency of labels also has a long tail. For both the buildings, the labels corresponding to *site* and *zoneRef* are most common. However, the distribution of labels show that there is also a long tail. These comprise of building specific sensors, alarms or status variables. There are also a lot of incomplete or inconsistent labeling of sensor names, the labels of which fall in this long tail. Thus, one of the main objectives of the learning algorithm is that it does not learn wrong classifiers for labels based on sensor names that fall in this long tail.

4.2 Choosing the Next Example

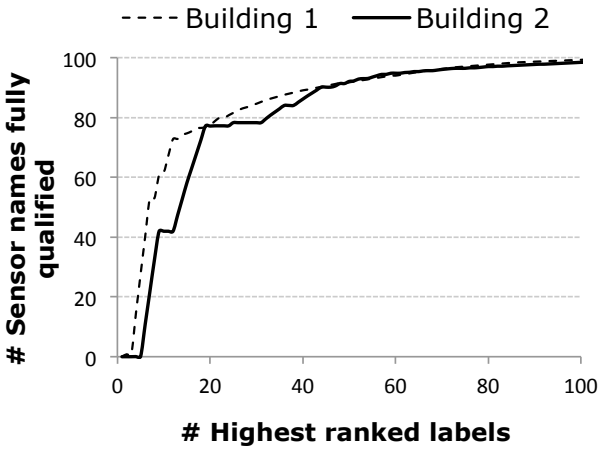
The large number of sensors in a building pose a challenge in selecting the next example to present to the expert. First, the expert might not always be able to browse through all sensor points to check correct qualification. Also, an expert might visally not be able to discern which points would add the most amount of information to the learning process.

This process can be facilitated by the internal notion the learning algorithm has of how much of each sensor name it has been able to qualify. This notion is maintained by simply comparing the sensor name to all the labels that the algorithm applied on it. Note that, the algorithm can incorrectly apply labels on a sensor name, which may lead to erroneous conclusions. The incorrect application of labels might be due to incorrectness or incompleteness of the boolean matching expression or the string extraction regular expressions.

We implemented four different generators to evaluate which example should be provided next to the expert:



(a) Percentage of sensor names each label appears in. The x-axis is sorted according to the frequency of occurrence of a label



(b) Percentage of sensor names fully qualified by the highest ranking labels. A point (x,y) indicates that y sensor names could be fully qualified by using labels ranked $1 \dots x$

Figure 5: Characteristics of labels from two buildings we generated ground-truth data to test our learning technique

Random: This generator just finds at random the next example to present to the expert. While choosing the example, the random algorithm chooses among the set of sensor names which it feels it has not been able to fully qualify.

MinRemaining : This generator chooses the example, that according to our tool, has the minimum string length left to qualify. The intuition behind this is to gain more concrete knowledge about a small number of labels.

MaxRemaining : This chooses the example, that the learning technique feels has the maximum string length left to qualify. These examples would help the learning technique gain coverage over the space of un-

seen labels. The more labels the learning technique knows, the more sensor name information it will be able to qualify.

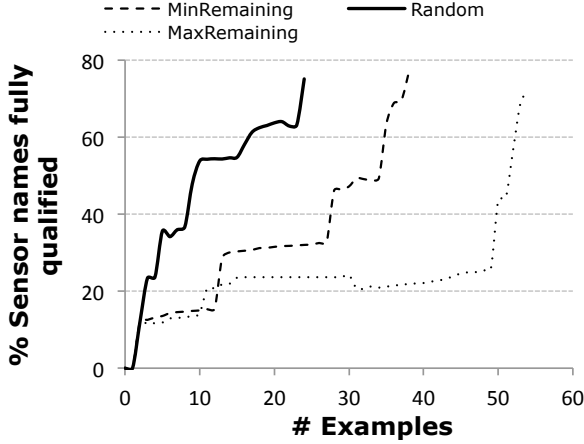
Self-Correcting : There are some sensor names where the learning algorithm can itself figure out that it has incorrectly qualified a sensor name. There can be three such indicators. First, for a sensor name which has matched its boolean classifier, none of its left or right position regular expressions is applicable. Second, if the sensor name has been qualified with labels that overlap over the same substring. Third, the learning algorithm can get a notion of the labelling uncertainty of a sensor name, if its qualification labels change drastically when a new example has been added. This generator gives the expert the examples that satisfy the most number of these three criteria. Once, none of the points satisfy these criteria, this generator defaults to the MinRemaining generator.

