What would a data scientist ask?  
Automatically formulating and solving prediction problems

Benjamin Schreck  
CSAIL, MIT  
Cambridge, MA - 02139  
bschreck@mit.edu

Kalyan Veeramachaneni  
LIDS, MIT  
Cambridge, MA- 02139  
kalyanv@mit.edu

Abstract—In this paper, we designed a formal language, called Trane, for describing prediction problems over relational datasets, as well as implemented a system that allows data scientists to specify problems in that language. We show that this language is able to describe prediction problems across many different domains, including those on KAGGLE—a data science competition website. We express 29 different KAGGLE problems in this language.

We designed an interpreter, which translates input from the user, specified in this language, into a series of transformation and aggregation operations to apply to a dataset in order to generate labels that can be used to train a supervised machine learning classifier. Using a smaller subset of this language, we developed a system to automatically enumerate, interpret and solve prediction problems. We tested this system on the Walmart Store Sales Forecasting dataset found on KAGGLE [1] by enumerating 1077 prediction problems and then building models that attempted to solve them, for which we produced 235 AUC scores. Considering that only one out of those 1077 problems was the focus of a 2.5 month-long competition on KAGGLE, we expect this system to deliver a thousandfold increase in data scientists’ productivity.

I. INTRODUCTION

A typical data science endeavor begins when domain experts decide to solve a particular problem by marshaling the data repositories available to them. Once this problem is defined, data scientists enter the picture, and must go through a sequence of steps in order to approach a solution. First, they have to work with the data collection and maintenance team to gain an understanding of how the data is organized and what it contains. Next, they must turn back to the domain expert, proposing potential predictive formulations, and asking whether particular types of predictive models would be helpful given some acceptable level of accuracy.

This process is typically noisy, since both domain experts and data scientists are attempting to learn about the data and formulate a prediction problem simultaneously. As a result, the process is iterative, carried out several times even before the first steps toward building a predictive model are taken. Upon its conclusion, the data scientists decide on a formulation for the prediction problem. However, since this process is executed prior to the actual model-building, no one involved can possibly deduce beforehand whether a good predictive model can be generated. What if a slightly different formulation, or the selection of a closely related field, would result in more accurate models? Moreover, when a new domain problem is encountered, the entire process must be repeated.

However, most of these prediction problem formulations that data scientists come up with share many common elements. For example, most formulations include picking a time-varying column, selecting a window, and applying a limited set of operations in order to create the outcome that one is interested in predicting. This begs the question: can we automatically enumerate all or many of the prediction problems we would want to consider for a given data set, and can we do so without extensive prior knowledge or data manipulation?

To do this, we need a common language to describe these prediction problems that is descriptive enough to be general-purpose, but limited enough to be enumerable. A language like this would allow us to automatically formulate hundreds, if not thousands, of prediction problems for a given data set. Since we already have both automated feature engineering [2] and machine learning tools [3] at our disposal, we could also automatically solve these predictive problems, and present the results to a domain expert or data scientist. This level of automation allows for the collapse of the iterative process described earlier into a single first pass, allowing data scientists and domain experts to collaboratively select problems to consider, while having immediate access to an estimate of each problem’s predictive accuracy (and therefore feasibility). It simultaneously frees up data scientists’ time for higher-level and more creative tasks, and increases the exploration of the prediction problem space a thousandfold. It grants domain experts and data scientists the ability to know what types of prediction problems are possible, and whether or not they can achieve reasonable predictive accuracy, so he or she can decide which are worth considering further. The system may even generate interesting problems that human actors would not have thought of.

In this paper, we demonstrate that this is possible by introducing the first version of a language that, given any relational data set, can iterate over thousands of data science problems, running each problem end to end on real data to come up with real predictive accuracy metrics. Our contributions can be summarized as follows:

1) **Trane**: A language general enough to describe arbitrary prediction problems over relational data sets. Trane is independent of any specific domain, and can be applied to any relational time-varying data set.

---

1We do not claim that this language is complete, or that it is the only way to define these types of problems. Rather, we present it as a language that works well for a wide variety of problems.
2) **LittleTrane**: A subspace of this language such that an algorithm can enumerate all problems in the subspace.

3) **An interpreter**: When given a problem defined in this language, this interpreter is capable of transforming the underlying data into learning segments and labels for use in a downstream feature engineering and machine learning system such as [2].

4) **An end to end demonstration of LittleTrane on a real-world dataset**: We generated more than 1000 prediction problems for a Walmart dataset found on Kaggle.com [1], and successfully ran 407 of these problems to generate cross-validated AUC scores. We found several of these generated (classification) problems to closely resemble the proposed regression problem on Kaggle (by humans).

The fact that Trane makes enumeration possible over a limited, practical subset of problems is extremely powerful, because along with its interpreter, it enables automatic training of machine learning models, and thus automatic discovery. We expect that this will have a transformative affect on how discoveries are made from data and how conversations are held around data. For example, users can plug in their data and expect the software to synthesize predictive problems, solve them and report back with minimal human involvement.

We organize our contributions as follows. We present related work in Section [II]. Section [III] outlines how this language is defined. Sections [IV] and [V] present two parts of the language. Section [VI] defines the interpreter we developed for this language. Section [VII] presents the experimental data we used to define the efficacy of this language, as well as the results.

II. RELATED WORK

Recently, experts in various domains have expressed great interest in the development and application of machine learning algorithms. This has led to a focus on the development of general-purpose abstractions, automation, and methods for addressing the complexities of data. In [4], the authors emphasize that *phrasing a problem as a machine learning task* is a necessary and important step. Works like [3] and [6] demonstrate the possible impacts when machine learning and domain experts combine to solve major challenges.

To smooth out these interactions, much focus has been placed on automating various steps, including data cleaning and preparation [7], feature or variable engineering [8], [9], [2], or providing useful and intuitive interfaces to complex algorithms [10]. However, to the best of our knowledge, no system exists for generating predictive modeling questions to ask of data. With the aforementioned tools available, we consider it timely to shift our focus towards a system that can automatically generate the questions themselves from this type of data.

