Towards An Automatic Predictive Question Formulation

by

Benjamin J. Schreck

S.B., MIT, 2015

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2016

Certified by . . . .
Kalyan Veeramachaneni
Principal Research Scientist
Thesis Supervisor

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Towards An Automatic Predictive Question Formulation

by

Benjamin J. Schreck

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we designed a formal language, called Trane, for describing prediction problems over relational datasets, implemented a system that allows humans to specify problems in that language, and allows them to build models that solve them using real data. We show that this language is able to describe all 54 prediction problems on the Kaggle data science competition website[14] and so is comprehensive.

The implemented system consists of a web application connected to a server-side interpreter, which translates input from the web application 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 software that enumerated 1077 prediction problems automatically for the Walmart Store Sales Forecasting dataset found on Kaggle[16], and built models that attempted to solve them, for which we produced 235 AUC scores.

The web application also allowed us to collect 157 ratings from humans on the meaningfulness of randomly-generated prediction problems. We used these ratings along with an enumeration of 6105 prediction problems and 7 datasets to train a collaborative-filtering based recommendation system to propose meaningful prediction problems on new, unseen datasets.

Thesis Supervisor: Kalyan Veeramachaneni
Title: Principal Research Scientist

Acknowledgments

I would like to thank my mom, my dad, my brother, my other brother, my cat, my pole vault coach Dr. Patrick Barragán, my five roommates over the past year Nicolas Rakover, Lee Gross, Nathan Landman, Daniel Mendelsohn, and Kris Frey, my longtime friend Mack Bleach, my other long time friends Erik Waingarten, Aaron Thomas, and Chris Davlantes, and all the members of the MIT Track Team.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we provide a framework for categorizing analytics problems, and an algorithm for automatically generating these problems from raw data. We then provide a method for identifying meaningful subsets of these problems from the large pool of possible ones, and some initial results using both supervised and unsupervised learning techniques on a set of 590 data sets found online. Lastly, because our learning algorithm will only get better with more training data, we present a web-based system that allows anyone to add their own data set to our repository, which will then trigger an online update of our learning algorithm.

1.1 Motivations for prediction problem automation

Data science has traditionally been motivated from a top-down perspective. A data scientist has a particular question she wants answered, and attempts to design a predictive modeling question and apply algorithms to data in order to answer that question. Unfortunately, while scientific in nature, this approach is increasingly taxing the data scientist's time and effort by requiring her to define the predictive question ahead of time. In our current data-clogged world, where much more data exists than systems to make sense of it, this top-down way of approaching data science is severely limiting its potential.

An alternative way of thinking about data science that lends itself well to automation is that of solution-selection. Instead of relying on the human data scientist, whose time is incredibly valuable, to produce the right questions to ask from raw data, we could instead present her with all the meaningful questions up front, and, coupled with the ability to automatically solve it, show her the solution. We could then ask her to pick questions from these to study.

This way of thinking forces data science to automate, since it relies on machine understanding of data at a higher level than is currently the norm. An automated system would need to know all the possible ways data is organized, and all the possible combinations of data that an algorithm could model. Much more challenging, it would need to understand what aspects of data are meaningful to humans, in order to reduce the solution space it presents to a human data scientist.

Automating data science in this manner is currently possible, yet not yet adopted by the scientific community. We believe this is because data science is still performed too haphazardly. There is no agreed upon standard for data categorization. Instead, each analytics problem is treated as a unique application, generally all the way up until it is ready for machine learning (the quintessential feature matrix). Recent advances by the auto-ML community have made generalization possible one step further, and have automated feature extraction itself [18], [10]. However, the basic problem of identifying what types of questions are meaningful to ask about data is strictly left to the human domain.

1.2 Outline of Thesis

In chapter 2, we describe prior research on data and dataset categorization, as well as mention some notable limitations of the current literature and state of the field. We then proceed to make formalized definitions of fundamental data science concepts that will help motivate the later contributions.

In chapter 3, we introduce a language for describing *prediction* problems on time-varying relational datasets. We call this language Trane, and it enables the definition of an almost arbitrary set of questions to ask about data in this form, thus enabling brand new sets of meta-machine learning algorithms that break away from the theoretical machine learning researcher's obsession with the numeric feature matrix. We describe two subsets of the Trane language, called BigTrane and LittleTrane. BigTrane is a larger version of the language that enables the definition of almost any prediction problem that is meaningful to define. LittleTrane is a much smaller subset that is much easier to present in a bite-sized form and which is much easier to actually develop learning algorithms to learn common patterns in the language. It still encompasses a large swath of possible prediction problems, including the most common ones in the wild.

In chapter 4, we present a web application that allows humans to directly interact with LittleTrane. We built a Interpreter for Trane, and include a backend feature-engineering and machine learning library, so that users of the web application can interactively build prediction problems using LittleTrane, and then simply click a button to run those problems on a dataset and view the resulting AUC. As we will see in later sections, this system is incredibly powerful, as it enables an entirely new form of interacting with and annotating data. One important aspect of this is the ability for users to *rate* algorithmically generated prediction problems, and to use these ratings as labels for performing machine learning on the prediction problems themselves.

In chapter 5, we demonstrate the power of Trane on a real dataset: the Walmart Store Sales Forecasting dataset available from Kaggle[16]. We generate hundreds of prediction problems for each one, and then generate features and run these problems to investigate their predictive accuracy.

In chapter 6, we show that automatically learning which prediction problems in LittleTrane are meaningful is a tractable learning problem, and present an algorithm to solve it. We call this algorithm Pernican. We also describe how we are able to use the web application from section 3 to collect ratings from humans about which prediction problems in LittleTrane are actually meaningful.

In chapter 7, we discuss our contributions as well as future directions in which to proceed with this work.

Chapter 2

Laying the Groundwork: Data Set Categorization

2.1 Prior Work

There is a dearth of formality with regards to defining what data fundamentally is and how we store it at an abstract level. Formal definitions of data are needed at some level in the fields of data mining, statistics, machine learning, and database management systems, as well as others that are less relevant to this thesis. Database management systems (DBMS) research references data at the level of abstraction needed for computer programs, and so defines specific types that mirror programming language types (i.e. integers, floating point numbers, strings, dates, etc.) [30]. This is very close to the level of abstraction we will be working with in this thesis. DBMS also introduced a concept called the Entity-Relationship model, which helps define database schema that can then be translated to a canonical *relational* database [9]. Relational databases are ubiquitous, and lend themselves well to analytics problems because they can easily be translated into a single data matrix that machine learning and statistics problems always assume as input [43]. Machine learning and statistics textbooks typically brush over real definition of the nature of data, and skip right to application-specific data sets, where the data is either a standard feature matrix where each row corresponds to a unique entity and associated attributes [39]. In many cases they will go one step further and define the difference between numeric and categorical data, and for application-specific research like Natural Language Processing or Computer Vision they will assume data to be in strings of words or matrices of pixels [24], [37]. Smola [35] does a particularly comprehensive job at describing fundamental data types before diving into math and algorithms. Data mining textbooks come closest to the formal definitions and characterization of data that is necessary to build up a structure and language to describe analytics problems. Aggarwal [3] provides the best introduction we have found to data and data set characterization. The textbook carefully describes the entire data mining process, from data collection to interpretation of analytical results, and carefully describes the differences between data types, and between data sets that include dependencies between entities. We will first build on the definitions laid out by Aggarwal.

2.2 Fundamental Data Types

Fundamentally, structured data consists of one or more variables of definite types. While these types can in theory all be represented as binary strings of information, we can apply standard human categories that work in almost all cases. These categories are presented in the following table. There are two base types: Numeric and Categorical, sometimes called Quantitative and Qualitative. Different authors differentiate the two in different ways[44], [3]. Generally, Numeric data is defined as that with a definite ordering, such as numbers in $\mathbb{R}$. Sometimes this type is instead labeled Continuous and so does not include integers, which would fall into the Categorical class [44]. However, in this thesis both integers, and any ordered type that can be represented by numbers fall into the Numeric class. Categorical values are those that are distinct, and do not have a natural ordering. An example of categorical set is the set of items purchased at a grocery store. There may be similarities between different items, but there is no intuitive ordering between them. Clearly we could be more precise about these categories, and discuss the fundamental mathematical relations between pairs of items but for our purposes that is unnecessary.

From these, we can derive other, possibly compound, types that mirror some aspect of the real world. For instance, we define a date/time type, which is just a numeric type that acts as a time axis. Similarly, we can define a spatial type, which consists of pairs of numbers (latitude & longitude) that act as a location axis. We also define a text type, which is an ordered sequence of categorical values, each representing a character in a human natural language. We could go on and define Image, Video, and Audio types, and continue to build up more and more complex types, but for our use case in understanding the types of questions that are asked about data these are unnecessary. We restrict ourselves to Numeric, Categorical, and Date/Time, leaving out Text-based data for simplicity as well as the spatial type mentioned above out because we will be focusing on temporal prediction problems, which are more natural than spatial prediction problems due to the inherent arrow of time that prevents prior knowledge of the future.

1. Numeric
   -subtype: integer, real number, complex number
2. Categorical
   -subtype: general, boolean, hash-string
3. Date/Time
   -desc: numeric type as a time axis
4. Text
   -desc: natural human language like English

2.2.1 Theoretical Aside: Implied Data Dependencies

Another way to think of the more complex types (and indeed, the order present in numeric types as well) is in terms of implied dependencies between values [3]. At the simplest level we have standard categorical data, which can only be represented numerically as completely orthogonal one-hot vectors of dimension d, the total number of elements. There is no relation between each value. Every other type comes about due to an added implied dependency between values. The numeric type includes prior information about the ordering of elements, and in the case of the real numbers and the integers, the distances between each element. Another common dependency is the graph model, where different values have arbitrary relationships defined as (potentially directed or weighted) connections between them. For instance, a social network data set contains undirected, generally unweighted connections between individual values which can be thought of as nodes in the graph.

These relationships can be arbitrarily complex. Since we restrict ourselves to temporal predictions problems, we do not consider arbitrary dependencies between data, though that could be a good direction for future work. The data we work with must fall into the three types defined above- numeric, categorical, date/time.

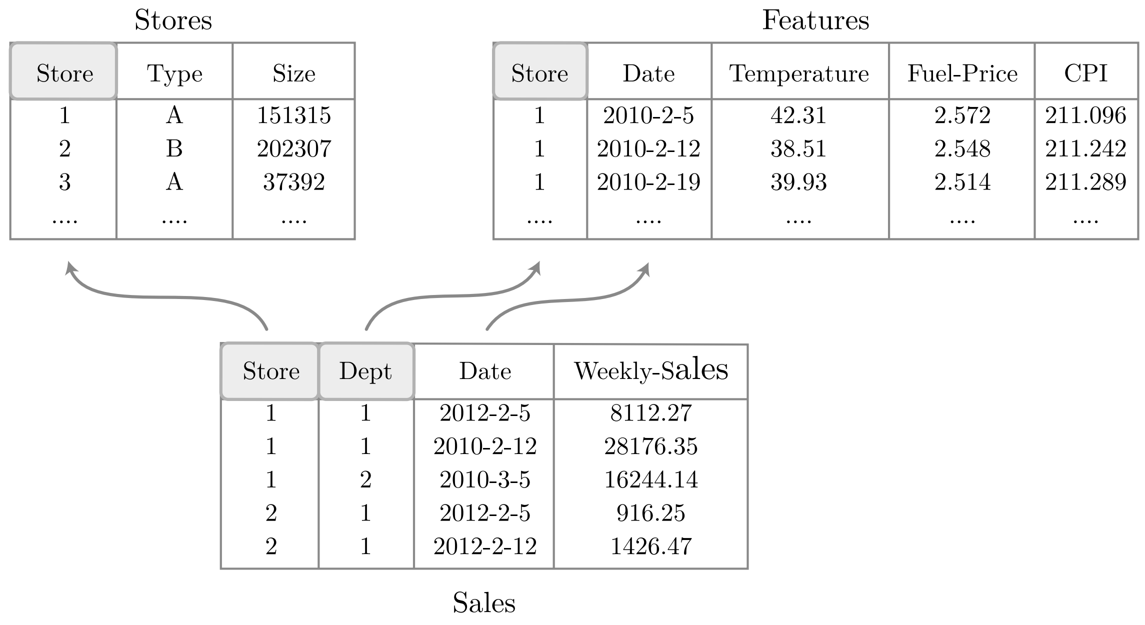
2.3 The Relational Dataset

For the context of this thesis, we also restrict ourselves to data that can be represented relationally. Let us define relational data as data that can be represented in table-like structures, linked by numeric indices or categorical hash-strings serving as the "relations". Each table contains a definite number of columns, each with a predefined data type, and an indefinite number of rows, where each row contains not more than one value for each column [9]. The vast majority of information collected today to be used for data science, aside from audio, images, and video, can be represented relationally, even though it may not be stored that way [30]. Much of the useful information for data mining and analytics from web logs, key-value databases in a NoSQL format, and raw documents can be extracted from these databases into relational tables [4]. Relational datasets works well as a bridge between storing data collected from the real world and providing a numerical matrix as input to machine learning applications.

For the purposes of this thesis, we also distinguish between *datasets* and *databases*. From now on, a *dataset* is defined as a set of schema for organizing data as well as the data itself, while a *database* is a particular stored instance of that data in a database management system on a computer somewhere. We mainly deal with the more abstract *dataset* for the rest of this thesis.

2.3.1 Relational Entities

In dealing with relational datasets, we can also define the notion of an *entity* [9], [30]. An entity is a categorical column representing some distinct object or idea. This definition is purposefully vague, because almost any categorical column can be thought of as an entity. In most cases, the designer of the data set had some subset of columns in mind as those representing entities, and thought of the rest of the columns as descriptive features of the entity, potentially changing in time. Entity *instances* are defined as the values present in that entity column. An example makes this concept more clear.



Stores

Store	Type	Size
1	A	151315
2	B	202307
3	A	37392
....	....	....

Features

Store	Date	Temperature	Fuel-Price	CPI
1	2010-2-5	42.31	2.572	211.096
1	2010-2-12	38.51	2.548	211.242
1	2010-2-19	39.93	2.514	211.289
....	....	....	....	....

Store	Dept	Date	Weekly-Sales
1	1	2012-2-5	8112.27
1	1	2010-2-12	28176.35
1	2	2010-3-5	16244.14
2	1	2012-2-5	916.25
2	1	2012-2-12	1426.47

Sales

Figure 2-1: Simplified version of the Walmart Store Sales forecasting dataset, with links connecting columns in different tables. Store and Dept are both entities, and the Store entity connects all three tables together. The Date column defines a time axis, which we can use for prediction.

Entities can function as identifiers or keys, and in general some combination of entities and a time axis should uniquely identify a table. In the Walmart dataset in 2-1, the Store entity uniquely defines the Stores table, the Store entity and Date column uniquely identify the Features table, and the Store entity, Dept entity, and Date column uniquely define the Sales table. Entities also provide the relationships between tables. Each Store entity in the three tables above describe the same underlying values, namely different physical Walmart stores. Each row in the Sales table describes the Weekly Sales for a given week, store, and department. By consulting the row in the Stores table with the same value for the Store column, we can find additional information about that store, such as its size.

2.3.2 Squashing Relational Tables Together

A nice property of relational datasets is that multi-table datasets are no more expressive than single-table datasets. Any multi-table dataset can be "squashed" into a single, larger table by following relational links [9], [30], [41]. Furthermore, thinking about data in terms of a single-table matrix lends itself well to machine learning problem formulation, and indeed is how almost all of the input data in the literature is defined [24], [43]. Multi-table datasets are more desirable for storage due to their ability to reduce redundancy, and when redundancy is eliminated, these tables are referred to as a *normalized* dataset. Similarly, a redundant, single-table version is called *denormalized* [41], [9].

We will not describe denormalization algorithms in detail that enable the conversion from a multi-table format into a single-table one, and instead refer the reader to [33]. However, for simplicity we will generally refer to denormalized single-table datasets in this thesis.

2.4 Classes of Expressive Power

Traveling further in our categorization of data, we can separate relational data sets by their expressive power, which fundamentally reduces to the number of entities and time axes.

Object Descriptions: Single Entity, No Time Dependence

The simplest case is a dataset with only a single entity, and no time axis. Intuitively, this type of dataset is a list of things (objects, concepts, or basically anything that can be represented by a noun), and columns describing that thing (think of these as adjectives). Call the dataset *consistent* if each entity is on its own row, because any duplicate rows are unnecessary, and non-duplicate rows of the same entity contain competing descriptions. If the descriptive column is allowed to contain multiple values for the same entity, then it can instead be represented as multiple columns with a single value for each entity.

The only type of supervised machine learning we can define on this type of dataset is a classification or regression over operations on the values in each row, possibly with a filter to specify which rows to include in the learning algorithm. The simplest case is that of predicting a binary descriptive column for each entity instance, effectively each row in this

Fruits	Glycemic Index	Average Weight (lb)	Average Width (cm)	Tons Consumed per Year
Apple	39	0.33	8.0	$80.8e10^6$

Table 2.1: Fruits schema

case. A more complex case is a series of operations defined over each row. Imagine the schema is defined as in 2.1.

A more complex problem could be defined as predicting whether the following equation is greater than $\frac{1}{2}$:

$$\frac{\sqrt{\text{Glycemic Index}^2 + \text{Average Weight}^2 + \text{Average Width}^2}}{\text{Pounds Consumed per Year}}$$

This problem is probably totally meaningless, but is well-defined, and it is easy to think of simpler, more meaningful problems, such as predicting the Pounds Consumed per Year column. In English, we label the verb associated with identifying unknown data as *predict*, yet here we are not exactly predicting since there is no time-component. A better verb is *classify*, and indeed that is why these problems are called *classification* problems.