We wrote a script that automatically gave the learning algorithm the example that it asked for, and evaluated the qualifications provided by the algorithm. We terminated when the number of correct full sensor qualifications reached 70%. Figures 7 show the results of the four generators on the two buildings. The *Random* generator took the least number of examples to achieve full qualification of 70% of the sensor names, achieving it much quicker than the others. The reason for this is due to the long tail of the label distribution as was shown in Figure ?? . The top 20 most occurring labels, by themselves, can fully qualify about 80% of the sensor names. A random generator has a high probability of finding one of these 80% of the points, thus acquainting itself more quickly of the most frequently occurring labels. Neither of the other three classifiers is able to achieve that. They get stuck trying to either get every bit of information from sensor with obscure labels (*MinRemaining*), or trying to cover more labels by first qualifying sensors which have been least qualified (*MaxRemaining*), or by choosing from the set of ill-formed sensor names, which would indicate errors to the learning algorithm (*Self-Correcting*).

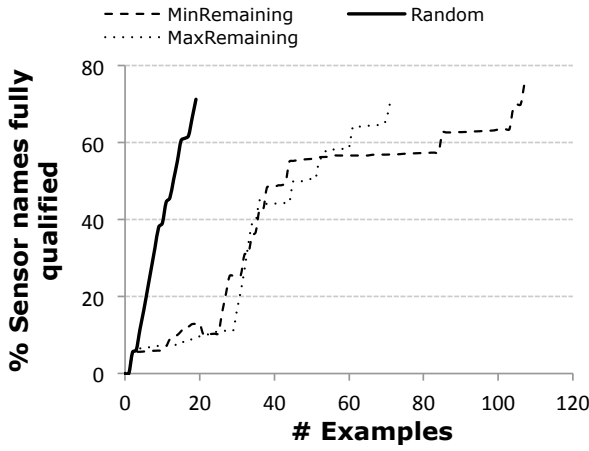
4.3 Applying to other buildings

One of the major goals of learning from examples provided by an expert is that the learnt regular expressions may be applicable to other buildings which have similar naming conventions. In this section, we evaluate the efficacy of the regular expressions learnt on Building 1 of our testbed with the remaining 55 buildings on campus, each of which was commissioned by the same vendor. As mentioned before, the remaining buildings comprise about 15,000 sensor points.

Figure 7a shows the applicability of tags learnt from Building 1 to the remaining buildings. Tags which were most frequent in Building 1 such as **zone**, **zoneRef** and **ahuRef**, were also widely applicable throughout other buildings. This is expected because all these buildings are commercial offices, with a large number of zone, and a set of points associated with each zone. There are also a few tags which are infrequent in a particular building,



(a) Building 1



(b) Building 2

Figure 6: The number of examples required to fully qualify 70% of sensor names in two buildings

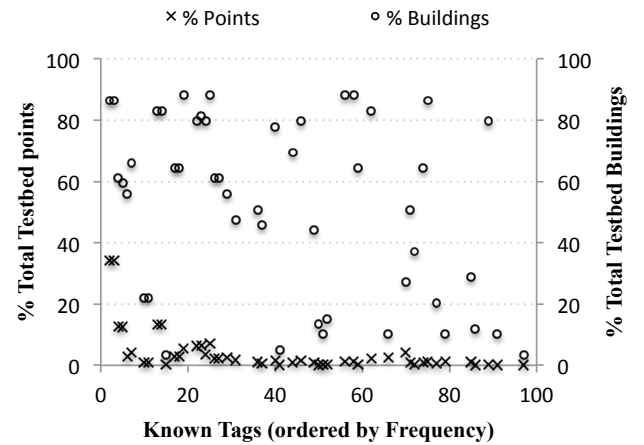
but applicable across all buildings, e.g the **return water temp** tag or the **outside air temp** tag. Thus, fully qualifying even one building has the potential to yield regular expressions that can then be propagated across buildings.

It was impractical to ground truth all the buildings, and its 15,000 sensor points for our experiment. Hence, we hand wrote regular expressions to the best of our efforts to qualify sensor names in those buildings. It is this manual qualification data that we treat as ground truth in the our experiment. We use the regular expressions obtained after 90% of the sensor points in Building 1 was fully qualified. While running these expressions on a different building, we just replaced the token corresponding to the tag **site** to the building-specific **site** token.