III. **Trane: A Language to Formally Express Prediction Problems**

Here we give an overview of the language that allows us to formally express numerous prediction problems, given a time-varying relational data set. With this language, we can then write algorithms to enumerate a large number of prediction problems. We do not claim that all possible prediction problems can be defined this way; however, almost any problem we have encountered can map to this space, and we believe the space is large enough to be practically useful. Later in this paper, we show that every time-dependent problem on the data science competition website, KAGGLE, can be mapped to this expressive language.

A typical data set contains various tables, each associated with an *entity*. Each *entity*’s table contains one or more columns that uniquely define it (ignoring the possibility of full duplicate rows). Since we care about the time-varying case, we assume that at least one of these entities contains a time column as a part of the set of columns We will refer to this time column as the *time axis*, and the rest of the columns in the unique set the *index*. For example, let us consider a data set that records the locations of several taxis at different time points during a typical day. In this example, each data point has < taxi_id, timestamp, latitude, longitude >, where taxi_id is the *index* that identifies each individual taxi, and timestamp is the *time axis*. The *index* contains many *instances* (in this case–taxis), each of which refers to some specific, real-world thing.

Each prediction problem needs a set of data points that can be used for learning the predictive model, as well as the value/label (or *class*, in classification problems) it needs to predict. To fully define a predictive problem, the language must establish two things:

1) **Learning and Label-Computing Segments for each instance**: In the language, we must define two segments of data for each *instance*. The first segment, referred to as the *learning segment* is used for learning the model, and the second, referred to as the *label-computing segment* is used for generating a label. There may be overlap between these two segments. To divide the data pertaining to these segments, we define multiple time points that bound these segments, as well as a logic to choose values for them from the data. Section [IV] details the process of setting these time points.

2) **The sequence of operations that computes the label(s) for each instance**. The labels or values we are interested in predicting are derived from the data set itself, and may make use of all recorded observations of the *instance*. For example, one may want to predict the total distance traveled by taxis. In this case, the labels would be calculated by applying the following operation: subtract the final location from the initial location. We call these operations *label-computing operations*, and the columns they are applied to as *label-generating columns*. Section [V] describes how operations are sequenced and executed.

IV. **Trane: Segmenting the Data**

When defining a prediction problem, the first step is to define two data segments: first, the segment that can be used to learn the model, and second, the segment used to generate the label we are trying to predict. There are many complexities inherent in defining these segments, and achieving true flexibility in optimally defining for a given problem is a challenging task. For instance, one might want to split the data for an *instance* into several windows, where each window is triggered by some event (e.g. a customer making a purchase). In [11] authors painstakingly describe the complexities of this
process and present solutions in detail and call it prediction engineering.

For our purposes, we do not need to produce the best or most amenable segmentation scheme for each prediction problem. Rather, we need one that is just general enough to work in most cases. Therefore, we split each instance’s data into just two segments: one for learning and one for generating the label. The learning segment always starts at the instance’s first recorded value, and ends with the cutoff point, defined below. The label-computing segment starts either at the first recorded value or at the cutoff point, and ends at the label-cutoff point. Before we present the logic for defining each of these two segments, we define these bounds in more detail.

Fig. 1: Visualization of ways to define cutoff (and end) points. One can either define a single cutoff or end point for all instances, or define separate points for each instance. Per-instance cutoffs are calculated in terms of each per-instance start and end values. One can define a single cutoff in terms of various start, end, and duration constants. We visualize these constants here.

**Cutoff point:** The last time point at which we allow data to be incorporated in the predictive model. This is the later bound of the learning segment, and sometimes the earlier bound of the label-computing segment.

**Label-cutoff point:** The last time point we allow in our label computation. This is the later bound of the label-computing segment.

**Black-hole point:** The time point for each instance at which the label is deterministically known. This point is used in certain cases to shift the cutoff point earlier so that we do not have any label leakage in our learning segment (i.e. so that the machine learning algorithm cannot cheat). Section [IV-B](#) explains this point in more detail.

### A. Setting the cut-off time points based on data

To set the cutoff and label-cutoff points, for each instance, we first note that all instances may not start and end at the same time point, thus, each has its own start and end point as illustrated in Fig. 1. We can set the bounds for the segments via two methods. Below we describe both.

**Setting them globally:** In this setting, every instance is given the same cutoff and label-cutoff time points. If, for any instance there is no data available in the resulting segment, the instance is ignored. To be able to set time points, globally, we first define a set of global time points based on the data. They are: global_start_first, the earliest recorded start time out of all instances, global_start_median, the median of all the recorded start times across instances. Similarly, we define global_end_first, global_end_last and global_end_median. Given these we can define segment bounds for all instances as:

- An explicit time point: a point in time common across all instances, for example August 1, 2015.
- A relative time point: This time point is computed by specifying an offset from a globally defined time point. This offset can either be a time delta from a global_start_x or global_end_x points, or a percentage of a global_duration_x.

**Setting them instance specific:** In this setting, each instance gets its own bounds set based on its time line. This allows the user to maximize the use of the available data. To set instance specific time points, we define instance_start, instance_end. Instance specific time points for cutoff and label-cutoff are computed by specifying an offset from an instance_start or instance_end. The offset could again be percentage of duration or a time delta.

**Programmatically setting different time points in Trane**

Let us now make explicit how various settings for the cut-off and end points are set in Trane via setting some flags. Below, we show these options.

```python
cutoff_settings = {
    pick_latest: Boolean
    specific_date: Datetime
    offset: T imedelta
    offset_from: String
    percentage: Float[0->1]
    percentage_of: String
}
endpoint_settings = {
    pick_latest: Boolean
    specific_date: Datetime
    offset: T imedelta
    offset_from: String
    percentage: Float[0->1]
    percentage_of: String
}
misc_settings = {
    lead: T imedelta
    step_function: Boolean
}
```

### Cutoff from Settings

1) If pick_latest is False, then we define the cutoff according to the options below; otherwise, we pick the maximum cutoff before the black hole point.
2) If specific_date is specified, then we simply set cutoff at that date.
3) Else if offset is specified, then we check the offset_from flag to see where to calculate the offset. If offset_from is one of the start times (global or instance), we add offset to create the cutoff. If it is an...
4) If percentage is specified, then we calculate the percentage of the duration specified by percentage_of and add to the start.