To be complete, we could add one more level of complexity by allowing operations over all the values for a single column. However, these operations would operate over multiple entity instances and could potentially violate the ability to separate the data into training and testing sets for accurate estimations of the true error, and in deployment we would have no way of applying these operations except on the data already seen. For instance, a common operation that does this is normalization, where the values of each column are normalized to the interval $[0, 1]$. This is impossible to do if the maximum or minimum is unknown. We ignore these types of operations for the rest of this thesis, although later on we do define operations over multiple rows and columns, but for individual entity instances.

Graph Descriptions: Multiple Entity, No Time Dependence

A slightly more expressive dataset is one that contains multiple entities in the table. In this case, a *consistent* data set is one in which in each combination of entities is a unique row. We call this kind of dataset a graph description since it describes both attributes of the entities themselves as well as connections between entities. There are a few additional

User	Movie	Rating
1	Honey, I Shrunk the Kids	5
1	Pitch Perfect	2
1	Inception	4
2	Shawshank Redemption	1
2	The Godfather	1
2	Pitch Perfect	5

Table 2.2: Multiple entity table: Users and Movies.

complexities introduced by this type of dataset in terms of defining prediction problems. The first is which entity to define the problem over. If defined for a single entity or for a combination of entities fewer than the total number of entities, then all of the rows for each instance of that entity (of which there are multiple now that the uniqueness of a row is determined by all of the entities) need to be aggregated together in some way. This could be through a sum, an average, a separation into multiple columns, or some thing more complex. However, if the problem is defined for a combination of all the entities (i.e. if there are two entities E1 and E2, then each instance is a unique combination of an E1 value and an E2 value). If the additional entities are aggregated into a list, then one can define another type of machine learning problem: that of recommending instances of other entities. Without loss of generality, if the problem is defined for a single entity, then for each additional entity, one can recommend instances not present in the main entity instance's list. For instance consider the classic problem of movie recommendation, made famous by the Netflix prize [19]. We show a visualization of the dataset in 2.2. The task is to recommend movies for each user. There are thus two entities, users and movies. A (user,movie) pair is present in the dataset if the user has rated that movie. We can aggregate over each movie the user has rated to produce a list of movies and ratings, and use that list to recommend additional movies that we determine the user would rate highly.

We still do not consider this *predictive* analytics even though we are inferring movies that the user would rate highly. If there was an additional time column detailing the time at which the user rated the movie, then this we could predict future ratings and this would become a prediction problem.

Simple Time Series: Single Entity with Time Dependence

An even more complex case is one where we consider one of the columns as a time axis. Now we have the power to predict values for an entity at future points in time. Current values and the way they are distributed in time can be used to build a model of the underlying time-dependent distribution of the data points, and if we assume continuity of the mechanism for that distribution in time, then we can use the model to predict future values (results from the online learning community show that we even model time-distributed data when the data generation mechanism changes over time, [7].

A consistent time-dependent single-entity data set is one where each combination of entity instance and time point is a unique row. That is, each instance can have multiple values so long as they are each unique points in time. If we aggregate all the timestamped rows for a single entity instance into a single row, we regress to the simple object description case and can perform correlative classification and regression.

In this thesis we neglect the possibility of multiple time axes because only one will be considered the predictive axis. Different time axes can be used for different entities, but considering multiple axes as orthogonal makes little intuitive sense for prediction (i.e we live in a world with a single time axis).

Dynamic Graph Description: Multiple Entities, Time Dependence

The final case is where the data set contains multiple entity columns as well as a time axis. A consistent time-dependent multi-entity data set is one where each combination of time point and an instance of each entity is a unique row. In this case we can think of the dataset as a dynamic graph, where each row identifies connections, or links in the graph, between entities. These connections change over time according to the time axis.

In this case, we could regress to the stationary graph description model by combining all the rows for each combination of entities into a single row and performing classification or regression on some operation of the columns. We could choose a single entity and aggregate the rest of the entities and descriptive columns in some fashion to regress to the single entity, time dependent model.

An interesting problem that arises form data in this form is *recommendation*, where we select an entity, and for each instance of that entity we predict which instances of other entities we believe will be associated with the original entity instance in the future. For instance, the dataset might contain entities for users and movies, and for each pairing it contains a rating indicating the rating that the user gave to the movie, and a timestamp. We can use past ratings of a user, as well as the ratings of other users, to predict which movies the user will rate highly in the future, and thus *recommend* those movies to the user.

2.4.1 Predictive vs. Correlative Analytics

The classes of expressive power of a data set described above limit the types of analytics problems that can be performed over that data set. More concretely, we define two classes of analytics problems: correlative and predictive.

Any of the above-mentioned relational data sets can be used to perform correlative analytics, which ignores implied dependencies between rows other than simple case of numeric data (in the context of this thesis this dependence between rows means a time-axis, since we are not considering spacial- or graph-type data). Correlative analytics consists of predicting attributes of an entity. These attributes can be derived from multiple operations over the data set and need not be present in their own right as columns in the data set. However, this type of problem assumes a stationary data set. The distribution of the underlying data does not change in time, and so we cannot predict multiple values of the same attribute for the same entity. In data sets that include a time axis, the attributes for which the correlative analytics problem is looking for need to be stationary values that do not depend on time. These can be actual columns that remain the same for any one entity across time, or can be some derived aggregate value that is assumed to be constant, such as the mean or maximum of a value across time.

Only data sets with a time axis can be used for predictive analytics. Predictive analytics problems are ones where known values of a single entity are used to predict unknown (generally future) values of the same entity. In predictive analytics we need not assume the underlying distribution of the data is stationary, and we can actually model the way it changes in time. Many times we will not need to (for instance if we assume an I.I.D. generating process for the data), but we can if we feel inclined. There are more subtleties and complexities associated with these types of problems, and in general we can easily regress to the stationary, correlative case using a simpler, modified version of the language and automation described later in this thesis. Therefore, from here on we only consider the predictive case.

Theoretical Aside: Different Types of Predictive Axes

A theoretical issue at play here is the notion of general human intuition for data and how it affects the way we collect and understand data sets. Theoretically any numeric column, column with an implied dependency, can be used as an axis for performing predictive analytics [5]. Our assumption behind the ability to predict unknown (larger) values on the axis is the underlying continuity of the data-generating mechanism, or at least the ability to model the change of the mechanism using known data. This assumption holds in the time domain in many cases because of the real-world "arrow" of time. It also may hold in a graph domain, where individual data points are connected arbitrarily through nodes, when that graph represents real world networks like connections between humans or or groups of people. An example where it might not hold at the scales and applications we typically care about is the space domain. To illustrate this, let us run through an example of where the continuity assumption holds and does not hold for the time and space domains.

First, we start with a data set recording levels of traffic on major highways in a city. Because the number of cars passing through any one location over the course of a day or a week is a function we can model that will probably remain more or less constant longer scales of time (like over multiple months or years), we can use the time axis as our predictive axis and predict future levels of traffic in say, one month's time.

In the space domain, the equivalent problem is using the number of cars that pass through one stretch of highway over time to estimate the number of cars that pass through a different stretch of highway over time. In this case, the numbers of cars passing through the two highways are probably not highly correlated and so the continuity principle does not hold. It *would* hold if these stretches of highway were part of the same road or were very close to each other.

Clearly, there are cases where different predictive axes can be used, and cases where the standard time axis is not useful for predictive analytics, but those cases are much less ubiquitous or useful to automate in the data science community. Hence for this thesis, we restrict ourselves to the time axis.

2.5 Predictive Analytics: The Prediction Problem

At the core of predictive analytics is the actual question being asked. That question typically takes the form of:

"What quantities can we predict in the future, given the data at hand?"

From here on, we refer to this question as the prediction problem. We would like to formalize the structure of this problem, automatically generate many of them, and learn which ones are meaningful. As a first step, we define here the attributes that make up a prediction problem.

2.5.1 Feature-Generating and Label-Generating Data

we need an entity column along with attributes that change over time. For each entity, we predict some future value, which could be the value of one or more attributes or an operation on some subset of all the attributes. Typically these problems require building a model that is trained via supervised learning [5]. Therefore, we need to define operations that split our dataset into two parts: features and labels. We assume both features and labels can be computed from columns in the data, and furthermore that both of them are computed from *multiple* rows of data, due to the time dependence. Therefore, we need to define one or more periods in time to use as *label-generating* data, and for each of these periods we also define earlier blocks of time to use as *feature-generating* data. We can usually think of this separation as that of a *cutoff* point, potentially defined uniquely for each entity, where rows with a timestamp that belongs to the period of time prior to the cutoff point is set aside as feature-generating data, and rows with a timestamp after the cutoff is used as label-generating data. There may also be a *lead* time defined, where some amount of data near the cutoff point is left out and not used for either feature- or label-generation.

In some cases it may make sense to periodically hold out data, and break up entities into multiple (feature, label) row pairings. As an example, imagine heart-rate ECG data, where we want to build a model that can predict heart-beat frequency in one hour's time. The scale of recordings allow entities to be broken up over longer periods of time. We may split up the data into periods of three hours at a time, so that we can use the first hour to generate features, the third hour to generate labels, and the second hour as ignored lead-time data.

2.6 What sorts of things can we predict?

The data science literature typically describes three types of supervised time-sensitive prediction problems: regression, classification, and recommender systems (one can argue that reinforcement learning is also time-sensitive but it is not predictive in the same way) [6], [5], [40], [24], and [35]. Regression problems consist of predicting numeric values; classification problems consist of predicting categorical values; recommendation problems or systems consist of predicting subsets of entities, possibly with an ordering defined on them. These are not specific enough for our use case, as we want to be able to generate many different prediction problems per data set.

As first step, we present a hierarchy of prediction problems in 2-2 based on the values we are predicting. Each of these problems is defined for each entity instance. For instance, if the entity column is "Houses", and we are predicting a numeric variable like "Market Value", then for each house we predict a different number that represents the "Market Value" for that house. We do not claim this is a complete list, but it represents a wide enough swath of problems with which to further develop a formalized language to describe them.

2.6.1 Human-Annotated Problems

Lastly, before we present our language for describing prediction problems, we mention that many problems exist in which the label is not calculated but instead collected from humans who are treated as *oracles*. For instance, imagine a setting where doctors are asked to label the point in time at which an ECG recording reads that a patient has entered cardiac arrest, or in the strictly correlative and non-predictive setting, humans may be asked to identify

– **Numeric Variable**

1. Value itself
2. Threshold of that variable, possibly with an any/all clause
3. Operation on variable
4. Operation on variable $\rightarrow$ threshold, possibly with an any/all clause

– **Categorical Variable (or Boolean)**

1. Value itself
2. Any/all of multiple Boolean values

– **Count or Exists**

1. Number of rows that exist (regression)
2. Thresholded number of rows that exist (classification)
   – typically whether or not the count is greater than 1, as in a customer churn example

– **Other Relational Entity(s)**
– Each of these defines a type of recommendation problem

1. Unordered
   –select a subset of entity instances
2. Order by a numeric variable
   –select an ordered list of entity instances
3. Associate with a numeric variable
   –select a subset of entity instances along with a numeric value

– **Date at which Numeric Variable Attains a Threshold**
– predict *when* something happens

– **Categorical Value at which Numeric Variable Attains a Threshold**
– predict descriptors of the situation when something happens

– **Operation on a Time Axis** – predict the average time between recordings in the future

– **Time Axis Itself**
– predict how long until next recorded value

Figure 2-2: Simple hierarchy of prediction problems which answers the question: what can we predict? In the next chapter we will codify these intuitions into a formal language that can actually be run implemented automatically.

objects that are present in images, or parts-of-speech of words within sentences [36], [32]

Chapter 3

Trane: A Language to Formally Express Prediction Problems

In this chapter, we describe a language that allows us to formally express numerous prediction problems given a time varying relational dataset. With this, we can then create algorithms to enumerate a number of prediction problems. We do not claim that this space is complete, or that all possible prediction problems can be defined this way; however, almost any problem that we have encountered can map to this space, and we believe the space is large enough to be practically useful. Later in this chapter, we show that every time-dependent problem on the data science competition website, *Kaggle*, can be mapped to this expressive language [14].

3.1 On the name

We named our language *Trane*after the late John Coltrane, legendary jazz saxophonist and improviser. We like to think that *Trane*allows us to improvise over data. With a formally defined language, we can now arbitrarily construct predictive modeling problems that effectively ask questions about data. If the *operations* we define in this language are musical notes, then defining a prediction problem in *Trane*implies stringing those *operations* together like a John Coltrane saxophone solo.

3.2 Two necessary components

A typical data set contains an *entity*and many instances of this *entity*. The *entity*may have numeric or categorical values associated with it that change or vary with time. For example, let us consider a data set that records several taxis' locations at different time points during a typical day. In this example, "taxi" is an *entity*, and it appears repeatedly. Each data point has $< taxi\_id, timestamp, latitude, longitude >$, where $taxi\_id$ uniquely identifies each individual taxi.

Each prediction problem needs a set of data points that can be used for making the prediction, and the value (called *labels* for classification problems) it needs to predict. To fully define a predictive problem, the language needs to define two things:

1. **Data segments for each *entity-instance***: In the language, we have to define two segments of data (portions of the *entity-instance*data). The first segment is used for learning the model, and the second 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 and a logic to choose them. Section 3.3 details the process of defining the time points.

2. **The sequence of operations that computes the label(s) for each *entity-instance.*** The labels or values we are interested in predicting are often derived from the data set itself, and may make use of all the values of each entity. For instance, 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 *labellabel-computing operations* and the columns they are derived from *label-generating columns*.

In the next subsections, we describe how *cutoff*and end points can be defined, and the ways in which transformation operations can be designed and applied to an *entity-instance*'s data to generate labels.

3.3 Segmenting the data

Our first step in defining a prediction problem is to define both the segment of data that can be used to build a prediction model and the segment of data that would be used to generate a label we are trying to predict. Before we present the logic for defining each of these two segments, we define the different time points that bound these segments.

Cutoff point: The last time point at which we allow data to be incorporated in the predictive model.

End point: The last time point we allow in our label computation.

Black-hole point: The time point for each entity instance at which the *label* is deterministically known.

3.3.1 Determining the slice for learning the model

This segment is the data prior to the cutoff point. We can define the cutoff point either:

1. as an explicit point in time
2. as a point in time for each entity instance
3. as some percentage of the total duration for each entity instance or overall duration
4. as some amount of time after the first recorded time or before the end point

To be able to define the cutoff point using 2, 3, 4, one needs to define the *end point* and *start point* for an entity instance's data. Since all instance's may not start and end at the same time points, we define a number of additional time points as shown in Fig. 3-1. Fig. 3-1 shows possible time points from which to define a single, overall cutoff (or end point - however, for the purposes of explanation, we assume the end point is the last recorded time point). We define three different start constants and three different end constants. Start 1 is the *earliest* first recorded time out of all entity instances, Start 2 is the *latest* first recorded time among all instances, and Start 3 is the *median* first recorded time. End 1, End 2, and End 3 are defined similarly for the last recorded times. We define Durations 1-5 as differences between various end and start times. The constants are listed and explained in table 3-2.

Cutoff point examples

1. **Single cutoff at** 10% **of Duration 1** This means we would first find the last recorded value out of all the instances: End 2, as well as the first recorded value out of all instances: Start 1. We subtract them to produce a timedelta object, multiply it by 10%, and add to Start 1. Imagine we are predicting student dropout from an online course, and we want to allow the model to use the first 10% of recorded data in order to predict which students had dropped out at 20%.

2. **Cutoffs per entity at** 10% This means that we take the first 10% of recorded time for each instance instead of calculating a single cutoff threshold. This value for cutoff works better for the taxi dataset, where each instance's start and end times might be wildly different.

3.3.2 Determining the slice for computing the label

The slice of data used to compute the label for an entity instance is the data between the cutoff point and the end 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 taxicab ride length was greater 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. Lastly, we define a *use_known* flag as a compiler directive to the *Trane*compiler to use pre-cutoff data when calculating labels, and so for this problem the *use_known* flag would be set to True.