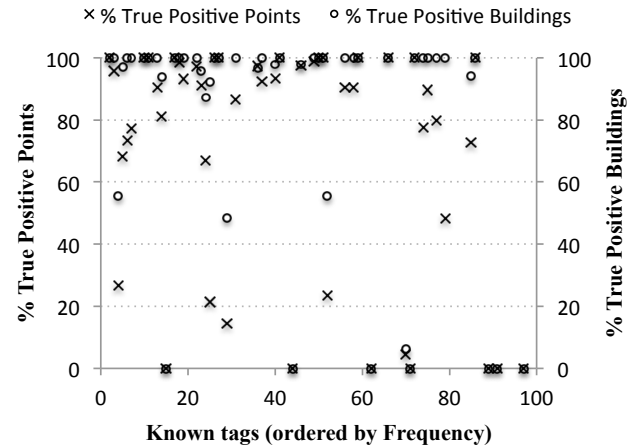
Figure 7b shows the true-positive rate based for each tag based on our ground truth data. The true posi-

tive rate indicates that the required tag was correctly applied where it was applicable, and it was able to qualify the correct substring from the corresponding sensor name. As we can see, the regular expressions were able to identify almost all the buildings that each tag learnt from Building 1 was applicable to. Some of the tags could not be applicable to other buildings, because they use different constant values for the same tag name. For instance, the **exhaust fan** tag is specified in certain buildings as EF, whereas in Building 1 it was specified as E. Also, certain variable tags vary in format across buildings, e.g the **zoneRef** tag, which leads to a lower true positive rate for such tags.

The implications of



(a) Building 1



(b) Building 2

Figure 7: The number of examples required to fully qualify 70% of sensor names in two buildings

5 Case Study

In this section, we demonstrate that with the meta-data automatically expanded and normalized using the

techniques in section ??, we are able to implement applications that are generalizable from one building to another building without modification. As a proof of concept, we implement two applications on the two building as test bed: a) identify uncomfortable rooms and b) detect rogue rooms. We also evaluate the metadata expansion technique in terms of the accuracy for both applications compared against the ground truth.

5.1 Experimental Setup

We implement two applications and perform the analysis on two buildings from a campus, and each building is installed with a different management system. Building A was built in mid-1990s using the system from Barrington []. Building B was recently built in 2004 installed with the Siemens BACnet system []. We used the temperature data as well as setpoint information of the rooms in each building. The temperature measurements are reported every 15 seconds and the data used for analysis is from one week in June 2009 and January 2012 respectively.

5.2 Uncomfortable Rooms

It's not unusual to have rooms in a building stay extremely cold or hot thus making the occupants feel uncomfortable and incurring energy waste. The discomfort is usually caused by improper setpoint configuration or dysfunction of the HVAC systems, and being able to identify these uncomfortable zones or rooms in the building is vital to improve occupant comfort as well as achieve potential energy savings. With the metadata normalized using our techniques, we are able to search for the desired streams, e.g., the temperature and setpoint of a room, and analyse the thermal performance of different buildings despite of the different naming schema used to label sensors and meters.

<i>Bldg1</i>		<i>Bldg2</i>	
room#	%	room#	%
326	1	S3-6	1
340	1	S2-1	1
352	1	S1-2	0.72
364	1	S6-09	0.54
376A	1	S7-16	0.49
380	1	S4-15	0.45
384	1	S6-10	0.44
405A	1	S7-10	0.42
405B	1	S5-15	0.35
410A	1	S5-13	0.29

Table 1: Rooms in each building are ranked by how much time they are uncomfortable throughout the one week period, and the first ten rooms in the ranking of each building are listed.

To identify the discomfort in a building, for each room we are particularly interested in a) how much does the temperature deviate from the comfort range? b) how much does the temperature deviate from the setpoint? c) how much does the setpoint deviate from

the comfort range? To answer these questions, we first search through the points in each building for distinct temperature stream of each room and the corresponding setpoint. Then we compare the temperature with the setpoint as well as the suggested comfort range by ASHRAE [] to compute the temperature deviations from the aforementioned three different perspectives in one-week period. We accumulate the results from all the rooms per building and generate the distribution as illustrated in Figure 8.