**Label-cutoff point from Settings**

1) If pick_latest is False, then we define the label-cutoff according to the options below; otherwise, we pick the last recorded value for each instance.

2) If specific_date is specified, then we simply pick the label-cutoff point at that date.

3) If offset is specified, then we check the offset_from flag to see where to calculate the offset. If offset_from is one of the start times (global or instance), we add to it and the create the label-cutoff point. If it is one of the end times, we subtract the offset to create the label-cutoff point. This flag can be also set as the cutoff point, in which case the label-cutoff point is determined by adding the offset to the cutoff point.

4) Else if percentage is specified, then we calculate the percentage of the duration specified by percentage_of. percentage_of can also be set as one of the end time points. When that is the case, the label-cutoff point is calculated through the following operation: (cutoff point + end constant) × percentage.

**B. Other considerations**

**Lead time:** We provide the option to specify a lead time, which is a time delta specifying how much data to take out from the end of the learning segment before supplying to a machine learning algorithm. Specifying a lead makes the problem harder, since it widens the time gap between the learning segment and the label-computing segment.

**Using data before cutoff in the label-computing segment:**

In the default case the label-computing segment is made up of the data between the cutoff point and the label-cutoff point. However, in some cases, it is necessary to use data prior to the cutoff point. This applies to some problems where the final label is dependent on the entire time series. Consider, for instance, the problem of predicting whether a taxicab ride was longer than a certain threshold. This would require computing how far the taxi traveled in meters. In order to compute this metric, an algorithm needs access to the starting location. We therefore define a use_known flag; when True, the start point for label-computing segment is the beginning of the instance data. One can imagine situations where the label-computing window should include data before the cutoff but not all of the data from the very beginning; however, we neglect this case for simplicity.

**Adjusting the cutoff based on the Black-Hole Point:** For some classification problems, we also need to find the time point for each instance at which, were any subsequent data included in learning the predictive model, it would be easy to directly calculate the label. Call this the Black-Hole Point, or BH-point.

Let us consider the problem of predicting whether a user of a website will become a repeat customer by making additional future purchases. Assuming the data consists of user website interactions (clicks), the BH-point is precisely the last interaction before the one in which the user pressed the “Buy” button for the second time. In this case, the use_known flag would be True, and the label-computing segment for each instance would include all of its data. In this case, the label-generating function is a step-function, meaning that if we plot the binary True/False label against the end point, the graph remains True for all time once it has switched from False. If the label-generating function is a step-function, then it never switches from True to False as time progresses. Therefore, if any data where the label-generating function evaluates to True is given to a learning system, that system can immediately determine the label. We consider this cheating, since we assume the label-generating function is known beforehand, and so we define the BH-point such that this knowledge is not revealed. If the explicitly-defined or implicitly-computed cutoff point is after the BH-point, we shift it to before.

**V. TRANE: LABEL COMPUTING OPERATIONS**

Once a segment of data is identified for computing the label, the next step is to apply a sequence of operations to an instance’s data in order to generate the labels for each instance. We define the two main types of operations: row and column operations. The overall workflow is to group the dataset by instance, identify the label-computing segment, apply a filter to remove rows that do not pass some criteria, apply a sequence of operations to the filtered rows, and then possibly repeat. The final output is a single value (for classification or regression problems).

**Row Operations:** Row operations act independently over each individual row, replacing each with a new row, possibly with a different number of columns. A simple example is the greater than operation (referred to as GT).

\[
\text{Row Operation:}
\begin{array}{cccc}
1 & 0.4 & 5.3 & 4/9 \\
\end{array}
\rightarrow
\begin{array}{c}
2 \\
\text{True}
\end{array}
\]

Illustrating how these operations are applied with an example. Imagine a dataset with 5 rows and 4 columns: “name of fruit”, “color”, and “price per pound” and “quantity”. We would like to apply the GT (2) row operation to the “price per pound” column. To do this, we would select each row independently, isolate the “price per pound” column, and replace the value in it with either True or False depending on whether the data was greater than 2. We then repeat this for each row. For a visualization, see [1]. Note that as part of defining the row operation such as GT, we have to set the numeric parameter it compares its input with and each row operation must define which column it operates on.

Since row operations operate independently on each row, it is not required to group the data set by instance before applying the row operations.

As an added complexity, we can imagine row operations that operate over multiple columns in a row, such as arithmetic operations that add or subtract two columns to produce a single output, or date/time operations that separate a date column into separate day/month/year columns. To reduce the search space of the enumeration algorithm, we limit row operations to very simple single-column functions.
Multi-Column Transformation Operation

<table>
<thead>
<tr>
<th>Constant Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>global_start_first</td>
<td>First recorded start time out of all instances.</td>
</tr>
<tr>
<td>global_start_last</td>
<td>Last recorded start time out of all instances.</td>
</tr>
<tr>
<td>global_start_median</td>
<td>Median of instance start times.</td>
</tr>
<tr>
<td>global_end_first</td>
<td>Earliest recorded end time out of all instances.</td>
</tr>
<tr>
<td>global_end_last</td>
<td>Last recorded end time out of all instances.</td>
</tr>
<tr>
<td>global_end_median</td>
<td>Median of instance end times.</td>
</tr>
<tr>
<td>global_duration1</td>
<td>global_end_last - global_start_first</td>
</tr>
<tr>
<td>global_duration2</td>
<td>global_end_last - global_start_last</td>
</tr>
<tr>
<td>global_duration3</td>
<td>global_end_first - global_start_first</td>
</tr>
<tr>
<td>global_duration4</td>
<td>global_end_first - global_start_last</td>
</tr>
<tr>
<td>global_duration5</td>
<td>global_end_median - global_start_median</td>
</tr>
<tr>
<td>instance_start</td>
<td>Start time of the instance, separately defined for each instance.</td>
</tr>
<tr>
<td>instance_end</td>
<td>End time of the instance, separately defined for each instance.</td>
</tr>
<tr>
<td>instance_duration</td>
<td>instance_end - instance_start</td>
</tr>
</tbody>
</table>

Fig. 2: Listing of time points available for defining cutoff and end points. Time points prefixed with global imply that they are to be used for a single defined cutoff or end point over all instances. Time points prefixed with per_instance imply that they are defined per instance.