3.3.3 Adjusting the cutoff based on the Black Hole Point

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

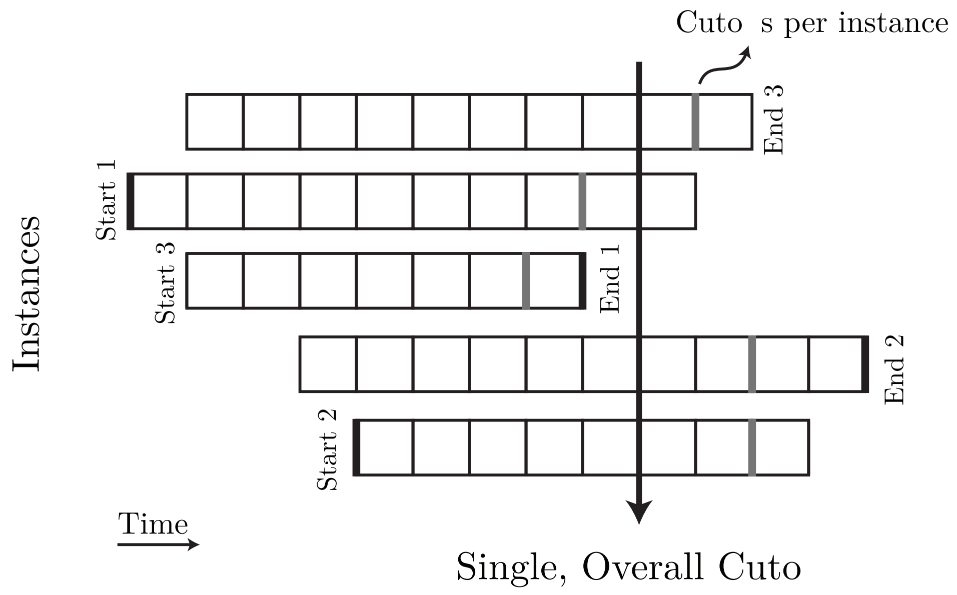


Figure 3-1: Visualization of ways to define cutoff (and end) points. One can either define a single cutoff or end point for all instances, or can define separate points for each instance. Per instance cutoffs are calculated in terms of the each per instance start and end values. If defining a single cutoff, one can define it in terms of various start, end, and duration constants. We visualize these constants here. Start 1 is the first recorded value out of all entity instances, while Start 2 is the last recorded starting point out of all entity instances. Similarly, End 2 is the last recorded value out of all entity instances, while End 1 is the *earliest* last value out of all entity instances. Start 3 is defined as the median of instance start times, and End 3 is defined as the median of instance end times. Durations are defined as differences between End constants and Start constants.

Constant Name	Explanation
`overall_start1`	First recorded value out of all entity instances.
`overall_start2`	Last recorded starting value out of all entity instances.
`overall_start3`	Median of entity instance start times.
`overall_end1`	Earliest last value out of all entity instances.
`overall_end2`	Last recorded value out of all entity instances.
`overall_end3`	Median of entity instance end times.
`overall_duration1`	`overall_end2` - `overall_start1`
`overall_duration2`	`overall_end2` - `overall_start2`
`overall_duration3`	`overall_end1` - `overall_start1`
`overall_duration4`	`overall_end1` - `overall_start2`
`overall_duration5`	`overall_en3` - `overall_start3`
`per_instance_start`	First recorded value, separately defined for each instance.
`per_instance_end`	Last recorded value, separately defined for each instance.
`per_instance_duration`	`per_instance_end` - `per_instance_start`

Figure 3-2: Listing of constants available for defining cutoff and end points. Constants prefixed with `overall` imply that they are constants to be used for a single defined cutoff or end point over all instances. Constants prefixed with `per_instance` imply that they are defined per instance.

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 entire data set is used to compute the label. In this example, the label-generating function is a *step-function*, meaning that if we plot the binary True/False label against the endpoint, 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. Thus in this case, if the cutoff point is after the BH-point, we shift it to before the BH-point.

Customer churn is another example of a non-trivial BH-point. Typical available data again consists of user-website interactions, with churn ordinarily defined as inactivity, en-

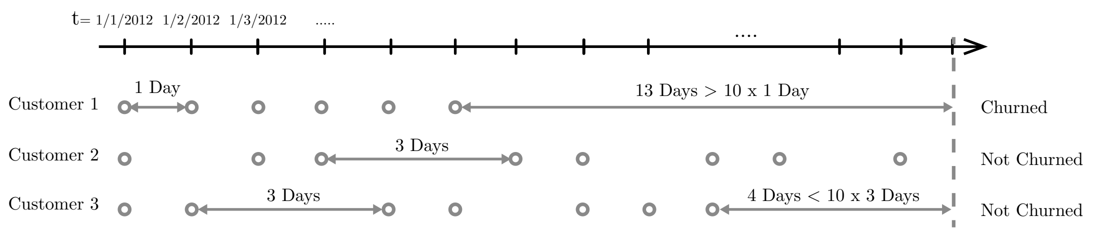


Figure 3-3: Visualization of the customer churn example. The points in time that each customer interacted with the website are shown as circles laid out horizontally over the time axis. We assume in this example that no data points were collected after the dotted line. We can see that Customer 1 has churned by the end point indicated by the dotted line, since his or her last recorded value was 13 days prior, and the maximum delay between two previous recorded values was a single day. Customer 2 has not churned, since her last recorded value was only a day prior to the end point, and Customer 3 has not churned since his last recorded value was 4 days prior to the end point.

acted when the user stops visiting the website. Unless the necessary length of inactivity is predefined, this example requires knowledge of the entire data set to determine with 100% certainty that a user has indeed churned. However, because our confidence that a user has churned increases as his or her last interaction grows more distant, we can define it approximately. We label an instance as False (churned) if the amount of time between the instance's last recorded value before its end point and the end point itself is significantly greater (10x) than the largest spacing in time between two previously recorded values for that instance. Figure 3-3 makes this more specific.

3.3.4 Setting different time points in *Trane*

Let us now make explicit how various settings for the cutoff and end point are defined in *Trane*. These are defined by setting some flags. Below, we show these options.

```
cutoff_settings = {
   explicit:      Boolean
   specific_date: Datetime
   offset:        Timedelta
   offset_from:   String
   percentage:    Float[0->1]
   percentage_of: String
 }
endpoint_settings = {
   explicit:      Boolean
   specific_date: Datetime
   offset:        Timedelta
   offset_from:   String
   percentage:    Float[0->1]
   percentage_of: String
}

misc_settings = {
   lead:          Timedelta
   step_function: Boolean
}
```

Cutoff and end point settings are very similar to each other.

Cutoff from Settings

1. If `explicit` is True, then we define the cutoff according to the following options, otherwise we pick the maximum cutoff before the black hole point

2. If `specific_date` is True, then we simply cut off at that date (if an instance has no values before that date, we ignore it).

3. Else if `offset` is True, then we check the `offset_from` flag to see where to calculate the offset. If `offset_from` is a start constant, we add it to the start constant to create the cutoff. If it is an end constant, or the explicitly defined end point, we subtract the offset to create the cutoff. This flag cannot be a duration constant.

4. Else if `percentage` is True, then we calculate the percentage of the constant defined by `percentage_of`, which must be a duration.

Endpoint from Settings

1. If `explicit` is True, then we define the endpoint according to the following options, otherwise we pick the last recorded value for each entity.

2. If `specific_date` is True, then we simply pick the end point at that date.

3. Else if `offset` is True, then we check the `offset_from` flag to see where to calculate the offset. If `offset_from` is a start constant, we add it to the start constant to create the end point. If it is an end constant, we subtract the offset to create the end point. This flag can also be defined as the cutoff point, in which case the end point is determined by adding the offset to the cutoff point. This flag cannot be a duration constant.

4. Else if `percentage` is True, then we calculate the percentage of the constant defined by `percentage_of`, which can be either a duration constant, or an end constant. If labeled as end constant, the end point is calculated through the following operation: (cutoff point + end constant)$*$ percentage.

3.4 Label Computing Operations

A Trane problem also consists of a sequence of operations applied to a set of columns in order to generate the labels for each *entity-instance*. We define the two main types of operation on a table: Row Operations and Column Operations.

Row Operations take in a single row as input, and output a new row, possibly with a different number of columns. We define one input parameter for the list of columns the Row Operation operates over, and another to specify whether the names of any of the columns change. As an example, the GT, or greater than, operation compares its input to a threshold given as a parameter. It outputs a single Boolean value for a single input row.

```
RowOp:
    -inputs:
        --optype (can be a list of types where each index refers
        to a different column)
        --columns (relevant columns to operate over, use * for all columns)
        --as_names (list of names to rename the columns to, defaults
        to the same as the input column names)
        --args (additional arguments such as threshold values, optional)
    -outputs:
        --new row
```

Column Operations take in a whole table as input, as well as an entity column, a Row Operation to apply on each row, and a Filter function that specifies which rows to include in the calculation. These operations group the table by entity, and apply themselves to each group separately. Each entity contains the information needed to produce the output label (or labels, for a multi-label problem) for a single input data point.

The output of a column operation is a new table, generally with a smaller number of columns, also grouped by entity. One can further divide column operations into *aggregation* operations and *transformation* operations. Aggregation operations combine all rows into a single row, while transformation operations return at least two rows as output, and generally do not change the number of rows from their input to their output. `Count`, which counts the total number of rows, is an example of an aggregate operation, while `Diff`, which returns the difference between neighboring row values, is an example of a transformation operation. Lists of possible operations are provided in the following section. Since transformation operations return multiple rows, it sometimes makes sense to chain these operations together. Aggregation operations cannot be chained in this way.

We therefore include in the language an argument to each operation naming the data source, which might be the output of a previous transformation operation.

```
AggOp:
    -inputs:
        -- entity
        -- time_axis
        -- data_source (either "base" or output of previous column operation)
        -- optype (which operation to use)
        -- col (relevant columns to operate over,
        optional if only one column in input)
        -- RowOp
        -- FilterOp
        -- args (additional arguments such as threshold values, optional)
    -outputs:
        --single table row, potentially with multiple columns
TransOp:
    -inputs:
        -- entity
        -- time_axis
        -- data_source (either "base" or output of previous column operation)
        -- optype (which operation to use)
        -- col (relevant columns to operate over,
        optional if only one column in input)
        -- RowOp
        -- FilterOp
        -- as_names (list of names to rename the columns to,
        defaults to the same as the input column names)
        -- args (additional arguments such as threshold values, optional)
    -outputs:
        -- new table
```

3.5 Filters

Additionally, 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. Filter operations can be arbitrarily complex in theory. Here we limit them to essentially the form: `[OP]([COL]) == [VALUE]`, where each element in `COL` is applied an operation and compared to `VALUE`. We also allow, for a limited set of operations like `MAX` and `MIN`, `[OP]([COL]) == [OP]([COL])`, where the right hand side looks over the whole column first to find the MAX or MIN. A 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

```
(HOUR(col1) == 3:00) && (IDENTITY(col2) > 5))
```

. See 3-4 for details.

3.6 Language Syntax

The full syntax of the prediction problem language requires at least one column operation, at least one row operation, and a filter operation, applied to the original data set. However, all of these could simply be `IDENTITY`, or, for the filter operation, `ALL`.

Because they only return a single row, the output of aggregation operations cannot be fed as input to additional column operations, but the output of transformation operations can.

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.

Lastly, we allow Boolean combinations of several sub-prediction problems, provided the output of the sub-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.

```
FilterOp:
    -inputs:
        --EITHER ARGUMENTS:
        --op1 (ALL, JOIN, DAY_OF_WEEK, HOUR, etc.)
        --col1 (column to operate over, only used if op1 is not JOIN or ALL
        --subfilter1 (left-side filter for JOIN op)
        --subfilter2 (right-side filter for JOIN op)
        --comparator (comparator like [==, <, >], only used if op1 is not JOIN or ALL)
        --joiner (Boolean op like [AND, OR, NOT], only used if op1 is JOIN)
        --op2 (optional second operation like [MAX, MIN] to apply to col2, only used if op1 is not JOIN or ALL)
        --val (threshold to compare to op1(col1), only used if op1 is not JOIN or ALL and op2 is not defined)
        --col2 (second column, used if op2 is defined)
        --OR STRING:
        --"[OP]([COL]) [COMP] [VAL]"
        --"[OP]([COL]) [COMP] [OP2]([COL2])"
        --"([SUBFILTER1] [JOINER] [SUBFILTER2])"
    -outputs:
        --Boolean function over each row
```

Figure 3-4: Filter Op Specificatino

3.7 Possible Operations

Many more operations are possible, but listed in tables 3-5, 3-6, 3-7 and 3-8 are all those we needed to define the Kaggle data sets in this language, as well as a few additional ones to define some internal data sets we were familiar with. The tables also explain the output types and allowed input types of each operation.

3.8 BigTrane and LittleTrane

We distinguish between two subsets of the Trane language: BigTrane and LittleTrane. BigTrane includes the entire language, while LittleTrane includes a simpler subset to enable easier learning of meaningful problems.

3.8.1 LittleTrane

LittleTrane is much simpler than BigTrane, yet still potentially defines thousands of prediction problems per dataset (as opposed to potentially hundreds of millions in BigTrane).

- We cap the nesting of transformation operations at two levels, 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 sub-problems.
- We do not define arguments/thresholds for the operations.

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

Figure 3-5: 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.

Transformation Operation	Arguments	Input Type(s)	Output Type(s)	Explanation
IDENTITY	No	Any	Same	Does nothing- preserves the values of input rows exactly.
DIFF	No	Numeric	Numeric	1st difference- takes the difference between consecutive rows and returns one fewer than the number of input rows.
IDENTITY_ORDER_BY	Yes	Categorical, Numeric	Categorical, Numeric	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.
NORM_DIFF	No	Numeric	Numeric	Takes the difference over the rows of each of its input columns, and then computes the L_2 norms of each output row to produce a single numeric output column.

Figure 3-6: 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.

Aggregation Operation	Argument Needed	Input Type(s)	Output Type(s)	Explanation
LAST	No	Any	Same	Returns the last value.
FIRST	No	Any	Same	Returns the first value.
LAST_MINUS_FIRST	No	Numeric	Numeric	Returns the last value minus the first value.
ANY	No	Boolean	Boolean	Takes a Boolean column as input, and returns True if *any* of the values (after a cutoff point) are True.
ALL	No	Boolean	Boolean	Takes a Boolean column as input, and returns True if *all* of the values (after a cutoff point) are True.
SUM	No	Numeric/Boolean	Numeric	Power- transforms numeric inputs by taking them to the power of the argument.
COUNT	No	Any	Integer	Counts the number of values (after a cutoff point).
EXISTS	No	Any	Boolean	Returns whether any values (after a cutoff point) are present. Effectively COUNT followed by EQ(1).

Figure 3-7: 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.

Comparators	=	>	<	≤ ≥
Joins	OR: ||	AND: &&	NOT: !	
Operations on LHS	DAY_OF_WEEK	HOUR	IDENTITY	
Operations on RHS	MIN	MAX	MEAN	
Special Operations	ALL			

Figure 3-8: 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.

- We do not define cutoff points.
- We restrict the output to binary classification, i.e. True/False, single row output for each entity instance.

3.8.2 Motivation: A Case for Simplicity

Out of 54 prediction problems defined on Kaggle and converted into Trane, 51 were able to be defined using only a single column operation. The other three were able to be defined using only two column operations. These problems represent a wide swath of possible business and research applications, and so we believe it is reasonable to value simplicity as germane to the meaningfulness or usefulness of a prediction problem. In Table A.1 we present a sampling of 10 of these problems converted into Trane (See appendix A).

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.

More important is the ease with which we can build a user interface to interact with LittleTrane, and thus generate ratings for these prediction problems from humans and feed them into a machine learning system to learn which ones are meaningful. While BigTrane is a language unbounded in complexity, LittleTrane is effectively just a sequence of five operations, allowing us to convert every problem defined in it to a format that humans can understand. As we will see in chapter 4, LittleTrane still enables us to define thousands of prediction problems per dataset.

3.8.3 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 Donor's Choose Competition on Kaggle [15], which requires Boolean joining of multiple binary sub-problems.

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 features.

3.9 Examples of Prediction Problems in Trane

It helps to illustrate this syntax with some examples, along with their intuitive meanings. We do not include cutoff flags for these examples, since those flags are set more or less independently from the problem. However, note that examples 7 and 8 need access to data from before the cutoff point, so they need the "use_known" cutoff flag set to True.

3.9.1 BigTrane Examples

See A.4, in Appendix A.

3.9.2 LittleTrane Examples

Examples in LittleTrane are much easier to digest, especially since we have a script that converts them into English sentences.

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"

Entity "Store"

Operations

`DIFF` $\rightarrow$ `IDENTITY` $\rightarrow$ `LT` $\rightarrow$ `LAST` $\rightarrow$ `NOP`

2. Predict whether any of the "Bank" column values are equal to some argument, for example "JP Morgan Chase".

   Label-generating column "Bank"

   Time axis "Date"

   Entity "LoanId"

   Operations

   `EQ` $\rightarrow$ `ANY` $\rightarrow$ `NOP` $\rightarrow$ `NOP` $\rightarrow$ `NOP`

3. 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"

   Entity "TransactionId"

   Operations

   `GT` $\rightarrow$ `SUM` $\rightarrow$ `GT` $\rightarrow$ `NOP` $\rightarrow$ `NOP`

Chapter 4

Interacting with Trane

4.1 Trane Interpreter

We 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
- a dictionary of metadata associated with the dataset in the form of either a JSON or Python file

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 *black hole* point of that instance. Recall that the *black hole* point is defined for each entity instance as the latest date/time in the data before the label can be deterministically inferred. See 4-1 for a visual of how data passes through the Trane interpreter.

The interpreter was implemented in Python, using the Pandas data science toolkit [21].