Figure 8 presents the ground truth of temperature deviation distribution in Figure 8, where we manually find the temperature and setpoint streams for all rooms and follow the above steps to generate the distributions. Each graph shows how much the temperature of a building deviates from the comfort range, the setpoint and how much the setpoint by itself deviates from the comfort range. On average, each building is uncomfortable to some degree and to better understand which specific rooms are uncomfortable, we rank the rooms in each building by how much they deviate from the comfort zone. The ranking results are shown in Table 1. We also break the identified uncomfortable rooms down to their corresponding air handler unit, as shown in Figure 9a. The ground truth analysis covers all the rooms in each building, so all the potential uncomfortable rooms are identified here. However, the analysis using the name points expanded with our techniques would miss some of the uncomfortable rooms because the expansion contains certain error rates. We will show the error later.

5.3 Rogue Rooms

Heating and cooling contribute to the largest portion of energy consumption of a building, and often, HVAC system operates abnormally either because the system fails itself or the schedule of the building is problematic. And there are often some zones and rooms in a building that are constantly cold or hot than the neighbors thus incurring energy waste. We demonstrated the temperature deviation distribution above, and we are particularly interested in the periods when a room deviates from the setpoint more than 3 Celsius degree, which indicates that the room is highly likely to be under either heating or cooling. Therefore, for each building, we zoom in to the interested portion and find the rooms falling into the interested area in most of the time, i.e., which room's temperature almost always deviates from the setpoint more than 3 Celsius degree, and the ground truth results are summarized in Table 2. Again, we group the rooms according to their air handler unit ID and get the results in Figure 9b.

5.4 How Many Rooms are Missed?

The metadata expansion contains certain errors therefore when we do a search over the expanded metadata we might not get all the desired streams for analysis. Figure 10 shows the error rates of the search results over expanded metadata for the two test bed buildings. On the left, 50% of the metadata of building 1 was correctly fully expanded, and doing the two searches for

<i>Bldg1</i>		<i>Bldg2</i>	
room#	%	room#	%
330B	1	S3-6	1
340	1	S2-1	1
420A	1	S1-2	0.93
420	1	S7-16	0.67
698	1	S3-13	0.62
442	0.996	S4-15	0.62
398	0.981	S5-5	0.57
336	0.96	S4-1	0.54
183	0.92	S5-16	0.48
498	0.91	S5-7	0.47

Table 2: Ground truth for how much time each room’s temperature deviates from the setpoint more than 3 Celsius degree. For each building, the first column is room number and the second column is the percentage of the one-week time that the room deviates that much.

“room temperature” and “room temperature setpoint” will get us all the desired streams. Therefore, performing the uncomfortable and rogue rooms analysis will not miss any rooms. Meanwhile, for building 2 on the right, when 50% of the metadata is correctly fully expanded, we missed X and Y streams for the same two searches respectively. Since not all of the X rooms are necessarily uncomfortable or rogue, we missed P and Q rooms for the two applications as a result.

With the expanded and normalized metadata of a building, we can run useful analysis and identify potential problems in the it. Even though the expansion has errors in it, we are still able to find most of the problematic rooms.

	Bldg 1	Bldg 2
Uncmft	0	4
Rogue	0	5

Table 3: The number of missed rooms for the two applications for the two test bed buildings.

6 Discussion

7 Related Work

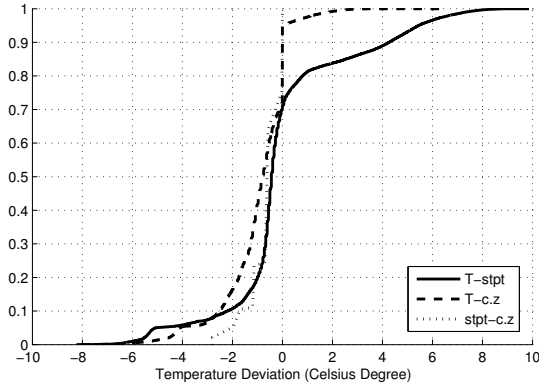
[3] [1] [4] [2]