Column Operations: Column operations differ from row operations in that they can aggregate or use information from multiple rows within the column. Hence, these operations are defined to first group the table by instance, and apply themselves to each group separately. Column operations are divided into aggregation and transformation operations. Aggregation operations return only a single value for each instance, while transformation operations can return multiple values.

A simple example of transformation operation is the DIFF operation, which calculates the difference between consecutive rows, visualized in [2]. If operating on an instance in the dataset containing \( n \) rows, the DIFF operation outputs a new set of \( n - 1 \) rows for that instance.

Column (Transformation) Operation

\[
\begin{array}{c|c|c}
\text{Column} & 4 & 5 \\
\text{DIFF} & 1 & 1 \\
\end{array}
\]  

\( \text{DIFF} \) is an example of an aggregation operation; it counts the total number of rows per instance. This operation is visualized in [3].

Column (Aggregation) Operation

\[
\begin{array}{c|c|c}
\text{Column} & 4 & 5 \\
\text{COUNT} & 1 & 1 \\
\end{array}
\]  

The distinction between aggregation operations and transformation operations is necessary because it allows us to chain together transformation operations. The output of one transformation operation produces a new table which can be fed into the input of another one. The same cannot be said of aggregation operations, since they produce a single value for each instance as output. Like row operations, more complex column operations can operate on multiple input columns and produce multiple output columns. One example operation we defined to allow us to build recommender systems is the IDENTITY_ORDER_BY operation. This operation takes two columns as input, reorders the first using the values in the second, and returns the ordered values in the first column, visualized in [4]. This is the only multi-column operation we consider here, but clearly there is no limit to the operations one can define.

Multi-Column Transformation Operation

\[
\begin{array}{c|c|c}
\text{Column} & \text{apple} & \text{orange} \\
\text{DIFF} & 4 & 5 \\
\text{COUNT} & 7 & 8 \\
\end{array} \rightarrow \begin{array}{c|c|c}
\text{IDENTITY_ORDER_BY} & \text{apple} & \text{orange} \\
\text{DIFF} & 2 & 3 \\
\text{COUNT} & 4 & 4 \\
\end{array}
\]  

Filters: In addition, we define filter operations, which take in some subset of columns in each row, perform an operation on them, and output True or False, which signifies whether to include that row to be further processed by row and column operations. Filter operations can be arbitrarily complex in theory. Here we limit them to essentially the form: \([\text{OP}](\text{COL1}) == \text{VALUE}\), where each element in \text{COL} is applied an operation and compared to \text{VALUE}. We also allow the right side to include an aggregate-type operation over another column to produce a single value based on the data. The syntax in this case looks like \([\text{OP}](\text{COL1}) == [\text{OP}](\text{COL2})\), where the right side is an operation like \text{MAX}, \text{MIN}, or \text{MEAN}. Another special case is the ALL filter, which returns True for every row. We also define JOIN type filters, where the input is two separate filter operations and a Boolean operation to compare the two, like \((\text{HOUR}(\text{col1}) == 3:00) \&\& (\text{IDENTITY}(\text{col2}) > 5))\).

Possible Operations: Many more operations are possible, but listed in tables [I][II][III][IV] are all those we needed to define the Kaggle data sets in this language, as well as a few additional ones needed to define some internal data sets with which we were familiar. The tables also explain the output types and allowed input types of each operation.

Language Syntax: The full syntax of the prediction problem language requires at least one filter operation, one row operation, and a column operation applied to the label-computing window of each instance in the dataset. However, all of these could simply be IDENTITY, or, for the filter operation, ALL. We define a group of one filter operation, one row Operation, and one column operation as a sequence. A second sequence can be chained together if the column operation in the first one was a...
transformation operation. To successfully construct a prediction problem, we have to specify an aggregation operation for the final column operation, so a single value could be generated for the instance. We also allow an optional final row operation at the end of each prediction problem, generally a thresholding operation like GT (greater than) or EQ (equal to). This is useful for converting regression problems into classification problems with a threshold.

Finally, we allow Boolean combinations of several prediction problems, provided the output of the prediction problems is binary. For instance, one might want the label to be True if all the values of a column are above a threshold and all the values of a different column are below a threshold.

Examples of Prediction Problems in Trane: It helps to illustrate this syntax with some examples, along with their interpretations. We do not include cutoff flags for these examples, since those flags are set more or less independently from the label computing operations.

1) Predict whether the last value of the differences between consecutive "Weekly_Sales" column values is less than some user-defined threshold.
   Label-generating column "Weekly_Sales"
   Time axis "Date"
   Index "Store"
   Operations
   DIFF → IDENTITY → LT → LAST → NOP

2) Predict whether the number of "Transaction_Amount" values that are greater than some threshold is greater than some other threshold.
   Label-generating column "Transaction_Amount"
   Time axis "Timestamp"
   Index "TransactionId"
   Operations
   GT → SUM → GT → NOP → NOP

3) Predict whether any values are recorded after cutoff. (This amounts to customer churn, like this Kaggle competition [12])
   Label-generating column N/A
   Time axis "Timestamp"
   Index "Customer"
   Operations
   IDENTITY → EXISTS → NOP → NOP → NOP

VI. ENUMERATION AND PRACTICALITIES

To enumerate number of possible prediction problems, we can design a simple algorithm, where given a dataset, it:

- identifies the entity of interest that has time varying data,
- identifies the time varying column associated with this entity,
- for every time varying column (or a set of columns) form an arbitrary sequence of operations that can eventually aggregate and give a single value.

For the sake of space, we do not present an enumeration algorithm here. However, when we attempted to enumerate even for a simple dataset, we ended up with hundreds of thousands of predictive problem formulations. We next decided to use a different tact. We attempted to express multiple publicly available predictive problems using this language and identified some common patterns.

A Case for Simplicity Out of 54 prediction problems defined on Kaggle and converted into Trane, 51 could be defined using only a single column operation, and the remaining three could be defined using only two column operations. Because these problems represent a wide swath of possible business and research applications, we value simplicity as germane to the meaningfulness or usefulness of a prediction problem.