4.1.1 Interpreter Input

The input to the interpreter is a JSON object or Python dictionary specifying separately the dataset configuration and the prediction problem configuration. The dataset configuration

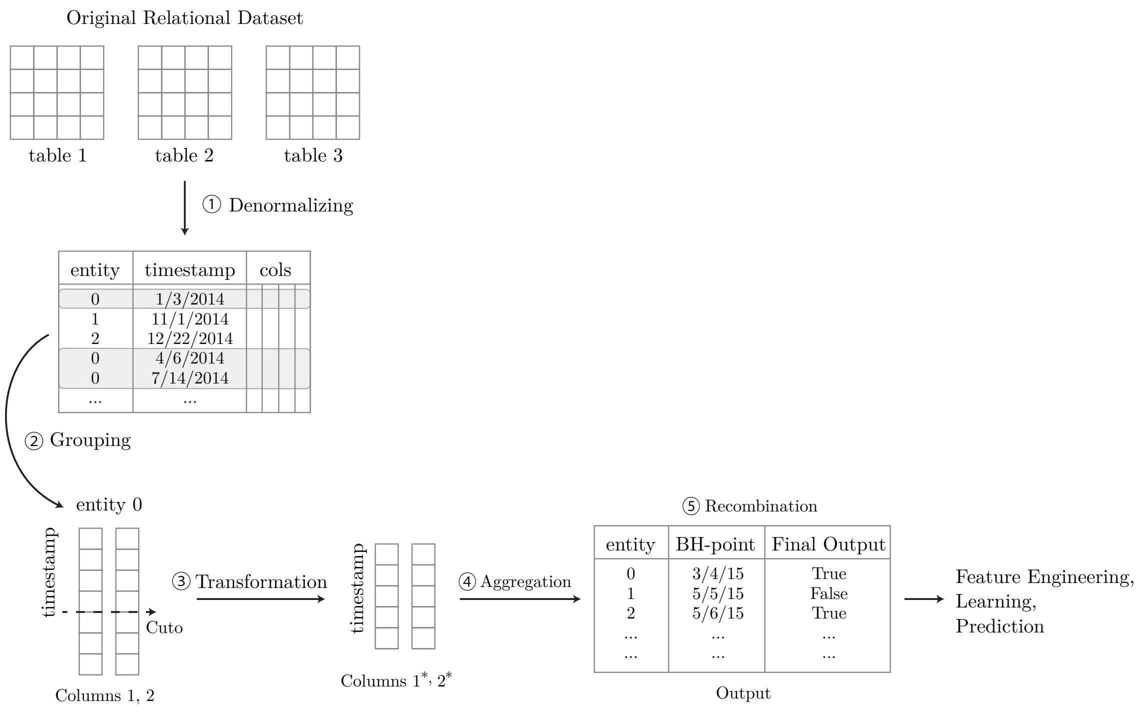


Figure 4-1: Visual representation of the Trane model. Data is first denormalized into a single table. Then it is grouped by entity, and cutoff at some time point, possibly a different point per entity. Transformation and aggregation operation operations are applied to each entity instance in order to produce labels, and BH-points are calculated for each label as well. The final output can be sent downstream to further data mining and machine learning tools.

settings are defined in Table 4-2, while the prediction problem dictionary is shown in figure 4-3.

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 Table 4-2), but we assume a single label-generating column, only problems with maximum depth two (i.e. two column operations), and we disallow logical combinations of multiple problems. This is purely for simplicity as a first pass. The next version will include these features.

4.1.2 Interpreter Output

The interpreter outputs a Pandas dataframe indexed by the entity, 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 the points specified by the flags in the input. Before calculating any labels, the interpreter finds the points in the data for each instance after which we can use for label calculation. This point includes the lead time, if defined. The internal view of the data seen by the label calculating subsystem is devoid of any points in time prior to this cutoff. Depending on the input settings, the cutoff points will be different, see 3.3.1 for more detail on how the cutoffs are defined.

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. It will first apply the filter operation at the lowest depth to discard unwanted rows, then it will apply the first row operation to each row in the data, and then the first column operation grouped by each entity 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) and the second row and column operations. Lastly, if

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 needed for determining the BH point.

Figure 4-2: Listing of interpreter settings for defining the cutoff point. Note that we do not define an end point; 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 Timedelta type is available in many scientific computing environments, and defines a duration of time.

```
{
    "entity" : [ "Dept", "Store" ],
    "time_axis": "Date",
    "original_table" : "Sales",
    "filter_op" : { "op" : "ALL" },
    "data_source" : "base",
    "col_op_type" : "AGG",
    "op_type" : "LAST",
    "args" : null,
    "cols" : [ {
        "col_name" : "Weekly_Sales",
        "col_type" : "number"
        "col_subtype" : "float",
        } ],

    "row_op" : {
        "args" : [2],
        "cols" : [ {
            "col_name" : "Weekly_Sales",
            "col_type" : "number"
            "col_subtype" : "float",
          } ],
        "op_type" : "POW" },

    "last_row_op" : {
        "args" : [100],
        "cols" : [ {
            "col_subtype" : "float",
            "col_name" : "Weekly_Sales",
            "col_type" : "number"
          } ],
        "op_type" : "LT" },
}
```

Figure 4-3: 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).

there is a final row operation (either the 3rd operation or the 5th operation), it applies that to each row in the transformed data.

Step 2.5: Generate Arguments for LittleTrane Problems

To enable direct compilation of prediction problems in LittleTrane, we additionally included functions to automatically generate a list of reasonable arguments or threshold values to use for the operations. These should be different depending on the column type, yet are somewhat difficult to intelligently guess since the types of the input to each operation depends on the output from the previous operation. We guess 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 $\pm[0, 0.5, 1, 2]*$(standard deviation) of the column
- mean of the differences of the column $\pm[0, 0.5, 1, 2]*$(standard deviation) of the column

Step 3: Calculate the BH Point

The last step is to calculate the BH points for each instance, if the problem is a classification problem. In the usual case this is just the last recorded time point minus the lead. However, for EXISTS or ANY functions, or when the problem is explicitly flagged as a step function, we need to do something more complex. We mentioned in section 3.3.3 that negative examples for EXISTS (EXISTS is the customer churn operation) are defined as those in which the amount of time between an instance's last recorded value and its end point is significantly greater the the maximum amount of time between any two of that instances recorded values. For EXISTS, we thus apply this operation to determine the negative examples and return the last recorded time points minus the lead. For the ANY operation or problem flagged as step functions, we instead return the last time point at which the pre-aggregated time series was still False.

4.2 Web App to Interact with Trane

We built a web app to directly interact with Trane through the browser. This web app accesses a database of datasets, and visualizes relevant meta-information about each one, such as the number of rows, the columns in each table, the number of uniques in each column, and an example data point from each column. It also color-codes foreign-key relationships between tables to facilitate understanding of the dataset as a whole.

Below each dataset is a series of reactive dropdown menus, consisting of suitable columns in the dataset to select for the prediction problem entity, time axis, and label-generating function.

We show each table in the dataset to the user as a table in its own right with columns on one axis, and the column name, type, subtype example data, and foreign-key reference on the other. We made a slightly counter-intuitive decision to lay out the columns vertically as opposed to horizontally, because otherwise for datasets with many columns the page would be too wide, and would require awkward horizontal scrolling. A screenshot is shown in 4-4, as well as a table describing the possible column types and subtypes in 4-5.

The web app serves three purposes.

1. It allows users to specify prediction problems
2. It allows users to annotate prediction problems
3. It allows users to run prediction problems

4.2.1 Prediction Problem Specification

Column Selection

We include dropdown menus for selecting the entity, date, and label-generating columns, which dynamically change based on which previous columns are selected. For instance, once an entity column for a particular table is selected, only date and label columns from that table are shown in the subsequent dropdowns. One can imagine a setup where problems are able to be defined across different tables, but for simplicity we restrict to a single table. Possible entity columns are those whose column type is "categorical", and subtype

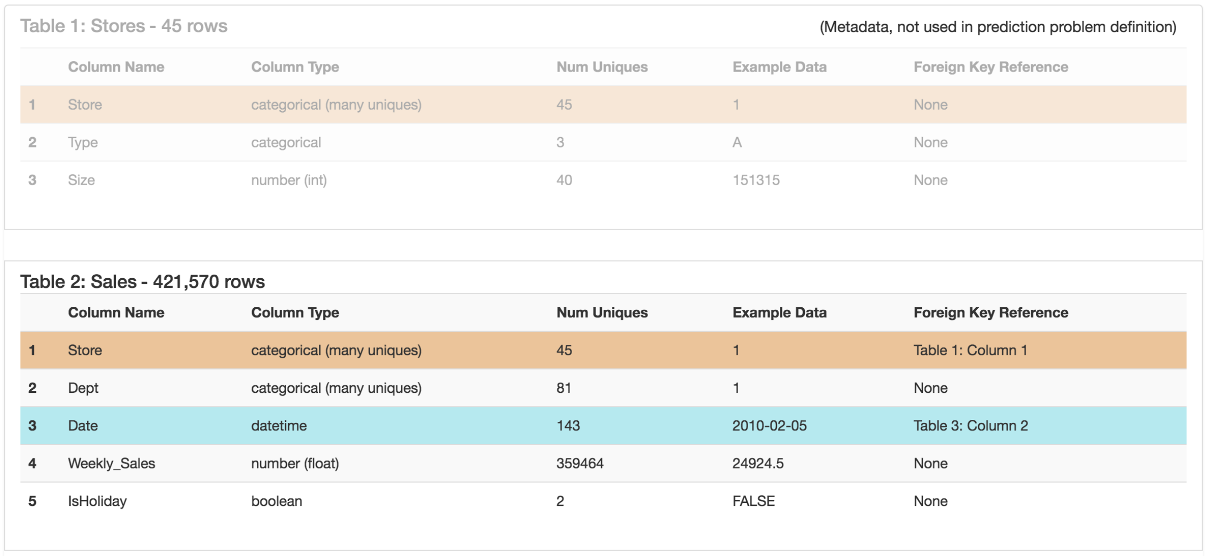


Table 1: Stores - 45 rows (Metadata, not used in prediction problem definition)

	Column Name	Column Type	Num Uniques	Example Data	Foreign Key Reference
1	Store	categorical (many uniques)	45	1	None
2	Type	categorical	3	A	None
3	Size	number (int)	40	151315	None

Table 2: Sales - 421,570 rows

	Column Name	Column Type	Num Uniques	Example Data	Foreign Key Reference
1	Store	categorical (many uniques)	45	1	Table 1: Column 1
2	Dept	categorical (many uniques)	81	1	None
3	Date	datetime	143	2010-02-05	Table 3: Column 2
4	Weekly_Sales	number (float)	359464	24924.5	None
5	IsHoliday	boolean	2	FALSE	None

Figure 4-4: Visualization of the Walmart Store Sales Dataset on the Web App. Each *column* in the actual dataset is a *row* here. Notice that the Store column in both tables is highlighted orange, indicating that there is a foreign key relationship via the Store column between the two tables. The Date column is highlighted blue because it indexes a different table, not shown. The Stores table has reduced opacity because it only contains metadata, and no time-varying columns that could be used for prediction.

Column Type	Column Subtype	Explanation
Number	Integer	
Number	Float	
Categorical	Boolean	2 uniques
Categorical	Categorical	More than 2 uniques: meant as a standard category
Categorical	Boolean	2 uniques
Categorical	Many_Uniques	Many uniques: meant as an entity or id column
Datetime	Datetime	

Figure 4-5: Listing of column types and subtypes.

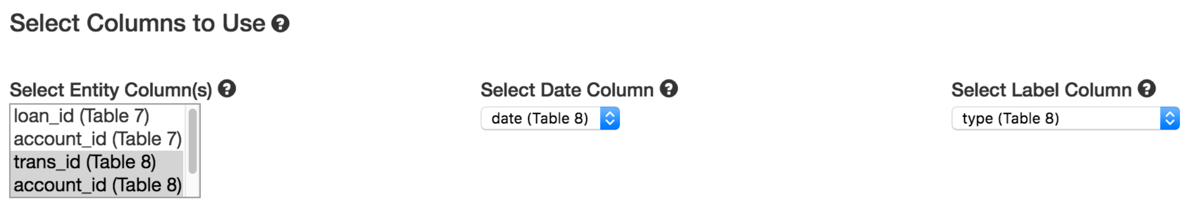


Figure 4-6: Screenshot from the web app showing the column selection dropdowns. When the user changes the entity column selection, the date and label-generating column options change to only allow columns from the same table. We allow multiple entities, but only from the same table, while date and label-generating columns are restricted to a single selection.

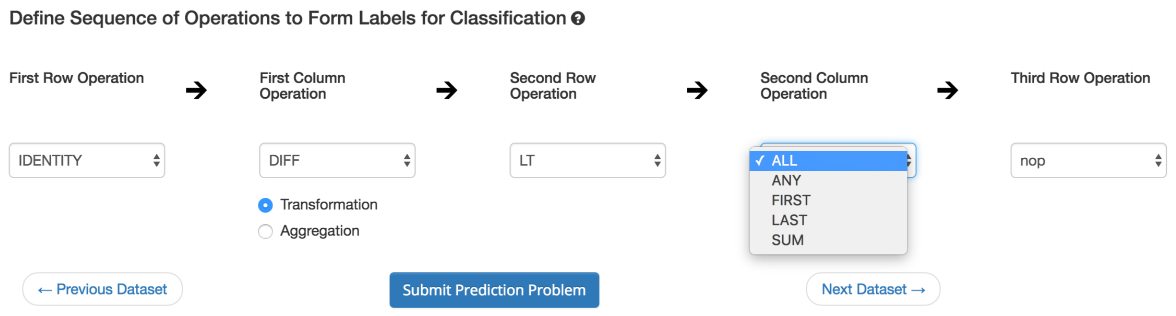


Figure 4-7: Screenshot from the web app showing the operation selection dropdowns. When the user changes an operation, all subsequent dropdowns limit their options according to the rules defined in the description of the language: 3.6.

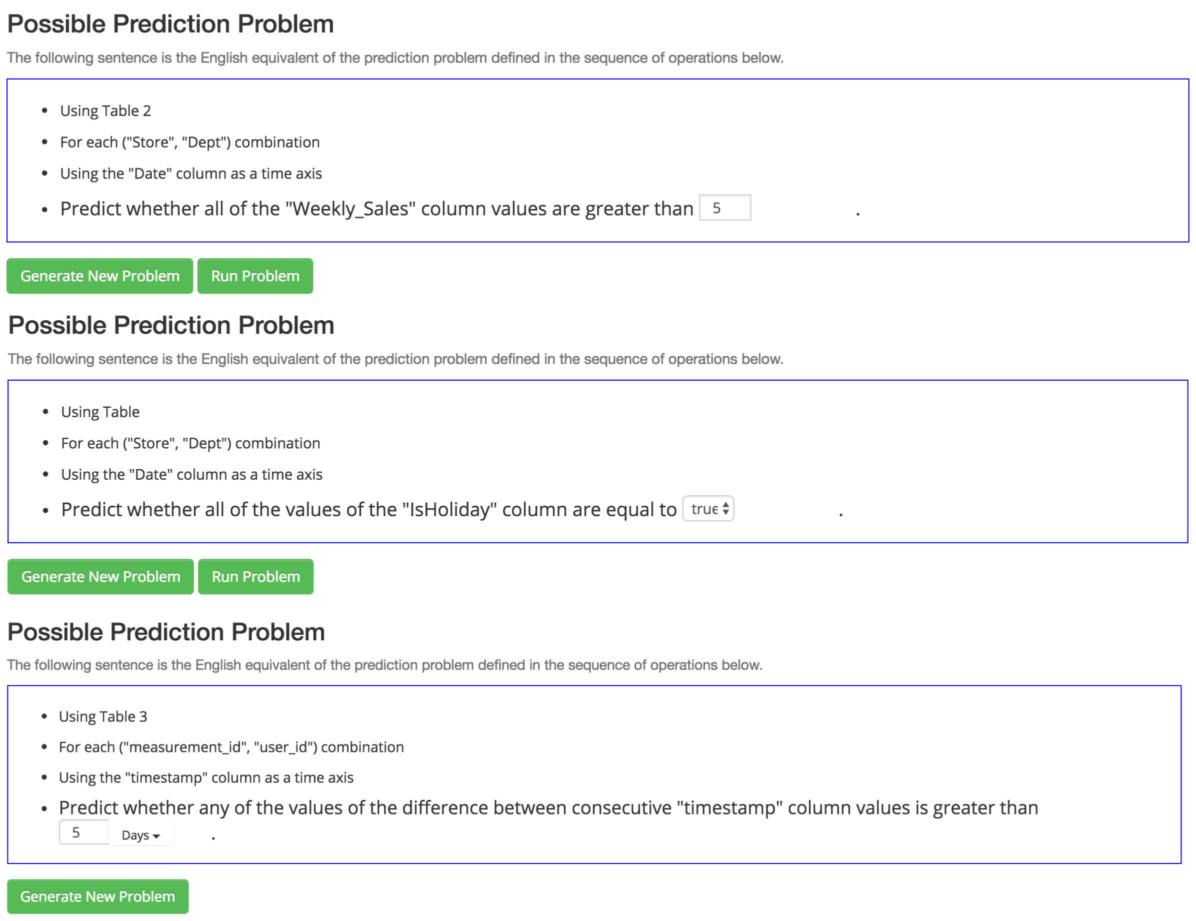
is "many_uniques". Possible date columns are those whose column type is datetime, and label-generating columns can be any column type. Several screenshots are included in 4-6.

Operation Selection

We also include dropdowns for selecting the Trane operations, which are also dynamic. As soon as a user selects an aggregation operation as the first column operation, for instance, the rest of the operations are restricted to NOP. A screenshot is displayed in 4-7.

Natural Language Description of Problems

Below the dataset we show a natural language description of the currently selected prediction problem in LittleTrane. Since LittleTrane is limited to 5 operations, we were to write a simple parsing program to convert a problem to an English sentence. These sentences went through many refinements upon showing them to people not so familiar with the project, and we believe they are at a point where even non-data-scientists can understand them with a few minutes of initial coaching. The sentences include the entity, date, and label-generating


Possible Prediction Problem

The following sentence is the English equivalent of the prediction problem defined in the sequence of operations below.

- Using Table 2
- For each ("Store", "Dept") combination
- Using the "Date" column as a time axis
- Predict whether all of the "Weekly_Sales" column values are greater than 5 .