8 Conclusion and Future Work

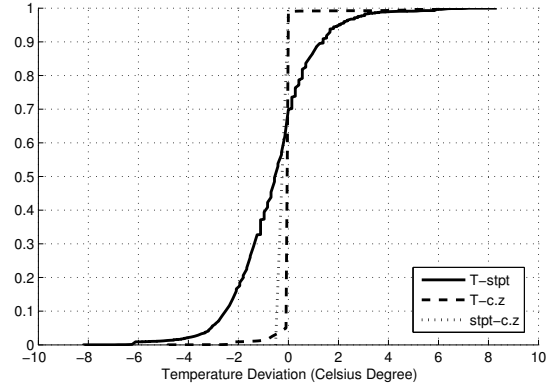
9 References

- [1] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’11, pages 317–330, New York, NY, USA, 2011. ACM.
- [2] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. In *In Communications of the ACM*, 2012.
- [3] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, pages 317–328, New York, NY, USA, 2011. ACM.

- [4] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proc. VLDB Endow.*, 5(8):740–751, Apr. 2012.

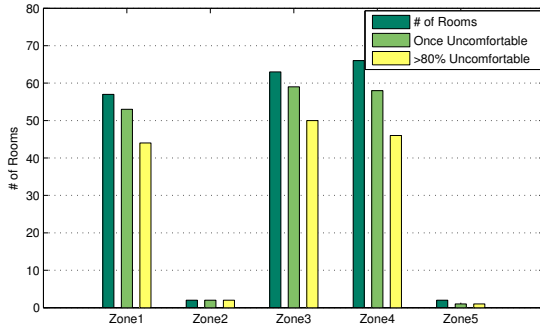


(a) Building 1

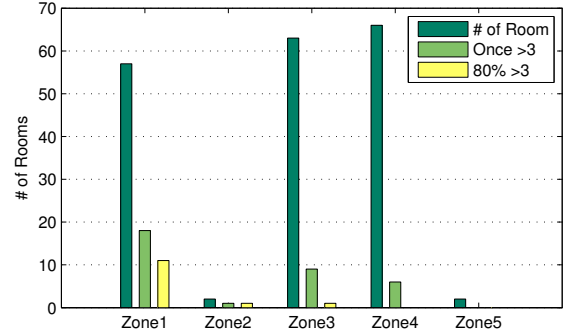


(b) Building 2

Figure 8: For each building, the distribution describes the temperature deviation between: a) room temperature and the corresponding setpoint (solid), b) room temperature and the comfort range suggested by ASHRAE (dashed), c) room temperature setpoint and the ASHARE comfort suggestion (dotted).

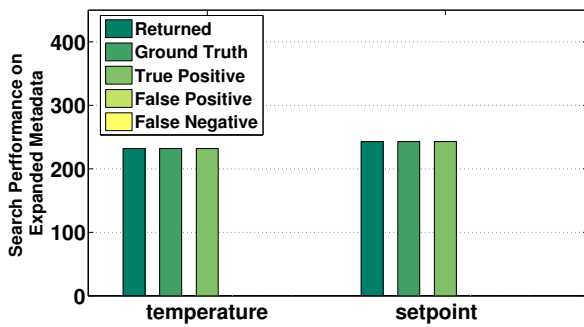


(a) Uncomfortable Rooms

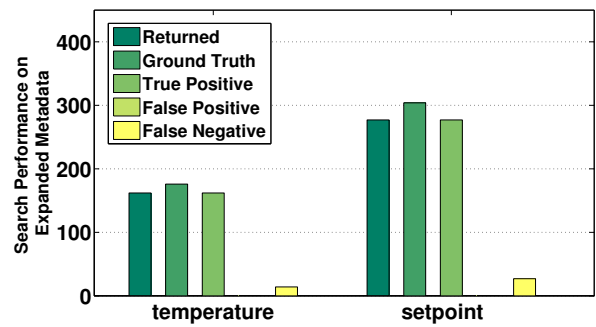


(b) Rogue Rooms

Figure 9: Breakdown of the uncomfortable and rogue rooms in building 1 by air handler unit zone. Uncomfortable rooms are those whose temperature at least once exceeds the comfort range suggested by ASHRAE. Rogue rooms are those whose temperature deviates from the setpoint more than 3 Celsius degree. For each zone, we show the number of rooms in it, the number of zones at least once meets the criteria, and the number of rooms that meet the criteria in more than 80% of the one-week period.



(a) Building 1



(b) Building 2

Figure 10: The error rates of searches over the expanded metadata using our techniques. Two searches are performed particularly: “room temp” and “room temp setpoint”.