With that said, this is a limited sample, and many of these data sets were at least partially formatted or simplified to make them easier to use in a competition setting. We can easily come up with data sets where it is meaningful to define a prediction problem with three layers of column operations. Hence we defined LittleTrane.

LittleTrane: LittleTrane includes a simpler subset, but still potentially defines thousands of prediction problems per dataset (as opposed to effectively no limit in BigTrane).

- We cap the nesting of transformation operations to atmost two sequences, so that there is either a single aggregation or transformation operation, or there is one transformation operation followed by one aggregation operation. This means we have a fixed number of operations: there are only five, with three row operations in positions 0, 2, and 4 and two column operations in positions 1 and 3. We allow the last two operations to be "NOP" (no-operation) for simplicity of compilation, instead of requiring them to be IDENTITY.
- We restrict LittleTrane to operate over single label-generating columns, which leaves out the IDENTITY_ORDER_BY and NORM_DIFF transformation operations.
- We disallow filters.
- We disallow combinations of multiple prediction problems.
- We do not define arguments/thresholds for the operations.
- We do not define cutoff points.
- We restrict the output to binary classification, i.e. True/False, single row output for each instance.

Limitations Obviously, with just 5 operations, LittleTrane is not nearly as expressive as BigTrane, and there are many useful problems we cannot define in LittleTrane. A real-world example is the Donors Choose Competition on Kaggle [13], which requires Boolean joining of multiple binary subproblems. Furthermore, problems in LittleTrane are not actually executable, since they do not define arguments and thresholds or flags for cutoffs. We cannot actually run these problems without manually inputting these additional settings.

VII. INTERACTING WITH TRANE

We next built a Trane interpreter, which takes as input:

- a prediction problem specified in Trane,
- a relational dataset in the form of Comma Separated Values (CSV) files, and
- a dictionary of metadata associated with the dataset in the form of either a JSON or Python file.
## Table I: Listing of available row operations, explanations of them, allowed input types, and output types. If an output type is labeled as same, then the input type is carried over to the output. If an input type is Any, then any input type is allowed.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Argument</th>
<th>Input Type(s)</th>
<th>Output Type(s)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENTITY</td>
<td>No</td>
<td>Any</td>
<td>Same</td>
<td>Does nothing- preserves the values of the specified columns exactly.</td>
</tr>
<tr>
<td>EQ</td>
<td>Yes</td>
<td>Numeric</td>
<td>Boolean</td>
<td>Equal to- transforms numeric inputs into Boolean True/False values by comparing them to the argument.</td>
</tr>
<tr>
<td>NEQ</td>
<td>Yes</td>
<td>Numeric</td>
<td>Boolean</td>
<td>Not equal to- transforms numeric inputs into Boolean True/False values by comparing them to the argument.</td>
</tr>
<tr>
<td>LT</td>
<td>Yes</td>
<td>Numeric/Categorical/Boolean</td>
<td>Boolean</td>
<td>Less than- transforms numeric inputs into Boolean True/False values by comparing them to the argument.</td>
</tr>
<tr>
<td>GT</td>
<td>Yes</td>
<td>Numeric/Categorical/Boolean</td>
<td>Boolean</td>
<td>Greater than- transforms numeric inputs into Boolean True/False values by comparing them to the argument.</td>
</tr>
<tr>
<td>POW</td>
<td>Yes</td>
<td>Numeric</td>
<td>Numeric</td>
<td>Power- transforms numeric inputs by taking them to the power of the argument.</td>
</tr>
</tbody>
</table>

## Table II: Listing of available transformation operations, explanations of them, allowed input types, and output types. Comma-separated types indicate that more than one input columns are necessary, with types specified by each of the comma-separated types.

<table>
<thead>
<tr>
<th>Transformation Operation</th>
<th>Arguments</th>
<th>Input Type(s)</th>
<th>Output Type(s)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENTITY</td>
<td>No</td>
<td>Any</td>
<td>Same</td>
<td>Does nothing- preserves the values of input rows exactly.</td>
</tr>
<tr>
<td>DIFF</td>
<td>No</td>
<td>Numeric</td>
<td>Numeric</td>
<td>1st difference- takes the difference between consecutive rows and returns one fewer than the number of input rows.</td>
</tr>
<tr>
<td>IDENTITY_ORDER_BY</td>
<td>Yes</td>
<td>Categorical, Numeric</td>
<td>Categorical, Numeric</td>
<td>Returns the identity of the input columns sorted by the values of the numeric column. Argument value is either 1 or -1, indicating whether to sort in ascending or descending order. Useful for recommender systems.</td>
</tr>
</tbody>
</table>

This interpreter applies the operations defined in the prediction problem to the dataset, and returns a new table with the labels for each instance, as well as the cutoff point for that instance. See figure 3 for a visual of how data passes through the Trane interpreter. The interpreter was implemented in Python, using the Pandas data science toolkit [14].

1) **Interpreter Input:** The input to the interpreter is a JSON object or Python dictionary specifying both the dataset configuration and the prediction problem configuration. The dataset configuration settings are defined in figure 5, while the prediction problem dictionary is shown in figure 4.

We limit the interpreter input to something in between BigTrane and LittleTrane. For instance, we allow various filter operations, as well as most of the settings for the cutoffs and arguments/thresholds (exceptions are detailed in figure 5), but we assume a single label-generating column and problems with maximum depth two (i.e. two column operations), and we disallow logical combinations of multiple problems. This is purely for simplicity on our first pass; the next version will include these features.

2) **Interpreter Output:** The interpreter outputs a Pandas dataframe using our column(s) as its index, where each row includes two columns: the label and the BH point. This output can then be fed into a feature engineering system along with the original training data in order to build a predictive model. To generate this output, the interpreter does the following: **Step 1:** **Cut off Data:** The first job of the interpreter is to cut off data at those time points specified by flags in the cutoff settings. The internal view of the data seen by the label calculating subsystem is devoid of any points in time prior to this cutoff.