Possible Prediction Problem

The following sentence is the English equivalent of the prediction problem defined in the sequence of operations below.

- Using Table
- For each ("Store", "Dept") combination
- Using the "Date" column as a time axis
- Predict whether all of the values of the "IsHoliday" column are equal to true .

Possible Prediction Problem

The following sentence is the English equivalent of the prediction problem defined in the sequence of operations below.

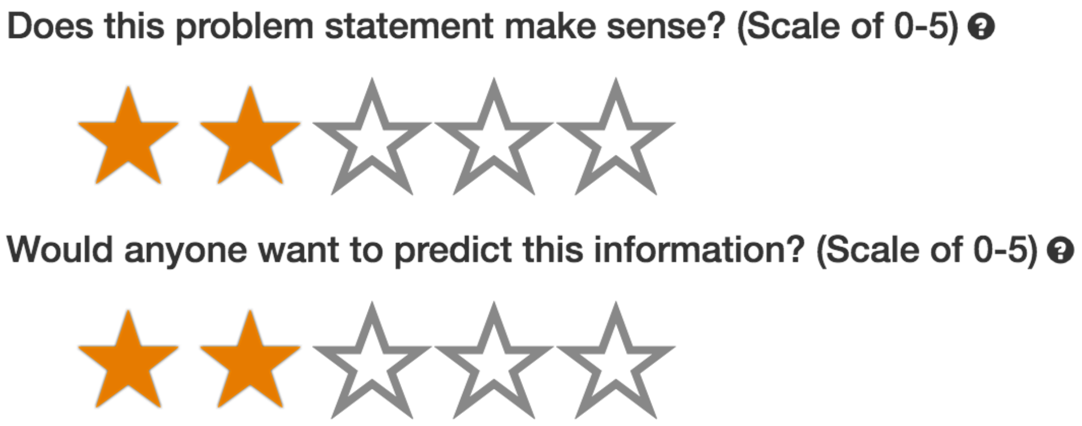
- Using Table 3
- For each ("measurement_id", "user_id") combination
- Using the "timestamp" column as a time axis
- Predict whether any of the values of the difference between consecutive "timestamp" column values is greater than 5 Days .

Figure 4-8: Three screenshots from the web app, showing three different prediction problems converted to English sentences generated randomly according to the rules described in this chapter. Notice that the web app allows users to select threshold values, respecting the types of the underlying operations and columns.

columns the problems are defined over, as well as input boxes for arguments/thresholds. Several screenshots are included in 4-8.

4.2.2 Prediction Problem Annotation

The web application allows us to collect annotations for prediction problems defined in Trane in order to actually start to learn which ones are meaningful. To do this, we removed the ability to manually select prediction problems, and replaced the dropdowns with two 0-5 star rating selectors, along with questions that explain the conditions with which we want these users to rate problems. A screenshot is shown in 4-9. We ask the following questions.


Does this problem statement make sense? (Scale of 0-5)

Would anyone want to predict this information? (Scale of 0-5)

Figure 4-9: Screenshot from the web app showing the questions posed to the user to allow them to rate prediction problems.

1. Does this problem statement make sense?

2. Would anyone want to predict this information?

In the learning system, we combine these two values for each prediction problem using a weighting scheme, where the weight given to the first question as opposed to the other is a hyperparameter.

The prediction problem shown to the user is randomly generated from the space of well-defined prediction problems, some with human-input weights over the columns so that the system selects prediction problems for certain columns more often than others (We did this to show more obviously meaningful problems more often and less obviously meaningful ones less often. These weights play no effect in determining the operations).

This website now allows us to ask non-expert humans, human who have never seen this dataset before, whether a prediction problem is meaningful or makes sense, and opens up a whole new class of annotations for much more structured and complex machine learning.

Rules for Randomly Generating Problems

Rules for defining problems in BigTrane and LittleTrane are given in 3.6. Here we define two additional rules.

- **Multiple Tables** We only allow entity, date, and label-generating columns from the same table currently. Future work will explore allowing problems defined over multiple tables.

- **Uniqueness of Entities** The selection of entities 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.

4.2.3 Prediction Problem Execution

One powerful data science tool that this web application and interpreter becomes is a way to test a large number of prediction problems end-to-end, without having to manually re-calculating the labels, or spend time thinking about all the possible problems that could be defined. To this effect, we connected the web application and interpreter to Kanter's FeatureTools software, coupled with Python Sci-Kit Learn's Random Forest implementation to automatically generate features and weighted predictive AUC for problems defined using the web application [18],[29]. We explore this in more depth in the following chapter.

Chapter 5

End to End Examples on Real Data

5.1 Experiment Definition

Using the Walmart Store Sales Forecasting dataset from Kaggle [16], 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).

Clearly many of them were ill-defined, so we have 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 guess between 5 and 20 settings of each argument. In some cases multiple operations in the problem require arguments, and so we enumerate all combinations of the possible arguments. We sample from the combination of arguments until five experiments are run successfully or we run out of arguments possibilities, which resulted in a total of 1923 experiments run successfully. The most common reason for a failed experiment is a lack of sufficient number of positive or negative examples.

For each experiment, we use Kanter's FeatureTools to generate 10 features, and run with 5-fold cross-validation using Sci-Kit Learn's Random Forest algorithm with 200 estimators. As a performance measure, we use the area under the ROC curve (AUC), weighted by the number of class labels.

5.2 Explanation of Data

This Walmart dataset describes 45 stores, and 99 departments within those stores. The dataset contains weekly recordings from each department within each store about how many goods it sold that week, in dollars, over a period of 2 years. These recordings are contained in the Sales table, which also connects to a separate Features table that contains additional weekly information recorded per-store, and to a separate Stores table that contains meta-information about each store (see 2-1). Each column is explained in more depth in 5-1, and we also detail some information affecting the generation of prediction problems below.

- **Possible Entities** Either the Store column, or a combination of Store and Department (Dept).
- **Possible Tables** If just a Store, we only allow label-generating columns from the Features table; if a Store, 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 entity is just Stores 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. Each Store, Dept combination with matching Stores contains the same information in the Features table, and so any prediction problem defined on a column in the Features table using the Store,Dept entity combination is redundant. We disallow columns from the Stores table since it only contains metadata.
- **Possible Columns** Any column in the Features or Sales table is fair game. See 5-1 for a reference.
- **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.

5.3 Results

We plot the mean of this AUC score against the standard deviation across all experiments in 5-2. There does not seem to be much of a pattern apparent here, however we encourage the

Column Name	Column Type	Table	Explanation
Store	Categorical (many uniques)	Stores, Sales, Features	Store Id
Type	Categorical	Stores	Either 1, 2, or 3
Size	Int	Stores	Size of store
Dept	Categorical (many uniques)	Sales	Department such as "Electronics"
Date	Datetime	Sales, Features	Timestamp (weekly intervals)
Weekly_Sales	Float	Sales	Sales for the department and store (dollars)
IsHoliday	Boolean	Features	Whether current date is a holiday
Temperature	Float	Features	Average temperature that week
Fuel_Price	Float	Features	Average cost of gas that week
CPI	Float	Features	Consumer price index
Unemployment	Float	Features	Unemployment rate
Markdown1-5	Float	Features	Anonymized data related to promotional markdowns that Walmart is running.

Figure 5-1: Explanation of columns in the Walmart dataset.

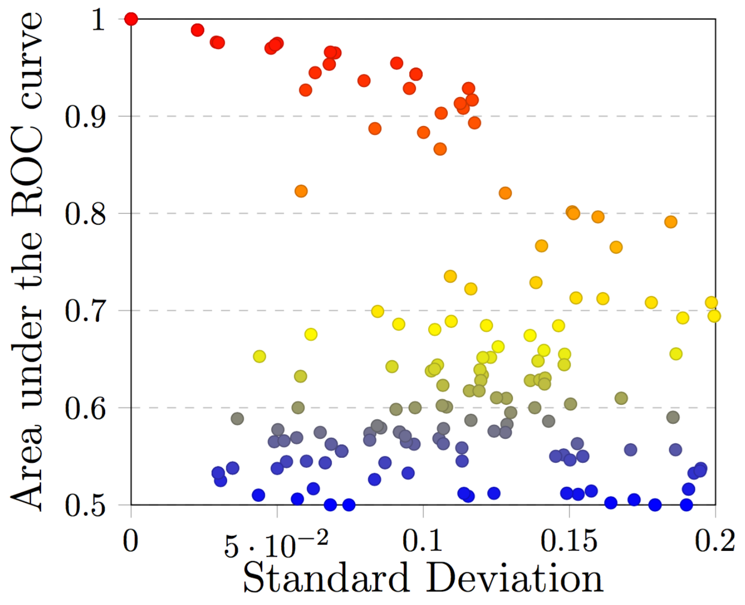


Figure 5-2: Plot of every successfully run experiment using 5-fold cross-validation with mean class-weighted AUC on the y-axis and standard deviation on the x-axis.

reader to notice the broad range of AUC scores recorded. We can draw from this result 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 deviation, and select a few examples for further investigation.

5.3.1 Sample Experiments

To pull out example experiments, 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 for the experiment, meaning 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 experiment.

4. Sample two experiments from each quantile, yielding 16 total 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. Imagine trying to predict whether number of tickets a major concert venue sells in the future is less than 5. Since this is an extremely unlikely result, the AUC score will be very high. A more meaningful setting is on the order of how many tickets the concert venue normally sells, and this would produce a harder problem (lower AUC).

Starting with Fig 5-3, we plot AUC against the various argument settings for each of these 16 problems. The most immediate conclusion is that most of the lines are horizontal! Looking carefully, we see that in many cases the AUC remains exactly the same across various argument settings. This is not an error. Recall that we are restricting ourselves to binary classification, and most of the arguments are threshold values. Changing these threshold values might produce the exact same output labels, and thus the exact same AUC (up to some small error due to the randomness of cross-validation). For an example, consider the ticket sales problem mentioned in the previous paragraph. Assume that in the data there are five nights where the venue sells under 100 tickets. Every other night it sells greater than 200 tickets. Then for threshold values between 100 and 200 of ticket sales, the number of positive and negative labels remains exactly the same, and thus the AUC should remain the same.

These 16 experiments provide with a broad picture of the types of prediction problems we can come up with, and which of those problems are actually meaningful.

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

The two problems we sampled from this region are defined as follows.

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 they are predicting does not change; it is part of the entity. For each Store we predict the future value of Store. In future versions of our Trane enumeration script we should leave these types of problems out. However, because the automated feature engineering library did not extract a feature for the value of the entity 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. Plotted in 5-3

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, at the end point is less than Y.

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. [1]

These are just overly complex problems that would be meaningful if the power operation was not present in both cases. Plotted in 5-5.

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 due to the feature engineering library not adequately generating features to predict them. Plotted in 5-7.

[1] At the time of writing this thesis, we are still investigating how to come up with better, unambiguous natural language interpretations of these complex problems. In English it is very hard to make these sentences not carry multiple meanings without including parentheticals.

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

1. For each Store, predict whether the last value of the difference between consecutive values of the Markdown2 column, to the power of X, is less than some value.

2. For each Store, predict whether the last value minus the first value of the difference between consecutive values of the Markdown4 column to the power of X, to the power of Y, is less 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. Plotted in 5-9.

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

1. For each Store, predict whether the sum of the difference of Markdown4 is 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 that happen to be easier than the ones in Bin (1,0). The second seems to be more meaningful than the first, because Weekly_Sales is semantically a column we are more likely to care about than Markdown4. Plotted in 5-11.

Bin (2,1) AUC < 0.95, High Standard Deviation

1. For each Store, predict whether any of the Markdown2 column is greater than X.

2. For each Store, predict whether the last minus first value of the difference between consecutive Markdown3 values is less than x.

These are also 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). Plotted in 5-13.

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 in this case we are able 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. Plotted in 5-15.

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 over 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. Plotted in 5-17.

Figure 5-3:
Entity (Dept,Store)
Column Store
Problem
EQ(x) → ALL → nop → nop → nop

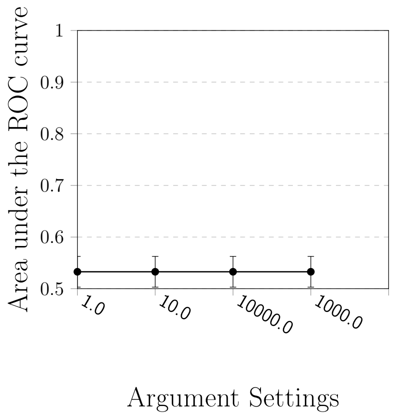


Figure 5-4:
Entity (Dept,Store)
Column Store
Problem
EQ(x) → LAST → nop → nop → nop

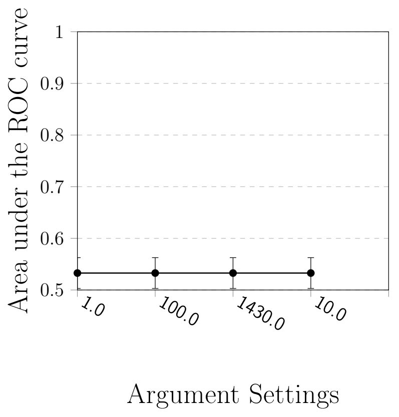


Figure 5-5:
Entity Store
Column Fuel_Price
Problem
IDENTITY → DIFF → POW(x) → LAST → LT(x)

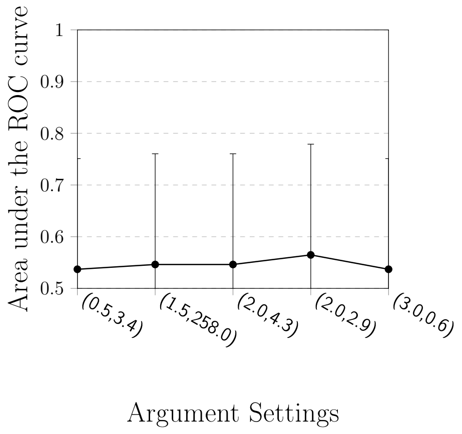


Figure 5-6:
Entity Store
Column MarkDown3
Problem
POW(x) → DIFF → POW(x) → LAST_MINUS_FIRST → LT(x)

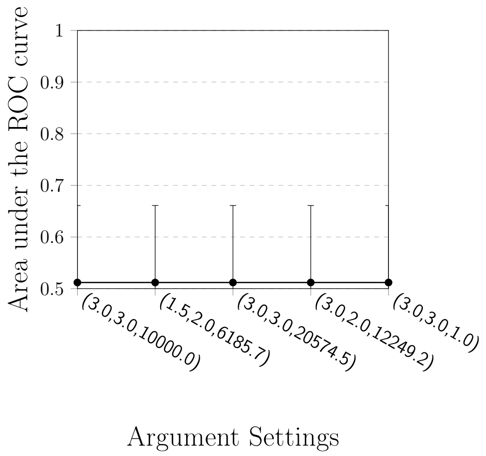


Figure 5-7:
Entity Store
Column MarkDown4
Problem
IDENTITY → LAST_MINUS_FIRST → LT(x) → nop → nop

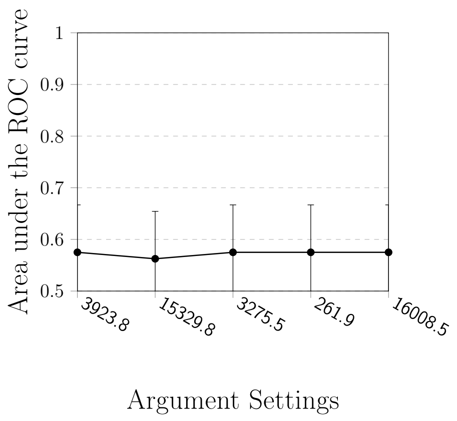


Figure 5-8:
Entity Store
Column CPI
Problem
GT(x) → LAST → nop → nop → nop

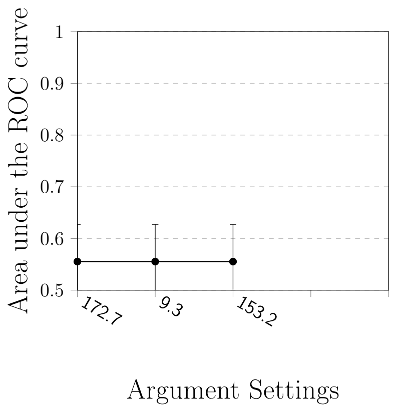


Figure 5-9:
Entity Store
Column MarkDown2
Problem
IDENTITY → DIFF → POW(x) → LAST → LT(x)

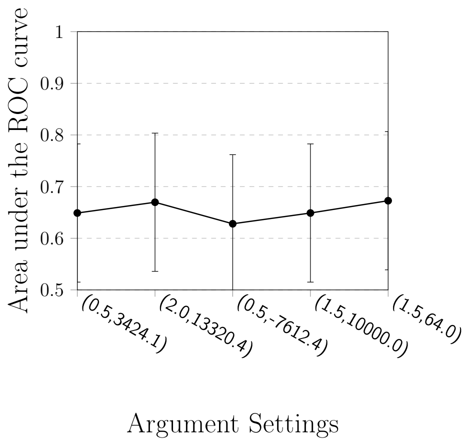


Figure 5-10:
Entity Store
Column MarkDown4
Problem
POW(x) → DIFF → POW(x) → LAST_MINUS_FIRST → LT(x)

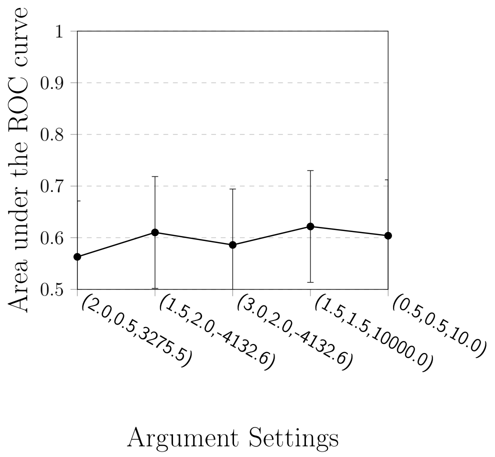


Figure 5-11:
Entity Store
Column MarkDown4
Problem
IDENTITY → DIFF → IDENTITY → SUM → GT(x)

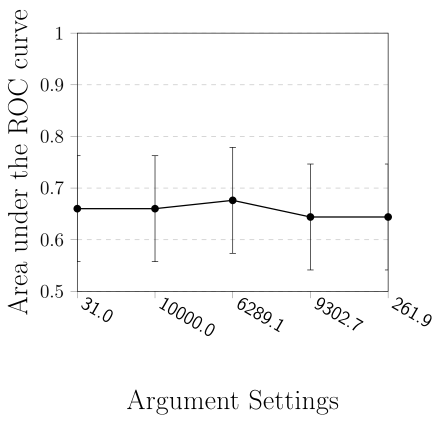


Figure 5-12:
Entity (Dept,Store)
Column Weekly_Sales
Problem
LT(x) → ALL → nop → nop → nop

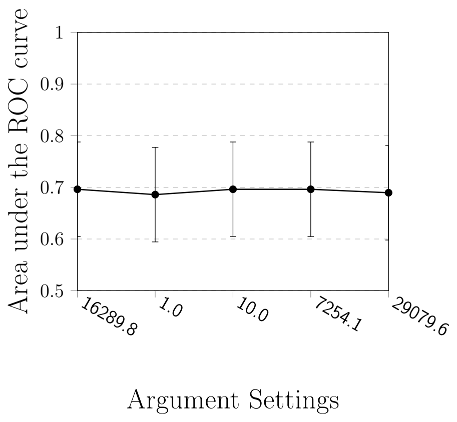


Figure 5-13:
Entity Store
Column MarkDown2
Problem
GT(x) → ANY → nop → nop → nop

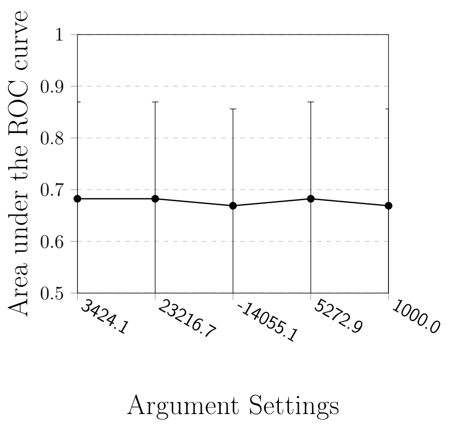


Figure 5-14:
Entity Store
Column MarkDown3
Problem
IDENTITY → DIFF → IDENTITY → LAST_MINUS_FIRST → LT(x)

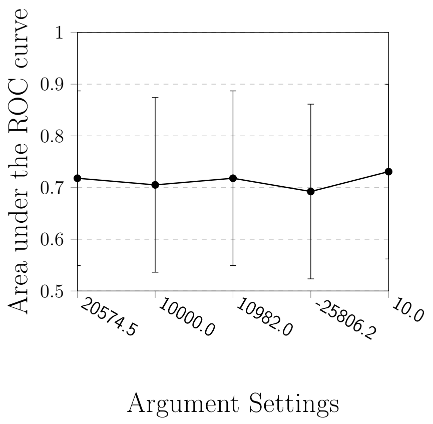


Figure 5-15:
Entity Store
Column Temperature
Problem
LT(x) → SUM → GT(x) → nop → nop

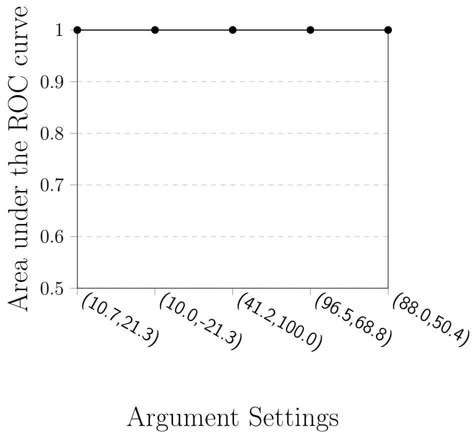


Figure 5-16:
Entity Store
Column Store
Problem
NEQ(x) → ALL → nop → nop → nop

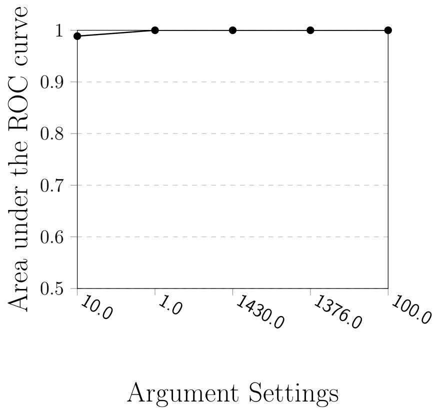


Figure 5-17:
Entity Store
Column MarkDown3
Problem
IDENTITY → SUM → GT(x) → nop → nop

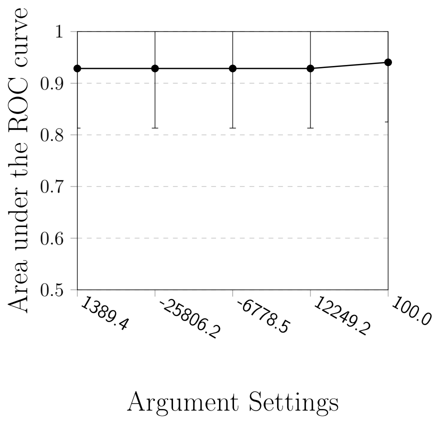
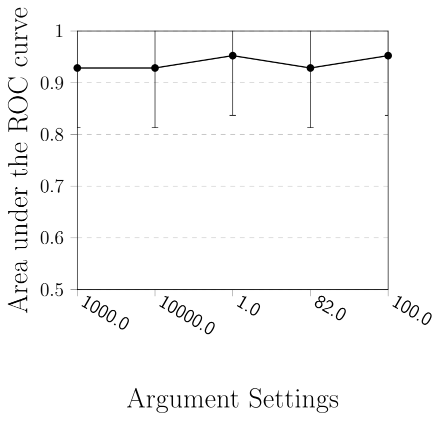


Figure 5-18:
Entity Store
Column MarkDown3
Problem
IDENTITY → SUM → LT(x) → nop → nop



Chapter 6

Pernican: A System That Learns What to Ask

We have described a language that formalizes the prediction problem space, an interpreter that can take problems defined in that space and map them into a machine learning system's inputs, and a user interface that allows the easy creation, visualization, and annotation of prediction problems. The next logical question to ask then, is can we build a system that navigates through this prediction problem space like a human does. In other words, can we automatically learn which prediction problems make sense (and perhaps useful) and which are meaningless. Such a system would have to learn across different data sets, transferring knowledge about which prediction problems are meaningful from one to another. In this chapter, we describe the challenges in building a learning system, discuss possible ways of implementing such a learning system, and describe in detail one particular version that we designed, implemented and tested.

6.1 On the Name

We named our learning system Pernican, after Nicolaus Copernicus, who questioned whether the Earth actually revolved around the Sun and kickstarted the Scientific revolution. While the likelihood this system ever generates a question as revolutionary as the one Copernicus did is vanishingly small, the goal of the system is to learn what are good questions to ask, and so conjures up the image of a scientist coming up with new and interesting hypotheses.

6.2 Large search space and complexities of learning

BigTrane is a highly expressive language, where the number of possible predictive problems that can be defined scales exponentially with the number of available *operations*, the *depth* of the sequence of operations, the *number of columns* in the dataset, and how many columns are allowed to be used together for a single prediction problem. Even LittleTrane, which only consists of a sequence of at most 5 *operations* applied to a selection of one or more entities, a single time axis, and a single *label-generating column* can enumerate thousands of legal problems for a single dataset. Expanding that space just a hair to two possible label-generating columns results in hundreds of thousands of problems.

If we try to learn some subset of BigTrane that is larger than LittleTrane, we can apply some techniques from structured prediction to prune the search space heuristically as we learn or predict. One method we considered is a probabilistic graphical model, where nodes in the graph represent entity/time/label-generating column decisions, operation decisions, or filtering decisions. We encounter a number of problems when designing a learning approach:

- **Datasets are vastly different**: A key problem, irrespective of how we constrain the language, is that the search space changes for each dataset: datasets contain varying numbers of columns and types of columns. This results in varying number of prediction problems and choices.

- **Prediction problems that are valid in one dataset may not be in another.** For example, a prediction problem may be valid in one dataset because it is defined for a specific column type which may not exist in the other dataset.

- **Columns have different semantic meaning**: Furthermore, since meaningfulness of problems is tied semantically to particular dataset columns, there is actually no direct mapping from one dataset to another. The same syntactic problem of predicting the future value of a real-valued column might have very different meanings depending on the semantic meaning of the column- in one dataset it might be applied to bread prices in the US, in another to the speed of a motor vehicle at the time of a collision.

6.3 The Pernican Learning System

To build a learning system, that can automatically generate meaningful prediction problems, we decided to take "recommender system" like approach. The key idea is to be able to recommend prediction problems for a new dataset based on human annotated ratings for prediction problems on a previous set of datasets. To set up the problem, we need the following key ingredients:

- **A set of datasets**
- **A fixed/standard set of prediction problem definitions that are valid across the datasets**
- **Human ratings for the standard prediction problems defined over different datasets**

If we achieve the three, we can build a matrix, and associate each prediction problem with an index into the columns of a matrix, and each dataset with an index into the rows of a matrix. The value of this matrix at index (i, j) is the human-generated rating for prediction problem i on dataset j. With a matrix of this form, the learning problem reduces to that of matrix completion: given a partially filled matrix of ratings, can we infer the values of all the other elements in the matrix? We can thus think of the setup as a recommender system, where we want to recommend prediction problems to a dataset. This setup is typically solved by assuming that the matrix is low-rank, which means in our case that there are a few different archetypal datasets and archetypal prediction problems, and each realized dataset/prediction problem is a linear combination of these archetypes. While directly optimizing for low-rank matrices is known to be NP-Complete, optimizing for low values of various matrix norms (e.g. trace norm or max norm) has been show to be both efficient and a good substitute [12].

In the next section, we describe in detail how we establish these three ingredients and thus the matrix. Following that, we explain the learning approach.

6.4 Setting up the matrix

Datasets: We have 7 datasets. Their details are given in Table 6-1.

6.4.1 Standardizing the set of prediction problems.

The large search space motivated our decision to stick to LittleTrane for learning. In LittleTrane, we can actually enumerate every possible prediction problem for every possible column type. resulting in a total of 407 prediction problems. While we can come up with a fixed set of prediction problems given possible column types, we still end up with different number of prediction problems for different datasets because they have different number of columns. Additionally, the columns, while they match in their type, they differ semantically as we pointed out before. To address this we first cluster the column names by processing them via natural language.

Clustering Column Names for Semantic Features As we pointed out, we still have to deal with the fact that there is no direct mapping between prediction problems defined over different columns in different datasets. A prediction problem is just a sequence of operations applied to a column in the data- how can we say that two problems are equivalently meaningful if the underlying columns have different real-world interpretations?

As alluded to before, if we cluster the column names into a fixed number of semantic clusters, then instead of defining a prediction problem over the column itself, we define it over the cluster the column belongs to.

To make our clusters more meaningful, we actually generate the clusters using words from approximately 582 relational datasets downloaded from various public data hubs on the web [2], [25], [27], [1], [26], and [11]. More specifically,

- We break up column names into individual words using common patterns such as spaces, underscores, and camel-casing and apply Google Word2Vec to each subword to generate a set of high-dimensional vectors [22]. We apply the same procedure to each word in each dataset described above (i.e. not just column names but also the names of the dataset and any description provided)

- We cluster these vectors using Mini-Batch KMeans algorithm provided by Sci-Kit Learn into 15 clusters [29]. Furthermore, to increase the size of each cluster so that words from new datasets are more likely to be present within some cluster, we augment each cluster with related nouns in the Google News Word2Vec space. We define related nouns as the 100 most similar words using cosine similarity of the vectors which are parsed as nouns using the WordNet annotated corpus [22], [38].

- There is not a simple way to select the appropriate cluster for the column from the multiple individual clusters each of its subwords belong to. Currently we just select the cluster with the greatest number of subwords belonging to it, breaking ties arbitrarily with the last subword in the column. One could think of more intelligent ways of selecting the cluster based on distinctness of the cluster, NLP measures like TF-IDF (Term Frequency-Inverse Document Frequency) [42].

Top 10 words in terms of L_2 distance to the cluster center are shown for each cluster in 6.1

Resulting fixed dimension matrix: Now we have prediction problems defined for each column type and enumerated for each cluster. There are 407 prediction problems times 15 clusters, so 6105 problems in all. We assign each an index. We also assign each dataset an index- we currently have annotations for 7 datasets. We can thus define a $7X6105$ matrix with which to train a collaborative filtering algorithm. Additional features about each dataset and the operations in each prediction problem are provided to the algorithm as well.

6.4.2 Acquiring human ratings

We can generate numerous prediction problem definitions from LittleTrane that do not make any sense or have any real-world significance. For example, our algorithm that randomly walks through the space of prediction problems generated a problem that reads like the following:

Table 6.1: Top 10 words by distance to cluster center for each cluster.

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4
zip	total	tchr	the	address
go	number	ball	in	email
get	amount	div1	for	phone
start	proportion	horseshoe	last	telephone
turn	nearly	lapa	this	service
pull	approximately	raquetball	after	fax
do	percentage	couverture	before	information
see	million	score	only	notification
back	five	x9	on	services
run	cumulative	indoor	from	respond

Cluster 5	Cluster 6	Cluster 7	Cluster 8	Cluster 9
ds	metrics	name	education	type
addr	data	date	health	any
adrian	sector	location	welfare	sort
albert	growth	names	children	not
ut	indicators	surname	schools	do
ple	systems	dates	teachers	types
andrew	sectors	description	child	or
ans	market	timestamp	population	specific
sd	infrastructure	locale	literacy	what
howard	markets	forename	mothers	if

Cluster 10	Cluster 11	Cluster 12	Cluster 13	Cluster 14
year	city	leur	average	approved
month	area	le	rate	requested
week	district	en	percentage	proposed
days	town	autres	percent	allocated
day	municipality	les	per	request
time	county	autre	ratio	budgeted
months	neighborhoods	demande	rates	grant
weeks	municipal	nouvelles	higher	requesting
years	community	niveau	lower	resubmitted
summer	residents	ayant	level	required

Predict whether the first date recorded (after a cutoff point) is before some threshold date.

In this case, every date was recorded at regular, one-week intervals, so this problem should be able to be solved with 100% accuracy. However, the problem is so simple that it is meaningless. In general, our human intuition tells us that meaningful problems are hard problems, but not so hard as to be impossible. We know of no good way to incorporate this nuance into a rating scheme for each problem besides asking the experts: real, live humans.

Therefore, a significant chunk of the time and effort expended into this project was focused on building an interactive application that allows humans, and ideally not just data science experts, to easily annotate prediction problems with ratings. This system was described in chapter four, and used to generate 84 unique ratings, and 157 total ratings, where multiple users were allowed to rate the same problem. In the future we plan on collecting many more ratings to build a more effective learning system.

Challenges in acquiring human input: A key issue at stake here is making sure to capture the right quantity. Computer vision researchers and natural language processing (NLP) researchers also turn to human intelligence to generate labels. They typically can make use of services like Amazon Mechanical Turk to show images or sentences that they want annotated, and can ask almost any human whether an image contains an certain object, or a sentence contains a noun [32], [36]. Here we are asking a more complex question: meaningfulness of predictive problem generated *via* a complex sequence of operations applied to an organized collection of data points.

For humans to make this judgement call, they need to understand both the dataset and problem semantically and syntactically, and to do this they need access to certain features of the dataset, like the names of the columns, the types of the columns and how they are organized. To automate this process, we most likely need to provide this same information in addition to the particular dataset and prediction problem to a learning machine.

6.5 Learning from the matrix

Encoding Side-Information into the Matrix: We would additionally like to encode side information about both the *datasets* and the *prediction* problems.

For the datasets, an obvious choice is the clusters that the words in each column name belong to, as well as the other words present in the dataset (such as the name of the dataset and a provided description). We encode the number of words belonging to each of the 15 clusters as features, as well as the following:

1. Number of Tables
2. Average Number of Columns per Table
3. Maximum Number of Columns per Table
4. Total Number of Columns
5. Average Number of Rows per Table
6. Maximum Number of Rows per Table
7. Average Number of Unique Values per Column
8. Maximum Number of Unique Values per Column
9. Number of Each Type of Column (types are integer, float, datetime, boolean, categorical, categorical with many uniques functioning as an id)

For the prediction problems, we would like to encode information about the individual operations used. We can simply binarize the type of operation and position within the prediction problem. Recall that each problem consists of 5 operations (where some of them may be no-ops). We define features as follows

$$f_{ij} = \begin{cases} 1 & p_j = o_i \\ 0 & \text{otherwise} \end{cases}$$

Where $p_j \in [p_1, p_2, p_3, p_4, p_5]$ is the sequence of operations in the prediction problem, and o_i is the i^{th} enumerated row or column operation: row if i is even, column in i is odd. o_1, o_3, o_5 is thus one of

$$\{\texttt{LT}, \texttt{GT}, \texttt{EQ}, \texttt{NEQ}, \texttt{POW}\}$$

, and o_2, o_4 is one of

$$\{\texttt{IDENTITY}, \texttt{DIFF}, \texttt{LAST}, \texttt{FIRST}, \texttt{LAST\_MINUS\_FIRST}, \texttt{ANY}, \texttt{ALL}, \texttt{SUM}, \texttt{COUNT}, \texttt{EXISTS}\}$$

.