**Step 2:** **Calculate Labels:** The interpreter then parses the input dictionary to produce the sequence of five row or column operations (some may be NOP's) and two filter operations. First, it applies these filter operations at the lowest depth to discard unwanted rows; second, it applies the first row operation to each row in the data, followed by the first column operation grouped by each instance. Recall that this first column operation can either be a transformation or an aggregation operation. If it is a transformation operation, and there is an additional row and column operation defined in the 3rd and 4th positions, then it applies the second filter operation (which might just be ALL).
TABLE III: Listing of available aggregation operations, explanations of them, allowed input types, and output types. If an output type is labeled as *same*, then the input type is carried over to the output. If an input type is *Any*, then any input type is allowed.

TABLE IV: Listing of available filter operations. Comparators are used to compare (scalar) operations on columns to either threshold values or (vector) operations on whole columns. The syntax is either ```[OP]([COL1]) [COMPARATOR] [THRESHOLD] or [OP]([COL1]) [COMPARATOR] [OP2]([COL2])``` where COL1 and COL2 could be the same column, but do not have to be. In the table, "Operations on LHS" denote the scalar operations applied to COL1 and "Operations on RHS" denote the vector operations applied to COL2. The Joins row just display the possible ways of combining multiple filter operations.

Fig. 3: Visual representation of data processing by the interpreter. Data is denormalized into a single table, grouped by instance, and cutoff at the time point specified in the settings, possibly a different point per instance. Transformation and aggregation operations are applied to each instance in order to produce labels, and BH-points are calculated for each label as well. The final output is a list of instance id, the BH-point and the label and this can be sent downstream to further data-mining and machine learning tools.

Step 2.5: Generate Arguments for LittleTrane Problems: To enable direct compilation of prediction problems in LittleTrane, we must define reasonable arguments or threshold values to use for the operations defined in tables I, II, and III. To do this, we included additional functions to automatically generate a list of reasonable arguments or threshold values. These should be different depending on the column type, but are somewhat difficult to intelligently guess, since each operation’s input range depends on the output from the previous operation. Thus we provide initial combinations of the following argument/threshold choices:

- powers of 10 from 1 up until the log of the number of rows in the dataset
- the top 5 most frequent values of the column
- mean of the column ± [0, 0.5, 1, 2]·(standard deviation) of the column
- mean of the differences of the column ± [0, 0.5, 1, 2]·(standard deviation) of the column

Step 3: Calculate the BH-Point: If the problem is a classification problem, the last step is to calculate the BH-points for each instance. In most cases, this is just the cutoff time point minus the lead. However, a more complex calculation is needed for EXISTS or ANY functions, or when the problem is explicitly flagged as a step function. We mentioned in section IV-B that problems whose label-generating function is a step function contain non-trivial BH-points. This is the case for the ANY function. For these cases, we return the last time point at
Fig. 4: This dictionary defines an input to the Trane interpreter. The sequence of operations defined is `POW(2), LAST, LT(100), NOP, NOP`, over the "Weekly_Sales" column. This translates to “For each Store-Dept instance, using the Date column as a time axis, predict whether the value of the Weekly_Sales column, to the power of 2, is less than 100”.

There is an initial filter that only looks at rows whose Date is between Tuesday and Friday. We include the original_table value for feeding into a downstream machine learning system that needs to know which table the label-generating column originated from (while the Trane interpreter uses the single denormalized table and does not need this value).

which the pre-aggregated time series was still False.

The EXISTS operation is also a step function in the True to False direction, but determining when an instance becomes False is a little complicated. This operation relates to problems such as customer churn, when we might want to know the point at which a user stopped interacting with a website. This point corresponds to when we know with certainty that the label for an instance is False; however, it always occurs after the last recorded data point! In order to find it, we consider the amount of time between an instance’s last recorded value and the overall latest end point (overall_end_2). If this amount of time is significantly greater than the maximum amount of time between any two of that instance’s recorded values, then we consider the label to be False (e.g. the user has churned). However, this complexity is only necessary to actually determine the label - we can still simply return the last recorded value as the BH-point and cutoff point.

Rules for Randomly Generating Problems: Here, we define two rules to randomly generate prediction problem definitions in LittleTrane.

- **Multiple Tables**: Currently, we only allow index, date, and label-generating columns from the same table. Future work will explore allowing problems defined over multiple tables.
- **Uniqueness of Index**: The selection of columns for the index must, along with the date column, uniquely identify the table. This is a simplification so that we do not have to ask the user for rules on how to aggregate duplicate rows.

Prediction Problem Execution: This interpreter offers a particularly powerful data science tool: namely, it becomes a way to test a large number of prediction problems end-to-end, without having to manually recalculate labels or spend time thinking about all the possible problems that could be defined. To explore this, we automatically generated features by connecting the interpreter to feature engineering software, and coupled it with the Python Sci-Kit Learn’s Random Forest implementation to generate weighted predictive AUC for problems defined using the Trane [9,3]. In the next section, we present our experimental setup and results.

VIII. DEMONSTRATION ON REAL DATA

Using the Walmart Store Sales Forecasting dataset from Kaggle [1], we generated 1077 binary classification prediction problems in LittleTrane and attempted to run them all on a scaled-down version of the dataset (to speed up the procedure).

As many of them were ill-defined, we ended up with results from 407 different prediction problems. Since these problems were defined in LittleTrane, they did not include argument settings, and so for each prediction problem we explored between 5 and 20 settings of each argument. In some cases, multiple operations in the problem required arguments, and so we enumerated all combinations of the possible arguments. We sampled from the combination of arguments either until five experiments were run successfully or we ran out of argument possibilities. This resulted in a total of 1923 experiments run successfully. The most common reason for a failed experiment was a lack of sufficient positive or negative examples.

For each experiment, we used feature engineering software to generate 10 features, and ran it with 5-fold cross-validation using Sci-Kit Learn’s Random Forest algorithm with 200 estimators. As a performance measure, we used the area under the ROC curve (AUC), weighted by the number of class labels.

Explanation of Data: This Walmart dataset describes 45 stores and 99 departments within those stores. The dataset contains weekly recordings from each department about how many goods they sold that week in dollars, measured over a period of 2 years. These recordings are contained in the Sales table. This table connects to a separate Features table with additional information recorded.