Feature-based Collaborative Filtering: Normal collaborative filtering only uses values of the matrix (in our case, the dataset/prediction problem matrix) to train a model and make predictions. It takes advantage of implicit features defined according to some the singular values of the matrix, which is assumed to be low-rank. In our case, taking advantage of additional features is both advantageous and necessary in order to make predictions for a completely new dataset with no human-annotated ratings defined. There exist methods in the literature for adding in additional features as an additional term to the loss-optimization function [31]. These typically take the form of

$$\min_{W,H} \frac{1}{2} \| P_\Omega \left(Y - WH^T \right) \|^2 + \frac{1}{2} \left\{ \operatorname{tr} \left(W^T L_w W \right) + \operatorname{tr} \left(H^T L_h H \right) \right\}$$

where $\|\cdot\|$ is a matrix norm, such as the trace norm or max norm, P_Ω is a projection operator that retains the observed elements, W, H are non-symmetric matrices that represent factorizations of the dataset/prediction problem matrix, and L_w, L_h are matrices that encode both regularization terms as well as side information in the form of graph Laplacians. See [31] for more details.

[8] implemented a version of this algorithm in C++ using a projected gradient descent method that allows for the easy incorporation of side information into the input feature file. We use their freely-available library to train Pernican and make predictions.

6.5.1 Learning System Workflow

The standard way of training and testing matrix completion problems involves leaving out a random portion of the matrix during the training phase, and testing the learning system's predictions against the held out data. In our case, since we care about the *most* meaningful prediction problems on new, unseen datasets rather than on specific ratings for arbitrary prediction problems on arbitrary datasets, we define a slightly different procedure as follows.

1. For each row i and column j in the matrix
   (a) Combine the logical ratings and semantic ratings from each user according to the linear weighting scheme $c = \alpha * l + (1 - \alpha) * s$, where $\alpha \in [0, 1]$, l is the logical rating, s is the semantic rating
   (b) take the median of all the combined ratings c collected from multiple users
2. Train the machine with some portion of entire datasets left out.
3. Generate ratings for every possible prediction problem for each of these datasets.
4. Sort these ratings highest-to-lowest
5. Return the highest-rated prediction problem that is defined for the dataset in question.

Step four is necessary because not every prediction problem is allowed for each dataset. Recall that prediction problems are associated with a column type (e.g. integer, datetime, categorical), and are enumerated for each semantic column cluster. A prediction problem is legal for a particular dataset if there is a column in the dataset with the same type and whose name belongs to the same cluster (more correctly, we break column names up into possibly several different words, and one of those words belongs to the same cluster as the one the prediction problem is defined over).

To test the system, we could compare the highest-rated prediction problem computed by the collaborative filtering algorithm against the highest-rated problem annotated by humans. We could also compare the whole set of legal prediction problems defined for that dataset against all of the human ratings. However, a more natural, on-line way of testing the system is to actually interact directly with the user as follows.

1. Train the machine using all of the currently annotated prediction problems
2. Generate ratings for every other legal prediction problem for the current dataset
3. Present the user with a randomly sampled prediction problem from this set of legal problems, weighted by the model's current beliefs about each problem's rating.

4. Allow the user to rate this problem

5. Incorporating this rating into the next round

Note that step three is noisy, so that the model can potentially be "surprised" either by a problem it believes has a high rating yet the user provides a low one, or by the opposite. Moreover, we can show new users ratings we already have for problems, to bolster our confidence in that rating, or even ask users to tweak problems such that the newer version is more meaningful, similar to what was done in [34]. As a future step we plan on implementing this system using methods from the online machine learning literature such as [13]. Additional theoretical work is necessary before we can directly use such approaches, as there currently exists no good way to incorporate the side information about each dataset and prediction problem into the online setting.

6.6 Results

We collected ratings from users on the seven datasets lists in Table 6-1, and trained Pernican separately using each one as a held-out dataset. The top prediction problems recommended by Pernican are shown in Appendix A. We also were able to run 55 prediction problems recommended for the Walmart dataset. We ran each with 5 choices of arguments, with arguments determined in the same way as Chapter 5, and plot the order Pernican selected these problems by their worst AUCs (and so the hardest problems) out of the 5 argument settings in 6-2. This plot is basically random, and suggests either that we do not have enough data to come to meaningful conclusions, or that predictive accuracy is not at all tied to meaningfulness.

6.6.1 Top Problems for Each Dataset

In this section we show the top rated legal problems by Pernican for each dataset, where in each case the dataset in question was left out of training. From this small initial experiment, we conclude the following, knowing that our confidence in these results is weighed by the small sample size.

Dataset Name	Description	Number of Ratings Collected
Walmart	Information including weekly sales within various Walmart stores and departments within them during different times of the year [16]	71
Czech Debit Card Company: Gas Station Transactions	Transactional data from Czech debit card company specialising on payments at petrol pumps, taken from the taken from the Relational Dataset Repository.[23]	36
Czech Bank	Artificial data from a Czech bank, taken from the Relational Dataset Repository.[23]	26
Loan Applications	606 successful and 76 unsuccessful loans along with their information and transactions from PKDD'99, taken from the Relational Dataset Repository.[23]	2
Safecast	Data from radiation measurement devices[17]	16
Shout!	Social network data from an MIT Media Lab project allowing users to trade retweets [20]	2
Social Evolution	Social network evolution and meme diffusion in a student dorm, tracks the lives of students within an MIT dorm over the course of a year through surveys and cell phone data [28]	4

Figure 6-1: Descriptions of datasets used from collection annotations and making predictions

- Simple problems are rated highly
- Problems that include a nesting (i.e. a second column operation), a POW operation, or a DIFF operation are rated very low
- Intuitively meaningful problems are near the top, such as, for the same PKDD Loan application dataset, "predict whether the any of transaction amounts are less than some threshold" (LT $\rightarrow$ ANY).
- Nonsense solutions very low, such as "last value of the difference in the number of measurements to the power of some number is less than some threshold" (POW $\rightarrow$ DIFF $\rightarrow$ LT $\rightarrow$ LAST).

These conclusions are very intuitive, and confirm our initial suspicions that simple problems would be more meaningful. One exciting conclusion is that the 5th highest rated problem in the Walmart dataset is a version of the problem presented in the actual Kaggle competition converted into the binary classification setting using a threshold value. The problem as presented on Kaggle is to predict future values of the "Weekly_Sales" column in a regression-type setting, and Pernican's output is to predict whether any future values of the "Weekly_Sales" column are greater than some threshold.

The data is organized in Appendix A as follows.

- Tables A.8 and A.9 show the top and bottom 10 rated problems by Pernican on the Czech bank dataset.
- Tables A.10 and A.11 show the top and bottom 10 rated problems by Pernican on the Gas Station Transactions dataset.
- Tables A.12 and A.13 show the top and bottom 10 rated problems by Pernican on the PKDD Loan Application dataset.
- Tables A.14 and A.15 show the top and bottom 10 rated problems by Pernican on the Safecast dataset.
- Tables A.16 and A.17 show the top and bottom 10 rated problems by Pernican on the Shout dataset.

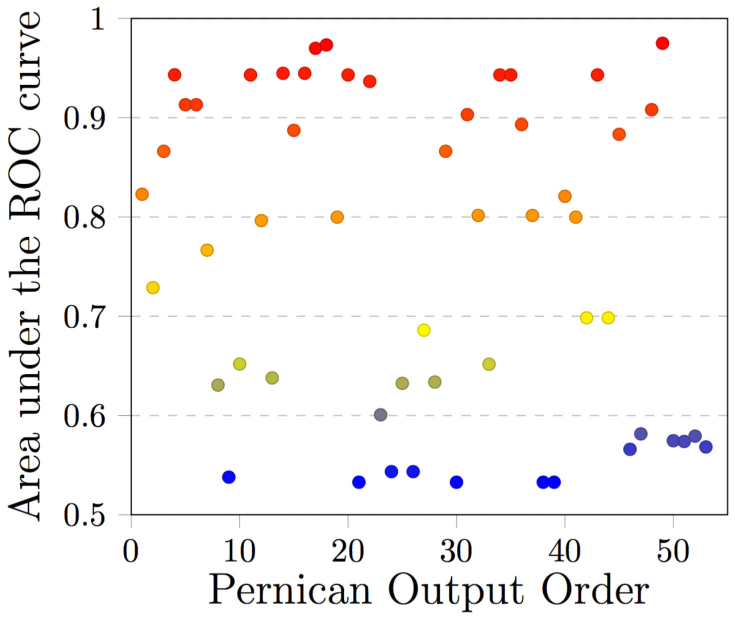


Figure 6-2: Plot comparing the output order of prediction problems recommended by Pernican with their AUC, a measure of predictive difficulty.

- Tables A.18 and A.19 show the top and bottom 10 rated problems by Pernican on the Walmart dataset.
- Tables A.20 and A.21 show the top and bottom 10 rated problems by Pernican on the Social Evolution dataset.

6.6.2 AUC Scores of Top Walmart Problems

As shown in chapter 5, we have already run all the possible Walmart problems using a Random Forest from Sci-Kit Learn and Feature Tools. We can thus compare the AUC's these problems achieved against the order in which Pernican rated the problems on a meaningfulness scale. The results are shown in 6-2. There is no discernible pattern suggested by the graph, and it seems as though predictive accuracy and meaningfulness are entirely independent axes. Future work with a larger sample size fed into Pernican and more variety of compared datasets will confirm whether this hypothesis in indeed accurate.

Chapter 7

Discussion and Future Work

In this chapter we discuss our contributions at a high level, and present future research directions.

7.1 Discussion of Results

In this thesis we took automatic data science one step further backward. By creating a way to define problems automatically, that can then be searched by an intelligent machine to build up a knowledge base of good *questions* to ask, we venture a tiny bit farther towards the eventual goal of understanding human intuition. We hope that in future iterations of this work we can refine the language enough, gather enough data, and make the underlying implementation run smoothly enough so that people who may not be strictly-speaking data scientists can make good use of it.

7.2 Future Work

As alluded to in several sections previously, there are many avenues of future expansions and improvements to consider. We detail some of them here.

7.2.1 Large-scale Data Collection and Experiments

The biggest potential for research in the near future involves scaling up the amount of human annotations we collect so that we can run Pernican with higher volumes of data. We believe there is much to be learned about the types of questions we as humans think are meaningful, and we have only scratched the surface.

User-submitted Datasets

Besides just scaling up the number of ratings collected, we would also like to increase the number of datasets we ascribe prediction problems to. One way to do this is to enable users to submit their own datasets in the form of structured metadata.

7.2.2 Online-Learning Methods and Algorithms Applied to Pernican

Pernican lends itself naturally to an online approach, and we would like to implement the iterative methods discussed in 6.5.1.

7.2.3 Expanding to Regression and Recommender Systems

Trane applies equally well to regression and recommender systems, and we would like to expand the web app and the Pernican learning system to include these types of problems.

7.2.4 Learning BigTrane

Eventually we would like to learn prediction problems that are completely defined and runnable, which would include cutoff flags and arguments. We would also like to expand the set of currently defined operations, as well as to allow filters in the learning stages.

7.2.5 Applications to Non-relational Datasets

We only define prediction problems on relational datasets, and we only implemented the Trane interpreter for single, denormalized tables, but we could potentially expand to multi-table relational datasets, as well as datasets defined other ways. For instance, we could incorporate graph data, or some carefully defined NoSQL formats.

MetaTrane: Turning Pernican's Gaze Inward

One dataset that would be fun to include is our own dataset describing the ways users interact with the web app, and the ratings they give to prediction problems. If we convert this dataset into a relational format, we could try to learn prediction problems about our own users.

Appendix A

Tables

Table A.1: Table showing prediction problems from various Kaggle competitions translated into Trane. Note that we simplify the syntax here for ease of display. Operations are simply instantiating by their name, entities and time axes are left out. The first five can be translated into LittleTrane. The next four cannot because of the filter operation, and the last one cannot because it contains two input columns.

1.

Description
Price of rental properties
URL
https://www.kaggle.com/c/deloitte-western-australia-rental-prices
Problem

```
LAST(cols=["ren_base_rent"],
    filter= "ALL",
    row_op = IDENTITY(
        cols=["ren_base_rent"]))
```

2.

Description
Customer churn
URL
https://www.kaggle.com/c/customer-retention
Problem

```
EXISTS(cols="*",
     filter= "ALL",
     row_op = IDENTITY(
         cols="*"))
```

3.

Description
Future prescription volume
URL
https://www.kaggle.com/c/RxVolumePrediction
Problem

```
LAST(cols=["PRESC_VOL"],
     filter= "ALL",
     row_op = IDENTITY(
        cols=["PRESC_VOL"]))
```

Table A.2: Kaggle problems in trane 4-7

4. **Description**

   City bikeshare system demand/use

   URL

   https://www.kaggle.com/c/bike-sharing-demand

   Problem

```
IDENTITY(cols=["count"],
        filter = "ALL",
        row_op= IDENTITY(
            cols=["count"]))
```

5. **Description**

   Short term movements in stock prices

   URL

   https://www.kaggle.com/c/informs2010

   Problem

```
IDENTITY(cols=["price"],
        filter="ALL",
        row_op = IDENTITY(
        cols=["price"]))
```

6. **Description**

   When flu virus hits

   URL

   https://www.kaggle.com/c/genentech-flu-forecasting

   Problem

```
LAST(cols=["date"],
    filter = "people_infected == MAX(people_infected)",
    row_op = IDENTITY(cols=["people\_infected"]))
```

7. **Description**

   Where flu virus hits

   URL

   https://www.kaggle.com/c/genentech-flu-forecasting

   Problem

```
LAST(cols=["location"],
    filter = "people_infected == MAX(people_infected)",
    row_op = IDENTITY(cols=["people\_infected"]))
```

Table A.3: Kaggle problems in trane 8-10

8. **Description**

How strong flu virus will be

URL

https://www.kaggle.com/c/genentech-flu-forecasting

Problem

```
LAST(cols=["people_infected"],
    filter= "people_infected == MAX(people_infected)",
    row_op = IDENTITY(
        cols=["people_infected"]))
```

9. **Description**

Risk of customer credit default

URL

https://www.kaggle.com/c/risky-business

Problem

```
ANY(cols=["loan_defaulted"],
    filter= "is_loan == True",
    row_op= EQ(
        cols=["loan_defaulted"],
        arg=1))
```

10. **Description**

Which songs a user will listen to

URL

https://www.kaggle.com/c/msdchallenge **Problem**

```
IDENTITY_ORDER_BY(cols = ["song", "play_count", args=[-1],
                  filter= "ALL",
                  row_op = IDENTITY(
                      cols=["song","play_count"]))
```

Table A.4: BigTrane Examples

1. For each type of laptop, predict whether the future sales price will ever be below 1000. Assume the time axis column is labeled "Month".

```
AggOp(
    entity = "laptop",
    time_axis = "month",
    data_source="base"
    optype="ANY",
    RowOp=(
        type="LT",
        cols=["price"],
        args=[10000]
    ),
    cols=["price"],
    FilterOp="ALL"
)
```

2. For each type of laptop, predict whether all future sales prices will be below 1000.

```
    AggOp(
    entity = "laptop",
    time_axis = "month",
    data_source="base",
    optype="ALL",
    RowOp=(
        type="LT",
        cols=["price"],
        args=[1000]
    ),
    cols=["price"],
    FilterOp="ALL"
)
```

Table A.5: BigTrane Examples 3-4

3. For each restaurant, predict whether the number of customers inside the restaurant at one time will ever be above 200 on Tuesdays.

```
AggOp(
    entity = "restaurant",
    time_axis = "day",
    data_source="base",
    optype="ANY",
    RowOp=(
        type="GT",
        cols=["num_of_customers_in_restaurant"],
        args=[200]
    ),
    cols=["num_of_customers_in_restaurant"],
    FilterOp="DAY_OF_WEEK(timestamp) == 2"
)
```

4. Predict whether the average number of customers inside the restaurant at one time will always be above 50 on Tuesdays.

```
AggOp(
    entity = "restaurant",
    time_axis = "day",
    data_source="base",
    optype="ALL",
    RowOp=(
        type="GT",
        cols=["num_of_customers_in_restaurant"],
        args=[50]
    ),
    cols=["num_of_customers_in_restaurant"],
    FilterOp=
        ("DAY_OF_WEEK(timestamp) == 2)  &&
        ("num_of_customers_in_restaurant" ==
            MEAN("num_of_customers_in_restaurant"))"
)
```