---

3 By “ill-defined,” we mean that the problems resulted in only a single class, so prediction is meaningless.
Flag | Options or Type | Explanation
---|---|---
explicit | True/False | Whether or not to explicitly cut off the data at some defined time
at_point | Datetime in range of time axis | Single point at which to cut off data (if defined, ignore all following options)
offset | Timedelta | Amount of time to cut off data from an offset point (if defined, ignore other options except explicit and offset_from)
offset_from | Start/End Constants 1-3 | If offset is defined, this is the point from which to find an offset
percentage | Float | Percentage of total time to cut off data (if defined, ignore other options except explicit and percentage_of)
percentage_of | Duration Constants 1-5 | If percentage_of is defined, this is the duration from which to take the percentage
keep_at_least | Timedelta or Integer | If explicit is False, defines how many values to reserve for training
lead | Timedelta | Defines lead time
step_function | True/False | Defines whether or not the label-generating function is a step function, meaning that once it becomes True, it stays True forever. This is necessary for determining the BH point.

Fig. 5: Listing of interpreter settings for defining the cutoff point. Note that we do not define an label-cutoff; for simplicity, we assume it is the last value per instance. We also do not allow explicitly defined cutoff points per-instance basis; we restrict the input to single, overall points. The Time delta type is available in many scientific computing environments, and defines a duration of time.

weekly per-store, and to a separate Stores table that contains meta-information about each store. We detail some information affecting the generation of prediction problems below.

- **Possible Indexes** Either the Store column, or a combination of Store and Department (Dept).
- **Possible Tables** If the index is just Stores, we only allow label-generating columns from the Features table; if it is the Store and Dept combination, we only allow label-generating columns from the Sales table. The Features table is uniquely defined by Store and Date, so if the index is just a Store, we do not have to define aggregation rules to combine non-unique rows in the Features table like we would have to in the Sales table.
- **Possible Columns** Any column in the Features or Sales table is fair game.
- **Size of Data** The original dataset is around 400,000 rows. We subsampled the data down to around 40,000 rows so that the experiments would run faster.

**Results** We plot the mean of AUC scores against the standard deviation across all experiments (for different settings of arguments) in figure 6. There does not seem to be much of a pattern apparent here; however, we encourage the reader to notice the broad range of AUC scores recorded. From this result, we can extrapolate that some experiments are much harder, or at least more well-defined, than others. Moreover, we can separate experiments into quantiles based on their AUC and standard deviations, and select a few examples for further investigation.

A. Sample Models

To pull out example models, we do the following.

1) Separate the AUC scores into four quantiles: 0-25%, 25-50%, 50-75%, and 75-100%, and the standard deviations into two: below and above the median.

2) Find the worst-performing argument setting, meaning the model with the lowest AUC score. Recall that each experiment was run with a number of argument settings.

3) Assign an AUC quantile and standard deviation quantile to the worst-performing model.

4) Sample two models from each quantile, yielding a total of 16 experiments, where each is a unique prediction problem.

We select the worst AUC, as opposed to the mean or the best, because arguments (and specifically thresholds) are intuitively more meaningful when they make a problem more difficult. These 16 experiments provide a broad picture of the variety of possible prediction problems, and which of those problems are actually meaningful.

Bin (0,0) AUC < 0.55, Low Standard Deviation

1) For each Store, Dept combination, predict whether all the future entries for the Store column are equal to some value.

2) For each Store, Dept combination, predict whether the last value recorded for Store (the Store ID) is equal to some value.
These problems are meaningless because the column it is predicting does not change; it is part of the For each Store we predict the future value of Store. In future versions of our Trane enumeration script, we must remove these. However, because the automated feature engineering library did not extract a feature for the value of the itself, our machine learning algorithm had no way of knowing the future value of Store, and so we achieve very low AUC and standard deviation.

**Bin (0,1) AUC < 0.55, High Standard Deviation**

1) For each Store, predict whether the difference between the consecutive values of fuel price, to the power of X, is less than Y at the label-cutoff point.
2) For each Store, predict whether the difference between the last and the first value of the difference between consecutive values of Markdown3 to the power of X, to the power of Y, is less than than Z.

These are just overly complex problems that would be meaningful if the power operation was not present in both cases.

**Bin (1,0) AUC < 0.65, Low Standard Deviation**

1) For each Store, predict whether the last minus the first value of Markdown4 is less than X.
2) For each Store, predict whether the last value of CPI is greater than X.

These are both well-defined, meaningful problems that are very hard, either due to their intrinsic difficulty or because the feature engineering library does not adequately generate features to predict them.

**Bin (1,1) AUC < 0.65, High Standard Deviation**

1) For each Store, predict whether the last minus the first value of Markdown4 column to the power of X, to the power of Y, is less than than Z.
2) For each Store, predict whether the last value of the difference between consecutive values of Markdown3 to the power of X, to the power of Y, is less than than Z.

These are very similar to Bin (0,1); they are just overly complex. Like those in Bin (0,1), they would be much more meaningful (and would probably achieve higher AUC) without the POW operation.

**Bin (2,0) AUC < 0.95, Low Standard Deviation**

1) For each Store, predict whether the sum of the difference of Markdown4 greater than X.
2) For each Store, Department combo, predict whether all of the Weekly_Sales column are less than X.

These are simple and somewhat meaningful problems, similar to Bin (1,0) and Bin (2,0) that happen to be easier than the ones in Bin (1,0).

**Bin (3,0) AUC > 0.95, Low Standard Deviation**

1) For each Store, predict whether the number of temperature recordings is greater than x.
2) For each Store, predict whether all of the values of the Store column are not equal to X.

These problems are similar to the (0,0) case, but allow us to generate features that separate the data. In these cases, the number of temperature recordings probably does not change much, and all of the store columns are either equal to X if X is the instance in question, or much more likely not equal to X if they are different.

**Bin (3,1) AUC > 0.95, High Standard Deviation**

1) For each Store, predict whether the sum of Markdown3 is greater than x.
2) For each Store, predict whether the sum of Markdown3 is less than x.

This is a simple and easy problem, probably because the mean value of Markdown3 does not change that much over long periods of time, and so its sum does not change much either. The standard deviation may be higher because in some special cases the mean does change a bit over time, so some training/testing splits produce slightly worse AUC.

**I. Conclusion**

In this paper we presented a language to enumerate a large, but finite and manageable, number of prediction problems from relational time-varying datasets. We also wrote an interpreter that, given a definition of a prediction problem, will generate inputs for feature engineering software, thus enabling us to develop predictive models simultaneously and without additional human brainpower. We demonstrate the efficacy of this language plus interpreter by enumerating multiple predictive problems, and then solving them.

This work allows for an automated *discovery* of knowledge using many different machine learning models, and hopefully provides a base for future projects that can go one step further to automatically choose meaningful problems to investigate further.

We consider Trane to be a first pass at this type of enumeration, and can think of many additional features or optimizations that would provide better generalizations and cover larger portions of the data science space.
List of publicly available data sets and their corresponding predictive problems, expressed in Trane. The first portion of the table is given by [https://www.kaggle.com/c/](https://www.kaggle.com/c/). These datasets include data released at international conferences- REC SYS, INFORMS.

<table>
<thead>
<tr>
<th>URL</th>
<th>Operations</th>
<th>Column(s)</th>
<th>Filter (applied to first ColOp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/yelp-recsys-2013</td>
<td>['IDENTITY', 'IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['business_id', 'rating']</td>
<td>all</td>
</tr>
<tr>
<td>/hivprogression</td>
<td>['IDENTITY', 'LAST_MINUS_FIRST', 'LT', 'None']</td>
<td>['col5'] (***)</td>
<td>all</td>
</tr>
<tr>
<td>/tourism1</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['y1..y518']</td>
<td>all</td>
</tr>
<tr>
<td>/acm-sf-chapter-hackathon-small</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['sku']</td>
<td>all</td>
</tr>
<tr>
<td>/chess</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['score']</td>
<td>all</td>
</tr>
<tr>
<td>/genentech-flu-forecasting</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['grant_status']</td>
<td>all</td>
</tr>
<tr>
<td>/acm-sf-chapter-hackathon-big</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['price']</td>
<td>all</td>
</tr>
<tr>
<td>/informsv2010</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['units']</td>
<td>all</td>
</tr>
<tr>
<td>/walmart-recruiting-sales-in-stormy-weather</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['votes.useful']</td>
<td>all</td>
</tr>
<tr>
<td>/yelp-recruiting</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['package']</td>
<td>all</td>
</tr>
<tr>
<td>/worldcupconf</td>
<td>['IDENTITY', 'IDENTITY', 'ORDER_BY', 'None', 'None']</td>
<td>['team', 'final_ranking']</td>
<td>all</td>
</tr>
<tr>
<td>/R</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['timestamp']</td>
<td>all</td>
</tr>
<tr>
<td>/pkdd-15-taxi-trip-time-prediction-ii</td>
<td>['IDENTITY', 'LAST_MINUS_FIRST', 'None', 'None']</td>
<td></td>
<td>all</td>
</tr>
<tr>
<td>/customer-retention</td>
<td>['IDENTITY', 'IDENTITY', 'ID']</td>
<td>*</td>
<td>date = MAX(date)</td>
</tr>
<tr>
<td>/RxVolumePrediction</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['presc_vol']</td>
<td>all</td>
</tr>
<tr>
<td>/genentech-flu-forecasting</td>
<td>['IDENTITY', 'IDENTITY', 'LAST', 'None', 'None']</td>
<td>['date']</td>
<td>(people_infected = MAX(people_infected))</td>
</tr>
<tr>
<td>/genentech-flu-forecasting</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['location']</td>
<td>(people_infected = MAX(people_infected))</td>
</tr>
<tr>
<td>/genentech-flu-forecasting</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['people_infected']</td>
<td>(people_infected = MAX(people_infected))</td>
</tr>
<tr>
<td>/riskiness</td>
<td>['EQ', 'ANY', 'None', 'None']</td>
<td>['loan_status']</td>
<td>all</td>
</tr>
<tr>
<td>/bike-sharing-demand</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['count']</td>
<td>all</td>
</tr>
<tr>
<td>/mschallenge</td>
<td>['IDENTITY', 'IDENTITY', 'ORDER_BY', 'None', 'None']</td>
<td></td>
<td>all</td>
</tr>
<tr>
<td>/predict-wordpress-likes/data</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['post_id']</td>
<td>all</td>
</tr>
<tr>
<td>/us-census-challenge/data</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['Mail_Return_Rate_CEN_2010']</td>
<td>all</td>
</tr>
<tr>
<td>/expedia-personalized-sort</td>
<td>['IDENTITY', 'IDENTITY', 'ORDER_BY', 'None', 'None']</td>
<td>['hotel_id', 'purchase_bool']</td>
<td>all</td>
</tr>
<tr>
<td>/caterpillar-tube-pricing/data</td>
<td>['IDENTITY', 'LAST', 'None', 'None']</td>
<td>['cost']</td>
<td>all</td>
</tr>
<tr>
<td>/restaurant-revenue-prediction/data</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['revenue']</td>
<td>all</td>
</tr>
<tr>
<td>/acquire-valued-shoppers-challenge</td>
<td>['IDENTITY', 'EXIST', 'None', 'None']</td>
<td>*</td>
<td>all</td>
</tr>
<tr>
<td>/predict-west-nile-virus/data</td>
<td>['EQ', 'IDENTITY', 'None', 'None']</td>
<td>['WnvPresent']</td>
<td>all</td>
</tr>
<tr>
<td>/coupon-purchase-prediction</td>
<td>['IDENTITY', 'IDENTITY', 'None', 'None']</td>
<td>['coupon_id']</td>
<td>all</td>
</tr>
<tr>
<td>heritagehealthprize.com/c/hhp</td>
<td>['IDENTITY', 'COUNT', 'None', 'None']</td>
<td>*</td>
<td>all</td>
</tr>
</tbody>
</table>

* Use any column
(***) First RowOp Argument: in_default
(****) Second RowOpArgument: -2
(†) First RowOp Argument: 1.15
(‡) First RowOp Argument: 1

**REFERENCES**