Table A.6: BigTrane Examples 5-6

5. Predict whether user exists after certain date. The EXISTS operator returns true if the number of rows is greater than 1. Notice that we could have used any of the columns for the RowOp because the AggOp is only counting the number of rows. This is one example of case where there are multiple ways of defining the same problem in the language, and highlights the need to specify equivalencies or define a unique convention. In this particular case, we allow for a "*" column name which implies that the column does not matter.

```
AggOp(
    entity="user",
    time_axis="timestamp",
    data_source="base",
    optype="EXISTS",
    RowOp=(
        type="IDENTITY",
        cols="*",
    ),
    cols="*",
    FilterOp="ALL"
)
```

6. Date at which flu virus cases in each municipality is greater than 10.

```
AggOp(
    entity="Municipality",
    time_axis="Date",
    data_source="base",
    optype="FIRST",
    RowOp=(
        type="IDENTITY",
        cols=["Date"]
    ),
    cols=["Date"],
    FilterOp="Flu_Virus_Cases > 10"
)
```

Table A.7: BigTrane Examples 7-8

7. Difference between a taxi trip's last location in meters and starting location. Notice here that the order of operation matters. Filters define which rows are used *before* the row operation operates over those rows. Here, NORM_DIFF takes the l2 norm of the difference between its input columns, returning one fewer rows than its input. Since the input is only two rows, the last and the first, the TransOp's output is only a single row (and only a single column). Not that the timestamp column is denoted "ts".

```
TransOp(
    entity="trip_id",
    time_axis="timestamp",
    data_source="base",
    optype="NORM_DIFF",
    RowOp=(
        type="IDENTITY",
        cols=["lat", "long"],),
    cols=["lat","long"],
    FilterOp="((ts == MIN(ts)) || (ts == MAX(ts)))")
```

8. Difference between a taxi trip's last location in meters and starting location is greater than 50. Note that this problem is nested. We first calculate the NORM_DIFF, and then we check if that value is greater than 50. Note that we only include entity and time axis for the outermost TransOp, since each operation must be defined for the same entity and time axis.

```
TransOp(
    entity="trip_id",
    time_axis="timestamp",
    data_source=TransOp(
        data_source="base",
        optype="NORM_DIFF",
        RowOp=(
            type="IDENTITY",
            cols=["lat", "long"]),
        as_names=["norm"],
        FilterOp="((ts == MIN(ts)) || (ts == MAX(ts)))")
    optype="GT",
    cols = ["norm"],
    args=[50],
    FilterOp="ALL")
```

Table A.8: Top 10 problems for the Czech bank dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
Party_Client_From_Date	LT	ALL	nop	nop	nop
Party_Client_From_Date	LT	ALL	nop	nop	nop
Transaction_Amount_Czk	LT	ALL	nop	nop	nop
Party_Type	IDENTITY	ALL	nop	nop	nop
Party_Birth_Date	LT	ALL	nop	nop	nop
Credit_Debit_Flag	IDENTITY	ALL	nop	nop	nop
Tax_Flag	IDENTITY	ALL	nop	nop	nop
Party_Client_From_Date	LT	ALL	nop	nop	nop
Credit_Debit_Flag	IDENTITY	ALL	nop	nop	nop
Transaction_Amount_Czk	GT	ALL	nop	nop	nop

Table A.9: Bottom 10 problems for the Czech Bank dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
Transaction_Amount_Czk	NEQ	SUM	LT	nop	nop
Transaction_Amount_Czk	POW	DIFF	LT	ANY	nop
City	NEQ	SUM	LT	nop	nop
Product_Agenda_Name	NEQ	SUM	LT	nop	nop
Transaction_Amount_Czk	POW	DIFF	LT	ALL	nop
Transaction_Amount_Czk	POW	DIFF	NEQ	LAST	nop
Transaction_Amount_Czk	POW	DIFF	POW	LAST	NEQ
Transaction_Amount_Czk	POW	DIFF	POW	LAST	EQ
Transaction_Amount_Czk	POW	DIFF	LT	SUM	LT
Transaction_Amount_Czk	POW	IDENTITY	LT	LAST	nop

Table A.10: Top 10 problems for the Gas Station Transaction dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
Amount	LT	SUM	EQ	nop	nop
Amount	LT	ANY	nop	nop	nop
Price	LT	ANY	nop	nop	nop
Timestamp	LT	ANY	nop	nop	nop
Timestamp	LT	SUM	EQ	nop	nop
Price	LT	LAST	nop	nop	nop
Amount	LT	LAST	nop	nop	nop
Timestamp	LT	LAST	nop	nop	nop
Price	LT	SUM	GT	nop	nop
Timestamp	LT	SUM	GT	nop	nop

Table A.11: Bottom 10 problems for the Gas Station Transaction dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
Amount	POW	DIFF	POW	SUM	LT
ChainID	NEQ	ALL	nop	nop	nop
Amount	POW	DIFF	POW	LAST	NEQ
Amount	POW	DIFF	POW	LAST	EQ
Amount	POW	DIFF	POW	LAST_MINUS_FIRST	LT
Price	POW	DIFF	POW	LAST_MINUS_FIRST	LT
Price	POW	DIFF	POW	FIRST	LT
Amount	POW	DIFF	POW	FIRST	LT
Amount	POW	DIFF	POW	LAST	LT
Price	POW	DIFF	POW	LAST	LT

Table A.12: Top 10 problems for the PKDD99 Loan Application dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
amount	LT	ANY	nop	nop	nop
n0_entrepreneurs	LT	ANY	nop	nop	nop
n0_entrepreneurs	LT	ANY	nop	nop	nop
ratio_urban	LT	ANY	nop	nop	nop
avg_salary	LT	ANY	nop	nop	nop
birth_date	LT	ANY	nop	nop	nop
no_small_municipalities	LT	ANY	nop	nop	nop
amount	LT	SUM	EQ	nop	nop
duration	LT	ANY	nop	nop	nop

Table A.13: Bottom 10 problems for the PKDD99 Loan Application dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
ratio_urban	POW	DIFF	LT	LAST	nop
n0_entrepreneurs	POW	DIFF	LT	LAST	nop
ratio_urban	POW	DIFF	LT	LAST	nop
avg_salary	POW	DIFF	LT	LAST	nop
payments	POW	DIFF	LT	LAST	nop
duration	POW	DIFF	LT	LAST	nop
no_crimes_1996	POW	DIFF	LT	LAST	nop
amount	POW	DIFF	LT	LAST	nop
no_small_municipalities	POW	DIFF	LT	LAST	nop
no_xl_municipalities	POW	DIFF	LT	LAST	nop
n0_entrepreneurs	POW	DIFF	LT	LAST	nop

Table A.14: Top 10 problems for the Safecast dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
number_of_measurements	LT	SUM	EQ	nop	nop
number_of_measurements	LT	ANY	nop	nop	nop
value	LT	ANY	nop	nop	nop
latitude	LT	ANY	nop	nop	nop
timestamp	LT	ANY	nop	nop	nop
timestamp	LT	SUM	EQ	nop	nop
latitude	LT	ALL	nop	nop	nop
number_of_measurements	LT	ALL	nop	nop	nop
timestamp	LT	ALL	nop	nop	nop
value	LT	ALL	nop	nop	nop

Table A.15: Bottom 10 problems for the Safecast dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
latitude	POW	IDENTITY	LT	LAST	nop
value	POW	IDENTITY	LT	LAST	nop
value	POW	DIFF	LT	SUM	LT
number_of_measurements	POW	DIFF	LT	SUM	LT
latitude	POW	DIFF	LT	SUM	LT
value	IDENTITY	DIFF	LT	LAST	nop
latitude	POW	DIFF	POW	LAST	LT
value	POW	DIFF	LT	LAST	nop
value	POW	DIFF	POW	LAST	LT
number_of_measurements	POW	DIFF	POW	LAST	LT
latitude	POW	DIFF	LT	LAST	nop
number_of_measurements	POW	DIFF	LT	LAST	nop

Table A.16: Top 10 problems for the Shout dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
other_trade_num	LT	ANY	nop	nop	nop
other_trade_num	LT	ANY	nop	nop	nop
other_trade_num	LT	ANY	nop	nop	nop
created_at	LT	ANY	nop	nop	nop
retweet_count	LT	ANY	nop	nop	nop
this_trade_num	LT	ANY	nop	nop	nop
timestamp	LT	ANY	nop	nop	nop
created_at	LT	ANY	nop	nop	nop
other_trade_num	LT	SUM	EQ	nop	nop
this_trade_num	IDENTITY	LAST_MINUS_FIRST	EQ	nop	nop

Table A.17: Bottom 10 problems for the Shout dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
language	EQ	SUM	LT	nop	nop
other_trade_num	EQ	SUM	LT	nop	nop
original_poster_id	EQ	SUM	LT	nop	nop
this_trade_num	EQ	SUM	LT	nop	nop
name	EQ	SUM	LT	nop	nop
timestamp	EQ	SUM	LT	nop	nop
retweet_count	EQ	SUM	LT	nop	nop
other_trade_num	EQ	SUM	LT	nop	nop
created_at	EQ	SUM	LT	nop	nop
other_trade_num	EQ	SUM	LT	nop	nop

Table A.18: Top 10 problems for the Walmart dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
Size	POW	LAST_MINUS_FIRST	EQ	nop	nop
Date	LT	SUM	EQ	nop	nop
Date	GT	ANY	nop	nop	nop
CPI	GT	ANY	nop	nop	nop
Weekly_Sales	GT	ANY	nop	nop	nop
MarkDown4	GT	ANY	nop	nop	nop
MarkDown4	GT	ANY	nop	nop	nop
Temperature	GT	ANY	nop	nop	nop
Size	GT	ANY	nop	nop	nop
Unemployment	GT	ANY	nop	nop	nop

Table A.19: Bottom 10 problems for the Walmart dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
Weekly_Sales	IDENTITY	DIFF	LT	LAST	nop
Unemployment	IDENTITY	DIFF	LT	LAST	nop
MarkDown4	IDENTITY	DIFF	LT	LAST	nop
Unemployment	IDENTITY	DIFF	LT	SUM	LT
MarkDown4	IDENTITY	DIFF	LT	SUM	LT
Size	IDENTITY	DIFF	LT	SUM	LT
Temperature	IDENTITY	DIFF	LT	SUM	LT
MarkDown4	IDENTITY	DIFF	LT	SUM	LT
CPI	IDENTITY	DIFF	LT	SUM	LT
Weekly_Sales	IDENTITY	DIFF	LT	SUM	LT
Date	IDENTITY	DIFF	LT	SUM	LT
Temperature	IDENTITY	DIFF	LT	LAST	nop

Table A.20: Top 10 problems for the Social Evolution dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
prob2	LT	ALL	nop	nop	nop
fever	IDENTITY	ALL	nop	nop	nop
aerobic_per_week	LT	ALL	nop	nop	nop
current_smoking	IDENTITY	ALL	nop	nop	nop
current_weight	LT	ALL	nop	nop	nop
time_stamp	LT	ALL	nop	nop	nop
aerobic_per_week	LT	ALL	nop	nop	nop
open.stressed	IDENTITY	ALL	nop	nop	nop
current_height	LT	ALL	nop	nop	nop
survey.date	LT	ALL	nop	nop	nop

Table A.21: Bottom 10 problems for the Social Evolution dataset.

Column Name	Row Op1	Col Op1	Row Op2	Col Op2	Row Op3
current_weight	POW	DIFF	POW	LAST	LT
aerobic_per_week	POW	DIFF	POW	LAST	LT
prob2	POW	DIFF	POW	LAST	LT
current_height	POW	DIFF	POW	LAST	LT
aerobic_per_week	POW	DIFF	POW	LAST	LT
current_height	POW	DIFF	LT	LAST	nop
current_weight	POW	DIFF	LT	LAST	nop
aerobic_per_week	POW	DIFF	LT	LAST	nop
aerobic_per_week	POW	DIFF	LT	LAST	nop
prob2	POW	DIFF	LT	LAST	nop

Bibliography

[1] U.S. General Services Administration. Data.gov, 2016. [Online; accessed 21-May-2016].

[2] Code For Africa. openafrica, 2016. [Online; accessed 21-May-2016].

[3] Charu C. Aggarwal. *Data Mining*, section 1.3, pages 6–14. Springer International Publishing Switzerland, 2015.

[4] Charu C. Aggarwal. *Data Mining*, chapter 2, pages 27–36. Springer International Publishing Switzerland, 2015.

[5] Charu C. Aggarwal. *Data Mining*, section 1.4, pages 14–21. Springer International Publishing Switzerland, 2015.

[6] Charu C. Aggarwal. *Data Mining*, table of contents 1.2, pages xiii–xv. Springer International Publishing Switzerland, 2015.

[7] Nicolo Cesa-Bianchi and Gabor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, New York, NY, USA, 2006.

[8] Tianqi Chen, Weinan Zhang, Qiuxia Lu, Kailong Chen, Zhao Zheng, and Yong Yu. SVDFeature: A toolkit for feature-based collaborative filtering. *Journal of Machine Learning Research*, 13:3619–3622, 2012.

[9] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[10] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2962–2970. Curran Associates, Inc., 2015.

[11] Open Knowledge Foundation. Search for a dataset | the datahub, 2016. [Online; accessed 21-May-2016].

[12] Rina Foygel, Nathan Srebro, and Ruslan R Salakhutdinov. Matrix reconstruction with the local max norm. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 935–943. Curran Associates, Inc., 2012.

[13] Elad Hazan, Satyen Kale, and Shai Shalev-Shwartz. Near-optimal algorithms for online matrix prediction. *CoRR*, abs/1204.0136, 2012.

[14] Kaggle Inc. Kaggle: Your home for data science, 2016. [Online; accessed 21-May-2016].

[15] Kaggle Inc. Kdd cup 2014 - predicting excitement at donors choose | kaggle, 2016. [Online; accessed 21-May-2016].

[16] Kaggle Inc. Walmart recruiting - store sales forecasting | kaggle, 2016. [Online; accessed 21-May-2016].

[17] Safecast Inc. About safecast | safecast, 2016. [Online; accessed 21-May-2016].

[18] James Max Kanter and Kalyan Veeramachaneni. Deep feature synthesis: Towards automating data science endeavors. In *DSAA*, pages 1–10. IEEE, 2015.

[19] Yehuda Koren. The bellkor solution to the netflix grand prize, 2009. [Online; accessed 21-May-2016].

[20] Ambika Krishnamachar and Cesar Hidalgo. Shout!, 2016. [Online; accessed 21-May-2016].

[21] Wes McKinney. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

[22] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *In Proceedings of Workshop at ICLR*, 2013.

[23] Jan Motl and Oliver Schulte. The CTU prague relational learning repository. *CoRR*, abs/1511.03086, 2015.

[24] Nils Nilsson. Introduction to machine learning, 2015. [Online; accessed 22-May-2016].

[25] City of Chicago. City of chicago data portal, 2016. [Online; accessed 21-May-2016].

[26] U.S. Department of Health and Human Services. Datasets | healthdata.gov, 2016. [Online; accessed 21-May-2016].

[27] The City of New York. Nyc open data, 2016. [Online; accessed 21-May-2016].

[28] D.O. Olguin, B.N. Waber, Taemie Kim, A. Mohan, K. Ara, and A. Pentland. Sensible organizations: Technology and methodology for automatically measuring organizational behavior. *IEEE Transactions on Systems, Part B: Cybernetics*, 39(1):43–55, February 2009.

[29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[30] Raghu Ramakrishnan and Johannes Gehrke. Database management systems, 2003.

[31] Nikhil Rao, Hsiang-Fu Yu, Pradeep K Ravikumar, and Inderjit S Dhillon. Collaborative filtering with graph information: Consistency and scalable methods. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2107–2115. Curran Associates, Inc., 2015.

[32] Marta Sabou, Kalina Bontcheva, and Arno Scharl. Crowdsourcing research opportunities: Lessons from natural language processing. In *Proceedings of the 12th International Conference on Knowledge Management and Knowledge Technologies*, i-KNOW '12, pages 17:1–17:8, New York, NY, USA, 2012. ACM.

[33] Seung Kyoon Shin and G. Lawrence Sanders. Denormalization strategies for data retrieval from data warehouses. *Decision Support Systems*, 42(1):267 – 282, 2006.

[34] Pannaga Shivaswamy and Thorsten Joachims. Online structured prediction via coactive learning. In *ICML*. icml.cc / Omnipress, 2012.

[35] Alex Smola and S.V.N Vishwanathan. *Introduction to Machine Learning*, section 1.1, pages 3–12. Cambridge University Press, 2008.

[36] Hao Su, Jia Deng, and Li Fei-Fei. Crowdsourcing annotations for visual object detection. In *AAAI Technical Report, 4th Human Computation Workshop*, 2012.

[37] Richard Szeliski. *Computer Vision: Algorithms and Applications*, section 2.1, pages 31–60. Springer, 2010.

[38] Princeton University. Wordnet, 2013. [Online; accessed 21-May-2016].

[39] Max Welling. A first encounter with machine learning, 2015. [Online; accessed 22-May-2016].

[40] Max Welling. A first encounter with machine learning, 2015. [Online; accessed 22-May-2016].

[41] Wikipedia. Database normalization - Wikipedia, the free encyclopedia, 2016. [Online; accessed 21-May-2016].

[42] Wikipedia. tf-idf - Wikipedia, the free encyclopedia, 2016. [Online; accessed 21-May-2016].

[43] Mohammed J. Zaki and Wagner Meira. *Data Mining and Analysis*, section 1.1, pages 1–2. Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013-2473, USA, 2014.

[44] Mohammed J. Zaki and Wagner Meira. *Data Mining and Analysis*, section 1.2, page 3. Cambridge University Press, 32 Avenue of the Americas, New York, NY 10013-2473, USA, 2014.